Flexible Spectral Algorithms for Simulating Astrophysical and Geophysical Flows

by

Keaton James Burns

MASt., Univ. of Cambridge (2013)
B.A., Univ. of California Berkeley (2012)

Submitted to the Department of Physics
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
at the
Massachusetts Institute of Technology

June 2018

Signature of Author .....
Department of Physics
May 25, 2018

Certified by .....
Nevin N. Weinberg
Associate Professor of Physics
Thesis Supervisor

Certified by .....
Glenn R. Flierl
Professor of Oceanography
Thesis Supervisor

Accepted by .....
Scott A. Hughes
Professor of Physics
Interim Associate Department Head

Flexible Spectral Algorithms for Simulating
Astrophysical and Geophysical Flows

by

Keaton James Burns

Submitted to the Department of Physics on May 25, 2018
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Physics

Abstract

Large-scale numerical simulations are key to studying the complex physical systems that surround us. Simulations provide the ability to perform simplified numerical experiments to build our understanding of large-scale processes which cannot be controlled and examined in the laboratory. This dissertation develops a new open-source computational framework, Dedalus, for solving a diverse range of equations used to model such systems and applies the code to the study of stellar and oceanic fluid flows.

In the first part, the spectral algorithms used in Dedalus are introduced and the design and development of the code are described. In particular, the code's symbolic equation specification, arbitrary-dimensional parallelization, and sparse spectral discretization systems are detailed. This project provides the scientific community with an easy-to-use tool that can efficiently and accurately simulate many processes arising in geophysical and astrophysical fluid dynamics.

In the second part, Dedalus is used to study the turbulent boundary layers that form at the interface between marine-terminating glaciers and the ocean. A simplified model considering the heat transfer from a heated or cooled wall in a stratified fluid is investigated. We find new scaling laws for the turbulent heat transfer from the wall as a function of the imposed thermal forcing, with potential implications for the sensitivity of glacier melting to warming ocean temperatures.

In the third part, Dedalus is used to study the stability of the tidal deformations experienced by binary neutron stars as they inspiral. We develop a numerical workflow for determining the weakly nonlinear stability of a tidally forced plane-parallel atmosphere and verify the results using fully nonlinear simulations. This framework may help determine whether tidal instabilities can be observed in gravitational wave signatures of binary neutron stars, which could provide observational constraints on the equation of state of matter above nuclear densities.

Thesis Supervisor: Nevin N. Weinberg	Thesis Supervisor: Glenn R. Flierl
Title: Associate Professor of Physics	Title: Professor of Oceanography

Acknowledgments

First, I thank my advisors Nevin Weinberg and Glenn Flierl for their generous support and supervision as I have jumped between studying algorithms, oceans, and astrophysics over the past five years. I also thank the other members of my thesis committee, Steven Johnson and Mark Vogelsberger, for their feedback and suggestions for improving my work.

Benjamin Brown, Daniel Lecoanet, Jeffrey Oishi, and Geoffrey Vasil deserve tremendous thanks for the help and mentorship they have continually provided to me. Working with them has been a highlight of my time in graduate school, and I look forward to continuing our collaboration.

Andrew Wells, Ian Hewitt, and Neil Balmforth have advised me on various projects over the past few summers at the Woods Hole Oceanographic Institute, and I am very grateful for the broader view of applied math and geophysics that they have imparted. I would also like to thank Eliot Quataert, Keith Julien, and Jörn Dunkel for their support and advice.

Finally, I would like to thank my parents, my family, and my friends for their help during my time graduate school. In particular, my fiancée Cristina has been an amazing source of inspiration and companionship as she has studied for her medical degree. Friends from home, college, and MIT have made the past five years in Boston extraordinarily fun and memorable.

During my PhD I have been supported by the MIT Kavli Graduate Fellowship, the National Science Foundation Graduate Research Fellowship under Grant No. (1122374), the WHOI Geophysical Fluid Dynamics Fellowship, the MIT PAOC group, and NASA ATP grant NNX14AB40G.

Contents

Chapter 0

Thesis outline

This dissertation focuses on the development of a numerical framework called Dedalus for simulating large-scale fluid flows found in nature. During my PhD, I developed a substantial portion of the Dedalus codebase and applied the code to study problems problems involving astrophysical and geophysical flows. The dissertation is divided into three parts describing the codebase, applications to glacial meltwater plumes, and applications to tidal instabilities in neutron stars.

0.1 The Dedalus Project

The first part describes the design and implementation of the Dedalus codebase. Dedalus uses a set of algorithms known as global spectral methods which expand variables over sets of basis functions to discretize and solve partial differential equations (PDEs). These methods typically provide exponentially increasing accuracy as more modes are added, but are unable to accommodate complex geometries. However, they are extremely well-suited to studying low Mach-number flows in simple geometries, such as those commonly found in astrophysics and geophysics. Spectral methods have been widely used to simulate fluids in the past, but typically with techniques that require solving dense matrices and in codes that implement one specific set of equations. In the applied math community, flexible algorithms producing sparse matrices for general equations have been developed, but principally applied to small-scale simulations. §1 provides an introduction to spectral algorithms and sparse techniques.

Dedalus bridges this gap by combining sparse spectral algorithms into a framework for the efficient solution of large-scale PDEs, as described in §2. The fundamental algorithm behind Dedalus combines a range of previous work on sparse Chebyshev methods into a

system for producing sparse spectral discretizations of nearly arbitrary systems of PDEs. This algorithm is an important contribution to the spectral methods community as it provides a robust and simple way of handling a wide array of equations, and is described in detail in §2.7. To make this system easily accessible, I have implemented a parsing system which allows users to symbolically specify systems of differential equations, as described in §2.6. The parser manipulates these equations into a generalized structure which is then processed by our spectral algorithm to produce a sparse system. Additionally, I have generalized the domain distribution algorithms traditionally used in large-scale spectral codes to handle problems in arbitrary dimensions and with arbitrary process meshes, as described in §2.3.

Together, these features make Dedalus an easy-to-use but highly efficient code for simulating large-scale PDEs with smooth solutions in simple geometries. Dedalus is open-source and has been applied to a wide range of problems in astrophysical, geophysical, and biological fluid dynamics.

0.2 Glacial Meltwater Plumes

The second part describes my use of Dedalus to study the turbulent boundary layers that form at the interface between marine-terminating glaciers and the ocean. As global sea and air temperatures rise, glaciers in Greenland and Antarctica are expected to begin melting at a faster rate, resulting in a net mass-loss of ice in these regions and raising global sea levels. A large uncertainty in the projected melt rates in Greenland, which have grown exponentially in recent decades, is how warming waters will influence melting in the fjords where Greenland's glaciers meet the sea. This problem is complex because meltwater is buoyant relative to the ambient fjord water and forms a highly turbulent plume as it rises along the submerged face of each glacier. The resulting turbulent heat transport across this plume determines the continued melt rate of the glacier. §3 provides an introduction to meltwater plumes and the equations governing ice-water interfaces.

I have used Dedalus to perform a series of simulations of a simplified analog problem of a heated interface in a stratified incompressible fluid, as described in §4. This simplified problem retains the essential characteristics of a laterally-driven buoyant plume, but reduces the complexity associated with solving the full equations for a melting interface. Our simulations allow us to fit a power-law trend between the thermal forcing and the resulting turbulent heat transfer at the wall (§4.4). Therefore, although we are unable to simulate a full glacier, we can extrapolate these results to the geophysical regime and predict the

melting rate from this process, as well as its sensitivity to increasing ocean temperatures. This is a highly simplified model that neglects many important processes in actual glacial fjords, but helps build our understanding of the fluid dynamics near ice-ocean interfaces.

0.3 Nonlinear Tidal Instabilities

The third part describes my use of Dedalus to study the stability of tidal deformations experienced by binary neutron stars as they inspiral. When the separation in binary systems decreases, the tidal deformation of the bodies in the binary increases. The amount of dissipation caused by the tide is poorly understood but is important in determining the evolution of binary stars and planetary orbits. Furthermore, it is theorized that instabilities in the tidal deformation of binary neutron stars may alter the gravitational waveforms emitted by such systems, potentially allowing detectors such as LIGO to observationally constrain the unknown interior structure of these objects. An introduction to astrophysical tides and tidal instabilities is provided in §5.

I have used Dedalus to develop a comprehensive workflow for studying the nonlinear stability of a plane-parallel atmosphere experiencing tidal deformations, which is detailed in §6. Using Dedalus, we solve for the background structure of the atmosphere, determine the eigenmodes of the atmosphere, and evaluate the predicted threshold amplitude for tidal instabilities using weakly nonlinear theory. We have additionally performed fully nonlinear simulations which agree with the predicted threshold. Although we have not yet examined realistic neutron star models, this work demonstrates the feasibility of using fully nonlinear calculations to test tidal stability predictions from perturbative theories. In the future, these simulations may help estimate the nonlinear tidal dissipation expected in neutron star binaries and the feasibility of observing this process with gravitational wave detectors.

Part I

The Dedalus Project

Chapter 1

Introduction to Spectral Methods

1.1 Spectral representations of functions

1.1.1 Spectral convergence and truncation errors

A spectral method is a technique for discretizing functions by expanding them over a set of basis functions. These methods find broad application in numerical analysis as they lead to highly accurate and efficient algorithms for manipulating functions and solving differential equations. Boyd (2001) is a classic reference on these methods, and covers the material in this section in great detail.

Consider a complete set of basis functions $\{\phi_n(x)\}$ that are orthogonal under some inner product $\langle \phi_n | \phi_m \rangle \propto \delta_{n,m}$. The spectral representation of a function $f(x)$ with respect to this basis is given by the set of coefficients $\{f_n^\phi\}$ appearing in the expansion of $f(x)$ as

$$f(x) = \sum_{n=0}^{\infty} f_n^\phi \phi_n(x) \tag{1.1}$$

$$f_n^\phi = \frac{\langle \phi_n | f \rangle}{\langle \phi_n | \phi_n \rangle} \tag{1.2}$$

In general, such a representation will require an infinite number of nonzero coefficients to be exact. Numerical spectral methods seek to find an approximate representation of a function, often the solution to a differential equation, using a truncated expansion with N modes

$$\tilde{f}(x) = \sum_{n=0}^{N-1} \tilde{f}_n^\phi \phi_n(x) \tag{1.3}$$

The error of this approximation consists of the *discretization error* from the difference

between the approximate coefficients $\tilde{f}_n^\phi$ and exact coefficients f_n^ϕ for $n = 0, ..., N-1$, and the *truncation error* due to neglecting the terms with $n \geq N$.

In practice, the discretization error is often assumed to be of the same order as the truncation error, which can be estimated using the convergence properties of the spectral representation of $f(x)$. For basis functions satisfying $|\phi_n(x)| \leq 1$, the truncation error satisfies

$$E_T(N) = \left| \sum_{n=N}^{\infty} f_n^\phi \phi_n(x) \right| \tag{1.4}$$

$$\leq \sum_{n=N}^{\infty} |f_n^\phi| \tag{1.5}$$

This error can be estimated in several cases:

- If the coefficients behave like $f_n^\phi \sim O(n^{-k})$ for large n, then the series is said to converge algebraically with an algebraic index of convergence equal to k. In this case, we have $E_T(N) \sim O((N-1)^{k-1}) \sim O((N-1)|f_{n-1}^\phi|)$, or roughly N times the amplitude of the last retained coefficient.

- If the coefficients behave like $f_n^\phi \sim O(\exp(-cn^r))$ for large n, then the series is said to converge exponentially with an exponential index of convergence equal to r. If $\mu = \limsup(-\log|f_n^\phi|/n)$ is zero, the series is said to converge sub-geometrically. If μ is infinite, the series is said to converge super-geometrically. If μ is constant, the series the said to converge geometrically with an asymptotic rate of geometric convergence equal to μ, and $E_T(N) \sim O(\exp(-\mu(N+1))) \sim O(|f_{n-1}^\phi|)$.

In both cases, we see that the amplitude of the last remaining coefficient can be used to estimate the error in our approximation if we know how the function of interest converges with respect to the given basis set.

1.1.2 Fourier series

The Fourier series is a powerful basis that is widely used in analytical and numerical spectral methods due to the rapid convergence of the Fourier representation of smooth functions on periodic intervals. The standard Fourier basis functions are simply complex exponential

functions e^{inx} where n is an integer, which are orthonormal under the inner product

$$\langle e^{imx}|e^{inx}\rangle = \frac{1}{2\pi}\int_0^{2\pi} e^{-imx}e^{inx}\mathrm{d}x = \delta_{m,n} \tag{1.6}$$

The standard Fourier expansion of a function that is periodic on the interval $[0, 2\pi]$ takes the form

$$f(x) = \sum_{n=-\infty}^{\infty} f_n e^{inx} \tag{1.7}$$

and when truncated to a finite number of modes the truncation is done symmetrically around $n = 0$.

It can be shown that if the first k derivatives of $f(x)$ are continuous, then the Fourier expansion of $f(x)$ converges algebraically with an algebraic index of convergence of at least $k+2$. If $f(x)$ is smooth (infinitely differentiable) on the real line, then the Fourier expansion converges exponentially. If $f(x)$ has singularities away from the real line in the complex plane, the convergence will be geometric with an asymptotic rate equal to the shortest distance from the real line to one of the singularities. If $f(x)$ is an entire function, then the Fourier expansion of $f(x)$ converges super-geometrically. These properties make Fourier series ideally suited for representing functions that are smooth or highly differentiable on a periodic interval. Similar results hold for the representation of functions with even or odd parity around the endpoints of an interval using cosine or sine series, respectively.

For a finite Fourier expansion, the Fast Fourier Transform (FFT) allows for the conversion between the Fourier coefficients of a function and the values of the function on the uniformly-spaced grid $x_i = 2\pi i/N,\ (i = 0, .., N-1)$ in $O(N \log N)$ time. This capability enables computations requiring both sets of values to be performed efficiently for large N.

1.1.3 Chebyshev series

Chebyshev polynomials are a family of classical orthogonal polynomials on the interval $[-1, 1]$. They are related to cosine functions under a simple change of variables as

$$T_n(x) = \cos(n\cos^{-1}(x)) \tag{1.8}$$

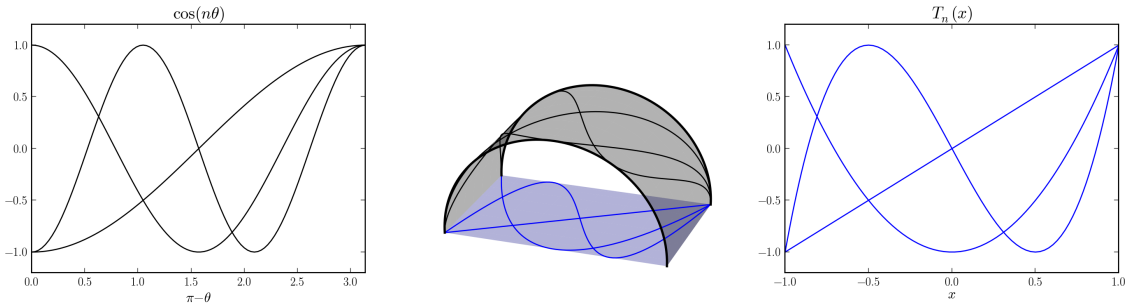

Figure 1.1: The Chebyshev polynomials can be viewed as projections onto the plane of cosine modes drawn on a cylinder. Figure adapted from Burns (2013).

and are orthogonal under the weighted inner product

$$\langle T_m | T_n \rangle = \int_{-1}^{1} \frac{T_m(x) T_n(x)}{\sqrt{1-x^2}} \mathrm{d}x = \frac{\pi}{2 - \delta_{m,0}} \delta_{m,n} \tag{1.9}$$

The coordinate transform used to define the polynomials provides the geometric interpretation of the $T_n(x)$ being the projection onto the plane of the function $\cos(nx)$ drawn on a cylinder bisected by the plane, as shown in Fig. 1.1.

The convergence properties of Chebyshev series follow closely from those of Fourier series due to their connection via a simple coordinate transform. In particular, if $f(x)$ is smooth on the interval $[-1, 1]$, then the Chebyshev expansion of $f(x)$ will converge exponentially. If $f(x)$ has singularities away from this real interval in the complex plane, the convergence will be geometric with an asymptotic rate equal to the smallest quasi-radius of the singularities in elliptic coordinates. If $f(x)$ is entire, then the Chebyshev expansion converges super-geometrically. These properties make Chebyshev series ideally suited for representing functions that are smooth or highly differentiable but non-periodic on finite intervals.

The relation to the cosine series can also be used to convert between the truncated Chebyshev coefficients of a function and its values on specific grids using fast discrete cosine transforms (DCTs). In particular, DCTs can be used to transform forward and backwards from the nodes of Gaussian quadrature for Chebyshev polynomials to the Chebyshev expansion coefficients. These nodes, also known as the "roots" grid for Chebyshev polyno-

mials, consist of the points

$$x_i = -\cos\left(\frac{\pi(i+1/2)}{N}\right) \qquad i = 0, ..., N-1 \tag{1.10}$$

DCTs can also be used to transform the expansion coefficients to the values on the "extrema" grid for Chebyshev polynomials, which contains the endpoints of the interval, given by

$$x_i = -\cos\left(\frac{\pi i}{N-1}\right) \qquad i = 0, ..., N-1 \tag{1.11}$$

The availability of a fast transform often makes Chebyshev polynomials series preferable to other polynomial series for representing functions on finite intervals.

1.2 Solving differential equations with spectral methods

Spectral methods are used to solve differential equations by expanding the unknown solutions in a truncated spectral series and solving the resulting algebraic equations for the expansion coefficients. There are a number of approaches for constructing and solving these equations, each with different advantages and disadvantages. In discussing the various options, we will consider first-order linear differential equations of the form

$$L(u)(x) = f(x) \tag{1.12}$$

with a general Dirichlet boundary condition

$$au(-1) + bu(1) = c \tag{1.13}$$

where L is a first-order differential operator, $f(x)$ and $\{a, b, c\}$ are specified, and $u(x)$ is the desired solution. Higher-order equations and/or equations with different boundary conditions (e.g. Neumann conditions) can generally be handled with similar methods, or expressed as systems of first-order equations of the desired form. The truncated version of the equation with N modes, supposing we can compute the exact truncation $\tilde{f}(x)$ of $f(x)$, reads

$$L(\tilde{u})(x) = \tilde{f}(x) \tag{1.14}$$

$$a\tilde{u}(-1) + b\tilde{u}(1) = c \tag{1.15}$$

1.2.1 Collocation method

Perhaps the most common formulation of Chebyshev spectral methods is the collocation approach, where the differential equation is enforced on the Chebyshev extrema points. For any set of N separate points $\{x_i\}$, the Cardinal polynomials $\{C_i\}$ for those points are the N unique polynomials of degree $(N-1)$ satisfying $C_i(x_j) = \delta_{i,j}$. Any polynomial of degree up to $(N-1)$ can be uniquely expanded in the cardinal polynomials using its values at the grid points as

$$\tilde{u}(x) = \sum_{j=0}^{N-1} \tilde{u}(x_j) C_j(x) \tag{1.16}$$

Standard collocation methods substitute this expansion and enforce the differential equations at each point in the extrema grid, except for at one of the boundary points where the boundary condition is instead enforced (without loss of generality, we will assume $b \neq 0$ and apply the boundary condition at $x_{N-1} = 1$). This scheme can be written as a matrix problem for the values of $\tilde{u}$ on the extrema grid as

$$\sum_{j=0}^{N-1} L(C_j)(x_i)\tilde{u}(x_j) = \tilde{f}(x_i) \qquad (i = 0, ..., N-2) \tag{1.17}$$

$$a\tilde{u}(x_0) + b\tilde{u}(x_{N-1}) = c \tag{1.18}$$

The collocation method has been successfully applied to an extremely broad range of applications. Its primary advantage is that the boundary conditions are easily enforced by replacing the endpoint rows of the discretized equation matrix, and the solution can be found directly in grid space. Its primary disadvantage is that the method generally results in fully dense matrices, and more complicated boundary conditions require more care to implement (Driscoll et al., 2015).

1.2.2 Galerkin method

If the equation can be rewritten with a homogeneous boundary condition ($c = 0$) and a basis set can be constructed which automatically satisfies the boundary conditions ($a\phi_i(-1) + b\phi_i(1) = 0$), then the full solution can be found by expanding the differential equation in this basis and enforcing the resulting constraints mode-by-mode. This method is referred

to as the Galerkin method, and can be written explicitly as

$$\sum_{j=0}^{N-1} \langle \phi_i | L\phi_j \rangle \tilde{u}_j^{\phi} = \langle \phi_i | \tilde{f} \rangle \qquad (i = 0, ..., N-1) \tag{1.19}$$

This method is commonly used with Fourier series, which automatically satisfy periodic boundary conditions. Furthermore, since the derivative of a complex exponential is proportional to itself, the derivative matrices produced by this method for Fourier series are banded:

$$\langle e^{inx} | \partial_x e^{imx} \rangle = im\delta_{n,m} \tag{1.20}$$

The Galerkin matrices for Fourier problems with constant coefficients are therefore typically diagonal, allowing them to be solved trivially.

Galerkin bases and methods can be constructed from Chebyshev polynomials for simple boundary conditions, and provide a particularly powerful analytical tool for such problems. However, such bases can be difficult to construct for complex boundary conditions, and require functions to be converted between bases before fast Chebyshev transforms can be used.

1.2.3 Tau method

The tau method is a modified form of the Galerkin method which enforces the spectral expansion of the equations mode-by-mode for $N-1$ modes, followed by the boundary condition. The classical tau method expands the equations in the same basis that is used for the solution, giving

$$\sum_{j=0}^{N-1} \langle \phi_i | L\phi_j \rangle \tilde{u}_j^{\phi} = \langle \phi_i | \tilde{f} \rangle \qquad (i = 0, ..., N-2) \tag{1.21}$$

$$\sum_{j=0}^{N-1} (a\phi_j(-1) + b\phi_j(1))\tilde{u}_j^{\phi} = c \tag{1.22}$$

The method is referred to as the tau method because the above equations are equivalent to solving the perturbed equation

$$L\tilde{u} + \tau\phi_{N-1}(x) = \tilde{f} \tag{1.23}$$

$$a\tilde{u}(-1) + b\tilde{u}(1) = c \tag{1.24}$$

where τ is unknown, to full order over all N modes. The removal of a row from the Galerkin matrix for the differential equation is equivalent to adding a degree of freedom to the equation, a "tau term", which appears as a coefficient in front of a polynomial corresponding to the removed rows. The tau method provides a conceptually straightforward way of enforcing general boundary conditions without requiring a change to a specialized basis.

1.3 Generating sparse spectral methods

1.3.1 Generalizing the tau method

The fundamental principle of the tau method is to close problems over finite truncations in polynomials by adding a tau term to the differential equation. In general, the tau polynomial does not need to be one of the basis functions used to represent the solution. Instead, we can write a general tau-modified system as

$$L\tilde{u} + \tau P(x) = \tilde{f} \tag{1.25}$$

$$a\tilde{u}(x)(-1) + b\tilde{u}(1) = c \tag{1.26}$$

where P is a polynomial of degree $N-1$. At this point, the problem is generally closed and a unique solution $\{\tilde{u}, \tau\}$ exists. The discretized solution can then be found by expanding it in terms of any polynomial basis of trial functions $\{\phi_i\}$, projecting the equations against any polynomial basis of test functions $\{\psi_i\}$, and solving the resulting discretized system

$$\sum_{j=0}^{N-1} \langle \psi_i | L\phi_j \rangle \tilde{u}_j^{\phi} + \tau \langle \psi_i | P \rangle = \langle \psi_i | \tilde{f} \rangle \qquad (i = 0, ..., N-1) \tag{1.27}$$

$$\sum_{j=0}^{N-1} (a\phi_j(-1) + b\phi_j(1))\tilde{u}_j^{\phi} = c \tag{1.28}$$

This formulation provides a unified framework for examining different solution strategies. For instance, the classical Chebyshev-tau method is recovered by picking $\psi_i = \phi_i = T_i$ and $P = T_{N-1}$.

1.3.2 Creating a sparse tau method

The drawback of the formulations discussed so far is that the discretized matrices for the Chebyshev derivative operator are dense. In particular, the derivatives of Chebyshev polynomials satisfy the recurrence relation

$$\frac{\partial_x T_n}{n} = 2T_{n-1} + \frac{\partial_x T_{n-2}}{n-2} \tag{1.29}$$

resulting in a dense upper triangular matrix when the derivatives of Chebyshev polynomials are projected back against themselves:

$$\langle T_i | \partial_x T_j \rangle = \frac{2j((j-i) \bmod 2)}{1+\delta_{i,0}} [i < j] \tag{1.30}$$

The structure of this matrix is shown on the left in Fig. 1.2. However, the derivatives of Chebyshev polynomials can be written sparsely in terms of the Chebyshev polynomials of the second kind, $U_n(x)$, as

$$\partial_x T_n(x) = nU_{n-1}(x) \tag{1.31}$$

The Chebyshev-U polynomials are defined trigonometrically as

$$U_n(x) = \frac{\sin((n+1)\cos^{-1}(x))}{\sin(\cos^{-1}(x))} \tag{1.32}$$

and are an orthogonal family of polynomials under the weight $w(x) = \sqrt{1-x^2}$.

If the Chebyshev-U polynomials are used as test functions, and the projection is done using the Chebyshev-U inner product, then the derivative matrix becomes banded, as shown on the right in Fig. 1.2:

$$\langle U_i | \partial_x T_j \rangle = j\delta_{i,j-1} \tag{1.33}$$

Additionally, non-differential terms can be sparsely converted to Chebyshev-U polynomials via the relation $2T_n(x) = U_n(x) - U_{n-2}(x)$, resulting in the banded conversion matrix

$$\langle U_i | T_j \rangle = \frac{1}{2}(\delta_{i,j} + \delta_{i,j-2}) \tag{1.34}$$

Together, these matrices can be used to render a first-order differential equation with constant coefficients sparse. Higher-order equations can be handled by applying equivalent relations for higher derivatives of the Chebyshev polynomials (known as the ultraspherical method

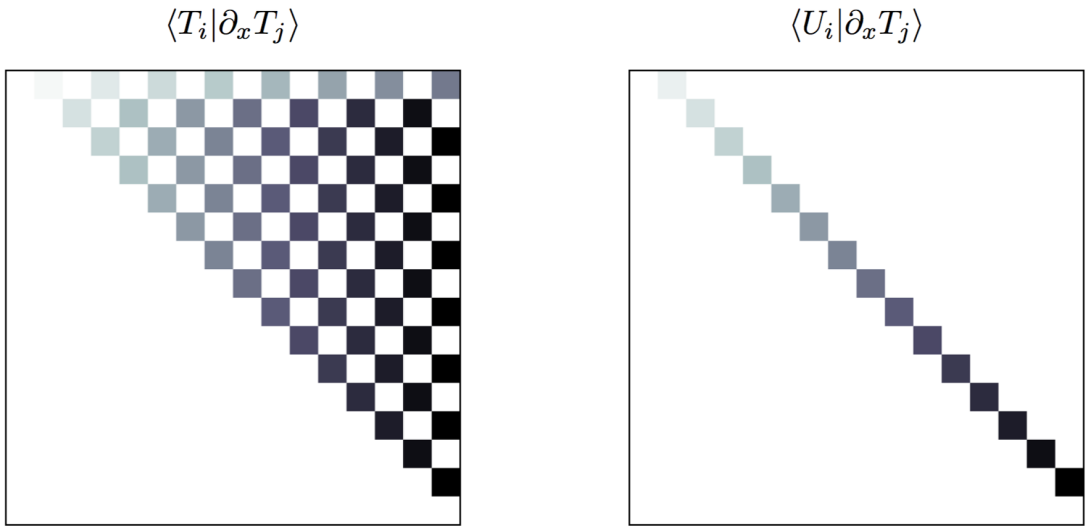

Figure 1.2: Derivative matrices using Chebyshev-T and Chebyshev-U polynomials as test functions. Using different families of test and trial functions allows general differential operators to be represented with sparse matrices.

(Olver et al., 2013)) or simply by reducing the equation to a first-order system.

1.3.3 Building a fully banded tau method

If the tau polynomial is chosen to be T_{N_1} or U_{N-1}, the tau column in the sparse tau method will only contain a few non-zero entries near $i \approx N$, and the equation matrix will be banded. If $P = U_{N-1}$, the tau column and the last row of the equation matrix can be dropped, and the coefficients of $\tilde{u}$ can be solved without finding τ. For simple boundary conditions, a right-preconditioner can then be applied to render the boundary conditions sparse and the system fully banded. For Dirichlet boundary conditions, one such preconditioner is given by changing the trial basis from the Chebyshev-T polynomials to the Chebyshev-D or Dirichlet polynomials, defined by

$$D_n(x) = \begin{cases} T_0(x) & n = 0 \\ T_1(x) & n = 1 \\ T_n(x) - T_{n-2}(x) & n \geq 2 \end{cases} \tag{1.35}$$

The D-to-T conversion matrix is then banded, with

$$\langle T_i | D_j \rangle = \delta_{i,j} - \delta_{i,j-2} \tag{1.36}$$

The Dirichlet polynomials are chosen in this manner to satisfy

$$D_n(\pm 1) = \begin{cases} 1 & n = 0 \\ \pm 1 & n = 1 \\ 0 & n \geq 2 \end{cases} \tag{1.37}$$

so that Dirichlet boundary conditions only involve the first two coefficients under this discretization:

$$a(u_0^D - u_1^D) + b(u_0^D + u_1^D) = c \tag{1.38}$$

Together, we have that for constant-coefficent first-order equations with Dirichlet boundary conditions, choosing $\phi_n = D_n$, $\psi_n = U_n$, and $P = U_{N-1}$ renders the resulting spectral discretization fully banded. This allows the system to be efficiently solved using sparse/banded matrix solution algorithms. Higher-order equations and different boundary conditions can generally be reduced to systems satisfying the necessary form. Equations with non-constant but slowly varying coefficients, as are often encountered in computational physics, can be accommodated by using band-limited expansions of the non-constant coefficients to maintain the overall bandedness and sparsity of the matrices (Olver et al., 2013). This formulation forms the basis of the spectral method implemented by Dedalus, as it allows broad classes of equations to be converted to a computationally efficient and spectrally accurate form.

Chapter 2

Design and Implementation of Dedalus

2.1 Introduction

2.1.1 Motivation for creating Dedalus

The Dedalus Project is a flexible framework for solving partial differential equations that has been developed by myself, Benjamin Brown, Daniel Lecoanet, Jeffrey Oishi, and Geoffrey Vasil. The goal of our collaboration is to build a code that accurately simulates a wide range of equations, particularly those describing fluid flows found in geophysical and astrophysical applications. A broad assortment of nonlinear PDEs are encountered when studying flows in environments such as planetary atmospheres, oceans, and stellar interiors. These include various hydrodynamical equation sets involving a diverse range of physical processes, including magnetic effects, chemical reactions, phase changes, radiative transport, active biological processes, and many more. Additionally, the governing equations are often filtered to produce reduced models which remove some dynamical processes, such as fast waves or boundary layers, while retaining the essential features and dynamics of the flow at the scales of interest.

Numerical studies of astrophysical and geophysical fluid processes are often performed using local or global domains with simple geometries, including spheres, cylinders, and closed or periodic boxes. Additionally, low-dimensional models are frequently utilized to study basic processes over a wider range of parameters than may be feasible with large 3D simulations. Global spectral methods offer a robust and highly efficient approach for solving smooth PDEs in these types of geometries. These methods represent unknown variables by expanding them over a set of global basis functions, rather than discretizing them at points on a numerical grid. For smooth solutions, such as those typically found in low-Mach-number flows, these representations typically provide exponentially increasing

accuracy as more basis functions are added, resulting in rapidly converging and extremely accurate numerical solutions.

Indeed, spectral methods have been widely used to simulate astrophysical and geophysical fluid flows in the past. However, even though these methods are widely applicable, previous spectral solvers have frequently been written with only a narrow range of physical models and geometric domains in mind. Additionally, different formulations of the spectral discretization of a PDE can lead to large differences in the efficiency with which the discretized system can be solved. In particular, recent literature has focused on developing sparse representations of differential operators acting on common spectral bases, which are substantially better conditioned and faster than traditional dense collocation techniques, but these techniques have primarily been applied to 1D problems and small-scale simulations.

In developing Dedalus, we have aimed to build a framework utilizing modern sparse spectral techniques that both flexibly handles different equations and domains and is capable of performing large-scale, highly parallelized simulations. While our 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.

2.1.2 A brief history of Dedalus development

Noting the absence of an easy-to-use, high-order code for simulating astrophysical flows, Jeffrey Oishi began developing the first version of Dedalus in 2010. I joined the project in the summer of 2011 as a student in the DOE SULI program. This first version of the code implemented Boussinesq magnetohydrodynamics in a 3D shearing box geometry using Fourier spectral methods in Python. The project demonstrated the feasibility of using a high-level language to write a simple yet high-performance, parallelized spectral solver for fluid simulations.

The following year, Benjamin Brown, Daniel Lecoanet, and Geoffrey Vasil joined the development team. Together we developed the concept for a new version of Dedalus implementing a matrix-based system for handling general equations using Fourier and Chebyshev spectral methods. As I began my PhD, we began implementing this concept in the second version of the Dedalus codebase. Over the next year, we developed and implemented the equation parsing and automatic parallelization schemes that form the foundation of the current framework. In the following years, we added a range of solvers and features to the codebase, and implemented a variety of performance enhancements.

This dissertation focuses on this version of the code.

In the past several years, we have focused on the development of novel spectral bases for geometries with coordinate singularities, particularly disks (G. M. Vasil et al., 2016) and spheres (Daniel Lecoanet et al., 2018; G. Vasil et al., 2018). We are currently implementing these bases and a number of other enhancements in a new version of the Dedalus codebase which will not be discussed in detail here.

2.1.3 Codebase and dependencies

The Dedalus codebase is primarily written in Python 3. We chose 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 may suffer from poor performance, it is quite easy to wrap optimized C libraries into Python. This allows the language to be used to combine optimized libraries for a variety of performance-critical routines together in a single codebase with a high-level interface. The primary dependencies of Dedalus include:

- The Numpy, Scipy, and Cython packages for Python (Behnel et al., 2011; Jones et al., 2001).
- The FFTW C-library for performing fast Fourier transforms (Frigo et al., 2005).
- An implementation of the MPI communication interface, and the Python wrapper mpi4py (Dalcin et al., 2008).
- The HDF5 C-library for writing and reading HDF5 files, and the Python wrapper h5py (Collette, 2013; The HDF Group, 1997).

The wide range of standard-library Python packages are used to build e.g. logging and configuration 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.

2.1.4 Documentation and community

Dedalus has been fully open-source since its creation, and is currently publicly hosted on Bitbucket[1]. Documentation is maintained online[2] and 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.

The core collaborators additionally maintain mailing lists for the growing Dedalus user and developer communities. The Dedalus user list currently has over 100 members, while the development list has 20 members. Dedalus has been utilized in publications[3] in a wide range of fields, including applied math, astrophysics, atmospheric science, biology, experimental fluid dynamics, planetary science, and plasma physics.

2.1.5 Comparison to other codes

A wide range of open-source codes have been developed to simulate astrophysical and geophysical flows. Several publicly available codes for astrophysical flows are Athena[4], Flash[5], Enzo[6], Pencil[7], and Gizmo[8]. Several publicly available codes for geophysical flows are MITGCM[9], Fluidity[10], and MOM[11]. More general tools for simulating incompressible and low Mach-number flows in complex geometries include the spectral element codes NEK5000[12] and Nektar++[13]. These codes are generally tuned to achieve high levels of performance and parallelism, enabling large-scale simulations of turbulent flows. However, they typically implement specific equations, such as fully compressible Navier-Stokes with a specific equation of state, and can be very difficult to modify if you are interested in a reduced model or physics that has not already been implemented.

A number of flexible PDE solvers are also currently available which can accommodate a

[1] https://bitbucket.org/dedalus-project/dedalus/src/default/
[2] http://dedalus-project.readthedocs.io/en/latest/
[3] http://dedalus-project.org/citations/
[4] https://github.com/PrincetonUniversity/Athena-Cversion
[5] http://flash.uchicago.edu/site/flashcode/
[6] http://enzo-project.org
[7] http://pencil-code.nordita.org
[8] http://www.tapir.caltech.edu/~phopkins/Site/GIZMO.html
[9] http://mitgcm.org
[10] http://fluidityproject.github.io
[11] https://www.gfdl.noaa.gov/ocean-model/
[12] https://nek5000.mcs.anl.gov
[13] https://www.nektar.info

wider range of physical models. One of the most popular is FEniCS[14], a finite-element code that allows users to symbolically enter their equations in variational form. FEniCS can solve many PDEs in a wide range of geometries, however for simple geometries its finite element discretization is less efficient than a global spectral method. One of the most versatile codes utilizing spectral methods is Chebfun[15], a MATLAB package for manipulating functions and solving differential equations using Chebyshev collocation methods. ApproxFun[16] is a similar Julia package for manipulating functions using sparse Chebyshev methods. Both of these packages provide functionality well beyond solving differential equations, but are generally focused on solving low-dimensional problems on shared-memory systems, rather than solving large-scale PDEs in parallel.

Dedalus aims to bridge this gap by providing a high-performance, flexible solver based on sparse spectral methods. Compared to traditional AFD/GFD codes, it provides a simple way of simulating a broader range of equations and achieving highly accurate solutions. The primary drawback of the code is its restriction to simple geometries that can be represented by the direct product of spectral bases. Finite element and spectral element solvers are well-suited to complex geometries, but in the simple domains that are often considered in astrophysics and geophysics, the global elements utilized by Dedalus are more accurate and efficient. In comparison to other flexible spectral codes, Dedalus focuses on solving large-scale PDEs in parallel environments.

Many of the building blocks of the spectral algorithms in Dedalus have been utilized by a number of authors for decades. The contribution of our work to the spectral community is assembling a range of techniques to produce a framework for efficiently solving a broad range of PDEs in parallel. In particular, §2.7 describes how we formulate sparse spectral matrices from symbolically entered systems of equations. By requiring problems to be first-order in temporal and Chebyshev derivatives, we are able to simply and consistently produce banded matrices for wide ranges of equation sets. This combines the well-established techniques of Chebyshev T-to-U conversion and Dirichlet preconditioning (Boyd, 2001) with the more recent technique of band-limited coefficient expansions (Olver et al., 2013) to simply accommodate equations with non-constant coefficients, high-order derivatives, and a variety of boundary conditions. Additionally, §2.3 describes our domain distribution system, which generalizes the slab and pencil distributions traditionally used in parallel spectral codes to

[14] https://fenicsproject.org
[15] http://www.chebfun.org
[16] https://github.com/JuliaApproximation/ApproxFun.jl

arbitrary dimensions and process meshes. Along with our simple parsing interface, these algorithms for producing banded systems and performing large-scale distributed solves for arbitrary PDEs make Dedalus a novel addition to the spectral methods community.

2.2 Spectral bases

Dedalus represents multidimensional fields using the direct product of one-dimensional spectral bases, and implementations of each type of basis form the lowest level of the program's class hierarchy. The primary responsibilities of each basis class are to define the collocation points of that basis and to provide an interface for performing spectral transforms between the spectral coefficients of a function and the values of the function on the 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 affinely transformed to a specified interval on the real line $[x_L, x_R] \subset \mathbb{R}$, and is instantiated with these arguments. Each basis class is defined with respect to a *native interval* on the real line $[X_L, X_R] \subset \mathbb{R}$, 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 and *problem coordinates* x are done via an affine transformation:

$$\frac{X - X_L}{X_R - X_L} = \frac{x - x_L}{x_R - x_L} \tag{2.1}$$

which is simply applied to a native grid to produce a *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. Along the transform dimension, the size of the coefficient data must be N_c and the size of the grid data must match the specified transform scale s. When $s < 1$, the coefficients are truncated after the first N_g modes before the transform is applied. 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 the transform is applied. Such transforms are useful for calculating nonlinear terms, such as products of multiple fields, in grid space without aliasing errors. They are also useful for performing spectral interpolation, i.e. to view low resolution data on a fine grid.

Using objects to represent bases is particularly useful because it allows the transform

methods to easily cache plans or matrices that may be costly to precompute. The basis classes also present a unified interface for implementing identical transforms using multiple libraries with potentially different performance and build requirements on different systems and problems. We will now define the basis functions, grids, and transform methods for the currently implemented spectral bases.

2.2.1 Fourier basis

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

$$\phi_k^F(x) = \exp(ikx) \tag{2.2}$$

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

$$x_i^F = \frac{2\pi i}{N_g} \qquad i = 0, ..., N_g - 1. \tag{2.3}$$

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

$$f(x) = \sum_{-k_m}^{k_m} f_k \phi_k^F(x) \tag{2.4}$$

where $k_m = \lfloor (N_c - 1)/2 \rfloor$ is the maximum resolved wavenumber, excluding the Nyquist mode $k_N = N_c/2$ when N_c is even. When f is a real function, only the (complex) coefficients corresponding to $k >= 0$ modes are stored, since they have a conjugate symmetry with the coefficients of the $k < 0$ modes. We generally discard the Nyquist mode since this mode is only marginally resolved by the grid: for real functions, for instance, $\cos(k_N x)$ can be captured by the Nyquist mode, but $\sin(k_N x)$ cannot, since the grid points fall on the zeroes of this function.

The expansion coefficients are given explicitly by

$$f_k = \frac{1}{2\pi} \int_0^{2\pi} f(x) \phi_k^{F*}(x) \mathrm{d}x \tag{2.5}$$

$$= \frac{1}{N_g} \sum_{i=0}^{N_g - 1} f(x_i^F) \phi_k^{F*}(x_i^F) \tag{2.6}$$

but can be computed in $\mathcal{O}(N_g \log N_g)$ using the fast Fourier transform (FFT). We implement FFTs from both the Scipy library and the FFTW library, and rescale the results to match the above normalizations, i.e. having 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 beginning with $-k_m$ and increasing to -1.

2.2.2 Sine/Cosine basis

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

$$\phi_k^c(x) = \cos(kx) \tag{2.7}$$

$$\phi_k^s(x) = \sin(kx) \tag{2.8}$$

and a grid consisting of evenly-spaced interior points:

$$x_i^p = \frac{\pi(i + 1/2)}{N_g} \qquad i = 0, ..., N_g - 1. \tag{2.9}$$

Functions that have even parity with respect to the interval endpoints are represented with a cosine series as

$$f(x) = \sum_{k=0}^{N_c-1} f_k \phi_k^c(x) \tag{2.10}$$

while functions that have odd parity with respect to the interval endpoints are represented with a sine series as

$$g(x) = \sum_{k=1}^{N_c-1} g_k \phi_k^s(x). \tag{2.11}$$

The Nyquist mode $k_N = N_c$ is dropped from each sine series, since the corresponding mode is unresolvable by an equivalent-size cosine series.

The expansion coefficients are given explicitly by

$$f_k = \frac{2 - \delta_{k,0}}{\pi} \int_0^\pi f(x)\phi_k^c(x)\mathrm{d}x \tag{2.12}$$

$$= \frac{2 - \delta_{k,0}}{N_g} \sum_{i=0}^{N_g-1} f(x_i^p)\phi_k^c(x_i^p) \tag{2.13}$$

$$g_k = \frac{2}{\pi} \int_0^{\pi} g(x)\phi_k^s(x)\mathrm{d}x \tag{2.14}$$

$$= \frac{2}{N_g} \sum_{i=0}^{N_g-1} g(x_i^p)\phi_k^s(x_i^p) \tag{2.15}$$

but can be computed in $O(N_g \log N_g)$ using the fast discrete cosine transform (DCT) and discrete sine transform (DST). The offset grid is chosen so that the same grid points can be used to represent cosine and sine series, requiring the use of type-II DCT/DSTs for the forward transforms, and type-III DCT/DSTs for the backward transforms. We implement FFTs from both the Scipy library and the FFTW library, and rescale the results to match the above normalizations, i.e. having 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. This is achieved by viewing a complex array as a real array with an additional dimension of size 2 appended to its data shape, since complex floating point numbers are stored as pairs of real floating points numbers representing their real and imaginary parts. 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, respectively.

2.2.3 Chebyshev basis

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

$$\phi_n^T(x) = \cos(n \cos^{-1}(x)) \tag{2.16}$$

The grid for the Chebyshev basis is given by the nodes of Gaussian quadrature for Chebyshev polynomials, also known as the *roots* or *internal* grid:

$$x_i^T = -\cos\left(\frac{\pi(i + 1/2)}{N_g}\right) \qquad i = 0, ..., N_g - 1. \tag{2.17}$$

Near the center of the interval, the grid points approach an evenly-spaced grid

$$x_i^T (i \approx N_g/2) \approx \frac{\pi(i + 1/2)}{N_g} - \frac{\pi}{2} \tag{2.18}$$

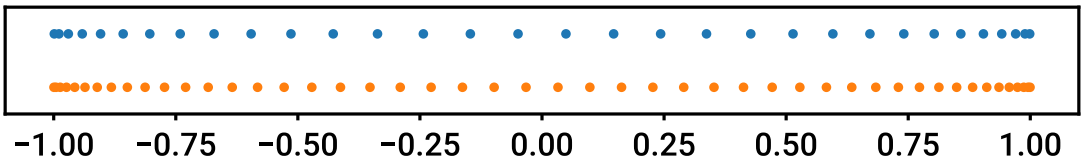

Figure 2.1: The native Chebyshev grids with $N_g = 32$ (top) and $N_g = 48$ (bottom).

while near the ends of the interval the grid points cluster like

$$x_i^T(i \ll N_g) \approx -1 + \frac{1}{2}\left(\frac{\pi(i+1/2)}{N_g}\right)^2 \tag{2.19}$$

$$x_i^T(N_g - i \ll N_g) \approx 1 - \frac{1}{2}\left(\frac{\pi(i+1/2)}{N_g}\right)^2 \tag{2.20}$$

allowing for very small structures near the endpoints to be resolved (Fig. 2.1).

A function is represented as a simple sum over polynomials as

$$f(x) = \sum_{n=0}^{N_c-1} f_n \phi_n^T(x) \tag{2.21}$$

The expansion coefficients are given explicitly by

$$f_n = \frac{2-\delta_{n,0}}{\pi}\int_{-1}^{1} \frac{f(x)\phi_n^T(x)}{\sqrt{1-x^2}}\mathrm{d}x \tag{2.22}$$

$$= \frac{2-\delta_{n,0}}{\pi}\int_{0}^{\pi} f(\cos(\theta))\phi_n^T(\cos(\theta))\mathrm{d}\theta \tag{2.23}$$

$$= \frac{2-\delta_{n,0}}{N_g}\sum_{i=0}^{N_g-1} f(x_i^T)\phi_n^T(x_i^T) \tag{2.24}$$

but can be computed in $O(N_g \log N_g)$ using the fast discrete cosine transform (DCT), since the basis functions and grid points are identical to those for the cosine basis under the change of variables $x^T = -\cos(x^p)$. The Chebyshev basis therefore uses the same Scipy and FFTW DCT functions as the cosine basis, wrapped to handle the sign difference in the change-of-variables and preserve the well-ordering of the Chebyshev grid points. It also behaves similarly for complex functions, preserving the data type and producing complex

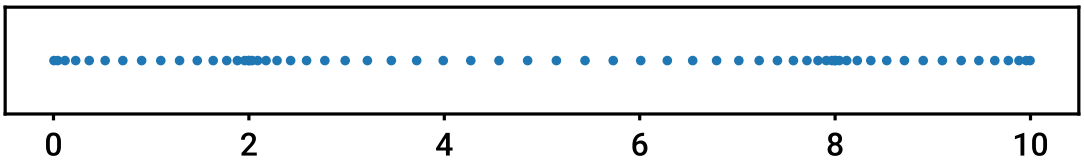

Figure 2.2: The combined grid of a compound basis consisting of a 16-point segment on $[0, 2]$, a 32-point segment on $[2, 8]$, and a 16-point segment on $[8, 10]$.

coefficients that represent the polynomial expansion of the real and imaginary parts of the function.

2.2.4 Compound bases

An arbitrary number of adjacent Chebyshev segments can be connected to from a *compound* Chebyshev basis. In this basis, the spectral coefficients of the Chebyshev expansion of a function on each subinterval are simply concatenated to form the full coefficient vector, and the regular Chebyshev transforms are applied to each subinterval. The compound basis grid is similarly the concatenation of the subinterval Chebyshev grids. Since the interior grid is used for the Chebyshev basis, there are no overlapping gridpoints at the interface between adjacent segments. Continuity of the series is not imposed a priori at the interfaces.

The subintervals making up a compound basis may have different resolutions and different lengths, but must be adjascent. Compound bases are useful for placing higher resolution, due to the clustering of the endpoints of the Chebyshev grid, at fixed interior positions in the domain. 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 non-differentiable are known. Fig. 2.2 shows the grid of a compound basis composed of three subintervals.

2.2.5 Multidimensional transform plans

Each basis implements spectral transforms using the FFTs implemented in both the Scipy library and the FFTW library. Scipy is a Python dependency of Dedalus and must be available on any system running the code. FFTW is a heavily optimized C library for computing FFTs, and allows for fine-grained control over memory usage and transform planning to find the optimal algorithms for a given system. To minimize code reuse

and maximize the extensibility of our algorithms, we require the ability to compute one-dimensional transforms along an arbitrary axis of a multidimensional array.

This functionality is included in the Scipy transforms, and we have constructed Cython wrappers around the FFTW Guru interface to achieve the same functionality using FFTW-based transforms. These 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 instance, 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_1N_2, N_3, N_4N_5) and a loop of $N_1N_2 \times N_4N_5$ transforms of size N_3 would be performed. This approach allows for the unified planning and evaluation of transforms along any dimension of an arbitrary dimension array, 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 for any requested transform for a corresponding data shape. 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.

2.3 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. An instance of the domain class 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.

2.3.1 Parallel data distribution

Parallelization in Dedalus is achieved by directly subdividing and distributing the domain (or rather, any fields defined over a domain) over the available processes in a distributed-memory MPI environment. The domain class internally constructs a *distributor* object that directs this decomposition and the communication necessary to transform 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, allowing spectral transforms to be performed locally on this dimension and requiring parallel data transposes to be performed to change the locality.

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 $(N_1, N_2, ..., N_D)$ distributed over a process mesh of dimension P and shape $(M_1, M_2, ..., M_P)$ [17]:

- The first layout is full coefficient space, where the first P dimensions of the problem are block-distributed over the corresponding dimensions of the process mesh, and the final $D - P$ dimensions are local. That is, the first dimension is split in adjacent blocks of size $B_{1,1} = \text{ceil}\,(N_1/M_1)$, and the process with index $(m_1, m_2, ..., m_P)$ in the mesh will contain the block from $m_1 B_{1,1} : (m_1 + 1)B_{1,1}$ in the first dimension, etc.
- The subsequent $D - P$ layouts will sequentially transform each axis to grid space starting from the last axis and moving backwards.
- After $D - P$ transforms, the first P axes will be distributed and in coefficient space, while the final $D - P$ axes will be local and in grid space. To proceed with the transforms, a global data transposition is necessary to make the P-th axis local in the next layout. This is achieved by transposing the data along the P-th axis of the mesh, which gathers the distributed data along the P-th axis of the array while distributing the data along the $(P + 1)$-th axis. This is an all-to-all communication within each one-dimensional subset of processes in the mesh defined by fixed $(m_1, ..., m_{P-1})$.
- The following layout will result from transforming the P-th axis, which is now local to grid space.
- The transposition step will then repeat to reach the next layout: all-to-all communi-

[17] The default process mesh is a one-dimensional mesh containing all available MPI processes.

cations will be used to change the distribution over the $(P-1)$-th axis of the mesh from being over the $(P-1)$-th dimension to the P-th dimension. The $(P-1)$-th dimension will be transformed to grid space, and the process will repeat until the first axis becomes local and is transformed to grid space.

The final layout is thus in full grid space, where the first dimension is local, the next P dimensions are distributed over the process mesh in blocks of size $B_{n+1,n} = \text{ceil}\,(N_{n+1}/M_n)$, 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 transposes.

Fig. 2.3 shows the data distribution in each layout for data with a global shape $(16, 16, 16)$ distributed over a process mesh of shape $(4, 2)$. This 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 simple bubbles down the first local axis 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 $(M_i - k)\text{ceil}\left(N_j/M_i\right) > N_j$ for any distributed state where the j-th dimension of the domain is distributed over the i-th dimension of the mesh, then the last k hyperplanes along the i-th dimension of the mesh will be empty. These cases are typically avoidable by choosing a different process mesh shape for a fixed number of processes.

For simplicity, we have considered the global data shape to be fixed throughout the transform process. The implementation, however, handles arbitrary transform scales along each axis, meaning $N_i = N_{c,i}$ when the i-th axis is in coefficient space, and $N_i = N_{g,i} = s_i N_{c,i}$ when the i-th axis is in grid space.

2.3.2 Transpose routines

Consider the first transpose in the process of moving from coefficient space to grid space, i.e. transposing the distribution over the P-th mesh axis from the P-th data axis to the $(P+1)$-th data axis. This transform does not change the data distribution over the previous process axes; instead it consists of m_P separate all-to-all calls within each one-dimensional subset of processes defined by fixed $(m_1, ..., m_{P-1})$. Thus each transpose is planned by first creating a subgrid MPI communicator from the cartesian MPI communicator built in accordance with the process mesh. This subgrid communicator consists of the m_p processes with fixed $(m_1, ..., m_{P-1})$. This communicator then plans for the transpose of

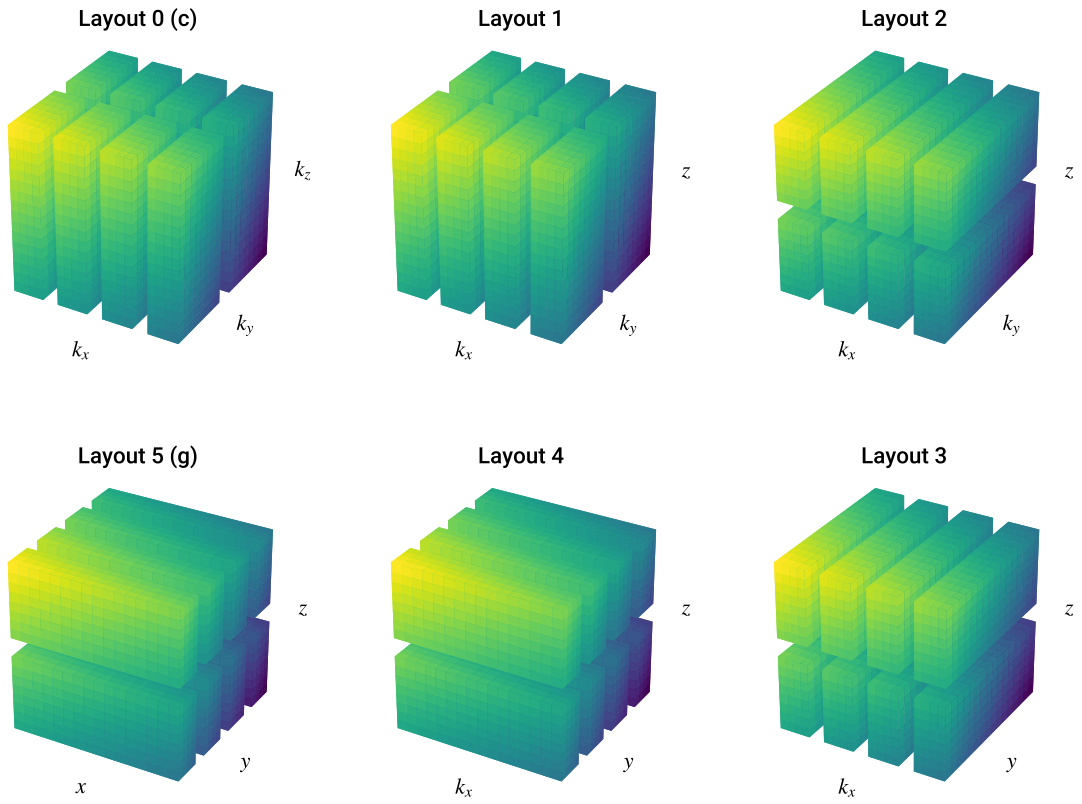

Figure 2.3: The parallel data distribution for data with global shape $(16, 16, 16)$ over a process mesh of shape $(4, 2)$. The global data is depicted as being split into the portions that are local to each process. The axes are labeled e.g. k_x when the corresponding axis is in coefficient space, and e.g. x in grid space. The following sequence of transforms (TF) and transposes (TP) is used to step between full coefficient space and full grid space: TF(z), TP(y,z), TF(y), TP(x,y), TF(x).

an array of shape $(B_0, ..., B_{P-1}, N_P, N_{P+1}, N_{P+2}, ..., N_D)$, i.e. the subspace of the global data spanned by the subgrid processes. This array is viewed as a four-dimensional array of shape $(B_0 \times ... \times B_{P-1}, N_P, N_{P+1}, N_{P+2} \times ... \times N_D)$, constructed by collapsing the pre- and post-transpose axes. In this way, the general case of transposing an arbitrary-dimensional array distributed over an arbitrary-dimensional process mesh along an arbitrary axis is reduced to the problem of transposing a four-dimensional array across its middle two axes.

In the general case of transposing the distribution along the p-th mesh axis between the p-th and $(p+1)$-th data axes, the *global subgrid shape* is given by $(n_1, ..., n_d)$ where

$$n_i = \begin{cases} B_{i,i} & i < p \\ N_i & i = p,\ p+1 \\ B_{i,i-1} & p+1 < i \leq P+1 \\ N_i & i > P+1 \end{cases} \tag{2.25}$$

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

$$G_1 = \prod_{i=1}^{p-1} n_i, \quad G_2 = n_p, \quad G_3 = n_{p+1}, \quad G_4 = \prod_{i=p+2}^{D} n_i \tag{2.26}$$

The general case is thus reduced to viewing the global subgrid data of a field as a reduced four-dimensional array, and transposing this array between having its second and third axes distributed over m_p processes.

Routines for performing this transpose are implemented using both basic MPI all-to-all calls, and FFTWs advanced distributed-transpose interface. The MPI version begins with the local subgrid 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

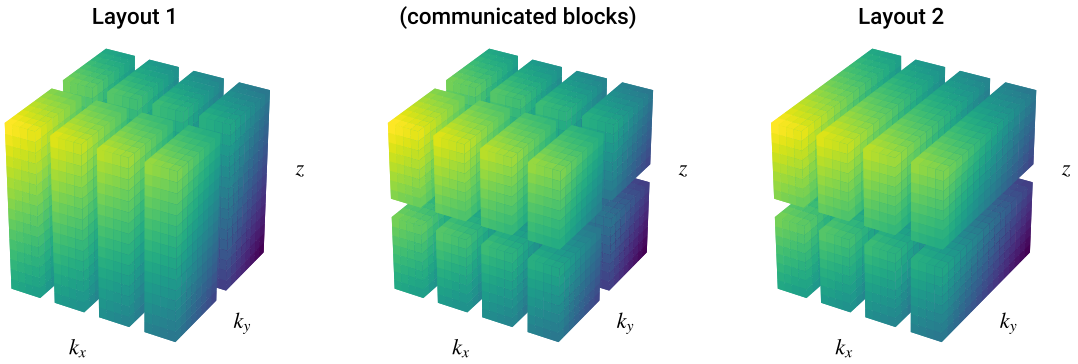

Figure 2.4: The effective data redistribution that occurs during the distributed transpose between layouts 1 and 2 of the example shown in Fig. 2.3. This transpose is switching the the second mesh axis with $m_2 = 2$ from distributing the k_y axis to distributing the z axis.

extracted from the MPI buffer to form the local subgrid 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 transpose 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) and an itemsize of $G_1 \times G_4 \times Q$, where Q is the actual data itemsize.

Fig. 2.4 shows the conceptual domain redistribution strategy for the transpose between layouts 1 and 2 of the example shown in Fig. 2.3. 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 a MPI-linked FFTW build is available. The FFTW planning rigor and in-place directives for the transposes are also wrapped through the Dedalus configuration interface.

These routines can also be used to group transposes of multiple fields simultaneously. When F fields need to be transposed, their local subgrid data is simply concatenated, and the reduced global subgroup shape is expanded to $(F \times G_1, G_2, G_3, G_4)$, and a plan is constructed and executed for this expanded shape. This method allows for the simultaneous transposition of multiple fields while reducing the latency associated with initiating a transpose for each

field. The option to group multiple transpositions in this manner is controlled through the Dedalus configuration interface.

2.3.3 Distributed data interaction

The layout objects contain methods providing the global data shape, local data shape, block sizes, local data coordinates, and local data slices for fields in that layout under arbitrary transform scalings in each dimension. These methods provide the user with 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 the global indices of the local spectral coefficients assigned to each process. These local arrays are useful for initializing the values of a field in either grid space or coefficient space. In particular, writing code that initializes all grid or coefficient data using references to these local arrays makes it robust to changing parallelization scenarios. That is, initial conditions can easily be written to construct the same fields regardless of the number of processes being used, allowing for scripts to be tested serially on local machines then executed on large systems without modification.

2.4 Fields

Field objects represent scalar-valued fields defined over a domain. They are instantiated with an optional name and the domain object representing their spectral domain. Each field object contains a metadata dictionary defining whether that field is constant along any axis, the transform scales to be used along each axis when transforming the field, and any other metadata associated with specific bases (such as the field parity for a the sine/cosine basis). 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 the current layout of the field, and a data attribute consisting of a view of its memory buffer using the local data shape and data type of the current layout.

2.4.1 Data manipulation

The field class defines a number of methods for transforming individual fields between layouts. The most basic are methods that move the field towards grid or coefficient space by calling the necessary transform or transpose to increment or decrement the layout by a single step. Other methods direct the transformation to a specific layout by taking sequential steps, and moving to full coefficient space or full grid space, the first and last layouts, respectively. These methods allow users to transparently 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 are set to allow retrieving or setting the local field data in any layout. Additionally, shortcuts `'c'` and `'g'` allow fast access to to the full coefficient and grid data, respectively. Completing a fully parallelized, distributed transform is simply achieved by:

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

The transform scales are modified using the `set_scales` method:

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

2.4.2 Field Systems

A set of fields can be grouped together to form a system using the `FieldSystem` class. The purpose of this class is to provide an interface to efficiently access the interleaved coefficients corresponding to the same horizontal mode, or pencil, of a group of fields. By *horizontal mode*, we mean 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 horizontal mode has a corresponding 1D *pencil* of coefficients along the last axis of the multidimensional coefficient data of a field. When solving PDEs that are only linearly coupled in the last dimension, the linear portion of the PDE splits into separate matrix systems for each horizontal mode, which makes efficiently accessing the corresponding pencils from each of the field variables desirable.

Specifically, a system of F-many fields will build an internal buffer of size

$$(B_1, ..., B_P, N_{P+1}, ..., N_{D-1}, N_D \times F) \tag{2.27}$$

that is, the local coefficient shape with the last axis size multiplied by the number of fields. The system methods `gather` and `scatter` will interleave the separate field coefficients into this buffer, with the first coefficients of the fields along the last axis together, following by the second coefficients, etc. Thus each system pencil of size $N_D \times F$ will contain the corresponding pencils of each of the fields, grouped by mode, in a contiguous block of memory for efficient access.

A `CoeffSystem` is a similar object, which merely allocates and controls the unified buffer rather than also instantiating field objects. These systems are used when temporary arrays are needed for all pencils, avoiding the memory overhead associated with instantiating new field objects.

2.5 Operators

2.5.1 Operator classes

Mathematical operations on fields, such as arithmetic, differentiation, integration, and interpolation, are represented by *Operator* classes. An instance of an operator class represents a specific mathematical operation on a field or set of fields, and provides an interface for the deferred evaluation of that operation.

Operators are based on an abstract base class that outlines and simplifies the implementation of individual operators. Operators may accept operands (fields or operators) from the same domain and other parameters such as constants as arguments. Each operator class must implement methods determining the metadata of the output based on the inputs, for instance the parity of the output if a sine/cosine basis is used. They must also implement a `check_conditions` method that checks whether or not the operation can be executed in a given layout. For instance, spectral differentiation along some axis requires that axis to be in coefficient space, and possibly local if the derivative couples different modes. Finally, operators must implement an `operate` method which performs the operation using the local data of the inputs once they have been placed in a suitable layout.

Operators can be composed to build complex expressions. An arbitrary expression is still an instance of the root operator class, but with operands that are themselves instances of operator classes, eventually with fields or other input parameters forming the leaves at the end of the expression tree (see Fig. 2.5). Such an operator is evaluated via the `evaluate` method, which is implemented by the operator base class. This method evaluates the

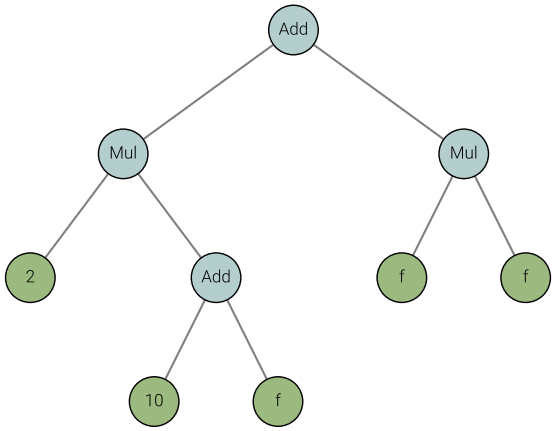

Figure 2.5: An operator tree representing the expression `2*(10+f) + f*f`.

operator by recursively evaluating all of its operator operands, transforming them to the proper layout, and calling the implemented `operate` method. Arbitrary expression trees are therefore evaluated in a depth-first traversal of the tree. The `evaluate` method can optionally cache its output if it may be called multiple times before the values of the field leaves change. The `attempt` method attempts to evaluate a field, but will not initiate 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 must implement a number of methods allowing for algebraic manipulation of expressions, which will be described in §2.5.5.

2.5.2 Arithmetic operators

Addition, multiplication, and exponentiation are implemented via the `Add`, `Multiply`, and `Power` classes, respectively. Different subclasses of these operators are invoked depending on the types of the operands. This multiple dispatch system is implemented as a Python metaclass – the class of a class – which overrides the `__call__` method on the class to examine the arguments before instantiating an operator of the proper subclass.

Addition of two fields is implemented in the `AddFieldField` subclass and operates by adding the local data of each field. This operation can be performed in any layout, as long as both fields are in the same layout. Addition between fields using the sine/cosine basis will raise an error if the fields have different parity. Addition of a field and a constant is

implemented in the `AddScalarField` subclass and operates by adding the constant to the local data in grid space. If the constant is `0`, instantiation will be skipped and the other operand will be returned. The addition of an odd-parity series and a scalar will raise an error, since a constant is a function of even parity.

Multiplication of two fields implemented in the `MultiplyFieldField` subclass, and operates by multiplying the local data of the two fields in grid space. Multiplication of fields using the sine/cosine basis yields a field with a parity that is the product of the input parities, that is an even parity output for two odd or two even inputs, and an odd parity output for an odd and an even input. Multiplication of a field and a constant is implemented in the `MultiplyScalarField` subclass and operates by multiplying the field's local data by the constant in any layout. If the constant is `0` or `1`, instantiation will be skipped and `0` or the other operand will be returned, respectively.

The *Power* operator allows a field to be raised to any scalar power, and is evaluated in grid space. A field using the sine/cosine basis can be raised to integer powers, with the parity of the output equal to the input parity raised to the same power. If the power is `0` or `1`, instantiation will be skipped and `1` or the operand will be returned, respectively.

The `__add__`, `__mul__`, and `__pow__` methods of the field and operator classes are overloaded to allow Python infix operators to be used to easily construct arithmetic expressions on fields. For instance, the expression `f + 5` with `f` a Dedalus field will produce an `AddScalarField` instance. The `__neg__`, `__sub__`, and `__truediv__` methods for negation, subtraction, and division are also overridden, and reduced to the other arithmetic operations as follows:

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

2.5.3 Unary grid operators

Many common nonlinear unary functions are implemented through the `UnaryGridFunction` class, which accepts as its arguments a Numpy universal function and an operand. The supported unary functions are: `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 unary function is stored as an argument alongside the target operand, and the

operation proceeds by applying this function to the local grid space data of the operand. The `__getattr__` method of the field and operator classes are overloaded to intercept Numpy universal function calls and instantiate the corresponding operator. This allows the direct use of Numpy ufuncs to create operators on fields, for instance `np.sin(f)` on a Dedalus field `f` will return `UniveryGridFunction(np.sin, f)`.

2.5.4 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 therefore

$$A_{ij} = \langle \phi_i | A \phi_j \rangle \tag{2.28}$$

where the inner product is that under which the basis functions are orthonormal. 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, these operators are applied by requiring that the corresponding axis of their operand is in coefficient space. Coupled operators further require that the corresponding axis is local. The local data of the operand is then contracted with the matrix form of the operator along this axis to produce the local output data. Operators may override this process by implementing a `explicit_form` method if a more efficient or stable algorithm exists for forward-applying the operator.

Differentiation

The `DifferentiateFourier` class implements differentiation of the Fourier basis. This is a separable operator as Fourier differentiation does not couple different modes. The Fourier differentiation matrix D^F is given by

$$\partial_x \exp(ikx) = ik \exp(ikx) \qquad \Longrightarrow \qquad D^F_{k,k'} = ik\delta_{k,k'} \tag{2.29}$$

where the k in the matrix entry expression is taken to be the signed wavenumber of the corresponding mode.

The `DifferentiateSinCos` class implements differentiation of the sine/cosine basis. This is a separable operator which flips the parity of its operand. The sine/cosine differentiation matrices D^S and D^C are given by

$$\partial_x \sin(kx) = k\cos(kx) \quad \Longrightarrow \quad D^S_{k,k'} = k\delta_{k,k'} \tag{2.30}$$

$$\partial_x \cos(kx) = -k\sin(kx) \quad \Longrightarrow \quad D^C_{k,k'} = -k\delta_{k,k'} \tag{2.31}$$

The `DifferentiateChebyshev` class implements differentiation of the Chebyshev basis. This is a coupled operator, as differentiation lowers the order of polynomials and therefore must couple different Chebyshev modes. The derivatives of the Chebyshev polynomials can be shown to satisfy the recurrence relation

$$\frac{\partial_x T_n}{n} = 2T_{n-1} + \frac{\partial_x T_{n-2}}{n-2} \tag{2.32}$$

which can be used to derive an explicit form for the entries of the Chebyshev differentiation matrix D^T:

$$D^T_{i,j} = \frac{2j((j-i) \bmod 2)}{1+\delta_{i,0}}[i<j] \tag{2.33}$$

This is a dense, upper triangular matrix mapping the derivatives of Chebyshev polynomials back to themselves. Forward-applying this matrix takes $O(N_c^2)$ time, but the result can be computed in $O(N_c)$ time using the following recursion for the output derivative coefficients `B` given the input coefficients A:

```
def recursive_chebyshev_derivative(A, B):
    N = A.size
    B[N-1] = 0
    B[N-2] = 2 * (N-1) * A[N-1]
    for n in range(N-3, 0, -1):
        B[n] = 2 * (n+1) * A[n+1] + B[n+2]
    B[0] = A[1] + B[2] / 2
```

A generalized version of this recursion enabling it to be applied to a multidimensional field is implemented in Cython as the `explicit_form` of the `DifferentiateChebyshev` class. We also note that this dense matrix form of the derivative is never used when implicitly solving a differential equation in Dedalus. As we will discuss in §2.7.1, the Chebyshev T-to-

U conversion is always applied before performing implicit solves, rendering the Chebyshev differentiation matrices banded.

The native differentiation matrices for each basis are rescaled by the inverse of the stretching in the affine transformation between the native and problem coordinates to produce the problem differentiation matrices. The differentiation matrix for the compound basis is simply the block-diagonal combination of the subbasis differentiation matrices. For each basis in a domain, a new differentiation subclass of the corresponding type is created and linked to the `Differentiate` attribute of the basis class. These methods are typically aliased to `'d%s'%basis.name`, e.g. `dx` for a basis with name `'x'`. 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 examining the bases of the operand and 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.

Integration

Integration over the native interval of a basis is a functional that returns a constant for any input basis series, so integration operators all set the `'constant'` metadata of the corresponding axes of their outputs to `True`. The operator matrices are therefore nonzero except in the first row, which we refer to as the operator vector.

The `IntegrateFourier` class implements integration along a Fourier basis. It is a separable operator, as only the constant term in the Fourier series has a non-zero integral over the native interval. The Fourier integration vector I^F is simply

$$I^F_j = 2\pi\delta_{j,0} \tag{2.34}$$

The `IntegrateSinCos` class implements integration along a sine/cosine basis. While integration of a cosine series is separable, since only the constant term has a non-zero integral over the native interval $[0, \pi]$, every other basis function in the sine series has a non-zero integral, so the operator is generally considered as coupled. The output of integrating

either series is a constant, and hence has even parity. The sine and cosine integration vectors are given by

$$I_j^C = \pi\delta_{j,0} \tag{2.35}$$

$$I_j^S = \frac{1-(-1)^j}{j} = \frac{2}{j}(j \bmod 2) \tag{2.36}$$

The `IntegrateChebyshev` class implements integration along a Chebyshev basis. This is a coupled operator since all the even Chebyshev polynomials have non-zero integrals. The Chebyshev integration vector is given by

$$I_j^T = \int_{-1}^{1} T_j(x)\mathrm{d}x = \frac{1+(-1)^j}{1-j^2} \tag{2.37}$$

The native integration matrices for each basis are rescaled by the stretching of the affine transformation between the native and problem coordinates to produce the problem integration matrices. Integration for the compound basis simply concatenates each of the subbasis integration vectors, and places the result in the rows corresponding to the constant terms in each subbasis. For each basis in a domain, a new integration subclass of the corresponding type is created and linked to the `Integrate` attribute of the basis class. 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')  # Becomes 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 operand and the problem interval length will be returned.

Interpolation

Interpolation operators are all coupled linear operators whose arguments are the operand to be interpolated and the position in problem coordinates where the value of the operand series should be interpolated. Interpolation operators are also all functions returning a constant and so set the `'constant'` metadata of the corresponding axes of their outputs to `True`. The operator matrices are again nonzero except in the first row, which we refer to as the operator vector, and depend on the interpolation position. In addition to any position

within the problem interval, the strings `'left'`, `'center'`, and `'right'` are acceptable inputs indicating the left endpoint, center point, and right endpoint of the problem interval. A numerically specified position is converted to native coordinates via the basis affine transformation, but this step is skipped for the string inputs as the corresponding positions are simply the left, center, and right of the native interval. Skipping this conversion avoids potential floating point errors that can occur when evaluating the affine transformation near the endpoints.

The interpolation classes for each basis simply construct interpolation vectors consisting of the pointwise evaluation of their respective basis functions at that position:

$$E_n(x) = \phi_n(x) \tag{2.38}$$

Interpolation for the compound basis takes the interpolation vector of the subbasis containing the interpolation position and places the result in the rows corresponding to the constant terms in each subbasis. If the interpolation position is at the interface between two subbasis, the first subbasis is used simply to break the degeneracy. For each basis in a domain, a new interpolation subclass of the corresponding type is created and linked to the `Interpolate` attribute of the basis class. 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:

```
integ(f, x=0.5, y=1)
    # Becomes 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.

Hilbert transforms

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

$$H(f)(x) = \frac{1}{\pi} \int_{-\infty}^{\infty} \frac{f(x')}{x - x'} \mathrm{d}x' \tag{2.39}$$

where the integral is evaluated in the principal value sense. The Hilbert transform has a particularly simple action on sinusoids, with

$$H(\exp(ikx))(x) = -i\mathrm{sgn}(k)\exp(ikx) \tag{2.40}$$

$$H(\sin(kx))(x) = -\mathrm{sgn}(k)\cos(kx) \tag{2.41}$$

$$H(\cos(kx))(x) = \mathrm{sgn}(k)\sin(kx) \tag{2.42}$$

The `HilbertTransformFourier` and `HilbertTransformSinCos` classes implement the Hilbert transform for the Fourier and sine/cosine bases. These are separable operators implementing the above identities in their matrix forms:

$$H^F_{k,k'} = -i\mathrm{sgn}(k)\delta_{k,k'} \tag{2.43}$$

$$H^S_{k,k'} = -\mathrm{sgn}(k) \tag{2.44}$$

$$H^C_{k,k'} = \mathrm{sgn}(k) \tag{2.45}$$

where k is taken to be the signed wavenumber of a mode in the Fourier case, and the parity operator flips the parity of its input.

For each Fourier or sine/cosine basis in a domain, a new Hilbert transform subclass of the corresponding type is created and linked to the `HilbertTransform` attribute of the basis class. These methods are typically aliased to `'H%s' %basis.name`, 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.

2.5.5 Manipulating expressions

The operator classes additionally implement a number of methods that allow for the algebraic manipulation of operator expressions, forming a simple computer algebra system. These methods are:

- `atoms`: This method recursively constructs the set of leaves of an expression matching a specified type.
- `has`: This method recursively determines whether an expression contains any given

variable or operator type.

- `expand`: This method 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.
- `canonical_linear_form`: This method 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.
- `split`: This method additively splits an expression into a set of terms containing specified operands and operators, and a set of terms not containing any of them.
- `replace`: This method performs a depth-first search of an expression, replacing any instances of a specified operand or operator with a specified replacement.
- `order`: This method recursively determines the number of times a specified operator is applied to an operand containing the same operator.
- `sym_diff`: This method produces a new expression containing its symbolic derivative with respect to a specified variable. These are computed recursively via the chain rule.
- `as_ncc_operator`: This method constructs the Chebyshev multiplication matrix associated with an operand. It requires that the corresponding domain only have a single Chebyshev basis, and that this basis forms the last axis of the domain, which we'll refer to as z. It further requires that the operand is constant along all other ("horizontal") axes, so that multiplication by the operand does not couple the horizontal modes. Given a Chebyshev expansion of a function f as

$$f(z) = \sum_{n=0}^{N_c-1} f_n \phi_n^T(z) \tag{2.46}$$

the corresponding Chebyshev multiplication matrix F is given by

$$F = \sum_{n=0}^{N_c-1} f_n M_n^T \tag{2.47}$$

where the individual-mode Chebyshev multiplication matrices are given by

$$(M_n^T)_{i,j} = \langle \phi_i^T | \phi_n^T \phi_j^T \rangle \tag{2.48}$$

$$= \frac{\delta_{i,|n+p|} + \delta_{i,|n-p|}}{2} \tag{2.49}$$

and the inner product is that under which the Chebyshev modes are orthonormal. This matrix therefore represents multiplication by the function f as a linear operator acting on the Chebyshev coefficients of another function. That is, if $g(z) = f(z) \times h(z)$, we have

$$g_i = \langle \phi_i^T | g(z) \rangle \tag{2.50}$$

$$= \langle \phi_i^T | \sum_n f_n \phi_n^T \sum_j h_j \phi_j^T \rangle \tag{2.51}$$

$$= \sum_{n,j} f_n h_j (M_n^T)_{i,j} \tag{2.52}$$

$$= \sum_j F_{i,j} h_j \tag{2.53}$$

The method producing this matrix allows the sum to be truncated at a maximum number of modes ($N_m < N_c$) and for terms to be excluded when the coefficients are below some threshold amplitude ($|f_n| < \delta$) so that the matrix is sparse for well-resolved functions (see Olver et al. (2013)). The resulting matrix is stored as a Scipy sparse matrix.

- `operator_dict`: This method constructs a dictionary representing an expression as a set of matrix operators acting on the pencils of coefficients for a specified horizontal mode of a set of variables. 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 z linear operator applying its matrix form to the matrices produced by its operand, and each horizontal linear operator multiplying its operand matrices by the element of its vector form corresponding to the specified horizontal mode. Addition operators simply sum the matrices produced by their operands. Multiplication operators build the matrices for their operand containing one of the specified variables, and multiply these by the NCC matrix form of their other operand.

With these methods, operators in Dedalus form a simple computer algebra system allowing for the basic symbolic manipulation of expressions. This functionality forms the basis of the code's ability to construct solvers for general partial differential equations, as described in §2.6.

2.5.6 Evaluators

An `Evaluator` object attempts to simultaneously evaluate a number of operator expressions as efficiently as possible, i.e. with the least number of spectral transforms and distributed transposes. An evaluator is instantiated with a reference to a domain object, and a dictionary of operators and fields that form a namespace for the relevant expressions.

The expressions to be evaluated are organized into `Handler` objects. An individual handler is a collection of operator expressions, a set of criteria for when to evaluate the handler, and a `process` method for dealing with the outputs from the evaluated expressions. Expressions, or tasks, are added to a handler via the `add_tasks` method, which directly accepts operator expressions or strings which are parsed into operator expressions using the evaluator namespace. When running an initial value problem, the handler can be set to 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 repeatedly save their outputs to disk in HDF5 files via the h5py package (§2.8).

When triggered, the evaluator examines the attached handlers and builds a list of the tasks from each handler that is scheduled to be evaluated. It then attempts each task using the operators' `attempt` methods to evaluate the expressions as far as possible without triggering any transforms or transposes. 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 have again not all completed, the evaluator again merges the remaining atoms from the remaining tasks, and 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 transforming, it makes sure that no layout changes are triggered when any operators are able to be evaluated. Additionally, it groups together all the fields

that need to be transformed between layouts so that grouped transforms and transposes can be performed, minimizing the overhead and latency of these functions.

2.6 Problems

The problem classes are used to construct and represent systems of partial differential equations. Separate classes are used for linear boundary value problems (`LBVP`), nonlinear boundary value problems (`NLBVP`), eigenvalue problems (`EVP`), and initial value problems (`IVP`). After a problem is created, the equations and boundary conditions are entered in plain text, with linear terms on the LHS and nonlinear terms on the RHS. The LHS is parsed into a sparse matrix formulation, while the RHS is parsed into an operator tree to be evaluated explicitly.

2.6.1 Problem creation

Each problem class is instantiated with a domain and a list of variable names. Domains may only have up to one Chebyshev basis, which must correspond to the last axis. The linear portion of the equations must be no higher than first-order in time derivatives and Chebyshev derivatives, so auxiliary variables should be added to the problem to render the system first-order. Optionally, an amplitude threshold and a cutoff mode number can be specified for truncating the spectral expansion of non-constant coefficients entered into the problem. For eigenvalue problems, the eigenvalue name must also be passed. 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'])
```

2.6.2 Meta-data and preconditioning

Metadata for the problem variables can be specified through the `meta` attribute, and indexing by variable name, axis, and property, respectively.

The most common metadata to set here is the constancy of any variables, the parity of all variables for each sine/cosine basis, and the `'dirichlet'` option for Chebyshev basis. This option performs a Dirichlet preconditioning / basis-recombination that sparsifies Dirichlet

boundary conditions (interpolation at the Chebyshev interval endpoints), at the expense of a slightly increased problem bandwidth. This can drastically improve performance for problems formulated with only Dirichlet boundary conditions. Because the formulation is first-order in Chebyshev derivatives, this often includes what would be e.g. Neumann boundary conditions in a higher-order formulation.

For instance, we can apply Dirichlet preconditioning to all the variables in our problem along the z axis with

```
problem.meta[:]['z']['dirichlet'] = True
```

2.6.3 Parameters and non-constant coefficients

Before the equations are added to the problem, any parameters, defined as fields or scalars used in the equations besides the problem variables, are added to the parsing namespace through the `problem.parameters` dictionary. Scalar parameters are simply entered by value. Non-constant coefficients (NCCs) are entered as fields with the desired data. NCCs used on the LHS can only couple the Chebyshev direction, so must be constant along all other axes.

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

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

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

2.6.4 Substitutions

To simplify equation entry, substitutions can be specified which act as string-replacement rules that will be applied during the parsing process. Substitutions can be used to provide short aliases to quantities computed from the problem variables and to define shortcut 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.
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 an extremely valuable tool for simplifying the entry of complex equation sets.

2.6.5 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 horizontal modes that equation applies to. This is necessary to close certain equation sets where, for instance, the basic equations become degenerate for the horizontal-mean mode and/or gauge conditions need to be set on certain variables.

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 variables, parameters and substitutions defined in the problem.
- The axis names representing the individual basis grids.
- The derivative, integration, and interpolation operators for each basis.
- Time and temporal derivatives as `'t'` and `'dt'` for the IVP.
- The eigenvalue string for the EVP.
- 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 Chebyshev derivatives. The expressions entered as boundary conditions must be constant along the Chebyshev axis.

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

Linear boundary value problem forms

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

$$\mathcal{L} \cdot \mathcal{X} = \mathcal{F} \tag{2.54}$$

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

Nonlinear boundary value problem forms

The nonlinear boundary value problem requires no additional conditions, allowing the RHS to be any nonlinear function of the problem variables. This corresponds to systems of equations of the form

$$\mathcal{L} \cdot \mathcal{X} = \mathcal{F}(\mathcal{X}) \tag{2.55}$$

In addition to the $\mathcal{L}$ and $\mathcal{F}$ expressions, the Frechet differential of the RHS with respect to the problem variables

$$\mathcal{F}_\mathcal{X} \cdot \Delta\mathcal{X} = \partial_\epsilon \mathcal{F}(\mathcal{X} + \epsilon\Delta\mathcal{X})|_{\epsilon=0} \tag{2.56}$$

is constructed. This is a linear operator indicating 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 §2.5.5 roughly as

```
    dF = 0
    for var, pert in zip(vars, pert):
        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 non-constant coefficients involving the problem variables $\mathcal{X}$, which would generally couple horizontal modes. Therefore, Dedalus only supports 1D nonlinear BVPs. The LHS and Frechet differential expressions are expanded and transformed into canonical linear form before being stored by the problem instance.

Eigenvalue problem forms

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

$$\sigma \mathcal{M} \cdot \mathcal{X} + \mathcal{L} \cdot \mathcal{X} = 0 \tag{2.57}$$

which are generalized linear eigenvalue problems. The $\mathcal{M}$ and $\mathcal{L}$ 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.

Initial value problem forms

The initial value problem requires that the LHS coefficients are time independent, the LHS is first-order in temporal derivatives, and the RHS is independent of temporal derivatives. This corresponds to systems of equations of the form

$$\mathcal{M} \cdot \partial_t \mathcal{X} + \mathcal{L} \cdot \mathcal{X} = \mathcal{F}(\mathcal{X}, t) \tag{2.58}$$

The $\mathcal{M}$ and $\mathcal{L}$ 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.

2.7 Solvers

Each problem type has a corresponding solver type which builds and solves the spectral matrices for the problem equations.

2.7.1 Matrix construction

The problem classes begin by building the spectral operator matrices for the LHS expressions groups ($\mathcal{M}$, $\mathcal{L}$, and $\mathcal{F}_\mathcal{X}$). The matrices are constructed by first taking the set of equations and boundary conditions that apply to 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, F. For a given pencil, we refer to the operator matrix from the i-th equation acting on the j-th variable as $E^{i,j}$. Each of these matrices is processed as follows:

- If the i-th equation contains a Chebyshev derivative on the LHS, each $E^{i,j}$ is left-multiplied by the Chebyshev T-to-U conversion matrix. This has the effect of reprojecting all Chebyshev differential equations into Chebyshev U polynomials, rendering all derivative matrices banded, at the expense of slightly increasing the bandwidth of the non-derivative matrices.
- If the i-th equation contains a Chebyshev derivative on the LHS, the last row of the each $E^{i,j}$ is dropped and replaced with one of the boundary conditions. This implements the boundary conditions using the tau method, with a tau polynomial corresponding to $\phi^U_{N_c-1}$. For the system to be solvable, it also requires that the same number of boundary conditions as Chebyshev differential equations.
- If the last basis is a compound basis, the rows corresponding to the final coefficient of each subbasis, except for the last, are dropped and replaced with internal boundary conditions matching the subbasis coefficients at each internal interface for each variable. This enforces continuity of all the fields across the Chebyshev subsegments.
- If the j-th variable has been marked for Dirichlet preconditioning, the boundary conditions are moved to the top of each $E^{i,j}$, which are then right-multiplied by

the Chebyshev T-to-D conversion matrix. This has the effect of rearranging the columns so that the matrix acts on the coefficients of the Chebyshev Dirichlet basis expansion of the corresponding variable, reducing all Dirichlet boundary conditions from spanning N_z^c columns to spanning the just the first two columns, at the expense of slightly increasing the bandwidth of the equation matrices.

Finally, the fully processed operator matrices are joined to produce the full preconditioned pencil matrix $\tilde{E}$. Conceptually, the full pencil matrix can be thought of as the block matrix combining the operator matrices, with the columns of blocks corresponding to the problem variables, and the rows of blocks corresponding to the equations:

$$\tilde{E} = \begin{bmatrix} \tilde{E}^{1,1} & \tilde{E}^{1,2} & \dots & \tilde{E}^{1,F} \\ \tilde{E}^{2,1} & \tilde{E}^{2,2} & \dots & \tilde{E}^{2,F} \\ \vdots & \vdots & \ddots & \vdots \\ \tilde{E}^{F,1} & \tilde{E}^{F,2} & \dots & \tilde{E}^{F,F} \end{bmatrix} \tag{2.59}$$

Such a matrix can be constructed via Kronecker products with a size $F \times F$ placement matrix $P^{i,j}$, where

$$(P^{i,j})_{k,l} = \delta_{i,k}\delta_{k,l} \tag{2.60}$$

as

$$\tilde{E} = \sum_{i,j=0}^{F} P^{i,j} \otimes \tilde{E}^{i,j} \tag{2.61}$$

However, even when the individual $\tilde{E}^{i,j}$ are band-limited, $\tilde{E}$ will have a bandwidth of order FN_z^c. By reversing the order of the Kronecker product and building the pencil matrix as

$$\tilde{E} = \sum_{i,j=0}^{F} \tilde{E}^{i,j} \otimes P^{i,j} \tag{2.62}$$

the columns and rows are grouped first by Chebyshev mode number rather than by field, as is the memory ordering for `FieldSystems`. The bandwidth of the pencil matrix then becomes F times the maximum bandwidth of any of the individual subblocks, which is roughly set by the bandwidth of the non-constant coefficient expansions. Interleaved-block matrices corresponding to the full left (T-to-U) and right (Dirichlet) preconditioning that was applied to the subblocks are created and stored, since they will need to be applied to the RHS and solution vectors, respectively, when solving the matrix system.

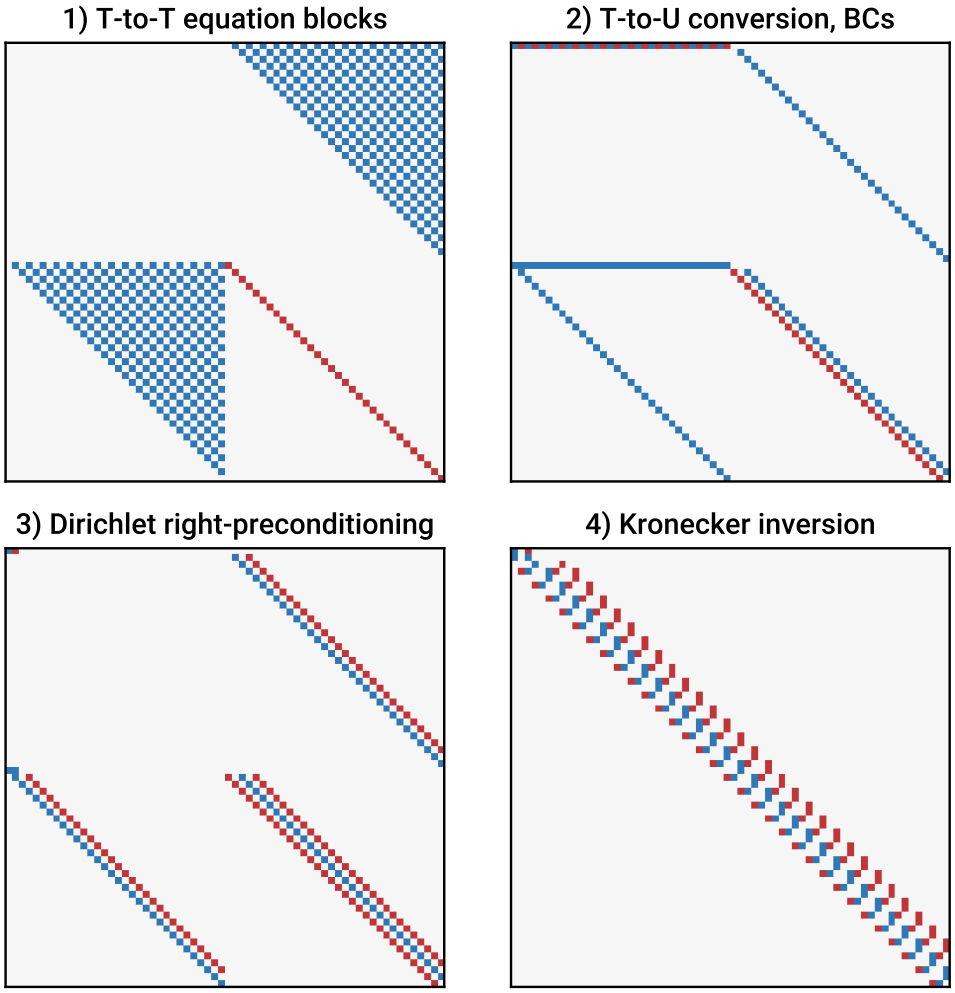

Figure 2.6: Various stages in constructing the matrices corresponding to the linear portion of Poisson's equation with Dirichlet boundary conditions. Top left: the original (T-to-T) operator matrices joined in block form, with the block columns corresponding to the variables `u` and `ux`, and the block rows corresponding to the LHS expressions `dx(ux)` and `dx(u) - ux`, respectively. The T-to-T differentiation matrices are dense and upper triangular. Top right: the matrices after T-to-U conversion has been applied and the boundary conditions have been prepended. The derivatives are rendered sparse by the T-to-U conversion at the expense of slightly increasing the bandwidth of the identity block. The boundary conditions involve all coefficients of `u`. Bottom left: the matrices after Dirichlet preconditioning has been applied from the right, sparsifying the boundary rows at the expense of slightly increasing the bandwidth of the equation blocks. Bottom right: the same matrix, but with the Kronecker product reversed to group by modes rather than by variables. This final matrix is highly sparse and completely banded.

Fig. 2.6 shows the system matrices at various points of this process for Poisson's equation in 1D with Dirichlet boundary conditions:

$$\frac{\partial^2 u}{\partial x^2} = 1 \tag{2.63}$$

$$u(\pm 1) = 0 \tag{2.64}$$

This equation is reduced to first-order and entered as a linear boundary value problem as

```
problem = de.LBVP(domain, variables=['u','ux'])
problem.meta[:]['x']['dirichlet'] = True
problem.add_equation("dx(ux) = 1")
problem.add_equation("dx(u) - ux = 0")
problem.add_bc("left(u) = 0")
problem.add_bc("right(u) = 0")
```

As the figure shows, the combination of writing equations as first-order systems, mapping derivatives to Chebyshev-U polynomials, performing Dirichlet preconditioning, and grouping Chebyshev modes before variables produces a highly sparse and banded matrix for the equation. Along with limiting the bandwidth of NCC expansions, this strategy generally produces such matrices for broad ranges of systems of differential equations.

The pencil matrices are all stored as Scipy sparse matrices. The matrices produced for each of the LHS expression groups ($\mathcal{M}$, $\mathcal{L}$, and $\mathcal{F}_\mathcal{X}$) are expanded to occupy the union of their sparsity patterns, so that they can be efficiently added to each other by directly adding their entry values.

2.7.2 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 $\mathcal{L}$. Here P_p^L and P_R explicitly indicate that the expression matrices have been preconditioned from the left and the right, and the p subscripts indicate that the left preconditioner and expression matrices depend vary by pencil. The solver then constructs system handlers for evaluated the RHS equation and boundary condition expressions $\mathcal{F}$.

The solver class contains a `solve` method, which first evaluates the RHS handlers 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

$$L_p X_p = F_p \tag{2.65}$$

The equivalent preconditioned system is given by

$$\underbrace{P_p^L L_p P^R}_{\tilde{L}_p} \underbrace{(P^R)^{-1} X_p}_{\tilde{X}_p} = \underbrace{P_p^L F_p}_{\tilde{F}_p} \tag{2.66}$$

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

- The RHS vector F_p is constructed by taking the pencil data from the RHS handlers, and replacing the proper rows in the equation vector with the boundary condition values.
- The pencil's left-preconditioner is applied to F_P to produce $\tilde{F}_p$.
- Since $\tilde{L}_p$ is sparse and banded, it can be efficiently solved against $\tilde{F}_p$ to produce $\tilde{X}_p$. This is done using the sparse direct solver from the SuperLU library, wrapped through the `scipy.sparse` package.
- The state-vector pencil is recovered from $\tilde{X}_p$ by reapplying the right-preconditioner as

  $$P^R \tilde{X}_p = P^R (P^R)^{-1} X_p = X_p \tag{2.67}$$

  and the result is assigned to the state-vector field system.

After the RHS field system 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 $O(N_z^c)$ 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 effectively reverses the implicit preconditioning of the unkowns.

2.7.3 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

$$L_p X_p = F(X)_p \tag{2.68}$$

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:

$$L_p X_p^{n+1} = F(X^{n+1})_p \tag{2.69}$$

$$L_p(X_p^n + \delta X_p^n) = F(X^n + \delta X^n)_p \tag{2.70}$$

$$\approx F(X)_p + F_X \delta X_p \tag{2.71}$$

$$\implies \underbrace{(L_p - F_X)}_{A_p} \delta X_p \approx \underbrace{F(X)_p - L_p X_p}_{B_p} \tag{2.72}$$

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:

- The RHS vector B_p is constructed by combining the pencil data from the RHS handlers and inserting the boundary condition values.
- The left-preconditioner is applied to produce $\tilde{B}_p$.
- The LHS matrices are combined to produce $\tilde{A}_p$, which is solved against $\tilde{B}_p$ using the SuperLU sparse direct solver to produce $\delta\tilde{X}_p$.
- The right-preconditioner is applied to recover δX_p.
- 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 sufficient good starting positions. The initial conditions are set by modifying the fields in `solver.state` field system.

2.7.4 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

$$\sigma M_p X_p + L_p X_p = 0 \tag{2.73}$$

Dense solver

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

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:

$$(\tilde{L}_p + \sigma_T \tilde{M}_p)^{-1} \tilde{M}_p \tilde{X}_p = -\frac{1}{(\sigma - \sigma_T)} \tilde{X}_p = \lambda_p \tilde{X}_p \tag{2.74}$$

A Scipy linear operator is constructed to represent the left-side operator. The operator 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 solve is done by precomputing the LU decomposition of the matrix using the SuperLU. 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 us to use sparse regular eigenvalue solvers for our generalized eigenvalue problem, with the requirement that $(\tilde{L}_p + \sigma_T \tilde{M}_p)$ is full rank.

2.7.5 Initial value solver

The initial value solver is instantiated from the initial value problem. It is instantiated with one of the timestepping 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 handlers for evaluating the RHS expressions $\mathcal{F}$.

Conceptually, the discretized problem takes the form

$$M_p \partial_t X_p + L_p X_p = F(X, t)_p \tag{2.75}$$

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 *temporally algebraic* or *diagnostic 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 timestepping loop is written by the user, allowing for detailed control of and interaction with the model as timestepping progresses.

Initial conditions

Before beginning a simulation, the initial conditions of the solver's state system must be set. 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. for a simulation involving a third Chebyshev derivative:

```
# Reference local grid and state fields
x = domain.grid(0)
u = solver.state['u']
ux = solver.state['ux']
uxx = solver.state['uxx']
# Setup smooth triangle with support in (-1, 1)
n = 20
u['g'] = np.log(1 + np.cosh(n)**2/np.cosh(n*x)**2) / (2*n)
u.differentiate('x', out=ux)
ux.differentiate('x', out=uxx)
```

When possible, it is best to begin a simulation with *consistent* initial conditions that satisfy the algebraic equations and boundary conditions in the problem. Initial conditions that are inconsistent may introduce persistent errors or stability problems with some timestepping schemes.

Initial conditions can also easily be loaded from the analysis files produced by Dedalus (§2.8) via the `solver.load_state` method. This is particularly useful for restarting simulations from a checkpoint saved by a previous simulation.

Time evolution

The `step` method of the initial value solver advances the state by one timestep, producing X^{n+1} from X^n, where the superscripts denote the temporal iteration of the state vector. The method accepts the timestep `dt` as an argument, along with an keyword flag `trim`. This flag will examine the handlers attached to the problem evaluator that are triggered to evaluate on a given cadence in simulation time, and trim the provided timestep in order to exactly hit the nearest multiple of these cadences, if it would be passed by taking the full timestep. The method then gathers the solve state system, calls the specified integration routine's `step` method to update the system, scatters the updated state system, and updates the solver's iteration count.

In general, the integration step will evaluate the RHS handlers to perform the temporal integration, and will simultaneously evaluate any other scheduled handlers attached the solver evaluator.

Timestep determination

All of the implemented timestepping schemes accommodate changing the timestep between iterations during a simulation. The `CFL` class in the `dedalus.extras.flow_tools` module can help determine what timestep might adequately resolve physical timescales in the evolving solution. The `add_frequencies` and `add_velocities` methods allows 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. Internally, the CFL class builds an auxiliary handler to evaluate these frequencies and velocities at a specified cadence, and as the simulation runs, the suggested timestep is determined via the `compute_dt` method as follows:

- At each point on the grid, all of the specified frequencies are added.

- The maximum total frequency from the entire grid is taken and inverted to determine the CFL timestep.
- This timestep is then multiplied by a *safety factor*, specified at the CFL instantiation with the `safety` keyword. Empirically, we have often found that setting this factor between 0.1 and 0.5 helps to maintain stability.
- The resulting timestep is then bounded to lie above an absolute minimum level and a minimum fraction of the previous timestep, specified by with the `min_dt` and `min_change` keywords, and below an absolute maximum level and a maximum fraction of the previous timestep, specified with the `max_dt` and `max_change` keywords.
- If the fractional change from the previously computed timestep to the newly determined timestep is sufficiently small, as determined by the `threshold` parameter, the previously computed timestep is returned. Otherwise, the newly determined timestep is returned.

The absolute minimum and maximum can be useful to prevent the timestep from grinding to a halt due to a spurious feature of the solution, or vastly overstepping relevant dynamics. The bounds on the relative change in the timestep help prevent ill-conditioning that may occur for some timestepping schemes when the timestep varies too suddenly. The thresholding option allows the timestep to be frequently determined, but avoids modifying the timestep by inconsequential amounts. This can have significant performance advantages, since the timestepping algorithms compute matrix factorizations depending on the timestep. Computing these factorizations is typically much slower than using them to perform an integration, so when the timestep remains the same for sequential iterations, the stored factorizations can be reused to dramatically increase the overall timestepping performance.

Termination

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

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

- 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 HPC job submissions have been reached, allowing for clean termination and potential post-processing of the data before a job is terminated by the system.

With these features, a typical timestepping loop that will advance in a Dedalus script may take the form:

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

This will continue timestepping until any of the specified stopping criteria are reached, adjusting the timestep along the way via the CFL handler.

2.7.6 Timesteppers

Rather than implementing a single specific timestepping 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, while avoiding nonlinear/iterative algorithms for integrating the nonlinear terms.

Multistep IMEX integrators

A general multistep IMEX scheme with s steps temporally discretizes systems of the form of Eq. (2.75) into the general form

$$\sum_{j=0}^{s} a_j M_p X_p^{n-j} + \sum_{j=0}^{s} b_j L_p X_p^{n-j} = \sum_{j=1}^{s} c_j F_p^{n-j} \tag{2.76}$$

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

$$\underbrace{(a_0 M_p + b_0 L_p)}_{A_p^n} X_p^n = \underbrace{\sum_{j=1}^{s} c_j F_p^{n-j} - a_j M_p X_p^{n-j} - b_j L_p X_p^{n-j}}_{B_p^n} \tag{2.77}$$

The `MultistepIMEX` class implements this generalized structure using the preconditioned matrices. The class uses double-ended queues to store `CoeffSystems` containing $\tilde{M}\tilde{X}$, $\tilde{L}\tilde{X}$, $\tilde{F}$, and $\mathrm{d}t$ for the most recent s steps. The class contains a `step` method, called with the latest timestep $\mathrm{d}t^n$, which evaluates the algorithm to produce X^n in the following steps:

- The timestep queue is rotated, the newest value replaces the oldest.
- The scheme coefficients a_j, b_j, c_j are evaluated using the timestep queue via the `compute_coefficients` method, which must be implemented by each subclass to define a specific multistep scheme.
- The $\tilde{M}\tilde{X}^{n-1}$ and $\tilde{L}\tilde{X}^{n-1}$ data for all local pencils are evaluated and stored in the respective coefficient systems. These can be evaluated pencil-by-pencil using the state vector data without building the dense inverse of the right preconditioner simply as e.g.
$$\tilde{L}_p \tilde{X}_p = P_p^L L_p P^R (P^R)^{-1} X_p = P_p^L L_p X_p \tag{2.78}$$
- The RHS handler is evaluated and the equation and boundary data for each pencil is combined, left-preconditioned, and stored in the $\tilde{F}^{n-1}$ coefficient system.
- The data from the $\tilde{M}\tilde{X}$, $\tilde{L}\tilde{X}$, and $\tilde{F}$ coefficient systems are combined using the scheme coefficients to produce the $\tilde{B}^n$ data.
- For each pencil, $\tilde{M}_p$ and $\tilde{L}_p$ are combined to produce $\tilde{A}_p^n$, which is solved against $\tilde{B}_p^n$ to produce $\tilde{X}_p^n$. This solve is done using the SuperLU sparse direct solver wrapped in Scipy. Optionally, the LU decomposition of each $\tilde{A}_p^n$ can be stored and reused to reduce the solve time if the coefficients a_0 and b_0 remain unchanged from the previous iteration.
- The right-preconditioner is applied to recover X_p^n, which is assigned to the state-vector field system.

Specific multistep schemes are implemented by subclassing the `MultistepIMEX` base class and implementing the `compute_coefficients` method to produce the scheme coefficients from the previous timesteps. Dedalus currently implements a number of Crank-Nicolson leap-frog, Crank-Nicolson Adams-Bashforth, and semi-implicit BDF methods from Wang et al. (2008), from first to fourth order. An advantage of the multistep methods is that they only require a single evaluation of the RHS per iteration. However, since they depend on previous iterations of the state variables, they cannot be ran at full-order when beginning a simulation. Instead, each scheme iteratively falls back on a lower-order scheme for the first several iterations of a simulation to build up a long enough history. These schemes may also become ill-conditioned if the timestep is varied too abruptly.

IMEX Runge-Kutta integrators

A general Runge-Kutta IMEX scheme temporally discretizes systems of the form of Eq. (2.75) by constructing s stages, indexed by i as

$$M_p X_p^{n,i} - M_p X_p^{n,0} + \mathrm{d}t \sum_{j=0}^{i} H_{i,j} L_p X_p^{n,j} = \mathrm{d}t \sum_{j=0}^{i-1} A_{i,j} F_p^{n,j} \tag{2.79}$$

where $F^{n,j}$ is evaluated at time $t^{n,j} = t^{n,0} + \mathrm{d}t c_j$, $X^{n,0} = X^n$, and $t^{n,0} = t^n$. The H, A, and c tableaus define the particular scheme. This expansion is rearranged to sequentially solve for the $i = 1, ..., s$ stages as

$$\underbrace{(M_p + \mathrm{d}t H_{i,i} L_p)}_{A_p^i} X_p^{n,i} = \underbrace{M_p X_p^{n,0} + \mathrm{d}t \sum_{j=0}^{i-1} A_{i,j} F_p^{n,j} - \sum_{j=0}^{i-1} H_{i,j} L_p X_p^{n,j}}_{B_p^i} \tag{2.80}$$

Our implemented methods use the final stage as the advanced solution, i.e. $X^{n+1} = X^{n,s}$ and $t^{n+1} = t^{n,s} = t^n + \mathrm{d}t$.

The `RungeKuttaIMEX` class implements this generalized structure using the preconditioned matrices. The class creates a `CoeffSystem` to store $\tilde{M}\tilde{X}^{n,0}$, and s systems each to store $\tilde{L}\tilde{X}^{n,i}$ and $\tilde{F^{n,i}}$ for all of the stages. The class contains a `step` method, called with the timestep $\mathrm{d}t^{n+1}$, which evaluates the algorithm to produce X^{n+1} in the following steps:

- The $\tilde{M}\tilde{X}^n$ data for all local pencils are evaluated and stored in the respective coefficient

system.

- Then each stage $i = 1, \ldots, s$:

  - The RHS handler is evaluated and the equation and boundary data for each pencil is combined, left-preconditioned, and stored in the $\tilde{F}^{n,i-1}$ coefficient system. For each pencil, $\tilde{L}\tilde{X}^{n,i-1}$ is evaluated and stored in the respective coefficient system.
  - The data from the $\tilde{M}\tilde{X}^{n,0}$, $\tilde{L}\tilde{X}^{n,\cdots}$, and $\tilde{F}^{n,\cdots}$ coefficient systems are combined using the scheme tableaus to produce the $\tilde{B}^i$ data.
  - For each pencil, $\tilde{M}_p$ and $\tilde{L}_p$ are combined to produce $\tilde{A}^i_p$, which is solved against $\tilde{B}^i_p$ to produce $\tilde{X}^{n,i}_p$. This solve is done using the SuperLU sparse direct solver wrapped in Scipy. Optionally, the LU decomposition of each $\tilde{A}^i_p$ can be stored and reused to reduce the solve time if the timestep $\mathrm{d}t$ has remained unchanged from the previous iteration.
  - The right-preconditioner is applied to recover $X^{n,i}_p$, which is assigned to the state-vector field system. The solver simulation time is set to $t^{n,0} + \mathrm{d}t c_i$.

Specific multistep schemes are implemented by subclassing the `RungeKuttaIMEX` base class and implementing the H, A, and c tableaus as array-valued class attributes. Dedalus currently implements a number of first, second, and third-order methods from Ascher et al. (1997) and Sprague et al. (2006). 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 steps at the beginning of a simulation and trivially accommodate adaptive timestepping. 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.

For both the multistep and Runge-Kutta schemes, the implemented base classes make it very straightforward to implement additional timestepping algorithms. This allows users to easily test a variety of schemes and find the best option for their particular problem. The use of coefficient systems in the base class methods makes it efficient to step forward multiple pencils simultaneously, reducing the potential overhead on applications that are not fully scaled out (that is, with potentially many pencils per process).

2.8 Analysis and post-processing

Dedalus includes a framework for evaluating and 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 produced analysis files.

2.8.1 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 saves a particular set of tasks at a particular cadence. The file handler is instantiated with a path for the output directory where the corresponding data should be saved, and the cadence at which handler's tasks should 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 instance, to setup a file handler to be evaluated every few iterations:

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

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

2.8.2 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 instance, creating a task to evaluate the kinetic energy density of a flow might look like:

```
analysis.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:

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

2.8.3 Post-processing

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

1. At the top level is the base folder taking the name that was specified when the handler was constructed, e.g. `./analysis/`.
2. Within the base folder are subfolders for each set of outputs, with the same name plus a set number, e.g. `analysis_s1/`.
3. Within each set subfolder and HDF5 files containing the local data for each process, with the same name plus a process number, e.g. `analysis_s1_p1.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 easily merged into global files for each set using the `merge_process_files` function from the `dedalus.tools.post` module. For some analyis, 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 analysis that simply slices over time, e.g. individually plotting each output of an analysis 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 efficiently 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 output visualizations in parallel are available. Additionally, the HDF5 output file format was chosen because it is widely used in the scientific community,

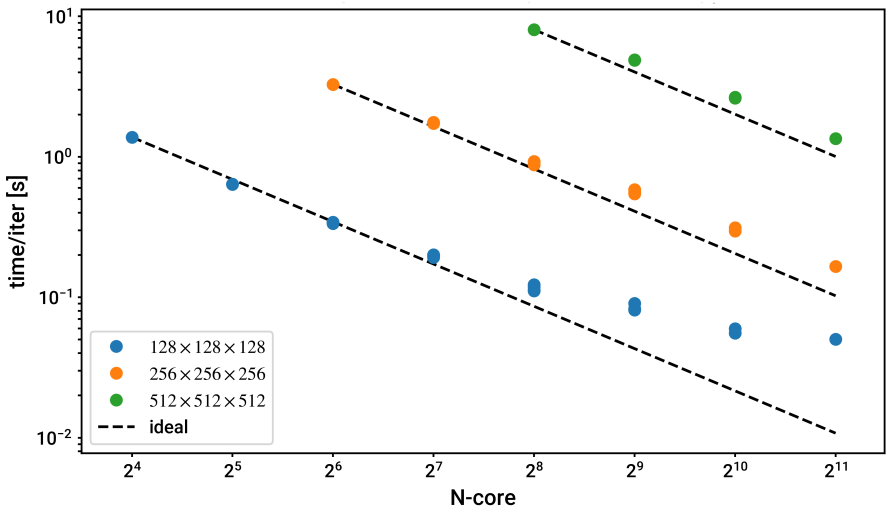

Figure 2.7: Strong scaling results from an incompressible hydrodynamics test problem at several resolutions on the NASA Pleiades supercomputer, with resolution increasing from the lower left to the upper right. Dashed lines indicate the ideal scaling from the lowest-core-count runs. Multiple dots for the same core count correspond to different process mesh shapes. Dedalus typically scales efficiently down to ~ 10 pencils per core.

and allows users to easily examine and visualize simulation outputs using a wide variety of tools and languages.

2.9 Benchmarks

2.9.1 Parallel scaling

A parallel scaling suite[18] for Dedalus has been developed and made publicly available by Benjamin Brown. Fig. 2.7 shows strong-scaling results for an incompressible hydrodynamics simulation on the NASA Pleiades supercomputer[19]. Dedalus shows efficient scaling for 3D problems to thousands of cores. In this test, and in other tests, we observe that the parallel scaling efficiency of Dedalus falls off when there are ~ 10 pencils per core. Further optimizations are planned to improve scaling efficiency, primarily the implementation of hybrid parallelization to reduce message-passing overhead.

[18] https://bitbucket.org/exoweather/incompressible_ns_tg/src/default/
[19] https://www.nas.nasa.gov/hecc/resources/pleiades.html

2.9.2 Kelvin-Helmholtz accuracy benchmark

Lecoanet et al. (2016) performed an accuracy benchmark comparing the finite volume code Athena[20] and Dedalus. Both codes were used to examine the Kelvin-Helmholtz instability in a moderate Mach-number compressible flow. It was found that 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 find that the finite-volume method requires a resolution of 16384^2 cells to avoid these spurious instabilities and achieve the same accuracy as the spectral method at a resolution of 2048^2 modes (Fig. 2.8).

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.

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

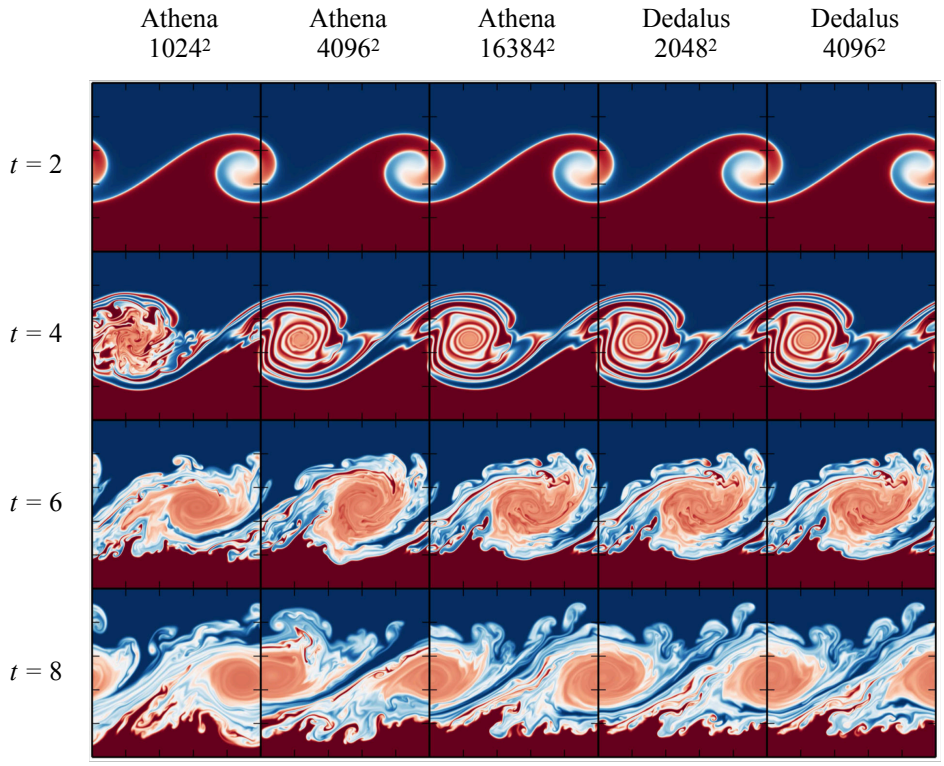

Figure 2.8: Snapshots of a moderate Mach-number Kelvin-Helmholtz instability test problem simulated at various resolutions with a finite volume code (Athena) and Dedalus. The finite volume method introduces small errors which trigger unphysical secondary instabilities in the vortex rolls. These spurious 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 the spectral method with 2048^2 degrees of freedom. Figure adapted from Lecoanet et al. (2016)

Part II

Glacial Meltwater Plumes

Chapter 3

Introduction to melt-driven plumes

3.1 Global ice balances and sea-level rise

Roughly 3×10^7 km^3 of water is stored as ice in glaciers, ice sheets, and ice shelves around the world. This volume of ice has the potential to raise global sea-levels by roughly 65 m if melted (Climate Change, 2009). This volume is dominated by dominated by the ice sheets and ice shelves of Antarctica and Greenland, which could contribute 58 m and 7 m to sea-level rise, respectively, and could potentially further alter the climate by modifying large-scale ocean currents. The melting of the remaining global stores of ice, primarily mountain glaciers and permafrost, could have substantial local impacts, but could only potentially contribute 0.5 m to sea-level rise. Understanding the dynamics and stability of the ice sheets and ice shelves in Antarctica and Greenland is therefore key to predicting the future sea level rise in response to a changing climate on Earth.

The basic hydrological cycle in Greenland and Antarctica begins with inland precipitation. Snowfall and freezing rain compact to form ice sheets $1 - 3$ km thick. These sheets flow downhill towards the coasts as viscous gravity currents. On large scales, ice typically flows as a shear-thinning fluid, meaning that the viscous resistance to flow decreases as the ice flows faster. This property leads to the formation of fast-moving ice streams as the ice sheets flow down through valleys nearing the coastlines. Melting may occur at the surface of the ice sheets once they reach low latitudes, creating water that runs off the surface and drains to the bed of the ice sheet. Upon reaching the coast, the ice sheets can spread to form floating ice shelves, or terminate as tidewater glaciers with nearly vertical faces. At this point, the glaciers lose mass to the ocean via calving (the mechanical breakup of ice into the ocean) and submarine melting.

The net mass trends of the Greenland and Antarctic ice sheets are determined by

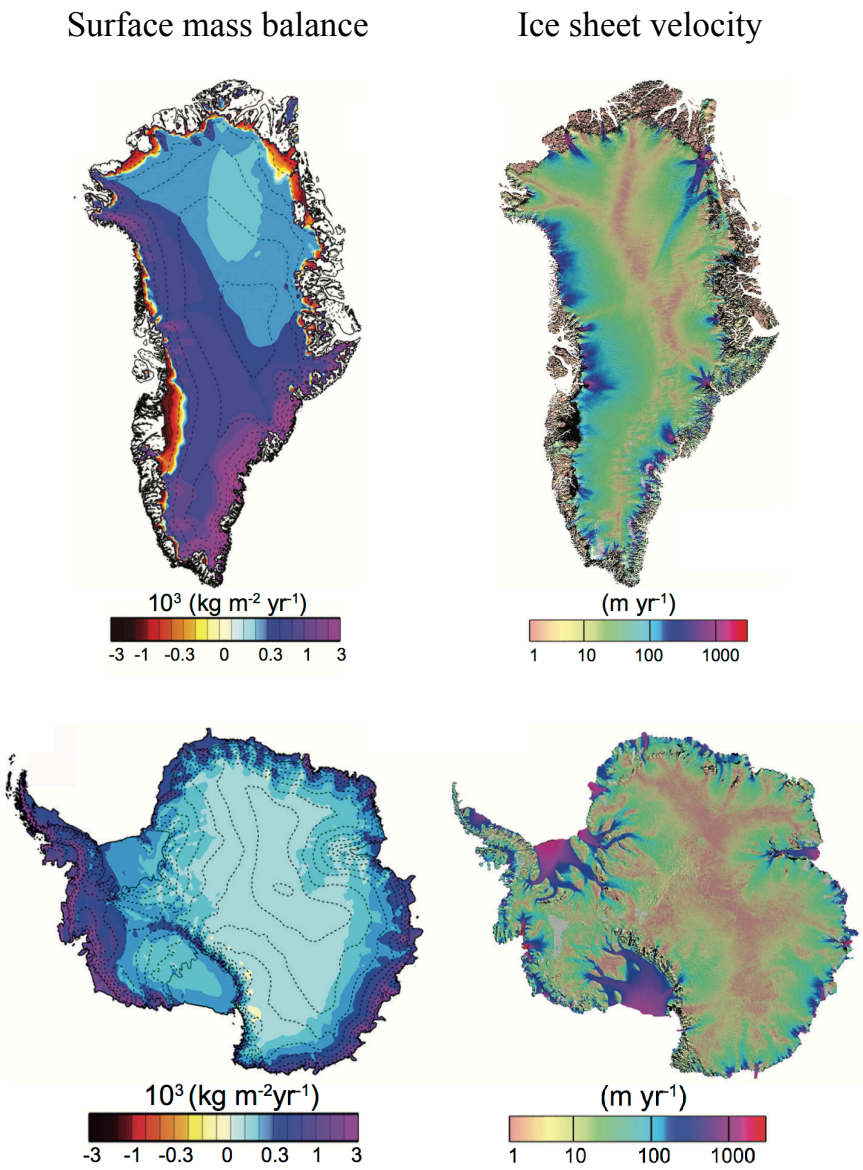

Figure 3.1: Left: Mean ice sheet surface mass balance (precipitation minus surface melt) in Greenland and Antarctica from 1989-2004 from regional climate modeling. Right: Ice sheet surface velocities in Greenland and Antarctica for 2007-2009 measured using satellite data. Figures adapted from Climate Change (2009). Greenland and Antarctica not shown to scale.

the balances of accumulation through precipitation and mass-loss due to surface melting, submarine melting, and calving. Fig. 3.1 shows measurements of the the surface mass balance (precipitation minus surface melting) and surface velocities of the Greenland and Antarctic ice sheets from the 2013 IPCC report. In Antarctica, the largest contributor to mass-loss is submarine melting under large, slow-moving ice shelves (Dinniman et al., 2016). In Greenland, the largest contributor to mass-loss is calving from the front of the vertical fronts of fast-flowing glaciers entering fjords (Straneo et al., 2015). Observations indicate current net mass loss rates of $\sim$ 150 Gt/yr from Antarctica and $\sim$ 215 Gt/yr from Greenland, combining to cause $\sim$ 1 mm/yr in global sea-level rise (Climate Change, 2009). These rates are increasingly rapidly, having more than doubled in the last decade.

3.2 Models of submarine melting

The rapid increase in net ice loss from Greenland and Antarctica has motivated the investigation of the physical processes determining mass loss rates and their sensitivity to changes in global climate. Medium and large-scale models of Greenland's glacial fjords and the Antarctic seas have been used to begin examining the interplay between changes in ocean circulations and submarine melting (Kimura et al., 2014; Sciascia et al., 2013). However, to capture such large-scale dynamics, these studies are forced to use subgrid closures to model the melting dynamics and flows at the ice-ocean interface.

Independent studies are required to develop and test effective subgrid models for the ice-ocean boundary layer for the broad ranges of conditions present in Greenland and Antarctica. Turbulent transfer closures have been calibrated against laboratory experiments (McConnochie et al., 2017) and observed melt rates (Jenkins et al., 2010) in the presence of imposed flows. These closures have been used effectively in models of sub-shelf circulations (Dinniman et al., 2016) and to parameterize melting from the subglacial discharge of surface meltwater (Jenkins, 2011, 2016; D A Slater et al., 2015; Donald A Slater et al., 2016).

When external flows are weak, however, the buoyancy of the melt from the face of the glacier drives a turbulent plume up along the ice-ocean interface. The statistics of such plumes and their impact on melt rates are uncertain since the buoyancy of the plume is directly tied to the molecular processes at the interface and may be strongly impacted by the ambient environment (Magorrian et al., 2016; McConnochie et al., 2016a). A number of studies have begun examining the melt rates in these plumes using turbulent plume theory, simulations, and experiments (Gayen et al., 2015, 2016; Kerr et al., 2015; McConnochie

et al., 2016b; Wells et al., 2008). Additional local models capable of resolving the turbulent dynamics at the ice-ocean interface may assist in the development and testing of more accurate closures for submarine melting in this regimes.

3.3 Governing equations for flow near melting boundaries

3.3.1 Interior equations

Away from the boundary, the appropriate equations for the water are the Navier-Stokes equations under the Boussinesq approximation. These equations describe the conservation of mass, momentum, heat, and salt in low-Mach number flows near constant density:

$$\nabla \cdot \mathbf{u} = 0 \tag{3.1}$$

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla p + \frac{\rho - \rho_a}{\rho_l} g \mathbf{e}_g + \nabla \cdot \nu (\nabla \mathbf{u} + \nabla \mathbf{u}^T) \tag{3.2}$$

$$\frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T = \nabla \cdot \kappa \nabla T \tag{3.3}$$

$$\frac{\partial S}{\partial t} + \mathbf{u} \cdot \nabla S = \nabla \cdot D \nabla S \tag{3.4}$$

for a hydrostatically balanced ambient state ρ_a, reduced pressure perturbation p, and reference liquid density ρ_l. For the ocean, the density follows the equations of state of seawater

$$\rho = \rho(T, S, d) \tag{3.5}$$

which shows appreciable nonlinearity over the range $-5^\circ < T < 35^\circ$, 0 psu $< S <$ 40 psu and 0 km $< d <$ 1 km. However, for small departures from the ambient temperature and salinity profiles, we may linearize the equation of state as

$$\rho - \rho_a \approx \rho_l(-\alpha(T - T_a) + \beta(S - S_a)) \tag{3.6}$$

With this approximation, and taking the viscosity ν and thermal and salt diffusivities κ and D to be constant, the equations simplify to

$$\nabla \cdot \mathbf{u} = 0 \tag{3.7}$$

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla p + (-\alpha(T - T_a) + \beta(S - S_a))g\mathbf{e}_g + \nu \nabla^2 \mathbf{u} \tag{3.8}$$

$$\frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T = \kappa \nabla^2 T \tag{3.9}$$

$$\frac{\partial S}{\partial t} + \mathbf{u} \cdot \nabla S = D \nabla^2 S \tag{3.10}$$

Further defining the thermal and salinity buoyancy anomalies as

$$b^T = \alpha g(T - T_a) \tag{3.11}$$

$$b^S = -\beta g(S - S_a) \tag{3.12}$$

gives us

$$\nabla \cdot \mathbf{u} = 0 \tag{3.13}$$

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla p - (b^T + b^S)\mathbf{e}_g + \nu \nabla^2 \mathbf{u} \tag{3.14}$$

$$\frac{\partial b^T}{\partial t} + \mathbf{u} \cdot \nabla b^T + \mathbf{u} \cdot \nabla \alpha g T_a = \kappa \nabla^2 b^T + \kappa \nabla^2 \alpha g T_a \tag{3.15}$$

$$\frac{\partial b^S}{\partial t} + \mathbf{u} \cdot \nabla b^S - \mathbf{u} \cdot \nabla \beta g S_a = D \nabla^2 b^S - D \nabla^2 \beta g S_a \tag{3.16}$$

3.3.2 Boundary conditions

At the ice-ocean interface, the boundary conditions are given by the conservation of mass, energy, and salt across the interface, and the constraint that the interface lies on the liquidus relation for ice and saltwater.

Consider an interface initially at a position $x = 0$, with positive values of x extending into the ocean, and negative values of x into the ice. In the following, subscripts l, s, and i will denote properties at the edge of the liquid water region, edge of the solid ice region, and at the interface, respectively.

Mass conservation across the melting interface is given by

$$\rho_s U_s = \rho_l U_l \tag{3.17}$$

with reference liquid density ρ_l, meltwater velocity U_l, reference solid density ρ_s, and solid velocity relative to interface U_s. U_l and U_s are both taken to be positive for melting.

Conservation of energy across the melting interface is given by

$$q_l = q_s + \rho_s L U_s \tag{3.18}$$

where q_l is the heat flux out of the liquid ($q = \rho c \kappa \partial_x T$ where c is the specific heat capacity), q_s is the heat flux into the solid, and L is the latent heat of fusion. The thermal profile in the ice can be solved with a simple boundary-layer approximation to give

$$q_s = \rho_s c_s (T_i - T_s) U_s \tag{3.19}$$

where T_s is the far-field temperature in the solid. The thermal balance can then be rewritten as

$$c_l \kappa_l \partial_x T_l = \left[c_s (T_i - T_s) + L \right] U_l \tag{3.20}$$

Conservation of salinity across the melting interface is given by

$$r_l = \rho_l S_i U_l - \rho_s S_s U_s = \rho_l U_l (S_i - S_s) \tag{3.21}$$

where r_l is the salt flux out of the liquid ($r = \rho D \partial_x S$), S_i is the interface salinity, and S_s is the solid salinity.

Finally, the interface is required to lie on the liquidus relation for ice and melt-water. We approximate the liquidus with a linearization in salinity and depth (as a proxy for pressure) as

$$T_i = T_L(S_i, d) \approx T_m - \lambda d - \Gamma (S_i - S_s) \tag{3.22}$$

where T_m is the reference freezing point, λ is the freezing point depth dependence, d is the depth from the surface, and Γ is the freezing point salt slope.

Over the range $0 < d < 1000$ m and $0 < S_i < 40$ psu, the density ratio of ice to water at the freezing point varies only slightly with $0.88 < \rho_s / \rho_l < 0.92$, so can be well-approximated as a constant. Over a similar range, the ratio of the latent heat of melting at the heat capacity of ice near the freezing point is $158 \text{ K} < L/c_s < 159 \text{ K}$, so the heat flux into the solid can be neglected in the thermal balance.

Even with these simplifications, the boundary conditions remain a nonlinear system coupling the interface values of T and S and their wall-normal derivatives. To make analytical progress and build intuition for the behavior of this system, we examined a simplified problem which may be seen as a limited analog of the full melting dynamics.

Chapter 4

Convection from a heated sidewall in a thermally stratified fluid

This work was supervised by Glenn Flierl and Andrew Wells.

4.1 Introduction

The classic paradigm of buoyancy-driven convection is Rayleigh-Benard convection, where the top and bottom boundaries force an unsteady vertical buoyancy gradient within a fluid. In many environmental and industrial processes, however, buoyancy may be provided via an inclined or vertical boundary. A wide range of studies have examined laminar and turbulent convection in a differentially heated vertical slot, a close analogy to the Rayleigh-Benard problem, but rotated to the vertical (Bergholz, 1978). For large forcings in these geometries, the turbulence fills the entire width of the slot (Cimarelli et al., 2017).

Semi-infinite domains bounded by a single wall serve as a better model for geophysical flows, such as meltwater plumes at the ice-ocean interface of marine-terminating glaciers. In an unstratified fluid, the flow driven by a lateral boundary becomes wider and more turbulent with height (Armfield et al., 2007; Wells et al., 2008). An ambient stratification, as is often present in geophysical scenarios, however, may alter the behavior of turbulent plume with height, since motion against the stratification modifies the energy of the flow.

As a first approximation to the geophysical problem of melt-driven convection in the presence of an ambient stratification, we consider the simple case of convection driven by wall held at a fixed temperature anomaly against a semi-infinite fluid with a stable thermal stratification. We further consider a local portion of the fluid near the forced boundary, and assume the flow to be periodic along the face of the boundary, removing end effects near the bottom or top of the wall. With this geometry, the problem becomes homogeneous along the boundary and admits a steady 1D solution consisting of oscillating buoyancy and

upflow layers parallel to the wall which decay with distance from the wall. The stability of this solution in 2D for a vertical wall was thoroughly studied by Gill et al. (1969).

Fedorovich et al. (2009) studied the turbulent saturation of a similar flow, where a buoyancy flux rather than a buoyancy anomaly is imposed at a vertical boundary in a stratified fluid. Here we explore the turbulent regime of the problem with a fixed buoyancy anomaly using direct numerical simulations, including walls inclined away from the vertical. In particular, we seek to determine how the heat flux from the wall scales with the thermal driving and fluid parameters in the turbulent regime. First, we introduce the model and summarize its laminar solutions and their stability. We then perform a series of two-dimensional direct numerical simulations of the governing equations to examine the transition to turbulence and the statistics of the turbulent state. Finally, we comment on the applicability of this work as a simplified model for melt-driven plumes in fresh water, and quantify the corresponding melt rate estimates.

4.2 Model definition

4.2.1 Governing equations

We consider a Boussinesq fluid with a single buoyancy component, constant kinematic viscosity μ, and constant buoyancy diffusivity κ. The total buoyancy field is decomposed into a steady ambient/background buoyancy and a buoyancy anomaly as $b = b_0 + b_1$. We consider stable linear stratifications for the background buoyancy, written in terms of the buoyancy frequency as $\nabla b_0 = -N^2 \mathbf{e}_g$. In terms of the buoyancy and pressure anomalies, the governing equations are then

$$\nabla \cdot \mathbf{u} = 0 \tag{4.1}$$

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla p_1 + \nu \nabla^2 \mathbf{u} - b_1 \mathbf{e}_g \tag{4.2}$$

$$\frac{\partial b_1}{\partial t} + \mathbf{u} \cdot \nabla b_1 = N^2 \mathbf{u} \cdot \mathbf{e}_g + \kappa \nabla^2 b_1 \tag{4.3}$$

We consider a semi-infinite fluid domain with a fixed and flat boundary on one side. The boundary may be tilted with respect to the vertical by an angle θ, which is taken to be positive for an overhanging tilt. We define a wall-aligned coordinate system, where x is the wall-normal coordinate, with $x = 0$ located at the wall, y extending horizontally across the wall, and z extending up the wall (Fig. 4.1). Then we have $\mathbf{e}_g = \sin\theta\, \mathbf{e}_x - \cos\theta\, \mathbf{e}_z$ and the

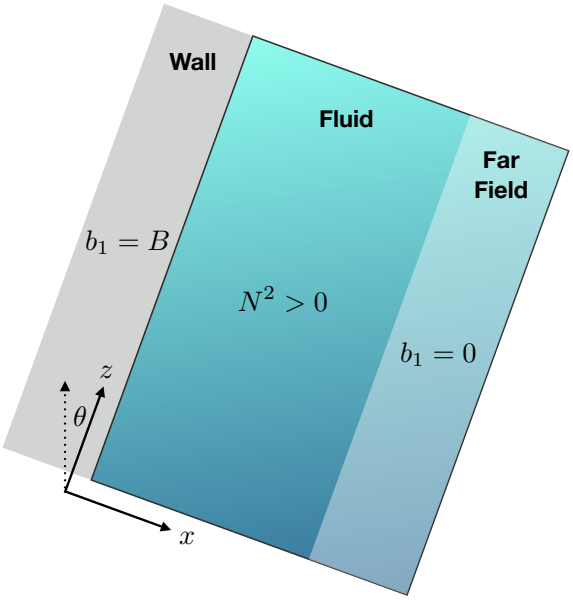

Figure 4.1: We consider a linearly stratified fluid bounded by a tilted wall with an applied buoyancy perturbation. The laminar solution is solved assuming a semi-infinite domain. The simulations are performed in a confined domain with a wall in the far field.

component-wise equations become

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0 \tag{4.4}$$

$$\frac{\partial u}{\partial t} + \mathbf{u} \cdot \nabla u = -\frac{\partial p_1}{\partial x} + \nu \nabla^2 u - \sin\theta \; b_1 \tag{4.5}$$

$$\frac{\partial v}{\partial t} + \mathbf{u} \cdot \nabla v = -\frac{\partial p_1}{\partial y} + \nu \nabla^2 v \tag{4.6}$$

$$\frac{\partial w}{\partial t} + \mathbf{u} \cdot \nabla w = -\frac{\partial p_1}{\partial z} + \nu \nabla^2 w + \cos\theta \; b_1 \tag{4.7}$$

$$\frac{\partial b_1}{\partial t} + \mathbf{u} \cdot \nabla b_1 = N^2(\sin\theta \, u - \cos\theta \, w) + \kappa \nabla^2 b_1 \tag{4.8}$$

The buoyancy anomaly is held to a fixed value B at the sidewall, and no slip of the velocity is allowed at the wall:

$$b_1(x = 0) = B \tag{4.9}$$

$$\mathbf{u}(x = 0) = 0 \tag{4.10}$$

When $B > 0$, the buoyancy forcing from the wall drives a flow up along the wall. Depending on the forcing magnitude, this may be a steady laminar flow, or a turbulent plume. By picking the buoyancy forcing to be fixed relative to the background buoyancy, the equations become homogeneous in y and z, and admit solutions that are periodic in these directions, which we will consider in the remainder of this work.

4.2.2 Reynolds decomposition

The turbulent solutions are found to reach statistically steady states which depend only on the distance from the wall, motivating a Reynolds decomposition of the equations based on averaging over y and z. We denote such an average with an overbar, and decompose each field into its average and a perturbation as $f = \bar{f} + f'$. Applying this average to the governing equations and assuming periodicity in y and z yields

$$\frac{\partial \bar{u}}{\partial x} = 0 \tag{4.11}$$

$$\frac{\partial \bar{u}}{\partial t} + \frac{\partial \overline{uu}}{\partial x} = -\frac{\partial \bar{p}_1}{\partial x} + \nu \frac{\partial^2 \bar{u}}{\partial x^2} - \sin\theta\ \bar{b}_1 \tag{4.12}$$

$$\frac{\partial \bar{v}}{\partial t} + \frac{\partial \overline{uv}}{\partial x} = \nu \frac{\partial^2 \bar{v}}{\partial x^2} \tag{4.13}$$

$$\frac{\partial \bar{w}}{\partial t} + \frac{\partial \overline{uw}}{\partial x} = \nu \frac{\partial^2 \bar{w}}{\partial x^2} + \cos\theta\ \bar{b}_1 \tag{4.14}$$

$$\frac{\partial \bar{b}_1}{\partial t} + \frac{\partial \overline{ub_1}}{\partial x} = N^2(\sin\theta\ \bar{u} - \cos\theta\ \bar{w}) + \kappa \frac{\partial^2 \bar{b}_1}{\partial x^2} \tag{4.15}$$

The divergence constraint can be integrated with the velocity boundary condition to give $\bar{u} = 0$.

In statistical equilibrium (i.e. assuming time-independence of the mean quantities), the equations then become

$$\frac{\partial}{\partial x}\left(\overline{u'u'} + \bar{p}_1\right) = -\sin\theta\ \bar{b}_1 \tag{4.16}$$

$$\frac{\partial}{\partial x}\left(\overline{u'v'} - \nu \frac{\partial \bar{v}}{\partial x}\right) = 0 \tag{4.17}$$

$$\frac{\partial}{\partial x}\left(\overline{u'w'} - \nu \frac{\partial \bar{w}}{\partial x}\right) = \cos\theta\ \bar{b}_1 \tag{4.18}$$

$$\frac{\partial}{\partial x}\left(\overline{u'b_1'} - \kappa\frac{\partial \bar{b}_1}{\partial x}\right) = -N^2\cos\theta\,\bar{w} \tag{4.19}$$

In statistical equilibrium, we see that the divergence of the vertical wall-normal momentum transport is balanced by the mean buoyancy, and the divergence of the wall-normal buoyancy transport is balanced by production from the mean flow moving against the background buoyancy gradient.

4.3 Laminar solution and linear stability

The equations admit a steady 1D solution depending only on x which is stable for low values of the sidewall forcing. Such a solution satisfies the steady Reynolds-averaged equations, with the Reynolds transport terms dropped. We denote the components of the steady 1D laminar solution with a tilde, i.e. $\tilde{f} = \bar{f}_{\text{lam}}$:

$$\frac{\partial \tilde{p}_1}{\partial x} = -\sin\theta\ \tilde{b}_1 \tag{4.20}$$

$$\nu\frac{\partial^2 \tilde{v}}{\partial x^2} = 0 \tag{4.21}$$

$$\nu\frac{\partial^2 \tilde{w}}{\partial x^2} = -\cos\theta\,\tilde{b}_1 \tag{4.22}$$

$$\kappa\frac{\partial^2 \tilde{b}_1}{\partial x^2} = N^2\cos\theta\,\tilde{w} \tag{4.23}$$

The first equation can be integrated to give the wall-normal hydrostatic balance. The second equation yields $\tilde{v} = 0$, assuming $\tilde{v}(x \to \infty) = 0$. The third and fourth equations can be combined to give

$$\tilde{w} = \frac{\kappa}{N^2\cos\theta}\frac{\partial^2 \tilde{b}_1}{\partial x^2} \tag{4.24}$$

$$\frac{\partial^4 \tilde{b}_1}{\partial x^4} + \frac{N^2\cos^2\theta}{\kappa\nu}\,\tilde{b}_1 = 0 \tag{4.25}$$

Defining the *laminar boundary layer scale* ℓ as

$$\ell^4 = \frac{4\nu\kappa}{N^2\cos^2\theta} \tag{4.26}$$

and letting $\eta = x/\ell$, the laminar solution (with decay in the far field) is found to be

$$\tilde{b}_1 = Be^{-\eta}\cos(\eta) \tag{4.27}$$

$$\tilde{w} = \frac{B}{N}\left(\frac{\kappa}{\nu}\right)^{1/2} e^{-\eta}\sin(\eta) = We^{-\eta}\sin(\eta) \tag{4.28}$$

The solution consists of out-of-phase, decaying oscillations in mean buoyancy and up-wall velocity. The total up-wall volume and buoyancy fluxes (per unit length in y) of this solution are

$$\int_0^\infty \tilde{w}\mathrm{d}x = \frac{\ell}{2}\frac{B}{N}\left(\frac{\kappa}{\nu}\right)^{1/2} \tag{4.29}$$

$$\int_0^\infty \tilde{b}_1\tilde{w}\mathrm{d}x = \frac{\ell}{8}\frac{B^2}{N}\left(\frac{\kappa}{\nu}\right)^{1/2} \tag{4.30}$$

and the total buoyancy flux out of the wall is

$$F = -\kappa\frac{\partial b}{\partial x}(x=0) \tag{4.31}$$

$$= -\kappa\frac{\partial(b_0+\tilde{b}_1)}{\partial x}(x=0) \tag{4.32}$$

$$= \kappa\left(N^2\sin\theta + B/\ell\right) \tag{4.33}$$

The laminar flow has a characteristic lengthscale ℓ and timescale $W^{-1}\ell$. These can be used to non-dimensionalize the equations, resulting in the constants being expressed as

$$\nu \to \frac{\nu}{W\ell} = \frac{1}{\mathrm{Re}_\ell} \tag{4.34}$$

$$\kappa \to \frac{\kappa}{W\ell} = \frac{1}{\mathrm{PrRe}_\ell} \tag{4.35}$$

$$N^2 \to \frac{N^2\ell^2}{W^2} = \frac{4}{\cos^2\theta}\frac{1}{\mathrm{PrRe}_\ell^2} \tag{4.36}$$

$$B^2 \to \frac{B^2\ell^2}{W^4} = \frac{4}{\cos^2\theta}\frac{1}{\mathrm{Re}_\ell^2} \tag{4.37}$$

where the Prandtl and laminar-solution Reynolds numbers are given by

$$\mathrm{Pr} = \frac{\nu}{\kappa} \tag{4.38}$$

$$\mathrm{Re}_\ell = \frac{W\ell}{\nu} = \left(\frac{4}{\cos^2\theta}\frac{B^4}{N^6\nu\kappa}\frac{\kappa^4}{\nu^4}\right)^{1/4} \tag{4.39}$$

In the turbulent regime, we expect fluid parcels with buoyancy anomalies on the order of B to convect and travel on the potential-rise scale $L = \frac{B}{N^2}$ over a timescale N^{-1}. Non-dimensionalizing with respect to these scales results in the constants being expressed as

$$\nu \to \frac{\nu}{NL^2} = \frac{1}{\mathrm{Re}_L} \tag{4.40}$$

$$\kappa \to \frac{\kappa}{NL^2} = \frac{1}{\mathrm{PrRe}_L} \tag{4.41}$$

$$N \to 1 \tag{4.42}$$

$$B \to 1 \tag{4.43}$$

where the rise-scale Reynolds number is given by

$$\mathrm{Re}_L = \frac{NL^2}{\nu} = \frac{B^2}{N^3\nu} \tag{4.44}$$

We generally choose to non-dimensionalize the forcing using the rise-scale Rayleigh number, given by

$$\mathrm{Ra}_L = \frac{BL^3}{\nu\kappa} = \frac{B^4}{N^6\nu\kappa} = \frac{4}{\cos^2\theta}\left(\frac{L}{\ell}\right)^4 \tag{4.45}$$

which is related to the two Reynolds numbers by

$$\mathrm{Ra}_L = \mathrm{Re}_L^2\mathrm{Pr} = \frac{\cos^2\theta}{4}\mathrm{Re}_\ell^4\mathrm{Pr}^4 \tag{4.46}$$

The equations governing perturbations about the steady laminar solution are

$$\frac{\partial u'}{\partial x} + \frac{\partial v'}{\partial y} + \frac{\partial w'}{\partial z} = 0 \tag{4.47}$$

$$\frac{\partial u'}{\partial t} + \mathbf{u}'\cdot\nabla u' + \tilde{w}\frac{\partial u'}{\partial z} = -\frac{\partial p_1'}{\partial x} + \nu\nabla^2 u' - \sin\theta\ b_1' \tag{4.48}$$

$$\frac{\partial v'}{\partial t} + \mathbf{u}'\cdot\nabla v' + \tilde{w}\frac{\partial v'}{\partial z} = -\frac{\partial p_1'}{\partial y} + \nu\nabla^2 v' \tag{4.49}$$

$$\frac{\partial w'}{\partial t} + u'\frac{\partial \tilde{w}}{\partial x} + \mathbf{u}'\cdot\nabla w' + \tilde{w}\frac{\partial w'}{\partial z} = -\frac{\partial p_1'}{\partial z} + \nu\nabla^2 w' + \cos\theta\ b_1' \tag{4.50}$$

$$\frac{\partial b_1'}{\partial t} + u'\frac{\partial \tilde{b}_1}{\partial x} + \mathbf{u}' \cdot \nabla b_1' + \tilde{w}\frac{\partial b'}{\partial z} = N^2(\sin\theta\, u' - \cos\theta\, w') + \kappa\nabla^2 b_1' \tag{4.51}$$

The linear stability of the background flow can be examined by dropping the terms that are nonlinear in perturbative quantities, substituting the form

$$f'(x, y, z, t) \rightarrow f'(x)e^{i(k_y y + k_z z - \omega t)} \tag{4.52}$$

for each perturbative quantity, and solving the resulting one-dimensional eigenvalue problem for ω and each $f'(x)$. The linear stability of this solution with $\theta = 0$ and $k_y = 0$ was examined in this manner by Gill over a wide range of Prandtl numbers. They found that at a given Prandtl number, the flow is linearly stable up to a critical Reynolds number, at which point a linear convective instability sets in.

By analyzing the energetics during the linear growth phase, Gill was able to classify unstable modes as being primarily mechanically driven, if the perturbation derives most of its energy from the mean flow, or buoyantly driven, if the perturbation derives most of its energy from buoyancy forces. They find that for $\mathrm{Pr} \lesssim 0.72$, the most-unstable mode is mechanically driven, while for $\mathrm{Pr} \gtrsim 0.72$ it is buoyantly driven. We note that due to the stratification, the symmetry between y and z is broken, meaning that Squire's theorem does not apply to this flow, however only perturbations with $k_y = 0$ were considered by Gill.

Fig. 4.2 plots the growth rate $\Im(\omega N)$ of the most unstable mode as function of Ra_L and Lk_z for $\mathrm{Pr} = 1$, $\theta = 0$, and $k_y = 0$. The growth rates were computed using the eigenvalue problem and dense solver in Dedalus. For these parameters, perturbations at all scales are stable until $\mathrm{Ra}_L \approx 10^7$. The figure also indicates the local maxima in the growth rates as a function of k_z for each Rayleigh number. Two branches of maximally unstable modes are present at intermediate Rayleigh numbers – the lower and upper branches consist of the buoyantly and mechanically driven modes described by Gill, respectively. At high Rayleigh numbers, a broad range of modes are mechanically unstable.

4.4 Simulations of unsteady solutions

4.4.1 Computational setup

To examine the solution beyond the linear instability threshold, we directly simulated the full nonlinear equations using the initial value problem and solver in Dedalus. We performed a range of simulations at varying Rayleigh numbers, Prandtl numbers, and overhang angles.

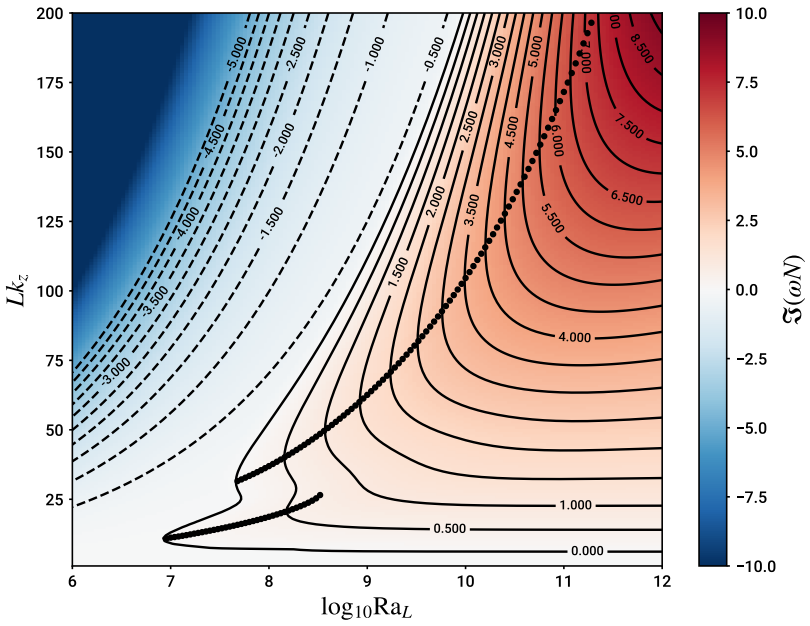

Figure 4.2: Maximum growth rate of perturbations to the laminar solution as a function of Rayleigh number Ra_L and up-wall wavenumber k_z for $\mathrm{Pr} = 1$, $\theta = 0$, and $k_y = 0$. Dots indicate the local maxima in the growth rate as a function of k_z for each Ra_L.

The simulations were done in two dimensions (x and z) for computational efficiency. Rather than simulating a semi-infinite fluid, the computational domain was taken to be a vertically periodic channel, with a Fourier discretization in the z direction and a Chebyshev discretization in the x direction. The buoyancy perturbation was applied to the left boundary at $x = 0$, while the buoyancy perturbation was fixed to zero at the right boundary at $x = L_x$. Both boundaries were taken to be no-slip surfaces.

To prevent the presence of the outer boundary from influencing the flow near the forced boundary, we expect that the domain width should be larger than the potential rise scale, $L_x \gtrsim L$. Simulations were typically performed with $L_x = 2L$, and tests were undertaken at various parameter values to check the independence of the flow statistics on the domain width. For the high Rayleigh numbers, the domain width was decreased to L or $0.5L$ to help ameliorate the increase in resolution needed to resolve turbulence near the forced boundary. In all cases, the simulations were visually inspected to verify that the domain was sufficiently wide that eddies shedding from the convective boundary layer near the forced boundary dissipated before reaching the outer boundary.

The Dirichlet preconditioned option in Dedalus was used, resulting in fully banded matrices representing the linear portion of the equations. Incompressibility is maintained by simultaneously solving the momentum and mass conservation equations for the velocity components and the pressure. The equations were then integrated with the RK222 timestepper, a two-stage second-order mixed implicit-explicit Runge-Kutta integrator. The timestep was determined by the CFL condition set by the advection time across the computational grid. The simulations were started from rest, with small random perturbations in the buoyancy anomaly field b_1. The buoyancy forcing at the wall was initially zero, and turned on as

$$b_1(x = 0, t) = B \tanh(t/\tau) \tag{4.53}$$

where the timescale was typically taken to be $\tau = 10N^{-1}$. This gradual application of the boundary forcing was found to reduce the resolution needed to integrate through the transient turbulent bursts which occur before a steady state is reached at high Rayleigh number. Simulations were typically ran for several hundred buoyancy periods until a statistically steady state was reached. The built-in analysis tools in Dedalus were used to save snapshots, profiles, and scalar statistics from the flow during the integration.

4.4.2 Simulation results

Two primary sets of simulations were performed to study the statistics of the turbulent steady states. In the first, we fixed $\theta = 0$ and performed simulations covering $10^3 \leq \mathrm{Ra}_L \leq 10^{13}$ and $0.1 \leq \mathrm{Pr} \leq 10$. In the second, we fixed Ra_L and Pr and varied the angle of the wall between $0^\circ \leq \theta \leq 30^\circ$.

The primary statistic of interest is the mean buoyancy flux out of the wall in the statistically steady state. This quantity is non-dimensionalized by dividing by the heat flux from the laminar flow for equivalent parameters, producing the non-dimensional Nusselt number:

$$\mathrm{Nu} = \frac{-\kappa\langle\partial_x(b_0 + \bar{b}_1)\rangle}{-\kappa(\partial_x(b_0 + \tilde{b}_1))}(x = 0) \tag{4.54}$$

$$= \frac{N^2 \sin\theta - \langle\partial_x \bar{b}_1\rangle(x = 0)}{N^2 \sin\theta + B/\ell} \tag{4.55}$$

where the brackets indicate a temporal average over the statistically steady state. With high Rayleigh numbers and angles away from the horizontal, the terms from the background are negligible in comparison to the anomaly terms.

Fig. 4.3 shows the computed Nusselt number as a function of Rayleigh number for $\mathrm{Pr} \in \{0.1, 1, 10\}$ with $\theta = 0$. At low Rayleigh numbers, the simulations attain the laminar solution, resulting a baseline Nusselt number of one. As the Rayleigh number is increased, the simulations surpass the linear stability threshold and enter a nonlinear wave regime. As the Rayleigh number is increased further, these nonlinear wave solutions become unstable and a turbulent flow develops, resulting in apparent power-law increases in the Nusselt number.

Fig. 4.4 shows the computed Nusselt number as a function of wall angle for $\mathrm{Pr} = 10$ and $\mathrm{Ra}_L = 10^{11}$. As the wall is tilted away from the vertical, the Nusselt number decreases, but does not appear to follow a simple power law in $\cos\theta$.

4.5 Discussion

4.5.1 Nonlinear wave regime

When the linear stability threshold is slightly exceeded, the linear instability saturates into traveling nonlinear wave solutions moving up the wall. In this regime, the measured Nusselt

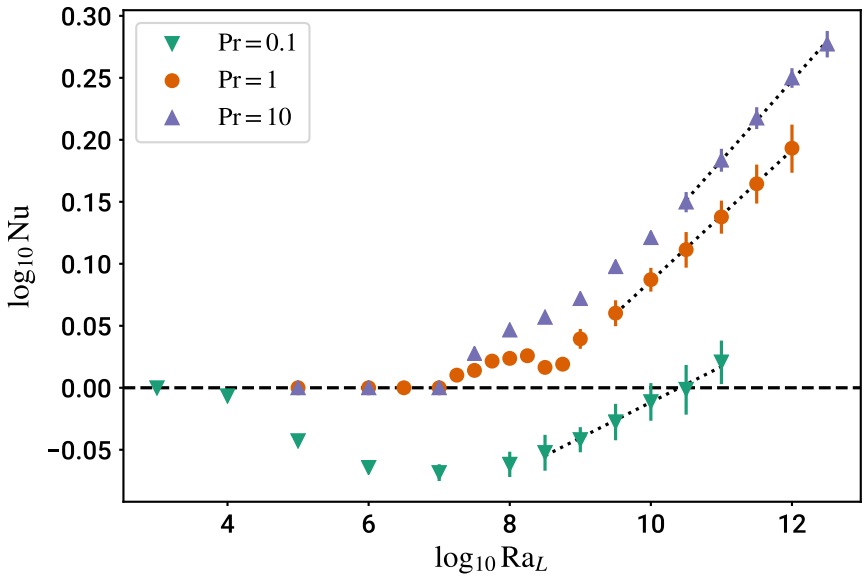

Figure 4.3: Nusselt number as a function of Rayleigh number and Prandtl number with $\theta = 0$. From left to right, the solutions transition from the steady laminar solution with $\mathrm{Nu} = 1$, to traveling nonlinear wave solutions with varying buoyancy flux behavior, to turbulent plume solutions with power-law relations between Nu and Ra_L. The vertical bars through each point indicate the variance of the buoyancy flux in the saturated state. The dashed lines represent power-law fits to the high Re_L simulations.

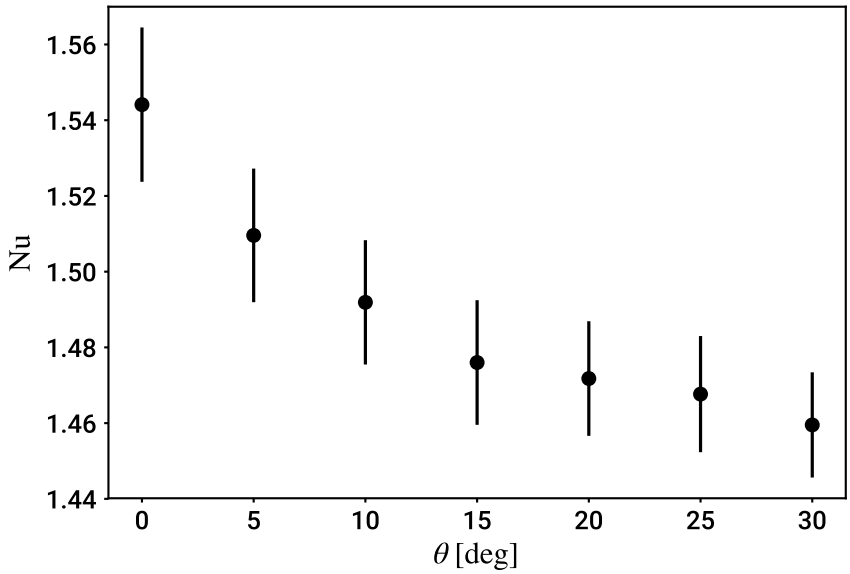

Figure 4.4: Nusselt number as a function of wall overhang angle for Pr = 10 and $\mathrm{Ra}_L = 10^{11}$. The vertical bars through each point indicate the variance of the Nusselt number in the saturated state.

number is marginally sensitive to the along-wall domain size L_z, since the periodicity of the domain in the z direction prevents the emergence of what would be the solution in a infinite domain when L_z is not an exact multiple of that solution's wavelength. In this regime, we typically performed simulations with $L_z = 4$ to capture $\mathcal{O}(10)$ wavelengths of the saturated solutions in an attempt to minimize this frustration, but extensive tests on the impact of L_z on the measured Nusselt number were not performed.

This weakly nonlinear regime was investigated for $\mathrm{Pr} \in \{0.1, 1, 10\}$. For Pr = 10, where the initial instabilities are primarily buoyantly-driven, the Nusselt number monotonically increases with the Rayleigh number in the nonlinear wave regime $10^7 \lesssim \mathrm{Ra}_L \lesssim 10^{10}$. For Pr = 1, the Nusselt number first increases with Rayleigh number above the stability threshold near $\mathrm{Ra}_L \sim 10^7$, then briefly decreases near $\mathrm{Ra}_L \sim 10^{8.5}$ which, from the linear stability theory, we suppose to correspond to the transition from a buoyantly-driven instability to a mechanically-driven instability. Fig. 4.5 shows a snapshot of the traveling nonlinear wave solution for Pr = 1, $\mathrm{Ra}_L = 10^8$, and $\theta = 0$. For Pr = 0.1, the Nusselt number decreases below unity as the Rayleigh number is increased past the stability threshold near $\mathrm{Ra}_L \sim 3 \times 10^3$. The Nusselt number attains a minimum of $\mathrm{Nu} \approx 0.85$ near $\mathrm{Ra}_L \sim 10^7$.

The tendency of the Nusselt number in the nonlinear wave regime thus appears to

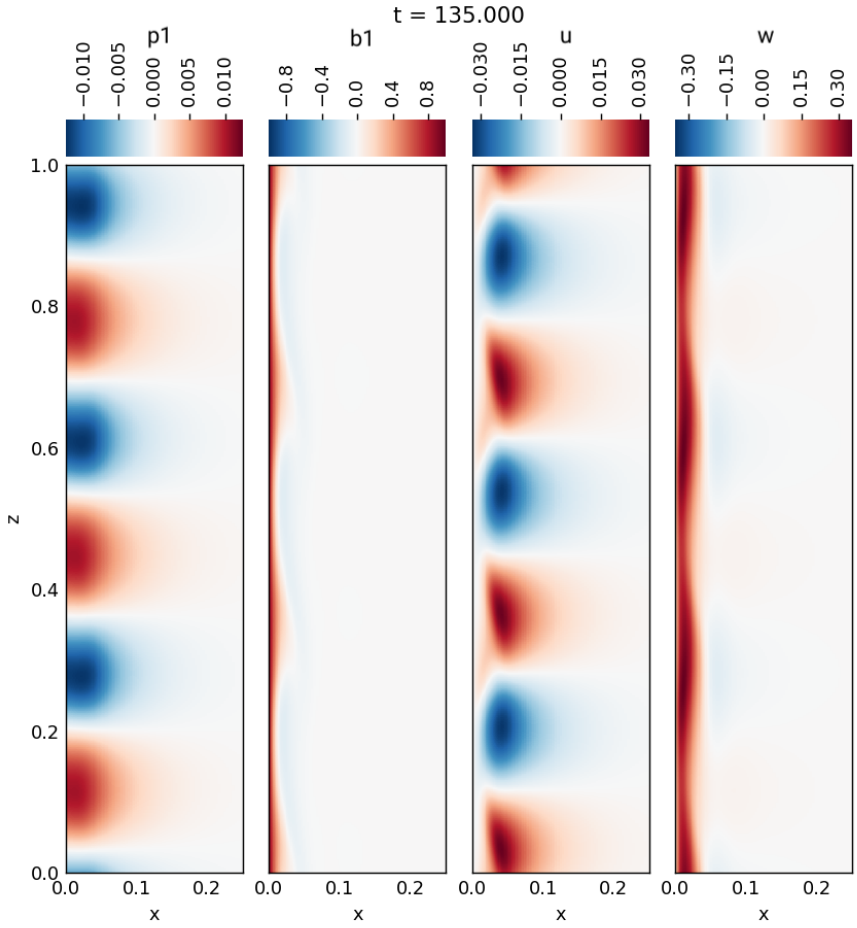

Figure 4.5: Zoom-in snapshots of the traveling nonlinear wave solution for Pr = 1, $\mathrm{Ra}_L = 10^8$, and $\theta = 0$. The simulation quantities are non-dimensionalized using the lengthscale L and timescale N^{-1}.

correspond to the energetic nature of the linear instability at the corresponding parameters, with buoyantly-driven instabilities at high Prandtl number increasing the total buoyancy flux from the wall, and mechanically-driven instabilities at low Prandtl number decreasing this flux.

4.5.2 Turbulent regime

As the Rayleigh number is increased, the nonlinear wave solutions become unsteady and begins to shed vortices into the interior of the domain. As the Rayleigh number is further increased, a well-developed turbulent plume forms along the forced wall as vortices rapidly and repeatedly shed from the boundary layer. To help visualize the evolution of the turbulent plume, the evolution of a passive tracer was integrated alongside the dynamical equations. The tracer field evolved with the same diffusivity and boundary conditions as the buoyancy, but with no background gradient, specifically

$$\frac{\partial c}{\partial t} + \mathbf{u} \cdot \nabla c = \kappa \nabla^2 c \tag{4.56}$$

$$c(x = 0) = B \tag{4.57}$$

The tracer field is thus an effective tool for determining the extent to which fluid parcels in diffusive contact with the forced wall spread throughout the domain.

Fig. 4.6 shows snapshots of the buoyancy anomaly and tracer field in the turbulent state with $\mathrm{Ra}_L = 10^{10}$, $\mathrm{Pr} = 1$, and $\theta = 0$. The turbulent plume, as visualized through the tracer field, is found to remain confined against the wall with a width $\lesssim L$ in all of our simulations. As vortices shed from the turbulent boundary layer, they reach a level of neutral buoyancy, and begin to spread laterally into the interior of the domain, as evidenced by the horizontal filaments in the tracer field extending from the turbulence plume. These vortices additionally excite internal waves which fill the interior domain and are visible in the buoyancy anomaly field. Simulations were performed with different domain widths to ensure the reflection of these internal waves off the far boundary were not substantially influencing the saturated state of the plume.

Fig. 4.7 shows a space-time diagram of the buoyancy flux from the forced wall. Vortices running up the wall are clearly visible as the inclined streaks in the buoyancy flux. The temporal average of the heat flux shows that it is statistically homogeneous along the wall. The spatial average of the heat flux shows that it is statistically steady in time with an

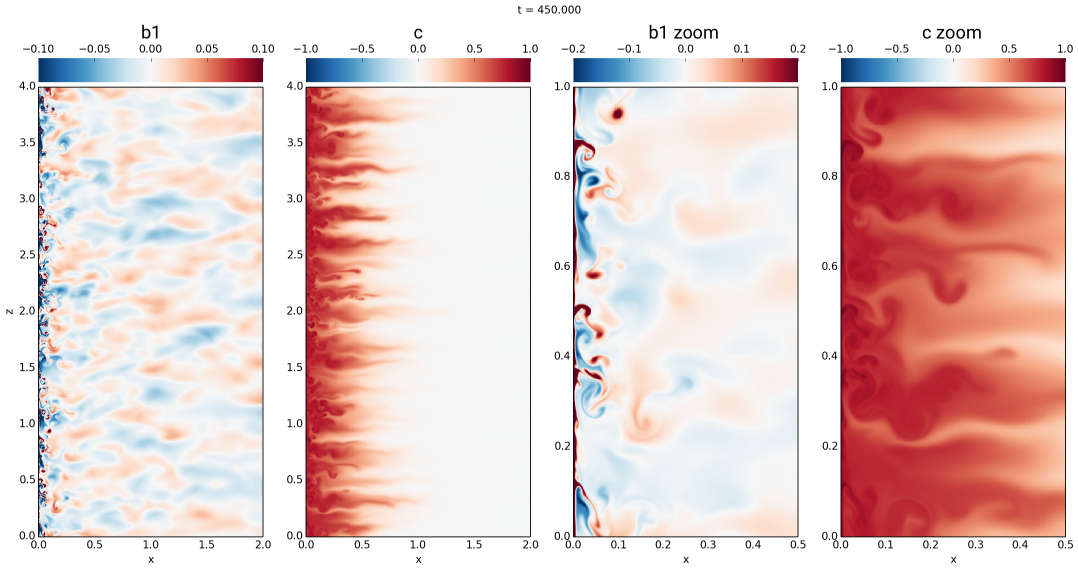

Figure 4.6: Buoyancy anomaly and tracer field in the turbulent state for $\mathrm{Ra}_L = 10^{10}$, $\mathrm{Pr} = 1$, and $\theta = 0$. The left two panels show the entire computational domain, while the right two panels zoom in to show the structure of the vortices shedding from the boundary layer. The simulation quantities are non-dimensionalized using the lengthscale L and timescale N^{-1}.

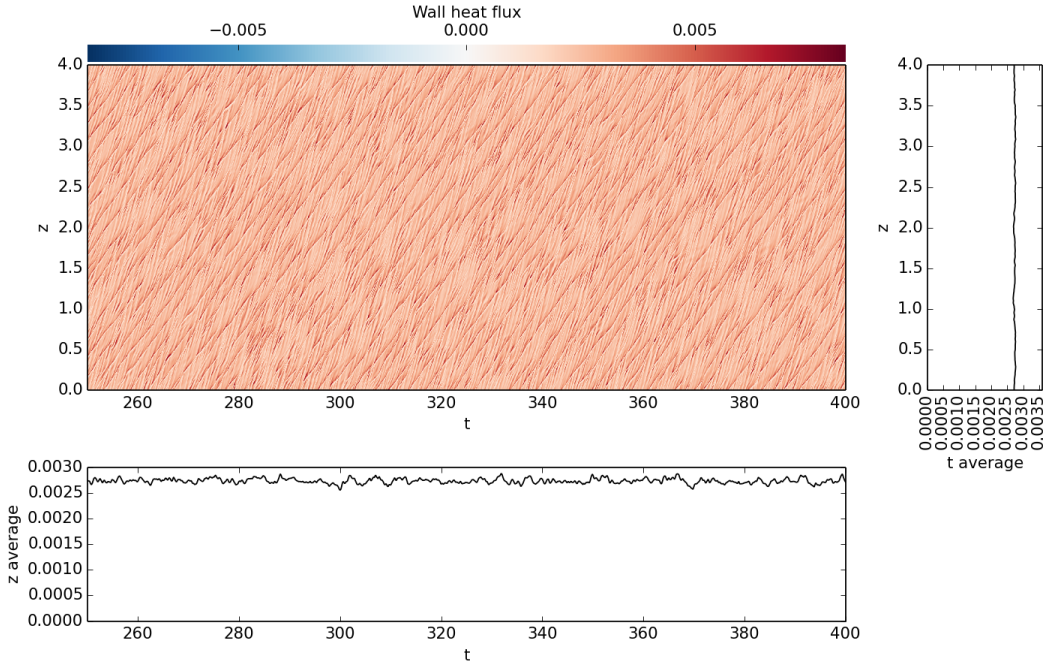

Figure 4.7: Buoyancy flux out of the forced wall as a function of distance along the wall and time in the turbulent state for $\mathrm{Ra}_L = 10^{10}$, $\mathrm{Pr} = 1$, and $\theta = 0$. The right panel shows the temporal average of the wall buoyancy flux as a function of z. The bottom panel shows the spatial average of the wall buoyancy flux as a function of t. The simulation quantities are non-dimensionalized using the lengthscale L and timescale N^{-1}.

Pr	α	β
$10^{-1.0}$	0.501 ± 0.006	34.6 ± 3.2
$10^{-0.5}$	0.515 ± 0.003	28.5 ± 1.2
$10^{0.0}$	0.355 ± 0.007	18.7 ± 0.3
$10^{0.5}$	0.281 ± 0.003	15.6 ± 0.5
$10^{1.0}$	0.308 ± 0.004	15.8 ± 0.7

Table 4.1: Best-fit parameters for power-law fit to the Nusselt number as a function of the Rayleigh number in the form $\mathrm{Nu} = \alpha \mathrm{Ra}_L^{1/\beta}$, with $\theta = 0$ and $\mathrm{Re}_L \geq 10^{4.75}$.

apparent variance, but no secular evolution. The temporal and along-wall homogeneity of all simulations in the turbulent state motivates the use of the z and t averaged statistics in characterizing the flow. The temporal average and variance of the wall-averaged buoyancy flux, with the temporal average being taken over the last 25% of the simulation time (typically $100 - 200N^{-1}$), is used to determine the simulations mean Nusselt number and its variance.

For each Prandtl number, the Nusselt number appears to scale as a power-law function of the Rayleigh number. For $\theta = 0$, simulations with $\mathrm{Re}_L \geq 10^{4.75}$ were used to find the best-fit power law parameters, which are listed in Table 4.1.

4.5.3 Analogy to a melting boundary

This simplified model may be viewed as a limit of the problem of the flow near an interface between ice and fresh water. With no salinity, the melting boundary conditions reduce to

$$c\kappa\partial_x T(x = 0) \approx LU \tag{4.58}$$

$$T(x = 0) \approx T_m - \lambda d \tag{4.59}$$

where c is the specific heat capacity of water, L is the latent heat of melting, U is the melting velocity of the ice interface, T_m is the reference freezing point, λ is the freezing point depth dependence, and d is the depth from the surface. Here we have neglected the thermal flux into the ice and linearized the liquidus relation. Converting the temperature into the buoyancy anomaly relative to an ambient linear temperature profile

$$T_a \approx T_0 - \frac{N^2}{\alpha g} d \tag{4.60}$$

and noting that $\alpha < 0$ for freshwater near the freezing point gives

$$\frac{c\kappa}{\alpha g}\partial_x b(x=0) = LU \tag{4.61}$$

$$b(x=0) = B = \alpha g(T_i - T_a) \tag{4.62}$$

$$= \alpha g(T_m - \lambda d - T_0 + \frac{N^2}{\alpha g}d) \tag{4.63}$$

$$= |\alpha| g(T_0 - T_m) + (N^2 + |\alpha| g\lambda)d \tag{4.64}$$

Although the buoyancy anomaly changes with depth, we may use the idealized solution as an approximate local model at a particular depth on scales $\sim L$. The importance of the meltwater flux from the wall can be determined by comparing it to the velocity scale of the laminar solution. Taking $B \approx |\alpha| g(T_0 - T_m)$, $\theta = 0$ and dropping λ gives

$$\frac{U}{W} = -\frac{c\kappa}{\alpha g L}\frac{\mathrm{Nu}B}{\ell W} \tag{4.65}$$

$$\approx \frac{T_0 - T_m}{L/c}\frac{\mathrm{Nu}}{\mathrm{PrRe}_\ell} \tag{4.66}$$

$$\approx 10^{-3}\frac{\mathrm{Nu}}{\mathrm{Re}_\ell}\left(\frac{T_0 - T_m}{\mathrm{K}}\right) \tag{4.67}$$

where $L/c \approx 83$ K and $\mathrm{Pr} \approx 13$ for fresh water at low temperatures. Since the Nusselt number is sublinear in the Reynolds number in the turbulent regime, the meltwater flux is expected to be negligible to the dynamics of the flow.

The expected Rayleigh number for the flow is

$$\mathrm{Ra}_L = \frac{B^4}{N^6\nu\kappa} \tag{4.68}$$

$$\approx \frac{\alpha^4 g^4 (T_0 - T_m)^4}{(\alpha g \partial_z T_a)^3 \nu\kappa} \tag{4.69}$$

$$\approx \frac{|\alpha| g\ \mathrm{Km}^3}{\nu\kappa}\left(\frac{T_0 - T_m}{\mathrm{K}}\right)^4\left(\frac{-\partial_z T_a}{\mathrm{K/m}}\right)^{-3} \tag{4.70}$$

$$\approx 2\times 10^9\left(\frac{T_0 - T_m}{\mathrm{K}}\right)^4\left(\frac{-\partial_z T_a}{\mathrm{K/m}}\right)^{-3} \tag{4.71}$$

with $\alpha \approx -5\times 10^{-5}$ 1/K, $g = 9.8$ m/s^2, $\nu \approx 1.8\times 10^{-6}$ m^2/s, and $\mathrm{Pr} \approx 13$ for fresh water

near the freezing point. Taking the power-law scaling from the simulations with Pr = 10 as $\mathrm{Nu} \approx 0.3\mathrm{Ra}_L^{1/16}$, the boundary layer and melt rate velocities are found to be

$$W = \frac{1}{\mathrm{Pr}^{1/2}} \frac{B}{N} \tag{4.72}$$

$$\approx \left(\frac{|\alpha| g \ \mathrm{Km}}{\mathrm{Pr}} \right)^{1/2} \left(\frac{T_0 - T_m}{\mathrm{K}} \right) \left(\frac{-\partial_z T_a}{\mathrm{K/m}} \right)^{-1/2} \tag{4.73}$$

$$\approx 6 \times 10^{-3} \ \mathrm{m/s} \left(\frac{T_0 - T_m}{\mathrm{K}} \right) \left(\frac{-\partial_z T_a}{\mathrm{K/m}} \right)^{-1/2} \tag{4.74}$$

$$U = \frac{\mathrm{K}}{L/c} \left(\frac{T_0 - T_m}{\mathrm{K}} \right) \frac{W\mathrm{Nu}}{\mathrm{PrRe}_\ell} \tag{4.75}$$

$$\approx 9 \ \mathrm{m/yr} \left(\frac{T_0 - T_m}{\mathrm{K}} \right)^{5/4} \left(\frac{-\partial_z T_a}{\mathrm{K/m}} \right)^{1/16} \tag{4.76}$$

A laboratory-scale melting experiment with $T_0 - T_m \approx 1$ K and $\partial_z T_a \approx -1$ K/m is therefore expected to be just moderately turbulent with $\mathrm{Ra}_L \approx 10^9$. A geophysical-scale problem with $T_0 - T_m \approx 3$ K and $\partial_z T_a \approx -10^{-2}$ K/m, however, is expected to be fully turbulent with $\mathrm{Ra}_L \approx 10^{17}$, Nu ≈ 3.5, and $U \approx 25$ m/yr.

Perhaps more importantly than the thermodynamic simplifications, this model also neglects surface roughness above the scale

$$\ell = \frac{B}{N^2} \left(\frac{\mathrm{Ra}_L}{4} \right)^{-1/4} \tag{4.77}$$

$$\approx 7 \times 10^{-3} \ \mathrm{m} \left(\frac{-\partial_z T_a}{\mathrm{K/m}} \right)^{-1/4} \tag{4.78}$$

which is sure to occur in any geophysical setting. However, it may still be useful as a first-order model when considering the dynamics of melt-driven flows.

4.6 Conclusion

We have used direct numerical simulations to examine the statistics of a convective plume driven by a lateral boundary providing a constant buoyancy anomaly to a linearly stratified fluid. At low Rayleigh number, a steady 1D laminar flow forms along the forced boundary. As the Rayleigh number is increased, this flow becomes linearly unstable, and eventually

saturates into a turbulent plume confined along the forced boundary. The Nusselt number, describing the ratio of the mean buoyancy flux from the wall in the turbulent and laminar states, is found to have a power-law dependence on the Rayleigh number, with the scaling exponent depending on Prandtl number. For Prandtl numbers characteristic of water, the buoyancy flux is found to scale as approximately the $5/4$-th power of the imposed buoyancy anomaly. The model may be applicable as a simplified local model of the flow formed along ice melting into fresh water.

This model provides a very simple landscape for investigating turbulent heat transfer from lateral boundaries in stratified fluids, and many aspects of the model deserve additional investigation. First, it would be interesting to complete Gill's investigation of the linear stability of the laminar flow by examining cases when $\theta \neq 0$. In particular, the behavior may abruptly change as θ becomes negative, and the buoyancy encourages the boundary layer to separate away from the boundary rather than pushing into it. The linear theory for $k_y \neq 0$ should also be investigated, as Squire's theorem does not apply to the flow. The turbulent solutions should also be examined in 3D to determine the impact of including the cross-wall dimension on the flow.

An advantage of performing direct numerical simulations rather than laboratory experiments or large-eddy simulations is that the computed solutions contain the full details of the turbulent flow. In particular, the simulation output can be averaged to determine the statistically-steady Reynolds stresses and transports as a function of distance from the wall. The balances in the Reynolds-averaged equations can then be directly examined, potentially aiding in the development of theories for the Nusselt-Rayleigh scaling exponents, which we leave to future work.

Part III

Nonlinear Tidal Instabilities

Chapter 5

Introduction to astrophysical tides

5.1 The influence of tides of binary systems

To a first approximation, the orbits of gravitationally bound stars and planets can be described by treating these bodies as point particles. For isolated binary systems with velocities that are small compared to the speed of light, this treatment yields classical Keplerian orbital dynamics, with the particular feature that the orbital parameters (semi-major axis, eccentricity, etc.) remain fixed in time. The gravitational field experienced by each body, however, is variable in space, leading to a differential gravitational force across the finite extent of each body, inducing a tidal deformation stretching the body in the direction of its companion (Fig. 5.1). The order of magnitude of the tide is characterized by the ratio of the differential gravitational pull across the radius of the body to its own gravity, as

$$\epsilon = \frac{\frac{GM_2}{a^3}R_1}{\frac{GM_1}{R_1^2}} = \frac{M_2}{M_1}\left(\frac{R_1}{a}\right)^3 \tag{5.1}$$

This tidal bulge breaks the spherical symmetry of the body, causing deviations from the Keplerian orbit. In particular, if the orbit is eccentric or one of the bodies is rotating asynchronously to the orbit, then the tidal bulge will be pulled slightly along with the rotation of the planet. In the frame of the asynchronous body, the tidal potential will be non-stationary and induce a nonuniform flow within the body. The associated dissipation from this flow drains energy from the orbit, causing the orbit to evolve towards a synchronized circular orbit. For example, the fast rotation of the earth compared to the moon's orbit causes the time-dependence of earth's tide, and the resulting dissipation is slowing the earth's rotation by ~ 20 μs per year while expanding the moons orbit by ~ 4 cm per year.

In addition to their impact on the orbital dynamics of a system, tidal dissipation both

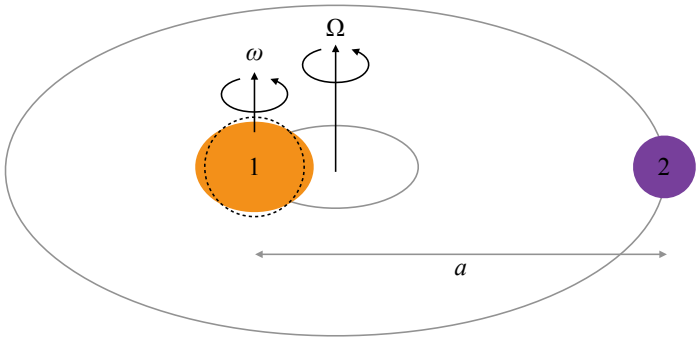

Figure 5.1: Schematic of a tidally-deformed body in a binary orbit. If the deformed body's rotation frequency ω is slower (faster) than the orbital frequency Ω, tidal dissipation will shrink (expand) the orbit.

reflects and affects the internal structure of stars and planets. Tidal dissipation removes energy from the orbit and deposits it as heat within tidally deformed bodies, potentially with a substantial impact in some systems including hot Jupiters. The exact response of a body to the tidal potential, and hence the magnitude of tidal dissipation, however, depends on the internal structure of that body. This fact raises the possibility of probing the interior structure of stars and planets via the tidally induced evolution of the orbits of those bodies.

5.2 Estimating tidal dissipation rates

In the limit of $\epsilon \rightarrow 0$, the structure of the tidal distortion of a body can be calculated using linear theory. For a slowly-rotating fluid body, the tidal response can be well-approximated by its expansion over the normal modes of the body, including pressure-support global sound waves (p-modes) and buoyancy-supported global internal waves (g-modes). The amplitudes of these modes satisfy the equations of damped-driven harmonic oscillators with nonlinear couplings. In the linear regime, the tidal response is simply computed from these amplitude equations, and depends on the spatial overlap of the modes with the tidal potential and the natural frequency of the modes.

The associated dissipation $\dot{E}$ due to the effective viscosity in turbulent/convective regions as well as radiation diffusion acting on the tidal perturbation can be then calculated. This dissipation is formally $O(\epsilon^2)$, and is commonly non-dimensionalized using the linear tide's energy and tidal frequency ω_T as $Q \sim \dot{E}/(\omega_T E)$. For solar-type stars, estimates from linear theory give $Q \sim 10^8$ (Ogilvie, 2014). Measuring the tidal evolution of individual systems is exceedingly challenging, but the study of Meibom et al. (2005) examined the circularization period as a function of age in collections of short-period solar-type binaries. This measurement implies dissipation rates substantially larger than those predicted by linear theory, consistent with $Q \sim 10^6$.

This and other results raise the possibility that nonlinear effects may be important in determining tidal dissipation in close binaries. As the amplitude of the tide is increased, instabilities due to nonlinear coupling of waves inside the deformed star or planet may lead to substantially enhanced dissipation if these unstable waves grow to large amplitudes and break. Recent studies have used stellar structure models to compute the normal modes, coupling coefficients, and stability thresholds for parametric instabilities of two g-modes in solar-type stars (Barker et al., 2011; Weinberg et al., 2012) and non-parametric instabilities due to the coupling of p-modes and g-modes in neutron stars (Venumadhav et al., 2014; Weinberg, 2016; Weinberg et al., 2013). Other studies have built on these calculations using oscillator networks and parameterized models to estimate the resulting saturated states and the corresponding nonlinear tidal dissipation (Essick et al., 2016a,b). However, uncertainties about the saturated statistics remain due to the complicated nature of these instabilities and many regimes remain to be explored.

5.3 Probing neutron star interiors

Understanding the nonlinear stability of tides in neutron stars has been made particularly timely by the recent and ongoing gravitational wave observations of neutron star binaries by LIGO. Neutron stars form a rich laboratory for studying physics under extreme conditions with strong gravity, strong magnetic fields, and rapid rotation. A major open problem is determining the composition and behavior of matter at the densities occurring in the deep interior of neutron stars (NS, hereafter). These regions are partially supported by neutron degeneracy pressure, but unlike in electron-denegeracy-supported white-dwarfs, the relevant densities are such that the inter-particle spacing is comparable to the neutron radius and the scale of the strong nuclear force. Interactions between the particles are

therefore significant, and simple neutron degeneracy pressure does not describe the equation of state of such material.

A wide variety of theoretical models for the equation of state of such matter have been proposed, and gravitational wave observations of neutron star binaries may provide a way to test these models. When a NS is in a close binary with another NS or black hole, the gravitational waves emitted by the system begin to drain significant energy from the orbit of the binary. As this energy is carried away by gravitational waves, the orbit shrinks and the amplitude of the gravitational waves increases. This process runs away, eventually resulting in the merger of the objects as the orbit shrinks below their physical sizes. The gravitational wave signature of one NS-NS merger was recently observed by the LIGO and VIRGO collaboration (Abbott et al., 2017) (Fig. 5.2).

Tidal dissipation also removes energy from the orbit and may introduce a measurable phase shift into the gravitational wave signature of the binary if it becomes sufficiently large relative to the gravitational wave emission. Dissipation from the linear tide is likely too small to be seen in all but the highest signal-to-noise events (Agathos et al., 2015; Damour et al., 2012; Hinderer et al., 2016; Lackey et al., 2015; Read et al., 2009). However, if the tide becomes nonlinearly unstable during the inspiral, the resulting dissipation may reach observable levels. The emergence of tidal instabilities critically depends on the coupling between the normal modes of the NS, which in turn depend on the background density structure and equation of state of the interior.

Determining the tidal dissipation due to nonlinear instabilities is beyond the reach of perturbative calculations, but Essick et al. (2016a) approached the problem by developing a coarse parameterization of the saturation of unstable collections of coupled modes. Their results indicate that with plausible values for the p-g instability bandwidth, growth rates, and thresholds, tidal dissipation might significantly impact LIGO observations and bias estimates of the observed system masses and distances inferred from templates neglecting these effects. They conclude that first-principles calculations of tidal instabilities with different neutron star models to confirm the growth rates and bandwidths from perturbation theory, and possibly determine their saturation, could significantly reduce the degeneracies in inferring binary-NS system parameters from LIGO waveforms and may help place useful constraints on the neutron-star equation of state.

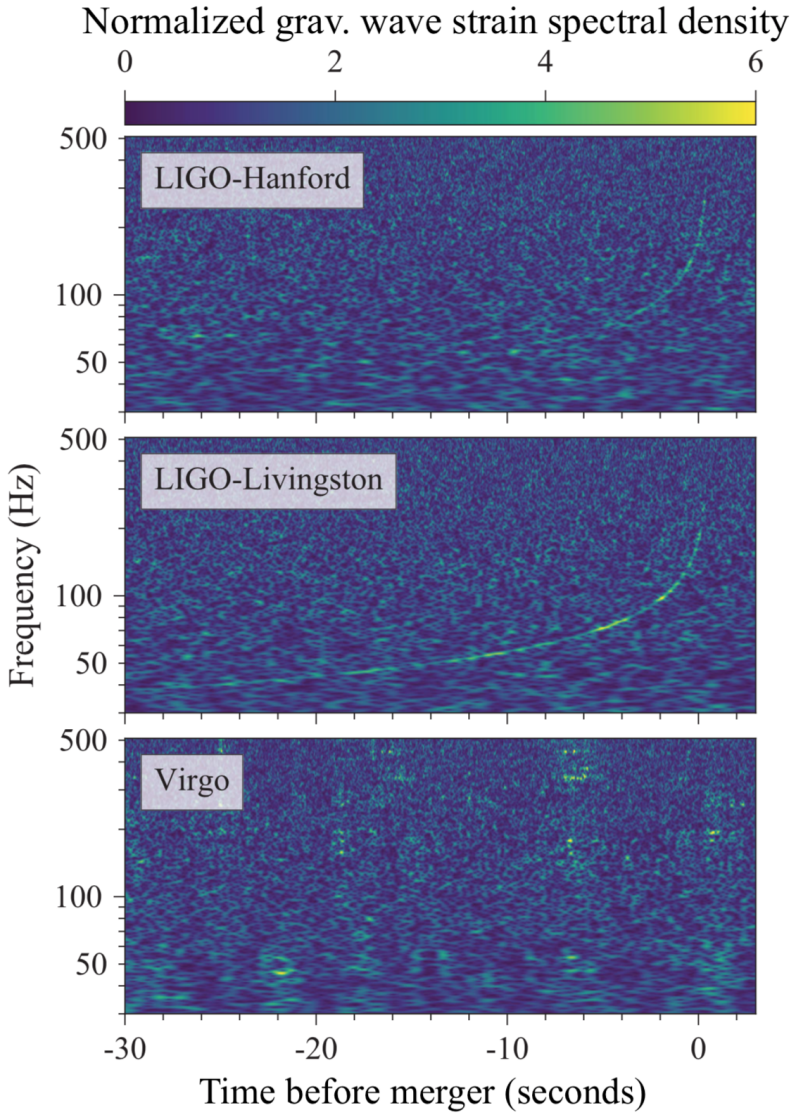

Figure 5.2: Gravitational wave signatures of a binary neutron-star inspiral as observed by the LIGO and VIRGO detectors. Figure adapted from Abbott et al. (2017).

Chapter 6

Direct simulations of tidal stability thresholds

This work was supervised by Nevin Weinberg.

6.1 Introduction

Understanding the rate at which orbital energy is dissipated by tidal processes remains an important open question in our understanding of the orbital dynamics and interior structure of stars, compact objects, and planets in close binaries (Ogilvie, 2014). Most investigations of tidal dissipation have considered the linear damping of the leading-order tidal response by viscosity and thermal diffusion. However, observations of solar binaries indicate that tidal dissipation can exceed the predictions from this linear theory by several orders of magnitude (Meibom et al., 2005).

Recent work has begun examining the role of nonlinear effects in determining the rate at which energy is dissipated in tidally forced systems. Barker et al. (2011) and Weinberg et al. (2013) showed that as the amplitude of the tide is increased, instabilities due to the nonlinear coupling of waves inside the star or planet may lead to substantially enhanced tidal dissipation. (Essick et al., 2016b) applied this theory to examine the onset of the parametric instability of g-modes in stars hosting hot Jupiters. (Essick et al., 2016a; Venumadhav et al., 2014; Weinberg, 2016; Weinberg et al., 2013) have investigated the possibility of instabilities due to coupled g-modes and p-modes in binary neutron stars, and the potential observability of the resulting dissipation in the gravitational wave signatures of these systems. All of these studies are based on complex calculations involving high-order perturbation theory and integrations treating the fluid response as a connected network of stellar normal modes. Determining the collective growth rate of many coupled modes, particularly in the case of p-g coupling where the modes are far from resonance with tidal

frequency, is a challenging task due to the complex nature of the weakly nonlinear dynamics.

In this work, we consider an alternative approach, directly simulating the fully nonlinear response of a fluid layer subjected to a tidal potential to computationally probe the onset of tidal instabilities. We start with a highly simplified model examining the tidal stability of a plane-parallel atmosphere. This model allows for the examination of the fundamental physics of nonlinear mode coupling in a simple 2D geometry. Using Dedalus, we solve for the background atmospheric structure, the eigenmodes of the atmosphere, the linear tidal solution, and the nonlinear coupling coefficients between the linear tide and the eigenmodes. We use these coefficients to predict the threshold amplitude for the nonlinear instability of the tidal solution, and compare the results with fully nonlinear simulations.

6.2 Background structure

We begin by solving for the background structure of the atmosphere. For computational simplicity, we specifically consider a thin section of a compressible atmosphere with rigid bottom and top boundaries. While the rigid boundaries are unrealistic for fluid bodies, they allow us to consider and simulate the flow simply from an Eulerian viewpoint. We define a 2D coordinate system (x, z), where x is the horizontal coordinate, and z is the vertical coordinate. We impose a constant downward gravitational acceleration g on the fluid, and neglect the self-gravity of the atmosphere.

The background density profile $\rho_0(z)$ and pressure profile $p_0(z)$ must be in hydrostatic equilibrium, satisfying

$$\frac{\partial p_0}{\partial z} = -\rho_0 g. \tag{6.1}$$

We are interested in producing atmospheres with specific background buoyancy profiles, given by

$$N^2(z) = -g\left(\frac{\partial_z \rho_0}{\rho_0} - \frac{1}{\Gamma_1}\frac{\partial_z p_0}{p_0}\right) \tag{6.2}$$

where N is the buoyancy frequency and Γ_1 is the first adiabatic exponent

$$\Gamma_1 = \left.\frac{\partial \ln p}{\partial \ln \rho}\right|_{\text{ad}}. \tag{6.3}$$

We therefore fix Γ_1, g, and $N_2(z)$, and implicitly solve for the background entropy profile. The above equations combine to produce a second-order equation for the background

pressure as

$$\partial_{zz} p_0 = \left(\frac{1}{\Gamma_1} \frac{\partial_z p_0}{p_0} - \frac{N^2(z)}{g} \right) \partial_z p_0 \tag{6.4}$$

which can be solved to determine the structure of the atmosphere given the basal boundary conditions $p_0(z = 0) = p_0^B$ and $\rho_0(z = 0) = \rho_0^B$. The system can be non-dimensionalized using the basal pressure scale height, the basal mass scale, and the basal dynamic time as characteristic length, mass, and time scales:

$$\hat{L} \equiv -\frac{p_0}{\partial_z p_0}(z = 0) = \frac{p_0^B}{\rho_0^B g} \tag{6.5}$$

$$\hat{M} \equiv \rho_0^B \hat{L}^3 \tag{6.6}$$

$$\hat{T}^2 = \hat{L}/g \tag{6.7}$$

Choosing these scales is equivalent to taking $p_0^B = 1$, $\rho_0^B = 1$, and $g = 1$.

We solve the non-dimensionalized form of the above equation using the nonlinear boundary-value solver in Dedalus. A fast shooting method is used to compute an approximate solution for the background pressure profile $p_0(z)$ from $z = 0$ up to a given height L_z. The Dedalus solver then uses a symbolically-computed functional Jacobian to perform Newton iterations of the Chebyshev expansion of p_0. This method rapidly converges to machine precision.

For our fiducial model, we consider an atmosphere with $\Gamma_1 = 5/3$, $L_z = 2$ (corresponding to approximately 4 pressure scale-heights), and a Gaussian buoyancy profile

$$N^2(z) = N_0^2 \exp\left(-\frac{(z - z_c)^2}{2 z_w^2} \right) \tag{6.8}$$

with a baseline amplitude of $N_0 = 0.2$, cavity center $z_c = 1$, and cavity width $z_w = 0.2$. We choose a Gaussian profile for the buoyancy frequency to create a confined cavity for the g-modes in the model, since the nonlinear coupling between modes may be strongest near their turning points (Weinberg et al., 2013). The smooth nature of the imposed buoyancy profile results in smooth background pressure and density profiles, with exponentially converging Chebyshev expansions. This property is key to the accuracy and efficiency of the remaining computations, as it allows us to represent the background atmospheric structure to high precision with a limited number of Chebyshev modes. For the fiducial model, approximately 50 Chebyshev modes are needed to resolve the background pressure

to double precision (roughly 15 digits).

6.3 Governing equations

We are interested in the nonlinear dynamics of flow driven by a tidal potential acting on the background atmosphere. The governing equations are given by the Navier-Stokes equations for the conservation of mass, momentum, and energy in the fluid:

$$\frac{\partial \rho}{\partial t} + \mathbf{u} \cdot \nabla \rho + \rho \nabla \cdot \mathbf{u} = 0 \tag{6.9}$$

$$\rho \left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \right) + \nabla p = \rho \mathbf{g} - \rho \nabla \phi + \nabla \cdot \tau \tag{6.10}$$

$$\rho T \left(\frac{\partial s}{\partial t} + \mathbf{u} \cdot \nabla s \right) = \tau : \nabla \mathbf{u} \tag{6.11}$$

where ϕ is the tidal potential, and τ is the viscous stress tensor. Here we have neglected thermal diffusion, as viscosity is expecting to be the primary damping mechanism for neutron star oscillations. We have again neglected the self-gravity of the atmosphere. The viscous stress tensor, neglecting bulk viscosity, is given by

$$\tau = \mu \left(\nabla \mathbf{u} + \nabla \mathbf{u}^T - \frac{2}{3} (\nabla \cdot \mathbf{u}) \mathbf{I} \right) \tag{6.12}$$

We take the plane-parallel layer to be horizontally periodic on a length L_x. We set the top and bottom boundaries to be impenetrable and stress-free, to minimize the development of boundary layers:

$$\mathbf{u} \cdot \mathbf{e}_z = 0 \qquad z = 0, L_z \tag{6.13}$$

$$\mathbf{e}_x \cdot \tau \cdot \mathbf{e}_z = 0 \qquad z = 0, L_z \tag{6.14}$$

For simplicity, we neglect the entropy production due to viscosity, which is expected to be small as it is second order in velocity amplitudes and proportional to the molecular viscosity. We then have $\mathrm{d}s/\mathrm{d}t \approx 0$ and rewrite the energy equation as a pressure equation

$$\frac{\partial p}{\partial t} + \mathbf{u} \cdot \nabla p + \Gamma_1 p \nabla \cdot \mathbf{u} = 0 \tag{6.15}$$

It is additionally convenient to change variables from the mass density to the specific volume

$\alpha = 1/\rho$ to remove the density factor from the temporal derivatives in the momentum equation, while maintaining a simple quadratic form for the nonlinearities. We split the specific volume and pressure between the background and deviations as $\alpha = \alpha_0 + \alpha_1$ and $p = p_0 + p_1$. The velocity is divided into a constant horizontal shift and deviations as $\mathbf{u} = \mathbf{u}_0 + \mathbf{u}_1 = (U\mathbf{e}_x) + (u\mathbf{e}_x + w\mathbf{e}_z)$. With these substitutions, and cancelling the hydrostatic terms ($\alpha_0 \nabla p_0 = \mathbf{g}$), the equations become

$$\frac{\partial \mathbf{u}_1}{\partial t} + U\frac{\partial \mathbf{u}_1}{\partial x} + \alpha_0 \nabla p_1 + \alpha_1 \nabla p_0 - \alpha_0 \nabla \cdot \tau_1 = -\nabla\phi - \mathbf{u}_1 \cdot \nabla \mathbf{u}_1 - \alpha_1 \nabla p_1 + \alpha_1 \nabla \cdot \tau_1 \quad (6.16)$$

$$\frac{\partial \alpha_1}{\partial t} + w\frac{\partial \alpha_0}{\partial z} + U\frac{\partial \alpha_1}{\partial x} - \alpha_0 \nabla \cdot \mathbf{u}_1 = -\mathbf{u}_1 \cdot \nabla \alpha_1 + \alpha_1 \nabla \cdot \mathbf{u}_1 \quad (6.17)$$

$$\frac{\partial p_1}{\partial t} + w\frac{\partial p_0}{\partial z} + U\frac{\partial p_1}{\partial x} + \Gamma_1 p_0 \nabla \cdot \mathbf{u}_1 = -\mathbf{u}_1 \cdot \nabla p_1 - \Gamma_1 p_1 \nabla \cdot \mathbf{u}_1 \quad (6.18)$$

$$w = 0 \qquad z = 0, L_z \quad (6.19)$$

$$\tau_{1,xz} = 0 \qquad z = 0, L_z \quad (6.20)$$

We have written the equations with all linear terms in the deviation variables on the left-hand side, and all inhomogeneous and nonlinear terms on the right-hand side. Many analyses of stellar oscillations proceed by converting this equations into Lagrangian equations for the fluid parcel displacement field ξ. The Lagrangian approach has the advantage of allowing one to follow the displacements of a free surface, but requires truncations in the evaluation of terms at the displaced position of a fluid parcel, leading to a complicated hierarchy of many terms depending on the expansion order under consideration. The Eulerian equations, however, are an exact nonlinear system with only quadratic terms. In the remainder of this paper, we will analyze the above system in the generalized system form considered by the Dedalus framework:

$$\mathcal{M} \cdot \frac{\partial \mathcal{X}}{\partial t} + \mathcal{L} \cdot \mathcal{X} = \mathcal{F}(\mathcal{X}, t) = \mathcal{H}(t) + \mathcal{G}(\mathcal{X}, \mathcal{X}) \quad (6.21)$$

Here $\mathcal{M}$ and $\mathcal{L}$ represent block-operators acting on the state vector $\mathcal{X} = (\alpha_1, p_1, u, w)$, and $\mathcal{F}(\mathcal{X})$ represent the nonlinear operators. We have further split $\mathcal{F}(\mathcal{X})$ into a homogeneous part $\mathcal{H}(t)$ which represents the tidal forcing, and a bilinear part $\mathcal{G}(\mathcal{X}, \mathcal{X})$ representing the quadratic nonlinearities.

The equations are discretized in Dedalus using the direct product of a Fourier series in the x direction and a Chebyshev series in the z direction. Since α_0 and ρ_0 are independent

of x, the linear portion of the equations is separable over the horizontal Fourier modes. For each wavenumber k, we then have a coupled system for the Chebyshev expansion of $\mathcal{X}_k(z) = \int e^{-ikx}\mathcal{X}(x,z)\mathrm{d}x$, which we denote by X_k. We write these Chebyshev discretized systems as

$$M^k \frac{\partial X_k}{\partial t} + L^k X_k = F_k(X,t) = H_k(t) + G_k(X,X) \tag{6.22}$$

Here M^k and L^k are the discretized matrix forms of the corresponding operators, and X_k is a column vector. We suppress the dot notation for matrix-matrix and matrix-vector multiplication with the discretized objects.

6.4 Non-adiabatic eigenmodes

6.4.1 Forward modes

For a given wavenumber k, the temporal eigenmodes of the linear portion of the deviation equations satisfy the generalized eigenvalue problem

$$\sigma_{k,i} M^k X_{k,i} + L^k X_{k,i} = 0 \tag{6.23}$$

We solve this eigenvalue problem numerically using the eigenvalue solver in Dedalus. The Dedalus solver implements both a sparse routine for quickly finding eigenmodes near a target eigenvalue and a dense routine for finding all of the numerical eigenmodes at a given resolution.

The eigenvectors $X_{k,i}$ produced by the numerical solvers are normalized by their L_2 vector norm, which is not a physically meaningful normalization. We renormalize each eigenmode so that its nondimensional energy is unity, i.e.

$$E = 2\int \rho_0 \mathbf{u}_1 \cdot \mathbf{u}_1 \mathrm{d}\mathbf{x} \tag{6.24}$$

$$X_{k,i} \rightarrow \sqrt{\frac{\hat{E}}{E_{k,i}}} X_{k,i} \tag{6.25}$$

where the leading factor of two in the energy comes from including the potential energy of the mode, the velocities implicitly include both a mode and its complex conjugate, and the energy scale is $\hat{E} = \hat{M}\hat{L}^2/\hat{T}^2$.

Fig. 6.1 shows a subset of the eigenmodes from the dense solver for horizontal

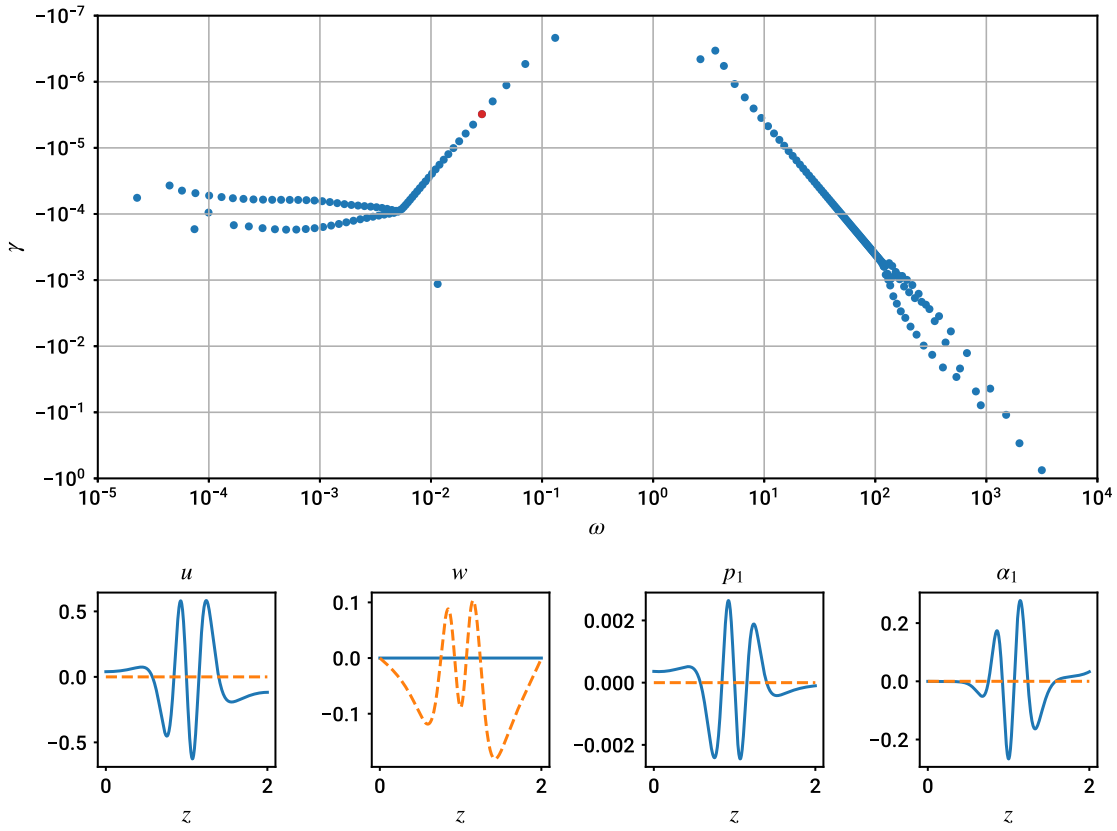

Figure 6.1: Top panel: Eigenmode frequencies in the complex plane for the fiducial model atmosphere with $\mu = 10^{-8}$. The left branch are g-modes and the right branch are p-modes. Departures at high and low frequencies from the power law trends indicate the end of the well-resolved modes. Bottom: Real (solid) and imaginary (dashed) components of the 5th g-mode.

wavenumber $k = \pi$, vertical resolution $N_z = 128$, and $U = 0$. The top panel shows the complex mode frequencies $i\sigma$ in the complex plane, or equivalently the growth rate γ vs. the wave frequency ω under the identification $\sigma = -i(\omega + i\gamma)$. The left branch of modes with low frequencies are g-modes, and the right branch with high frequencies are p-modes. The well-resolved modes follow smooth power-laws with $\gamma \sim -|\omega|^{\pm 2}$ for p-modes and g-modes, respectively. For $|\omega| \lesssim 5 \times 10^{-3}$ and $|\omega| \gtrsim 10^2$, the eigenvalues depart from these relations as the corresponding eigenmodes are underresolved by the numerical discretization. For a Chebyshev resolution of $N_z = 128$ modes, there are roughly 30 well-resolved g-modes and 80 well-resolved p-modes. The structure of the 5th g-mode is plotted in the panels along the bottom of the figure.

6.4.2 Dual modes

We are interested in tracking the amplitudes of the eigenmodes in fully nonlinear simulations. This requires a method for decomposing an arbitrary state of the simulation into a sum of the linear eigenmodes. However, since we are directly including viscous dissipation, the equations are not self-adjoint and hence the eigenmodes are not orthogonal under the energy inner product. Performing an eigenmode decomposition therefore requires us to find a new basis that is dual to our nonadiabatic eigenmodes under some inner product.

One option would be to derive the equations that are adjoint to our linear system under the usual energy inner product. The eigenmodes of those equations would then be orthogonal to the forward eigenmodes under the energy inner product. Computing each eigenmode coefficient, however, would require evaluating this projection as an integral operator. A more computationally efficient option is to use a basis that is dual under the Chebyshev inner product, since the corresponding projection is simply the dot product between the Chebyshev expansion of the dual mode and the Chebyshev expansion of the given state.

The dual modes are derived by simply considering the left generalized eigenvectors of the Chebyshev discretized system. Dropping the wavenumber for clarity, the forward (or right) generalized eigenmodes again satisfy

$$\sigma_i M X_i + L X_i = 0 \tag{6.26}$$

The left generalized eigenvectors of the same system satify

$$\gamma_i^* Y_i^* M + Y_i^* L = 0 \tag{6.27}$$

which can be arranged to show that these modes are the right generalized eigenvectors of the Hermitian-transposed system as

$$\gamma_i M^\dagger Y_i + L^\dagger Y_i = 0 \tag{6.28}$$

For a particular wavenumber k, we solve for these dual modes in Dedalus simply by replacing the L^k and M^k matrices in the eigenvalue solver with their Hermitian transposes. The resulting left and right generalized eigenvectors then satisfy a modified orthogonality relation

$$Y_j^* L X_i = -\sigma_i Y_j^* M X_i = -\gamma_j^* Y_j^* M X_i \tag{6.29}$$

$$\implies Y_j^* M X_i (\sigma_i - \gamma_j^*) = 0 \tag{6.30}$$

It is simple to show that for each right eigenmode with σ_i, there must be a corresponding left eigenmode with $\gamma_i = \sigma_i^*$. The orthogonality relation is therefore $Y_j^* M X_i \propto \delta_{i,j}$, and the proportionality is made into an equation simply by renormalizing each Y_j. The final metric matrix given by $Y_j^* M X_i *$ is found to be close to $\delta_{i,j}$ to within $\sim 10^{-10}$ to the identity for the well-resolved spectrum. Numerical errors, related to large differences in the initial and final scaling of the forward and dual modes, result in projection errors on the order of $\sim 10^{-6}$ for the non-resolved modes. This numerical dual basis therefore presents a reliable means of orthogonally projecting out the amplitude of the resolved eigenmodes in a given solution.

Finally, we note that due to the non-Hermitian nature of these operators, we are not certain that the set of discrete eigenmodes forms a complete basis for the solutions. In practice, however, we have found that we are able to reconstruct solutions to high accuracy by decomposing them across a large set eigenmodes using the numerical dual basis, and summing together the elements of this decomposition.

6.4.3 Modal equations

Again assuming completeness of the forward eigenmodes, we can write an arbitrary state of the full simulation as

$$X = \sum_{k,i} a_{k,i}(t) X_{k,i} \tag{6.31}$$

Substituting this form into the evolution equation (Eq. (6.22)), again suppressing the wavenumber for simplicity, gives

$$\sum_i \frac{\mathrm{d}a_i}{\mathrm{d}t}(MX_i) + a_i L X_i = F(X,t) \tag{6.32}$$

$$\sum_i \left(\frac{\mathrm{d}a_i}{\mathrm{d}t} - \sigma_i a_i\right)(MX_i) = F(X,t) \tag{6.33}$$

Projecting against the derived dual basis produces the modal equations governing the evolution of the eigenmode amplitudes:

$$\frac{\mathrm{d}a_i}{\mathrm{d}t} - \sigma_i a_i = Y_i^* F(X,t) = f_i(X,t) \tag{6.34}$$

Given completeness of the eigenmodes and a perfect dual projection scheme, integrating this set of equations would be equivalent to integrating the system for the Chebyshev discretization. However, the evaluation of the nonlinear terms requires the construction of the full solution X in grid space, which is accomplished using the FFT in the spectral method taking $O(N_z \log N_z)$ time, but would have to be done with a matrix-multiply transform taking $O(N_z^2)$ time using the eigenmode decomposition. Integrating the equations in the Chebyshev spectral method is therefore faster and better conditioned than using the eigenmode basis, but the latter is a useful analytical tool for understanding the evolution of the solution.

6.5 Linear tidal solution

For our fiducial model, we consider a traveling external tidal potential given in the stationary frame of the atmosphere as

$$\phi^S(x,z,t) = \epsilon \cos(k_T x - \omega_T t)\exp(k_T(z - L_z)) \tag{6.35}$$

This form is chosen as it is the simplest horizontally periodic and harmonic potential. Instead of integrating the equations in the stationary frame, we boost to a frame moving with the tide with $U = -\omega_T/k_T$. The tidal potential in this "comoving" frame is time-independent and given by

$$\phi^C(x,z) = \epsilon \cos(k_T x)\exp(k_T(z - L_z)) \tag{6.36}$$

The linear tide is the solution of the equations recovered in the limit of a small amplitude

tide, i.e. $H \sim \epsilon \ll 1$. This leading order solution is found by dropping the nonlinear terms $G(X, X)$ and solving the resulting linear system, which is time-independent in the comoving frame:

$$L^k X_k = H^k \tag{6.37}$$

This system is directly solved using the linear boundary value solver in Dedalus to produce the linear tidal solution. The mode amplitudes for the linear tide are simply given by

$$a_{k,i} = -\frac{h_{k,i}}{\tilde{\sigma}_{k,i}} \tag{6.38}$$

where the eigenvalues of the eigenmodes for a given wavelength k are shifted from their stationary values σ to

$$\begin{aligned}
\tilde{\sigma}_{k,i} &= \sigma_{k,i} - iUk && (6.39)\\
&= \sigma_{k,i} + i(k/k_T)\omega_T && (6.40)\\
&= -i(\omega_{k,i} - (k/k_T)\omega_T + i\gamma_{k,i}) && (6.41)\\
&= -i(\Delta_{k,i} + i\gamma_{k,i}) && (6.42)
\end{aligned}$$

Since the tidal potential is proportional to $\cos(k_T x)$, the linear tide only consists of modes with $k = \pm k_T$. An eigenmode can be driven to large amplitude in the linear tide by either coupling strongly to the vertical structure of the tide (large $h_{k,i}$), or being near resonance with the tidal frequency (small $\tilde{\sigma}_{k,i}$ with $\omega_{k,i} \approx \omega_T$). The amplitude distribution with $\omega_T = 0$ is known as the equilibrium tide, while the corrections and resonances due to the finite frequency of the tide is known as the dynamical tide.

Fig. 6.2 shows the linear tidal solution solved using Dedalus with $N_z = 128$, $k_T = \pi$ and $\omega_T = 0.0287517$, on resonance with the 5th g-mode of the fiducial model. Fig. 6.3 shows the corresponding eigenmode amplitudes, computing by projecting the linear solution against the numerically determined dual eigenvectors with $10^{-2} \leq |\omega| \leq 10^2$. The decaying portion apparent in the pressure and specific volume perturbations is the equilibrium tide, corresponding to the broad excitation of p-modes and g-modes seen in the amplitude diagram. The centralized oscillatory feature is the dynamical tide, dominated by the resonant g-mode. The extracted mode amplitudes are found to agree with the analytical results to high precision, and the summation of the reconstitution of the linear response using the depicted modes closely matches the original solution. These tests confirm the

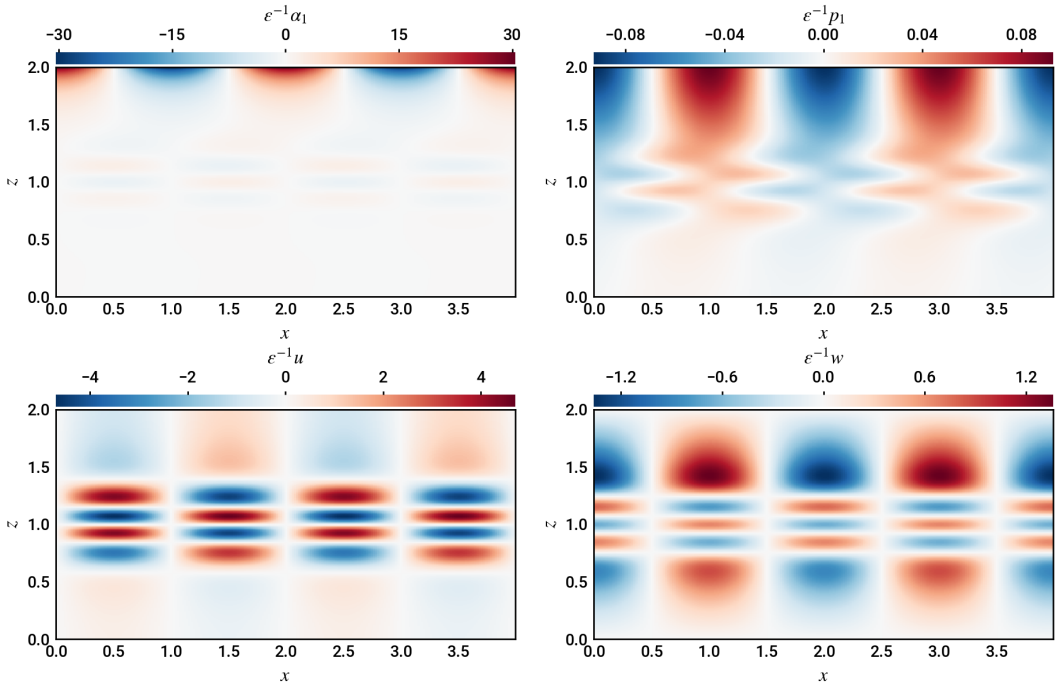

Figure 6.2: The specific volume, pressure, and velocity deviations of the linear tidal solution of the fiducial model in the comoving frame. The decaying feature in the specific volume and pressure is the equilibrium tide, while the oscillatory feature near $z = 1$ is the dynamical tide, dominated by a resonant g-mode.

validity of the dual basis and justify the assumption that the resolved eigenmodes form a sufficient basis representing the tidal solution.

6.6 Weakly nonlinear stability

We are interested in studying the stability of perturbations around the stationary tidal response. First, we show that to leading order, this is equivalent to examining the stability of the linear tide.

We indicate the linear tidal solution as $\bar{X}$, satisfying $L\bar{X} = H$, and split the general solution into the linear tide and perturbations as $X = \bar{X} + X'$. The exact evolution of the

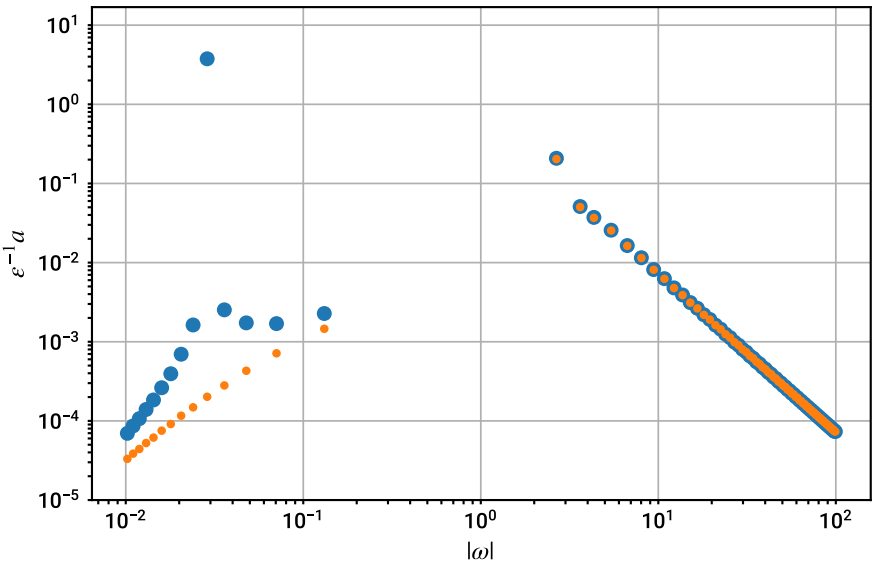

Figure 6.3: Eigenmode amplitudes in the linear tidal solution of the fiducial model. The blue points represent modes with positive ω, and the orange points represent modes with negative ω. The broad distribution of the p-modes corresponds to the equilibrium tide, while the peak in the g-modes corresponds to the resonantly-excited dynamical tide.

deviations is given by

$$M\frac{\partial X'}{\partial t} + LX' = G(\bar{X} + X', \bar{X} + X') \tag{6.43}$$

$$= G(\bar{X}, \bar{X}) + G(\bar{X}, X') + G(X', \bar{X}) + G(X', X') \tag{6.44}$$

using the bilinearity of G. Consider an exact nonlinear solution to the stationary equations $\bar{Y}$, satisfying $L\bar{Y} = H + G(\bar{Y}, \bar{Y})$. Expanding $\bar{Y}$ in positive powers of ϵ, we find that the first-order solution is simply the linear tide, and hence we can write $\bar{Y} = \bar{X} + O(\epsilon^2)$. Deviations Y' from this solution would be governed by

$$M\frac{\partial Y'}{\partial t} + LY' = G(\bar{Y}, Y') + G(Y', \bar{Y}) + G(Y', Y') \tag{6.45}$$

Taking the deviations to be small with $Y' \sim O(\eta)$, and substituting the expansion for $\bar{Y}$, this becomes

$$M\frac{\partial Y'}{\partial t} + LY' = G(\bar{X}, Y') + G(Y', \bar{X}) + O(\epsilon^2\eta) + O(\eta^2) \tag{6.46}$$

Assuming $\eta \ll \epsilon$, we see that to leading order $O(\epsilon\eta)$, the stability of the nonlinear fixed point is governed by the homogeneous linearized dynamics around the linear tide, even though the linear tide is not a fixed point of the nonlinear system.

We therefore proceed by considering this linearized problem:

$$M\frac{\partial X'}{\partial t} + LX' \approx G(\bar{X}, X') + G(X', \bar{X}) \tag{6.47}$$

We are unable to directly examine this problem using the current version of Dedalus because the RHS terms, although linear in X', couple different Fourier modes due to the x-dependence of the linear tide $\bar{X}$. Instead, we use Dedalus to calculate the mode coupling coefficients, and determine the stability by posing the eigenvalue problem in the eigenmode space. The amplitude equations for the above problem are

$$\frac{\mathrm{d}a'_{k,\ell}}{\mathrm{d}t} - \tilde{\sigma}_{k,\ell}a'_{k,\ell} = g_{k,\ell}(\bar{X}, X') + g_{k,\ell}(X', \bar{X}) \tag{6.48}$$

Expanding the RHS terms gives

$$\begin{aligned}
g_{k,\ell}(\bar{X}, X') &= Y^*_{k,\ell}\int e^{-ikx}G(\bar{X}, X')\mathrm{d}x && (6.49)\\
&= Y^*_{k,\ell}\int e^{-ikx}G\left(\sum_{m,i}\bar{a}_{m,i}e^{imx}X_{m,i},\ \sum_{n,j}a'_{n,j}e^{inx}X_{n,j}\right)\mathrm{d}x && (6.50)\\
&= \sum_{m,n,i,j}\bar{a}_{m,i}a'_{n,j}Y^*_{k,\ell}\int e^{-ikx}G\left(e^{imx}X_{m,i},\ e^{inx}X_{n,j}\right)\mathrm{d}x && (6.51)\\
&= \sum_{m,n,i,j}\bar{a}_{m,i}a'_{n,j}Y^*_{k,\ell}\int e^{-ikx}e^{imx}e^{inx}G^{m,n}_{i,j}\mathrm{d}x && (6.52)\\
&= \sum_{m,n,i,j}\bar{a}_{m,i}a'_{n,j}Y^*_{k,\ell}G^{m,n}_{i,j}\delta_{m+n,k} && (6.53)\\
& && (6.54)
\end{aligned}$$

where the fourth line requires G to be horizontally homogeneous. Substituting this expres-

sion, and the equivalent for $g_{k,\ell}(X', \bar{X})$, we get

$$\frac{\mathrm{d}a'_{k,\ell}}{\mathrm{d}t} - \tilde{\sigma}_{k,\ell} a'_{k,\ell} = \sum_{m,n,i,j} (\bar{a}_{m,i} a'_{n,j} + a'_{m,i} \bar{a}_{n,j}) Y^*_{k,\ell} G^{m,n}_{i,j} \delta_{m+n,k} \tag{6.55}$$

$$= \sum_{m,n,i,j} \bar{a}_{m,i} a'_{n,j} Y^*_{k,\ell} (G^{m,n}_{i,j} + G^{n,m}_{j,i}) \delta_{m+n,k} \tag{6.56}$$

$$= \sum_{n,j} \left[\sum_{m,i} \bar{a}_{m,i} Y^*_{k,\ell} (G^{m,n}_{i,j} + G^{n,m}_{j,i}) \delta_{m+n,k} \right] a'_{n,j} \tag{6.57}$$

$$= \sum_{n,j} C_{k,\ell,n,j} a'_{n,j} \tag{6.58}$$

where $C_{k,\ell,n,j}$ is the nonlinear coupling coefficient between the linear tide, the mode $\{n, j\}$, and the mode $\{k, \ell\}$.

The coupling coefficients are computed in Dedalus by utilizing its abilities to symbolically compute the Frechet differentials of arbitrary expressions. The Frechet differential of each mode around the linear tide is computed as

$$K_{n,j} = \partial_X F(\bar{X}) \cdot (e^{inx} X_{n,j}) \tag{6.59}$$

$$= G(\bar{X}, e^{inx} X_{n,j}) + G(e^{inx} X_{n,j}, \bar{X}) \tag{6.60}$$

$$= \sum_{m,i} \bar{a}_{m,i} e^{imx} e^{inx} (G^{m,n}_{i,j} + G^{n,m}_{j,i}) \tag{6.61}$$

The coupling coefficients are then recovered by projecting the differential against the desired wavenumber and dual mode as

$$C_{k,\ell,n,j} = Y^*_{k,\ell} \int e^{-ikx} K_{n,j} \mathrm{d}x \tag{6.62}$$

Finally, we have

$$\frac{\mathrm{d}a'_{k,\ell}}{\mathrm{d}t} = \sum_{n,j} (\tilde{\sigma}_{k,\ell} \delta_{k,n} \delta_{\ell,j} + C_{k,\ell,n,j}) a'_{n,j} \tag{6.63}$$

which we write in the matrix form $\mathrm{d}_t A = QA$. Seeking modes that behave as $A \sim e^{st}$, we now have a standard eigenvalue problem $sA = QA$ for the collective growth rate of nonlinearly coupled modes. The coupling coefficients are proportional to the tidal amplitude ϵ, so the threshold amplitude for a nonlinear instability of the tide is the minimum ϵ for which there is some eigenvalue s with $\Re(s) > 0$.

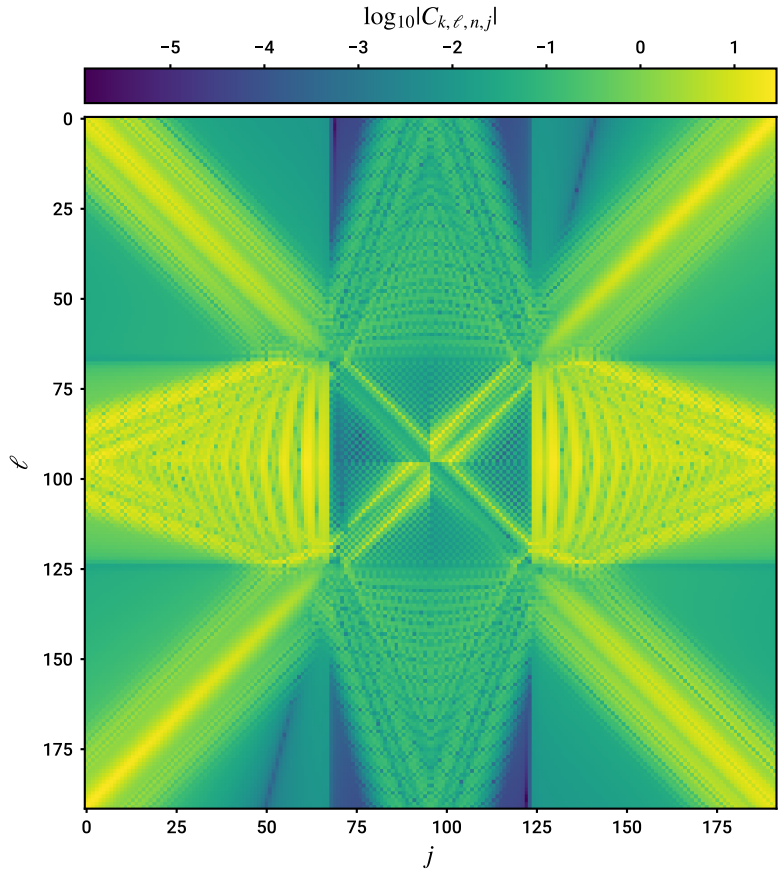

Figure 6.4: Coupling coefficient matrix for the fiducial model with $k = -n = 0.5k_T$. The modes are ordered by ω, producing a block structure corresponding to negative p-modes, negative g-modes, positive g-modes, then positive p-modes.

6.7 Threshold calculations and comparison to simulations

Fig. 6.4 shows the coupling coefficient matrix for the resolved modes ($|\omega k_T/k| \geq 5 \times 10^{-3}$ and $|\omega| \leq 10^2$) of the fiducial model with $k = -n = 0.5k_T$. The modes are ordered by ω, so the three blocks in each direction correspond to the negative p-modes, negative then positive g-modes, and positive p-modes. Fig. 6.5 shows the Q matrix for the fiducial model including resolved modes with $k, n \in \{-2, -1.5, -1, -0.5, 0, 0.5, 1, 1.5, 2\}k_T$ and $\epsilon = 10^{-4}$. The block-banded structure is due to the horizontal selection rule $n \pm k_T = m$. The eigenvalues of this Q matrix with $\epsilon = 10^{-4}$, and the structure of the most-unstable coupled mode, are shown in Fig. 6.6. For small ϵ, the eigenvalues of Q simply match the natural frequencies of the eigenmodes from each of the included horizontal wavenumbers, and the eigenmodes of Q

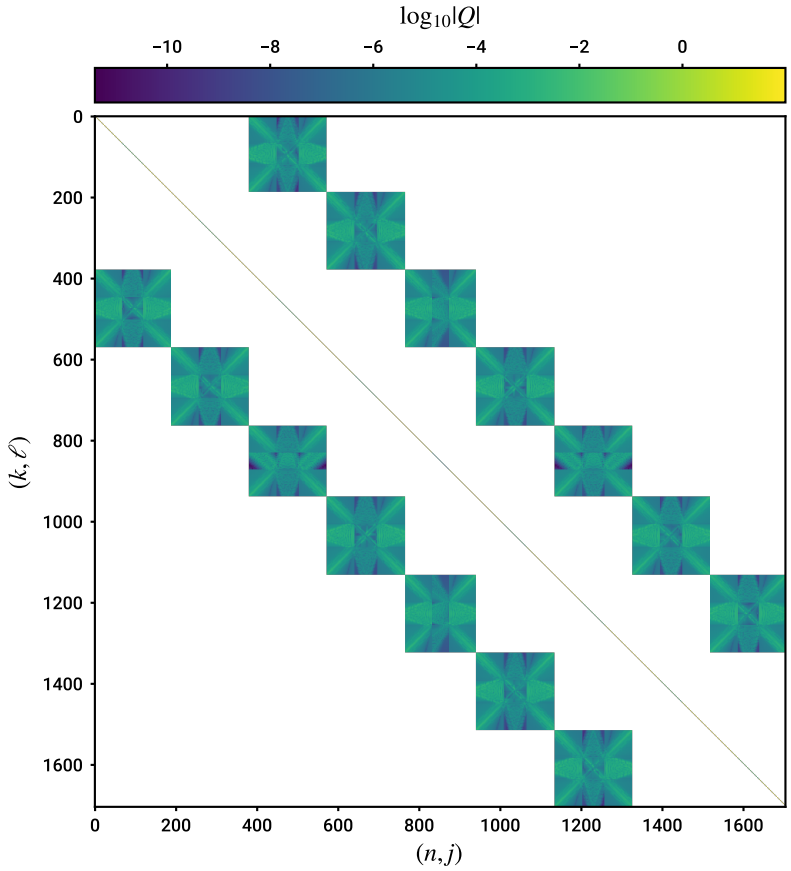

Figure 6.5: Q matrix describing the weakly-nonlinear dynamics of perturbations to the linear tide in the eigenmode basis for $k, n \in \{-2, -1.5, -1, -0.5, 0, 0.5, 1, 1.5, 2\}k_T$ and $\epsilon = 10^{-4}$. The diagonal entries contain the modes' natural frequencies and linear damping. The off-diagonal blocks are the coupling coefficients between groups of modes satisfying the horizontal selection rule $n \pm k_T = m$.

are dominated by the associated uncoupled modes. As ϵ is increased, the eigenvalues of Q are perturbed from the natural mode frequencies and the eigenmodes of Q gain contributions from a range of uncoupled modes. As shown in the figure, the most unstable mode for this model contains contributions from a wide range of horizontal wavenumbers, indicating that the collective behavior of many coupled modes is important to the stability of the tide in this case.

Fig. 6.7 shows the maximum growth rate of the eigenmodes of Q as a function of ϵ for this set of modes. For $\epsilon \gtrsim 3 \times 10^{-5}$, Q has eigenvalues with positive real part, indicating that perturbations to the linear tide are unstable in this range. This calculation was repeated with different groups of horizontal wavenumbers. The threshold amplitude was found to decrease as more wavenumbers were included in the calculation, and roughly saturated with the set listed above. Since not all wavenumbers and eigenmodes can be included, this procedure provides what is likely a conservative estimate of the threshold amplitude for nonlinear tidal instabilities.

To test the prediction of the weakly nonlinear stability calculation, we integrate the fully nonlinear system using the initial value solver in Dedalus. We have performed simulations with typical resolutions of $N_x = 256$ and $N_z = 1024$. The equations were integrated with the RK222 timestepper, a two-stage second-order mixed implicit-explicit Runge-Kutta integrator. Although the resolution requirements are not too taxing, there is a large dynamic range in timescales between the p-modes and g-modes of interest. Since the terms producing linear acoustic waves are implicitly integrated, the acoustic timescales do not need to be resolving for stability, but for accuracy in assessing the amplitude of the corresponding p-modes. We run the initial value problem for order 100 tidal periods, but with typical timesteps of $10^{-4}\omega_T^{-1}$.

The simulations were started from the linear tidal solution. For the fiducial model, we find that simulations with $\epsilon \leq 10^{-5}$ demonstrate no instabilities on the simulated timescale, and instead underwent a slight adjustment to a steady nonlinear equilibrium. Simulations with $\epsilon \geq 10^{-4}$ develop nonlinear instabilities that eventually lead to the crash of the simulation, since insufficient resolution was used to examine their saturation. Overall, these simulations roughly bracket the transition to high-dissipation tidal states to occur within $10^{-5} < \epsilon < 10^{-4}$, in agreement with the predictions from the computed coupling coefficients.

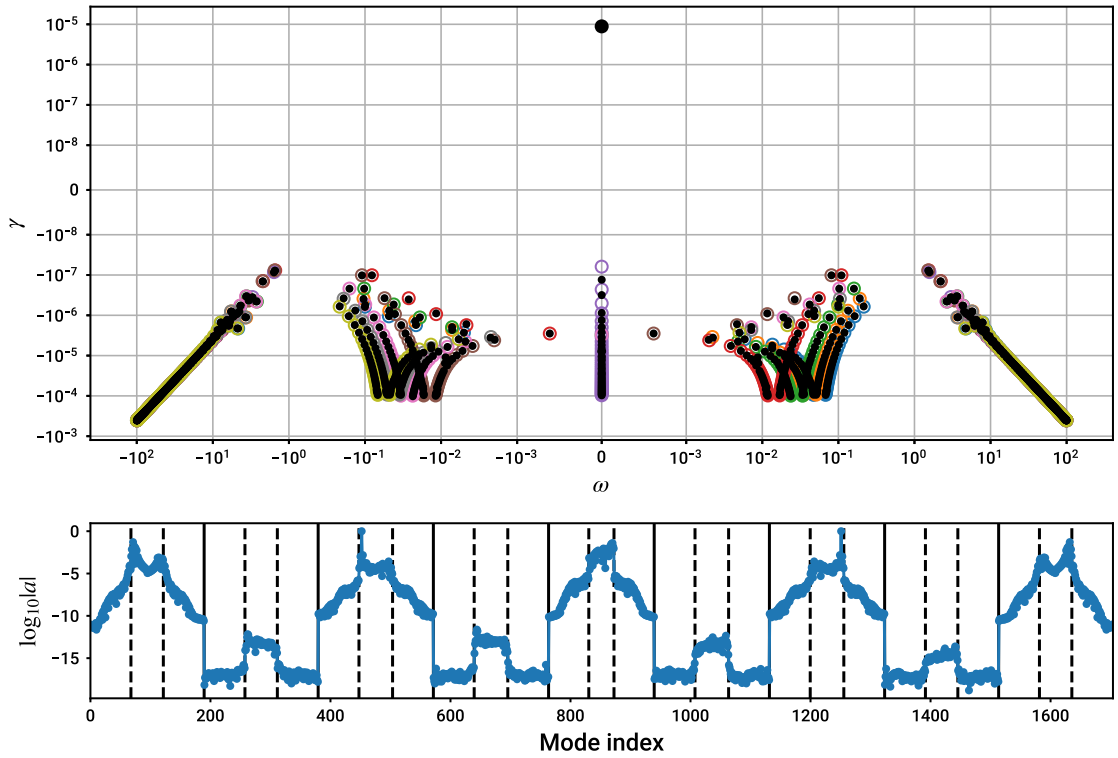

Figure 6.6: Top panel: frequencies and growth rates of the eigenmodes of the Q matrix for the fiducial model with $\epsilon = 10^{-4}$. The circles indicate the natural frequencies of the included modes with various horizontal wavenumbers, shifted to the comoving frame. The dots indicate the eigenvalues of Q, which become increasingly perturbed from the natural frequencies as ϵ increases. Bottom panel: Distribution of mode amplitudes of the unstable mode with $\gamma \sim 10^{-5}$. The modes are ordered by ω and then by k. Solid vertical lines separate the groups by k, and dashed vertical lines separate the p and g modes for a given k.

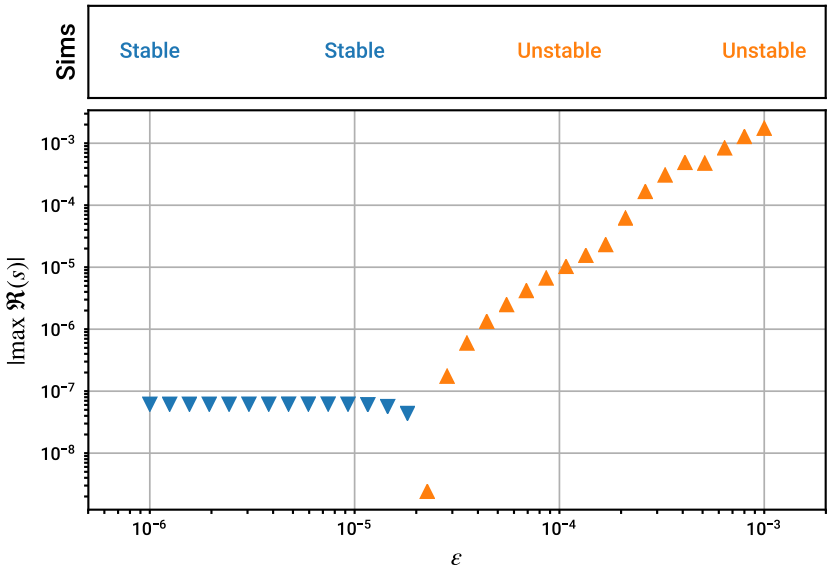

Figure 6.7: Bottom: Maximum collective growth rate, solved by computing the eigenvalues of the Q matrix, as a function of the tidal amplitude ϵ. The downward blue arrows indicate negative growth rates (damping), while the upward orange arrows indicate positive growth rates. The threshold for nonlinear instability of the included modes is $\epsilon_c \sim 3 \times 10^{-5}$. Top: Observed stability of tidal response in fully nonlinear simulations with $\epsilon = 10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}$.

6.8 Conclusion

We have used an Eulerian spectral framework to examine the threshold for nonlinear tidal instabilities in a plane-parallel atmosphere. We first develop a background structure model comprising several scale-heights with a prescribing buoyancy profile. We then compute the non-adiabatic eigenmodes of this atmosphere, and a numerical dual basis that can be used to decompose a general solution into these modes. We solve for the linear response to an externally imposed tidal force, and use a symbolic linearization of the fully nonlinear dynamical equations to compute the coupling coefficients that control the evolution of deviations from the linear tidal solution. Solving an auxiliary eigenvalue problem with the computed mode frequencies, damping rates, and coupling coefficients yields the increasingly coupled behavior of the atmospheric modes as the tidal amplitude is increased. We are able to predict a threshold amplitude for the nonlinear instability of the linear tide in a fiducial model which agrees with simulations of the fully nonlinear equations.

This work primarily serves as a demonstration of the possibilities for using a generalized spectral method for computing tidal instability thresholds. Although limited by resolution constraints, this approach's ability to internally and self-consistently build, analyze, and simulate a tidally-influenced atmosphere may provide a useful alternative for testing analytical and semi-analytical approaches to this problem. The capability of Dedalus to perform all of the above steps also provides a great deal of flexibility for exploring different tidal models in the future. Changes to the governing equations and parameters are easily propagated through the numerical workflow due to the symbolic representation used in Dedalus, so nearly identical code could be used to perform weakly nonlinear stability analyses of many other systems, as well.

In future work, we plan to explore a variety of models with higher resolution simulations to more precisely test the predicted instability threshold and growth rates. Confirming the validity of the weakly-nonlinear stability analysis in the plane-parallel atmosphere will be a useful step towards more realistic computations. In particular, we plan to extend this work to full spherical geometries with newly developed sparse spectral bases (Daniel Lecoanet et al., 2018; G. Vasil et al., 2018) to examine the progression of nonlinear instabilities in more realistic models of tidally perturbed stars and planets. Computations of the saturation of instabilities in realistic geometries would be particularly useful as it is a very difficult feature to study using semi-analytical approaches, yet is key to using observations of binary orbital evolution to infer interior properties of stars and planets.

Bibliography

Abbott, B P et al. (2017). "GW170817: Observation of Gravitational Waves from a Binary Neutron Star Inspiral". In: *Physical Review Letters* 119.1, p. 161101.

Agathos, M et al. (2015). "Constraining the neutron star equation of state with gravitational wave signals from coalescing binary neutron stars". In: *Physical Review D* 92.2, p. 023012.

Armfield, S W, John C Patterson, and Wenxian Lin (2007). "Scaling investigation of the natural convection boundary layer on an evenly heated plate". In: *International Journal of Heat and Mass Transfer* 50.7, pp. 1592–1602.

Ascher, Uri M, Steven J Ruuth, and Raymond J Spiteri (1997). "Implicit-explicit Runge-Kutta methods for time-dependent partial differential equations". In: *Applied Numerical Mathematics* 25.2-3, pp. 151–167.

Barker, Adrian J and Gordon I Ogilvie (2011). "Stability analysis of a tidally excited internal gravity wave near the centre of a solar-type star". In: *Monthly Notices of the Royal Astronomical Society* 417.1, pp. 745–761.

Behnel, S. et al. (2011). "Cython: The Best of Both Worlds". In: *Computing in Science Engineering* 13.2, pp. 31–39. ISSN: 1521-9615. DOI: 10.1109/MCSE.2010.118.

Bergholz, R F (1978). "Instability of steady natural convection in a vertical fluid layer". In: *Journal of Fluid Mechanics* 84.04, pp. 743–768.

Boyd, J P (2001). *Chebyshev and Fourier Spectral Methods: Second Revised Edition*. Dover Books on Mathematics. Dover Publications.

Burns, Keaton J (2013). "Chebyshev Spectral Methods with applications to Astrophysical Fluid Dynamics". MA thesis. University of Cambridge.

Cimarelli, Andrea and Diego Angeli (2017). "Routes to chaos of natural convection flows in vertical channels". In: *International Communications in Heat and Mass Transfer* 81, pp. 201–209.

"Observations: Cryosphere" (2009). In: *Climate Change 2013 - The Physical Science Basis*. Ed. by Intergovernmental Panel on Climate Change. Cambridge: Cambridge University Press, pp. 317–382.

Collette, Andrew (2013). *Python and HDF5*. O'Reilly.

Dalcin, Lisandro, Rodrigo Paz, Mario Storti, and Jorge D'Elia (2008). "MPI for Python: Performance improvements and MPI-2 extensions". In: *Journal of Parallel and Distributed Computing* 68.5, pp. 655–662. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2007.09.005. URL: http://www.sciencedirect.com/science/article/pii/S0743731507001712.

Damour, Thibault, Alessandro Nagar, and Loïc Villain (2012). "Measurability of the tidal polarizability of neutron stars in late-inspiral gravitational-wave signals". In: *Physical Review D* 85.1, p. 123007.

Dinniman, Michael et al. (2016). "Modeling Ice Shelf/Ocean Interaction in Antarctica: A Review". In: *Oceanography* 29.4, pp. 144–153.

Driscoll, Tobin A and Nicholas Hale (2015). "Rectangular spectral collocation". In: *IMA Journal of Numerical Analysis*, dru062.

Essick, Reed, Salvatore Vitale, and Nevin N Weinberg (2016a). "Impact of the tidal p -g instability on the gravitational wave signal from coalescing binary neutron stars". In: *Physical Review D* 94.1, p. 103012.

Essick, Reed and Nevin N Weinberg (2016b). "Orbital Decay of Hot Jupiters Due to Nonlinear Tidal Dissipation within Solar-type Hosts". In: *The Astrophysical Journal* 816.1, p. 18.

Fedorovich, Evgeni and Alan Shapiro (2009). "Turbulent natural convection along a vertical plate immersed in a stably stratified fluid". In: *Journal of Fluid Mechanics* 636, pp. 41–.

Frigo, Matteo and Steven G. Johnson (2005). "The Design and Implementation of FFTW3". In: *Proceedings of the IEEE* 93.2. Special issue on "Program Generation, Optimization, and Platform Adaptation", pp. 216–231.

Gayen, Bishakhdatta, Ross W Griffiths, and Ross C Kerr (2015). "Melting Driven Convection at the Ice-seawater Interface". In: *Procedia IUTAM* 15, pp. 78–85.

— (2016). "Simulation of convection at a vertical ice face dissolving into saline water". In: *Journal of Fluid Mechanics* 798, pp. 284–298.

Gill, A E and A Davey (1969). "Instabilities of a buoyancy-driven system". In: *Journal of Fluid Mechanics* 35.04, pp. 775–798.

Hinderer, Tanja et al. (2016). "Effects of Neutron-Star Dynamic Tides on Gravitational Waveforms within the Effective-One-Body Approach". In: *Physical Review Letters* 116.18, pp. 301–6.

Jenkins, Adrian (2011). "Convection-Driven Melting near the Grounding Lines of Ice Shelves and Tidewater Glaciers". In: *Journal of Physical Oceanography* 41.12, pp. 2279–2294.

— (2016). "A Simple Model of the Ice Shelf-Ocean Boundary Layer and Current". In: *Journal of Physical Oceanography* 46.6, pp. 1785–1803.

Jenkins, Adrian, Keith W Nicholls, and Hugh F J Corr (2010). "Observation and Parameterization of Ablation at the Base of Ronne Ice Shelf, Antarctica". In: *Journal of Physical Oceanography* 40.10, pp. 2298–2312.

Jones, Eric, Travis Oliphant, Pearu Peterson, et al. (2001). *SciPy: Open source scientific tools for Python*. [Online; accessed <today>]. URL: http://www.scipy.org/.

Kerr, Ross C and Craig D McConnochie (2015). "Dissolution of a vertical solid surface by turbulent compositional convection". In: *Journal of Fluid Mechanics* 765, pp. 211–228.

Kimura, Satoshi, Paul R Holland, Adrian Jenkins, and Matthew Piggott (2014). "The Effect of Meltwater Plumes on the Melting of a Vertical Glacier Face". In: *Journal of Physical Oceanography* 44.12, pp. 3099–3117.

Lackey, Benjamin D and Leslie Wade (2015). "Reconstructing the neutron-star equation of state with gravitational-wave detectors from a realistic population of inspiralling binary neutron stars". In: *Physical Review D* 91.4, p. 043002.

Lecoanet, Daniel, Geoffrey M Vasil, Keaton J Burns, Benjamin P Brown, and Jeffrey S Oishi (2018). "Tensor calculus in spherical coordinates using Jacobi polynomials, Part-II: Implementation and Examples". In: *arXiv.org*, arXiv:1804.09283. arXiv: 1804.09283 [1804].

Lecoanet, D et al. (2016). "A validated non-linear Kelvin-Helmholtz benchmark for numerical hydrodynamics". In: *Monthly Notices of the Royal Astronomical Society* 455.4, pp. 4274–4288.

Magorrian, Samuel J and Andrew J Wells (2016). "Turbulent plumes from a glacier terminus melting in a stratified ocean". In: *Journal of Geophysical Research: Oceans* 121.7, pp. 4670–4696.

McConnochie, Craig D and Ross C Kerr (2016a). "The effect of a salinity gradient on the dissolution of a vertical ice face". In: *Journal of Fluid Mechanics* 791, pp. 589–607.

McConnochie, Craig D and Ross C Kerr (2016b). “The turbulent wall plume from a vertically distributed source of buoyancy”. In: *Journal of Fluid Mechanics* 787, pp. 237–253.

— (2017). “Enhanced ablation of a vertical ice wall due to an external freshwater plume”. In: *Journal of Fluid Mechanics* 810, pp. 429–447.

Meibom, Søren and Robert D Mathieu (2005). “A Robust Measure of Tidal Circularization in Coeval Binary Populations: The Solar-Type Spectroscopic Binary Population in the Open Cluster M35”. In: *The Astrophysical Journal* 620.2, pp. 970–983.

Ogilvie, Gordon I (2014). “Tidal Dissipation in Stars and Giant Planets”. In: *Annual Review of Astronomy and Astrophysics* 52.1, pp. 171–210.

Olver, S and A Townsend (2013). “A fast and well-conditioned spectral method”. In: *SIAM Review* 55.3, pp. 462–489.

Read, Jocelyn S et al. (2009). “Measuring the neutron star equation of state with gravitational wave observations”. In: *Physical Review D* 79.1, p. 124033.

Sciascia, R, F Straneo, C Cenedese, and P Heimbach (2013). “Seasonal variability of submarine melt rate and circulation in an East Greenland fjord”. In: *Journal of Geophysical Research: Oceans* 118.5, pp. 2492–2506.

Slater, D A, P W Nienow, T R Cowton, D N Goldberg, and A J Sole (2015). “Effect of near-terminus subglacial hydrology on tidewater glacier submarine melt rates”. In: *Geophysical Research Letters* 42.8, pp. 2861–2868.

Slater, Donald A, Dan N Goldberg, Peter W Nienow, and Tom R Cowton (2016). “Scalings for Submarine Melting at Tidewater Glaciers from Buoyant Plume Theory”. In: *Journal of Physical Oceanography* 46.6, pp. 1839–1855.

Sprague, Michael, Keith Julien, Edgar Knobloch, and Joseph Werne (2006). “Numerical simulation of an asymptotically reduced system for rotationally constrained convection”. In: *Journal of Fluid Mechanics* 551.-1, pp. 141–174.

Straneo, Fiamma and Claudia Cenedese (2015). “The Dynamics of Greenland’s Glacial Fjords and Their Role in Climate”. In: *Annual Review of Marine Science* 7.1, pp. 89–112.

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

Vasil, Geoff, Daniel Lecoanet, Keaton Burns, Jeff Oishi, and Ben Brown (2018). “Tensor calculus in spherical coordinates using Jacobi polynomials. Part-I: Mathematical analysis and derivations”. In: *arXiv.org*, arXiv:1804.10320. arXiv: 1804.10320 [1804].

Vasil, Geoffrey M et al. (2016). “Tensor calculus in polar coordinates using Jacobi polynomials”. In: *Journal of Computational Physics* 325, pp. 53–73.

Venumadhav, Tejaswi, Aaron Zimmerman, and Christopher M Hirata (2014). “The Stability of Tidally Deformed Neutron Stars to Three- and Four-mode Coupling”. In: *The Astrophysical Journal* 781.1, p. 23.

Wang, D and S J Ruuth (2008). “Variable step-size implicit-explicit linear multistep methods for time-dependent partial differential equations”. In: *Journal of Computational Mathematics* 26.6.

Weinberg, Nevin N (2016). “Growth Rate of the Tidal p-Mode g-Mode Instability in Coalescing Binary Neutron Stars”. In: *The Astrophysical Journal* 819.2, p. 109.

Weinberg, Nevin N, Phil Arras, and Joshua Burkart (2013). “An Instability due to the Nonlinear Coupling of p-modes to g-modes: Implications for Coalescing Neutron Star Binaries”. In: *The Astrophysical Journal* 769.2, p. 121.

Weinberg, Nevin N, Phil Arras, Eliot Quataert, and Josh Burkart (2012). “Nonlinear Tides in Close Binary Systems”. In: *The Astrophysical Journal* 751.2, p. 136.

Wells, Andrew J and M Grae Worster (2008). “A geophysical-scale model of vertical natural convection boundary layers”. In: *Journal of Fluid Mechnanics* 609, pp. 111–137.