Simulation of the Navier-Stokes Equations in Three Dimensions with a Spectral Collocation Method

by

Christopher Subich

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Applied Mathematics

Waterloo, Ontario, Canada, 2011

© Christopher Subich 2011

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

This work develops a nonlinear, three-dimensional spectral collocation method for the simulation of the incompressible Navier-Stokes equations for geophysical and environmental flows. These flows are often driven by the interaction of stratified fluid with topography, which is accurately accounted for in this model using a mapped coordinate system. The spectral collocation method used here evaluates derivatives with a Fourier trigonometric or Chebyshev polynomial expansion as appropriate, and it evaluates the nonlinear terms directly on a collocated grid. The coordinate mapping renders ineffective fast solution methods that rely on separation of variables, so to avoid prohibitively expensive matrix solves this work develops a low-order finite-difference preconditioner for the implicit solution steps. This finite-difference preconditioner is itself too expensive to apply directly, so it is solved approximately with a geometric multigrid method, using semicoarsening and line relaxation to ensure convergence with locally anisotropic grids. The model is discretized in time with a third-order method developed to allow variable timesteps. This multi-step method explicitly evaluates advective terms and implicitly evaluates pressure and viscous terms. The model's accuracy is demonstrated with several test cases: growth rates of Kelvin-Helmholtz billows, the interaction of a translating dipole with no-slip boundaries, and the generation of internal waves via topographic interaction. These test cases also illustrate the model's use from a high-level programming perspective. Additionally, the results of several large-scale simulations are discussed: the three-dimensional dipole/wall interaction, the evolution of internal waves with shear instabilities, and the stability of the bottom boundary layer beneath internal waves. Finally, possible future developments are discussed to extend the model's capabilities and optimize its performance within the limits of the underlying numerical algorithms.

Acknowledgements

Where to begin? Although my name is on the front cover, this work owes a great deal to everyone who has offered their support during my seemingly never-ending tenure as a graduate student. To begin, I wouldn't even be here without the support of my family growing up. From the time I was a toddler just learning to read, my parents always encouraged me to learn and think, and they were more patient than most with my unusual, geeky childhood.

I arrived in Waterloo in 2005 for a Master's degree, but my eyes were bigger than my calendar when it came to picking a project. This work took a *little* longer than two years to finish, and even still it is little less ambitious than I would have wanted at the start. Years' worth of thanks go to my supervisors, Kevin Lamb and Marek Stastna for having patience with code development when others wouldn't and also for pushing me to finally finish when it was time.

Without question, the inspiration for this goes to Lloyd Trefethen and John Boyd (who was also kind enough to serve on my examining committee) for their books on spectral methods. Both of these books took a complicated topic, entirely new to me, and made it simple and *intuitive*. The idea that spectral methods could be easy encouraged me to try something this large.

Francis Poulin and Timothy Rees (who will be Dr. Rees by the time this is out in print) were the first independent beta-testers of my code, and their feedback helped me a great deal in development. Sharing a work environment definitely motivated me to provide bug-free code!

The administrtive staff of the Applied Mathematics department, especially Helen Warren, does an amazing and thankless job keeping all of the paperwork straight and things running smoothly, and I would have had a far harder time navigating the university bureaucracy if not for their hard work.

Last but by no means least, words cannot express the full depth of gratitude and happiness I have for my lovely wife Diana Papiz. She is my partner in all things, and having her by my side for these years has made them a pleasure.

Table of Contents

List of Figures viii

1 Introduction 1

1.1 Equations of fluid motion . . . 1

1.1.1 Boussinesq approximation . . . 2

1.2 Internal wave dynamics . . . 3

1.3 Existing numerical models . . . 4

1.4 Organization and Algorithm Overview . . . 5

2 Discretization & Methods 7

2.1 Time discretization . . . 8

2.1.1 The pressure problem . . . 9

2.1.2 Algorithm overview . . . 10

2.1.3 Boundary Conditions . . . 11

2.2 Third-order timestepping . . . 13

2.2.1 The full projection algorithm . . . 14

2.2.2 Accuracy and stability . . . 15

2.2.3 Physical interpretation . . . 20

2.2.4 Variable timesteps . . . 22

2.2.5 Startup . . . 24

2.3 Spatial discretization . . . 28

2.3.1 Interpolation . . . 30

2.3.2 Fourier methods 36
2.3.3 Polynomial methods 44
2.3.4 Grid mapping 51
2.4 Preconditioning & Multigrid 57
2.4.1 GMRES 57
2.4.2 Finite difference preconditioning 59
2.4.3 Multigrid 64

3 Code documentation and case studies **85**
3.1 A trivial example 85
3.1.1 A brief interlude on Blitz++ 87
3.2 Kelvin-Helmholtz billows 89
3.2.1 Problem parameters 90
3.2.2 Forcing and initialization 92
3.2.3 Data analysis and output 95
3.2.4 Accuracy 98
3.3 Parallel processing 99
3.3.1 Allocation 99
3.3.2 Analysis routines 100
3.4 The translating dipole and boundary interaction 101
3.4.1 Initialization 103
3.4.2 Analysis and the gradient operator 104
3.4.3 Convergence 107
3.5 Spectral filtering 109
3.5.1 Hyperviscosity 111
3.6 Mapped grids and internal wave generation 113
3.6.1 Physical configuration 113
3.6.2 Grid generation 114
3.6.3 Initialization and forcing 116
3.6.4 Analysis and results 118

4 Research and Future Work **121**
4.1 Three dimensional dipole/wall interaction . . . 121
4.2 Internal waves with shear instabilities . . . 123
4.3 Boundary layer instability under internal waves . . . 127
4.3.1 Background shear . . . 132
4.3.2 Three dimensional evolution . . . 133
4.4 Future work . . . 137
4.4.1 Two-dimensional topography and the no-slip cube . . . 137
4.4.2 General boundary conditions . . . 138
4.4.3 Lagrangian particles . . . 140
4.4.4 Sediment tracers . . . 141
4.4.5 Anelastic equations . . . 141
4.4.6 Numerical & IO optimization . . . 142
4.4.7 Available potential energy . . . 143
4.5 Conclusions . . . 144

Appendix **145**

A Source listings **145**
A.1 Minimal code . . . 145
A.2 Kelvin-Helmholtz Billows . . . 147
A.3 Dipole-wall interaction . . . 152
A.4 Internal wave generation by topograhpy . . . 159

References **165**

List of Figures

2.1 Stability contours for the first-order (dashed), second-order (solid), and third-order (shaded) multi-step methods. Along the contours, the maximum root of (2.26) has magnitude one and the method is neutrally stable. The first and second-order methods are *not* stable along $R = 0$, and so are unsuitable for strictly advective flows. The graphs are symmetric along the imaginary parameter, so only the upper half-plane is shown for simplicity. 19

2.2 Two of the cardinal polynomials from the third-order method, at $t = -1$ (solid) and $t = 1$ (dotted). The other two polynomials (at $t = 0$ and $t = -2$) are mirror images of those shown, reflected about $t = -0.5$. A cardinal polynomial is that given by (2.30) when f^j is nonzero (one) at a single grid point. 21

2.3 Stability contours for the variable-timestep extension, using the method described in the text. Depicted are the minimum zero-stability contours of 100 random realizations, with timesteps of $[0, 1+\varepsilon_1, 2+\varepsilon_2, 3]$, where $\varepsilon_{1,2}$ varied uniformly between ± 0.1 (solid) and ± 0.25 (dashed). The method is stable with evenly spaced timesteps ($[0, 1, 2, 3]$) in the shaded region. 24

2.4 Accuracy of the variable-timestep startup method, as described in the text. The model ODE was solved to $t = 25$ with a maximum Δt of 1, with the initial condition $f^0 = 1$. Shown are accuracy contours of 10^{-1}, 10^{-2}, 10^{-3}, and 10^{-4} from outside and inside for $\Delta t_{min} = 1$ startup (solid), $\Delta t_{min} = \frac{1}{4}$ (dashed), and the full multi-step method (shaded). The multi-step method was initialized with the exact solution for f^{-1} and f^{-2}. Along the imaginary axis, the exact solution is purely advective and the contoured regions there reflect coincidental phase-matching near the stability limit of the multi-step scheme. 26

2.5 Stability of the variable-timestep startup method. The model ODE was solved to $t = 4$ with a maximum Δt of 1, using the initial condition $f^0 = 1$. The final time was chosen as the first timestep to cover only a single Δt (1) after startup. Plotted is the stability contour ($|f| = 1$) for $\varepsilon = 1$ (solid) and $\varepsilon = \frac{1}{4}$ (dashed), the latter of which is stable for purely advective motion. The shaded region is the stability region for the three-level method, reproduced from figure 2.1. 27

2.6 Cardinal functions for second-order (dashed) and tenth-order (solid) polynomials, using a centered interpolation stencil of the proper order. Even with high order, the cardinal functions have internal discontinuities, caused by the stencil-switches halfway between each grid point. 32

2.7 The cardinal function with continuous second derivative implied by the compact, second-order finite difference stencils (solid) along with its first (dashed) and second (dash-dotted) derivatives. The derivatives are continuous up to second order, and they reproduce the standard three-point finite difference stencils at the integer grid points. 33

2.8 Error in interpolating $\exp(-x^2)$ for the piecewise quadratic interpolant (solid, top), C^2 finite difference interpolant (dash-dotted), piecewise sixth-order interpolant (dashed), and sinc interpolant (solid, bottom). Also included are Δx^3 and Δx^7 asymptotic lines. 36

2.9 Aliasing error, with $\Delta x = .1$. The high-frequency wave ($k = 14\pi$, solid) is not well-resolved on the grid (points), and its discrete representation is the same as the lower-frequency wave ($k = -8\pi$, dashed). 38

2.10 Possible two-gridpoint layouts for $-\cos(\pi x)$ (solid) and its derivative (dashed). With gridpoints at the ends of the interval (bullets), the derivative is numerically zero (dash-dotted). Placing the gridpoints at $\frac{\Delta x}{2} + n\Delta x$ (crosses) does not have this problem. 43

2.11 Polynomial interpolation of $(1+9x^2)^{-1}$ (thin, solid) using an equally spaced grid at sixth order (thick, solid) and twelfth order (dashed). While the interpolation is good in the central part of the domain, the interpolant oscillates wildly near the ends. These oscillations become worse at higher order. 45

2.12 Illustration of the grid mapping $x = \cos(\theta)$, for $N = 21$ points. Although the points are equally spaced around the semicircle $z = \sqrt{1-x^2}$, when projected onto the x-axis points cluster near the boundaries, with spacing proportional to N^{-2}, compared to N^{-1} in the middle of the domain. Reproduced from Boyd [2001]. 46

2.13 $(1+9x^2)^{-1}$ under the mapping $x = \cos(\theta)$ (top), along with the magnitude of the even coefficients in its cosine series expansion (bottom, log scale). The odd coefficients are zero due to symmetry, and the even coefficients decay exponentially in magnitude, reaching the level of rounding error after $k = 120$. 47

2.14 Polynomial interpolation as in figure 2.11 on the nonuniform grid $x_j = \cos(\pi \frac{j}{N-1})$ at sixth order (solid) and twelfth order (dashed). Unlike on the equispaced grid, the approximations quickly converge to the true function. 48

2.15 Accuracy of the lowest eigenvalue solution to $u_{xx} = -\lambda u$ ($\lambda = \frac{pi^2}{4}$) when the problem is discretized on N points with the Chebyshev differentiation matrix (2.67) (solid), second order finite differences on the same grid (dashed), and second order finite differences on a uniform grid (dot-dashed). The thin line is $\mathrm{O}(N^{-2})$ convergence. The Chebyshev differntiation method converges much more quickly, with the error dominated by roundoff after $N = 15$ points. 50

2.16 Nonzero structure of the discrete Laplacian defined by (2.69) on a 20×10 grid with Dirichlet boundary conditions (top, with nonzero entries as a dot and zero entries blank) and the resulting tensor product grid (bottom). Grid points cluster near the edges and especially corners. The matrix is sparse and structured, with central 10×10 blocks on the diagonal and nonzero entries on 19 non-central diagonals. 52

2.17 Maximum eigenvalue of the 1D discrete Laplacian operator with Dirichlet boundary conditions, as a function of the number of grid points (N). The Chebyshev (solid) expansion and the 3-point finite difference operator on the same grid (dashed) give N^4 scaling (top thin line); equispaced points (dot-dashed) show only $\mathrm{O}(N^2)$ growth (bottom thin line). 58

2.18 Convergence of GMRES for the two-dimensional problem $\nabla^2 u = f$ with zero Dirichlet boundary conditions for a random vector f. The x-axis is scaled by $(N-2)^2$ (the number of interior grid points) for a 16×16 grid (solid), 32×32 (dashed), and 64×64 (dot-dashed). 60

2.19 Convergence of GMRES for the same 2D problem as figure 2.18, save that the finite difference operator given by the standard three-point stencil is used as a preconditioner. Unlike the unpreconditioned iteration, convergence is extremely rapid and proceeds at a nearly identical rate for the 16×16 (solid), 32×32 (dashed), and 64×64 (dot-dashed) grids. In MATLAB, using the matrix forms of the operator, solution of the 64×64 case took only 1% of the unpreconditioned time. 61

2.20 Schematic for splitting a computational array among four processor nodes. By default, the array has a number of contiguous y/z planes belonging to each individual processor (top), and Fourier-type transformations can be taken along those dimensions. When a transformation along the x dimension is needed, the array is transposed (bottom) so that each processor has contiguous x/y planes. . . 63

2.21 Nonzero entries of the Red-Black Gauss-Seidel update matrix (2.85) for $N = 29$ interior points and the three-point finite-difference second derivative operator. Entries included from the full operator are filled circles; neglected entries are hollow circles. Even grid points (rows) have only the diagonal entries and can be computed independently in parallel, while odd points can then be computed based on the neighbouring even values. 69

2.22 The semicorsening/line-smoothing problem applied to a 13×13 cosine grid, mapped to include a small hill (thick line) at the bottom of the domain. At the smoothing steps, each vertical line is smoothed in its entirety (line smoothing), and the points marked with squares are kept on the coarser level (semicoarsening). The coarsening and interpolation operators are those of (2.79) and (2.82), applied along horizontal grid lines. 72

2.23 Eigenvalues (top) of $L_{fd}P^{-1}$, where L is the 63×63 finite-difference discretization of $\nabla^2 u = f$ with Dirichlet boundary conditions included in the operator and P^{-1} is the two-grid iteration (2.90) with one pass of red-black line smoothing before and after the coarse-grid solve, along with the convergence history (residual error, bottom) of the direct iteration $u_n = u_{n-1} + P^{-1}(f - L_{fd}u_{n-1})$. with a random initial f. 75

2.24 Eigenvalues (top) of $L_{cheb}P^{-1}$, where L is the Chebyshev-based discretization of the operator in figure 2.23 and P^{-1} is the same two-grid operator, along with the GMRES convergence history (bottom) of the two-grid operator used as a preconditioner. 76

2.25 Illustration of the multigrid V-cycle, for $n = 5$ levels in the multigrid hierarchy. The cycle descends to the coarsest grid once, performs a direct solve (open circle), and ascends the hierarchy back to the finest level. 79

2.26 Illustration of the multigrid W-cycle, for $n = 3$ levels in the multigrid hierarchy. Unlike the V-cycle (figure 2.25), the W-cycle visits the coarser grids twice, recursively. As a result, there are many more direct solves on the coarsest grid (open circles). 80

2.27 Illustration of the multigrid F-cycle, for $n = 5$ levels in the multigrid hierarchy. The F-cycle visits the coarser grids twice like the W-cycle (figure 2.26), but unlike the W-cycle only the first visit is recursive; the second traversal is a V-cycle (figure 2.25). The F-cycle still has several direct solves on the coarsest grid (open circles), but the number increases linearly with depth rather than exponentially. . 81

3.1 Departure from the computed growth rate for Kelvin-Helmholtz billow formation in time-dependent simulations for 32 (top), 64, 128, and 192 (bottom, dashed) points in the vertical. For early times, the error is dominated by contribution from secondary modes. The net error reduces to a resolution-dependent level, before again increasing due to nonlinear effects. The exact value was computed with polyeig in MATLAB, and itself is accurate to approximately the 10^{-6} level due to rounding effects. 99

3.2 Initial conditions for the dipole/wall interaction. 103

3.3 The dipole-boundary interaction at $t = 0.38$, near the time of greatest enstrophy, for a 1024×1024 grid. The dipole induces opposing vorticity at the boundary, which begins to wrap around the primary dipole. The boundary vorticity is locally more intense than the primary dipole pair. 107

3.4 The time-evolution of kinetic energy (top) and enstrophy (bottom) for the dipole-boundary interaction at 64×64 (dot-dashed), 128×128 (dashed), and 1024×1024 (dot-dashed) grid points. The intermediate grid sizes of 256×256 and 512×512 are not distinguishable on this graph from 1024×1024 108

3.5 Spectra (top, log scale) and kernel functions (bottom) for the exponential filter with cutoff (solid, with cutoff of 60% of the Nyquist frequency and second-order falloff), fourth-order hyperviscosity (dashed), and for comparison second-order, positivity-preserving viscosity (thin, solid). 110

3.6 Comparison of the exponential filter and hyperviscosity for the Kelvin-Helmholtz billow case of section 3.2 at 256 vertical points. At top, the total density field for the exponential filter (left) and hyperviscosity filter (right). At bottom, a slice through the $z = 0.5$ level, for $-1 < x < 1$, comparing the hyperviscosity (thick, solid) and exponential filter (dashed) with the 1024 vertical-level case (thin, solid). 112

3.7 Internal waves generated from the interaction of tidal flow with topography, shown through the perturbation salinity $S(t) - S(0)$. After one tidal period (top), mode-one waves have propagated some distance away from the hill. After four tidal periods (bottom), the contribution from higher-mode waves begins forming beams. 114

3.8 Convergence of the internal-wave case after one tidal period, with 256×8 resolution (dot-dashed), 512×16 (dashed), 1024×32 (solid, thick), and 2048×64 (solid, thin). The top view is detail of surface velocity ($u(z = 0)$) near the topography, and the bottom is the amplitude of the spectrum of the surface velocity. . . . 118

3.9 Comparison of surface velocities (top) and u-velocity profile at the middle of the domain (bottom) between this work at 2048×64 (dashed) and the model of Lamb [1994] at 4096×258 (solid) at one tidal period. The curves lie nearly on top of one another. 119

3.10 Space-time plot of surface velocities (mean removed) for the 2048×64 case. The wave crests are clearly visible, and they move at the phase speed predicted by linear analysis (dashed lines). After 3 tidal periods, waves that exit on one side of the domain re-enter on the other and begin interacting. 120

4.1 Enstrophy (top) and kinetic energy (bottom) of the three-dimensional dipole/wall interaction (solid line), compared with the two-dimensional equivalent (dashed). The three-dimensional interaction undergoes a second production of enstrophy (from three-dimensional effects) after the primary dipole-wall interaction. This is associated with an increase in energy dissipation. 122

4.2 Isosurface plot of vorticity for the three-dimensional dipole-wall interaction, with the surface at $|\omega| = 50$ shaded by the orientation of vorticity, with lighter being more spanwise-directed. The slices at the front and back planes plot ω_y, vorticity in the direction of the original axis. 123

4.3 Table of initial wave conditions. Waves 1, 2, and 3 are tall, relatively thin waves with crests that are well above the mid-depth of $-50\,\mathrm{m}$, while waves 4a-4c are broad waves with crests that nearly reach the mid-depth. $\min(\mathrm{Ri})$ is the minimum Richardson number in the pycnocline at the centre of the wave, and ω is the frequency of the oscillation with fastest spatial growth rate (k_i). 124

4.4 Visualization of the pycnocline centres of the waves from table 4.3. Top: waves 1a (solid), 2 (dashed), and 3 (dot-dashed). Bottom: waves 4a (solid), 4b (dashed), and 4c (dot-dashed). 125

4.5 Energy extraction over time (log scale) for (top) waves 1a (long dashes), 1b (bottom solid), 2 (top solid), and 3 (short dashes) and (bottom) waves 4a, 4b, and 4c (from bottom to top) 126

4.6 Density profile (top) of the three-dimensional simulation of wave 4a in the x-z plane at time $2650\,\mathrm{s}$, along with the three-dimensional kinetic energy fraction (bottom) $\frac{u^2+v^2+w^2}{\langle u\rangle_y^2+\langle w\rangle_y^2}$ (darker is more three-dimensional). The boxed region is visualized in figure 4.7, and the contours visualize the wave pycnocline at $t = 0$, before the production of any billows. 127

4.7 Three-dimensional view of the simulation of wave 4a, showing the $\rho = 1$ isosurface shaded by the spanwise velocity in the tail of the wave. The three-dimensional evolution of the billows is first characterized by production of spanwise velocity, which then affects the three-dimensional shape of density isosurfaces. The background velocity is in the positive x direction, so the rightmost billows have had the longest time to evolve. 128

4.8 Vorticity in the wave field (top, with contours showing the pycnocline) and bottom boundary (bottom) for the large-wave case ($a = 2$ in (4.4)) at $t = 500$, high-pass filtered to remove the large-scale vorticity caused by the wave motion. The short-scale motion in the pycnocline is fairly strong with localized regions of overturning, but the short-wavelength vorticity at the boundary is significantly stronger. 129

4.9 Evolution of the boundary vorticity of the case in figure 4.8 in a selected region at $t = 300$ (a), 400 (b), 450 (c), and 500 (d). Layers of opposite-signed vorticity overlie each other, leading to bursts of vorticity away from he boundary. 130

4.10 As figure 4.8, for waves of half the amplitude ($a = 1$). The smaller waves interact more weakly, and they do not generate shorter waves in the pycnocline. Short-scale vorticity generation at the boundary is suppressed, and does not result in strong billows. 131

4.11 As figure 4.10, with the addition of a background current ($u_0 = -0.2$) at the beginning of the simulation. This quickly adjusts to the no-slip boundaries and forms a thin boundary layer, which provides a source of vorticity for billow-type instabilities at the boundary. 132

4.12 The central pycnocline (top) and bottom boundary (bottom) density for a case similar to 4.11 with the addition of a small secondary stratification at the bottom boundary. The background shear still provides a mechanism for instability, but the resulting billows remain lower in the water column. 133

4.13 $\int v(t)^2 \mathrm{d}V$ for the three-dimensional wavefield-boundary interaction with background shear. The integrated v^2 initially decays from the white-noise perturbed initial conditions, but as the flow evolves the billows at the bottom boundary undergo three-dimensional undulations, causing an increase in v^2. The dot is placed at $t = 750$, which is pictured in figure 4.14. 134

4.14 Root mean square spanwise velocity $(\langle v^2 \rangle_y)^{1/2}$ of the three-dimensional interaction at time $t = 750$, a local maximum of $\int v^2 \mathrm{d}V$. At top, the full field (x-z plane) with contours visualizing the pycnocline. The spanwise velocity is localized to a small region of the bottom, viewed in detail in the bottom panel. 135

4.15 Bottom stress $\frac{\partial u}{\partial z}$ (top) and $\frac{\partial v}{\partial z}$ (bottom) for the region shown in detail in figure 4.14 at time $t = 750$. $\frac{\partial u}{\partial z}$ is dominated by the background and wave-induced currents, but still has significant modulation from the three-dimensional character of the instability. $\frac{\partial v}{\partial z}$ is controlled entirely by three-dimensionalization. 136

Chapter 1

Introduction

The field of computational fluid dynamics has always been limited by available computational resources. Until comparatively recently, this restricted simulations of well-resolved, nonlinear dynamics to two dimensions. Two dimensional flows are much easier to study than their three-dimensional counterparts, but nature is rarely so accommodating. Two-dimensional dynamics describe only a small subset of geophysical and environmental flows, and most notably three-dimensional dynamics are necessary for the development of turbulence [Kundu and Cohen, 2004].

Taking advantage of modern, high-performance computers, this work proposes and develops a three-dimensional, spectral collocation method for the simulation of the Navier-Stokes equations with the Boussinesq approximation. Particular care is taken to ensure the model is appropriate for the simulation of large, internal waves at lake and ocean scales, including their interaction with topographic features. The accurate inclusion of topographic interactions is a key feature, having an immense impact on algorithm design.

1.1 Equations of fluid motion

For Newtonian fluids, which have a linear relationship between stress and strain that depends only on local properties of the fluid, the momentum of the fluid is given in indicial notation (with summation of repeated indices) by the Navier-Stokes equations [Kundu and Cohen, 2004, eqn. 4.44]:

$$\rho \frac{\mathrm{D}u_i}{\mathrm{D}t} = -\frac{\partial p}{\partial x_i} + \rho g_i + \frac{\partial}{\partial x_j}\left(\mu\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) - \tfrac{2}{3}\mu\frac{\partial u_k}{\partial x_k}\delta_{ij}\right), \tag{1.1}$$

where δ_{ij} is the Kronecker delta, u_i is the ith component of velocity, x_i is the ith dimension, g_i is the ith component of the gravity vector, p is pressure, ρ is density, μ is the coefficient of

viscosity (which may depend on local fluid properties such as temperature), and $\frac{\mathrm{D}}{\mathrm{D}t} = \frac{\partial}{\partial t} + u_i \frac{\partial}{\partial x_i}$ is the material derivative, which is the derivative following the local motion of the fluid. This is combined with the conservation of mass:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \vec{u}) = 0, \tag{1.2}$$

which simply states that the change of density with time is proportional to the net mass flux out of the local fluid parcel. Using the chain rule and rewriting (1.2) with the material derivative gives:

$$\frac{1}{\rho} \frac{\mathrm{D}\rho}{\mathrm{D}t} + \nabla \cdot \vec{u} = 0. \tag{1.3}$$

The combination of (1.1) and (1.3) is not a complete system; it needs an equation of state to relate density, pressure, and internal energy (the equation for which is neglected here). This equation is medium-dependent.

1.1.1 Boussinesq approximation

In the case of geophysical and environmental flows of water, the density varies only slightly in the domain, affected mostly by temperature and salinity variations. Furthermore, these flows are nearly incompressible – typical wave speeds are much less than the speed of sound. Thus, it is fair to expand (1.1) and (1.3) about a background density, with $\rho = \rho_0 + \rho'$, with $\rho' \ll \rho_0$. This gives for (1.3):

$$\nabla \cdot \vec{u} = -\frac{1}{\rho_0 + \rho'} \frac{\mathrm{D}\rho'}{\mathrm{D}t} \approx 0, \tag{1.4}$$

which implies that the fluid is approximately incompressible (with zero divergence). This simplifies the form of (1.1), which can be written in vector form as:

$$\rho \frac{\mathrm{D}\vec{u}}{\mathrm{D}t} = -\nabla p + \rho \vec{g} + \mu \nabla^2 \vec{u}. \tag{1.5}$$

Now, if the flow is still and uniform density ($\rho = \rho_0$), then the only significant terms in (1.5) are the pressure and gravity terms. Assuming the gravity vector points in the $-z$ direction, $\nabla p = \rho_0 \vec{g}$ implies that $p_0 = -\rho_0 g z$ is a background pressure field that cancels the contribution from the background density[1]. Expanding pressure as $p_0 + p'$ in (1.5) gives:

$$\rho \frac{\mathrm{D}\vec{u}}{\mathrm{D}t} = -\nabla p' + \rho' \vec{g} + \mu \nabla^2 \vec{u}. \tag{1.6}$$

[1] In incompressible fluid mechanics, pressure is only defined up to a constant factor, as it enters the equations of motion only through its gradient.

Now, expanding ρ on the left-hand side of (1.6) and dividing by the background density gives:

$$\left(1+\frac{\rho'}{\rho_0}\right)\frac{\mathrm{D}\vec{u}}{\mathrm{D}t} = -\frac{1}{\rho_0}\nabla p' + \frac{\rho'}{\rho_0}\vec{g} + \nu\nabla^2\vec{u}, \tag{1.7}$$

where ν is the kinematic viscosity ($\rho_0^{-1}\mu$). The $\frac{\rho'}{\rho_0}$ term on the left-hand side is much less than one (by assumption) and can be neglected[2], giving:

$$\frac{\mathrm{D}\vec{u}}{\mathrm{D}t} = -\frac{1}{\rho_0}\nabla p' + \frac{\rho'}{\rho_0}\vec{g} + \nu\nabla^2\vec{u}. \tag{1.8}$$

If the fluid is not isothermal, the temperature fluctuations also follow the equation [Kundu and Cohen, 2004]:

$$\frac{\mathrm{D}T}{\mathrm{D}t} = \kappa\nabla^2 T, \tag{1.9}$$

where κ is the thermal diffusivity.

The combination of equations (1.4), (1.8), and (1.9) with a suitable equation of state[3] gives a system suitable for numerical simulation.

1.2 Internal wave dynamics

Through temperature and salinity variation, the ocean is a stratified medium [Cushman-Roisin and Beckers, 2011], and tidal motion provides a continual, periodically-oscillating source of energy[4]. Because the bottom boundary is not flat, the fluid is forced up and down slopes. In the stratified medium, this converts kinetic energy of motion to potential energy, which results in wave motion.

These generated waves can be very large in amplitude and carry significant amounts of energy. For example, off of the Luzon Strait [Warn-Varnas et al., 2010] in the South China Sea, internal waves can be greater than 150 meters in amplitude and carry several gigajoules of energy per meter. These waves are essentially unchanging in form as they propagate, and when they shoal the energy is partially reflected and partially induces mixing [Aghsaee et al., 2010]. Both the generation and shoaling steps depend greatly on the topography [Lamb, 1994], so numerical simulations must accurately model the topographic effects in order to have physical relevance.

[2] Every term on the right-hand side of (1.6) is divided by ρ_0, so the perturbation terms may not be neglected there.

[3] Generally, a linearized equation of state $\rho = \rho_0 + \alpha T$ is sufficient, for appropriate parameters. If temperature variations are large, the small-parameter expansion of the Boussinesq approximation will be inaccurate.

[4] Dynamics in lakes are similar, although lakes have little salinity generation and the primary source of energy for fluid motion is the wind.

1.3 Existing numerical models

There are a great variety of numerical models for the simulation of the Navier-Stokes equations, and a comprehensive review is far beyond the scope of even this thesis. However, select models provide a reasonable overview of how this kind of fluid problem has previously been simulated.

At one extreme, low-order methods such as SUNTANS [Fringer et al., 2006] discretize entire large-scale regions with full, measured topographical features with fixed vertical levels[5]. Approaches such as this generally use energy-conserving discretizations for velocity advection, which have the convenient property of diffusing fronts that cannot properly be resolved on the grid. Another approach is to use a coordinate mapping method to directly, exactly map the topography to the boundary of the domain. This is the technique used in the two-dimensional model of Lamb [1994], which also uses a finite-volume method to ensure conservation. Even when the boundary is exactly discretized with a mapping, however, low-order methods still require many gridpoints per characteristic wavelength of the flow to ensure accuracy. This may not be a problem when studying very large-scale features or running simulations restricted to two dimensions (where increased resolution is less of an issue), but the combination can lead to prohibitive memory usage.

At the other extreme, high-order spectral methods seek to simulate fluid motion extremely accurately by using trigonometric (Fourier) or polynomial functions to evaluate derivatives. In comparison to finite difference or finite volume methods, in spectral methods any particular grid point will influence the rest of the domain; these are effectively global methods. Consequently, the computational cost per gridpoint is greater than that of low-order methods, but ideally the cost is recovered in needing fewer gridpoints in total. One example of such a method is that of Diamessis et al. [2005], which uses a Fourier expansion in the x and y dimensions along with layered Chebyshev polynomials in the z dimension to simulate three-dimensional stratified turbulence. Other permutations are possible, such as in the model of Winters et al. [2004], which uses an elegant formulation of the Navier-Stokes equations in terms of Fourier series. A more detailed discussion of the application of spectral methods to the Navier-Stokes equation is in Peyret [2002]. In general, high-order spectral methods (and mixed spectra/finite-difference methods) have been an important tool in turbulence studies; see Moin and Mahesh [1998] for a review.

These high-order methods have a common problem, however; directly implementing topography is quite difficult. Unlike low-order methods, global spectral approaches rely on a structured grid defined on the entire domain[6]. Simply truncating the discretization to interior points is ef-

[5]SUNTANS uses a triangular prism grid, whereas other models such as MITgcm [Adcroft et al., 2011] may use rectangular grids.

[6]High-order finite element methods such as those described in Karniadakis and Sherwin [2005] are a compromise between locality and order that may possibly bridge this gap. Implementation is difficult, however, with one of the best candidates developed so far being GASpAR [Rosenberg et al., 2006].

fectively impossible, with the closest approach being penalization methods [Keetels et al., 2007], which damps the governing equations outside of the domain. This adjustment, done in a smooth manner at the boundary, creates a thin layer at the boundary where the governing equations are not exactly satisfied.

This work applies a coordinate mapping to the domain, so that the spectral expansion is applied to a "computational box". Mapping methods such as this complicate the solution procedure, and many of the simplifications that other spectral methods have used will not apply. One model that comes close to this approach is that of Tsai and Hung [2007], which uses mapping to directly simulate flows with a free surface. Even here, however, the vertical coordinate is expanded with a second-order finite difference stencil, reducing overall accuracy for the sake of computational simplicity.

1.4 Organization and Algorithm Overview

This work is divided into two main parts, separating the numerical methods used from the implementation and use of the resulting code.

Chapter. 2 covers the numerical methods used in this work, beginning in section 2.1 with an overview of the time discretization. This is done with a third-order multi-step method based on backwards-differentiation (section 2.2, with stability discussed in 2.2.2) which forms an implicit equation at each timestep for computing pressure (section 2.1.1). At startup (section 2.2.5), small timesteps are taken with a lower-order method to initialize the multi-step process.

The implicit equations generated in timestepping are discretized with the methods discussed in section 2.3. A mixture of Fourier expansions (section 2.3.2) and Chebyshev polynomials (section 2.3.3) are used as needed based on the underlying grid. When a domain with topography is used, the grid is fitted to the topography through a coordinate mapping discussed in section 2.3.4.

These resulting discrete problems must be solved iteratively, and section 2.4 discusses the techniques used. This work uses the generalized minimum residual method [Saad, 1993], discussed in section 2.4.1 as the iterative framework, but the Poisson and Helmholtz equations must still be preconditioned with a low-order, finite difference preconditioner (section 2.4.2) for acceptable convergence. Direct solvers for the finite preconditioner are still not efficient enough however, and this work uses a geometric multigrid method (section 2.4.3) to efficiently solve the preconditioning problem.

Chapter. 3 documents how the developed code can be used to actually compute flow evolution, including the idiosyncrasies of the resulting implementation; the cases are also analyzed to show numerical convergence. Section 3.1 begins with the trivial case that does nothing, in order

to show proper initialization and shutdown methods. Section 3.2 computes Kelvin-Helmholtz billows in a shear layer, including computation of the linear growth rates. Section 3.3 discusses the initialization and execution of the code in parallel, including pitfalls related to analysis that needs global quantities. Section 3.4 replicates the work of Clercx and Bruneau [2006] and Kramer et al. [2007] to look at the interaction of a translating dipole with a no-slip boundary, and section 3.5 discusses the spectral filtering algorithms used to help ensue stability in under-resolved simulations. Finally, section 3.6 demonstrates internal wave generation over topography, using a coordinate-mapped grid to include a large hill in the middle of the domain.

Chapter. 4 discusses the results of several medium to large-scale studies undertaken with this code during its development. Section 4.1 looks at the three-dimensional equivalent to the case of section 3.4, where the translating dipole is subject to a three-dimensional elliptic instability that complicates the interaction with the boundary. Section 4.2 looks at the evolution of shear instabilities in the central regions of several large waves, which result in the formation of Kelvin-Helmholtz billows that extract energy from the wave and propagate downstream in a reference frame moving with the internal wave. Section 4.3 considers the stability of the boundary layer underneath internal waves of moderate size, discussing the conditions necessary for the generation of billow-type instabilities at the boundary, including for a select case its three-dimensional evolution.

Finally, section 4.4 discuses several modifications that could be made to this code to improve its generality and efficiency, and 4.5 contains concluding remarks.

Chapter 2

Discretization & Methods

The numerical methods in this work are defined by how they discretize the Navier-Stokes equations in space and time. Restating equations (1.8), (1.4), and (1.9) from Chapter 1 and adding a forcing term $\vec{F}(x,y,z,t)$ to the momentum equations, the Navier-Stokes equations under the Boussinesq approximation are:

$$\frac{\mathrm{D}\vec{u}}{\mathrm{D}t} = \frac{\partial \vec{u}}{\partial t} + \vec{u}\cdot\nabla\vec{u} = -\frac{1}{\rho_0}\nabla p' + \frac{\rho'}{\rho_0}\vec{g} + \nu\nabla^2\vec{u} + \frac{1}{\rho_0}\vec{F} \tag{2.1a}$$

$$\nabla\cdot\vec{u} = 0 \tag{2.1b}$$

$$\frac{\mathrm{D}T}{\mathrm{D}t} = \frac{\partial T}{\partial t} + \vec{u}\cdot\nabla T = \kappa\nabla^2 T, \tag{2.1c}$$

where $\vec{F}$ includes all of the momentum-forcing relevant to the particular problem, including Coriolis terms that arise from a rotating reference frame. Additionally, the system (2.1) includes the medium-dependent equation of state $\rho(T)$ for density as a function of temperature; this could easily be extended to a more variables ($\rho(T,S)$, for example, to include salinity) with the addition of more equations like (2.1c) for the other constituents. Also, for simplicity the factor ρ_0 in (2.1) can be included in the perturbation density (ρ'), perturbation pressure (p'), and forcing terms. This is equivalent to partially nondimensionalizing the system to set the background density as 1.

For this work, the system is discretized with the method of lines, treating the temporal and spatial discretizations independently. The time discretization is discussed in section 2.1, ultimately using a third-order multi-step method for the time marching. The spatial discretization is discussed in section 2.3, along with the properties of the global spectral collocation method used.

2.1 Time discretization

The system (2.1) has two time scales that scale with grid size (Δx): the advective timescale of approximately $\Delta x / |\vec{u}|$ and the diffusive timescale of approximately $\Delta x^2 / \nu$ (or κ for (2.1c)).

Consider a simplified, one-dimensional advection-diffusion problem as schematic for the system (2.1):

$$\frac{\partial f}{\partial t} + u(x)\frac{\partial f}{\partial x} = \nu \frac{\partial^2 f}{\partial x^2}. \tag{2.2}$$

Under the method of lines, the stability of a time-discretized version of (2.2) is governed by the most extreme eigenvalue of the operator $(\nu \frac{\partial^2}{\partial x^2} - u(x)\frac{\partial}{\partial x})$. Assuming further that $u(x)$ is a constant u_0[1], the Fourier transform of (2.2) gives:

$$\frac{\partial \hat{f}}{\partial t} = -(\nu k^2 + \mathbf{i} u_0 k)\hat{f}(k), \tag{2.3}$$

where k is the Fourier wavenumber. Clearly, the eigenvalue $(\nu k^2 - \mathbf{i} u_0 k)$ has largest amplitude when k is large, and on an equispaced grid the largest frequency k is given by the Nyquist frequency $k = \frac{\pi}{\Delta x}$. In the limit of zero viscosity (or alternately very large u_0), this reduces to the strictly imaginary eigenvalues of an advection equation. Additionally, when u_0 is zero, (2.3) reduces to a diffusion equation.

What does this imply for the timestepping? In order to be stable and properly integrate equations (2.1), the time discretization will have to handle eigenvalues with real components on the order of $\frac{-1}{\Delta x^2}$ and imaginary components on the order of $\frac{1}{\Delta x}$. Strictly explicit timestepping schemes require that $\Delta t \times |k|$ be on the order of 1. This consequently gives a rough timestep restriction of $\Delta t = O(\Delta x^2)$, but that is much finer than the advective dynamics call for.

Conversely, an implicit timestepping scheme could allow extremely large timesteps, but that would require solving the advective part of equations (2.1) implicitly as well. In general, this nonlinear solve is not easy, requiring techniques like semi-Lagrangian advection [Staniforth and Côté, 1991] to do an adequate job without excessive damping. Furthermore, this work is designed around simulating dynamically active geophysical flows, which requires a timestep small enough to resolve the active scales. The gains of solving the advective part of fluid motion implicitly are therefore extremely small.

As a compromise, this work uses a mixed explicit-implicit scheme, described in Karnidakis et al. [1991] as "stiffly-stable," which takes advection explicitly and diffusion implicitly. Section 2.2 discusses the details of the full, third-order scheme used in this work. At first-order, this

[1] This holds true, locally, even if $u(x)$ is not constant but instead varies more slowly than the grid scale. In that case, WKB theory allows analysis of the behaviour near a given point x_0, where $u(x)$ may to leading order be interpreted as the constant $u(x_0)$

scheme reduces to a combination of Forwards and Backwards-Euler methods[2], which apply to equation (2.2) as:

$$f(x,t+\Delta t) = f(x,t) + \Delta t \left(-u(x,t)\frac{\partial f(x,t)}{\partial x} + \nu \frac{\partial^2 f(x,t+\Delta t)}{\partial x^2} \right). \tag{2.4}$$

Likewise, the equations of motion (2.1) become:

$$\vec{u}^{n+1} - \nu \nabla^2 \vec{u}^{n+1} = \vec{u}^n + \Delta t \left(-\vec{u}^n \cdot \nabla \vec{u}^n - \nabla p^* + \rho^n \vec{g} + \vec{F}(\vec{u}^n, \rho^n) \right) \tag{2.5a}$$

$$T^{n+1} - \kappa \nabla^2 T^{n+1} = T^n - \Delta t \left(\vec{u^n} \cdot \nabla T^n \right), \tag{2.5b}$$

Subject to

$$\nabla \cdot \vec{u}^{n+1} = 0, \tag{2.5c}$$

where $\vec{u}^n = \vec{u}(x,t)$, $\vec{u}^{n+1} = \vec{u}(x,t+\Delta t)$, and likewise for ρ. Additionally, the prime is dropped from ρ and p, and ρ_0 is incorporated into p and ρ for simplicity.

2.1.1 The pressure problem

Equations (2.5) are not directly suitable for timestepping – there is no evolution equation for the pressure, and the incompressibility condition (2.5c) further complicates direct evolution of the velocity field. The pressure term enforces incompressibility, and the lack of an evolution equation is equivalent to an infinite sound speed in the compressible Navier-Stokes equations.

Taking the curl of the momentum equations (2.5a) would eliminate pressure from the system entirely by transforming the momentum equations into equations for the vorticity $\vec{\omega}$ [Kundu and Cohen, 2004]. This transformation does not simplify the system, however, since finding the velocities $\vec{u}$ from the vorticities $\vec{\omega}$ is equally complicated. Indeed, for viscous problems finding the proper boundary vorticity to ensure no-slip flow at boundaries is an expensive step, using an influence matrix method [Peyret, 2002] that is more expensive than the inverse problem itself[3]

One approach to resolve this pressure problem is to introduce an artificial compressibility [Langtangen et al., 2002]. This would replace the incompressibility condition with a more traditional evolution equation for pressure, which could then be timestepped along with the velocity.

[2] The first-order variant of this scheme is not stable for a purely advective problem ($\nu = 0$), but the third-order variant is stable in the diffusion-free limit. This is discussed further in section 2.2.2.

[3] In two dimensions, an influence matrix would find, for a delta-element of boundary vorticity, its influence on the resulting streamfunction (and its derivative in the boundary-normal direction – the wall-tangent velocity). Applying the inverse of this $N \times N$ matrix is itself an $O(N^2)$-complexity operation. As will be discussed in section 2.3, this work uses matrix-free methods that result in better asymptotic complexity.

Unfortunately, as mentioned above the incompressible Navier-Stokes equations arise in the limit of an *infinite* speed of sound. Good convergence of the artificially-compressible equations to the incompressible limit would require a prohibitively large sound-speed, with a correspondingly small timestep.

Instead, this work uses a physically-motivated splitting of the operator. Taking the divergence of the momentum equations (2.5a) gives:

$$\nabla \cdot \left(\vec{u}^{n+1} - \nu \nabla^2 \vec{u}^{n+1}\right) = \nabla \cdot \vec{u}^n + \Delta t \left(-\nabla \cdot (\vec{u}^n \cdot \nabla \vec{u}^n) - \nabla^2 p^* + \nabla \cdot \rho^n \vec{g} + \nabla \cdot \vec{F}\right). \tag{2.6}$$

Since the divergence and Laplacian operators commute, $\nabla \cdot \nabla^2 \vec{u}^{n+1} = \nabla^2 \nabla \cdot \vec{u}^{n+1}$, and this is identically zero after applying the incompressibility condition (2.1b), eliminating the left-hand side of (2.6). Simplifying gives a Poisson equation to solve for pressure at each timestep:

$$\nabla^2 p^* = -\nabla \cdot (\vec{u}^n \cdot \nabla \vec{u}^n) + \nabla \cdot \rho^n \vec{g} + \nabla \cdot \vec{F} + \Delta t^{-1} \nabla \cdot \vec{u}^n, \tag{2.7}$$

where the right-hand side of the equation depends exclusively on already-known values, removing the dependence of pressure on the unknown $\vec{u}^{n+1}$ values. Including the divergence of the previous velocities ($\nabla \cdot \vec{u}^n$) on the right-hand side is not strictly necessary because (2.5c) applies equally to the previous timestep, so that term should be zero. However, its inclusion helps deal with initial conditions (which may not be divergence-free), rounding error, and incompatible boundary conditions that arise from no-slip conditions.

The elimination of $\nabla \cdot \nabla^2 \vec{u}$ assumes that the divergence ($\nabla \cdot$) and Laplacian (∇^2) commute. This is true for the exact, continuously defined mathematical operators, but it is not true in general for discretized versions thereof. Unless a numerical method takes special care[4], the result of this splitting method may still have numerical divergence. Fortunately, for the high-order operators used in this work, the operators almost exactly commute and the problem is negligible.

2.1.2 Algorithm overview

With (2.7) giving an expression for pressure, the splitting method suggests an algorithm for timestepping. Ignoring boundary conditions and using the operator splitting from above, momentum equations (2.5a) and incompressibility condition (2.5c) can be rewritten as the sequence:

$$\vec{u}^* = \vec{u}^n + \Delta t \left(-\vec{u}^n \cdot \nabla \vec{u}^n + \vec{g} \rho^n + \vec{F}^n\right) \tag{2.8a}$$

$$\Delta t \nabla^2 p^* = \nabla \cdot \vec{u}^* \tag{2.8b}$$

$$\left(1 - \Delta t \nu \nabla^2\right) \vec{u}^{n+1} = \vec{u}^* - \Delta t \nabla p^*, \tag{2.8c}$$

[4]One such approach is that used by Lamb [1994], where the velocity is directly projected onto a basis of divergence-free velocities.

where $\vec{u}^*$ is introduced as a "predicted velocity," which is affected by the advective terms and forcing, without the influence of either pressure or viscosity. This predicted velocity, which corresponds to the terms on the right-hand side of (2.7), is used to compute the proper pressure at each timestep. Finally, the pressure and viscosity are applied via a Helmholtz equation, completing the momentum equations.

This splitting is known as the projection method, after Bell et al. [1991], because the predicted velocities are projected onto a divergence-free subspace (required by (2.5c)). Provided the discrete divergence and Laplacian operators commute, this splitting method is also exact away from boundaries.

2.1.3 Boundary Conditions

The algorithm of (2.8) is unfortunately more complicated in implementation than specification because there are no immediately obvious boundary conditions to make (2.8b) solvable. Ordinarily, pressure is constrained by a free surface (where pressure at the interface must equal the environmental pressure) and the fluid's equation of state. Making the Boussinesq approximation eliminates the effect of pressure on density, however, and without a free surface there are no natural pressure boundary conditions to apply.

Using a staggered grid, where pressure nodes are not collocated with velocity nodes, can eliminate the problem of boundary conditions. In a trivial one-dimensional example, if the velocities were taken to lie on N grid points (including left and right boundaries), a staggered grid would take pressure to lie in the $N-1$ interior spaces, generally at the midpoints of the velocity nodes. The velocity divergence ($\nabla \cdot \vec{u}^*$) would be taken on the staggered grid, and (2.8b) could be applied directly. Unfortunately, the amount of bookkeeping for a staggered grid is staggering[5], especially in multiple dimensions: gradient operations would have to transfer between the pressure and velocity grids, and the Laplacian operations in (2.8b) and (2.8c) would operate on different grids. This work solves (2.8b) on the same grid as velocity, using boundary conditions.

Fortunately, the inviscid case ($\nu = 0$) provides insight into appropriate boundary conditions for the pressure Poisson problem. Without viscosity, (2.8c) becomes trivial, since the Laplacian is multiplied by ν. The equation becomes:

$$\vec{u}^{n+1} = \vec{u}^* - \Delta t \nabla p^*. \tag{2.9}$$

By definition, a solid boundary does not allow fluid to throw through it, so $\hat{\mathbf{n}} \cdot \vec{u} = 0$, where $\hat{\mathbf{n}}$ is the (outward) unit normal vector at the boundary. Taking $\hat{\mathbf{n}} \cdot$ (2.9) at the boundaries gives:

$$\partial_n p^* = \Delta t^{-1} \hat{\mathbf{n}} \cdot \vec{u}^*, \tag{2.10}$$

[5] No pun intended

after the substitution of no normal flow, where ∂_n is the derivative in the outward direction. This provides a Neumann-type boundary condition for pressure, allowing the solution of (2.8b) (up to an arbitrary constant).

While exact for inviscid flows with free-slip boundary conditions (no flow normal to the boundary), (2.10) is inaccurate for viscous flows. While applying (2.8b) with (2.10) results in a divergence-free flow field, subsequently applying the viscosity in (2.8c) can reintroduce divergence in the near-boundary flow.

This arises because (2.9) is only true in the limiting case of zero viscosity. For nonzero viscosity, the more general form of $\hat{\mathbf{n}} \cdot$(2.8c) is:

$$\partial_n p^* = \Delta t^{-1} \hat{\mathbf{n}} \cdot \vec{u}^* + \nu \hat{\mathbf{n}} \cdot \nabla^2 \vec{u}^{n+1}. \tag{2.11}$$

Unfortunately, the velocities at the next time level ($\vec{u}^{n+1}$) do not drop out of the equation, so (2.8b) and (2.8c) remain coupled. Ignoring the additional term still gives a stable algorithm, but the near-boundary pressure gradient is incorrect. Following Karnidakis et al. [1991], taking the divergence of (2.5a), without assuming the incompressibility condition gives:

$$D - \nu \Delta t \nabla^2 D = -\Delta t \nabla^2 p + \nabla \cdot \vec{u}^* \tag{2.12}$$

where $D = \nabla \cdot \vec{u}^{n+1}$ and $\vec{u}^*$ is as defined in (2.8a). Defining pressure according to (2.8b) sets the right-hand side to zero. Now, if on the boundary D is nonzero anywhere, as a result velocity divergence will be nonzero in a boundary layer of thickness $\sqrt{\nu \Delta t}$. This also causes a velocity error of $\mathrm{O}(\Delta t \nu \nabla^2 \vec{u})$, the size of the term dropped from (2.8c).

This coupling between pressure and viscosity is resolvable, at least approximately. Since the projection method is stable, it's possible to iteratively apply (2.8b) and (2.8c), using (2.11) (with the previous iteration's velocity field). As well, the boundary condition can be better-approximated by extrapolating the solenoidal part of the velocity field to the new time-level.

In this approach, the $\nabla^2 \vec{u}^{n+1}$ term in (2.11) is approximated. The standard projection method neglects this term entirely, effectively approximating it with zero. Now, direct approximation of $\vec{u}^{n+1}$ is possible via extrapolation from previous time-levels, but this approach is unstable, and it leads to a growing mode of nonzero divergence [Orszag et al., 1986]. Instead, the method makes use of the incompressibility of $\vec{u}^{n+1}$ to write:

$$\nabla^2 \vec{u}^{n+1} = \nabla \left(\nabla \cdot \vec{u}^{n+1} \right) - \nabla \times \nabla \times \vec{u}^{n+1} = -\nabla \times \nabla \times \vec{u}^{n+1}, \tag{2.13}$$

since $\nabla \cdot \vec{u}^{n+1} = 0$. Using this identity in (2.11) gives:

$$\partial_n p^* = \Delta t^{-1} \hat{\mathbf{n}} \cdot \vec{u}^* + \nu \hat{\mathbf{n}} \cdot \nabla \times \nabla \times \vec{u}^{n+1}. \tag{2.14}$$

Here, $\vec{u}^{n+1}$ can be extrapolated from previous time-levels without introducing an instability. The $\nabla \times \nabla \times$ identity for the Laplacian removes any contribution from nonzero divergence at previous time-levels.

Coefficient	1st Order	2nd Order	3rd Order
α_0	1	3/2	11/6
α_1	-1	-2	-3
α_2		1/2	3/2
α_3			-1/3
β_0	1	2	3
β_1		-1	-3
β_2			1

Table 2.1: Coefficients for the stiffly-stable timestepping method of (2.15), for first through third orders and a constant timestep. Reproduced from Karnidakis et al. [1991].

2.2 Third-order timestepping

This model uses a so-called "stiffly-stable" multi-step method, after Karnidakis et al. [1991]. In these methods, the terms that give the most severe restrictions on timestep are evaluated implicitly, while the other terms have an explicit treatment. In the case of the Navier-Stokes equations, the stiff terms come from viscosity and molecular diffusivity. This work uses a multi-step method to avoid extra evaluations of derivatives, which are fairly expensive in a spectral collocation method such as this one.

Speaking broadly, a multi-step method uses information from previous timesteps ($n-k$ to n) to increase the accuracy of prediction for step $n+1$. In the stiffly-stable multi-step methods, the implicit terms (viscosity and diffusivity) are evaluated only at the $n+1$ level, while explicit terms (nonlinear terms and forcing) are extrapolated to the $n+1$ level from previous timesteps.

The schematic for the family of methods is given by:

$$\Delta t^{-1}\left(\sum_{j=0}^{j=k}\alpha_j\vec{u}^{n+1-j}\right)=\mathbf{L}(\vec{u}^{n+1})+\sum_{j=0}^{j=k-1}\beta_j\mathbf{N}(\vec{u}^{n-j}), \tag{2.15}$$

where $\mathbf{N}$ represents the nonlinear and explicitly-evaluated terms and $\mathbf{L}$ represents the implicitly-evaluated linear terms. At first-order ($k=1$), this reduces to a combination of the forwards and backwards Euler methods, with $\alpha_0=1$, $\alpha_1=-1$, and $\beta_0=1$. Table 2.1 lists the coefficients for first through third orders.

2.2.1 The full projection algorithm

With the schematic of (2.15), the pressure projection steps of equations (2.8) can finally be written without resort to a specific forward/backward Euler formulation. As a reminder, the pressure projection method steps advection and forcing explicitly, while pressure and viscosity are determined implicitly, in that order.

The development of the projection algorithm in section 2.1.1 applies without modification to the discretization in (2.15). The key insight for the pressure projection is that the divergence and Laplacian operators commute, so taking the divergence of the time-discretized momentum equations eliminates the new time-level ($\vec{u}^{n+1}$) in the interior. This *applies in exactly the same way* to (2.15), because $\vec{u}^{n+1}$ appears with only linear differential operators.

Incorporating the stiffly-stable multi-step method gives the revised algorithm:

$$\alpha_0\vec{u}^* = \sum_{j=0}^{j=k-1} -\alpha_{j+1}\vec{u}^{n-j} + \Delta t\beta_j\big(-\vec{u}^{n-j}\cdot\nabla\vec{u}^{n-j} + \vec{g}\rho^{n-j} + \vec{F}(\vec{u}^{n-j},\rho^{n-j})\big), \tag{2.16a}$$

$$\Delta t\nabla^2 p^* = \alpha_0\nabla\cdot\vec{u}^*, \tag{2.16b}$$

subject to the boundary condition $\partial_n p^* = \Delta t^{-1}\alpha_0\hat{\mathbf{n}}\cdot\vec{u}^* + \nu\hat{\mathbf{n}}\cdot\nabla\times\nabla\times\vec{u}^{n+1}$ from (2.14), and

$$\left(\alpha_0 - \nu\Delta t\nabla^2\right)\vec{u}^{n+1} = \alpha_0\vec{u}^* - \Delta t\nabla p^*, \tag{2.16c}$$

where the standard no-slip boundary conditions apply. Finally, tracers that are conserved following the fluid motion (satisfying (2.1c)) can be updated much like (2.16a):

$$\alpha_0 T^{n+1} - \kappa\nabla^2 T^{n+1} = -\sum_{j=0}^{j=k-1}\alpha_{j+1}T^{n-j} + \Delta t\beta_j\vec{u}^{n-j}\cdot\nabla T^{n-j}, \tag{2.17}$$

where here T takes the place of any advected tracer, and κ is its molecular diffusion. If $\kappa \neq 0$, then the equation requires boundary conditions, where Dirichlet-type ($T = 0$ on the boundary) or Neumann-type ($\partial_n T = 0$) are used as physically appropriate.

The pressure boundary condition for (2.16b) is still written implicitly, involving $\vec{u}^{n+1}$ on both sides. The terms on the right-hand side of the equation, however, can be extrapolated to the new time level, and the β terms in table 2.1 provide a scheme for doing precisely that. That is:

$$f^{n+1} = f^n + \mathrm{O}(\Delta t),$$
$$f^{n+1} = 2f^n - f^{n-1} + \mathrm{O}(\Delta t^2), \text{ and}$$
$$f^{n+1} = 3f^n - 3f^{n-1} + f^{n-2} + \mathrm{O}(\Delta t^3).$$

These are directly provable by considering the Taylor expansion of f about $t^{n+1} = t^n + \Delta t$, and this will be discussed in more detail in section 2.2.2. This extrapolation gives for the corrected pressure boundary condition:

$$\partial_n p^* = \Delta t^{-1} \hat{\mathbf{n}} \cdot \vec{u}^* + \nu \hat{\mathbf{n}} \cdot \nabla \times \nabla \times \sum_{j=0}^{j=k-1} \beta_j \vec{u}^{n-j}. \tag{2.18}$$

The extrapolation error is $\mathrm{O}(\Delta t^k)$, meaning that the near-boundary velocities will be accurate to $\mathrm{O}(\nu \Delta t^{k+1})$, since velocities are adjusted by $\Delta t \nabla p^*$ in (2.16c). In practice, for high Reynolds-number flows ν is very small, and the extrapolation can be neglected entirely.

Over and above the abbreviated algorithm described in 2.1.1, the full algorithm here requires storage of several previous time-levels. The Euler-derived method (first-order) requires only $\vec{u}^n$ and $\mathbf{N}(\vec{u}^n)$ to find $\vec{u}^{n+1}$. Each additional order of accuracy requires storing a further time-level; at third order the model must keep $\vec{u}^n$, $\vec{u}^{n-1}$, $\vec{u}^{n-2}$, and the equivalent-time nonlinear terms. Since $\vec{u}$ is a vector quantity, this will result in significant memory usage. Pressure, however, does *not* need to be stored at multiple time-levels; it is computed "from scratch" at each step.

2.2.2 Accuracy and stability

Of course, the claim that the timestepping schemes of table 2.1 are actually first through third-order needs verification. Since the scheme is a mix of explicit and implicit terms, we will first verify the accuracy of the fully implicit part ($\mathbf{N} = 0$), and then extend the analysis to the mixed scheme.

The implicit scheme

Neglecting the explicit terms in (2.15) gives a simplified scheme, which for the scalar equation $\frac{\partial f}{\partial t} = \mathbf{L}(f)$:

$$\sum_{j=0}^{j=k} \alpha_j f^{n+1-j} = \Delta t \mathbf{L}(f^{n+1}). \tag{2.19}$$

To demonstrate the proper order of accuracy, expand the terms of the left-hand side in a Taylor series about t^{n+1}, and use $f_t = \mathbf{L}(f)$ to cancel the first derivative term. Proceeding in the order of table 2.1 gives:

$$\begin{aligned} f^{n+1} - f^n - \Delta t f_t^{n+1} =& f^{n+1} - \left(f^{n+1} - \Delta t f_t^{n+1} + \mathrm{O}(\Delta t^2 f_{tt})\right) - \Delta t f_t^{n+1} \\ =& \mathrm{O}(\Delta t^2 f_{tt}) \end{aligned} \tag{2.20a}$$

$$
\begin{aligned}
\tfrac{3}{2}f^{n+1} - 2f^{n} + \\
\tfrac{1}{2}f^{n-1} - \Delta t f_t^{n+1} =& \tfrac{3}{2}f^{n+1} - \Delta t f_t^{n+1} \\
& -2\left(f^{n+1} - \Delta t f_t^{n+1} + \tfrac{\Delta t^2}{2} f_{tt}^{n+1} + \mathrm{O}(\Delta t^3 f_{ttt})\right) \\
& + \tfrac{1}{2}\left(f^{n+1} - 2\Delta t f_t^{n+1} + 2\Delta t^2 f_{tt}^{n+1} + \mathrm{O}(\Delta t^3 f_{ttt})\right) \\
=& \mathrm{O}(\Delta t^3 f_{ttt})
\end{aligned} \tag{2.20b}
$$

$$
\begin{aligned}
\tfrac{11}{6}f^{n+1} - 3f^{n} + \tfrac{3}{2}f^{n-1} - \\
\tfrac{1}{3}f^{n-2} - \Delta t f_t^{n+1} =& \tfrac{11}{6}f^{n+1} - \Delta t f_t^{n+1} \\
& -3\left(f^{n+1} - \Delta t f_t^{n+1} + \tfrac{\Delta t^2}{2} f_{tt}^{n+1} - \tfrac{\Delta t^3}{6} f_{ttt}^{n+1} + \mathrm{O}(\Delta t^4 f_{tttt})\right) \\
& \tfrac{3}{2}\left(f^{n+1} - 2\Delta t f_t^{n+1} + 2\Delta t^2 f_{tt}^{n+1} - \tfrac{8\Delta t^3}{6} f_{ttt}^{n+1} + \mathrm{O}(\Delta t^4 f_{tttt})\right) \\
& -\tfrac{1}{3}\left(f^{n+1} - 3\Delta t f_t^{n+1} + \tfrac{9\Delta t^2}{2} f_{tt}^{n+1} - \tfrac{27\Delta t^3}{6} f_{ttt}^{n+1} + \mathrm{O}(\Delta t^4 f_{tttt})\right) \\
=& \mathrm{O}(\Delta t^4 f_{tttt}).
\end{aligned} \tag{2.20c}
$$

Each of these schemes is locally accurate at one order higher than the global accuracy – the one-step scheme, for example, is locally second order in time. The reduction in order occurs because integrating to a finite final time t_{fin} requires $\Delta t^{-1} t_{fin}$ timesteps, which cancels one order of Δt in accuracy.

As an additional note, this analysis assumes that both f and $\mathbf{L}(f)$ are continuously differentiable in time to the proper order (fourth-order for the three-step scheme). If this is not the case, then f will not have a Taylor-series expansion of the proper number of terms. For freely-evolving fluid governed by the incompressible Navier-Stokes equations this is generally not a concern, but this assumption can be broken with forcing that impulsively starts or stops.

The mixed scheme

Returning to the general (scalar) equation $\frac{\partial f}{\partial t} = \mathbf{L}(f) + \mathbf{N}(f)$, the only difference over the simplified equation above is the reintroduction of the explicitly-stepped $\mathbf{N}$ operator. Unlike the development in equations (2.20), the $\mathbf{N}$ operator is never directly taken at t^{n+1}. In general, when $\mathbf{N}$ is nonlinear, this would involve an expensive, iterative solution process. Avoiding this via extrapolation from previous time-levels introduces additional error.

The β terms in table 2.1 act to extrapolate $\mathbf{N}(f)$ to $\mathbf{N}(f^{n+1})$. That is:

$$
\mathbf{N}(f^{n+1}) = \sum_{j=0}^{j=k-1} \beta_j \mathbf{N}(f^{n-j}) + \mathrm{O}(\Delta t^{\alpha}), \tag{2.21}
$$

where Δt^{α} is the error term, which depends on which case from table 2.1 is under consideration. Provided this error is one order less than that in (2.20) (because $\mathbf{L}$ and $\mathbf{N}$ are multiplied by Δt), the accuracy of the overall scheme is preserved.

Verifying this proceeds as in (2.20). Writing $g^k = \mathbf{N}(f^k)$ for notational convenience and expanding it in a Taylor series about t^{n+1} gives:

$$\begin{aligned} g^{n+1} - g^n =& g^{n+1} - \left(g^{n+1} + \mathrm{O}(\Delta t g_t)\right) \\ =& \mathrm{O}(\Delta t g_t) \end{aligned} \tag{2.22a}$$

$$\begin{aligned} g^{n+1} - 2g^n + g^{n-1} =& g^{n+1} - 2\left(g^{n+1} - \Delta t g_t^{n+1} + \mathrm{O}(\Delta t^2 g_{tt})\right) \\ &+ \left(g^{n+1} - 2\Delta t g_t^{n+1} + \mathrm{O}(\Delta t^2 g_{tt})\right) \\ =& \mathrm{O}(\Delta t^2 g_{tt}) \end{aligned} \tag{2.22b}$$

$$\begin{aligned} g^{n+1} - 3g^n + 3g^{n-1} - g^{n-2} =& g^{n+1} - 3\left(g^{n+1} - \Delta t g_t^{n+1} + \tfrac{\Delta t^2}{2} g_{tt} + \mathrm{O}(\Delta t^3 g_{ttt})\right) \\ &+ 3\left(g^{n+1} - 2\Delta t g_t^{n+1} + 2\Delta t^2 g_{tt} + \mathrm{O}(\Delta t^3 g_{ttt})\right) \\ &- \left(g^{n+1} - 3\Delta t g_t^{n+1} + \tfrac{9\Delta t^2}{2} g_{tt} + \mathrm{O}(\Delta t^3 g_{ttt})\right) \\ =& \mathrm{O}(\Delta t^3 g_{ttt}) \end{aligned} \tag{2.22c}$$

Indeed, the extrapolation for the explicit terms is sufficiently accurate to preserve the overall time-accuracy of the mixed scheme. The error contributed by the explicit terms depends on $\frac{\partial \mathbf{N}(f)}{\partial t}$ which must have a continuous third-derivative (for (2.22c)) for the scheme to reach its full accuracy.

Stability

Of course, accuracy is only one necessary component of a time-marching scheme. Highly accurate schemes are useless for long-time integrations unless they are also stable. In the sense of time-marching schemes, stable methods are those were transient error decays to zero. When the exact solution of the equation is itself decaying, this becomes a requirement that the discrete solution also decays in time.

Direct analysis of the time-discretized Navier-Stokes equations in (2.16) is extremely difficult. The pressure is governed by the relatively complicated projection operator, and the nonlinear advection prevents direct normal-mode analysis. Instead, this section will consider the stability of the simpler but still illustrative one-dimensional advection-diffusion equation:

$$f_t = \nu f_{xx} - V f_x, \tag{2.23}$$

where V is the advective speed and ν is the diffusivity. With periodic boundary conditions on a domain $x \in [-\pi, \pi]$, the Fourier series decomposition of (2.23) decouples the Fourier modes. Writing the kth Fourier mode of f as f^k gives, for each mode independently:

$$\mathsf{f}_t^k = -k^2 \nu \mathsf{f}^k - \mathbf{i} k V \mathsf{f}^k, \tag{2.24}$$

where k is an integer, and using the identity that the Fourier transform of f_x is $\mathbf{i}k\mathsf{f}$. To discretize (2.24) in time, we take note that the f_x term is a proxy for the nonlinear advection of the Navier-Stokes equations and the f_{xx} term is a proxy for the viscosity. Therefore, we want to march the f_x ($\mathbf{i}kV\mathsf{f}$) term explicitly and the f_{xx} ($-k^2\nu\mathsf{f}$) term implicitly. This assumption will be further justified later in terms of the maximum permissible timestep for stability.

For a particular k, discretizing (2.24) with (2.15) (and dropping the superscript k on f) gives:

$$\sum_{j=0}^{j=k} \alpha_j \mathsf{f}^{n+1-j} = -\Delta t R \mathsf{f}^{n+1} - \mathbf{i} I \Delta t \sum_{j=0}^{j=k-1} \beta_j \mathsf{f}^{n-j}, \tag{2.25}$$

writing $R = k^2\nu$ and $I = kV$. This forms a k-step recurrence relation for f, which means that f is a linear combination of k terms, each of the form r^n for some r[6]. Since r^n is itself a solution, substituting $\mathsf{f}^n = r^n$ into (2.25) gives:

$$\sum_{j=0}^{j=k} \alpha_j r^{k-j} = -\Delta t R r^k - \mathbf{i} \Delta t I \sum_{j=0}^{j=k-1} \beta_j r^{k-1-j}, \tag{2.26}$$

which is a kth order polynomial. Provided that all the roots have magnitude of less than one ($|r| < 1$), the solution is necessarily decaying as $n \to \infty$. For $k = 1$, the solution of (2.26) is relatively simple, with the basic (linear) equation being:

$$r - 1 = -\Delta t R r - \mathbf{i} \Delta t I. \tag{2.27}$$

This gives $r = \frac{1-\mathbf{i}\Delta t I}{1+\Delta t R}$, which has magnitude less than one provided $\Delta t^2 I^2 < \Delta t^2 R^2 + 2\Delta t R$.

This relation has a few specific implications. Firstly, in the diffusion-free case $R = 0$, so *the first-order method is unstable for pure advection*. On the other hand, with no advection $I = 0$ and $R > 0$, and *the first-order method is absolutely stable for pure diffusion*. More generally, the relation implies $\Delta t^2 V^2 k^2 < \Delta t^2 k^4 \nu^2 + 2\Delta t k^2 \nu$, and after cancelling k^2 that recovers the timestep restriction:

$$\Delta t < 2\frac{\nu}{V^2}, \tag{2.28}$$

in the limit that $k \to 0$. Although independent of resolution (k), this timestep is uselessly small for advection-dominated flows.

[6]Ignoring the possibility of multiple roots to the recurrence relation, which would add terms of the form nr^n for a double root, $n^2 r^n$ for a triple, and so on. This possibility does not affect the analysis of this section, however.

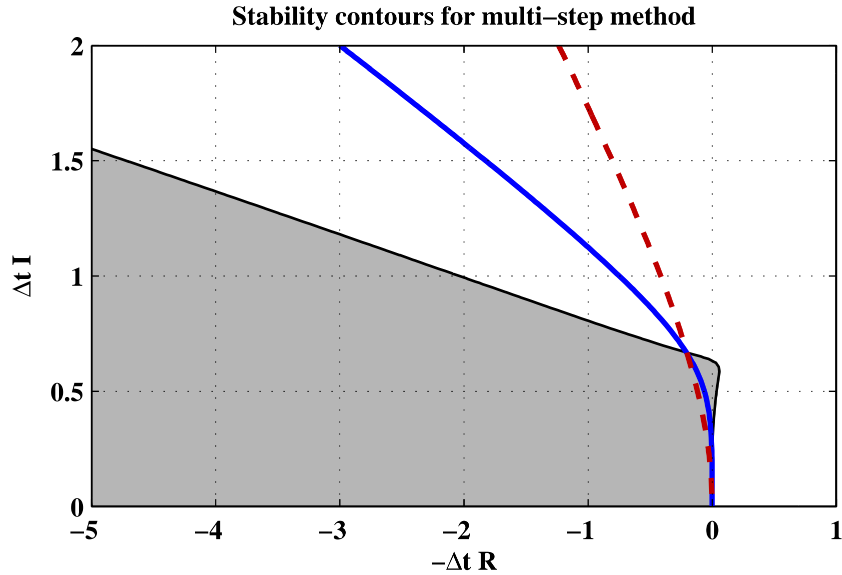

Figure 2.1: Stability contours for the first-order (dashed), second-order (solid), and third-order (shaded) multi-step methods. Along the contours, the maximum root of (2.26) has magnitude one and the method is neutrally stable. The first and second-order methods are *not* stable along $R = 0$, and so are unsuitable for strictly advective flows. The graphs are symmetric along the imaginary parameter, so only the upper half-plane is shown for simplicity.

For the second and third-order methods, it is easier and more informative to find the roots of (2.26) numerically, and the resulting stability contours are plotted on the complex plane in figure 2.1. Provided the eigenvalues of the spatial differential operators lie inside the stable region, the timestepping is stable. For the special cases discussed above, purely advective flow ($\nu = R = 0$) has eigenvalues that lie strictly on the imaginary axis, and purely diffusive flow ($V = 0$) has eigenvalues that lie strictly on the negative real axis. All of the methods are stable for purely diffusive flow, but only the third-order method is stable for purely advective flow.

Even for mixed advection and diffusion, the multi-step method is stable provided that $\Delta t I$ is small enough. Purely-advective flow presents the strictest cutoff, of $\Delta t I < 0.6$ approximately (from figure 2.1). With the assumption that I scales linearly with velocity and grid resolution (trivially true for (2.24), and will be more rigorously justified in the context of Chebyshev polynomials in section 2.3), this gives a timestep restriction of $\Delta t = \mathrm{O}(V^{-1}\Delta x)$ under a wide variety of conditions.

2.2.3 Physical interpretation

A second interpretation for the timestepping scheme of table 2.1 is possible. Instead of the coefficients being chosen specifically to cancel the first few terms of the relevant Taylor series ((2.20) or (2.22)), it is possible to consider the coefficients as those of a polynomial interpolant [Ascher and Petzold, 1998].

In general, a kth order timestepping method will exactly reproduce polynomials of kth degree. That is, the Euler method will be exact for the differential equation $f_t = 1$ (reproducing $f(t) = t$), but it will be inaccurate for a higher-order polynomial: $f_t = t$ will be exactly reproduced by a second-order method. With that in mind, it makes sense to model f locally (about $t = t^{n+1}$) as a polynomial of the proper order.

A polynomial of kth order needs $k+1$ constraints. Setting f_t^{n+1} equal to the right hand side at t^{n+1} provides one, and requiring that the model polynomial pass through the previous k points. That is, for the polynomial $P(t)$:

$$\frac{\partial}{\partial t} P(t^{n+1}) = \text{right-hand-side at } t^{n+1} \tag{2.29a}$$

$$\begin{aligned} P(t^n) &= f^n \\ &\vdots \\ P(t^{n+1-k}) &= f^{n+1-k}. \end{aligned} \tag{2.29b}$$

Then, by construction f^{n+1} is equal to $P(t^{n+1})$. The value f^{n+1} is of course unknown, but with the constraints (2.29b) to satisfy, the polynomial is fixed save for f^{n+1} as a parameter.

With the Lagrange interpolation formula, it is possible to build an explicit representation of P, leaving f^{n+1} as a free parameter. Then, analytically taking the derivative gives a relation to satisfy (2.29a). The Lagrange interpolation formula is:

$$P(t) = \sum_{i=0}^{k} f^{n+1-i} \prod_{j \neq i} \frac{t - t^{n+1-j}}{t^{n+1-i} - t^{n+1-j}}. \tag{2.30}$$

At each point t^{n+1-i} in the stencil, only a single term of the summation is nonzero. This relation is linearly dependant on the f^j values, and for the third-order method two of the resulting basis polynomials are shown in figure 2.2.

Taking the derivative of (2.30) generally results in a double summation over the product, as $\prod(t - t^{n+1-j})$ breaks up into k terms of degree $k-1$. However, for $P_t(t^{n+1})$, with the exception of the f^{n+1} term in the summation only one of these k terms is nonzero: the others all include $(t - t^{n+1})$. For the f^{n+1} term, the opposite happens: the $(t - t^{n+1-j})$ terms in the numerator

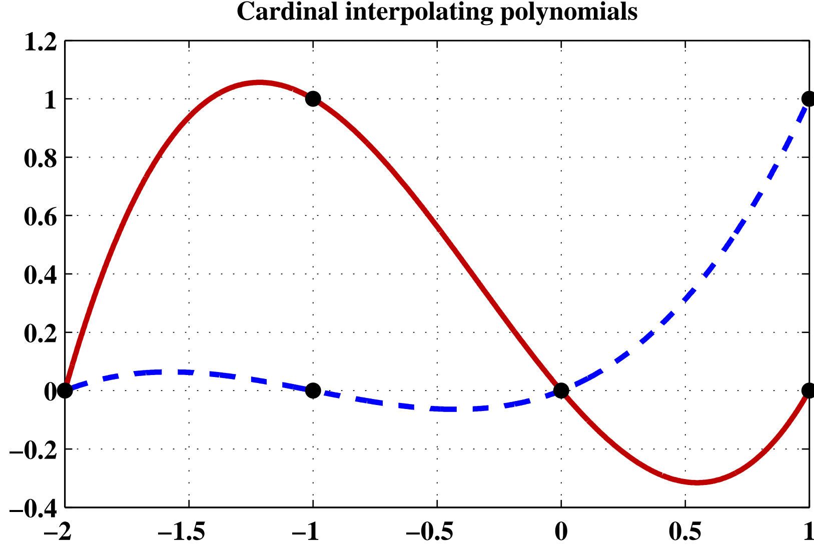

Figure 2.2: Two of the cardinal polynomials from the third-order method, at $t = -1$ (solid) and $t = 1$ (dotted). The other two polynomials (at $t = 0$ and $t = -2$) are mirror images of those shown, reflected about $t = -0.5$. A cardinal polynomial is that given by (2.30) when f^j is nonzero (one) at a single grid point.

will cancel with equivalent terms in the denominator, removing the product. Performing the differentiation gives:

$$P_t(t^{n+1}) = \sum_{i=1}^{k} \frac{f^{n+1-i}}{t^{n+1-i} - t^{n+1}} \prod_{j \neq i,0} \frac{t^{n+1} - t^{n+1-j}}{t^{n+1-i} - t^{n+1-j}} + f^{n+1} \sum_{i=1}^{k} (t^{n+1} - t^{n+1-i})^{-1}. \tag{2.31}$$

Evaluating (2.31) recovers the α coefficients of table 2.1. As an example, for the third-order method (letting $t^{n-2+i} = i\Delta t$):

$$\begin{aligned} P_t(t^{n+1}) =& \frac{f^{n-2}}{-3\Delta t} \frac{\Delta t}{-2\Delta t} \frac{2\Delta t}{-\Delta t} + \frac{f^{n-1}}{-2\Delta t} \frac{\Delta t}{-\Delta t} \frac{3\Delta t}{\Delta t} \\ &+ \frac{f^n}{-\Delta t} \frac{2\Delta t}{\Delta t} \frac{3\Delta t}{2\Delta t} + f^{n+1} \left((3\Delta t)^{-1} + (2\Delta t)^{-1} + \Delta t^{-1} \right) \\ =& \Delta t^{-1} \left(-\frac{1}{3} f^{n-2} + \frac{3}{2} f^{n-1} - 3 f^n + \frac{11}{6} f^{n+1} \right), \end{aligned} \tag{2.32}$$

recovering the α coefficients for the third-order method in table 2.1 without direct use of Taylor series.

Recovering the β coefficients of table 2.1 is the same, conceptually. Instead of trying to find the derivative of a Lagrange polynomial, the explicit terms are extrapolated to t^{n+1}. Since

(obviously) the values of the explicit terms are not available yet at the new timestep, extrapolation must be performed from the previous few values. At this point, the choice of a (locally) kth order polynomial for f provides useful motivation. The derivative of a kth order polynomial is a $(k-1)$th order polynomial, and that is the consistent representational choice for the explicit terms.

Following as in (2.30) for the explicit terms and modelling them collectively as a $(k-1)$th order polynomial in time ($Q(t)$, after $P(t)$ above) gives:

$$Q(t) = \sum_{i=0}^{k-1} N^{n-i} \prod_{j \neq i} \frac{t - t^{n-j}}{t^{n-i} - t^{n-j}}, \tag{2.33}$$

using the simplified notation that $N^{n-i} = \mathbf{N}(f^{n-i})$. Evaluating (2.33) at t^{n+1} to recover the β terms is straightforward, and repeating the example of (2.32) gives:

$$\begin{aligned} Q(t^{n+1}) =& N^n \frac{2\Delta t}{\Delta t} \frac{3\Delta t}{2\Delta t} + N^{n-1} \frac{\Delta t}{-\Delta t} \frac{3\Delta t}{\Delta t} + N^{n-2} \frac{\Delta t}{-2\Delta t} \frac{2\Delta t}{-\Delta t} \\ =& 3N^n - 3N^{n-1} + N^{n-2}, \end{aligned} \tag{2.34}$$

matching the proper β terms of table 2.1.

2.2.4 Variable timesteps

The development of the multi-step algorithm through the Lagrange interpolating polynomials allows for an obvious generalization of the method to variable timesteps. Indeed, the formulas for the coefficients in (2.31) and (2.33) works as written for time levels that are not spaced a fixed Δt apart.[7]

This generalization is motivated by noting that the maximum stable timestep scales inversely with the maximum velocity. For flows where the maximum velocity is not known in advance, requiring a fixed timestep is a trial and error process. A too-large choice for Δt results in instability, but a too-small choice for Δt wastes computational resources on conservative timestepping. This problem is made worse by intermittent or oscillatory flows where the maximum velocity changes significantly with time: for these flows, there is no single Δt that is near-optimal for an entire simulation.

The chief computational problem of a variable-timestep method is that it changes the implicit part of the operator. Reproducing (2.15) in slightly modified form, for the third-order operator:

$$(\alpha_0 - \mathbf{L})\, \vec{u}^{n+1} = -\alpha_1 \vec{u}^n - \alpha_2 \vec{u}^{n-1} - \alpha_3 \vec{u}^{n-2} + \beta_0 \mathbf{N}(\vec{u}^n) + \beta_1 \mathbf{N}(\vec{u}^{n-1}) + \beta_2 \mathbf{N}(\vec{u}^{n-2}), \tag{2.35}$$

[7] The polynomial form is not required to generate variable-timestep methods; Wang and Ruuth [2008] works directly from the Taylor series to consider an entire family of related methods. The key advantage of the Lagrange-polynomial form is that the generation of the method is physically intuitive.

where the Δt has been included inside the α terms, as suggested by (2.31). The right-hand side does not substantially change – although the α and β weights may differ between timesteps, they are straightforward multiples of terms already computed. The $(\alpha_0 - \mathbf{L})$ term on the left hand side, however, must have its inverse applied at each timestep. Since α_0 may change, it becomes unproductive to compute $(\alpha_0 - \mathbf{L})^{-1}$ in advance – each change to α_0 would require recomputing the inverse. Instead, variable timesteps demand a solution technique that either does not factor/invert the operator or can do so relatively cheaply. Since this depends on the spatial discretization, this will be discussed in detail in section 2.3

The chief *analytical* problem of a variable-timestep method is ensuring stability. With fixed timesteps, (2.25) is a stationary method that can be repeated infinitely. With variable timesteps, however, infinite repetition doesn't make sense. If Δt changes stepwise, then the jump will be "seen" progressively earlier in the algorithm: first from $t^{(n+1)}$ to $t^{(n)}$, then $t^{(n)}$ to $t^{(n-1)}$, and so on. (For the remainder of this section, notation is abused slightly: $t^{(k)}$ refers to the time corresponding to the kth timestep, rather than t to the kth power.)

The discrete timestepping scheme is no longer a simple recurrence relation, so the solution technique of (2.26) is no longer directly applicable. However, assuming that instability takes the form of exponential growth, the equation can be recast in the form of a (possibly) growing mode r^t (contrasting with r^n in (2.26)), giving:

$$\sum_{j=0}^{k} \alpha_j r^{t^{(k-j)}} = -R r^{t^{(k)}} - \mathbf{i} I \sum_{j=0}^{k-1} \beta_j r^{t^{(k-1-j)}}, \tag{2.36}$$

where r^{t^0} has been divided out so that the lowest-order term is $\mathrm{O}(1)$. With evenly-spaced timesteps, (2.36) reduces exactly to (2.26). The first-order method ($k = 1$) also involves only a single r exponent, so it reduces exactly to the fixed-timestep result with $\Delta t = t^{(1)} - t^{(0)}$.

(2.36) is not a polynomial, but at least for modest perturbations from evenly-spaced timesteps its solution shows much the same characteristics.[8] Straight root-finding is no longer applicable, however, and finding the roots requires a nonlinear solver. Fortunately, straightforward techniques are useful: `fsolve` in MATLAB converges easily to the largest magnitude root, given a proper initial guess.

To study the stability of variable timesteps, the trial case of integration from $t = 0$ to $t = 3$ was considered. For the third-order method, constant timesteps ($t = [0, 1, 2, 3]$) give a region of stability shaped exactly like that in figure 2.1. In contrast, variable timesteps (where the intermediate two time levels are perturbed slightly) would be expected to have a slightly different stability bound. To test this, a number of random intermediate timelevels ($1 + \varepsilon_1$ and $2 + \varepsilon_2$)

[8] Wang and Ruuth [2008] preserves a polynomial form to discuss the 0-stability of these methods more analytically, but for the third and fourth-order methods the bounds still have to be found numerically. This work avoids the cumbersome analysis in favour of direct numerical experiments.

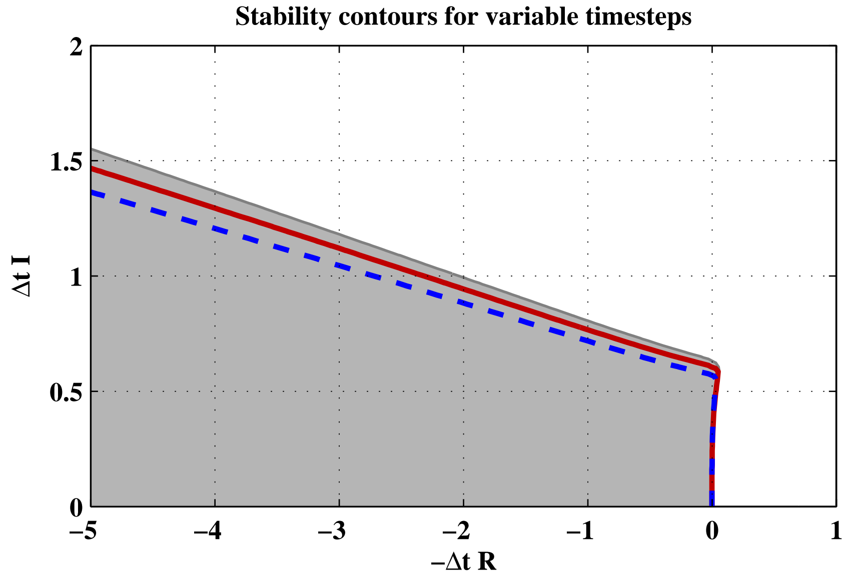

Figure 2.3: Stability contours for the variable-timestep extension, using the method described in the text. Depicted are the minimum zero-stability contours of 100 random realizations, with timesteps of $[0, 1+\varepsilon_1, 2+\varepsilon_2, 3]$, where $\varepsilon_{1,2}$ varied uniformly between ± 0.1 (solid) and ± 0.25 (dashed). The method is stable with evenly spaced timesteps ($[0,1,2,3]$) in the shaded region.

were considered (with ε uniformly varying between ± 0.1 or ± 0.25) and the largest magnitude root of (2.36) was found[9]. The results are shown in figure 2.3, and the neutral stability curve has much the same shape as for constant timesteps. The maximum permissible timestep for an advective problem (the point of intersection of the curve with the imaginary axis) is slightly reduced, however.

2.2.5 Startup

A key disadvantage of the multi-step method is that it is not self-starting. For the third-order method, computing f^{n+1} always requires f^n through f^{n-2}, but initial conditions only provide a single value.

Several schemes exist to find these startup values, but they all suffer from problems. At the simplest level, f^{n-1} and f^{n-2} could be assumed to be zero. This would give a stable algorithm,

[9]To make sure that the largest magnitude root was indeed found, the analysis considered a number of initial guesses of magnitude 10, spaced evenly around zero in the complex plane. This method converges to the proper stability curve for constant timesteps.

since only the stable third-order method is used throughout, but the improper initialization would contribute significant error. Assuming that the "true" values are $\mathrm{O}(1)$, and neglecting them imparts $\alpha_{2,3}\mathrm{O}(1)$ error to the solution. Setting $f^{n-2} = f^{n-1} = f^n$ replaces this with $\mathrm{O}(\Delta t)$ error, which still dominates the $\mathrm{O}(\Delta t^3)$ error from the remainder of the timestepping. Using the Taylor series expansion with "reverse-Euler" steps could give more plausible values ($f^{n-1} = f^n - \Delta t f_t^n$), but even still the locally $\mathrm{O}(\Delta t^2)$ error is significantly greater than the combined error from the non-startup time stepping[10].

Richardson extrapolation [Boyd, 2001] provides another means of taking the startup steps at high-order, without needing startup values from negative times. This method takes advantage of the local error properties of the first-order Euler method, which is locally second-order (see (2.20a)). Stepping from $t^{(0)}$ to $t^{(1)} = t^{(0)} + \Delta t$ with the Euler method gives:

$$f_1^1 - f^{true} = \mathrm{O}(\Delta t^2 f_{tt}) = \gamma f_{tt}(s)\Delta t^2, \tag{2.37}$$

where f^{true} is the true solution at $t^{(1)}$, s is some unknown time between $t^{(0)}$ and t^1, and γ is a constant determined by the error in the Taylor series approximation used for (2.20a). If instead $t^{(1)}$ is reached through two steps of size $0.5\Delta t$, the error is instead:

$$f_2^1 - f^{true} = \frac{\gamma}{4} f_{tt}(s_0)\Delta t^2 + \frac{\gamma}{4} f_{tt}(s_1)\Delta t^2, \tag{2.38a}$$

where s_0 is between t^0 and $t^0 + 0.5\Delta t$ and s_1 is between $t^0 + 0.5\Delta t$ and t^1. Now, assuming f is appropriately differentiable in time, $f_{tt}(s_0)$ and $f_{tt}(s_1)$ can both be written as $f_{tt}(s) + \mathrm{O}(\Delta t f_{ttt})$. Making this substitution in (2.38a) gives:

$$f_2^1 - f^{true} = \frac{\gamma}{2} f_{tt}(s)\Delta t^2 + \mathrm{O}(\Delta t^3 f_{ttt}). \tag{2.38b}$$

The numerical solutions from (2.37) and (2.38b) both give approximations for f^{true} with the same error terms, save for a constant multiple. Taking $2 \cdot (2.38b) - (2.37)$ gives:

$$2f_2^1 - f_1^1 - f^{true} = \mathrm{O}(\Delta t^3 f_{ttt}), \tag{2.39}$$

which is locally third-order. Using this method to find f^1 (from only f^0) gives an appropriate starting value to find f^2 with the two-level method, also locally third-order from (2.20b). With these values for f^1 and f^2, the three-level method can begin from t^3 onwards.

While this algorithm provides a consistent, globally third-order method, it is still unsatisfying. The $\mathrm{O}(\Delta t^3)$ error from the two starting steps is of the same order as the error from the entire remainder of the simulation. In principle this can be fixed by repeating (2.37) and (2.38) recursively, to find f^1 and f^2 to $\mathrm{O}(\Delta t^4)$ error, but this is extremely cumbersome, requiring a total of eighteen sub-steps[11], compared to two if full-sized steps could be taken.

[10] This "reverse Euler" initialization would also be very difficult for the full Navier-Stokes equations of (2.1), moreso than the model advection-diffusion equation. With the projection algorithm (2.8), pressure is only found at t^{n+1}, and it need not be given in initial conditions.

[11] Using Richardson extrapolation to find initial steps to $\mathrm{O}(\Delta t^4)$ error prevents the use of the two-step method to find f^2.

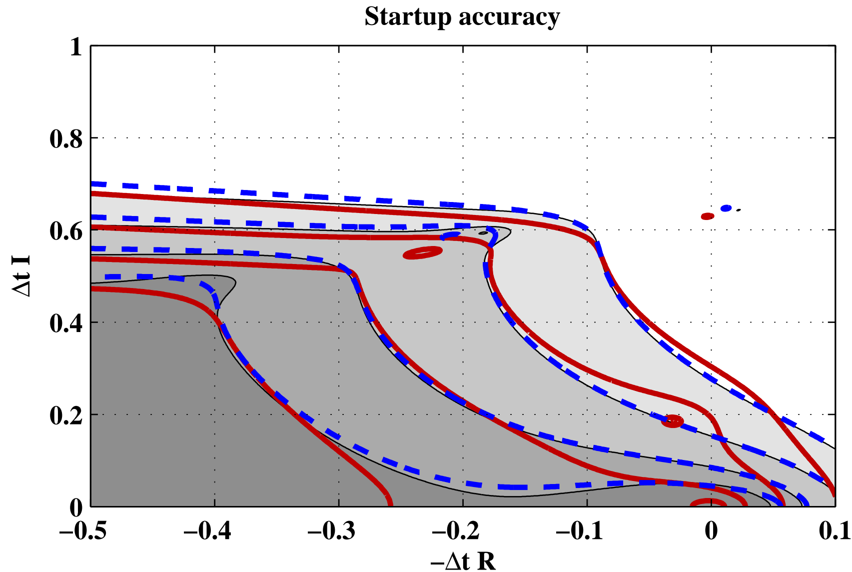

Figure 2.4: Accuracy of the variable-timestep startup method, as described in the text. The model ODE was solved to $t = 25$ with a maximum Δt of 1, with the initial condition $f^0 = 1$. Shown are accuracy contours of 10^{-1}, 10^{-2}, 10^{-3}, and 10^{-4} from outside and inside for $\Delta t_{min} = 1$ startup (solid), $\Delta t_{min} = \frac{1}{4}$ (dashed), and the full multi-step method (shaded). The multi-step method was initialized with the exact solution for f^{-1} and f^{-2}. Along the imaginary axis, the exact solution is purely advective and the contoured regions there reflect coincidental phase-matching near the stability limit of the multi-step scheme.

The idea behind Richardson extrapolation is useful, however. If starting steps f^1 and f^2 can be found to about the same local accuracy as in the bulk of the time integration, then the three-step method can continue from there. This is true *regardless of how f^1 and f^2 are computed* – they do not even need to be a fixed Δt apart, given the extension of the three-step method to variable timesteps in 2.2.4. This suggests that we can find proper starting steps "for free" simply by taking the Euler and two-step methods with small enough timesteps. That is:

$$t^{(1)} = t^{(0)} + \varepsilon \Delta t \text{ with the Euler method, and}$$
$$t^{(2)} = t^{(0)} + 2\varepsilon \Delta t \text{ with the two-step method.}$$

From there, integration can continue. The only open question is how small ε must be to give appropriate accuracy.

This question is best answered numerically. As a test case, consider integrating (2.35) for the advection-diffusion equation (2.25) to a final time $t_{fin} = 25$, with overall (non-startup) timestep $\Delta t = 1$ and initial condition $f^0 = 1$. The exact solution is $f(t) = \exp(t(-R + \mathbf{i}I))$, which is easily

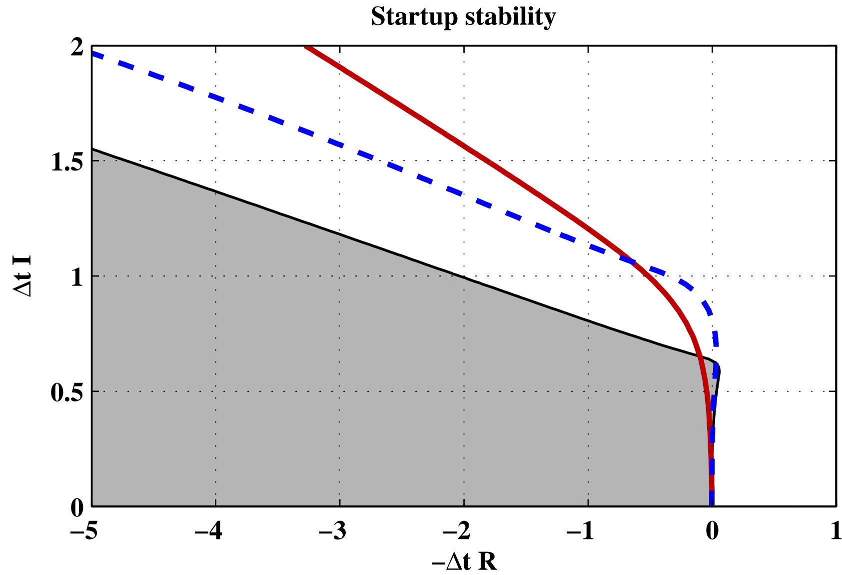

Figure 2.5: Stability of the variable-timestep startup method. The model ODE was solved to $t = 4$ with a maximum Δt of 1, using the initial condition $f^0 = 1$. The final time was chosen as the first timestep to cover only a single Δt (1) after startup. Plotted is the stability contour ($|f| = 1$) for $\varepsilon = 1$ (solid) and $\varepsilon = \frac{1}{4}$ (dashed), the latter of which is stable for purely advective motion. The shaded region is the stability region for the three-level method, reproduced from figure 2.1.

computable. For ε less than one, the timestep size needs to increase to $\Delta t = 1$. Taking two steps of size ε and then progressively doubling the step size gradually increases Δt while conveniently exactly reaching $t = 1$ with $\Delta t = 1$ – the time levels reached are $[0, \varepsilon, 2\varepsilon, 4\varepsilon, \cdots, 1, 2, \cdots, 25]$.

Results of this experiment for $\varepsilon = 1$ (no timestep reduction) and $\varepsilon = \frac{1}{4}$ are shown in figure 2.4. Already, even for the modest startup step of one-quarter the maximum, the accuracy contours are very nearly the same as the analytically started three-step method. Indeed, in some cases the startup method is *more* accurate – this reflects the difficulty of properly starting a multi-step method, even analytically. For figure 2.4, the exact analytic solution was used for f^{-1} and f^{-2}. However, if $|R + \mathbf{i}I|\Delta t$ is $\mathrm{O}(1)$ then the polynomial representation used by the method poorly represents the exponential, and significant transient error results. For $\varepsilon = \frac{1}{4}$, the startup adds only two timesteps ($t = 0.25$ and 0.5) before reaching a full Δt, a significant savings compared to Richardson extrapolation.

Since the startup method is not iterated, its stability is of much less importance. In principle, however, it is possible to iterate the startup – step to some time t_{fin}, discard data from all previous

time-levels, and use the startup algorithm again to step to $2t_{fin}$[12]. In such a case, whether $|f^{fin}| \leq |f^0|$ governs the stability of the iterated-startup. Fortunately, the complicated root-finding of (2.26) or (2.36) is neatly avoided here: the method takes as input only f^0, so f^{fin} must be a single multiple of that, rather than a linear combination of roots.

Following in the analysis for accuracy, the stability of the startup method is presented in figure 2.5. Taking a very conservative estimate, we assume that the startup "restarts" after the first full, three-level step that does not contain a lower-order (reduced timestep) level. For $\Delta t = 1$, that would be the step that finds $f(t=4)$. Broadly speaking, the startup method is actually *more* stable than the three-level method, both by virtue of lower-order methods having a larger stability range (see figure 2.1) and by the reduced timestep used for the starting steps. By the time $\varepsilon = \frac{1}{4}$, the stable region also includes a segment of the imaginary axis, showing stability for purely advective motion.

2.3 Spatial discretization

The time discretization of section 2.1 is literally only half of the problem. The multi-step method accurately replaces evolution of $\vec{u}(\vec{x},t)$ on continuous time with discrete time t^n, but space remains continuous. The challenge of the spatial discretization is to model the function with a small number of degrees of freedom, related in a way such that the error from the approximation is small.

That task sounds hopelessly vague because specified so generally, it is hopeless. The (finite) sum of step functions $\alpha_k f(x - x_k)$ for ($f(x) = 1$ if $x \geq 0$ and 0 otherwise), for example, can be modelled *exactly* by the location and magnitude of the discontinuities. But that kind of piecewise-discontinuous approximation is wretched at modelling something smoother, such as a sine wave.

For the specific problem of this work, the Navier-Stokes equations (2.1) for primarily internal wave calculations, we can take inspiration from the behaviour of physical fluids. Since we are interested in only the low Mach-number limit (where wave speeds are much slower than the speed of sound), we do not have to consider the problems of transonic flow, where shock waves spontaneously develop. Additionally, any discontinuity of velocity is immediately smoothed out by viscosity, and molecular diffusivity provides the same smoothing for temperature or salinity. As a result, the spatial discretization of the Navier-Stokes problems is *primarily a problem of modelling smooth fields.*

[12] In fact, this is the principle behind restarting a job in this work. The step of "discard data from all previous time-levels" is accomplished by "notice that a cluster node has crashed, utter a few choice words, and submit the job again to the queue".

Collocation methods

The primary remaining open question is what physical meaning to attach to the discrete degrees of freedom. Three major divisions exist in the literature for this physical meaning: meshfree methods, element-based methods, and collocation methods. These differ in how concretely each computational degree of freedom is "attached" to physical meaning.

At extreme coupling, "meshfree" methods [Monaghan, 1988] take the discrete elements to be particles at arbitrary locations in space. Each of these particles carries with it the fundamental quantities of interest that are conserved following motion; in the case of the Navier-Stokes equations the quantities would be the momentum and density. To create a continuous field from the discrete set of particles, the effect of the particles is smoothed over a characteristic length scale; the effective (momentum/density) at an arbitrary point is given by the contribution of particles within that length scale from the point. While these methods have had excellent utility in astrophysics where there is a wide separation of length and density scales [Bate et al., 2003], it is difficult to solve implicit equations with such an irregular organization. Even ignoring viscosity, the pressure projection step would be extremely troublesome with such a discretization.

At the other extreme, element-based methods [Karniadakis and Sherwin, 2005] consider the degrees of freedom to be multipliers of basis functions, generally polynomials, that are defined over a fixed (small) element. The domain is then discretized into many of these elements. The relevant Partial Differential Equation is solved in weak form, where:

$$\int \mathbf{L}(f) \cdot \phi \mathrm{d}V = 0 \tag{2.40}$$

where f is the solution to the PDE, $\mathbf{L}$ is the operator corresponding to the PDE (including any forcing terms), and ϕ is an arbitrary member of a test function space, usually (but not always) taken to be the same as the basis functions. If $\mathbf{L}$ is linear, then (2.40) gives a linear system for the discretized solution. Additionally, provided the basis and test vectors have compact support, only a few terms will interact, leading to a sparse system. The primary sources of error in this approximation are the projection of f onto the finite-dimensional space spanned by the basis functions and the equivalent restriction on the test function ϕ, which in the full weak form can be replaced by *any* function with proper regularity.

The key advantage to a finite element method comes from its flexible discretization, and it can treat very general problem domains through triangularization. However, this is overkill for this work, focused on process studies of geophysical flows. Here, we can rely on having a more structured grid (see 2.3.4), and we can avoid problems of grid quality that arise from triangular mesh generation. In addition, the finite element method can have problems with nonlinearities[13]. The condition for reducing (2.40) to a linear system is that $\mathbf{L}$ is itself linear, and this does not hold

[13] The product $\vec{u} \cdot \nabla \vec{u}$ must be projected onto the basis functions in an element method, whereas grid-based methods allow evaluating the product on the grid.

for the Navier-Stokes equations. Computing the nonlinear terms of (2.1) is of course possible, but then the solution process is inherently nonlinear. The computation process itself can also be cumbersome, since the interaction between basis elements may not lie in the same functional space[14]. Finally, the matrix implied by (2.40) is only sparse provided the basis and test functions only strongly interact in a few cases. If the order of approximation in an element is large, then many more basis and test functions have strong interactions. The resulting matrix becomes more dense, and solving for f becomes difficult.

In between meshfree and element methods, collocation methods consider the degrees of freedom to be the value of the function *sampled* at a discrete location $\vec{x}_j$. This representation is simple, entirely intuitive, and allows for obvious computation of the nonlinear terms; that is why this work is based on collocation methods (also called pseudospectral methods when the method is of high order). The sampling of some (local) nonlinear relation on f is exactly that given by the same relation on f_j (the sampled value of f at x_j). The approximation error introduced instead comes from how derivatives of the function are computed. The classic method is to use finite differences (e.g. $f'(x_j) \approx (2\Delta x)^{-1}(f(x_j+\Delta x) - f(x_j - \Delta x))$ for centered differences on a uniform grid of spacing Δx), and this introduces truncation error. The key philosophical problem of collocation methods is that error, constrained to be zero at grid points (given the approximations for derivatives), does not necessarily have to be small between grid points. While this is generally not a problem for finite difference methods, this is the ultimate source of the Runge phenomenon for poorly-defined polynomial methods (see section 2.3.3). Ultimately, it is a matter of how the collocation method interpolates between grid points, turning the discrete $f(x_j)$ into a continuous $\tilde{f}(x)$.

2.3.1 Interpolation

For linear methods, the interpolation problem reduces to how the method interpolates the discrete delta[15]:

$$\delta(x) = \begin{cases} 1 & \text{if } x = 0, \\ 0 & \text{otherwise,} \end{cases} \tag{2.41}$$

and on a discretized grid where $x \in x_j$, $\delta_j(x) = \delta(x - x_j)$. The discrete delta is 1 *at a single grid point x_j, and* 0 *at every other grid point.* Along with a set of interpolating functions $L_j(x)$, where $L_j(x_k) = \delta_j(x_k)$ for x_k on the discrete grid, a collocation method gives a unique reconstruction:

$$\tilde{f}(x) \approx \sum_j f(x_j) L_j(x). \tag{2.42}$$

[14]An example of this is trivial enough: consider piecewise linear basis elements $\phi(x)$. $\phi^2(x)$ is piecewise quadratic, and that cannot be written exactly in a piecewise linear form. This problem is closely related to the concept of aliasing error in spectral methods, and it will be discussed from that perspective here.

[15]An important consideration is that this definition of the discrete delta is 1-norm 0, in that $\int_{-\infty}^{\infty} \delta(x) = 0$. However, on a discretized grid $\sum_{j=-\infty}^{\infty} \delta(x_j) = 1$.

The error in reconstruction is $||f(x) - \tilde{f}(x)||$ under an appropriate norm. As the discrete grid becomes denser ($\max(x_{j+1} - x_j) \to 0$ uniformly), a consistent interpolation has error approach zero in the limit.[16]

For the remainder of this section, we can simplify matters considerably by considering an infinite grid $x \in (-\infty, \infty)$ of uniform spacing, where $x_j = j\Delta x$. With this simplification, we can take the interpolating functions to be index-independent: $L_j(x) = L(x - x_j) = L(x - j\Delta x)$.

Polynomial interpolation

The most straightforward interpolation to use for this kind of spatial grid is polynomial interpolation. Away from the gridpoints, $\tilde{f}(x)$ is taken to be given by the Lagrange interpolant of the nearest $2n+1$ points, for $2n$th order interpolation. Provided the number of points is odd, the resulting interpolation stencil has a midpoint that is also a grid point, and the interpolating function is continuous (up to the $2n$th derivative) at the grid points.

As a trivial example, nearest-neighbour interpolation corresponds to $n = 0$, and it is given by:

$$L(x) = \begin{cases} 1 & \text{if } |x| \leq 0.5\Delta x, \\ 0 & \text{otherwise.} \end{cases}$$

More generally, the interpolant is supported on $|x| \leq n + 0.5$, and it is given by:

$$L(x) = \begin{cases} \prod_{j=k-n, j\neq k}^{k+n} \frac{x - j\Delta x}{-j\Delta x} & \text{if } |x| \leq n + 0.5, \\ 0 & \text{otherwise,} \end{cases} \tag{2.43}$$

where $k = \lfloor (\Delta x)^{-1} x + 0.5 \rfloor$ is the number of gridpoints the stencil is centered away from 0, and $\lfloor \cdot \rfloor$ is the greatest integer function.

The resulting cardinal functions for a few orders of interpolation are plotted in 2.6. The function is bounded and smooth at gridpoints, and there the continuous derivative is the same as that given by the appropriate central finite difference stencil of half-width n. However, between the grid points the interpolant is not continuous. The only general way to ensure continuity is to demand that the same stencil be used everywhere – then the polynomial of (2.43) would be defined for the entire domain, rather than piecewise. This is obviously not possible for an infinite domain, but it is possible on a finite domain; the extension to allow for smooth, well-conditioned interpolation is discussed in section 2.3.3.

The problem of the interior discontinuity does not affect the overall accuracy of the interpolation. For a polynomial of order at most $2n$, the discontinuities between grid points given by

[16] This consistency depends on smoothness of f and the error norm. For example, under the infinite norm $||\cdot||_\infty$ no interpolant that is continuous at x_j will have decaying error for $f(x) = \delta(x)$. Fortunately, this is generally not a problem with the smooth functions we consider.

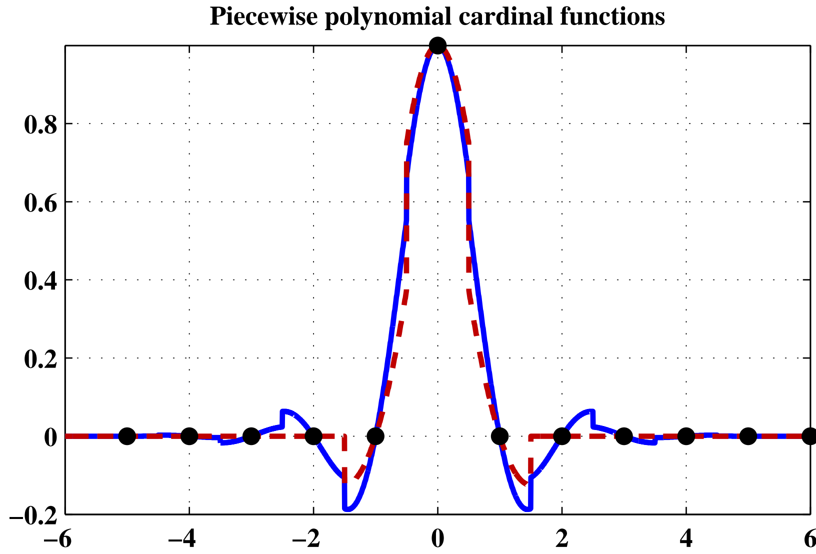

Figure 2.6: Cardinal functions for second-order (dashed) and tenth-order (solid) polynomials, using a centered interpolation stencil of the proper order. Even with high order, the cardinal functions have internal discontinuities, caused by the stencil-switches halfway between each grid point.

(2.43) exactly cancel, and $\tilde{f}(x)$ is exactly equal to $f(x)$. Even when $f(x)$ is not a polynomial of proper order, the error in interpolation is the same $O(\Delta x^{2n+1})$, provided f is differentiable to order $2n$[Trefethen, 2000].

Finite difference interpolation

The discontinuities of (2.43) make grid points special. While $\tilde{f}(x)$ is continuous and differentiable at grid points, between them $\tilde{f}$ is in general not even continuous. The problem of smooth interpolation is more subtle, since continuity adds additional constraints that must be reflected in the interpolant.

As an example of continuous interpolation, consider an interpolant consistent with the compact, central finite difference stencil:

$$\begin{aligned} \tilde{f}(x_j) &= f(x_j) \\ \tilde{f}'(x_j) &= (2\Delta x)^{-1}(f(x_{j+1}) - f(x_{j-1})) \\ \tilde{f}''(x_j) &= (\Delta x)^{-2}(f(x_{j+1}) - 2f(x_j) + f(x_{j-1})). \end{aligned} \tag{2.44}$$

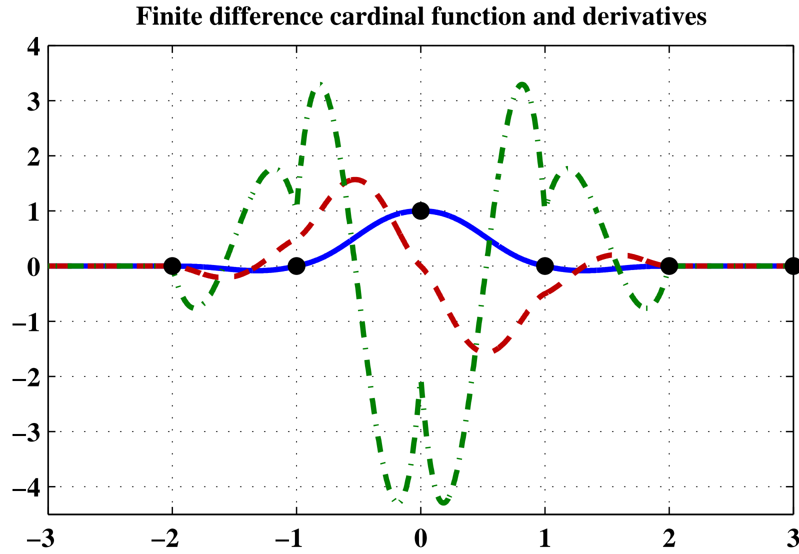

Figure 2.7: The cardinal function with continuous second derivative implied by the compact, second-order finite difference stencils (solid) along with its first (dashed) and second (dash-dotted) derivatives. The derivatives are continuous up to second order, and they reproduce the standard three-point finite difference stencils at the integer grid points.

The conditions of (2.44) are satisfied at x_j by second-order (three-point) interpolation of (2.43), but requiring a continuous second derivative needs a higher polynomial order.

Consider $\tilde{f}$ to be piecewise polynomial, defined on $[x_{j-1}, x_j)$, $[x_j, x_{j+1})$, and so on. At the endpoints of each subinterval, the constraints (2.44) hold, giving a total of six constraints. Matching these constraints requires a fifth-order polynomial. Rescaling the interval $[x_j, x_{j+1})$ to $[0,1)$ for notational simplicity, letting $L(x) = Ax^5 + Bx^4 + Cx^3 + Dx^2 + Ex + F$ and applying the constraints (2.44) gives the matrix problem:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 5 & 4 & 3 & 2 & 1 & 0 \\ 20 & 12 & 6 & 2 & 0 & 0 \end{pmatrix} \begin{pmatrix} A \\ B \\ C \\ D \\ E \\ F \end{pmatrix} = \begin{pmatrix} f(0) \\ 0.5(f(1) - f(-1)) \\ f(1) - 2f(0) + f(-1) \\ f(1) \\ 0.5(f(2) - f(0)) \\ f(2) - 2f(1) + f(0) \end{pmatrix}. \tag{2.45}$$

Solving this for the discrete delta function gives:

$$L\left(\frac{x}{\Delta x}\right) = \begin{cases} -3x^5 + 7.5x^4 - 4.5x^3 - 1x^2 + 1 & \text{if } 0 \le \frac{x}{\Delta x} < 1 \\ 1(x-1)^5 - 2.5(x-1)^4 + 1.5(x-1)^3 \\ \quad + 0.5(x-1)^2 - 0.5(x-1) & \text{if } 1 \le \frac{x}{\Delta x} < 2 \\ 0 & \text{if } \frac{x}{\Delta x} \ge 2 \\ L\left(-\frac{x}{\Delta x}\right) & \text{if } x < 0. \end{cases} \tag{2.46}$$

This function is graphed in 2.7. The interpolant has support on $\Delta x^{-1} x \in (-2, 2)$, and its second derivative is continuous everywhere. Just like second-order Lagrange interpolation, this interpolant will exactly reproduce up to quadratic polynomials (where the finite difference terms are exact).

Trigonometric interpolation

So far, the interpolants considered have all had compact support, but have had limited continuity. The Lagrange interpolants are discontinuous, and the smooth interpolant derived from the finite difference method has a discontinuous second derivative. An alternative approach is to give up on the idea of compact support on x and instead use a spectral approach.

For the spectral approach, consider the discrete Fourier transform of $f(x)$, defined on the grid $x_j = j\Delta x$. The Fourier transform, defined as:

$$\mathsf{f}(k) = \mathsf{F}[f(x)](k) = \sum_j f(j\Delta x) \exp(\mathbf{i} k j \Delta x) \Delta x \tag{2.47}$$

expresses the *discretized* $f(x_j)$ on the *infinite* grid as a *continuous* $\mathsf{f}(k)$ on a *bounded* wavenumber k. The boundedness of k comes from sampling theory [Trefethen, 2000]; on a grid of spacing Δx, the waves $\exp(\mathbf{i}kx)$ and $\exp(\mathbf{i}(k + 2n\pi\Delta x^{-1})x)$ are indistinguishable from each other. This clips the valid range of wavenumbers to $k\Delta x \in (-\pi, \pi]$. Provided f is (nearly) band-limited in this range, the discrete Fourier transform correctly captures (nearly) all of the function's energy.

Turning this approach to interpolation is relatively straightforward. From (2.47), which is only valid for k in the appropriate range $\pm\pi\Delta x^{-1}$, make the assumption that for $\tilde{f}$ there is zero contribution from any frequency outside this range. Then, $\tilde{f}$ is given by the continuous inverse Fourier transform:

$$\tilde{f}(x) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \mathsf{f}(k) \exp(\mathbf{i}kx) \mathrm{d}k, \tag{2.48}$$

where the bounds on the integral can become $-\pi\Delta x^{-1}$ and $\pi\Delta x^{-1}$ from the assumption on zero high-frequency energy.

Applying this to the discrete delta function is straightforward. By definition, the discrete delta is zero at every grid point save $x = 0$, so the sum in (2.47) is trivial:

$$\mathsf{L}(k) = \Delta x \exp(\mathbf{i}k0\Delta x) = \Delta x, \tag{2.49}$$

for k in the appropriate range. Inverting this is nearly as trivial:

$$\begin{aligned} L(x) &= \frac{1}{2\pi}\int_{-\pi\Delta x^{-1}}^{\pi\Delta x^{-1}} \Delta x \exp(\mathbf{i}kx)\mathrm{d}k \\ &= \frac{\Delta x}{2\pi}\frac{1}{\mathbf{i}x}\left(\exp(\mathbf{i}\pi\Delta x^{-1}x) - \exp(-\mathbf{i}\pi\Delta x^{-1}x)\right) \\ &= \frac{\sin\left(\frac{\pi x}{\Delta x}\right)}{\frac{\pi x}{\Delta x}} \\ &= \operatorname{sinc}\left(\frac{\pi x}{\Delta x}\right), \end{aligned} \tag{2.50}$$

where $\operatorname{sinc}(x) = x^{-1}\sin(x)$, with $\operatorname{sinc}(0) = 1$ by definition. As mentioned, $L(x)$ here has global support (on $x \in (-\infty, \infty)$), but in exchange $L(x)$ is continuous and infinitely differentiable (C^∞).

Unlike the polynomial interpolants (2.46) and (2.43), the spectral interpolant (2.50) does not reproduce (non-constant) polynomials exactly. However, it will exactly reproduce any $f(x)$ which is band-limited spectrally, where all of its energy is contained in the $(-\pi\Delta x^{-1}, \pi\Delta x^{-1}]$ wavenumber range. The most straightforward physical example is a sine wave – $\sin(kx)$ is reproduced exactly, provided k is in the proper range. As $\Delta x \to 0$, the error in reproducing $f(x)$ depends on what fraction of its energy lies outside the resolved band. For C^∞ functions, this is *exponentially* small [Boyd, 2001], suggesting that (2.50) has the potential of being much more accurate than (2.46) or (2.43) for a finite Δx.

Over a fairly broad range, this assumption holds true. Figure 2.8 shows the $||\cdot||_\infty$ error of interpolating $\exp(-x^2)$ from a regular grid ($|x| \leq 10$) using the discussed cardinal functions. All of the polynomial cardinal functions decay as Δx^{k+1} for kth order interpolation, but the sinc interpolation asymptotically decays faster than any polynomial order.

This kind of result is motivating. For a fixed accuracy threshold ε, global interpolation has the potential to reach that accuracy for far, far fewer degrees of freedom than polynomial interpolation. For the example of figure 2.8, the sinc interpolation reaches $\varepsilon = 10^{-5}$ error with $\Delta x = 0.5$, an accuracy not reached by the second-order interpolation until an order of magnitude denser grid ($\Delta x = 0.05$). This difference is magnified for tighter error tolerances and higher dimensions. In two dimensions, this factor of ten difference becomes a factor of one hundred; three dimensions gives a factor of one thousand. Given the constraints of finite memory, a computation that is possible with a global method may be simply intractable with a lower-order method.

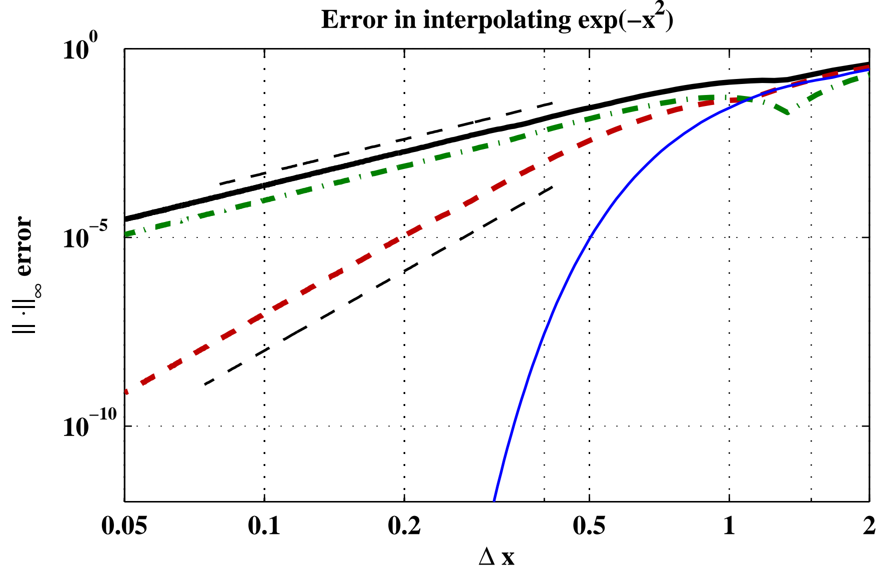

Figure 2.8: Error in interpolating $\exp(-x^2)$ for the piecewise quadratic interpolant (solid, top), C^2 finite difference interpolant (dash-dotted), piecewise sixth-order interpolant (dashed), and sinc interpolant (solid, bottom). Also included are Δx^3 and Δx^7 asymptotic lines.

2.3.2 Fourier methods

Specializing the high-order sinc interpolation of (2.50) to a real grid of finite extent is fairly straightforward. Instead of proceeding from a continuous Fourier transform in (2.50), use a discrete Fourier transform:

$$\mathsf{F}[f(x)] = \frac{1}{2\pi N}\sum_{j=0}^{N-1} f(j\Delta x)e^{\mathbf{i}kj\Delta x} \tag{2.51a}$$

$$\mathsf{F}^{-1}[\mathsf{f}(k)] = \sum_{j=-N}^{N} \mathsf{f}\left(\frac{2\pi j}{L}\right)e^{\mathbf{i}\frac{2\pi j}{L}x} \tag{2.51b}$$

where k is discretized, ranging from $-\frac{\pi}{\Delta x}$ to $\frac{\pi}{\Delta x}$ (the Nyquist frequency) in steps of $\frac{2\pi}{L}$, where L is the domain length. With this definition, all of the component waves of the discretized f have symmetry over the domain length, and so $f(x)$ is periodically extended outside of the domain.

After the Fourier transform, $\mathsf{f}(k)$ is complex-valued, even where $f(x)$ is real-valued. This does not, however, double complexity of the system. $f(x)$ being real implies that for a particular pair of $k = \pm\frac{2\pi j}{L}$, (2.51b) gives a wholly real result. Setting $\mathsf{f}(+k) = A_r + \mathbf{i}A_i$ and $\mathsf{f}(-k) = B_r + \mathbf{i}B_i$ and substituting into (2.51b), cancelling the imaginary terms gives the requirement that

$A_r = B_r$ and $A_i = -B_i$ – $\mathsf{f}(k)$ and $\mathsf{f}(-k)$ are complex conjugates. Thus, the coefficients of negative frequencies can be wholly predicted from the coefficients of the positive frequencies.[17]

With this formulation, all of the standard properties of the continuous Fourier transform are preserved. Most importantly for the purposes of this work, differentiation along x becomes an *algebraic* relation in wavenumber space:

$$\frac{\mathrm{d}}{\mathrm{d}x} f(x) = \mathsf{F}^{-1}[\mathbf{i}k\mathsf{F}[f(x)]], \tag{2.52}$$

where k is discretized as above. (2.52) follows from taking the derivative of (2.51b), where $\mathrm{d}/\mathrm{d}x$ and the summation may be interchanged because the sum is finite.

In multiple dimensions, the discrete Fourier transform extends directly. Using the notation F_x, F_y, and F_z for transforms in the x, y, and z directions respectively (using k, l, and m for the respective wavenumbers), the solution of the Poisson equation $\nabla^2 u = f$ on a periodic domain becomes:

$$u = \mathsf{F}_{xyz}^{-1}\left[-\frac{\mathsf{F}_{xyz}(f)}{k^2 + l^2 + m^2}\right]. \tag{2.53}$$

This is a matrix-free method, and the linear algebra required can be done in $\mathrm{O}(N \log(N))$ operations (per dimension) with the Fast Fourier Transform [Cooley and Tukey, 1965]. In three dimensions with N points in each dimension, that becomes $\mathrm{O}(N^3 \log(N))$ work.

In contrast, a matrix-based approach requires significantly more storage and computation. Looking at the forward application of ∇^2, perturbing a single element of the exact (discretized) solution u will affect any element in f that differs along a single index. With N points in each dimension, this means that each element of u influences $3N$ elements of f. Consequently, building the discretization of ∇^2 as a matrix would involve $\mathrm{O}(N^4)$ nonzeros. With standard lexical ordering, this matrix would have a bandwidth of $\mathrm{O}(N^2)$, so applying the inverse to f (via LU factorization) to solve the Poisson problem would require $\mathrm{O}(N^7)$ operations [Boyd, 2001]– clearly impractical, especially in comparison to the matrix-free method.

Knowing the solution for equations like (2.53) permits application of the Fourier method to the full Navier-Stokes equations. For the projection method, (2.16b) (pressure) and (2.16c) (viscosity) already look like (2.53). After computing the respective right-hand side, solving for pressure (velocity) is a simple matter of inverting the Laplacian. For (2.16c), the timestepping also adds a constant weighting to the $k^2 + l^2 + m^2$ term in the denominator.

The more interesting development is the treatment of the nonlinear term ($\vec{u} \cdot \nabla \vec{u}$) in the momentum equation (2.16a). Computation of every other term in the right-hand sides of (2.16) involves only addition and constant multiplication; these can be done in the Fourier domain quite

[17] At the Nyquist frequency, a complex wave goes through a 180-degree phase shift between grid points, but f being real-valued further restricts the phase to be precisely 0 or 180 degrees at the grid points. The only admissible wave of the sort is $\cos(\pi \Delta x^{-1} x)$, so $\mathsf{f}(\pm\pi\Delta x^{-1})$ must be strictly real-valued.

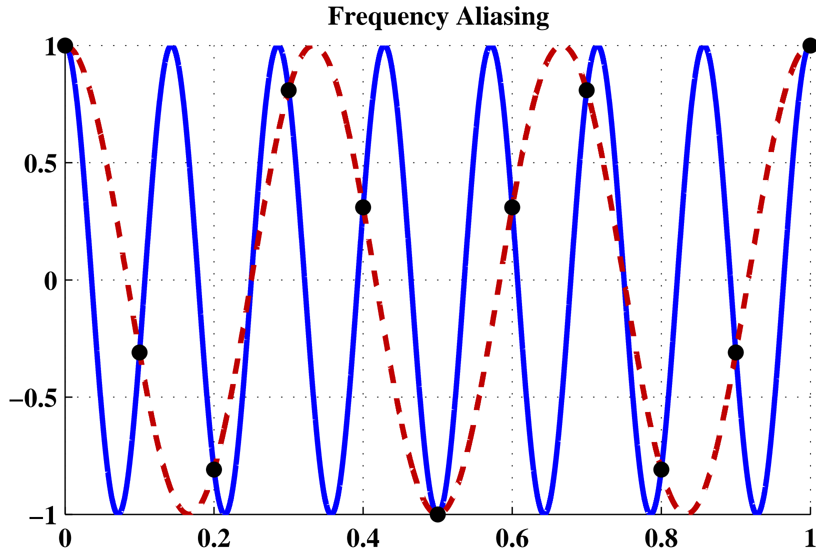

Figure 2.9: Aliasing error, with $\Delta x = .1$. The high-frequency wave ($k = 14\pi$, solid) is not well-resolved on the grid (points), and its discrete representation is the same as the lower-frequency wave ($k = -8\pi$, dashed).

easily. However, $\vec{u} \cdot \nabla \vec{u}$ involves both differentiation and multiplication by a non-constant function. Working strictly in the Fourier domain, this multiplication becomes a convolution – an $O(N^2)$ operation in one dimension. It is computationally more efficient to perform the multiplication in "physical" (x, y, z) space, even given the $O(N \log N)$ cost of transforming between spectral (k, l, m) and physical domains.

Aliasing

The downside of this approach is that it introduces aliasing error. The nonlinear term for advection is the only term in the Navier-Stokes equations that transfers energy between different scales; waves of frequency k_1 and k_2 interact to produce a wave at the sum $(k_1 + k_2)$ frequency. At spatial frequencies beyond the Nyquist frequency ($k_N = \pi \Delta x^{-1}$), a wave is indistinguishable from a wave of lower frequency[18] ($k_* = k - 2k_N$) which is well-resolved on the grid (in the frequency interval $[-k_N, k_N]$). This is illustrated in figure 2.9 in one dimension.

Some amount of error is unavoidable, simply because any method will have finite resolution (and thus a finite extent in wavenumber space). Ideally, aliasing error would just be truncated

[18] This section assumes that k is an *positive* frequency, greater than the Nyquist frequency. The analysis here repeats identically for $k < -k_N$, only with the addition rather than subtraction of $2k_N$

– any waves that are higher frequency than can be represented on the grid should be removed. Ultimately, this is a matter of filtering (discussed in section 3.5), but writing the advective term in (2.16a) as $\frac{1}{2}\vec{u}\nabla\vec{u} + \frac{1}{2}\nabla(\vec{u}\vec{u})$ ($\frac{1}{2}u_j\frac{\partial u_i}{\partial x_j} + \frac{1}{2}\frac{\partial u_i u_j}{\partial x_j}$ in indicial notation) helps reduce contamination from aliasing error [Boyd, 2001].

That alternative expressions for advection help is best examined through the simple, single-dimension expression uu_x, for a scalar u. Assuming that u is a single complex exponential $\exp(\mathbf{i}kx)$, $uu_x = \mathbf{i}k\exp(2\mathbf{i}kx)$. However, if $k > \frac{1}{2}k_N$, then this does not hold true on the discrete domain. Instead, after discretization:

$$uu_x \to \mathbf{i}ke^{2\mathbf{i}(k-k_N)x}, \tag{2.54a}$$

a wave of the proper amplitude but incorrect frequency.

In contrast, $uu_x = \frac{1}{2}(u^2)_x$ is an equivalent expression, but it has different numerical properties. On the discrete grid, $u^2 \to \exp(2\mathbf{i}(k-k_N)x)$, and taking the derivative gives:

$$\frac{1}{2}(u^2)_x \to \mathbf{i}(k-k_N)e^{2\mathbf{i}(k-k_N)x}, \tag{2.54b}$$

which has the same frequency as (2.54a), but different amplitude. Taking the average of (2.54a) and (2.54b) gives:

$$\frac{1}{2}uu_x + \frac{1}{4}(u^2)_x \to \mathbf{i}\left(k - \frac{k_N}{2}\right)e^{2\mathbf{i}(k-k_N)x}. \tag{2.55}$$

If the original spatial frequency k was not too far above half the Nyquist frequency, then the amplitude of the resulting wave will be small. For waves with frequency below half the Nyquist frequency, the form in (2.55) will still produce the exact result, since no aliasing is involved. Well above $\frac{k_N}{2}$ the error becomes substantial, but a well-resolved function has the bulk of its energy in low-frequency components.

Algorithm

With the advection term discretized, (2.16) and (2.17) can be updated to give a full algorithm for triply periodic flow. Expanding $\vec{u}$ in components (u, v, w) as necessary and incorporating Δt into

the α and β coefficients to allow for variable timesteps gives the algorithm:

$$\begin{aligned}
\alpha_0 u^* = \sum_{j=0}^{2} -\alpha_{j+1} u^{n-j} + \beta_j \Big(& F_{(x)}(\vec{u}^{n-j}, \rho^{n-j}) - \\
& \frac{1}{2}\left(u^{n-j}\mathsf{F}_x^{-1}(\mathbf{i}k\mathsf{F}_x(u^{n-j})) + v^{n-j}\mathsf{F}_y^{-1}(\mathbf{i}l\mathsf{F}_y(u^{n-j})) + w^{n-j}\mathsf{F}_z^{-1}(\mathbf{i}m\mathsf{F}_z(u^{n-j}))\right) - \\
& \frac{1}{2}\left(\mathsf{F}_x^{-1}(\mathbf{i}k\mathsf{F}_x(u^{n-j}u^{n-j})) + \mathsf{F}_y^{-1}(\mathbf{i}l\mathsf{F}_y(u^{n-j}v^{n-j})) + \mathsf{F}_z^{-1}(\mathbf{i}m\mathsf{F}_z(u^{n-j}w^{n-j}))\right)\Big)
\end{aligned} \tag{2.56a}$$

$$\begin{aligned}
\alpha_0 v^* = \sum_{j=0}^{2} -\alpha_{j+1} v^{n-j} + \beta_j \Big(& F_{(y)}(\vec{u}^{n-j}, \rho^{n-j}) - \\
& \frac{1}{2}\left(u^{n-j}\mathsf{F}_x^{-1}(\mathbf{i}k\mathsf{F}_x(v^{n-j})) + v^{n-j}\mathsf{F}_y^{-1}(\mathbf{i}l\mathsf{F}_y(v^{n-j})) + w^{n-j}\mathsf{F}_z^{-1}(\mathbf{i}m\mathsf{F}_z(v^{n-j}))\right) - \\
& \frac{1}{2}\left(\mathsf{F}_x^{-1}(\mathbf{i}k\mathsf{F}_x(u^{n-j}v^{n-j})) + \mathsf{F}_y^{-1}(\mathbf{i}l\mathsf{F}_y(v^{n-j}v^{n-j})) + \mathsf{F}_z^{-1}(\mathbf{i}m\mathsf{F}_z(v^{n-j}w^{n-j}))\right)\Big)
\end{aligned} \tag{2.56b}$$

$$\begin{aligned}
\alpha_0 w^* = \sum_{j=0}^{2} -\alpha_{j+1} w^{n-j} + \beta_j \Big(& F_{(x)}(\vec{u}^{n-j}, \rho^{n-j}) + g\rho^{n-j} - \\
& \frac{1}{2}\left(u^{n-j}\mathsf{F}_x^{-1}(\mathbf{i}k\mathsf{F}_x(w^{n-j})) + v^{n-j}\mathsf{F}_y^{-1}(\mathbf{i}l\mathsf{F}_y(w^{n-j})) + w^{n-j}\mathsf{F}_z^{-1}(\mathbf{i}m\mathsf{F}_z(w^{n-j}))\right) - \\
& \frac{1}{2}\left(\mathsf{F}_x^{-1}(\mathbf{i}k\mathsf{F}_x(u^{n-j}w^{n-j})) + \mathsf{F}_y^{-1}(\mathbf{i}l\mathsf{F}_y(v^{n-j}w^{n-j})) + \mathsf{F}_z^{-1}(\mathbf{i}m\mathsf{F}_z(w^{n-j}w^{n-j}))\right)\Big)
\end{aligned} \tag{2.56c}$$

$$p^* = \alpha_0 \mathsf{F}_{xyz}^{-1}\left(\frac{\mathbf{i}k\mathsf{F}_x(u^*) + \mathbf{i}l\mathsf{F}_y(v^*) + \mathbf{i}m\mathsf{F}_z(w^*)}{k^2+l^2+m^2}\right) \tag{2.56d}$$

$$u^{n+1} = \mathsf{F}_{xyz}^{-1}\left(\frac{\mathsf{F}_{xyz}(u^* - \mathsf{F}_x^{-1}(\mathbf{i}k\mathsf{F}_x(p^*)))}{\alpha_0 - \nu(k^2+l^2+m^2)}\right) \tag{2.56e}$$

$$v^{n+1} = \mathsf{F}_{xyz}^{-1}\left(\frac{\mathsf{F}_{xyz}(v^* - \mathsf{F}_y^{-1}(\mathbf{i}l\mathsf{F}_y(p^*)))}{\alpha_0 - \nu(k^2+l^2+m^2)}\right) \tag{2.56f}$$

$$w^{n+1} = \mathsf{F}_{xyz}^{-1}\left(\frac{\mathsf{F}_{xyz}(w^* - \mathsf{F}_z^{-1}(\mathbf{i}m\mathsf{F}_z(p^*)))}{\alpha_0 - \nu(k^2+l^2+m^2)}\right) \tag{2.56g}$$

$$\begin{aligned}
T^{n+1} = -\mathsf{F}_{xyz}^{-1}\Bigg(& \frac{1}{\alpha_0 - \kappa(k^2+l^2+m^2)}\mathsf{F}_{xyz}\Big(\sum_{j=0}^{2} \alpha_{j+1}T^{n-j} + \\
& \beta_j\left(u\mathsf{F}_x^{-1}(\mathbf{i}k\mathsf{F}_x(T^{n-j})) + v\mathsf{F}_y^{-1}(\mathbf{i}l\mathsf{F}_y(T^{n-j})) + w\mathsf{F}_z^{-1}(\mathbf{i}m\mathsf{F}_z(T^{n-j}))\right)\Big)\Bigg)
\end{aligned} \tag{2.56h}$$

This algorithm is substantially similar to that used by Winters et al. [2004], although that model allows for only a constant timestep (with third-order Adams-Bashforth timestepping), with viscosity treated via an integrating factor approach.

Solid boundaries

Implementing general boundary conditions with a Fourier method is difficult. The underlying expansion assumes symmetry, and boundary conditions generally do not respect this symmetry. After periodic extension, the boundaries of the domain are adjacent to each other, and any incompatibility between the boundary conditions manifests as a discontinuity in the underlying function (or its derivatives). This discontinuity gives slow convergence.

In special cases, boundary conditions can be enforced through altering the symmetry of the problem. If (in one dimension) a continuous function has odd symmetry about the domain endpoints, then by definition its value at the endpoints will be zero. Likewise, if a C^1 function has even symmetry about the domain endpoints, then it must have zero-derivative there.

For inviscid flow, at solid walls the physical boundary conditions ($\hat{\mathbf{n}} \cdot \vec{u} = 0$) allow for just this sort of symmetry. If the flow is reflected about a wall such that tangential velocities have even symmetry and normal velocities have odd symmetry, then the no normal flow physical condition is automatically satisfied. Additionally, this reflection is exact for potential (irrotational) flow [Kundu and Cohen, 2004].

Numerically, there are two ways to implement this symmetry. The first is to extend the domain of f from $[0,L]$ to $[-L,L]$, and apply the Fourier transform (2.51) on a domain of twice the length. The grid spacing is unchanged, so the wavenumbers k still range from $-\frac{\pi}{\Delta x}$ to $\frac{\pi}{\Delta x}$, just in steps of $\frac{\pi}{L}$ (and as with the full Fourier transform, only frequencies from 0 to $\frac{\pi}{\Delta x}$ are of interest). The additional symmetry prevents this reflection from doubling the size of the problem.

Consider a single wave on this periodically extended grid, so that $\mathsf{f}(k)$ is nonzero at only a single pair of wavenumbers $k = \pm\frac{\pi j}{L}$. Letting $f(+k) = A_r + \mathbf{i}A_i$ and substituting into (2.51b) gives:

$$f(x) = 2A_r \cos\left(\frac{\pi jx}{L}\right) - 2A_i \sin\left(\frac{\pi jx}{L}\right). \tag{2.57}$$

For a symmetric problem, the constraint that $f(0)$ ($\frac{\partial}{\partial x} f(x)|_{x=0}$) is zero gives the requirement that the cosine (sine) term vanishes, thus A_r (A_i) is identically zero. Since (2.57) applies for an arbitrary k, it holds true over the entire range, and $\mathsf{f}(k)$ is wholly real or imaginary, rather than complex-valued.

The second way of implementing this symmetry is to incorporate it directly into the Fourier transform (2.51). Instead of embedding the domain in one of twice the length, only including the parts of $\exp(\mathbf{i}kx)$ with appropriate symmetry gives revised transforms, S for sine and C for

cosine:

$$\mathsf{S}(f(x)) = \frac{1}{2\pi N} \sum_{j=0}^{N-1} f(\Delta x(j+\tfrac{1}{2})) \sin(k\Delta x(j+\tfrac{1}{2})) \tag{2.58a}$$

$$\mathsf{S}^{-1}(\mathsf{f}(k)) = \sum_{j=0}^{N} f\left(\frac{\pi j}{L}\right) \sin\left(\frac{\pi j}{L} x\right) \tag{2.58b}$$

$$\mathsf{C}(f(x)) = \frac{1}{2\pi N} \sum_{j=0}^{N-1} f(\Delta x(j+\tfrac{1}{2})) \cos(k\Delta x(j+\tfrac{1}{2})) \tag{2.58c}$$

$$\mathsf{C}^{-1}(\mathsf{f}(k)) = 2f(0) + \sum_{j=1}^{N} f\left(\frac{\pi j}{L}\right) \cos\left(\frac{\pi j}{L} x\right). \tag{2.58d}$$

Algebraic derivative relationships between the transforms also hold, as in the periodic case, but here they mix transform types. If f is even-symmetric about the boundary, then f' is odd-symmetric, and:

$$\frac{\mathrm{d}}{\mathrm{d}x} f(x) = \mathsf{S}^{-1}(-k\mathsf{C}(f(x))), \tag{2.59a}$$

whereas if f is odd-symmetric, then f' is even-symmetric, and:

$$\frac{\mathrm{d}}{\mathrm{d}x} f(x) = \mathsf{C}^{-1}(k\mathsf{S}(f(x))) \tag{2.59b}$$

In (2.58), as opposed to (2.51), the grid is actually specified. For $x \in [0, L]$ with grid spacing Δx, the grid points are taken to lie at the half-way marks: $x = \frac{\Delta x}{2} + n\Delta x$. This choice is not arbitrary, and it comes from the boundary conditions implied by the symmetry.

For a function with odd symmetry about the boundary, $f(0) = f(L) = 0$. Placing any grid points at the boundary is meaningless because of that constraint. This is not a problem for a function with even symmetry, but differentiating also changes the symmetry type. $\cos(\frac{\pi}{L}x)$ is perfectly well-represented with a grid of two points, even including endpoint placement, but its derivative is not well-represented on the endpoint grid. This is illustrated in figure 2.10.

Applying even and odd-symmetry to flow in a box gives the following modifications to the periodic algorithm (2.56):

- In the momentum equations (2.56a-2.56c), the advection term corresponding to a velocity's own direction has odd-symmetry, although component terms may have even-symmetry. For example, $uu_x = u\mathsf{C}_x^{-1}[k\mathsf{S}_x[u]]$, and $(uu)_x = \mathsf{S}_x^{-1}[-k\mathsf{C}_x[uu]]$.
- Advection terms corresponding to the other directions have even-symmetry, although sub-components may have odd-symmetry. Repeating the previous example, $vu_y = v\mathsf{S}_[(-l\mathsf{C}_y[u]]$ and $(uv)_y = \mathsf{C}_y^{-1}[l\mathsf{S}_y[uv]]$.

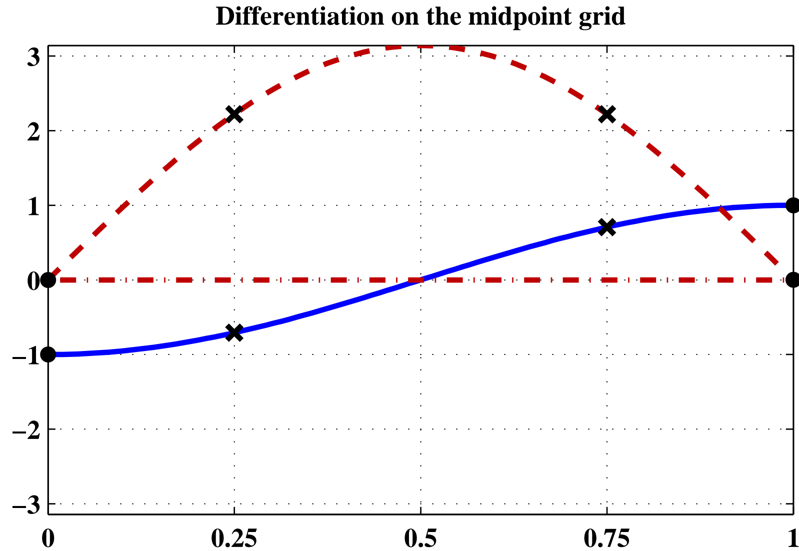

Figure 2.10: Possible two-gridpoint layouts for $-\cos(\pi x)$ (solid) and its derivative (dashed). With gridpoints at the ends of the interval (bullets), the derivative is numerically zero (dash-dotted). Placing the gridpoints at $\frac{\Delta x}{2} + n\Delta x$ (crosses) does not have this problem.

- Pressure (2.56d) has even-symmetry about the boundaries, so F_{xyz} and its inverse become C_{xyz}. The divergence involves strictly odd-symmetry, so those derivatives are replaced according to (2.59b).

- The viscosity terms (2.56e-2.56g) involve a mix of symmetries, but are otherwise unchanged. For example, (2.56e) will involve $\mathsf{S}_x\mathsf{C}_{yz}$ and its inverse. The derivative of pressure has odd symmetry, and so is replaced according to (2.59a).

- From physical reasoning, a component tracer with nonzero diffusion should have no flux through the (insulated) boundary[19], giving even-symmetry. The F_{xyz} in (2.56h) is replaced by C_{xyz}, and the individual derivatives in the advective terms can be taken according to (2.59a).

[19]This is appropriate if T represents salinity or temperature in an insulated domain. Other, more general physical boundary conditions are reasonable, but do not lend themselves so well to a single symmetry type

2.3.3 Polynomial methods

Unfortunately, the Fourier method of (2.56), even with even or odd symmetric extensions from (2.58), fails to properly handle viscous boundary conditions. Consider the case of plane Poiseuille flow, steady, parallel flow between two flat, stationary plates driven by a pressure gradient [Kundu and Cohen, 2004]. Here, $\vec{u}$ reduces to strictly $u(z)$ and the flow is by construction divergence-free, so the Navier-Stokes equations simply reduce to:

$$\nu u_{xx} = \frac{1}{\rho} p_x, \tag{2.60}$$

with the no-slip conditions of $u(\pm L) = 0$. Letting L, ρ, and ν be 1 and $p_x = -2$ gives the solution $u(z) = 1 - z^2$.

This solution, however, is incompatible with a trigonometric basis. Since $u(\pm 1) = 0$, it would seemingly make sense to expand u in terms of $\sum_k \alpha_k \cos(\pi \frac{k+1}{2} z)$[20], but this series converges only slowly, with α_k proportional to k^{-3}. Truncating the series after N terms gives an overall error of $\mathrm{O}(N^{-2})$, far worse than the excellent interpolation performance of figure 2.8.

In fact, for this particular case the picture is even worse. A second-order finite difference method (with the Lagrange interpolation (2.43)) reconstructs quadratic polynomials *exactly*, so with a mere three points a finite difference method will solve (2.60).

This reduction in accuracy comes from an incompatibility at the boundary. The boundary conditions on (2.60), that $u(\pm 1) = 0$, do not constrain u_z, u_{zz}, or higher derivatives. However, by expanding u in the cosine basis, the expansion assumes u has odd symmetry about the boundary. This implies by construction not only $u(\pm 1) = 0$, but $\frac{\partial^{2n}}{\partial z^{2n}} u(\pm 1)$ (the even derivatives) must also be 0. This is explicitly incompatible with (2.60), where $u_{zz} = -2$ everywhere with the given values. The incompatibility results in Gibbs oscillations[21] in the second derivative.

Removing this incompatibility while preserving high order accuracy suggests the use of a polynomial method. The problem with the polynomial interpolants of the form (2.43) is the discontinuities associated with moving the interpolation stencil. On a finite domain, however, this presents no problem – by including every point in the domain in the stencil, there is no need to shift stencils around grid points. The resulting interpolating polynomial on the N points has degree $N - 1$, should have $\mathrm{O}(\Delta x^N)$ accuracy.

Runge phenomenon and nonuniform grids

As always, the trouble is in the implementation. Directly extending (2.43) to a uniformly spaced grid of finite extent is extremely problematic. While the computation of the Lagrange interpolant

[20] This would be a sine expansion if the domain were $x \in [0, 2]$ rather than $[-1, 1]$.

[21] At a discontinuity, Gibbs oscillations make a truncated Fourier series overshoot the jump. As $N \to \infty$, the function converges in $\|\cdot\|_2$ but not $\|\cdot\|_\infty$.

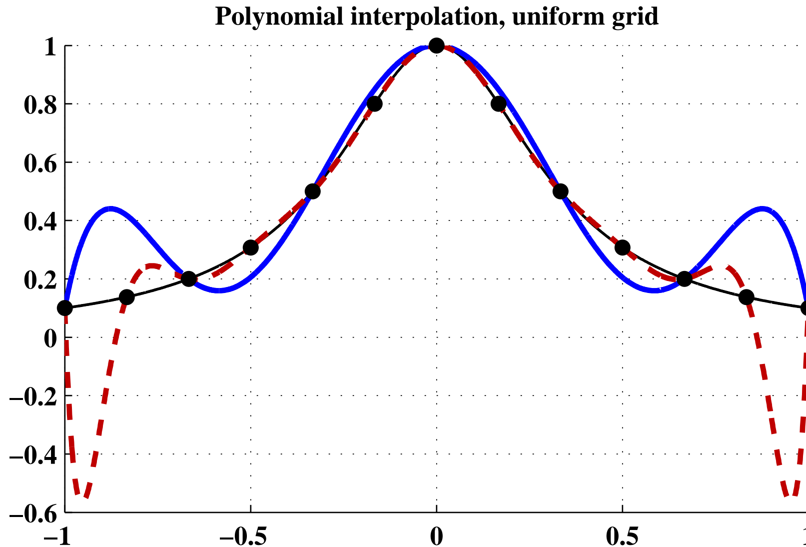

Figure 2.11: Polynomial interpolation of $(1+9x^2)^{-1}$ (thin, solid) using an equally spaced grid at sixth order (thick, solid) and twelfth order (dashed). While the interpolation is good in the central part of the domain, the interpolant oscillates wildly near the ends. These oscillations become worse at higher order.

is extremely simple, the interpolant is generally *not* a good approximation of the underlying function.

Figure 2.11 demonstrates this for the smooth, well-behaved function $(1+9x^2)^{-1}$. Even though the underlying function is infinitely differentiable, the Nth order polynomial interpolation *diverges* near the ends of the domain for increasing N. This is an example of the Runge Phenomenon [Boyd, 2001], and in brief it reflects that the function's singularity at $\pm\frac{1}{3}\mathbf{i}$ destroys the interpolation's convergence even on the real interval $[-1,1]$.

That the oscillations occur near the endpoints suggests a possible remedy. By placing proportionally more grid points near the boundary, the oscillations can be eliminated. A fixed skew will not work, however; as $N \to \infty$, if the grid points near the boundary were spaced at a finite ratio C to that in the center[22], then the Runge phenomenon will return if the function is stretched appropriately.

Instead, eliminating the Runge phenomenon requires a *singular* transformation, where in the case $N \to \infty$ the grid points near the boundary become spaced infinitely more closely than the grid points near the middle (that is, $\Delta x_{bdy}/\Delta x_{mid} \to 0$ as $N \to \infty$). One such transformation is to let $x = \cos(\theta)$, where θ is evenly spaced on $[0,\pi]$. This transformation is illustrated in figure

[22] That is, at the boundary the spacing is $C\Delta x$ $(0 < \varepsilon < C < 1)$ when at the center the spacing is Δx.

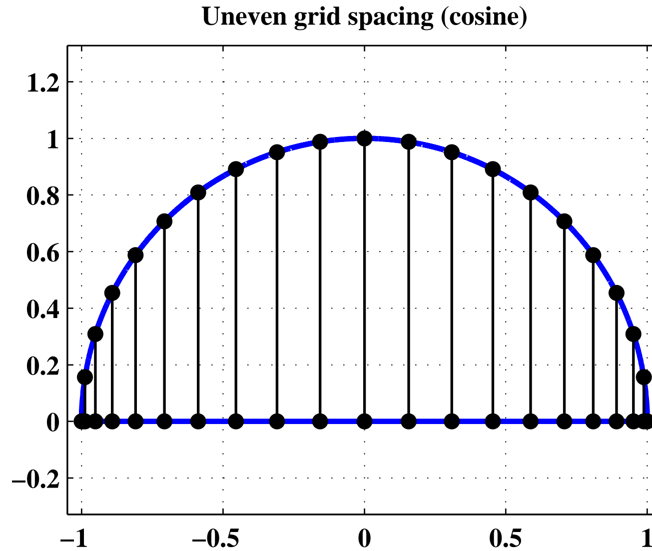

Figure 2.12: Illustration of the grid mapping $x = \cos(\theta)$, for $N = 21$ points. Although the points are equally spaced around the semicircle $z = \sqrt{1-x^2}$, when projected onto the x-axis points cluster near the boundaries, with spacing proportional to N^{-2}, compared to N^{-1} in the middle of the domain. Reproduced from Boyd [2001].

2.12.

This transformation has a few additional properties that make it particularly suitable. Firstly, the underlying θ variable is discretized evenly, and this will allow for a simple evaluation algorithm. Secondly and most importantly, this transformation ensures the first (and all odd) derivatives with respect to θ are zero at the boundaries.

For proof, first make the fairly general assumption that f is analytic in some finite region around[23] $x = 1$, where $\theta = \cos^{-1}(1) = 0$. Since f is analytic, it has a Taylor series that converges in a neighbourhood of $x = 0$, giving:

$$f(x) = f(1) + f_x(1)(x-1) + \frac{f_{xx}(1)}{2}(x-1)^2 + \cdots . \tag{2.61}$$

Now, make the substitution $x = \cos(\theta)$, and expand the cosine in terms of its Taylor series about $\theta = 0$:

$$f(x) = \sum_{i=0}^{\infty} \frac{f_{x^{(i)}}(1)}{i!} \left(\sum_{j=1}^{\infty} (-1)^j \frac{\theta^{2j}}{(2j)!} \right)^i . \tag{2.62}$$

[23] The proof in this section repeats about $x = -1$ and $\theta = \pi$ with only small modifications.

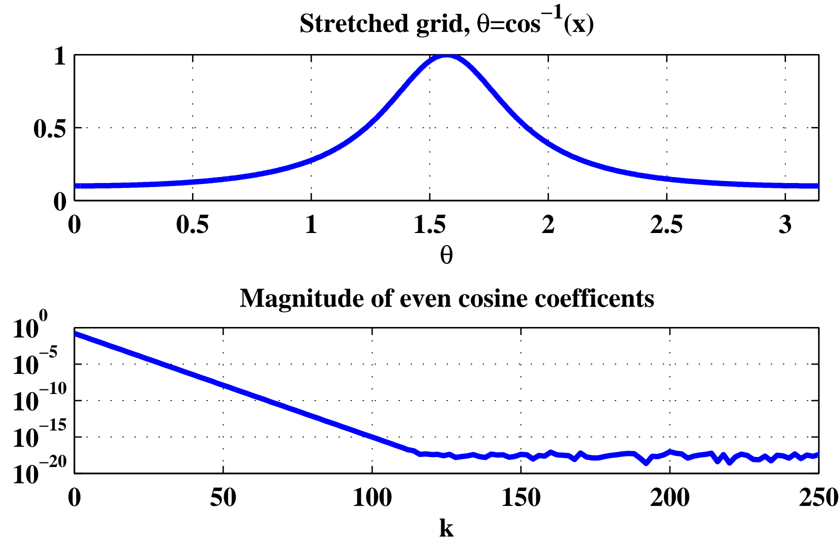

Figure 2.13: $(1+9x^2)^{-1}$ under the mapping $x = \cos(\theta)$ (top), along with the magnitude of the even coefficients in its cosine series expansion (bottom, log scale). The odd coefficients are zero due to symmetry, and the even coefficients decay exponentially in magnitude, reaching the level of rounding error after $k = 120$.

Now, $\frac{\partial^n}{\partial \theta^n} f$ is equal to $n!$ times the θ^n term in the mess of (2.62). However, θ *appears only in even powers*, so the odd derivatives are all zero.

This conclusion is not just a curiosity. Having all the odd derivatives exactly equal to zero at the boundaries is a requirement for having a quickly-convergent cosine (even) expansion. Indeed, applying this mapping to $(1+9x^2)^{-1}$ gives a very rapidly converging cosine series, as shown in figure 2.13.

This discussion is here in section 2.3.3 on polynomial methods because under the transform $x = \cos^{-1}(\theta)$, the cosine series $\mathsf{C}_\theta(f)$ of f on θ is *identical to a polynomial expansion on x.* This is obvious for low orders:

$$T_0(x) = \cos(0 \cdot \cos^{-1}(x)) = 1 \tag{2.63a}$$

$$T_1(x) = \cos(1 \cdot \cos^{-1}(x)) = x, \tag{2.63b}$$

and for higher degrees the polynomials are given by the recurrence relation[24]:

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x). \tag{2.63c}$$

[24]Proof follows from the identity $\cos((n+1)\theta) + \cos((n-1)\theta) = 2\cos(n\theta)\cos(\theta)$.

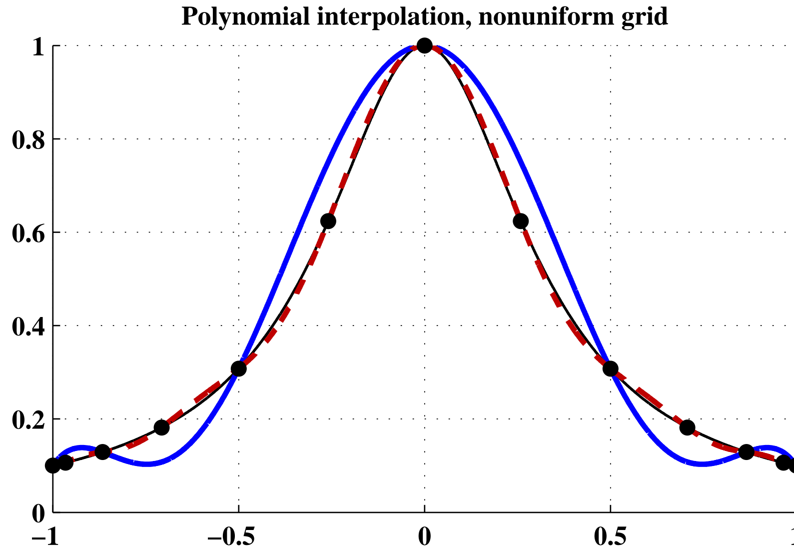

Figure 2.14: Polynomial interpolation as in figure 2.11 on the nonuniform grid $x_j = \cos(\pi \frac{j}{N-1})$ at sixth order (solid) and twelfth order (dashed). Unlike on the equispaced grid, the approximations quickly converge to the true function.

These polynomials are the Chebyshev polynomials [Boyd, 2001]. They are also generated as the (bounded) solutions to the differential equation $(1-x)^2\partial_x^2 T_n - x\partial_x T_n + n^2 T_n = 0$ on $x \in [-1,1]$, although with this form the relationship to the cosine transform is not obvious.

As expected from the results of figure 2.13, interpolating using the non-equispaced grid converges quickly. The example of figure 2.11 is repeated on this grid with figure 2.14.

Differentiation of a function expressed as a sum of Chebyshev polynomials ($f(x) = \sum_k \alpha_k T_k(x)$) is simple. The most straightforward way is to again consider the series as a modified cosine series in the variable θ, perform differentiation in that variable, and transform back to the variable x using the chain rule:

$$\frac{\partial f(x)}{\partial x} = \frac{\partial f(\theta)}{\partial \theta}\frac{\partial \theta}{\partial x} = \frac{1}{\sin(\theta)}\frac{\partial f(\theta}{\partial \theta}. \tag{2.64}$$

Evaluating f_θ is simple, given the differentiation formula of (2.59a). Expressed in series form, this gives:

$$\frac{\partial f(x)}{\partial x} = \frac{1}{\sin(\theta)}\sum_{k=0}^{N-1} -k\alpha_k \sin(k\theta), \tag{2.65}$$

with the substitution $x_j = \cos^{-1}(\frac{\pi j}{N-1})$ to evaluate (2.65) at grid points. This expression for the derivative has removable singularities at $\theta = 0, \pi$, where the sine terms are all zero. Applying L'hôpital's rule at the endpoints gives the proper value.

This method is preferred by Boyd [2001] for its conceptual simplicity, since differentiation is treated identically to a Fourier cosine method with a change-of-variables. However, since f is already expanded in polynomial form, it is possible to write its derivative directly in that form, avoiding the need for special expansions for the endpoints. From Boyd [2001, Appendix A], given $f(x) = \sum_{k=0}^{N-1} \alpha_k T_k(x)$, $f'(x) = \sum_{k=0}^{N-2} \beta_k T_k(x)$ is given by:

$$\beta_k = \frac{1}{1+\delta_{0k}} \left(2(k+1)\alpha_{k+1} + \beta_{k+1}\right), \tag{2.66}$$

where δ_{0k} is 1 if $k = 0$ and β_{N-1} (the coefficient of the highest-order polynomial in the expansion of f) is taken to be 0, and the recurrence relation is evaluated from $k = N-2$ down to $k = 0$.

The differentiation in (2.66) has the advantage of not changing spectral representations, so the same cosine transform used to compute α_k can be inverted to find the derivative at gridpoints. Additionally, this form requires only a single extra variable for storage (for α_{k+1}) when computed in-place, and it does not require the relatively expensive computation (or pre-calculation) of sine terms.

For analysis of this method, it is also convenient to look at the differentiation matrix. The matrix D is the application of differentiation, incorporating both the cosine transform and its inverse. For a grid of N points where $x_i = \cos(\pi \frac{i}{N-1})$ $(0 \le i \le N-1)$ and letting $M = N-1$ for notational simplicity, the off-diagonal entries of D are given by [Trefethen, 2000]

$$D_{ij} = \frac{1+\delta_{i0}+\delta_{iM}}{1+\delta_{j0}+\delta_{jM}} \frac{(-1)^{i+j}}{x_i - x_j}, \tag{2.67}$$

and the diagonal elements are set such that the row-sum of D is zero[25].

One property of this differentiation matrix is that the (well-resolved) eigenvalues of the continuous differential operator are well-approximated by the discrete eigenvalues of the matrix. To demonstrate, consider the simple problem:

$$\frac{\partial^2}{\partial x^2} u = -\lambda u \tag{2.68}$$

with the Dirichlet boundary condition $u(\pm 1) = 0$. The exact solution is the sinusoid modes $\sin(\sqrt{\lambda_n}(x+1))$, with $\lambda_n = (\frac{n\pi}{2})^2$.

The discrete approximation to ∂_x^2 is simply D^2, and the boundary conditions can be applied by removing the first and last row and column from the D^2 matrix[26], making it size $(N-2)\times(N-2)$

[25] There is an explicit formula for the diagonal elements, but setting them in this manner is equivalent and is more stable with finite-precision arithmetic. With the zero row-sum condition, application of this matrix to a constant vector gives exactly zero.

[26] Removing the first and last columns has the effect of treating the first and last (boundary) points as zero; removing the first and last rows reflects that the eigenvalue problem applies only in the interior.

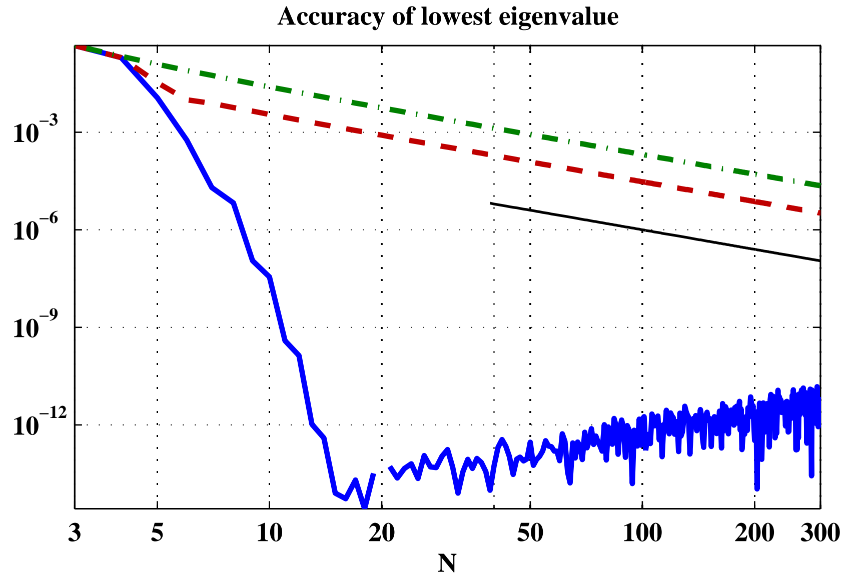

Figure 2.15: Accuracy of the lowest eigenvalue solution to $u_{xx} = -\lambda u$ ($\lambda = \frac{pi^2}{4}$) when the problem is discretized on N points with the Chebyshev differentiation matrix (2.67) (solid), second order finite differences on the same grid (dashed), and second order finite differences on a uniform grid (dot-dashed). The thin line is $O(N^{-2})$ convergence. The Chebyshev differntiation method converges much more quickly, with the error dominated by roundoff after $N = 15$ points.

[Trefethen, 2000]. The resulting accuracy for the lowest eigenvalue is shown in figure 2.15, in comparison with second order finite differences on the same and a uniform grid. The Chebyshev spectral method gives exponential convergence, reaching the level of numerical roundoff by $N =$ 15 points in the grid.

Calculus in multiple dimensions

Just as in the case of the trigonometric transforms, in multiple dimensions each dimension operates independently. That is, if in one dimension $f(x) = \sum_k \alpha_k T_k(x)$, then $f(x,y) = \sum_k \sum_l \alpha_{k,l} T_k(x) T_l(y)$. Most importantly, differentiation by (2.66) also acts independently in the x and y directions; taking the x derivative, for example, proceeds by fixing l in the summation and treating $f(x,y)$ as a number of one-dimensional functions.

The differentiation matrix form of (2.67) also translates directly to two dimensions via the Kronecker product. For an $N \times M$ grid, $x(i,j)$ is given simply by $x(i) = \cos(\pi \frac{i}{N-1})$ and $y(j) = \cos(\pi \frac{j}{M-1})$. For a column-major ordering of points, the full matrix form for ∂_x is $D_x \otimes I_y$, where I_y

is an identity matrix of size $M \times M$. Likewise, the matrix form of ∂_y is $I_x \otimes D_y$. The discretization of the Laplacian, required for (2.16) and (2.17) is equally straightforward. Letting L be the discretization of ∇^2, it is given by:

$$L = D_x^2 \otimes I_y + I_x \otimes D_y^2. \tag{2.69}$$

This tensor product formulation is extremely flexible, and it gives the key advantage of allowing mixed expansion types. For example, fluid flow in a periodic channel can be given simply by expanding in a Fourier basis for the x (and y, for three-dimensional calculations) direction and a Chebyshev-polynomial basis in the z direction. With the Kronecker product used for the operators, the directions do not interfere with each other, and forward operations in two or three dimensions can be considered repeated applications of the underlying one-dimensional operators.

The tensor product grid and nonzero structure of the matrix L are shown in figure 2.16, and here the structure of the matrix reveals a problem for numerical methods. The timestepping of (2.16) requires applying the inverse of L (plus some diagonal weight for the viscous terms), but even though L is sparse, it has a very high bandwidth. In comparison, a simpler differentiation method like the standard 5-point Laplacian stencil has a bandwidth of only N (on an $N \times N$ grid). The expanded bandwidth of (2.69) makes banded solvers impractical – little effort is saved over simply treating L as a full matrix.

Use of specialized sparse solvers such as `UMFPACK` [Davis, 2004a] helps somewhat, as more specialized algorithms can take advantage of the sparsity of L without requiring a banded structure[27]. However, there is still a limit to the gains possible. On an $N \times N$ grid, L has $\mathrm{O}(2N^3)$ nonzero entries (N^2 grid points, and each grid point influences the N points in its row and column). This places a strict lower bound on the amount of work necessary to apply the inverse of L, but this also compares unfavourably with the application of L.

Going forwards and applying the Laplacian operator can be done with the cosine transform ($\mathrm{O}(N^2 \log N)$ on the $N \times N$ grid) and application of (2.66) ($\mathrm{O}(N^2)$ work), for a total $\mathrm{O}(N^2 \log N)$ cost. This is asymptotically faster than *any* possible direct application of the Laplacian matrix, and it motivates the use of iterative, matrix-free methods to solve the implicit steps of (2.16).

2.3.4 Grid mapping

Accounting for fluid-topography interaction requires a proper model of the topography. For models that permit unstructured grids, this is of no difficulty. The underlying triangular grid simply molds around the basic topography, and the basic numerical algorithm can proceed without further change.

[27] While not used for the direct inversion of L, `UMFPACK` will return as a coarse-grid solver in section 2.4.3

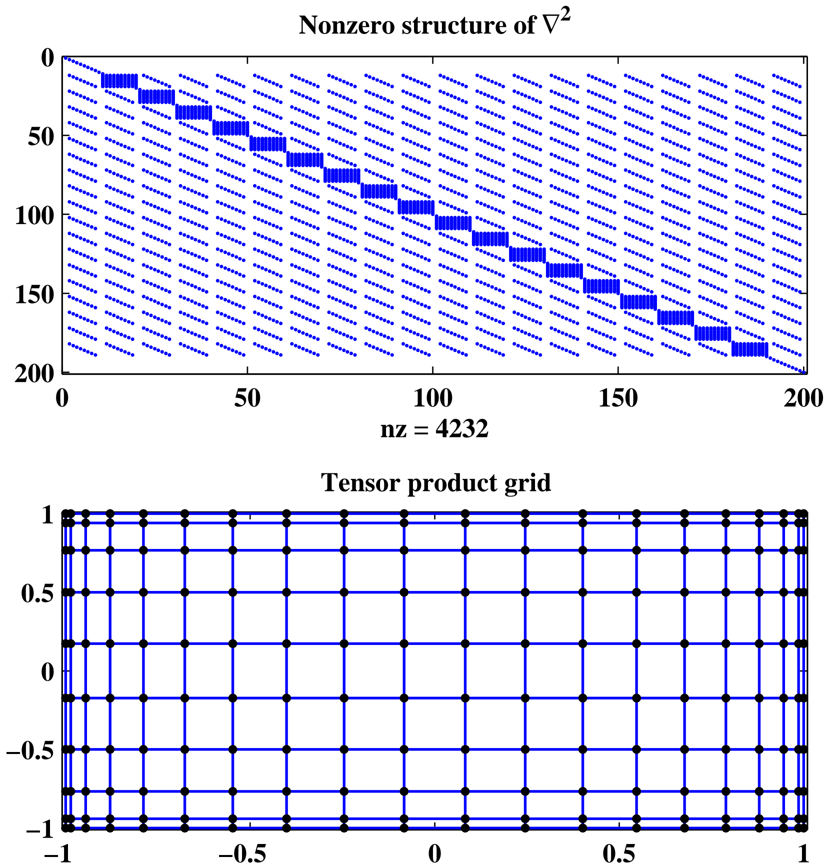

Figure 2.16: Nonzero structure of the discrete Laplacian defined by (2.69) on a 20×10 grid with Dirichlet boundary conditions (top, with nonzero entries as a dot and zero entries blank) and the resulting tensor product grid (bottom). Grid points cluster near the edges and especially corners. The matrix is sparse and structured, with central 10×10 blocks on the diagonal and nonzero entries on 19 non-central diagonals.

Structured grids, such as that used in this work, present a greater challenge. Several approaches to the problem of topography exist in the literature:

- The most direct approach is that of "z-level methods" such as those used in Fringer et al. [2006] or Winters et al. [2004]. In this method, grid cells remain the same size and shape throughout the domain, but portions of the domain that are blocked by topography simply are not discretized[28]. This has the chief advantage that the resolution remains the same throughout the domain, but in exchange topography is only modelled coarsely, with sharp corners at the grid scale. Most relevantly for this work, it's unclear how to adapt a global discretization to a partial grid; Chebyshev methods rely on the singular grid stretching of figure 2.12 to preserve boundary accuracy.

- At the other extreme, Brinkman penalization methods [Keetels et al., 2007] incorporate the topography inside the domain via body forcing. The domain is modelled via a switch function, $I(x,y)$, which is 0 inside the domain and 1 outside. This general form allows for essentially any boundary shape, including interior boundaries (such as for flow past a cylinder) that change the topology of the domain. The momentum equations (2.16) are updated with a body force that, over a very fast time scale, forces all flow outside the domain to 0. In practice, these penalization methods require a smoothed $I(x,y)$, and to be well-resolved with a spectral method it must make the transition over a few grid points. Thus, the boundary can only be localized to $\mathrm{O}(\Delta x)$ accuracy, which partially defeats the purpose of a high-order methods and makes detailed analysis of the boundary layer difficult.

- The method used in this work is the same as that of Lamb [1994], in that the grid is mapped to a "computational box" using a smooth coordinate map, and the grid becomes "terrain-following." The smooth map is applied via a Jacobian operator in the differential equations (2.16), which become spatially varying in the computational coordinates. With this approach, boundary conditions can be specified exactly and the grid retains all of its structure. The mapping requirements are strict, however, and spectral methods require a C^∞ mapping to fully preserve accuracy. As a result, the terrain-following map is unsuitable for sharp topography, such as abrupt shelves.

The process of mapping is a direct – if sometimes notationally cumbersome – application of multivariate calculus. The mapping has an underlying vector-valued map function:

$$\vec{M}(\alpha,\beta) = \begin{bmatrix} x(\alpha,\beta) \\ z(\alpha,\beta) \end{bmatrix}, \tag{2.70}$$

where α and β are the coordinates inside the computational box, taken to be α and β in $[-1,1]$ and x and z are the coordinates in the physical domain. The boundaries of the computational box

[28] Or blocked portions of topography are discretized with "inactive" cells that have no information.

must correspond to the boundaries of the physical domain. The mapping must also fully cover the (finite[29]) physical domain and be (in principle) invertible.

This mapping technique does extend to three dimensions, but this work only (nontrivially) maps two of the dimensions, arbitrarily the x and z dimensions. The solution procedure used for the Poisson and Helmholtz equations of (2.16) requires some adaptation for three dimensions in the presence of mapping, and the necessary modifications are discussed in sections 2.4.3 and 4.4.

Application of the mapping to the underlying differential equations enters through the Jacobian, defined for this work[30] as:

$$J(\alpha,\beta) = \begin{pmatrix} \frac{\partial\alpha}{\partial x} & \frac{\partial\beta}{\partial x} \\ \frac{\partial\alpha}{\partial z} & \frac{\partial\beta}{\partial z} \end{pmatrix}. \tag{2.71}$$

The application of the Jacobian is straightforward via the chain rule:

$$\frac{\partial}{\partial x} = \frac{\partial\alpha}{\partial x}\frac{\partial}{\partial\alpha} + \frac{\partial\beta}{\partial x}\frac{\partial}{\partial\beta} = J_{11}\frac{\partial}{\partial\alpha} + J_{12}\frac{\partial}{\partial\beta} \tag{2.72a}$$

$$\frac{\partial}{\partial z} = \frac{\partial\alpha}{\partial z}\frac{\partial}{\partial\alpha} + \frac{\partial\beta}{\partial z}\frac{\partial}{\partial\beta} = J_{21}\frac{\partial}{\partial\alpha} + J_{22}\frac{\partial}{\partial\beta}. \tag{2.72b}$$

Jacobian calculation

The mapping function $\vec{M}$ need not be analytically specified in order to find (2.71). Provided the mapping can be *numerically* evaluated at each grid point (α_i,β_j) in the computational box, the Jacobian of (2.71) is given by the local application of the Inverse Function Theorem:

$$J(\alpha,\beta) = \begin{pmatrix} \frac{\partial x}{\partial\alpha} & \frac{\partial x}{\partial\beta} \\ \frac{\partial z}{\partial\alpha} & \frac{\partial z}{\partial\beta} \end{pmatrix}^{-1}, \tag{2.73}$$

where the partial derivatives can be numerically computed from (2.52), (2.59), or (2.66), based on the underlying expansion type.

The inversion is given by the matrix inversion, and for the 2×2 matrix of (2.73), the formula is:

$$J(\alpha,\beta) = \frac{1}{x_\alpha z_\beta - x_\beta z_\alpha}\begin{pmatrix} \frac{\partial z}{\partial\beta} & -\frac{\partial x}{\partial\beta} \\ -\frac{\partial z}{\partial\alpha} & \frac{\partial x}{\partial\alpha} \end{pmatrix}. \tag{2.74}$$

[29] Infinite domains can be modeled with a spectral method, and the mapping procedure is discussed in Boyd [2001]. This is beyond the scope of the current work, however, and with such a mapping convergence is generally sub-exponential in order.

[30] This is the transpose of the traditional definition of the Jacobian, used mainly because it allows convenient left-multiplication of the gradient operator. That is, $\nabla_{x,z} = J\nabla_{\alpha,\beta}$.

An important point for this approach is that only a Chebyshev expansion naturally differentiates the transformation, and other bases suffer from Gibbs oscillations. As a trivial example, consider the identity mapping $(\alpha,\beta) \in [-1,1]^2$, with $x(\alpha,\beta) = \alpha$ and $z(\alpha,\beta) = \beta$. If α is expanded in a trigonometric basis, evaluation of $\frac{\partial x}{\partial \alpha}$ will be problematic. If α is expanded as a Fourier (periodic) or sine series, x will effectively have a discontinuity at the endpoints and the expansion will suffer from Gibbs oscillations. Alternatively, if α is expanded as a cosine series, the nonzero derivative of x at the endpoints will result in Gibbs oscillations in the derivatives used in (2.73).

Fortunately, this problem can be mitigated by treating the constant portion of the mapping specially. Assuming that a constant stretching factor C is known in advance, the map (in one dimension for simplicity) can be written as:

$$x(\alpha) = C\alpha + f(\alpha), \tag{2.75}$$

where f reflects the dynamic part of the map. By construction $f(1) = f(-1)$, and x_α can be computed as $C + f'(\alpha)$, for application in (2.73). The derivative f' is much more intuitively related to the underlying transformation. For example, a map that has no skew at the ends will have $f' = 0$, and f' will be computable without oscillation from a cosine transformation.

Mapped Laplacian operator

Applying (2.72) twice and omitting a great deal of algebra gives a formula for the Laplacian operator on the mapped grid:

$$\begin{aligned} \nabla^2 = &(\alpha_x^2 + \alpha_z^2)\frac{\partial^2}{\partial \alpha^2} + (\beta_x^2 + \beta_z^2)\frac{\partial^2}{\partial \beta^2} + 2(\alpha_x\beta_x + \alpha_z\beta_z)\frac{\partial^2}{\partial\alpha\partial\beta} + \\ &\left(\alpha_x\frac{\partial \alpha_x}{\partial \alpha} + \beta_x\frac{\partial \alpha_x}{\partial \beta} + \alpha_z\frac{\partial \alpha_z}{\partial \alpha} + \beta_z\frac{\partial \alpha_z}{\partial \beta}\right)\frac{\partial}{\partial \alpha} + \\ &\left(\alpha_x\frac{\partial \beta_x}{\partial \alpha} + \beta_x\frac{\partial \beta_x}{\partial \beta} + \alpha_z\frac{\partial \beta_z}{\partial \alpha} + \beta_z\frac{\partial \beta_z}{\partial \beta}\right)\frac{\partial}{\partial \beta}, \end{aligned} \tag{2.76}$$

where all of the terms inside the parentheses are either directly given by (2.71) or can be computed from it. Aside from being spatially varying, the most important implication of the mapping in (2.76) is that the $\frac{\partial^2}{\partial\alpha\partial\beta}$ term is nonzero, introducing a mixed second derivative[31].

When discretized with a global expansion, this mixed partial derivative has a *global* pattern. The reasoning is obvious; consider $f(\alpha,\beta) = \delta_{ik}\delta_{jl}$ for some (k,l) grid cell. Differentiating with

[31] In general, the $(\alpha_x\beta_x + \alpha_z\beta_z)$ term is nonzero, but this term would vanish for a conformally mapped grid. Such an assumption, however, prevents the use of any domain that does not have right-angle corners, such as a uniformly sloping bottom.

respect to α influences every point in the same column (second index); differentiating that with respect to β will extend that influence to every point in the domain.

This global pattern, even moreso than the "sparse but not banded" unmapped Laplacian of figure 2.16, requires the use of matrix-free methods for (2.16). On an $N \times M$ grid (in two dimensions), even computing the full mapped Laplacian matrix would require N^2M^2 entries, absolutely dominating the memory cost of the flow variables themselves!

Normal-derivative boundary conditions

Imposing Dirichlet boundary conditions on the mapped Laplacian (2.76) is straightforward, and proceeds in an identical matter. However, the pressure computation in (2.16b) has Neumann-type boundary conditions, where only the derivative is specified in the boundary-normal direction. On an unmapped grid this is no problem, as the normal derivative is always in the direction of a single coordinate, but this assumption no longer holds with a mapping.

Instead, geometric principles can give the proper form and coefficients for the boundary condition, directly from (2.71) and derived quantities.

Consider the domain boundary corresponding to the "bottom" of the computational box ($\beta = -1$). The coordinate line running along that boundary is $(\alpha, -1)$, so the tangent to the boundary is in the direction of α. On the physical grid, that corresponds do (x_α, z_α), and the normal direction is proportional to $(-z_\alpha, x_\alpha)$. Now, the orientation of this vector should be in the negative β direction (corresponding to the outward normal), and β is in the direction of (x_β, z_β). Thus, $\hat{\mathbf{n}}$ is given by:

$$\hat{\mathbf{n}}_{bot} = \frac{\operatorname{sgn}(z_\beta x_\alpha - x_\beta z_\alpha)}{\sqrt{x_\alpha^2 + z_\alpha^2}}(-z_\alpha, x_\alpha). \tag{2.77a}$$

Similarly, for the other boundaires:

$$\hat{\mathbf{n}}_{top} = \frac{-\operatorname{sgn}(z_\beta x_\alpha - x_\beta z_\alpha)}{\sqrt{x_\alpha^2 + z_\alpha^2}}(-z_\alpha, x_\alpha) \tag{2.77b}$$

$$\hat{\mathbf{n}}_{left} = \frac{-\operatorname{sgn}(x_\alpha z_\beta - z_\alpha x_\beta)}{\sqrt{x_\beta^2 + z_\beta^2}}(-z_\beta, x_\beta) \tag{2.77c}$$

$$\hat{\mathbf{n}}_{right} = \frac{\operatorname{sgn}(x_\alpha z_\beta - z_\alpha x_\beta)}{\sqrt{x_\beta^2 + z_\beta^2}}(-z_\beta, x_\beta). \tag{2.77d}$$

The Neumann boundary condition is then given by its usual form ($\hat{\mathbf{n}} \cdot \nabla p = f$, where now the gradient is on the computational box. The one catch in implementation is that the terms in (2.77) are from the *inverse* of the Jacobian matrix (2.73), but computing this locally is just a matter of applying (2.74) a second time.

2.4 Preconditioning & Multigrid

Taking advantage of the easy spectral differentiation of fields requires an iterative solver for the implicit equations in (2.16b) and (2.16c). Unfortunately, such solvers are not simple. For Chebyshev expansions[32], the wide difference in scales between the edges of the grid and central portion of the grid create special numerical issues.

This section will concentrate on the efficient solution of the scalar Poisson's equation (in two dimensions, where appropriate):

$$\nabla^2 u = f, \tag{2.78}$$

with Dirichlet or Neumann boundary conditions as appropriate. This problem is essentially the "hardest" problem to solve in this work. Advection is a matter of explicit stepping, and viscosity turns (2.78) into a well-defined Helmholtz problem that is somewhat simpler to solve, although amenable to the same techniques as Poisson's equation.

Ultimately, the difficulty imposed by Chebyshev methods comes from the eigenvalues of the discretized Laplacian. On a fixed domain (say, $[-1,1]$ in one dimension), the eigenvalues of ($\nabla^2 u = -\lambda u$) with Dirichlet boundary conditions are $(\frac{n\pi}{2})^2$ (see figure 2.15 along with the associated discussion), and the largest frequency that is representable on an equispaced grid is the Nyquist frequency, setting $n = (N-2)$ (the number of interior grid points) as the largest possible mode. This gives an $\mathrm{O}(N^2)$ scaling of the largest mode.

For the particular case of the stretched (cosine) grid for the Chebyshev basis, matters are worse. The maximum eigenvalue scales as $\mathrm{O}(N^4)$ [Boyd, 2001], while the minimum eigenvalue remains very close to $\frac{\pi^2}{4}$. This eigenvalue range is an unfortunate property of the cosine-based grid, and is illustrated in figure 2.17. This scaling is required for an accurate operator. Ignoring aliasing error, consider the boundary point ($x_0 = 1$) and its nearest neighbour, ($x_0 - x_1 = 1 - \cos(\pi N^{-1}) \approx \frac{\pi^2}{2N^2}$). With Dirichlet boundary conditions and a suitably accurate operator[33], the interval can contain a quarter-period of a sine wave, giving an eigenvector of the form $\sin(\frac{N^2}{\pi}(x-1))$, which would have a corresponding eigenvalue of $\frac{N^4}{\pi^2}$.

2.4.1 GMRES

This wide range is of more than theoretical interest; it greatly impacts the convergence of iterative methods. Consider the use of the Generalized Minimum Residual (GMRES, Trefethen

[32]Chebyshev expansions will be considered most in this section, because they provide the most challenging numerical problems. The methods discussed here, however, will apply equally to periodic, sine, and cosine expansions with a suitable change in boundary conditions.

[33]This operator cannot actually exist. By the pigenhole principle, a linear operator with N degrees of freedom cannot accurately operate on up to N^2 eigenfunctions of the continuous Laplacian. The cosine-based grid gets around this by confining high frequency regions to the boundaries.

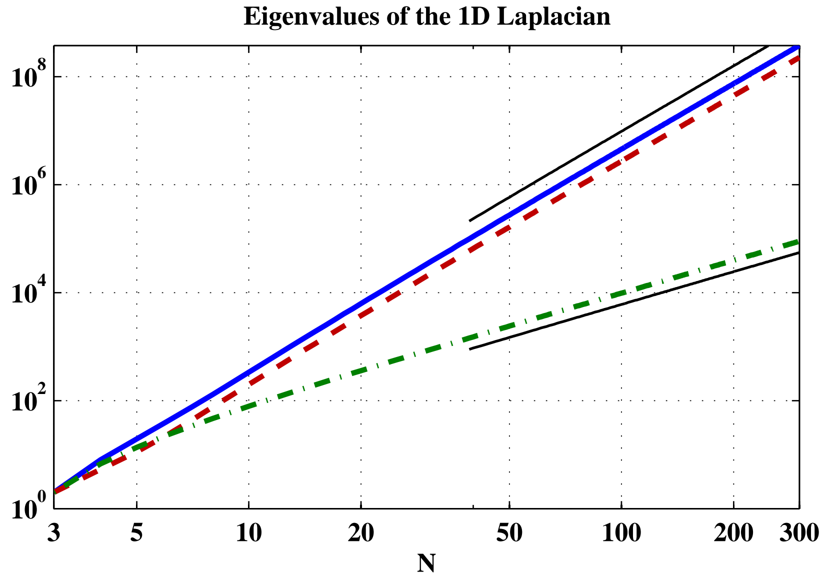

Figure 2.17: Maximum eigenvalue of the 1D discrete Laplacian operator with Dirichlet boundary conditions, as a function of the number of grid points (N). The Chebyshev (solid) expansion and the 3-point finite difference operator on the same grid (dashed) give N^4 scaling (top thin line); equispaced points (dot-dashed) show only $O(N^2)$ growth (bottom thin line).

and Bau, III [1997, sec. 35]) method to solve Poisson's equation (2.78) in two dimensions with Dirichlet boundary conditions. For the matrix problem $A\vec{u} = \vec{f}$, the GMRES method builds a Krylov subspace of $\{\vec{f}, A\vec{f}, A^2\vec{f}, \cdots A^{n-1}\vec{f}\}$, and then finds the $\vec{u}_n$ inside that space that minimizes $\left\|\vec{f} - A\vec{u}_n\right\|$.

This becomes an iterative procedure when the Krylov subspace is formed through an orthogonalization procedure (algorithm 1). A key advantage of the GMRES algorithm is that it makes few assumptions about the underlying operator A. Unlike the three-term recurrence methods, of which the conjugate gradient method is most well-known [Trefethen and Bau, III, 1997], A need not be symmetric, nor does the algorithm involve the transpose A^T. This property is ideal for this work, where use of the Fourier transform and (2.52), (2.59), or (2.66) allows evaluation of $A\vec{u}$ without needing a direct specification of A[34] By constructing successively larger subspaces, the GMRES algorithm finds progressively better iterates $\vec{u}_j$, and it will converge under a wide variety of conditions. By the time it takes n iterations, the Krylov subspace will be the same as the full domain of the problem, and the solution is exact [Trefethen and Bau, III, 1997].

[34]Hypothetically, GMRES could even be applied *analytically* for an infinite-dimensional linear operator, but then it would be impossible to guarantee convergence.

Algorithm 1 The GMRES algorithm.

Set $\vec{r}_0 = \frac{\vec{f}}{\|\vec{f}\|}$
for $j = 1$ to the maximum desired iteration, or until convergence **do**
  Set $\vec{r_{j+1}} = A\vec{r}_j$
  for $k = 1$ to j **do**
    Set $H_{kj} = \vec{r}_{j+1} \cdot \vec{r}_k$
    Set $\vec{r}_k = \vec{r}_k - H_{kj}\vec{r}_{j+1}$
    {At the end of this loop, $\vec{r}_k$ is normal to every other $\vec{r}$}
  end for
  Set $H_{j+1,j} = \left\|\vec{r}_{j+1}\right\|$
  {H is a rectangular matrix of size $(j+1 \times j)$, with zeros below the first subdiagonal.}
  Set $\vec{r}_{j+1} = \frac{\vec{r}_{j+1}}{H_{j+1,j}}$
  Solve $H\tilde{u} = [1,0,0,\cdots]^T$ (in the least squares sense)
  $\vec{u}_j = \left\|\vec{f}\right\| [\vec{r}_1, \vec{r}_2, \cdots, \vec{r}_j]\tilde{u}$ is the solution with minimum residual.
end for

The downside of the GMRES algorithm is that constructing the matrix H is not cheap, at least for large numbers of iterates. Aside from the least-squares solution itself ($O(j^2)$ operations for iterate j naively, and $O(j)$ with Givens rotations [Trefethen and Bau, III, 1997]), the algorithm requires computing the dot product of $A\vec{r}_j$ with every other $\vec{r}_k$, which is $O(Nj)$ work[35] ($O(Nm^2)$ if m is the number of the final iteration). The numerical efficiency of the GMRES algorithm depends crucially on the rate of convergence.

Unfortunately, the direct application of GMRES to the Chebyshev-discretized Laplacian does not converge quickly, as shown in figure 2.18. Performance for (2.78) with Dirichlet boundary conditions is still better than application of a direct solver to the Laplacian, but without modification GMRES still takes $O(N^2)$ iterations. This gives an overall $O(N^4 \log(N))$ workload, since each application of the Laplacian operator is $O(N^2 \log(N))$ work.

2.4.2 Finite difference preconditioning

The spread in eigenvalue of the Chebyshev-discretized Laplacian is responsible for the poor convergence of figure 2.18. The GMRES algorithm is based on the Arnoldi iteration [Trefethen and

[35] One popular variant of GMRES is *restarted GMRES*, where instead of iterating algorithm (1) to convergence, it is stopped after every n iterations and restarted with the then-best-guess. This preserves a monotonic decrease in residual and provides a strict bound on the amount of memory required, but it is impossible to guarantee convergence of the method. This work uses the restarted variant of GMRES, with fairly generous limits about the number of inner iterations allowed.

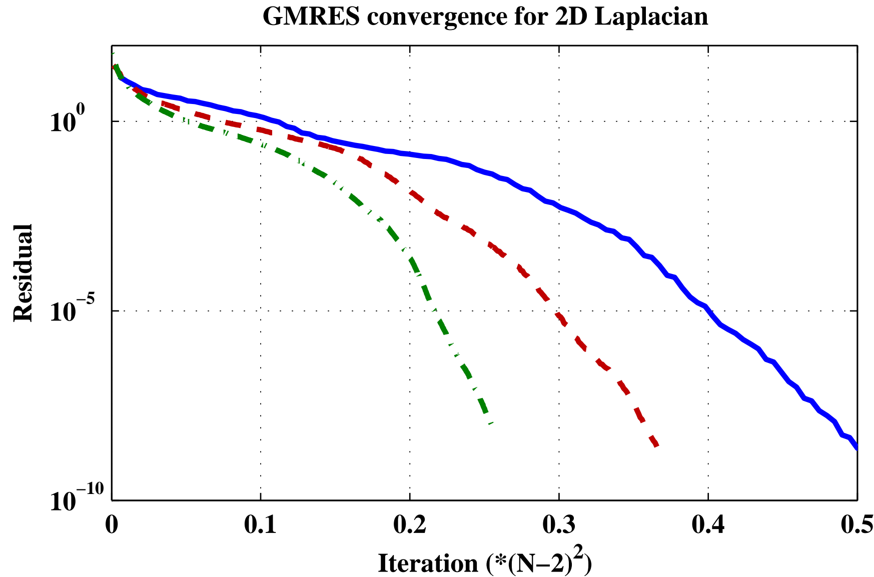

Figure 2.18: Convergence of GMRES for the two-dimensional problem $\nabla^2 u = f$ with zero Dirichlet boundary conditions for a random vector f. The x-axis is scaled by $(N-2)^2$ (the number of interior grid points) for a 16×16 grid (solid), 32×32 (dashed), and 64×64 (dot-dashed).

Bau, III, 1997], which uses the same Krylov subspace to compute eigenvalues. In the Arnoldi iteration, the 1×1, 2×2, etc. square matrices at the top-left of H in algorithm 1 have eigenvalues that (often quickly) converge to the largest-magnitude eigenvalues of A. The implication for the Laplacian is obvious: the GMRES algorithm will quickly isolate and eliminate the large-eigenvalue (high-frequency) part of f in (2.78), but the lower-frequency error will only be reduced after many steps.

This suggests a two-step process. If the low-frequency error can be reduced separately, then the GMRES algorithm should be able to quickly remove the high-frequency error. The standard method of accomplishing this is through preconditioning the GMRES method, which replaces A in algorithm 1 with AP^{-1}, where P^{-1} is the (easily-applied) inverse of some preconditioning operator P.

This definition is vague because there is a great deal of freedom in applying a preconditioner. One common method, naive in that it applies independently of the operator being considered, is for P to be the incomplete LU-factorization of A, with an experimentally-determined drop tolerance[36] [Trefethen and Bau, III, 1997]. This approach still requires building the matrix A,

[36] Another approach for the incomplete LU factorization is to keep the structure of A intact. This is more relevant for highly-sparse finite difference methods.

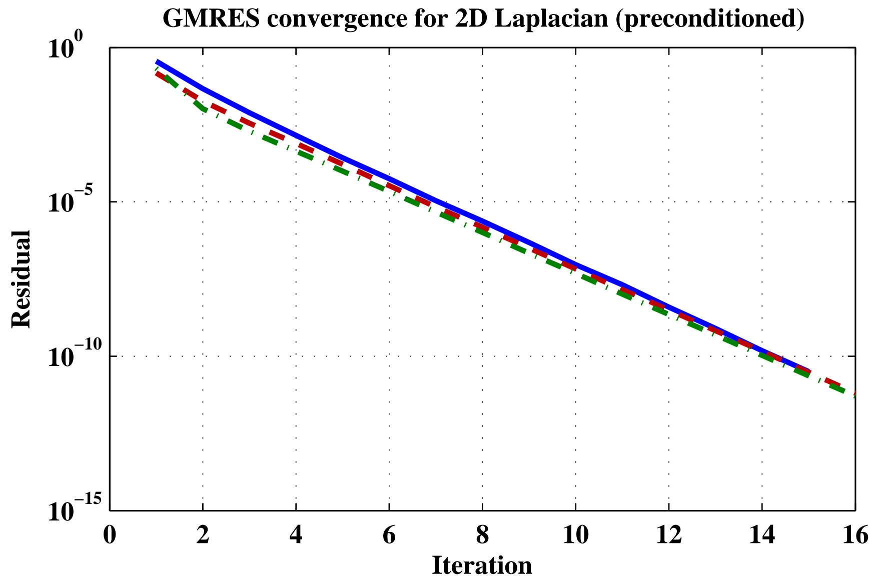

Figure 2.19: Convergence of GMRES for the same 2D problem as figure 2.18, save that the finite difference operator given by the standard three-point stencil is used as a preconditioner. Unlike the unpreconditioned iteration, convergence is extremely rapid and proceeds at a nearly identical rate for the 16×16 (solid), 32×32 (dashed), and 64×64 (dot-dashed) grids. In MATLAB, using the matrix forms of the operator, solution of the 64×64 case took only 1% of the unpreconditioned time.

however, so it is unsuitable for this work. It has been used successfully for low-resolution problems, though [Soontiens et al., 2010].

Another approach to separate the lower-frequencies from the GMRES iteration is through a geometric multigrid method applied to A. Multigrid, described in more detail in section 2.4.3, uses a lower-resolution version of A to approximately solve for the lower frequencies in a recursive method. This approach is suggested in Canuto et al. [1988], although it requires a fairly complex nested iteration, and it is implemented there with further preconditioning for the high frequencies.

A simpler approach to the preconditioning is possible. The *finite difference* operator on the same grid has many of the same properties as the full spectral expansion, and it is far easier to invert. This approach is fairly standard in spectral methods, and it is described in Boyd [2001, sec. 15.3] and Canuto et al. [1988, sec. 5.2]. The qualitative rationale is obvious; for sufficiently high resolution, a finite difference method will still resolve the lower frequencies to a couple of digits accuracy (see figure 2.15). The eigenvalue scaling in figure 2.17 also offers hope that the higher frequencies will also be helped somewhat by a finite difference preconditioner, even if the

operator is only accurate to first-order[37].

Indeed, the finite difference operator is an excellent preconditioner, as shown in figure 2.19. Convergence of the preconditioned algorithm is excellent and independent of resolution, gaining approximately two decimal digits of accuracy per three iterations. This result comes from the preconditioner's performance for the *high frequency* components; better than just being "helped somewhat," the eigenvalues of the composite operator $\nabla^2(\nabla^2_{fd})^{-1}$ are bounded between 1 (for the low frequencies) and approximately 2.5 (for the high frequencies)[38] [Boyd, 2001, sec. 15.3].

Efficiency

The finite difference preconditioning accelerates the convergence of the GMRES algorithm to the point that the costs of algorithm 1 are not asymptotically significant. That's not to say that the iteration is *free*, just that if convergence is obtained in a (small) resolution-independent number of iterations, the costs of GMRES itself do not scale with problem size. Instead, the dominant factor in performance comes from the cost of an individual iteration.

Without modification, finite difference preconditioning still imposes significant costs. With the tensor-product grid of figure 2.16 and a lexical ordering, the finite difference operator in matrix form has five nonzero diagonals. Unfortunately, (on an $N \times N$ grid) the matrix has bandwidth $2N$ – the Laplacian at point j is determined[39] by the values at j, $j \pm 1$, and $j \pm N$. Direct LU-factorization of this matrix for a direct solver fills in the zero entries between the outer diagonals and the main diagonal, giving $O(N^3)$ nonzero entries [Boyd, 2001, sec. B.2], and consequently the same order of work in the solution. This is better than the $O(N^4)$ implied by unpreconditioned GMRES, but still worse than the $O(N^2 \log(N))$ for a single application of ∇^2.

Generally, spectral methods avoid this issue by weakening the preconditioner slightly. Instead of using the full finite difference operator (or equivalently its LU factors), using the *incomplete* LU factorization, where the sparse structure of the matrix is preserved by only keeping entries in the L and U matrices that are in the same locations as entries in the ∇^2_{fd} discretization. This keeps the number of entries down to $O(N^2)$, but it makes convergence once again depend on resolution [Canuto et al., 1988, sec. 5.4.2][40].

[37] The formal accuracy of the three-point stencil for the second derivative on a nonuniform grid is $O(\Delta x^{-1})$, where the commonly-used second-order accuracy comes from error cancellation. However, this is a pessimistic estimate. On the cosine-mapped grid, where Δx is largest the grid is nearly uniform, and where the skew is largest Δx is $O(N^{-2})$.

[38] This result is derived for the periodic Fourier discretization, but it effectively generalizes to the Chebyshev discretization because of the grid transformation.

[39] This result is for an unmapped grid. If the grid is mapped via (2.76), the cross-derivative term makes this a 9-point stencil, of bandwidth $2N+2$.

[40] Canuto et al. [1988] and Boyd [2001] discuss restoring resolution-independent convergence by using the incomplete LU factorization as the smoother in a spectral multigrid iteration. This work takes a more direct approach.

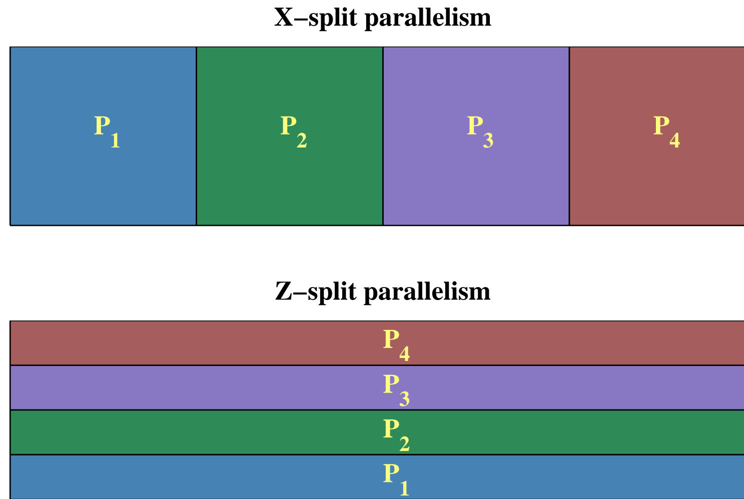

Figure 2.20: Schematic for splitting a computational array among four processor nodes. By default, the array has a number of contiguous y/z planes belonging to each individual processor (top), and Fourier-type transformations can be taken along those dimensions. When a transformation along the x dimension is needed, the array is transposed (bottom) so that each processor has contiguous x/y planes.

Parallelism

Even worse for the purposes of this work, both the direct solution of the finite difference operator and the backsolves of the incomplete LU factorization do not parallelize well. For problem sizes relevant to geophysical applications, N^2 (or N^3 in three dimensions) is very large by itself, making distributed-memory parallelism ideal. However, the backwards substitution required to apply L^{-1} and U^{-1} introduce data dependencies that severely limit the possible parallelism.

This limit on parallelism is a fundamental problem to overcome for the preconditioner. At first glance, the Fourier method, as a global transformation, would appear to not parallelize, this is not a limit in practice. While a single transformation is global in nature (such as $\mathsf{F}_x[f(x,y,z)]$), that means that there area a large number of independent one dimensional transformations happening in parallel. As shown in figure 2.20, by splitting the array among computational nodes along a single dimension at a time, transformations along the other two dimensions require no communication. Transformation along the remaining dimension requires a transpose of the array to make each individual line held on a single processor.

This distribution of data is ideal for spectral methods such as this one, and it is the same approach as used in Winters et al. [2004]. Unlike finite difference and finite element codes,

which involve strictly boundary communication, the array transpose is a global communication operation; there is no benefit to having a fancier distribution of points with this parallelism.

The price paid for this parallelism of the spectral method is the global communication. For a three-dimensional $N \times N \times N$ grid, each of P processors will "own" N^3/P elements, which have to be communicated to other processors (and the same number received) every time there is a Fourier transform along the split direction. This gives a local communication cost[41] of $\mathrm{O}(N^3/P)$, compared to a local computation cost of $\mathrm{O}(N^3 \log(N)/P$. Computation is still asymptotically of a higher order than communication costs, but it is a very near thing – details of computer architectures can play a significant role in how well this algorithm scales in parallel.

This scaling is the objective for the preconditioner. To be effective:

- The preconditioner must give a good convergence rate for the GMRES algorithm. Resolution-independent convergence is ideal.
- The preconditioner must be applicable in $\mathrm{O}(N^2 \log(N))$ (in two dimensions) computational work or better.
- The preconditioner must not introduce new serial dependencies; parallel computational work must be $\mathrm{O}(N^2 \log(N)/P)$ or better on P processors to avoid swamping the cost of the Fourier transforms.

2.4.3 Multigrid

Multigrid methods have a long history of application to finite difference methods, and the underlying algorithm is quite simple [Briggs, 1987]. The basic idea is the same as that for preconditioning a spectral method. Iterative solutions have poor convergence because of scale difference between the low and high frequencies, so removing the low frequencies via another algorithm lets a cheap iterative method concentrate on just the high frequencies.

In this case, the "cheap iterative method" is a pointwise smoother such as Jacobi iteration, rather than something as complicated as GMRES, and the "other algorithm" for removing the low frequencies is the operator itself, just discretized more coarsely. The only additional components required are a coarsening (restriction) operator that transfers a residual from a finer to coarser grid and an interpolation (prolongation) operator that performs the inverse and transfers a lower-resolution solution from a coarser grid to a finer grid.

Trottenberg et al. [2001] is an excellent reference to the theory and implementation of multigrid, and it provides much better coverage than is in the scope of this work. This section will

[41] This assumes that each processor has an independent communication channel to each other processor. If communication goes through a bandwidth-limited hub, saturation of that hub can dominate the communication costs. In development, the communication pattern of this work triggered esoteric bugs on at least one cluster used for testing.

focus on the design choices relevant for the particular problem of the scalar Poisson equation (2.78), discretized with the standard 3-point Laplacian finite difference stencil and the tensor product extension to two dimensions. On a grid mapped with (2.76) as the operator, the full operator is treated as a variable-coefficient differential operator with the finite difference stencil used for the partial derivatives on the computationally-rectangular grid. The coefficients, derived from the Jacobian matrix (2.71) are evaluated with the proper spectral differentiation operators on the finest grid and are smoothed on the coarser grids using the same restriction algorithm used for the residuals.

The coarsening operator

In the geometric multigrid method used in this work, points on the coarser grid ($x_{j,k+1}$ in one dimension) are a subset of points on the finer grid ($x_{j,k}$). If the grid has an odd number of interior points, then exactly every other grid point is kept on the coarser level, and $x_{j,k+1} = x_{2j,k}$ with x_0 being a boundary point.

The coarsening operator for transferring a residual f_k to the coarser level $(k+1)$ is the full weighting operator [Trottenberg et al., 2001, sec. 2.3.3], given by:

$$f_{j,k+1} = \tfrac{1}{4} f_{2j-1,k} + \tfrac{1}{2} f_{2j,k} + \tfrac{1}{4} f_{2j+1,k}. \tag{2.79}$$

This transfer operator is very simple to implement, and it has a key advantage that it minimizes aliasing of high frequency components on the finer grid to low frequency components on the coarser grid. A sawtooth (Nyquist-frequency) wave on the finer grid has the pattern $[-1,+1,-1,\cdots]$, and such a wave is entirely zeroed out by applying (2.79). At the same time, low frequencies are preserved – a linear slope is also preserved exactly.

For a uniform grid, (2.79) preserves the first two moments of f. That is, for a grid of spacing h on the finer level (and $2h$ on the coarser level):

$$\sum_j \frac{f_{j,k}}{h} = \sum_j \frac{f_{j,k+1}}{2h} \tag{2.80a}$$

$$\sum_j \frac{j}{h} f_{j,k} = \sum_j \frac{j}{2h} f_{j,k+1}. \tag{2.80b}$$

This relationship holds exactly for the discrete sums, and if f is periodic or has even or odd symmetry, it also holds to $O(\exp(-kN))$ accuracy for the continuous-integral equivalent[42]. If the grid is *not* uniform, such as for the cosine-mapped grid, then (2.80) loses its connection to the continuously-defined moments, but it still holds true in the uniformly-weighted discrete sense.

[42] As discussed in Trefethen [2000, ch. 12], the trapezoidal rule is exponentially accurate for periodic functions.

Near the boundaries, (2.79) needs modification because the boundary points are not part of the PDE – they live as part of the boundary conditions. If f is discretized with a Fourier method, then the boundaries can simply be extended with appropriate symmetry and $f_{j,k}$ evaluated as if it was part of the interior. If f is discretized with the Chebyshev method on the cosine grid, however, the boundaries need special treatment. The boundary point $f_{0,k+1}$ can simply be handled by injection ($f_{0,k+1} = f_{0,k}$, and the same for the opposite boundary). The next point ($f_{1,k+1}$) is not next to the boundary on the finer grid, and (2.79) still applies.

For a grid with an even number of interior points (an odd number of intervals between grid points), it is impossible to both skip every other grid point and keep the boundaries as part of the coarser level. The latter is the more important property, since the boundary conditions determine the nature of the solution at all scales. In order to keep the coarser grid $x_{j,k+1}$ as a subset of the finer grid, (at least) one extra grid point must be kept and transferred from the finer to coarser level. This kind of decision is an actively researched topic (see Larsson et al. [2005] as an example), but for the purposes of using multigrid as a preconditioner, the simple strategy of "keep the grid point that corresponds to the largest gap" works well. If the underlying fine grid is uniform, keeping any arbitrary grid cell will do, and in implementation the decision is left to rounding error[43].

Dealing with the extra grid point in the coarsening operator requires conceptually splitting the coarsening into two steps: a low-pass filter and a subsampler. The low-pass filter is the full weighting operator (2.79), and there is no reason that it cannot be applied everywhere. The subsampler removes points from the fine grid to create the coarse grid, and only this needs to be altered to allow the extra point in the coarse grid. If the extra grid point is adjacent to the boundary, then the weighting operator does need to be modified; the simplest approach and the one used is to simply reduce the order, and set:

$$f_{1,k+1} = \tfrac{1}{2} f_{1,k} + \tfrac{1}{2} f_{2,k} \tag{2.81}$$

for a point next to the left boundary, with the mirrored extension used near the right boundary.

The interpolation operator

The interpolation operator is much simpler to consider than the coarsening operator. In the multigrid algorithm, only functions from the solution space (u in (2.78)) are transferred from coarser to finer levels. The boundary conditions of (2.78) is already satisfied, so there is no need for special consideration near the boundary.

In one dimension, the interpolation operator requires two rules:

[43] From the perspective of the finer level, over most of the domain every other point is translated to the coarse level, save for one location. The grid cells kept form the sequence $[x_{0,k}, x_{2,k}, \cdots, x_{2j,k}, x_{2j+1,k}, x_{2j+3,k}, \cdots]$.

- At points $x_{j,k}$ that are on the coarse grid $x_{j,k+1}$, the value is transferred directly. $u_{2j,k} = u_{j,k+1}$ for a grid with an even number of interior points, with a suitable offset for a grid with an odd number of interior points.
- At points that are not on the coarse grid, the value on the fine grid is the average of the neighbouring two points on the coarse grid:

$$u_{2j+1,k} = \frac{1}{2}u_{j,k+1} + \frac{1}{2}u_{j+1,k+1}. \tag{2.82}$$

The implementation of this operator is straightforward. Much like with the coarsening operator, the interpolation operator will leave a solution of the form ($u_j \propto \alpha + \beta j$) unchanged. For an equispaced grid, this corresponds to preserving linearity, but on an irregular grid such as the cosine-mapped grid this particular interpretation does not hold.

It may seem counterintuitive to not care about the cosine-mapped grid for the coarsening and interpolation operators, but in the development of this work numerical testing showed that more "proper" operators did not make much difference in convergence rates. Additionally, these operators remain constant even with mapped grids (or equivalently variable coefficients) from (2.76). Given that both the cosine-based grid and any valid Jacobian mapping are smooth, it seems irrational to include one effect in the grid transfer operators but not the other. A full treatment would use operator-dependent coarsening and interpolation [Trottenberg et al., 2001, sec. 7.7.3], but using the simpler implementation for this work is a permissible shortcut.

Error Smoothing

The remaining component of a multigrid method is a relaxation (smoothing) operator for the fine grid. Ideally, the low-frequency components of the error are eliminated on the coarse grid, so the smoothing operator can be tailored for rapid convergence of the high-frequency components. An ideal smoothing operator can be applied easily, removes high-frequency error at a resolution-independent rate, and (for this work) can be parallelized.

Such operators exist because of the assumption of band-limited error. Without removing the low-frequency error on the coarse grid, simple smoothing schemes run into the same problems that GMRES (see figure 2.18) did for the full spectral operator. With an ideally accurate[44] coarse grid solution, however, the iteration on the fine grid only has to deal with frequencies from $k_{max}/2$ to k_{max} – a much easier problem.

There are three main smoothing methods for the fine grid solution:

[44] Exact accuracy for the coarse grid solution is not necessary, nor is it even desired for use of a multigrid algorithm as an iterative method. Instead, based on an equal-errors argument, the coarse-grid solution only needs to be about as accurate (per iteration) as the residual error on the fine grid after smoothing.

- The Jacobi iteration is the simplest, and it forms a new correction $\tilde{u}_{n+1}$ by solving for each grid point as if all of its neighbours were zero. Rewriting the differential operator as $A = (D+L+U)$ where D comprises the diagonal terms (and L and U the lower and upper off-diagonal entries respectively), the Jacobi iteration is:

$$\tilde{u}_{n+1} = D^{-1} f_n, \tag{2.83a}$$

  or for the total iterate u_{n+1}:

$$u_{n+1} = u_n + D^{-1}(f - Au_n). \tag{2.83b}$$

  For the standard finite difference operator, this is not quite enough – (2.83) ends up doing a poor job at removing error near the Nyquist frequency. The solution is to replace it with a weighted-Jacobi smoother [Briggs, 1987, ch. 2], where (2.83b) becomes:

$$u_{n+1} = u_n + \omega D^{-1}(f - Au_n). \tag{2.83c}$$

  A weighting factor of $\omega = \frac{2}{3}$ [Briggs, 1987] gives good smoothing results for the three-point stencil for the second derivative on a uniform grid. The Jacobi iteration is completely parallelizable – D^{-1} can be applied independently for each grid point.

- The Gauss-Seidel iteration takes advantage of existing results to form a more accurate iteration. Using backwards substitution, a lower (upper) triangular matrix can be solved with computational work proportional to the number of nonzero entries. This modifies (2.83b) by making $(D+L)^{-1}$ the update operator:

$$u_{n+1} = u_n + (D+L)^{-1}(f - Au_n). \tag{2.84}$$

  Adding a weight is possible, much like (2.83c), but it does not significantly improve the smoothing properties [Trottenberg et al., 2001, sec. 4.3]. However, Gauss-Seidel is a more powerful smoother than the Jacobi iteration [Trottenberg et al., 2001, sec. 2.1.3].

  Unlike the Jacobi iteration, the Gauss-Seidel iteration is not parallelizable[45], since the update of point k requires that the point $k-1$ already be computed.

- Combining the smoothing properties of Gauss-Seidel iteration with the parallelization of Jacobi iteration requires an unusual grid ordering. The problem with standard (lexical) Gauss-Seidel iteration is that the dependencies are inherently sequential – the operator splits such that dependencies on the previous point is maintained, while dependencies on

[45] The Gauss-Seidel iteration is slightly parallelizable in two dimensions, in that for an $N \times N$ grid up to N points can be computed in parallel with a standard grid ordering. However, this result does not apply in one dimension, and this work uses a different approach for two-dimension-al multigrid.

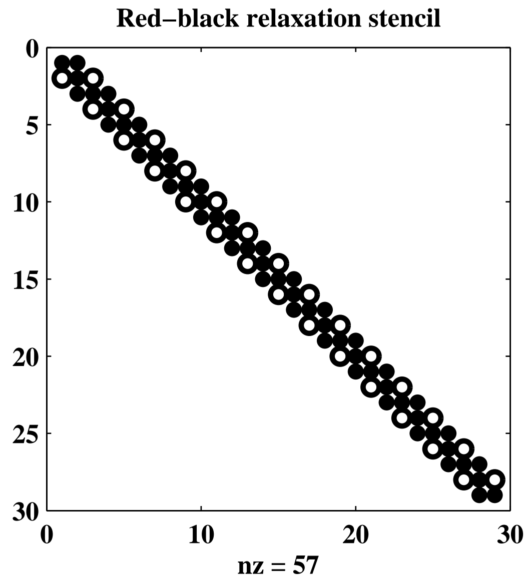

Figure 2.21: Nonzero entries of the Red-Black Gauss-Seidel update matrix (2.85) for $N = 29$ interior points and the three-point finite-difference second derivative operator. Entries included from the full operator are filled circles; neglected entries are hollow circles. Even grid points (rows) have only the diagonal entries and can be computed independently in parallel, while odd points can then be computed based on the neighbouring even values.

the next point is neglected. This sort of dependency splitting is not unique, and one alternative, symmetric way to split the operator is to allow odd points to depend on even points, but neglect the opposite dependencies. This approach gives the Red-Black Gauss-Seidel smoother [Briggs, 1987], so-called because applied in two dimensions the two types of grid points stagger the grid like a checkerboard.

Updating (2.84) for this method requires a different notation, since the lower and upper triangular parts of A are no longer are individually significant. Instead, consider $A = D + O$, where D is once again the main diagonal of A, and this time O contains all of the off-diagonal entries of A. Following the description of the dependencies, the only entries of O that this iteration considers are those on odd rows (corresponding to odd-numbered points) and even columns (corresponding to dependencies on even-numbered points). For the three-point second derivative stencil, this simplifies further since there there are no mutual dependencies between odd-numbered points[46], so it is possible to write O_{even} for the even rows of O and correspondingly O_{odd} for the odd rows. The update operator then becomes

[46]For wider stencils where the checkerboard decomposition doesn't fully decouple dependencies, it is possible to extend the approach with more than two colours. See section 5.4.2 of Trottenberg et al. [2001] for a discussion.

$(D+O_{odd})^{-1}$ (see figure 2.21 for its structure), and (2.84) becomes:

$$u_{n+1} = u_n + (D+O_{odd})^{-1}(f - Au_n). \tag{2.85}$$

The parallel properties of (2.85) are nearly as good as the Jacobi iteration. For a single pass, all even-numbered points can be computed simultaneously (and in fact are equal to those from the Jacobi iteration), and then all odd-numbered points can be computed simultaneously based on the even-numbered points. In addition, the Red-Black Gauss-Seidel iteration (without a weighting factor) is a better smoother than either the lexical Gauss-Seidel or the (weighted) Jacobi iterations [Trottenberg et al., 2001, sec. 2.1.2-3]. Because of the effectiveness, parallel performance, and simple implementation, the Red-Black Gauss Seidel iteration is the error smoothing method used in this work.

Anisotropy

Up to this point, the multigrid algorithm has been treated exclusively from a one-dimensional viewpoint. While the algorithm itself is interesting, one-dimension multigrid is also useless for this kind of finite difference problem: a tridiagonal matrix can already be directly factored in $O(N)$ operations, so a multigrid procedure doesn't accelerate anything. Multigrid only becomes necessary in two dimensions, when $O(N \times N)$ direct solutions do not exist.

The algorithms described above for coarsening, interpolation, and error smoothing all have direct two dimensional analogues. Coarsening and interpolation extend via the tensor product, and Red-Black Gauss-Seidel iteration works perfectly in two dimensions using the five-point Laplacian stencil. Unfortunately for the purposes of this work, the underlying grid does not permit such a simple implementation.

The root problem preventing application of simple two-dimensional multigrid algorithms is the anisotropy introduced by the cosine-mapped grid. Anisotropy is a general problem for multigrid methods (see section 5.1 of Trottenberg et al. [2001] for a full, enlightening discussion), whether it is introduced in the form of unequal coefficients:

$$u_{xx} + \varepsilon u_{yy} = f \tag{2.86}$$

or a non-square grid, which has the same effect under a coordinate transformation. The anisotropy couples the solution more strongly along one dimension, and the standard pointwise two-dimensional algorithms rely on the implicit assumption that errors are correlated roughly equally in both dimensions. As $\varepsilon \to 0$ in (2.86), the problem looks more and more like entirely decoupled one-dimensional problems (in x). Coarsening the grid in two dimensions improperly mixes the residual, rendering the resulting coarse-grid approximation meaningless; the effect is very slow convergence of the full multigrid iteration.

Two modifications of the multigrid method restore proper convergence:

- Semicoarsening modifies the coarsening and interpolation operators. Instead of coarsening in both dimensions, a semicoarsened grid applies the 1D coarsening and interpolation operators (2.79) and (2.82) in the x direction only for (2.86). On the coarse grids, the effective contributions between the x and y directions balance more closely, restoring proper convergence [Trottenberg et al., 2001, pp. 134]. The downside of this approach is that the coarse grid has only a factor of 2 fewer points, rather than a factor of 4 for the direct two-dimensional multigrid algorithm.

- Line smoothing techniques allow for coarsening in both directions simultaneously, but they modify the error smoothing. Instead of applying an iteration like the Red-Black Gauss-Seidel on a pointwise basis, these techniques consider an entire line at a time, in the direction of the strong coupling[47]. By removing all error from the strongly-coupled direction at once, the remaining error must come from the direction of weak coupling; line smoothing is an exact solver for (2.86) in the limit of 0 ε.

  Development of the line smoothing is essentially identical to (2.85), save that the notion of an odd-numbered or even-numbered point now relates to lines, rather than points. For (2.86), connections within a single x-line are included, as well as the dependence of odd-numbered x-lines on the even-numbered x-lines. The resulting smoothing (for a $N \times N$ grid) solves N tridiagonal systems, for a combined work cost of $\mathrm{O}(N \times N)$ – the same as for pointwise smoothing.

  The one disadvantage to a line smoothing technique is that the smoother is less parallel. Unlike pointwise smoothing, only adjacent lines are independent in this technique, so an $N \times N$ grid only permits N operations (line smoothings) in parallel.

The particular case of a tensor-product, cosine-mapped grid (figure 2.16) introduces another wrinkle – the anisotropy of the grid does not have a single direction. Instead, unknowns are coupled more strongly near edges, and less strongly in the middle of the domain. This prevents the straightforward application of either technique for anisotropy. Applying either approach twice (giving alternating-line smoothing or multiple semicoarsening) gives good convergence, at the expense of a simple implementation.[48]

This work combines the two approaches, for a hybrid semicoarsening and line-smoothing scheme. Smoothing proceeds along entire lines in the z-direction, and coarsening occurs only in the x-direction; this process is illustrated in figure 2.22. The only disadvantage to this method is conceptual, in that it does not treat the x and z dimensions in identical ways. The choice in this work of smoothing along z-lines and coarsening along the x-dimension is motivated by the

[47] The resulting smoothing pattern looks less like a checkerboard and more like a pattern of stripes, leading to the name "zebra line Gauss-Seidel smoothing" [Trottenberg et al., 2001, pp. 136].

[48] In fact, proper coarsening for this kind of problem is an area of active research. Adaptive semicoarsening methods [Larsson et al., 2005] or other methods that break the tensor-product geometry [bin Zubair et al., 2010] can give good results with simple smoothers, albeit again with fairly complicated implementations

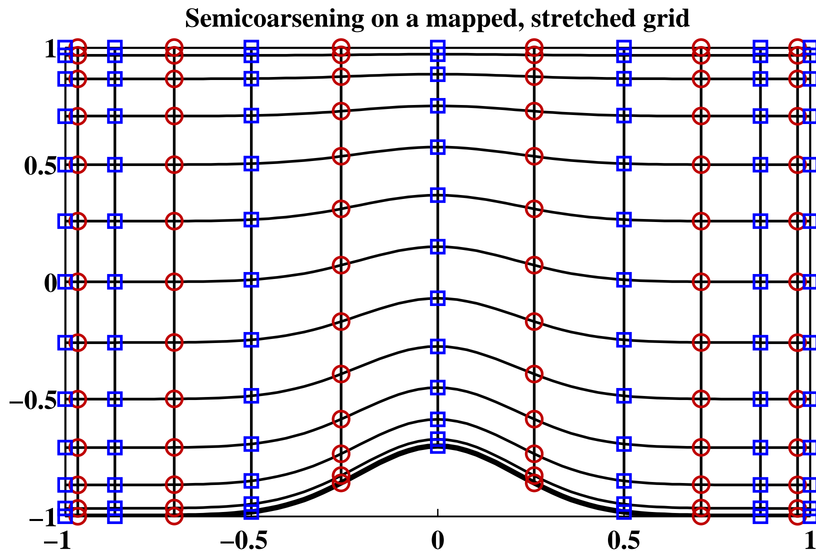

Figure 2.22: The semicorsening/line-smoothing problem applied to a 13×13 cosine grid, mapped to include a small hill (thick line) at the bottom of the domain. At the smoothing steps, each vertical line is smoothed in its entirety (line smoothing), and the points marked with squares are kept on the coarser level (semicoarsening). The coarsening and interpolation operators are those of (2.79) and (2.82), applied along horizontal grid lines.

parallel splitting of figure 2.20 – by default a given processor will "own" a number of contiguous z-lines, so it makes sense to use that direction for smoothing.

Coarse-grid solution

At some point in the multigrid algorithm, it becomes pointless to coarsen the grid further. The semicoarsening algorithm of figure 2.22 deals with so few points (z-lines) that the cost of a direct solution is small. Additionally, the parallelization of figure 2.20 is impossible to maintain when the number of z-lines is below the number of processors[49].

Calling the coarsest problem "small" may seem counterintuitive. After all, since coarsening occurs only in the x-direction, the coarsest grid will still have $O(N)$ points (for an $N \times N$ original grid), compared to a truly constant value for a fully-coarsening scheme. However, the important

[49] In fact, with the red-black line smoothing the algorithm needs at least $2P$ points in the x direction for P processors. Below this number, some fraction of processors must remain idle, and the problem is collapsed onto a subset of the processors. Using this idle CPU time for accelerated convergence is an area of active research; see Douglas [1996].

factor is that the coarse-grid solution must be computable in $O(N)$ time (the same order as the error-smoothing step), and in fact this is possible.

Consider the coarsest-possible grid as an example: $3 \times N$ points, where the leftmost and rightmost lines are the physical domain boundaries, and number the points lexically, increasing in the x-direction. In this case, the bottom-middle is point 2, its upward neighbour is point 5, and its right neighbour is point 3. In the worst-case, the finite difference stencil for (2.76) is a nine-point stencil (with the corner terms contributed by the ∂_{xz} derivative, so the derivative of point 5 (one up from the bottom-middle) depends on the points 1–9. Likewise, point 8 depends on 4–12, and so on going upwards from the bottom. Expressed as a matrix, the operator is tightly banded, with a bandwidth of 4 entries above and below the main diagonal. This matrix can be factored with $O(20N)$ multiplications and $O(16N)$ additions, and the factored matrix can be backsolved with $O(9N)$ multiplications and $O(8N)$ additions [Boyd, 2001, sec. B.2]. The small bandwidth of the matrix makes the solution just about as efficient as a smoothing pass on the grid, and that is more than enough to ensure a rapid solution of the coarse-grid problem.

In practice, the coarse grid can be significantly wider than $3 \times N$. The bandwidth of the resulting matrix increases proportionally, but it retains its sparse structure. Additionally, the banded solver is an excellent theoretical tool, but general sparse solvers such as `UMFPACK` (Davis [2004a] and Davis [2004b]) can take advantage of the additional sparsity and perform as well. This approach does not require the particular ordering of the banded matrix, either; this is an advantage since the other lexical ordering (z-lines numbered sequentially) is most useful for transferring data between processors.

The only mathematical caveat to the coarse-grid solver is that for the pressure problem, the Poisson problem (2.78) may not have a solution. This is in the strict, mathematical sense, and it reflects the compatibility condition for the Poisson equation. In brief, for the problem:

$$\begin{aligned} \nabla^2 u &= f \text{ in } D, \text{ with} \\ \hat{\mathbf{n}} \cdot \nabla u &= g \text{ on } \delta D \end{aligned} \tag{2.87}$$

the solution is determined only up to a constant factor, and the right-hand side f must satisfy a compatibility condition (from the Divergence theorem):

$$\int_D f \mathrm{d}V = \int_{\delta D} g \mathrm{d}C. \tag{2.88}$$

In the discrete case, the null space of (2.87) has one component – the constant function $u = C$, which is the same as the continuous case. Likewise, the discrete right-hand-side must satisfy a compatibility condition equivalent to (2.88).

Modifying the right-hand side of (2.87) to ensure that the compatibility condition is satisfied is problematic. From a matrix point of view, the discrete operator has exactly one more row and column than is necessary. Removing the entires corresponding to one grid point will normalize u

(by setting $u(i,j) = 0$ at the removed point), but it will also fail to properly satisfy the discretized Poisson's equation at that point. Since this problem arises for the pressure projection step (2.16b), the error will show up as artificial compressibility at a single grid point.

Instead, this work follows Trottenberg et al. [2001, sec. 5.6.4] and extends the system by introducing a new variable (σ), that implicitly modifies the right-hand side to ensure a solution. The mean condition ($\langle u \rangle = 0$) is then explicitly included in the system to keep the number of (discretized) equations the same as the number of degrees of freedom. The resulting system is:

$$\nabla^2 u = f - \sigma \text{ in } D, \tag{2.89a}$$

$$\hat{\mathbf{n}} \cdot \nabla u = g - \sigma \text{ on } \delta D \text{, and} \tag{2.89b}$$

$$\langle u \rangle = 0. \tag{2.89c}$$

The net effect of σ is to "spread out" the effect introduced by a right-hand side that does not satisfy the compatibility condition. In practice, the pressure projection equation (2.16b) comes from adjusting the advection-modified velocities to ensure incompressibility, and σ remains small – usually on the same order as the discretization error. Even with a very well-resolved simulation with minute discretization error, σ will not be exactly zero due to rounding error. The formulation of this problem in equations 2.89 also has key advantage that no grid point is uniquely treated to satisfy Neumann-type boundary conditions.

It is also worthwhile noting that while the compatibility condition is well-defined in the continuous sense, its exact effect on the discretized system depends on the choice of discretization. Modifying the right-hand side to satisfy the compatibility condition with a Chebyshev discretization is not helpful in making the finite-difference preconditioner solvable. The extension of (2.87) to (2.89) makes the system solvable regardless of discretization.

From a matrix perspective, the discrete (finite-difference) operator corresponding to (2.89) is no longer bounded, but it still has a useful structure. The two new elements each contribute a full row and column, making the matrix a "bordered" matrix, with the same banded structure in the interior as (2.78). Using block-matrix methods and considering the new row and column as $N \times 1$ and $1 \times N$ rectangular matrices, a matrix of this form can be factored almost as efficiently as a regular banded matrix [Boyd, 2001, sec. B.5]. In implementation, this detail is left to a generalized sparse-matrix solver (`UMFPACK` for this work) – a testament to the convenience of such solvers.

In a multigrid context, σ applies everywhere, at all levels, but it only needs to be computed on the coarsest grid, at the same time as the direct solve of the coarsest matrix [Trottenberg et al., 2001]. The smoothing operator is well-defined even without the addition of σ. It may not converge as an actual *solver* for the full finite difference problem, but it still effectively removes high-frequency components from the error. Before the coarse-grid solve (when σ is entirely unknown), the red-black line smoothing can ignore σ entirely. After the coarse-grid solve, the line smoothing can properly apply σ.

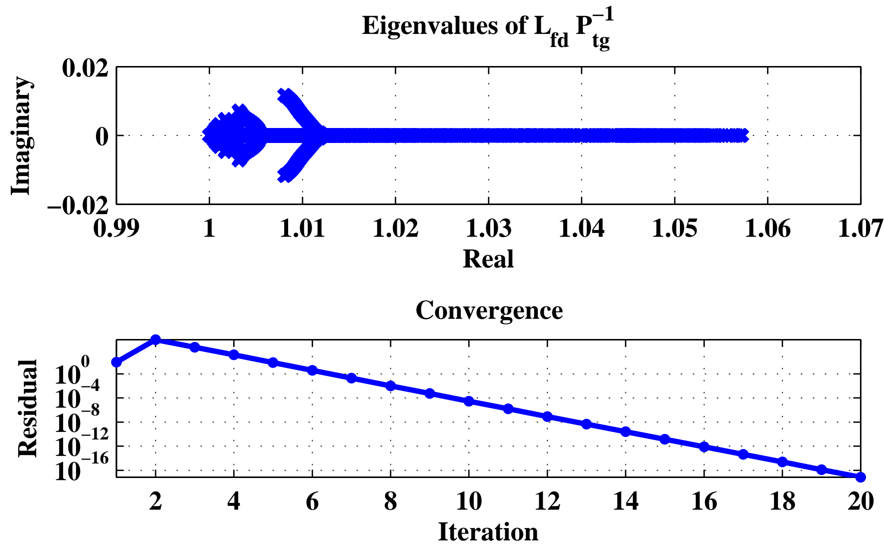

Figure 2.23: Eigenvalues (top) of $L_{fd}P^{-1}$, where L is the 63×63 finite-difference discretization of $\nabla^2 u = f$ with Dirichlet boundary conditions included in the operator and P^{-1} is the two-grid iteration (2.90) with one pass of red-black line smoothing before and after the coarse-grid solve, along with the convergence history (residual error, bottom) of the direct iteration $u_n = u_{n-1} + P^{-1}(f - L_{fd}u_{n-1})$. with a random initial f.

The complexity of extending the system by (2.89) is neatly avoided for the spectral definition. Since σ is part of the solution (u), including it in the computation of the residual for (2.89a) is trivial. Likewise, computing the pointwise mean for (2.89c) is equally simple.

Preconditioner convergence

The discussion to this point has focused a great deal on what multigrid approaches do not work for this problem, characterized by anisotropy near each edge of the domain. The combination of semicoarsening and line smoothing described above should be a rigorous application to the problem, but these results were presented without proof. Unfortunately, the most commonly-applied tool for the analysis of multigrid methods, Local Fourier Analysis [Trottenberg et al., 2001, ch. 4] does not directly apply to this problem. Local Fourier Analysis depends on freezing the coefficients of the differential equation and analyzing local smoothing factors, however the line-smoothing in this approach is inherently nonlocal. Rigorous Fourier analysis [Trottenberg et al., 2001, sec. 3.3] considers the entire grid, but the nonuniform grid does not give simple eigenfunctions that are preserved under smoothing.

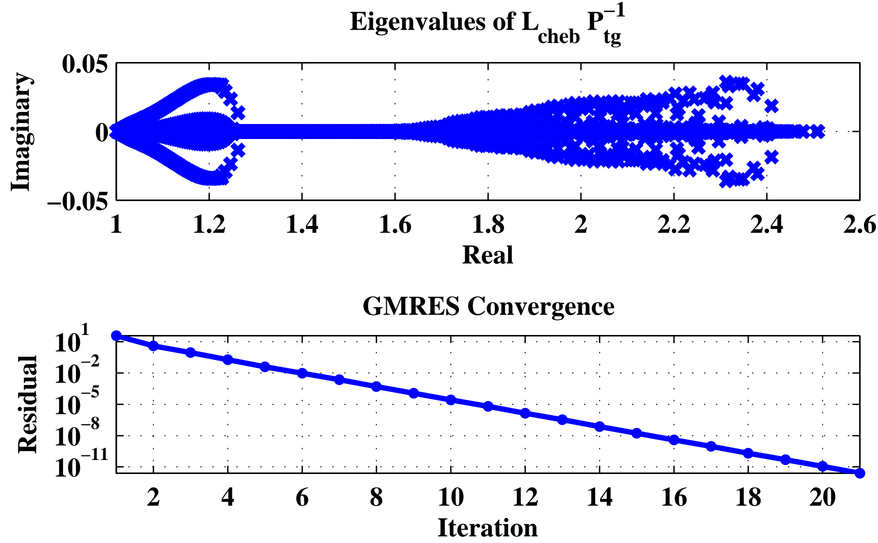

Figure 2.24: Eigenvalues (top) of $L_{cheb}P^{-1}$, where L is the Chebyshev-based discretization of the operator in figure 2.23 and P^{-1} is the same two-grid operator, along with the GMRES convergence history (bottom) of the two-grid operator used as a preconditioner.

Instead, a practical approach is to look at the two-grid operation. In the two-grid approach, the multigrid iteration is truncated at one level. That is, for an initial right-hand side f:

- Apply (one or more) smoothing steps on the full problem.
- Coarsen the remaining residual by one level, and directly solve the problem on the coarse grid.
- Interpolate the coarse-grid solution, and apply another one or more smoothing passes to the remaining residual.

Analytically, taking L to be the differential operator (with boundary conditions) on the finest level, L_{RB}^{-1} to be the red-black line smoothing on the finest level, C to be the (semi-) coarsening operator, P to be the interpolation (prolongation) operator, and L_{CG}^{-1} to be the solve on the coarse grid, the two-grid operator corresponding to one smoothing pass before and after the coarse-grid solve is given by:

$$L_{pre}^{-1} = L_{RB}^{-1} \tag{2.90a}$$

$$L_{coarse}^{-1} = L_{pre}^{-1} + PL_{CG}^{-1}C(1 - LL_{pre}^{-1}) \tag{2.90b}$$

$$L_{tg}^{-1} = L_{post}^{-1} = L_{coarse}^{-1} + L_{RB}^{-1}(1 - LL_{coarse}^{-1}). \tag{2.90c}$$

This splits the operator into several parts for notational and conceptual simplicity. (2.90a) is the application of just the initial smoothing, (2.90b) applies the initial smoothing and coarse-grid solve, and (2.90c) applies all three steps. This operator does not have to be constructed in practice, since each individual step is relatively simple.

Explicitly constructing (2.90) allows for detailed analysis of convergence, however. Applied to the basic finite difference problem, (2.90) is an extremely effective preconditioner. As shown in figure 2.23 for a 63×63 grid, the eigenvalues of the resulting preconditioned operator $L_{fd}L_{tg}^{-1}$ are tightly bound, between approximately 1 and 1.06 on the real axis and -0.02 and 0.02 on the imaginary axis. The resulting iterative algorithm converges extremely quickly, at an asymptotic rate of about 0.05.

The same approach can be used to look at applying (2.90) to the full, Chebyshev-discretized operator, with results shown in figure 2.24. The preconditioned operator $L_{cheb}L_{tg}^{-1}$ does not have as tight of a bound on the eigenvalues as figure 2.23, but the resulting eigenvalues are still in the range of $[1,2.5]+[-0.05,0.05]\mathbf{i}$. When used as a preconditioner for the full spectral problem with the GMRES algorithm, the two-grid operator still converges at a rate of about 0.3.

The story of figure 2.24 is remarkable, and extremely fortunate for this work. One application of the two-grid operator – an *approximate* solver for the *finite difference* problem – is an excellent preconditioner for the *spectral* operator. The convergence rate of figure 2.24 is not much worse than the convergence rate of figure 2.19, which uses the full finite-difference operator as the preconditioner.

Computational costs & parallel efficiency

In implementation, the "coarse-grid solve" of the two-grid iteration (2.90) is itself a finite-difference problem on the coarser grid; this lends itself to a solution (recursively) with the same algorithm. The manner of this recursion – the type of multigrid cycle – affects the convergence and performance properties of the full algorithm.

The most straightforward recursion replaces L_{CG}^{-1} in (2.90b) with a single application of the full iteration. The resulting algorithm (2) is the V-cycle, visualized in figure 2.25. The characteristic feature of the V-cycle is that each grid is visited only twice, once descending the hierarchy and once ascending it, and smoothing is applied both before and after the nested coarse-grid solve.

The choice to visit each level only once is not required, and making more than one visit to the coarse grid can often improve convergence factors [Trottenberg et al., 2001, sec. 2.5]. The most straightforward modification is the W-cycle, where during the iteration two recursive calls, rather than one, are made to the coarse grid. The resulting cycle is pictured in figure 2.26, and the downside of this approach is apparent. The total number of a visits to a level grows exponentially with its depth in the grid hierarchy.

Algorithm 2 The multigrid V-cycle

u = V_CYCLE(n_1,n_2,k,f)

Solve the problem $Au = f$ for a given discretized differential equation (A) on the current grid-level (k) using a multigrid V-cycle

if Grid k is the coarsest level **then**

  $u \leftarrow A^{-1}f$

  return u

else

  $def \leftarrow f$ {def is the defect – the remaining residual}

  $u \leftarrow 0$

  for $k = 1$ to n_1 **do** {Smoothing before the coarse-grid solve}

    $cor \leftarrow$ RB_line_smooth(cor) {cor is the correction from the local smoothing}

    $def \leftarrow def - A \cdot cor$

    $u \leftarrow u + cor$

  end for

  $def_{coarse} \leftarrow$ coarsen(def)

  $u_{coarse} \leftarrow$ V_CYCLE(n_1,n_2,$k+1$,def_{coarse})

  $cor \leftarrow$ interpolate(u_{coarse})

  $def \leftarrow def - A \cdot cor$

  $u \leftarrow u + cor$

  for $k = 1$ to n_2 **do** {Smoothing after the coarse-grid solve}

    $cor \leftarrow$ RB_line_smooth(cor)

    $def \leftarrow def - A \cdot cor$

    $u \leftarrow u + cor$

  end for

end if

return u

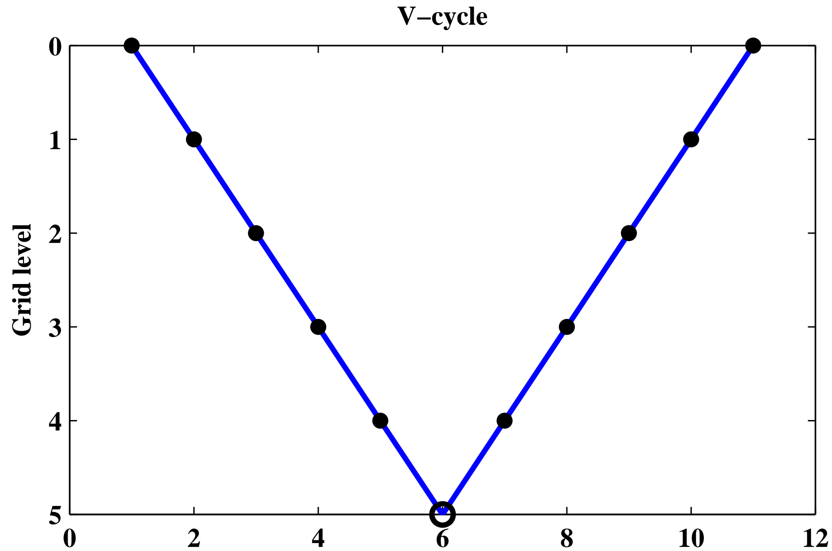

Figure 2.25: Illustration of the multigrid V-cycle, for $n = 5$ levels in the multigrid hierarchy. The cycle descends to the coarsest grid once, performs a direct solve (open circle), and ascends the hierarchy back to the finest level.

The impact of this decision on the computational performance (per cycle) is significant. For an $N \times N$ finest grid, the multigrid hierarchy contains $\lceil \log_2 N \rceil$ levels, and on each level the smoothing operator contributes work proportional to the number of points on that level ($\mathrm{O}(2^{-j}N^2)$ on the jth level). For the V-cycle, this gives a total workload of:

$$\mathrm{Work}_V = \sum_{j=0}^{\lceil \log_2 N \rceil} \mathrm{O}(2^{-j}N^2) = \mathrm{O}(N^2 + \tfrac{1}{2}N^2 + \cdots) = \mathrm{O}(N^2). \tag{2.91}$$

In contrast, the W-cycle visits the jth level $3 \cdot 2^j$ times, giving:

$$\mathrm{Work}_W = \sum_{j=0}^{\lceil \log_2 N \rceil} 3 \cdot 2^j \mathrm{O}(2^{-j}N^2) = \mathrm{O}(N^2 \log_2 N), \tag{2.92}$$

which is an asymptotically worse workload.

A compromise between the V-cycle and W-cycle is the F-cycle [Trottenberg et al., 2001, sec. 2.4]. Like the W-cycle, the F-cycle makes two calls to the coarse-grid solve, but only the first call is recursive. The second call is to a V-cycle. The relevant part of the algorithm is given in algorithm 3, and the schematic is illustrated in figure 2.27. In the F-cycle, the jth level is

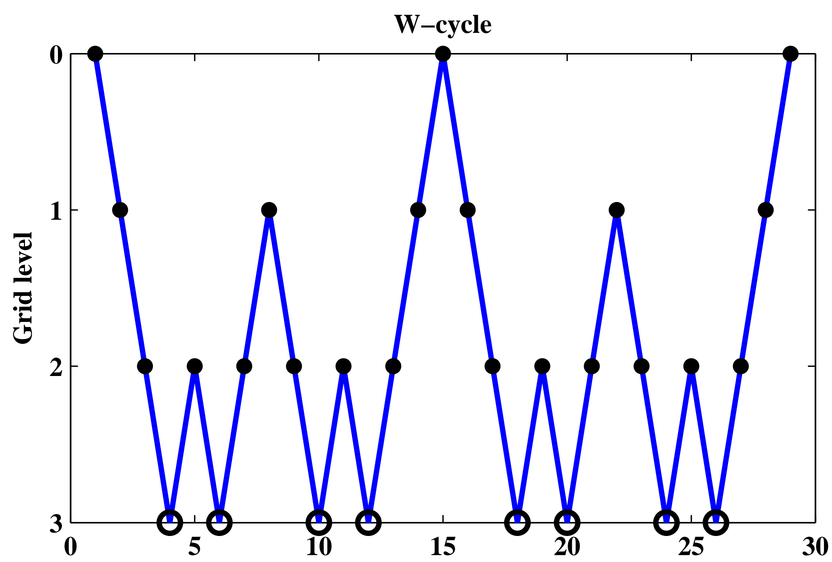

Figure 2.26: Illustration of the multigrid W-cycle, for $n = 3$ levels in the multigrid hierarchy. Unlike the V-cycle (figure 2.25), the W-cycle visits the coarser grids twice, recursively. As a result, there are many more direct solves on the coarsest grid (open circles).

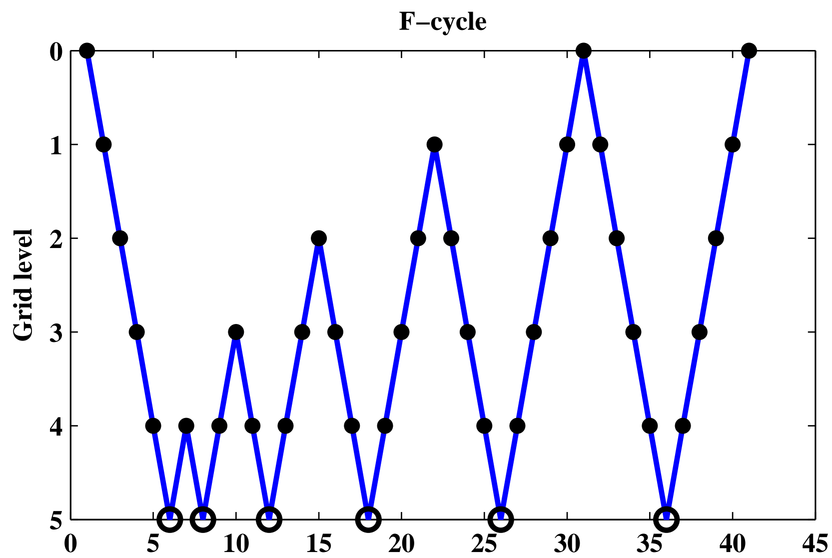

Figure 2.27: Illustration of the multigrid F-cycle, for $n = 5$ levels in the multigrid hierarchy. The F-cycle visits the coarser grids twice like the W-cycle (figure 2.26), but unlike the W-cycle only the first visit is recursive; the second traversal is a V-cycle (figure 2.25). The F-cycle still has several direct solves on the coarsest grid (open circles), but the number increases linearly with depth rather than exponentially.

Algorithm 3 The portion of the multigrid F-cycle that differs from the V-cycle (2).

u = F_CYCLE(n_1,n_2,n_3,k,f)

⋮

{Directly solve on the coarsest level, and perform n_1 pre-coarsening smoothings as in the V-cycle}

$def_{coarse} \leftarrow$ coarsen(def)

$u_{coarse} \leftarrow$ F_CYCLE(n_1,n_2,n_3,$k+1$,def_{coarse}) {The recursive call.}

$cor \leftarrow$ interpolate(u_{coarse})

$def \leftarrow def - A \cdot cor$

$u \leftarrow u + cor$

for $k = 1$ to n_2 **do** {Smoothing after the first coarse-grid solve}

  $cor \leftarrow$ RB_line_smooth(cor)

  $def \leftarrow def - A \cdot cor$

  $u \leftarrow u + cor$

end for

$def_{coarse} \leftarrow$ coarsen(def)

$u_{coarse} \leftarrow$ V_CYCLE(n_1,n_3,$k+1$,def_{coarse}) {The nonrecursive call to the V-cycle}

$cor \leftarrow$ interpolate(u_{coarse})

$def \leftarrow def - A \cdot cor$

$u \leftarrow u + cor$

⋮

{Perform n_3 post-coarsening smoothings, as in the V-cycle.}

return u

visited $2j+3$ times, which gives a total workload of:

$$\text{Work}_F = \sum_{j=0}^{\lceil \log_2 N \rceil} (2j+3)\mathrm{O}(2^{-j}N^2) = \mathrm{O}(3N^2 + \tfrac{5}{2}N^2 + \tfrac{7}{4}N^2 + \tfrac{9}{8}N^2 + \cdots) = \mathrm{O}(N^2). \quad (2.93)$$

The resulting algorithm has the same asymptotic workload as (2.91), although the constant of proportionality is greater.

On a parallel system, these performance arguments change. The general principle that applies is that no matter how many processors are available, it is inefficient for a processor to work on fewer than some number of points. For the multigrid algorithm of this section, the limit is given by the red-black line smoothing: each processor needs at least two complete lines ($2N$ points – one red line and one black line) to avoid stalls while waiting for neighbouring processors to finish. That implies that for P processors, a grid-level with fewer than $2P \times N$ points will require some processors to remain idle, assuming communication costs are negligible. On real machines, this may not be the case and the smallest efficient-in-parallel grid may be a wider $N_c \times N$ points. This complicates the performance analysis.

On an individual grid level of grid size $M \times N$, the work of the line smoothing parallelizes evenly over each processor, and the $\mathrm{O}(MN)$ work takes $\mathrm{O}(P^{-1}MN)$ time, provided $M > PN_c$. When that restriction does not hold, some processors remain idle; the working processors each have a grid of size $N_c \times N$, so the resulting work takes $\mathrm{O}(N_cN)$ time, regardless of the true grid size. The performance analysis leading to (2.91)-(2.93) splits into three cases:

- For levels with more than $N_cP \times N$ points, the problem fully parallelizes. Each processor has $P^{-1}M \times N$ points. For a finest-grid of $M \times N$ points, there are $\left\lfloor \log_2(\frac{M}{PN_c}) \right\rfloor$ levels where this holds.
- For levels with fewer than $N_cP \times N$ points, only $\frac{M}{N_c}$ processors work; the remainder are idle. Each processor has approximately $N_c \times N$ points, giving an $\mathrm{O}(N_cN)$ computational time per smoothing on that level. Successively coarser levels will idle half of the remaining processors, so this case holds for $\log_2 P$ levels.
- With fewer than $N_c \times N$ points in the entire grid, the problem fits only on a single processor. This is the "base-case" of parallelization, where the results for the single-processor multigrid iteration again apply because there is no parallelization. Assuming that N_c is not too large – a fair assumption for parallel systems with speedy communication – this is also approximately the stage at which the coarsest (direct) solve will take place, with an $\mathrm{O}(N_cN)$ workload once again.[50]

[50] If N_c is larger than when the direct solve is appropriate, the serial multigrid iteration can continue. V-cycles and F-cycles still have an $\mathrm{O}(N_cN)$ workload for the cycle (from (2.91) and (2.93)), but a W-cycle would have $\mathrm{O}(N_cN\log(N_c))$ work.

An effective performance analysis must take these three cases into account. The V-cycle is simplest to analyze: the first few levels (fully parallelized) contribute $\mathrm{O}(\frac{N^2}{P})$ work, the middle $\log_2(P)$ levels each contribute $\mathrm{O}(N_c N)$ work, and the coarsest level contributes $\mathrm{O}(N_c N)$ work as well. Each level save for the coarsest is visited two times, so the resulting workload in parallel is:

$$\mathrm{PWork}_V = \mathrm{O}(\tfrac{N^2}{P}) + \mathrm{O}((1+\log_2 P)N_c N). \tag{2.94}$$

Provided $N \gg PN_c \log_2(P)$, the V-cycle parallelizes almost ideally. The other cycle types have more complicated analyses, since coarser levels are visited more frequently. To simplify the notation, let $l_p = \log_2(\frac{N}{PN_c})$ be the number of levels that fully parallelize and $l_s = \log_2(P)$ be the number of levels that only partially parallelize, where each non-idle processor has $N_c \times N$ points locally. For the W-cycle, this gives a parallel workload of:

$$\begin{aligned}
\mathrm{PWork}_W &= \sum_{j=0}^{l_p-1} 3\cdot 2^j \mathrm{O}(2^{-j}\tfrac{N^2}{P}) + \sum_{j=l_p}^{l_p+l_s} 3\cdot 2^j \mathrm{O}(N_c N) \\
&= \mathrm{O}(l_p \tfrac{N^2}{P}) + 3\cdot 2^{l_p}(2^{l_s+1}-1)\mathrm{O}(N_c N) \\
&= \mathrm{O}((\log_2(N) - \log_2(N_c P))\tfrac{N^2}{P}) + \mathrm{O}(\tfrac{N}{PN_c} PN_c N) \\
&= \mathrm{O}((\log_2(N) - \log_2(N_c P))\tfrac{N^2}{P}) + \mathrm{O}(N^2).
\end{aligned} \tag{2.95}$$

The W-cycle almost does not parallelize at all. (2.95) is hardly better than the $\mathrm{O}(N^2\log(N))$ of (2.92), and it comes at the cost of interprocessor communication and a significantly more complicated implementation.

The F-cycle does not require so many visits to the coarse grid, growing linearly rather than exponentially. Its workload is therefore:

$$\begin{aligned}
\mathrm{PWork}_F &= \sum_{j=0}^{l_p-1} (2j+3)\mathrm{O}(2^{-j}\tfrac{N^2}{P}) + \sum_{j=l_p}^{l_p+l_s} (2j+3)\mathrm{O}(N_c N) \\
&= \mathrm{O}(\tfrac{N^2}{P}) + (4l_s + l_s^2 + l_p l_s)\mathrm{O}(N_c N) \\
&= \mathrm{O}(\tfrac{N^2}{P}) + \mathrm{O}((\log(P)^2 + \log(P)(4+\log(\tfrac{N}{N_c P})))N_c N).
\end{aligned} \tag{2.96}$$

This expression is more complicated than (2.94), but it does not represent disastrously more work. In the limit $N \gg PN_c \log(P)$, (2.96) and (2.94) are the same order.

As a consequence, the F-cycle is the default multigrid cycle type used in this work. It has much the same performance characteristics as the V-cycle, but trial-and-error during development showed that the greater attention to coarser levels gave a more robust preconditioner.

Chapter 3

Code documentation and case studies

Implementing the algorithms of Chapter 2 in C++ to create a useful fluid dynamics model has its own set of design considerations. The algorithms apply to idealized arrays, without regard to implementation details like memory allocation. As always, the devil is in the details, and this chapter documents the implementation choices and what is necessary to use this code to perform useful, physical calculations.

At a basic level, any useful application of the pseudospectral algorithms does much the same thing. At each timestep, each step of (2.56) executes, regardless of whether the underlying goal is to model microscopic diffusive motion or large, three-dimensional geophysical motion. Between different physical applications, the only significant differences are in initialization, forcing, and analysis. This project's underlying organization is that scientifically useful cases should require as little boilerplate code as possible.

This chapter will cover the use of this fluid dynamics model for scientifically useful applications, beginning with a trivial example (section 3.1) that does no work at all. Along the way, appropriate test cases will be introduced to demonstrate the code's accuracy.

3.1 A trivial example

The simplest possible code does nothing – not in the sense of literally exiting as soon as it is run, but in the sense of going through proper setup and shutdown, without performing useful computation. This "null code" illustrates the basic, skeletal features required of more advanced codes, without complicating the matter with pesky science.

The best place to start such an illustration is with `main()` – a short stub of a function in this program.

Listing 3.1: The minimal main()

```
   /* The ''main'' routine */
55 int main(int argc, char ** argv) {
     /* Initialize MPI. This is required even for single−processor runs,
        since the inner routines assume some degree of parallelization,
        even if it is trivial. */
     MPI_Init(&argc, &argv);
60   minimal mycode; // Create an instantiated object of the above class
     /// Create a flow−evolver that takes its settings from the above class
     FluidEvolve<minimal> do_nothing(&mycode);
     // Initialize the flow
     do_nothing.initialize();
65   // Run to a final time of 1.
     do_nothing.do_run(1);
     MPI_Finalize(); // Cleanly exit MPI
     return 0; // End the program
   }
```

The distinguishing feature of listing[1] 3.1 is that nearly all of the case-dependent code is encapsulated in a user-supplied class, `minimal` in this example. The underlying Navier-Stokes integrator is entirely contained in `FluidEvolve`, which is a template class so that it can call the user-supplied code directly, without indirection through virtual functions. The MPI initialization and teardown on lines 59 and 67 are unfortunately necessary, even for single-processor runs. In order to present as unified an interface as possible, the Navier-Stokes integrator and support routines all assume that the program is being run in a parallel environment, and they internally make calls to the MPI message-passing library [Message Passing Interface Forum, 2008]. A standards-compliant MPI application requires the initialization (`MPI_Init`) and teardown (`MPI_Finalize`) even when run with only one processor.

Line 64 of 3.1 runs the initialization code. Internally, this creates most of the permanent arrays used for the time stepping, and then the initialization methods of the user-supplied class are called. The relevant section of this minimal case is:

Listing 3.2: Minimal initialization

```
   class minimal : public BaseCase {
     public:
       /* Set up a 100 x 1 x 100 grid */
       int size_x() const { return 100; }
20     int size_y() const { return 1; }
       int size_z() const { return 100; }
```

[1] This listing is included in its entirety as A.1.

```
    /* Set all boundaries to be periodic */
    DIMTYPE type_default() const { return PERIODIC; }
25
    /* The grid corresponds to a 1 (x 1) x 1 physical space */
    double length_x() const { return 1; }
    double length_y() const { return 1; }
    double length_z() const { return 1; }
30
    /* Use no tracer variables */
    int numtracers() const { return 0; }

    /* Start at t=0 */
35  double init_time() const { return 0; }

    /* Initialize velocities at the start of the run. For this simple
       case, initialize all velocities to 0 */
    void init_vels(DTArray & u, DTArray & v, DTArray & w) {
40    u = 0; // Use the Blitz++ syntax for simple initialization
      v = 0; // of an entire (2D or 3D) array with a single line
      w = 0; // of code.
      return;
    }
```

There are two main parts to this initialization. Lines 19–35 set properties: in order they are the array size (a $100 \times 1 \times 100$ grid), boundary type (periodic in all three dimensions), physical lengths ($1 \times 1 \times 1$), number of tracers (0), and initial time (0). Lines 39–44 set up the starting velocities – all zero in this example. They also provide a first look at the Blitz++ array library, whose use is integral to this work.

3.1.1 A brief interlude on Blitz++

This code makes substantial use of the Blitz++ array library (briefly described in Veldhuizen [1998], with full documentation in Veldhuizen [2006]); it is the one non-standard library that is heavily used even by the case-dependent code. The library provides the framework for storing, accessing, and computing expressions on three-dimensional arrays. The Blitz++ library uses C++ templates to support arbitrary contained datatypes, and this code uses double-precision floating point for the vast majority of its arrays, including all of the ones used to interact with case-dependent code[2].

[2]The C++ type `complex<double>` is used internally with periodic Fourier transforms, but all quantities in physical space are real-valued.

An array in Blitz++ is initialized as a standard C++ object, with two template parameters in the type that specify the contained datatype and number of dimensions; constructor parameters determine the array size at runtime. For example:

```
Array<double,2> my_2d_array(10,10);
```

creates a 2-dimensional array (my_2d_array) of size 10×10. By convention the indices in each dimension run from 0 to 9, and the array is created with the standard row-major storage order (with my_2d_array(0,0) being followed in memory by my_2d_array(0,1)). These choices are not fixed, and they can vary at runtime based on optional parameters passed to the constructor[3].

An element of a Blitz-array can be accessed using mathematically standard notation, where the $(i, j)th$ element of the array is my_2d_array(i,j). This differs from the C-syntax of c_array[i][j], in part because the Blitz-array allows more flexible stride lengths and memory orderings. In addition to elementwise access, the Blitz-array allows assignment of the entire array at once. For example,

```
my_2d_array = 1;
```

assigns the value 1 to each element of my_2d_array. More usefully, the library also allows array expressions with a syntax similar to equivalent expressions in MATLAB:

```
// Assuming A and B are already−defined 10 x 10 arrays:
Array<double,2> C(10,10);
C = A+(B*sin(A)); // Perform elementwise operation
```

This works provided all the involved arrays have the same logical shape[4].

Finally, Blitz++ allows multidimensional arrays to be initialized through expressions with *index placeholder* variables. These function like tensor indices without automatic summation, and allow multidimensional arrays to be initialized from lower-dimensional components. For example:

```
double Lx=1.0, Ly=2.0; // Arbitrary lengths in X and Y
Array<double,1> xx(10), yy(20); // Create one−dimensional X and Y arrays
// Create index placeholder variables for the first and second dimensions
firstIndex ii; secondIndex jj;
xx = ii*Lx/10; // Initialize xx between 0 and Lx
yy = ii*Ly/20; // Same for yy, between 0 and Ly
Array<double,2> A(10,20); // Two−dimensional array
// And initialize A to a combination of X and Y, without
// two−dimensional temporary arrays.
A = sin(M_PI*xx(ii)/Lx)*cos(M_PI*yy(jj)/Ly);
```

[3]This code adjusts the indices of arrays to support parallel operation – this will be discussed in more detail in section 3.3.

[4]Identical memory storage order is -not- required, but will likely have a performance penalty.

The most important distinction in the above listing is that the index placeholders correspond to dimensions on the *left* side of the assignment. That is why the expressions in lines 5 and 6 to initialize the coordinate system both use `ii`[5], while the expression in line 10 uses `ii` for the x-coordinate and `jj` for the y-coordinate.

3.2 Kelvin-Helmholtz billows

Using this code to compute physical scenarios is not much more complicated. As an example, consider the growth of Kelvin-Helmholtz billows in a shear-unstable stratified layer. The basic setup is a simple, two-layer fluid with a smooth pycnocline at the center of the domain, with a coincident horizontal shear. According to linear stability theory [Kundu and Cohen, 2004], the stability to infinitesimal perturbations is governed by the Richardson number, defined as:

$$\mathrm{Ri} = \frac{N^2}{\left(\frac{\partial u}{\partial z}\right)^2} = \frac{g}{\rho_0}\frac{\frac{\partial \bar{\rho}}{\partial z}}{\left(\frac{\partial u}{\partial z}\right)^2}, \tag{3.1}$$

where $\bar{\rho}(z)$ is the background density profile, ρ_0 is the reference density, and N^2 is the square of the buoyancy frequency. The profile may have instabilities when Ri drops below 0.25 in the pycnocline. When the profile is unstable, perturbations grow in size and form characteristic Kelvin-Helmholtz billows, eventually resulting in secondary instabilities (see Smyth et al. [2005]).

To demonstrate the growth of Kelvin-Helmholtz billows with this code, consider the background density and velocity profiles:

$$\bar{\rho}(z) = \frac{0.01}{2}\tanh\left(\frac{z-0.5}{0.1}\right) \tag{3.2a}$$

$$u(z) = 0.1808\tanh\left(\frac{z-0.5}{0.1}\right), \tag{3.2b}$$

in a vertical domain of length 1 ($0 \le z \le 1$), with reference density ρ_0 also equal to 1. The profiles (3.2) give a minimum Richardson number of 0.15, well inside the unstable range.

The exact form of the instability can be computed through the Taylor-Goldstein equation [Kundu and Cohen, 2004, equation 12.58]:

$$(U-c)\left(\psi_{zz} - k^2\psi\right) - U_{zz}\psi + \frac{N^2}{U-c}\psi = 0, \tag{3.3}$$

[5] As a personal convention, this work uses `ii`, `jj`, and `kk` for indices (and placeholders) in the first, second, and third dimensions respectively. The doubled letter improves readability in large blocks of code.

where $\psi(z)$ is the eigenfunction of the instability, U is the background velocity profile, N^2 is again the background buoyancy frequency, k is the spatial wavelength of the instability, and c is its phase speed. Fixing a value of k and multiplying (3.3) through by $(U-c)$ gives a quadratic eigenvalue problem for c (the eigenvalue) and ψ, which is solvable numerically.

The eigenfunction is a function of z alone. The full streamfunction is given by $\psi(z)\exp(\mathbf{i}k(x-ct))$, with velocities given by $u=\psi_z$ and $w=-\psi_x$. The density perturbation according to linear theory is given by the slightly more complicated relationship [Kundu and Cohen, 2004, equation 12.57]:

$$\rho' = \frac{\rho_0 N^2}{g(U-c)}\psi(x,z,t). \tag{3.4}$$

Eigenvalues with a nonzero imaginary part correspond to growing ($c_i > 0$) or decaying ($c_i < 0$) modes. By performing a search on k, the most-unstable mode can be found by repeated solution of (3.3). For this demonstration, this was computed in MATLAB using the `polyeig` routine with a Chebyshev discretization corresponding to (2.67). This gave a most-unstable wavenumber κ of $2.38434\,\mathrm{m}^{-1}$, with a corresponding c_i of $4.412\cdot 10^{-2}\,\frac{\mathrm{m}}{\mathrm{s}}$.

3.2.1 Problem parameters

For comparison, the time-dependent code[6] was initialized with a corresponding background and density profile. The parameters are given generally in preprocessor definitions:

Listing 3.3: Parameter definitions

```
// Physical constants
const double g = 9.81;
const double rho_0 = 1; // Units of kg / L

// Pysical parameters
const double pertur_k = 2.38434; // Wavelength of most unstable perturbation
const double LENGTH_X = 8*M_PI/pertur_k; // 4 times the most unstable wavelength
const double LENGTH_Z = 1; // depth 1
const double delta_rho = 0.01; // Top to bottom density difference
const double RI = 0.15; // Richardson number at the centre of the pycnocline
const double dz_rho = 0.1; // Transition length for rho
const double dz_u = 0.1; // Transition length for u

const double N2_max = g*delta_rho/2/dz_rho; // Maximum N2
const double delta_u = 2*dz_u*sqrt(N2_max/RI); // Top−to−bottom shear
```

[6] The full source listing for this case is listing A.2, to provide context for the sections quoted here.

These parameters include the unstable wavenumber (`pertur_k`), and the shear strength (`delta_u`) is computed based on the pycnocline width and the desired Richardson number (`RI`). The domain is a periodic channel in x, with free-slip boundaries at the top and bottom in z. Much like in section 3.1, this is simply specified:

Listing 3.4: Domain definitions

```
   // Resolution in X, Y (1), and Z
55 int size_x() const { return NX; }
   int size_y() const { return 1; }
   int size_z() const { return NZ; }

   /* Set periodic in x, free slip in z */
60 DIMTYPE type_z() const {return FREE_SLIP;}
   DIMTYPE type_default() const { return PERIODIC; }
```

The parameters `NX` and `NZ` are this time global variables, so that they can be specified at the command-line (runtime) for easier resolution comparison. The basic MPI and runtime initialization is nearly identical to the minimal case, with the addition of minor command-line processing:

Listing 3.5: Initialization and command line processing

```
    int main(int argc, char ** argv) {
      /* Initialize MPI. This is required even for single-processor runs,
         since the inner routines assume some degree of parallelization,
195      even if it is trivial. */
      MPI_Init(&argc, &argv);
      if (argc > 1) { // Check command line arguments
        NZ = atoi(argv[1]); // Read in number of vertical points, if specified
      } else {
200     NZ = 256;
      }
      NX = rint(NZ*LENGTH_X/LENGTH_Z);
      if (master()) {
        fprintf(stderr,"Using a grid of %d x %d points\n",NX,NZ);
205   }
      helmholtz mycode; // Create an instantiated object of the above class
      /// Create a flow-evolver that takes its settings from the above class
      FluidEvolve<helmholtz> do_helmholtz(&mycode);

210   do_helmholtz.initialize();
      do_helmholtz.do_run(final_time);
      MPI_Finalize(); // Cleanly exit MPI
      return 0; // End the program
```

```
}
```

3.2.2 Forcing and initialization

Unlike the minimal case, this code includes the interactions between a scalar field (density) and the velocity field. The scalar field acts like an active tracer, which affects the flow around it. Including such a tracer is a relatively simple matter of adding:

```
/* Use one actively-modified tracer */
int numActive() const { return 1; }
```

inside the user-case object (now called `helmholtz` for this section). Furthermore, this case will use the *perturbation* density, $\rho'(x,z) = \rho(x,z) - \bar{\rho}(z)$. The perturbation density is governed by the equation:

$$\rho'_t - \vec{u} \cdot \nabla \rho' = w\bar{\rho}_z, \tag{3.5}$$

where $\bar{\rho}_z$ is given in (3.2a). The $w\bar{\rho}_z$ term introduces a forcing term for the perturbation density, and this is simply implemented:

Listing 3.6: Density forcing

```
135    // Forcing of the perturbation density
       void tracer_forcing(double t, const DTArray & u, const DTArray & v,
          const DTArray & w, vector<DTArray *> & tracers_f) {
        /* Since the perturbation density is a perturbation, its forcing is
          proportional to the background density gradient and the w-velocity */
140     *tracers_f[0] = w(ii,jj,kk)*0.5*delta_rho/dz_rho*pow(cosh((zz(kk)-0.5)/dz_rho),-2);
       }
```

Listing 3.2.2 introduces the forcing function for tracers in this code[7]. As parameters, it takes the current time and velocities (passed as constant references to prevent accidental change) along with a vector of (pointers to) tracer forcings, one per tracer.

Forcing in the momentum equation is implemented almost identically, with the (nonhydrostatic) force due to gravity being given by $g\rho'\rho_0^{-1}$:

Listing 3.7: Vertical-velocity forcing

```
       // Forcing in the momentum equations
130    void vel_forcing(double t, DTArray & u_f, DTArray & v_f, DTArray & w_f,
          vector<DTArray *> & tracers) {
```

[7]A more general function is available that combines velocity and tracer forcings in one large function. This is split into `tracer_forcing` and `vel_forcing` by the `BaseCase` class for notational simplicity.

```
    u_f = 0; v_f = 0;
    w_f = −g*((*tracers[0]))/rho_0;
  }
```

The physics of the problem are also governed by the initial conditions. This initialization is split into a few conceptual pieces. The first is the initialization of the grid coordinates, two one-dimensional arrays:

Listing 3.8: The custom class constructor

```
    helmholtz():
180   xx(split_range(NX)), zz(NZ)
    { // Initialize the local variables
      plot_number = 0;
      last_plot = 0;
      // Create one−dimensional arrays for the coordinates
185   xx = LENGTH_X*(−0.5 + (ii + 0.5)/NX);
      zz = LENGTH_Z*((ii+0.5)/NZ);
    }
```

This is the constructor for the `helmholtz` object, and it serves two purposes. The first is initialization for data analysis (`plot_number` and `last_plot`) and will be discussed there. The second is to initialize one-dimensional arrays `xx` and `zz` to serve as the coordinate axes. Both are defined as type **Array**<**double**,1>, so in lines 185 and 186 they are initialized with a Blitz array expression.

These arrays are constructed differently, however, because of parallelization. Except as needed for taking derivatives, arrays in this program are split by process along the first dimension, generally taken by convention to be the horizontal or streamwise direction. An individual process does not need to "own" an entire horizontal extent, so the `xx` array is constructed with the parameter `split_range(NX)`, rather than the entire extent. The function `split_range` will be discussed along with the other high-level parallelism in section 3.3; in short it determines the correct local subrange (as a Blitz `Range` object) for the current process.

The initialized coordinates are used to define the background velocity profile, once again using Blitz array expressions:

Listing 3.9: Velocity intialization

```
    // Initialize velocities at the beginning of the run
    void init_vels(DTArray & u, DTArray & v, DTArray & w) {
      // Use the background shear profile for u
      u = 0.5*delta_u*tanh((zz(kk)−0.5)/dz_u);
100   v = 0; // The other velocities are initially zero
      w = 0;
      // Also, write out the (zero) initial velocities and proper M−file readers
```

```
    write_reader(u,"u",true);
    write_reader(w,"w",true);
105 write_array(u,"u",0);
    write_array(w,"w",0);
    return;
  }
```

This method also introduces the primary output routine of this code, `write_array` and the companion function `write_reader`. `write_array(Array<double,3> & ar, string name, int sequence)` writes the contents of the Blitz array `ar` to disk, enforcing a serial ordering when run in parallel. The array is written to the file `name.sequence`, with both parameters given in the function call. The sequence number is optional and intended for output of time-varying fields; if it is unspecified the file is written without an extension.

The companion method `write_reader(Array<double,3> ar, string name, bool sequence)` writes a short MATLAB routine by the name of `name_reader.m` that will open the `name.sequence` file written by `write_array`, read in the data with correct byte-ordering, and assign logical dimensions that match those used in the code. The `sequence` parameter specifies whether the resulting MATLAB function should take a mandatory sequence number as an input argument, in order to determine which array from a time-sequence to read in. If the parameter is false (the default), the resulting MATLAB function does not take a sequence number.

Finally, the perturbation density is initialized:

Listing 3.10: Density initialization

```
110 /* Initialze the temperature perturbation to a small value */
  void init_tracer(int t_num, DTArray & rhoprime) {

    /* We want to write out a grid in order to make plots later,
       so let's re-use rhoprime to that end */
115
    // Assign the x-array to the two-dimensional grid
    rhoprime = xx(ii) + 0*kk;
    write_array(rhoprime,"xgrid"); write_reader(rhoprime,"xgrid",false);

120 // Assign the z-array to the two-dimensional grid
    rhoprime = 0*ii + zz(kk);
    write_array(rhoprime,"zgrid"); write_reader(rhoprime,"zgrid",false);

    rhoprime = (cos(pertur_k*xx(ii)))*pow(cosh((zz(kk)-0.5)/dz_rho),-2)
125         *delta_rho*1e-8;
    write_array(rhoprime,"rho",0); write_reader(rhoprime,"rho",true);
  }
```

This method takes two parameters: the tracer number (`t_num`) and the array in which to write its initial value. On initialization, this method is called once per tracer[8] (with the appropriate `t_num`) to define the tracer's initial value. In this particular case, the perturbation density is the only tracer, so `t_num` can be safely ignored.

The initial density perturbation is given on lines 124-125 of listing 3.10 with a horizontal frequency of the (pre-calculated) maximally-growing perturbation. It is vertically localized to the region of the pycnocline by a multiplicative factor of $\text{sech}^2(\frac{z-0.5}{0.1})$, and its maximum amplitude is 10^{-8} times the top-to-bottom density difference. This perturbation is very small compared to the base values, so the resulting growth should be governed by linear theory for a long time.

Additionally, `init_tracer` serves the dual function of writing to disk the (two-dimensional) grid. `rhoprime` is used as the temporary array for this, since it is overwritten by the initial perturbation. The grid initialization and output occurs on lines 117-122, and it follows the same syntax as the output of initial velocities in listing 3.9. Of course, the grid is static in time so no sequence number is necessary.

3.2.3 Data analysis and output

The final piece of the puzzle for this case is analysis and output of the data as it is generated. For this case, there are two components to the analysis; the first is to write the full, two-dimensional fields ($\vec{u}$ and ρ') to disk periodically for visualization, and the second is to write diagnostic data to disk at each step for fine-grained analysis.

Visualization is most convenient with equally-spaced output times (arbitrarily chosen to be every 5 time-units in the length-200 run), but there is no guarantee that the dynamically-chosen timestep will fall nicely on a desired output time. Adjusting the timestep is the function of a specialized method in the user-supplied class:

Listing 3.11: Timestep control

```
    /* Modify the timestep if necessary in order to land evenly on a plot time */
    double check_timestep (double intime, double now) {
      // Firstly, the buoyancy frequency provides a timescale that is not
      // accounted for with the velocity-based CFL condition.
75    if (intime > 0.5/sqrt(N2_max)) {
        intime = 0.5/sqrt(N2_max);
      }
      // Now, calculate how many timesteps remain until the next writeout
```

[8] `init_tracer(int t_num, DTArray & init_val)` is provided for syntactic convenience by the `BaseCase` class. The underlying, general method is `init_tracers(vector<DTArray *> & init_vals)`, which initializes all of them at once through a slightly more cumbersome notation. This form is appropriate if there is substantial coupling between tracers and it would be inappropriate to define their initial values separately.

```
    double until_plot = last_plot + plot_interval - now;
80  int steps = ceil(until_plot / intime);
    // And calculate where we will actually be after (steps) timesteps
    // of the current size
    double true_fintime = steps*intime;

85  // If that's close enough to the real writeout time, that's fine.
    if (fabs(until_plot - true_fintime) < 1e-6) {
      return intime;
    } else {
      // Otherwise, square up the timeteps. This will always shrink the timestep.
90    return (until_plot / steps);
    }
  }
```

This method is called at each full timestep[9] in order to adjust the size of the next timestep. The input parameters are the suggested step size (`intime`, determined by a velocity-based CFL condition) and the current time (`now`), and the return value of the method is used without further modification for the size of the next timestep.

For this case, this adjustment has two parts:

- The first step (lines 73-77 of listing 3.11) is to adjust the timestep based on the background density profile. In the absence of any other interaction with the fluid, a fluid parcel disturbed infinitesimally will oscillate up and down with a frequency proportional to $\sqrt{N^2}$ (hence the term buoyancy frequency). This acts as a linear oscillator, and since the forcing terms are stepped explicitly the timestep must be small enough to resolve the oscillation. The resulting constraint is proportional to $\frac{1}{\sqrt{N^2}}$, and a safety factor of 0.5 is included.

  It is important to note that this timescale is resolution independent, and does not become more restrictive as the grid size approaches zero. This is in contrast to the velocity-based CFL conditions already computed (and included in `intime`), where the maximum permissible timestep approaches zero like $u\Delta x$. This condition is generally only relevant in nearly still flows.[10]

- The second step (lines 78-91) further adjusts the timestep in order to evenly land on output times for analysis. The code determines how many steps (at the current size) remain until

[9] The caveat "full" timestep is required because of initialization. Because there is no previous history, as described in section 2.2.5 the code makes initial steps of fractional size for startup, adding to a single full step.

[10] When the condition is violated, the resulting instability has a very curious pattern. Short-wave perturbations grow from the numerical instability, but they saturate when they grow sufficiently large that the induced velocities cause the computed (velocity-based) timestep to fall below the buoyancy-condition. This instability is difficult to detect from volume-level data such as total energy, since it does not grow indefinitely.

the next output time, rounding up, and it returns a step size which will hit the output time in exactly that many steps. The current input time is used if it will land "close enough" (a margin of 10^{-6} here) to the output time, in order to accommodate rounding error.

Reaching the output time is a signal for the code to write out diagnostic data. At each output time, all active fields (u, w, and ρ) are written to disk for visualization, and this is done in the `analysis` method:

Listing 3.12: Periodic field otput

```
   /* The analysis routines are called at each timestep, since it's
145   impossible to predict in advance just what will be interesting.
      This function will write out volume average data and flow fields. */
   void analysis(double time, DTArray & u, DTArray & v, DTArray & w,
         vector<DTArray *> & tracer, DTArray & pressure) {
      /* If it is very close to the plot time, write data fields to disk */
150   if ((time - last_plot - plot_interval) > -1e-6) {
         plot_number++;
         if (master()) fprintf(stderr,"*");
         write_array(u,"u",plot_number);
         write_array(w,"w",plot_number);
155      write_array(*tracer[0],"rho",plot_number);
         last_plot = last_plot + plot_interval;
      }
```

This section updates `plot_number`, which is used as a sequence number for outputs, as well as the last plot time.

Additionally, the onset of the Kelvin-Helmholtz billows is visible in volume-integrated fields, especially potential energy ($\int \rho' gz \mathrm{d}V$). These fields are also computed here and written each timestep to a text file for further processing. The relevant code is also in `analysis`, and the computation is simple:

Listing 3.13: Per-timestep analysis

```
      // Also, calculate and write out useful information: maximum u, w, and t'
      double max_u = psmax(max(abs(u)));
160   double max_w = psmax(max(abs(w)));
      double max_t = psmax(max(abs(*tracer[0])));
      // Energetics: mean(u^2), mean(w^2), and mean(rho*h)
      double usq = pssum(sum(u*u))/(NX*NZ);
      double wsq = pssum(sum(w*w))/(NX*NZ);
165   double rhogh = pssum(sum(*tracer[0]*g*zz(kk)))/(NX*NZ);
      if (master()) fprintf(stderr,"%.2f: %.2g %.2g %.2g\n",time,max_u,max_w,max_t);
      if (master()) {
```

```
        FILE * vels_output = fopen("velocity_output.txt","a");
        if (vels_output == 0) {
170       fprintf(stderr,"Unable␣to␣open␣velocity_output.txt␣for␣writing\n");
          exit(1);
        }
        fprintf(vels_output,"%.16g␣%.16g␣%.16g␣%.16g␣%.16g␣%.16g␣%.16g\n",
            time,max_u,max_w,max_t,usq,wsq,rhogh);
175     fclose(vels_output);
      }
    }
```

The corresponding code is on lines 163–165, computing the volume average of u^2, w^2, and $\rho' gz$. These are related to the full energies by normalization factors. The computation also makes use of the `pssum` function, which is defined in this work to act on parallel arrays; this is described in more detail in section 3.3, along with the other technical aspects of parallel computation.

3.2.4 Accuracy

The accuracy of this case is measurable via the growth rate of the most unstable mode. The initialization, however, does not take into account the vertical structure of the most unstable mode, instead perturbing only density with the proper horizontal wavelength. This projects onto a number of eigenmodes of the Taylor-Goldstein equation, but over time the fastest-growing mode becomes dominant. The mode's energy is proportional to the square of its amplitude, and since the form of the mode is time-invariant a fixed fraction of the mode's energy is in (perturbation) potential energy, provided the total amplitude is small. The potential energy can then be used to determine the instantaneous growth rate, via $\omega_i = (t_2 - t_1)^{-1} \ln\left(\frac{PE(t_2)}{PE(t_1)}\right)$.

The results for several different resolutions are shown in figure 3.1, and they show the spectral accuracy of this code. The initial departure from the theoretical growth rate comes from the initialization of the problem. By initializing with only a density perturbation rather than the exact form of the growing instability, the resulting evolution is governed by a mix of the growing instability, a companion decaying mode, and several translating modes. The resulting growth of the potential energy is affected by the interference between these modes (creating the dips in figure 3.1) until the growing mode is large enough to dominate these effects.

Doubling the resolution from 32 points in the vertical[11] (top line of 3.1) to 64 points (second from top) reduces the error by approximately two orders of magnitude. Doubling the resolution again to 128 points in the vertical reduces the error to within the limits of MATLAB's `polyeig` routine.

[11] with a corresponding increase in horizontal resolution to keep the grid isotropic

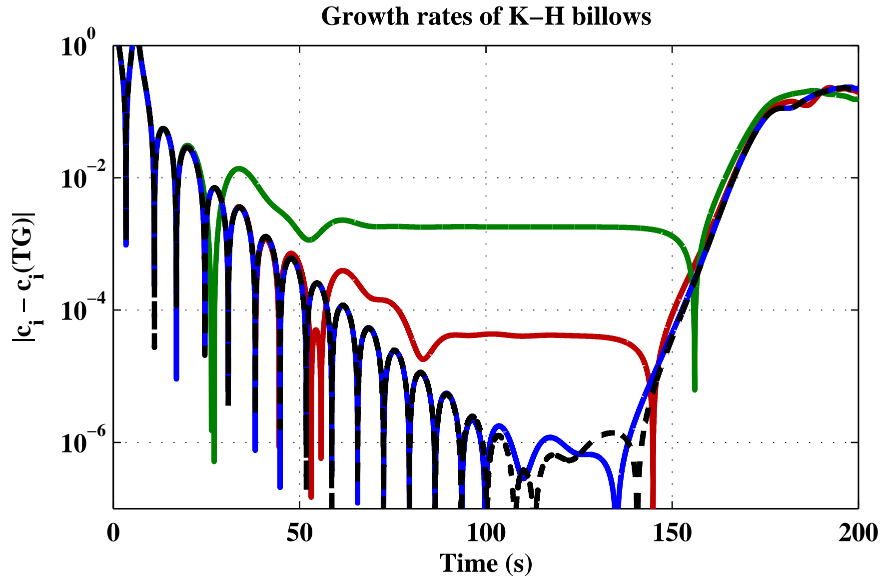

Figure 3.1: Departure from the computed growth rate for Kelvin-Helmholtz billow formation in time-dependent simulations for 32 (top), 64, 128, and 192 (bottom, dashed) points in the vertical. For early times, the error is dominated by contribution from secondary modes. The net error reduces to a resolution-dependent level, before again increasing due to nonlinear effects. The exact value was computed with `polyeig` in MATLAB, and itself is accurate to approximately the 10^{-6} level due to rounding effects.

3.3 Parallel processing

The code in section 3.2 makes use of parallel processing, where the computational arrays are split between processors (as in figure 2.20) which communicate with MPI message passing. Since the case-specific operations of initialization and forcing are pointwise-independent, they do not require significant modification for parallel processing. The allocation of arrays, however, does rely on the details of the parallelism.

3.3.1 Allocation

In section 3.2, the parallelism on allocation is used in the constructor (listing 3.8) for the one-dimensional x-coordinate. This uses the `split_range(`**int** `N)` function, which returns the range (as a Blitz `Range` object) of indices (along the first dimension) that the current processor[12] should

[12]All of the parallel helper functions described in this section take an MPI communicator as an optional argument. The default value for the communicator is `MPI_COMM_WORLD` (all processors), and this suffices for the

control in an array with N points in the first dimension. This function is only the simplest of the helper functions used for parallel allocation, and it is only suitable for one-dimensional arrays.

Other helper functions are defined (in Par_util.hpp) for allocation of multi-dimensional arrays. They are:

- split_range(**int** N) is described above, and returns the local subrange of the first dimension that the current processor should hold, if the global array has its first dimension of size N.
- alloc_array(**int** Nx, **int** Ny, **int** Nz) allocates a full, three-dimensional array of global dimensions $Nx \times Ny \times Nz$, properly split among all processors. This function returns the allocated (local portion of the) array as a **DTArray**$*$, which must be deleted later to avoid memory leaks.
- For inline initialization where a **DTArray** is used for the duration of a function or object, the more specific functions alloc_lbound, alloc_extent, and alloc_storage (all taking (**int** Nx, **int** Ny, **int** Nz) as parameters). They return the lower bounds, extents, and storage order[13] respectively. Inline initialization of an array would be done through the notation:

  ```
  DTArray new_array(alloc_lbound(Nx,Ny,Nz),
     alloc_extent(Nx,Ny,Nz),alloc_storage(Nx,Ny,Nz));
  ```

  Using the explicit initialization also allows creating arrays of general Blitz-templated types, rather than requiring the use of a **DTArray**.

3.3.2 Analysis routines

The other influence of parallel execution on the case-specific code is in the analysis routines. While simply writing out the respective fields to disk (listing 3.12) is straightforward and does not directly consider parallelism[14], the volume-analysis routines involve reduction operators that give a scalar answer for parallel input.

These routines are also specified in Par_util.hpp, and they are generally thin wrappers around the respective Blitz++ array reduction operators.

case-specific code. Other communicators are used internally in the multigrid algorithm, which does not always use every processor for inner iterations.

[13]Blitz++ separates the memory storage ordering of an array into a separate object. This allows flexibility between C, Fortran, and more general array orderings. This work uses the default C-language storage order (column major) for one-processor runs and a custom storage order for multiprocessor runs, where the second index (spanwise columns) are stored together in memory.

[14]The function write_array internally ensures proper serialization of output. It must be called collectively, over all processes, in order to finish execution without deadlock, but since array output involves the entire logical array rather than a processor-local segment arranging this is no difficulty.

The first-used operator is `master()`, which simply returns **true** if the calling process is the master process (MPI rank 0), and **false** otherwise. If this is a single-processor run, then this function will always return **true**. In the analysis routines, this is used for screen and file output, to prevent output from being written multiple times.

The other operators come in two classes: scalar operators that act on individual (per-processor) and vector operators that act on an entire (split) array. Both are notationally convenient wrappers around the more cumbersome `MPI_Reduce` operator, and they serve to keep the details of the parallel implementation abstracted away from the details of data analysis.

The scalar operators are implemented in the functions `pssum`, `psmin`, `psmax`, `psany`, and `psall`, with the first two letters of each function name standing for "parallel, scalar". These functions are implemented via template, and they can take as a parameter values of type **int**, **float**, **double**, `complex`<**float**>, and `complex`<**double**>[15] Each function takes one per-processor argument, performs the appropriate mathematical reduction ("any" and "all" are "logical and" and "logical or", respectively), and returns the result to all processors.

The vector operations are implemented in the functions `pvsum`, `pvmax`, and `pvmin`. They take Blitz++ arrays (including the specialized **DTArray**) containing the same permissible datatypes as for the scalar reduction operations. These functions combine the scalar reductions with built-in Blitz array reductions: for an array `A`, `pvsum(A)` is equivalent to `pssum(sum(A))`. These functions are additionally templated to operate on arrays with any number of dimensions.

Unfortunately, the interaction between the reduction operations and Blitz expressions is incomplete. Because of the design of the Blitz array library, array expressions are combined at compile-time and do not have the same basic types as the underlying arrays. Performing a reduction operation on an expression must be done with the scalar functions; for example for arrays `A` and `B`, the maximum of `A+B` is given by `psmax(max(A+B))`.

These array reduction operators behave properly in the single-processor case, reducing to the Blitz operators (for the vector reductions) or the identity operator (for scalar reductions).

3.4 The translating dipole and boundary interaction

Using this code to compute flows with no-slip walls is only slightly more complicated than the free-slip case of 3.2. To illustrate this, this section covers the code required to simulate the two-dimensional collision of a dipole with a no-slip boundary, repeating the case in Clercx and Bruneau [2006] presented as an appropriate test case for simulation codes.

[15] The datatype restriction comes from a mapping between the C++ datatype and MPI datatypes. This mapping is done with the `MPI_Typer` class defined in `Splits.hpp`, and these datatypes are added to the mapping there.

The initial conditions consist of a pair of opposite-signed "shielded monopoles" [Kramer et al., 2007], with local vorticity given by the functional form:

$$\omega(r) \propto \left(1 - \frac{r^2}{r_0^2}\right) \exp\left(-\frac{r^2}{r_0^2}\right), \tag{3.6}$$

where r is the distance from the centre of the monopole and r_0 is the characteristic scale. For this section, the simulations are conducted on a grid of physical dimensions 2×2 with $r_0 = 0.1$. The comparatively short radius ensures that the initial disturbance is isolated well away from the boundary. Each shielded monopole induces the velocities: [Clercx and Bruneau, 2006, eqn. 10]

$$u \propto -\frac{1}{2}(z - z_0) \exp\left(-\frac{r^2}{r_0^2}\right) \tag{3.7a}$$

$$w \propto \frac{1}{2}(x - x_0) \exp\left(-\frac{r^2}{r_0^2}\right) \tag{3.7b}$$

given the central location (x_0, z_0). For this section, the positive-core pole is initially located at $(0.1, 0)$, and the negative-core pole is located at $(-0.1, 0)$ (illustrated in figure 3.2), with the initial velocities being the sum of (3.7) applied to each pole individually. The close initial spacing allows the poles to interact, producing a strong downward-translating dipole[16] that interacts with the no-slip boundary around $t = 0.35$.

The remaining freedom in initialization is the vortex strength. This section follows Clercx and Bruneau [2006] in normalizing the initial kinetic energy ($0.5 \int u^2 + w^2 dV$) to 2, which coincidentally gives a root-mean-squared velocity of 1. This RMS velocity is used with the domain half-width (1) to define the Reynolds' number for the simulation. This condition is satisfied when the strength of each shielded monopole is approximately 299.5284.[17]

[16]There is also a weaker, upward-translating dipole that is produced from the opposite-signed "shield" around the cores. This translates much more slowly, and does not significantly impact the analysis.

[17]This is incorrectly given as 320 in Clercx and Bruneau [2006], but correctly listed as approximately 300 in Kramer et al. [2007]. The discrepancy comes from the superposition of the two poles; the opposite-signed shield from the neighbouring pole adds to the vorticity in the core.

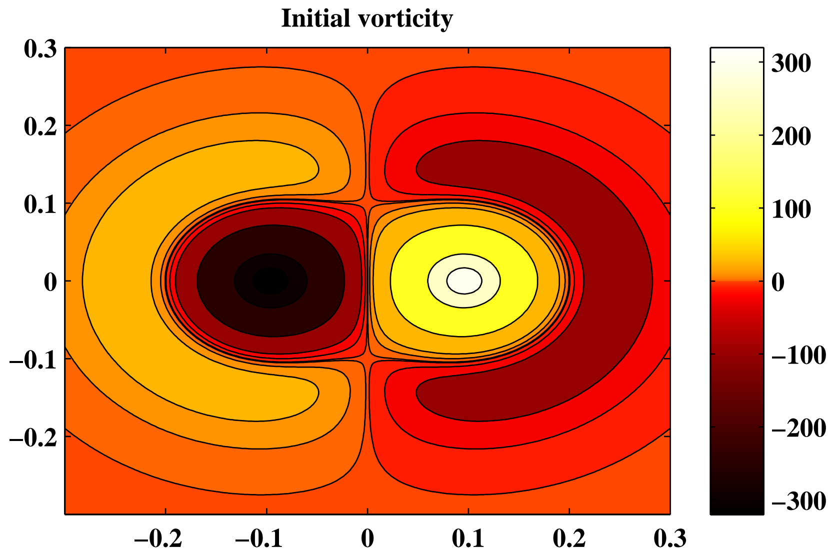

Figure 3.2: Initial conditions for the dipole/wall interaction.

3.4.1 Initialization

Listing 3.14: Dipole domain initialization

```
   // A no−slip box, with periodic spanwise
50 DIMTYPE type_x() const { return NO_SLIP; }
   DIMTYPE type_y() const { return PERIODIC; }
   DIMTYPE type_z() const { return NO_SLIP; }

   // Use a grid−scale Rynolds−number of 1250
55 double get_visco() const {
     return 1.0/1250;
   }

   // Give a 2x1x2 box
60 double length_x() const { return 2; }
   double length_y() const { return 1; }
   double length_z() const { return 2; }
```

The domain setup is as simple as in listing 3.4 for the Kelvin-Helmholtz billows, and is reproduced in listing 3.14. The new features of the dipole-wall interaction are that the x and z dimensions are defined as no-slip surfaces (`NO_SLIP` in `type_x` and `type_z`), and the nonzero viscosity, given by the return value of `get_visco` on line 55.

This run is unstratified, so there is no need to define or initialize any tracers. Velocity initialization is also straightforward according to (3.7), but it must take into account the nonuniform grid spacing implied by the Chebyshev expansion:

Listing 3.15: Grid initialization

```
void init_vels(DTArray & u, DTArray & v, DTArray & w) {
185   xx = −length_z()/2∗cos(M_PI∗ii/(szx−1));
      yy = −(length_y()/2) + length_y()∗(ii+0.5)/szy; // unused in 2D
      zz = −(length_z()/2)∗cos(M_PI∗ii/(szz−1));
```

The velocity initialization is then a direct implementation of (3.7) with array expressions:

Listing 3.16: Velocity initialization

```
      u = 299.5284/2∗(
        +(zz(kk)−D1_Z)∗exp(−(pow(xx(ii)−D1_X,2)+pow(zz(kk)−D1_Z,2))/RAD2) −
195     (zz(kk)−D2_Z)∗exp(−(pow(xx(ii)−D2_X,2)+pow(zz(kk)−D2_Z,2))/RAD2));
      w = 299.5284/2∗(
        −(xx(ii)−D1_X)∗exp(−(pow(xx(ii)−D1_X,2)+pow(zz(kk)−D1_Z,2))/RAD2) +
        (xx(ii)−D2_X)∗exp(−(pow(xx(ii)−D2_X,2)+pow(zz(kk)−D2_Z,2))/RAD2));
      v = 0;
```

The flow evolves freely from its initial state, and no further forcing is required. The timestep restriction is properly handled by the CFL condition, and the timestep is further controlled with code substantially identical to listing 3.11 for the Kelvin-Helmholtz billows.

3.4.2 Analysis and the gradient operator

Comparison with Clercx and Bruneau [2006] and Kramer et al. [2007] require computation of kinetic energy and enstrophy[18], along with periodic views of the vorticity field. The first two require volume integration and ideally should be computed at each timestep, and the latter two require computation of the vorticity. This computation is accomplished via re-use of a helper-class defined for the internal Navier-Stokes integration.

This class is creatively named `Grad`, and it is defined in `grad.hpp`. The primary function of the class is to provide a wrapper around differentiation in physical coordinates, including the influence of coordinate mapping. This presents a unified interface to both internal and case-specific code, so that the code can be as agnostic as possible about the underlying physical configuration. The case-specific object gains access to the internal gradient operator via the

[18] Enstrophy is defined as $\frac{1}{2}\int \|\omega\|_2^2 \,\mathrm{d}V$, where vorticity is ω.

set_jac(Grad * in_op) method. The object passed in is owned by the underlying FluidEvolve class[19], but the user-object can keep a reference around for use during the simulation.

Using the Grad object is a two-step procedure. First, Grad::setup_array[20] initializes the gradient operator based on the array to be operated upon, with exact specification of the array expansion types in each (numerical) dimension. After the initialization, the physical derivatives are available with Grad::get_dx(**DTArray** * out, **bool** accumulate), get_dy, and get_dz. If the accumulate parameter is set (the default is **false**), then the derivative computation will add to the output array rather than overwrite its contents. The array setup and derivative computations are split in this model so that the gradient object can re-use derivatives along the computational coordinates as necessary, given any coordinate mapping.

In the case of the dipole-wall simulation, the computation of the two-dimensional vorticity ($w_x - u_z$) is straightforward with this framework:

Listing 3.17: Vorticity computation

```
void compute_vorticity(DTArray & u, DTArray & w) {
  // Compute vorticity
  gradient_op->setup_array(&u,CHEBY,FOURIER,CHEBY);
  // Put du/dz in vorticity
  gradient_op->get_dz(&vorticity,false);
  // Invert that to get -du/dz
  vorticity = vorticity*(-1);
  // And add dw/dx
  gradient_op->setup_array(&w,CHEBY,FOURIER,CHEBY);
  gradient_op->get_dx(&vorticity,true);
}
```

A nearly identical approach could be used to calculate the local buoyancy frequency without postprocessing for stratified simulations.

Once the vorticity is computed, the total kinetic energy and enstrophy can be computed by numerical quadrature on the domain. This is accomplished through helper functions that compute and return quadrature weights on a per-dimension basis. The computation of the weights (uniform weights for Fourier, sine, and cosine expansions, with Gauss-Lobatto weights [Trefethen, 2000] for Chebyshev expansions) is triggered via the compute_quadweights function, defined in Science.hpp. This is generally called from the case-specific constructor:

[19]This implies that it is not safe to use the Grad object after the end of the simulation, when the FluidEvolve object has been destroyed. For the standard "initialize, simulate, exit" paradigm used in these cases, this is not an issue.

[20]The full prototype for this function is setup_array(**DTArray** * A, S_EXP Tx, S_EXP Ty, S_EXP Tz), where Tx, Ty, and Tz are the expansion types in the first, second, and third dimensions respectively.

Listing 3.18: Quadrature weight computation

```
   userControl(int s):
     // Setup a 2D run, of size S x 1 x S
     szx(s), szy(1), szz(s),
     // Write out fields every 0.005 timeunits for detailed graphics
230  plotnum(0), plot_interval(.005),
     nextplot(plot_interval), lastplot(0),
     itercount(0),last_writeout(0),
     // Initalize arrays for 1D grid coordinates
     xx(split_range(szx)), yy(szy), zz(szz),
235  // Initalize the array for vorticity computation
     vorticity(alloc_lbound(szx,szy,szz),
        alloc_extent(szx,szy,szz),
        alloc_storage(szx,szy,szz)) {
       compute_quadweights(szx,szy,szz,
240       length_x(),length_y(),length_z(),
          type_x(),type_y(),type_z());
       if (master()) {
         printf("Using array size %d\n",s);
       }
245  }
```

where the function is called on lines 239-241. This constructor also initializes one-dimensional arrays for coordinates in x, y, and z dimensions (line 234) and a three-dimensional array for vorticity (lines 236-238) using the allocation routines in section 3.3.

The computed quadrature weights are valid for a "numerical box" of dimensions specified in the last three parameters to `compute_quadweights`. The weights are stored internally, and they are accessed per-dimension via `get_quad_x()`, `get_quad_y`, and `get_quad_z`, which return a (constant) pointer to one-dimensional arrays of weights[21]. These weights can then be used in an array expression to explicitly integrate over the domain. This is used in the analysis code for the dipole-wall interaction.

Listing 3.19: Computation of kinetic energy and enstrophy

```
120  // Compute (twice) enstrophy, sum(vort^2)
     enst = enst_record[itercount-last_writeout-1] =
       pssum(sum((*get_quad_x())(ii)*(*get_quad_y())(jj)*
            (*get_quad_z())(kk)*vorticity*vorticity));
```

[21] Using one-dimensional arrays here serves two purposes. First, it saves space in the common case of an unmapped grid, since a full three-dimensional array is not necessary. Second, it allows for integration over some-but-not-all dimensions, which is useful for computing quantities like vertical averages.

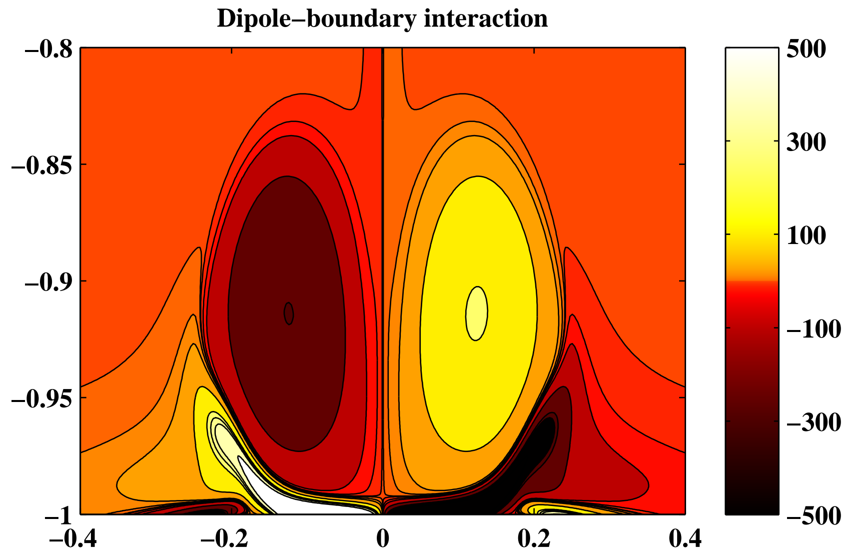

Figure 3.3: The dipole-boundary interaction at $t = 0.38$, near the time of greatest enstrophy, for a 1024×1024 grid. The dipole induces opposing vorticity at the boundary, which begins to wrap around the primary dipole. The boundary vorticity is locally more intense than the primary dipole pair.

```
125     // And KE sum(u^2+w^2). It needs to be divided by 2 for true energy.
        ke = ke_record[itercount−last_writeout−1] = pssum(sum(
              (*get_quad_x())(ii)*(*get_quad_y())(jj)*(*get_quad_z())(kk)*
              (pow(u(ii,jj,kk),2)+pow(v(ii,jj,kk),2)+pow(w(ii,jj,kk),2))));
```

The resulting array expressions make use of `get_quad_weight`, the array-expression primitive `sum`, and the parallel-scalar sum (`pssum`) defined in section 3.3. The computed values are stored in a record array for periodic output to a text file, in a manner substantially identical to that of listing 3.13 for the Kelvin-Helmholtz billows.

3.4.3 Convergence

This case is comparable with both Clercx and Bruneau [2006] and Kramer et al. [2007], which give details on both the total enstrophy in the domain and the location of the primary dipole pair over time. Both of these are used her for comparison.

As the translating dipole nears the boundary, the viscosity and no-slip boundary conditions produce a thin but intense layer of vorticity of opposite sign to each dipole half. This boundary

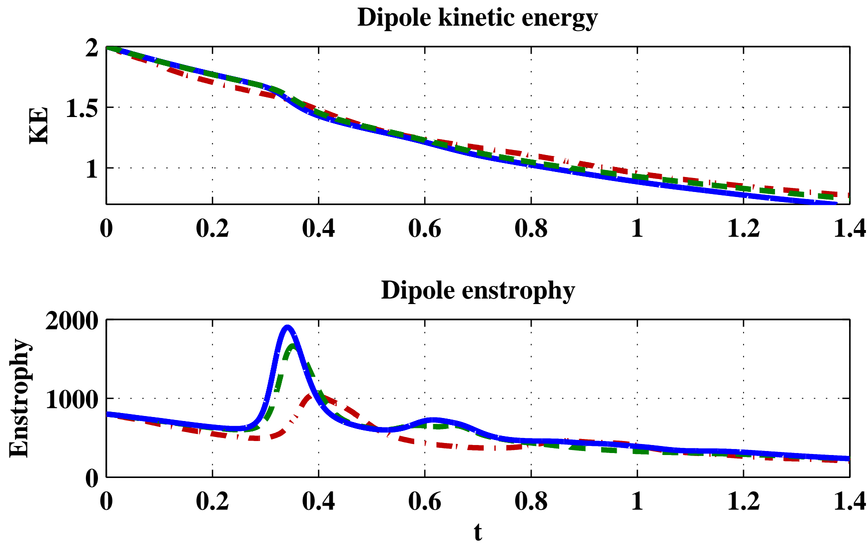

Figure 3.4: The time-evolution of kinetic energy (top) and enstrophy (bottom) for the dipole-boundary interaction at 64×64 (dot-dashed), 128×128 (dashed), and 1024×1024 (dot-dashed) grid points. The intermediate grid sizes of 256×256 and 512×512 are not distinguishable on this graph from 1024×1024

vorticity layer is lifted away from the boundary as the dipole makes its closest approach (see figure 3.3) and wraps around the main dipole. This boundary vorticity also has secondary instabilities at high Reynolds numbers [Kramer et al., 2007], but those are suppressed at the Reynolds number of 1250 used here.

This thin boundary layer holds a great deal of enstrophy. During the approach and interaction (see figure 3.4), the total enstrophy in the domain approaches 2000, over double the starting value of 800. This approach also dissipates energy rapidly, corresponding with the dip in kinetic energy around $t = 0.38$ in figure 3.4. Proper resolution of the thin boundary layers is challenging, but the broad features are resolved with a grid size of at least 256×256.

Tables 3.1 and 3.2 quantitatively compare the total enstrophy in the domain at its maximum and the location and strength of the positive vortex at $t = 0.6$. Additionally, both Clercx and Bruneau [2006] and Kramer et al. [2007] have simulated an identical setup[22] with a spectral method based on a vorticity-streamfunction formulation. The results from this work compare favourably with their converged results.

[22] Kramer et al. [2007] used a slightly simpler periodic channel geometry, with periodic boundary conditions at $x = \pm 1$. This did not significantly affect their results, and the paper reports that doubling the domain length in trials altered the position of the dipole by only $O(10^{-3})$

Case	Enstrophy	Time
64×64 grid	1046.038	0.392941
128×128 grid	1663.578	0.352281
256×256 grid	1882.399	0.341870
512×512 grid	1896.657	0.341510
1024×1024 grid	1899.921	0.341369
Clercx and Bruneau [2006]	1899	0.3414
Kramer et al. [2007]	1899.2	0.3414

Table 3.1: Maximum enstrophy and time of occurance for the dipole-wall interaction at grid sizes from 64×64 through 1024×1024 along with comparisons with Clercx and Bruneau [2006] and Kramer et al. [2007].

Case	x	y	$\omega_{\max}$
64×64 grid	0.2708	0.2669	165.4408
128×128 grid	0.1845	0.1319	238.8713
256×256 grid	0.1534	0.1309	216.7327
512×512 grid	0.1501	0.1274	218.9647
1024×1024 grid	0.1514	0.1257	219.2434
Clercx and Bruneau [2006]	0.151	0.126	219.4
Kramer et al. [2007]	0.1506	0.1260	219.29

Table 3.2: Location and strength of the maximum positive vorticity in the primary dipole at $t = 0.6$, at grid sizes from 64×64 through 1024×1024 along with comparisons with Clercx and Bruneau [2006] and Kramer et al. [2007].

3.5 Spectral filtering

The primary source of error in these simulations has been unresolved scales. Transitions sharper than the grid frequency simply cannot be resolved on a discrete grid, and they are associated with spatial frequencies higher than the Nyquist frequency. Furthermore, the nonlinearity of the momentum equations can generate high (unresolved) frequencies even when the original velocity fields are smooth and otherwise well-resolved.

This problem is compounded with aliasing error. Although the formulation of the advection equations used in this work (2.54a) minimizes the impact of aliasing for waves near half of the Nyquist frequency, higher-frequency waves can still produce significant, unphysical aliasing into the low frequencies dominated by the physical flow. The effective result is the buildup of energy at the grid scale and eventual instability due to “spectral blocking” [Boyd, 2001].

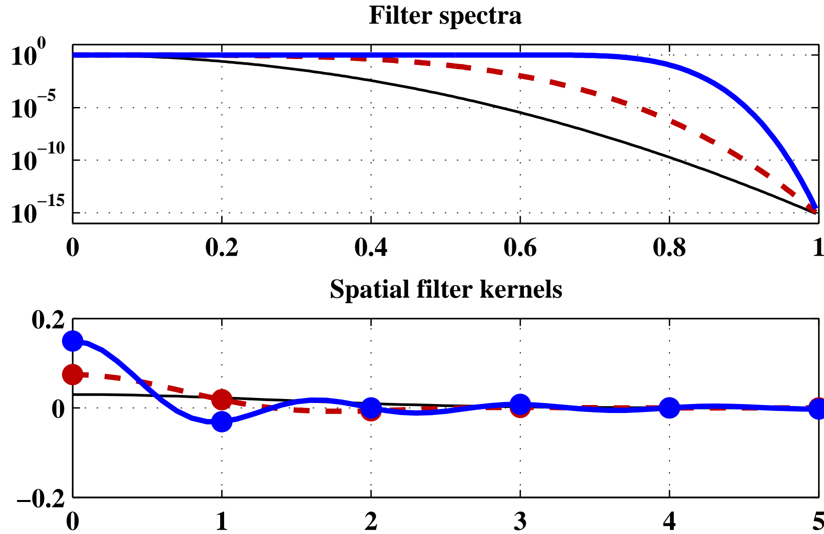

Figure 3.5: Spectra (top, log scale) and kernel functions (bottom) for the exponential filter with cutoff (solid, with cutoff of 60% of the Nyquist frequency and second-order falloff), fourth-order hyperviscosity (dashed), and for comparison second-order, positivity-preserving viscosity (thin, solid).

Often, this situation is remedied through a "$2/3$ cutoff filter" [Orszag, 1971], which removes the upper $1/3$ of the wavenumber spectrum at each timestep, before and after the nonlinear terms. With quadratically nonlinear terms, this ensures that aliasing error cannot contaminate the preserved frequencies, and any signal that would be above the cutoff frequency is removed. Unfortunately, this filter is extremely sharp, creating strong Gibbs oscillations over a wide area near a sharp transition.

Additionally, the exact-preservation property is lost when using a mapped grid. There, the derivative mapping of (2.72) contributes a spatially-varying term that is *multiplied* with the function after taking the derivative, creating a nonlinear term of the form $(f(x,y)uu_\alpha)$ in one dimension. This additional multiplication becomes a *convolution* in spectral space, expanding the range of frequencies subject to possible aliasing in the nonlinear term. This allows aliasing error to spread beyond the upper third of frequencies, and so it would not be completely removed by this filter.

Instead, this work uses by default an exponential filter with cutoff [Godon and Shaviv, 1993]. After the advection step (2.8a) and before the pressure projection (2.8b), each component of

velocity (and each tracer) is multiplied in spectral space (per-dimension) by:

$$s(\kappa) = \begin{cases} 1 & \text{if } |\kappa| \leq \kappa_{\text{cut}} \\ \exp\left(-\alpha\left(\frac{|\kappa|-\kappa_{\text{cut}}}{\kappa_{\text{nyq}}-\kappa_{\text{cut}}}\right)^{\beta}\right) & \text{if } |\kappa| > \kappa_{\text{cut}} \end{cases} \tag{3.8}$$

where α, β, and κ_{cut} set the filter strength, order, and cutoff frequency respectively[23]. The grid-scale wave of the Nyquist frequency is reduced by a factor of $\exp(-\alpha)$. Most importantly, waves below the cutoff frequency are left unchanged; a simulation where the dissipation scales are resolved will be essentially unaffected by filtering.

This filter is continuous in κ to the β order, making the overall filter order $\beta + 1$ [Hesthaven et al., 2007]. Additionally, the resulting spatial kernel function (figure 3.5) is sharply peaked, at the expense of oscillations. Local features stay localized with this filter kernel, avoiding artificial dissipation, but the ringing may affect nearby regions.

The filter parameters are controlled in this work through a trio of global variables, which are fixed during the execution of a run. These parameters are:

- `f_cutoff`, which sets the cutoff frequency (κ_{cut}) where the exponential filter begins,
- `f_order`, which sets the falloff order (β) after the cutoff frequency, and
- `f_strength`, which sets the filter strength (α) at the Nyquist frequency.

The default values are 0.6, 2, and 20 respectively, setting the exponential filter to second-order rolloff that begins at 60% of the Nyquist frequency, and the Nyquist frequency itself is reduced to $2 \cdot 10^{-9}$ of its unfiltered strength.

3.5.1 Hyperviscosity

The relatively weak filtering of the exponential filter does an excellent job of preserving sharp transitions, but the strong ringing can be problematic for density fields. As an option, this work also implements a hyperviscosity filter, which removes a parameter from the exponential filter to have the spectral form:

$$s(\kappa) = \exp\left(-\alpha\left(\frac{|\kappa|}{\kappa_{nyq}}\right)^{\beta}\right), \tag{3.9}$$

which is identical to (3.8) with a zero cutoff frequency. For a fixed order β, the strength of this filter is determined solely by the α parameter. Choosing a single suitable value for α is a difficult

[23] In implementation, κ is normalized so that the Nyquist frequency is 1. For the periodic Fourier transform, $-1 < \kappa \leq 1$, and for the real trigonometric and Chebyshev expansions $0 \leq \kappa \leq 1$.

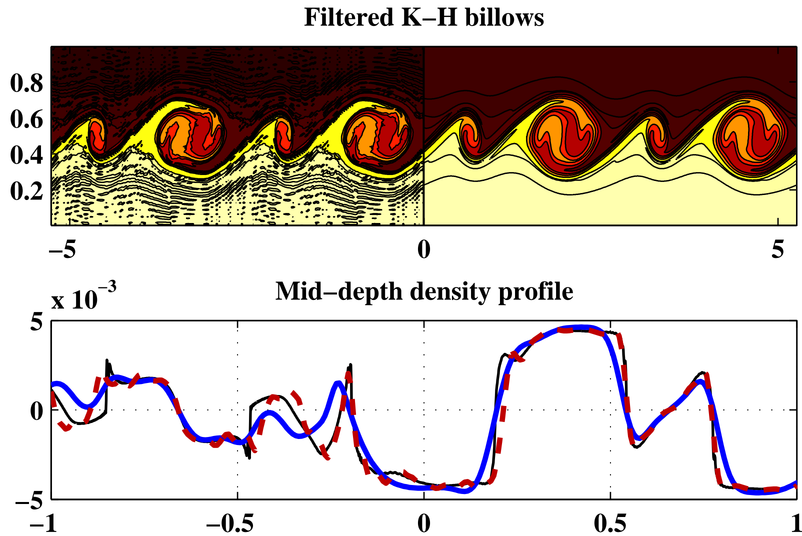

Figure 3.6: Comparison of the exponential filter and hyperviscosity for the Kelvin-Helmholtz billow case of section 3.2 at 256 vertical points. At top, the total density field for the exponential filter (left) and hyperviscosity filter (right). At bottom, a slice through the $z = 0.5$ level, for $-1 < x < 1$, comparing the hyperviscosity (thick, solid) and exponential filter (dashed) with the 1024 vertical-level case (thin, solid).

balancing act between under-damping aliasing error and removing important energy-containing scales during the course of a simulation. To help solve this issue, this work sets the filter strength *implicitly*, by comparing the amplitude spectrum at high frequencies with that at low frequencies and ensuring a favourable ratio.

In implementation, this work takes the notationally-inconvenient choice to re-use the parameters `f_strength`, `f_cutoff`, and `f_cutoff` with different definitions. The hyperviscosity filtering module is activated when `f_strength` is set to a negative value, and the application proceeds in two steps. The first step of this filter analyzes the spectrum and determines the amplitude ratio of the high wavenumbers (those greater that `f_cutoff` in magnitude) to the low wavenumbers (the lowest 10% in magnitude). In the second step, this ratio determines the proper filter strength required to ensure that all wavenumbers greater than `f_cutoff` are reduced to at most 10^{-6} of the low-wavenumber amplitude. The computed strength (α in (3.9)) is multiplied by −`f_strength` to give the final filter strength[24], which is applied to the field's spectrum. The filter strength is computed on a per-direction basis, so the resulting filter varies in time, direction, and flow variable.

[24]This multiplication smooths the filter's application, preventing time-oscillation from abrupt changes in effective filter strength.

The complicated analysis procedure helps avoid the worst of the dissipation caused by the lack of a hard cutoff frequency. When the flow is smooth and does not have high frequency content, the hyperviscosity filter can be disabled entirely, selectively activating itself when the flow becomes marginally-resolved. In practice, the filter is still more dissipative than the exponential filter with cutoff; applying it to the dipole case of section 3.4 on a 256×256 grid gives a maximum enstrophy of only 1319.27, compared with 1882.4 at the same resolution with the exponential filter and 1899.9 at 1024×1024.

However, this filter is much more effective at mitigating oscillations in density fields. The results of applying this filter to the Kelvin-Helmholtz billows of section 3.2[25] is shown in figure 3.6. On the total density field (top panel), the exponential filter (left side) results in significant oscillations, which effectively create new maxima and minima; this is avoided with the hyperviscosity filter (right side). The bottom panel, however, shows that the exponential filter is better than the hyperviscosity filter at preserving sharp transitions when compared to a fully-resolved (1024 vertical points) simulation.

3.6 Mapped grids and internal wave generation

Finally, using this code with a mapped boundary is a fairly simple extension over simpler cases. As an example, consider the generation of internal waves by the interaction of tidal flow with undersea topography[26]

3.6.1 Physical configuration

For an ocean-scale simulation[27], consider a domain 400km in horizontal extent (L) by 5km in vertical extent (H), at a hypothetical mid-latitude where the Coriolis parameter is $f = 0.5 \cdot 10^{-4}\,\mathrm{s}^{-1}$. For simple analysis, additionally assume that the background density profile is linear with a constant buoyancy frequency N_0, and the primary constituent of density is salt. With typical ocean values [Cushman-Roisin and Beckers, 2011], this gives a background (initial) salt profile of:

$$S(z) = 35 - \tfrac{N_0^2}{\beta g} z, \tag{3.10}$$

where g is the gravitational acceleration and $\beta = 7.610^{-4}$ is the fractional density increase per practical salinity unit. This gives a net (initial) density profile of:

$$\rho(z) = \rho_0(1 + \beta(S(z) - 35)), \tag{3.11}$$

[25] The code presented in that section was re-run with a higher-amplitude initial perturbation in order to fully develop the billows.

[26] The code discussed in this section is listed in its entirety as listing A.4.

[27] The configuration for this case was graciously provided by Michael Dunphy, who is looking at similar situations in his ongoing research.

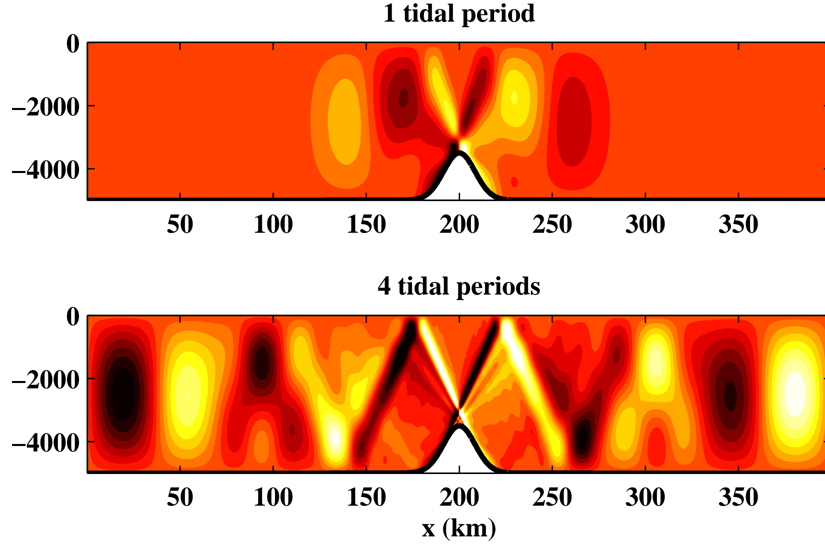

Figure 3.7: Internal waves generated from the interaction of tidal flow with topography, shown through the perturbation salinity $S(t)-S(0)$. After one tidal period (top), mode-one waves have propagated some distance away from the hill. After four tidal periods (bottom), the contribution from higher-mode waves begins forming beams.

where $\rho_0 = 1028\,\mathrm{kg\,m^{-3}}$ for ocean water.

The flow is accelerated by the M2 tide with period 44712s at low amplitude ($1\,\mathrm{cm\,s^{-1}}$)) over an undersea hill, with a profile given by:

$$h(x) = A\exp\left(-\left(\tfrac{x-x_0}{\Delta x}\right)^2\right), \tag{3.12}$$

where $A = 1.5\,\mathrm{km}$ is the hill amplitude, $\Delta x = 200\,\mathrm{km}$ is the location of the hill, and $\Delta x = 12\,\mathrm{km}$ is the width. The interaction with the topography generates internal waves, illustrated in figure 3.7. Provided the background flow is slower than the group velocities of the lowest-mode internal waves, energy can propagate both upstream and downstream.

3.6.2 Grid generation

A simple grid mapping accommodates the hill as the bottom boundary of the domain. Using $\chi \in [0,1]$ and $\zeta \in [-1,1]$, the grid is defined as:

$$x(\chi,\zeta) = L_x\chi \text{ and} \tag{3.13a}$$

$$z(\chi,\zeta) = \tfrac{L_z}{2}(1+\zeta) + \tfrac{1}{2}(1-\zeta)h(L_x\chi), \tag{3.13b}$$

where $L_x = 400\,\text{km}$, $L_z = 5\,\text{km}$, and $h(\cdot)$ is the hill profile as defined in (3.12)

The implementation in code is nearly as straightforward:

Listing 3.20: Grid mapping

```
    bool is_mapped() const {return true;}
    void do_mapping(DTArray & xg, DTArray & yg, DTArray & zg) {
70    xgrid = alloc_array(NX,1,NZ);
      zgrid = alloc_array(NX,1,NZ);
      Array<double,1> xx(split_range(NX)), zz(NZ);
      // Use periodic coordinates in horizontal
      xx = (ii+0.5)/NX; // x−coordinate
75    zz = cos(ii*M_PI/(NZ−1)); // Chebyshev in vertical

      xg = LENGTH_X*xx(ii) + 0*jj + 0*kk;
      *xgrid = xg;

80    hill = H_HEIGHT*exp(−pow(LENGTH_X*(xx(ii)−1/2)/H_LENGTH,2));
      zg = −LENGTH_Z/2+LENGTH_Z/2*zz(kk) + 0.5*(1−zz(kk))*
           hill(ii);;
      *zgrid = zg;

85    yg = 0;

      write_array(xg,"xgrid");
      write_reader(xg,"xgrid",false);
      write_array(zg,"zgrid");
90    write_reader(zg,"zgrid",false);
    }
```

This code marks the grid as mapped with the single-line method on line 68, and it performs the mapping with the following method. The domain length, domain depth, hill height, and hill scale length are parameterized as the compile-time constants `LENGTH_X`, `LENGTH_Z`, `H_HEIGHT`, and `H_LENGTH` respectively, with units in meters. A copy of the grid is also saved in the member variables `xgrid` and `zgrid` for initialization and forcing purposes.

The `do_mapping`(**DTArray**& `xg`, **DTArray**& `yg`, **DTArray**& `zg`) method is quite simple: when called, it initializes the three-dimensional grid[28] into the output parameters `xg`, `yg`, and `zg` respectively. After initialization, the integration routines take the partial derivatives of the mapped grid with respect to the numerical-box coordinates in order to compute the coefficients for (2.72).

[28]Although this method uses three-dimensional arrays, currently only two-dimensional mappings are supported. See section 4.4.1 for further details.

Even with a mapped grid, the grid-length functions `length_x`, `length_y`, and `length_z` retain their traditional meaning. This length is subtracted from the mapping in order to directly determine the constant portion of the coordinate partial derivatives. Without this direct specification, the partial derivatives (for example $\frac{\partial x}{\partial \chi}$ in (3.13a)) would be contaminated by Gibbs oscillations and not give an appropriate mapping.

3.6.3 Initialization and forcing

Initializing this case is somewhat problematic. The objective is to demonstrate the propagation of waves generated by interaction of the tidal flow with the topography, so the initial conditions should have no wave motion. However, this is distinct from having the fluid at rest, because the Coriolis parameter couples u and v independently of internal wave motion. Without wave motion, in the far field the horizontal velocity $u(t)$ should behave like $u_m \sin(\omega_{\text{tide}} t)$ (with $u_m = 1 \frac{\text{cm}}{\text{s}}$), and the spanwise velocity $v(t)$ should be in balance. Initializing with the fluid at rest ($u = v = 0$) would break this balance and create an unwanted transient response. Instead, more compatible initial conditions can be derived from shallow water theory[29], which neglects vertical variation in u and v.

In order to conserve mass, the horizontal velocity must have greater magnitude over the hill crest than in the far field. This requirement gives a profile of:

$$u(t) = u_m \frac{H}{H - h(x)} \sin(\omega_{\text{tide}} t), \tag{3.14}$$

when wave motion is neglected. This is uniformly zero at $t = 0$.

For the spanwise velocity v, the only force acting on it is from rotation, so $v_t = f \cdot u$. Integrating this using (3.14) gives a wave-free profile of:

$$v(t) = -u_m f \omega_{\text{tide}}^{-1} \frac{H}{H - h(x)} \cos(\omega_{\text{tide}} t), \tag{3.15}$$

which is nonzero at $t = 0$. This gives as initial conditions:

$$u(x, y, z, t = 0) = 0, \tag{3.16a}$$

$$v(t = 0) = -u_m f \omega_{\text{tide}}^{-1} \frac{H}{H - h(x)}, \text{ and} \tag{3.16b}$$

$$w(t = 0) = 0. \tag{3.16c}$$

[29] A more robust initial condition would use potential flow theory to include vertical motion, and beginning the simulation would simply amount to "turning gravity on." In practice, initial conditions given by shallow water theory were accurate enough to avoid strong transient responses.

Forcing

The tidal flow must be driven by a pressure gradient. Ideally, that gradient would be implicitly defined based on direct specification of inflow and outflow velocities at the domain edges, but that is not possible with this work when using a Fourier-based periodic discretization in the horizontal. That discretization was chosen because it allows for a mean flow through the domain, but it comes at the price of periodicity – waves leaving one end of the domain will re-enter through the other. Fortunately, that is not a significant issue for this case, where the run is terminated not long after waves exit the domain.

The greatest impact of the periodic discretization in this section is on the velocity forcing. Since direct specification of the boundary condition is impossible without a sponge layer[30], the case requires a direct specification of the forcing. The straightforward approach of taking the derivative of (3.14) does not work because of interaction with the rotational force. Again using the goal profile of $u(t) = u_m \sin(\omega_{\text{tide}} t)$ in the far field along with (3.15), this gives a net forcing[31] of:

$$u_t = -f \cdot v + u_m \left(\omega_{\text{tide}} - \frac{f^2}{\omega_{\text{tide}}} \frac{H}{H - h(x)} \right) \cos(\omega_{\text{tide}} t). \tag{3.17}$$

This is implemented in the forcing function, which must use the most general form because of the combination of rotational and density forces:

Listing 3.21: Forcing function

```
150  // Forcing must be done generally, since both rotation and density are
     // involved
     void forcing(double t, const DTArray & u, DTArray & u_f,
         const DTArray & v, DTArray & v_f, const DTArray & w,
         DTArray & w_f, vector<DTArray *> & tracers,
155      vector<DTArray *> & tracers_f) {
       // Rotation couples u and v, plus a source term for the tide
       u_f = -ROT_F*v + cos(t*TIDE_M2)*TIDE_STRENGTH*
         (TIDE_M2-ROT_F*ROT_F/TIDE_M2*LENGTH_Z/(LENGTH_Z-hill(ii)));
       v_f = ROT_F*u;
160    w_f = -g*beta*(*tracers[0]-S0);
       // And since the salt content is expressed as total content rather
       // than perturbation, no forcing is necessary.
       *tracers_f[0] = 0;
     }
```

[30]See section 4.4.2 for discussion of sponge layers and outflow conditions.

[31](3.17) does not take into account the effects of the topography, except in the correction for rotation. The resulting balance between u and v is not perfect, but it is effective at reducing the variation to negligible levels.

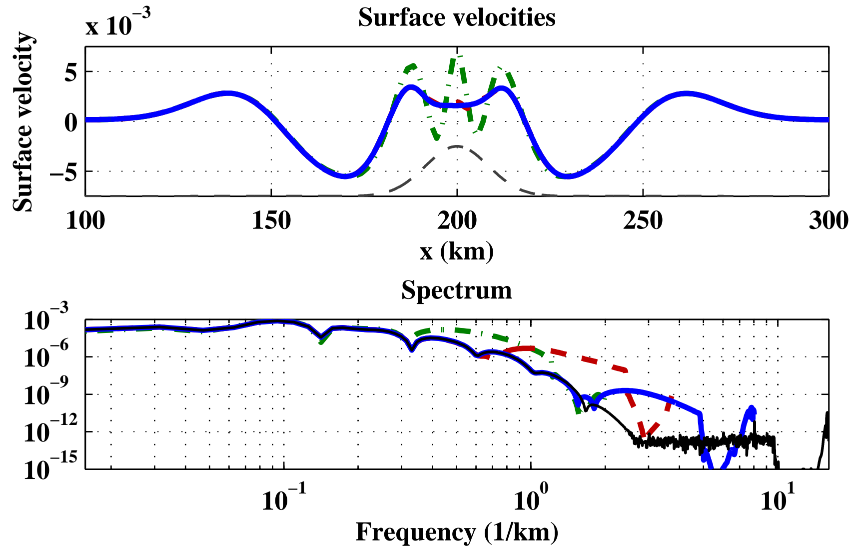

Figure 3.8: Convergence of the internal-wave case after one tidal period, with 256×8 resolution (dot-dashed), 512×16 (dashed), 1024×32 (solid, thick), and 2048×64 (solid, thin). The top view is detail of surface velocity ($u(z=0)$) near the topography, and the bottom is the amplitude of the spectrum of the surface velocity.

3.6.4 Analysis and results

The simplest convergence-check for this case is resolution convergence, the results of which are illustrated in figure 3.8 for surface velocities on grid sizes between 256×8 points through 2048×64. After one tidal period, the surface velocities (top of figure) have converged within plotting error at a grid of 1024×32. This is verified by the amplitude spectrum of the velocity (bottom), where the 1024×32 case is identical to the 2048×64 case for six orders of magnitude in amplitude. Additionally, the 2048×64 case shows the roundoff plateau (at 10^{-12} magnitude), where the spectrum is limited by finite numerical precision.

More quantitatively, the root mean square surface velocity is $2.090369 \cdot 10^{-3} \frac{\text{m}}{\text{s}}$ at 256×8, 1.866669 at 512×16, 1.866290 at 1024×32, and 1.866291 at 2048×64.

Comparison with a nearly identical case run in the model of Lamb [1994] is given in figure 3.9, where the surface velocities and horizontal velocity over the hill crest are plotted after one tidal period. The curves are nearly identical, with the largest differences arising from the tidal forcing. The model of Lamb [1994] allows direct specification of inflow and outflow velocities at the ends of the domain, which eliminates the need for the body forcing of equation (3.17).

Physical comparisons are possible by looking at the waves that propagate away from the

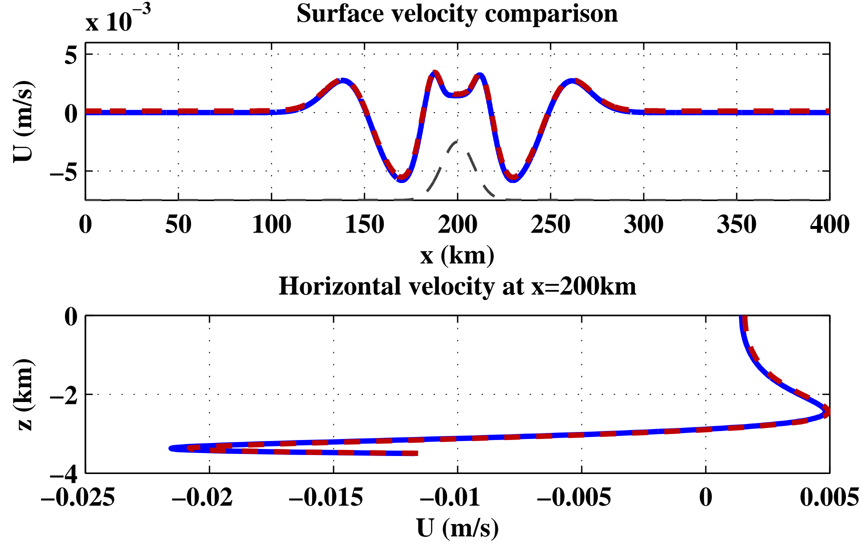

Figure 3.9: Comparison of surface velocities (top) and u-velocity profile at the middle of the domain (bottom) between this work at 2048×64 (dashed) and the model of Lamb [1994] at 4096×258 (solid) at one tidal period. The curves lie nearly on top of one another.

topography. The linearized Navier-Stokes equations are [Kundu and Cohen, 2004]:

$$\begin{aligned} u_t &= -p_x - fv, \\ v_t &= fu, \\ w_t &= -p_z - \rho' \tfrac{g}{\rho_0}, \\ u_x + w_z &= 0, \text{ and} \\ \rho_t &= N^2 w, \end{aligned} \tag{3.18}$$

where the spanwise (y) derivative has been dropped for this two-dimensional case.

Internal waves have the functional form:

$$\begin{aligned} (u, v, p) &\propto \cos\left(\frac{m\pi}{H} z\right) e^{\mathbf{i}(kx - \omega t)} \text{ and} \\ (w, \rho) &\propto \sin\left(\frac{m\pi}{H} z\right) e^{\mathbf{i}(kx - \omega t)}, \end{aligned} \tag{3.19}$$

where H is the domain depth (5km), m is the vertical mode number, k is the horizontal spatial frequency, and ω is the temporal frequency. Substituting this into (3.18) gives the dispersion relation:

$$\omega^2 = \frac{N^2 k^2 H^2 + f^2 m^2 \pi^2}{m^2 \pi^2 + k^2 H^2}, \tag{3.20}$$

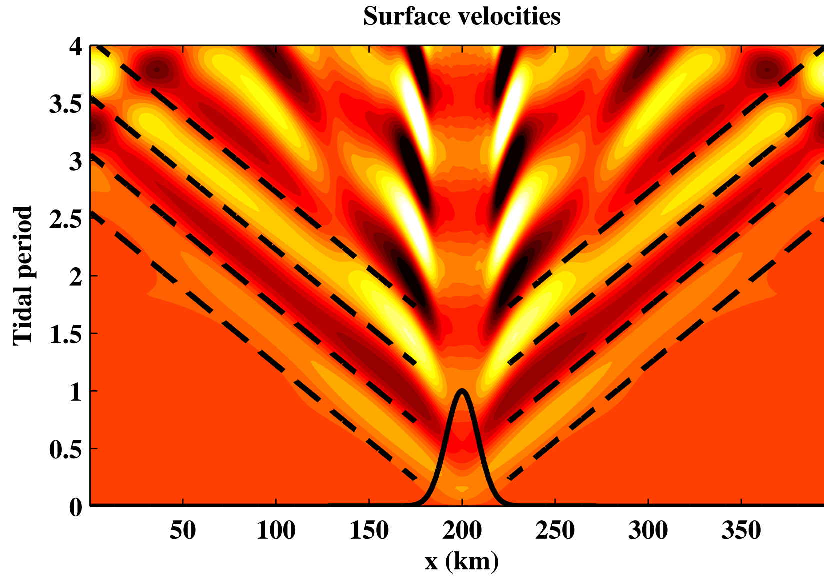

Figure 3.10: Space-time plot of surface velocities (mean removed) for the 2048×64 case. The wave crests are clearly visible, and they move at the phase speed predicted by linear analysis (dashed lines). After 3 tidal periods, waves that exit on one side of the domain re-enter on the other and begin interacting.

which implies (since ω is fixed at the tidal frequency):

$$k^2 = \frac{m^2\pi^2}{H^2}\frac{\omega^2 - f^2}{N^2 - \omega^2}. \tag{3.21}$$

The values for this section give $k = 8.3310^{-5}\,\mathrm{m}^{-1}$ (wavelength 75.39km), and consequently a phase speed of $\omega_{\text{tide}}k^{-1} = 1.69\,\mathrm{m\,s}^{-1}$.

The surface velocity profiles are plotted for the duration of the simulation in figure 3.10. The mode one waves are generated and radiate away from the topography (sketched in the figure to show its location) soon after the beginning of the simulation. The wave crests propagate outward in both directions, and their speed of propagation matches the calculated linear phase speed. The maximum wave-induced velocity was less than $2\,\frac{\mathrm{cm}}{\mathrm{s}}$ and the maximum isopycnal displacement was less than $20\,\mathrm{m}$, both of which confirm that the produced waves are well-described by linear theory.

Chapter 4

Research and Future Work

The code documented in Chapter 3 has been used to simulate larger-scale problems. The results presented here showcase the capabilities of this code with problems that are of physical interest, rather than problems that are specially crafted as numerical tests.

Each of the results in this chapter involves some degree of three-dimensional simulation, both with (4.1 and 4.3) and without (4.2) a viscous boundary layer. Reasonable accuracy in each case required large grids, and these problems were only possible because the code runs with acceptable parallel performance.

4.1 Three dimensional dipole/wall interaction

The example code in section 3.4 reproduce the two-dimensional dipole-wall collision of Clercx and Bruneau [2006], but a three-dimensional case is not equivalent. In three dimensions, a well-separated vortex pair undergoes an elliptic instability [Leweke and Williamson, 1998] which breaks the two-dimensional structure of the dipole and can lead to turbulence [Laporte and Corjon, 2000] through vortex-tube stretching, a mechanism with requires three-dimensional motion.

The dipole setup of section 3.4 is complicated by the close separation between the vortex cores and by the interaction with the boundary. The collision changes the dynamics of the translating dipole pair at a finite time, so initial instabilities have only a short time to grow before the interaction generates more complicated dynamics. Experimental observations [Cieslik et al., 2009] have shown the persistence of three-dimensional motion during and after the collision.

In three-dimensions, the setup of section 3.4 is repeated as a periodic channel (matching Kramer et al. [2007]), with a spanwise length of 0.4 units, twice the initial separation distance between the vortex cores. The total grid size was $768 \times 192 \times 769$ points, enough for two-

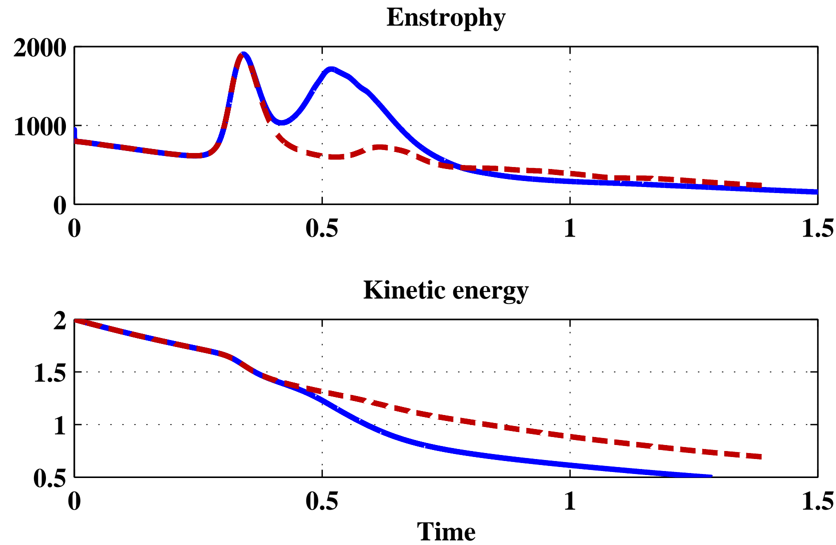

Figure 4.1: Enstrophy (top) and kinetic energy (bottom) of the three-dimensional dipole/wall interaction (solid line), compared with the two-dimensional equivalent (dashed). The three-dimensional interaction undergoes a second production of enstrophy (from three-dimensional effects) after the primary dipole-wall interaction. This is associated with an increase in energy dissipation.

dimensional convergence.[1] In order to trigger three-dimensional effects, random white noise[2] of standard deviation 10^{-2} was added to each velocity component at initialization.

For early times, the three-dimensional dipole behaved like its two-dimensional equivalent, following the same path and interacting with the boundary in the same manner. The enstrophy (figure 4.1) peaked with that collision, and the dipole rebounded from the wall in much the same manner. After the collision, however, three-dimensional effects became significant. With three-dimensional motion, vortex tube stretching resulted in additional production of enstrophy, which is conserved in inviscid two-dimensional motion. Additionally, the three-dimensional effects transfer energy to shorter scales, where viscous dissipation extracts more energy from the flow compared to the two-dimensional dipole.

Just after the second enstrophy peak, the vortex cores are nearly unrecognizable as two dimensional objects (see figure 4.2. Although the boundary-generated vorticity is still largely two-dimensional, the dipole itself undergoes significant three-dimensional oscillation. Additionally,

[1]The results of this section have been presented by the author at the November, 2009 APS Division of Fluid Dynamics meeting.

[2]The noise perturbation was neither incompressible nor satisfying of the boundary conditions, but the pressure projection made it so after the first time step. The resulting effect on the simulation was negligible.

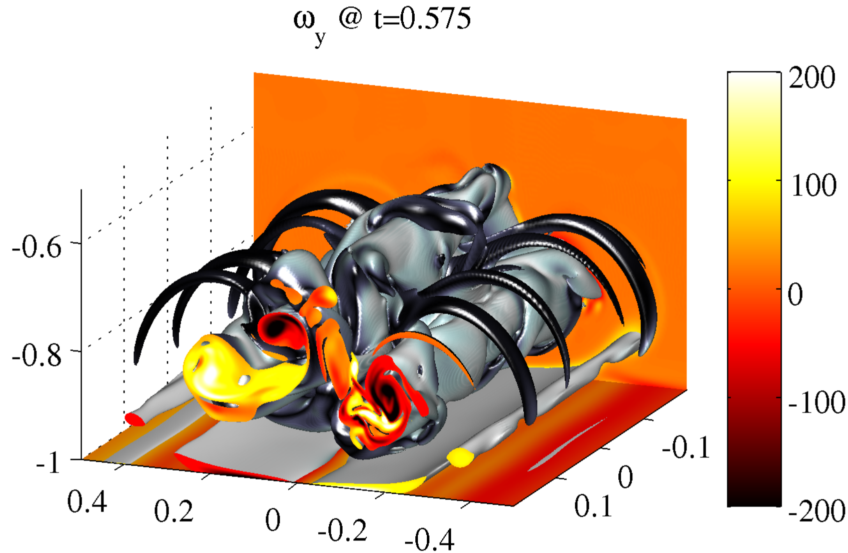

Figure 4.2: Isosurface plot of vorticity for the three-dimensional dipole-wall interaction, with the surface at $|\omega| = 50$ shaded by the orientation of vorticity, with lighter being more spanwise-directed. The slices at the front and back planes plot ω_y, vorticity in the direction of the original axis.

peripheral vortex rings form that are purely three-dimensional.

Further work is necessary to clarify the exact mechanism of instability, since the basic two-dimensional flow is highly active. The elliptic instability of the vortex pair [Leweke and Williamson, 1998] is undoubtedly responsible for some of the three-dimensional motion, but the boundary interaction introduces a wide range of scales in the flow and its behaviour post-interaction is no longer accurately approximated by a single dipole, even in two dimensions. Additionally, at Reynolds numbers above 2500 [Kramer et al., 2007], the two-dimensional interaction produces secondary instabilities in the boundary, and it is unclear how these structures would evolve in three dimensions.

4.2 Internal waves with shear instabilities

A second large-scale problem simulated is the effect of shear instabilities in internal near-solitary waves. Waves of this type were observed off of the Oregon Shelf, and analysis [Lamb and Farmer, 2011] shows that large waves can have regions of shear instability in the pycnocline, resulting in the formation of Kelvin-Helmholtz billows at the back of the wave that slowly extract energy.

Wave	z_0 (m)	Δz_i (m)	Δz_f (m)	min(Ri)	ω (s^{-1})	k_i (m^{-1})	Length (m)	Amplitude (m)
1a	−10	10	8	0.095	0.165	0.045	159.2	27.54
1b	−10	10	5	0.065	0.240	0.080	157.6	27.07
2	−10	15	10	0.089	0.155	0.041	201.3	33.56
3	−5	10	8	0.090	0.160	0.049	116.5	27.10
4a	−15	15	10	0.099	0.155	0.037	273.0	31.94
4b	−15	15	10	0.103	0.154	0.036	255.1	31.03
4c	−15	15	10	0.108	0.150	0.034	240.9	30.48

Figure 4.3: Table of initial wave conditions. Waves 1, 2, and 3 are tall, relatively thin waves with crests that are well above the mid-depth of -50 m, while waves 4a-4c are broad waves with crests that nearly reach the mid-depth. min(Ri) is the minimum Richardson number in the pycnocline at the centre of the wave, and ω is the frequency of the oscillation with fastest spatial growth rate (k_i).

Adaptive simulations of large internal waves generated by the steepening and dispersion of a density field with a depression were conducted in Barad and Fringer [2010] and concluded that the large-amplitude internal waves required low Richardson numbers (below 0.1) for the onset of Kelvin-Helmholtz billow generation.

To further study this phenomenon, several initial waves[3] were considered, with parameters listed in table 4.3 and plotted in 4.4. The waves were generated by taking an initial two-layer stratification of the form:

$$\overline{\rho}(z) = 1 - 10^{-3}\tanh(k\tfrac{z-z_0}{\Delta z}) \tag{4.1}$$

where $-100\,\text{m} < z < 0$ is the vertical coordinate of the domain, z_0 is the location of the centre of the pycnocline, and $k = 2\tanh^{-1}(0.99)$ such that Δz is the length scale over which 99% of the density transition occurs.

The basic stratification of (4.1) was used as the boundary condition for a numerical solution of the Dubreil-Jacotin-Long equation [Lamb, 2002], which computes solitary waves as a nonlinear eigenvalue problem in terms of isopycnal displacement η:

$$\nabla^2\eta + \frac{N^2(z-\eta)}{c^2}\eta = 0, \tag{4.2}$$

where c is the resulting wave phase speed. The equation is solved iteratively with a continuation method to increase wave energy. Finally, in order to increase the shear at the pycnocline, the computed wave is adjusted in a time-dependent finite-volume simulation using the (second-order

[3]The waves for these simulations were provided by Dr. Kevin Lamb of the University of Waterloo, and the computer time to run these simulations was given as a Dedicated Resources Grant by SHARCNET. Some of the results in this section were previously presented at the 2010 SHARCNET Research Day at York University.

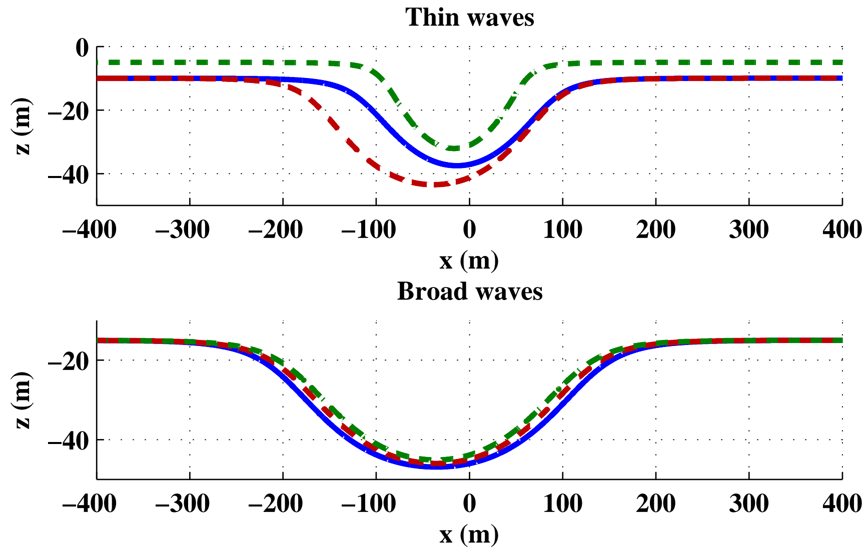

Figure 4.4: Visualization of the pycnocline centres of the waves from table 4.3. Top: waves 1a (solid), 2 (dashed), and 3 (dot-dashed). Bottom: waves 4a (solid), 4b (dashed), and 4c (dot-dashed).

in space) model of Lamb [1994] by propagation into a region with a sharper pycnocline (lower Δz). The time-dependent adjustment allowed the generation of waves with extremely strong velocity shear in the pycnocline that were difficult to solve for directly.

To test the formation of Kelvin-Helmholtz billows from the resulting adjusted waves, the waves were simulated individually on a $1000\,\mathrm{m} \times 100\,\mathrm{m}$ domain with a 6400×640 grid, in a reference frame moving with the waves' phase speeds. Upstream of the wave, the vertical momentum equation was forced in a region around the pycnocline with the frequency ω given in table 4.3. As the disturbance propagates through the wave, the unstable region amplifies the oscillations, producing Kelvin-Helmholtz billows which may overturn. In the wave-local reference frame, these billows propagate out the tail of the wave, carrying wave energy with them, where they are damped via a numerical sponge layer[4], which returns the flow variables to the upstream values.

The energy extracted over time from each wave is plotted in figure 4.5. Although all of the waves in table 4.3 have central regions with very low Richardson numbers, only wave 1b (top panel of 4.5) and wave 4a (bottom panel) develop billows that overturn and extract significant amounts of energy from the wave. This suggests two mechanisms of action for energy loss:

[4]Although sponge layers often need careful crafting to avoid significant wave reflection, this is mitigated here because the reference velocity is supercritical – above the linear long wave speed. Small-amplitude waves that may be generated by the sponge layer cannot propagate back upstream.

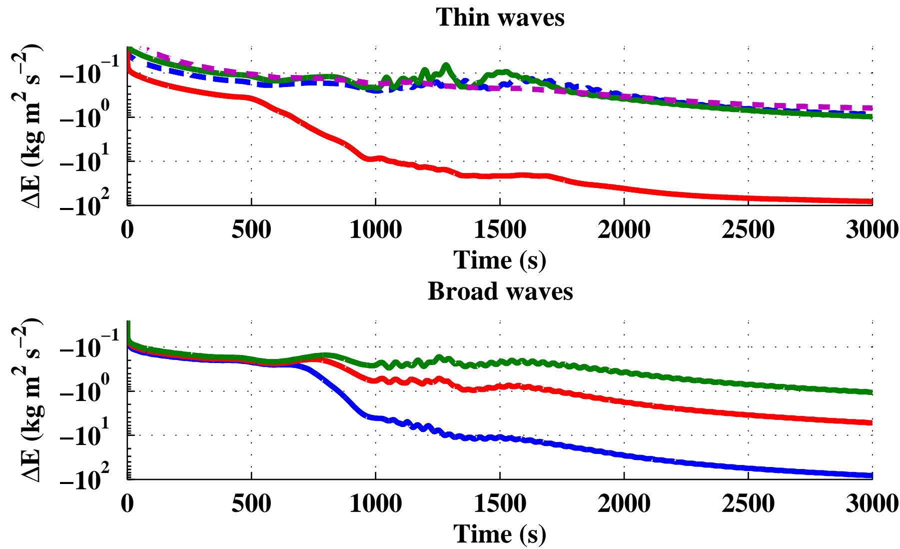

Figure 4.5: Energy extraction over time (log scale) for (top) waves 1a (long dashes), 1b (bottom solid), 2 (top solid), and 3 (short dashes) and (bottom) waves 4a, 4b, and 4c (from bottom to top)

either the wave must have a very strong instability (wave 1b, with minimum Richardson number of 0.065), or the wave must be very long (wave 4a, with length 273m), giving the billows a long region to extract energy.

Overturning billows also undergo secondary, three-dimensional instabilities [Smyth et al., 2005] after growing to finite size. To test whether this is a significant effect in the case of shear-unstable waves, the wave 4a was simulated in three dimensions by extending the grid to a $1000\,\mathrm{m} \times 48\,\mathrm{m} \times 100\,\mathrm{m}$ grid with 50 points in the spanwise direction, chosen to have a spanwise length of approximately twice the billow-separation distance with an isotropic grid. In order to trigger three-dimensional development, the amplitude of the forcing function was modulated at 10% magnitude by the first two periodic modes $(\sin(\frac{2\pi}{50}y) + \sin(\frac{4\pi}{50}y))$.

The produced billows did three-dimensionalize (see figures 4.6 and 4.7). However, the three-dimensional development happened at the tail of the wave, after the billows had already exited the shear-unstable region. Consequently, the billows were unable to extract additional energy from the wave, and the energy loss rates in the two and three dimensional cases are nearly identical. The three-dimensional evolution at the tail of the wave is still significant, however, and the ejected billows are not well-described by strictly two-dimensional profiles.

Further work here will concentrate on two paths. First, the simulations were conducted without viscosity, and at too-coarse of a resolution to resolve physical dissipation scales anyway.

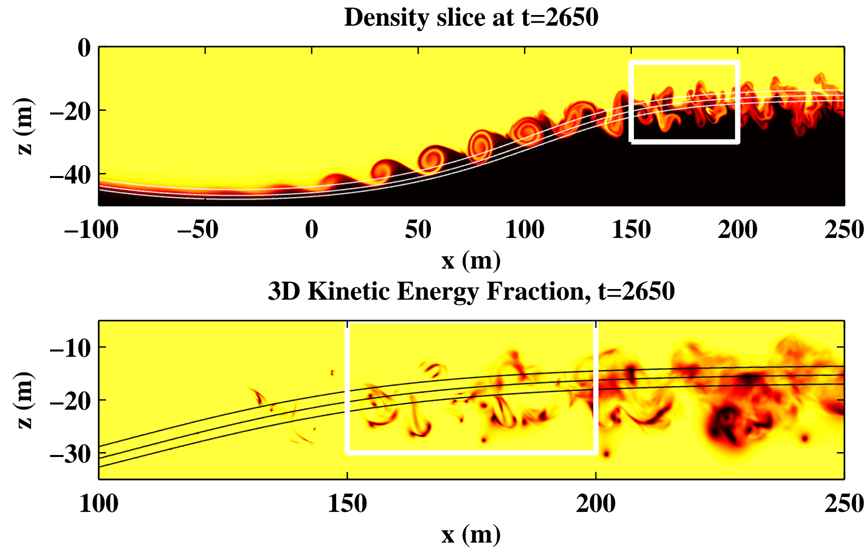

Figure 4.6: Density profile (top) of the three-dimensional simulation of wave 4a in the x-z plane at time 2650 s, along with the three-dimensional kinetic energy fraction (bottom) $\frac{u^2+v^2+w^2}{\langle u \rangle_y^2 + \langle w \rangle_y^2}$ (darker is more three-dimensional). The boxed region is visualized in figure 4.7, and the contours visualize the wave pycnocline at $t = 0$, before the production of any billows.

Even if three-dimensional motion does not extract additional energy from the wave, the motion may have implications for how effectively the produced billows mix the pycnocline as they decay. Second, the difference between cases 4a and 4b was stark, even though the underlying waves were not much different in length and minimum Richardson number. Still broader waves may have a long-enough unstable region for the secondary instabilities of the billows to extract additional energy.

4.3 Boundary layer instability under internal waves

Internal waves can produce strong currents at the top and bottom of the domain. In viscous simulations, these currents produce boundary layers at no-slip boundaries (generally the bottom of the domain), creating strong shear. For particularly large internal waves, this shear has the potential to become unstable [Carr and Davies, 2010; Diamessis and Redekopp, 2006]; this section focuses on the less-intense case of smaller-scale waves left to propagate for long times[5], in or-

[5]This work was done in partnership with Dr. Marek Stastna.

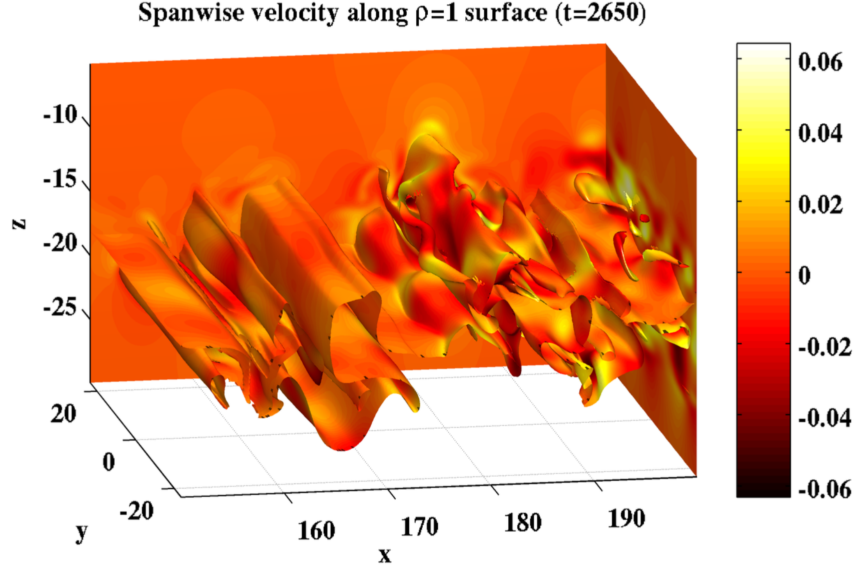

Figure 4.7: Three-dimensional view of the simulation of wave 4a, showing the $\rho = 1$ isosurface shaded by the spanwise velocity in the tail of the wave. The three-dimensional evolution of the billows is first characterized by production of spanwise velocity, which then affects the three-dimensional shape of density isosurfaces. The background velocity is in the positive x direction, so the rightmost billows have had the longest time to evolve.

der to study the development of instabilities that result from repeated interactions between wave currents and the boundary layer.

The domain used is approximately lab-scale, 200 meters long ($0 < x < 200$) by 10 meters deep ($0 < z < 10$), discretized by 2048 points in the (periodic) horizontal direction and 256 points in the vertical direction, with no-slip boundaries at the top and bottom of the domain. The background stratification is a smoothed two-layer stratification of the form:

$$\overline{\rho}(z) = -\frac{\Delta\rho}{\rho_0} \tanh\left(\frac{z - z_0 + \eta(x)}{\Delta z}\right), \tag{4.3}$$

where $\frac{\Delta\rho}{\rho_0}$ is 1% (for a two-percent top-to-bottom density difference), z_0 is 3 meters, Δz was 0.5 meters, and $\eta(x)$ was the profile of the initial density disturbance, given by:

$$\eta(x) = a\left(\operatorname{sech}^2\left(\frac{x - x_0}{\Delta x}\right) + \tfrac{1}{2}\operatorname{sech}^2\left(\frac{x - x_1}{\Delta x}\right)\right), \tag{4.4}$$

where $x_0 = 60\,\text{m}$ and $x_1 = 140\,\text{m}$ are the positions of the two first-mode disturbances of elevation and $\Delta x = 20$ is the length scale of those disturbances. The simulations included both viscosity

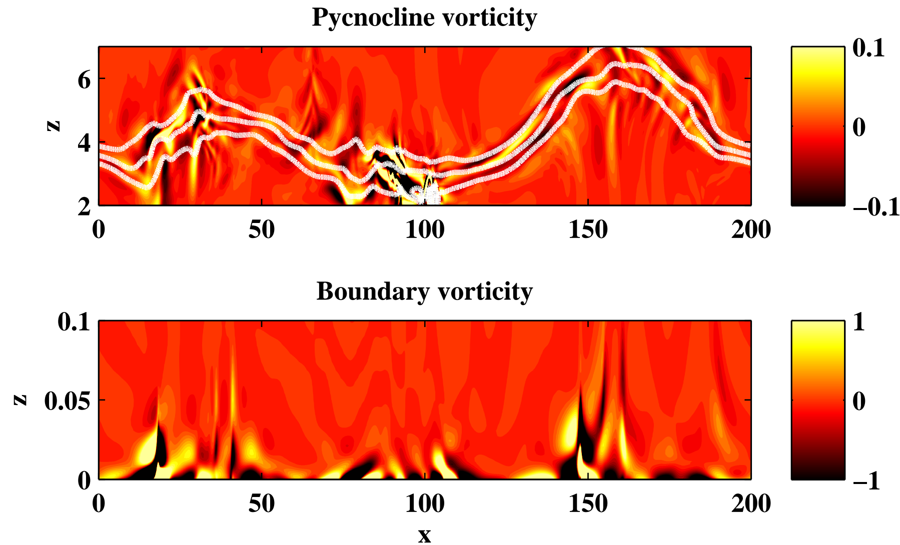

Figure 4.8: Vorticity in the wave field (top, with contours showing the pycnocline) and bottom boundary (bottom) for the large-wave case ($a = 2$ in (4.4)) at $t = 500$, high-pass filtered to remove the large-scale vorticity caused by the wave motion. The short-scale motion in the pycnocline is fairly strong with localized regions of overturning, but the short-wavelength vorticity at the boundary is significantly stronger.

and density diffusivity of $\nu = \kappa = 10^{-6}\,\frac{\mathrm{m}^2}{\mathrm{s}}$. The amplitude of the initial density disturbance is parameterized by a, which is varied to test different cases. After initialization, the flow is left to freely evolve without further forcing, and the disturbances split into leftward and rightward-moving internal waves of varying sizes. Head-on collisions produce higher-mode waves and shorter wavelengths, leading to a rich wave-field.

Large waves are generated with $a = 2\,\mathrm{m}$ in (4.4), which produces disturbances strong enough to cause eventual overturning in the pycnocline. The strong wave-induced currents also create strong vortices in the boundary layer (figure 4.8), which develop into billow-type instabilities. These are most visible after filtering the vorticity field with a high-pass filter[6], which removes the large-wavelength vorticity produced by the broadest waves. As the waves propagate in the pycnocline, the back-and-forth motion at the bottom boundary produces pancake-like layers of opposite-signed vorticity (figure 4.9), which organize into billow-type structures. These structures are not related to the overturning in the pycnocline.

[6]The vorticity was filtered in the x-spectral domain with a filter of the type $(1 - \exp(-\frac{\kappa^2}{\kappa_c^2}))$, for a filtering scale κ_c chosen to have wavelength of 15% of the domain length. The visualized high-frequency vorticity was not particularly sensitive to the choice of filtering scale.

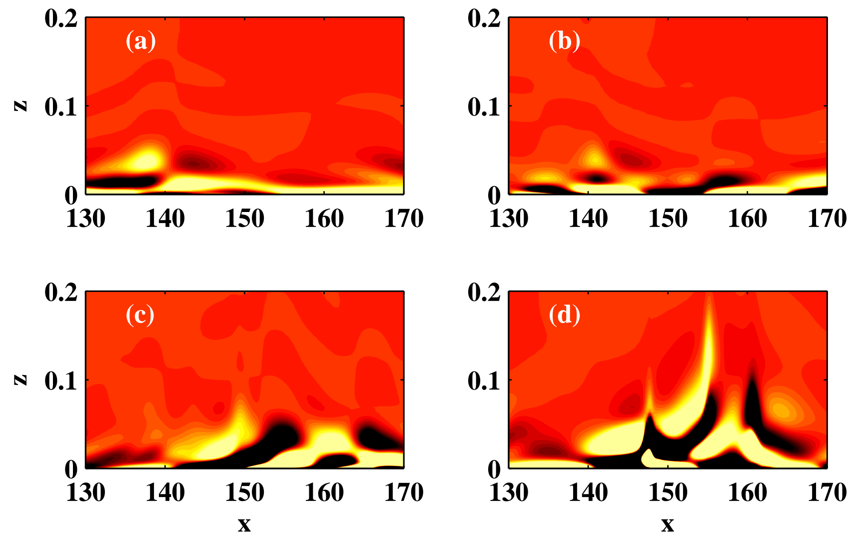

Figure 4.9: Evolution of the boundary vorticity of the case in figure 4.8 in a selected region at $t = 300$ (a), 400 (b), 450 (c), and 500 (d). Layers of opposite-signed vorticity overlie each other, leading to bursts of vorticity away from he boundary.

With a smaller initial disturbance ($a = 1\,\mathrm{m}$ in (4.4)) visualized in figure 4.10, the wave-induced currents are significantly smaller, and nonlinear effects are less prominent in the pycnocline. Additionally, the boundary vorticity does not develop as fully as with the larger waves. The short-scale vorticity near the boundary does not form oppositely-directed layers, and there are no bursts of vorticity away from the boundary.

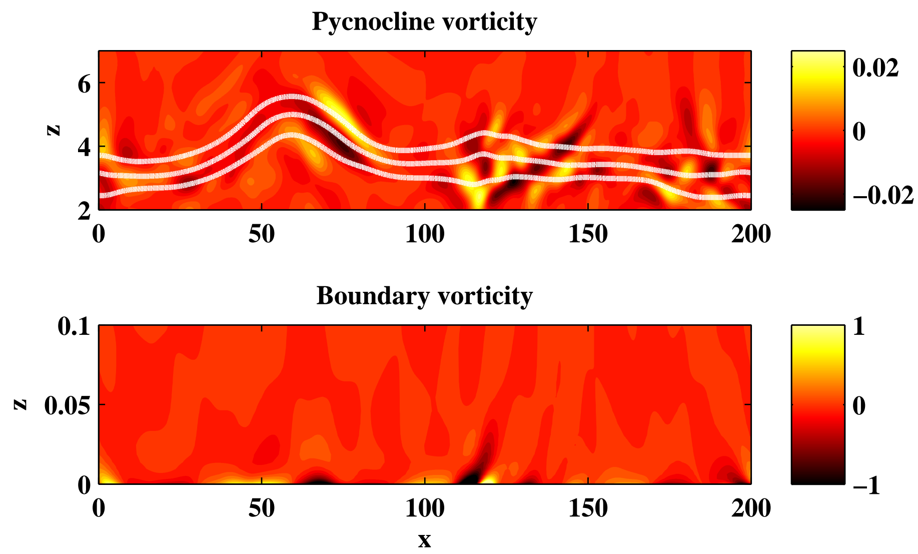

Figure 4.10: As figure 4.8, for waves of half the amplitude ($a = 1$). The smaller waves interact more weakly, and they do not generate shorter waves in the pycnocline. Short-scale vorticity generation at the boundary is suppressed, and does not result in strong billows.

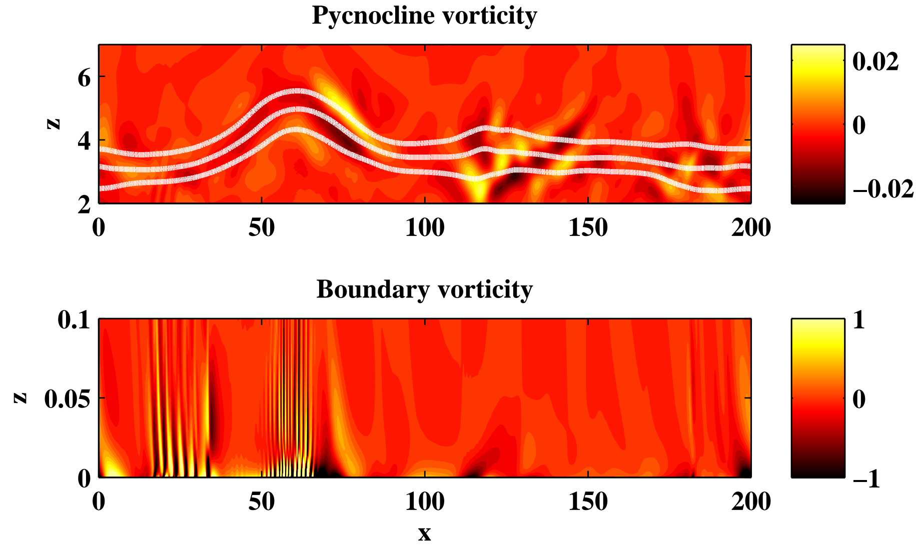

Figure 4.11: As figure 4.10, with the addition of a background current ($u_0 = -0.2$) at the beginning of the simulation. This quickly adjusts to the no-slip boundaries and forms a thin boundary layer, which provides a source of vorticity for billow-type instabilities at the boundary.

4.3.1 Background shear

Larger waves are not necessary for the development of billow-type structures in the bottom boundary layer, however. For large internal waves, adding a background shear current can induce boundary-layer instability and sediment resuspension [Stastna and Lamb, 2002]. Adding a background current ($u_0 = -0.2\,\frac{\mathrm{m}}{\mathrm{s}}$, approximately one-third of the linear long-wave speed relative to the background stratification) to the small-amplitude case ($a = 1\,\mathrm{m}$) gives the case visualized in figure 4.11. Here, the initial background current quickly adjusts to the no-slip boundary conditions and forms a thin boundary layer. The vorticity in this layer provides a background for the development of vigorous billow-type instabilities which reach much further into the water column than even the large-wave case of figure 4.8.

Including an additional, weak stratification at the bottom boundary suppresses but does not eliminate these billows. In addition to the background shear, the case of figure 4.12 includes an additional stratification at initialization of the form:

$$\rho_b = \frac{\Delta\rho_b}{\rho_0}\left(1 - \tanh\left(\frac{z}{\Delta z_b}\right)\right), \tag{4.5}$$

where $\frac{\Delta\rho_b}{\rho_0}$ was 10^{-3} and Δz_b was 0.1, increasing the total (initial) top-to-bottom density difference by 5%, although this was reduced somewhat over time as the bottom stratification adjusted

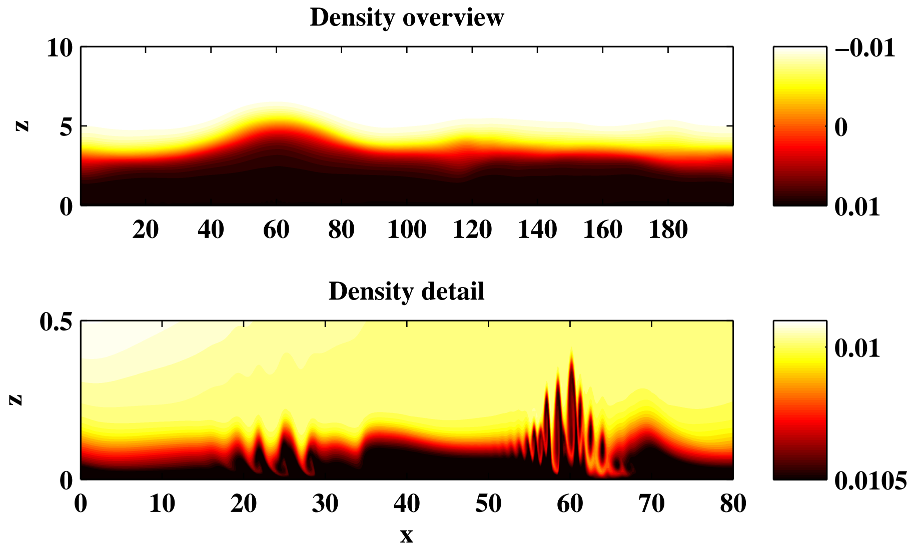

Figure 4.12: The central pycnocline (top) and bottom boundary (bottom) density for a case similar to 4.11 with the addition of a small secondary stratification at the bottom boundary. The background shear still provides a mechanism for instability, but the resulting billows remain lower in the water column.

to the no-flux boundary condition for density. The interaction between the background shear and wave motion in the pycnocline still generated billows at the bottom boundary, but their influence on the water column was reduced somewhat because they occurred in higher-density fluid and could not as easily travel upwards. The billows do result in density overturns at the bottom boundary however, and this most likely has implications for near-boundary mixing.[7]

4.3.2 Three dimensional evolution

The low-amplitude case with shear at the bottom boundary (figure 4.11) can also be studied as a three-dimensional case. Extending the domain in the spanwise by 5 meters ($0 < y < 5$), half the domain height and using 96 points in that direction gives a well-resolved grid. The initial conditions match that case (density as in (4.3) and (4.4) with $a = 1$) with the addition of u, v, and w set pointwise to white noise with standard deviation 10^{-4}. This low noise does not affect the large-scale development of the flow, but it suffices to trigger three-dimensional instabilities without biasing towards pre-selected modes.

[7] Further analysis of mixing in the bottom boundary layer would require study of the available potential energy (section 4.4.7), and the code for computing this is not yet developed.

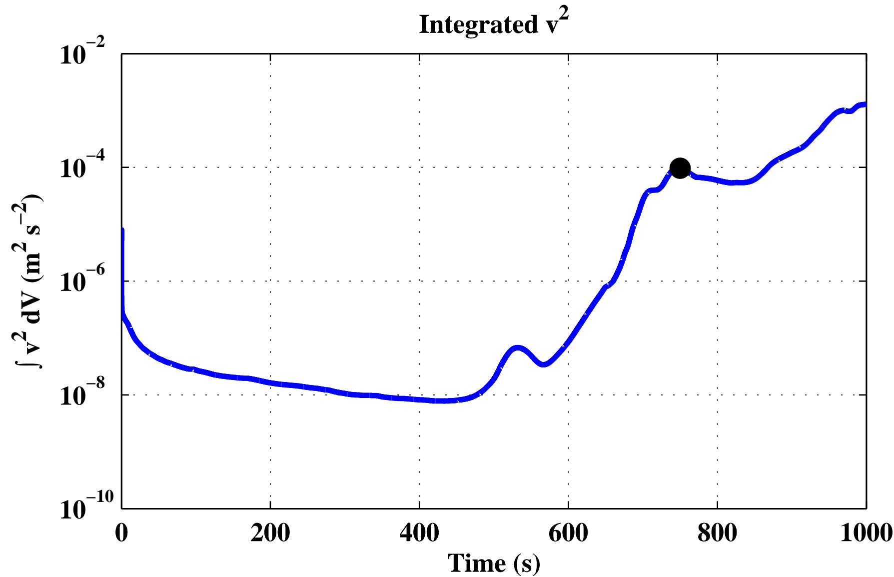

Figure 4.13: $\int v(t)^2 \mathrm{d}V$ for the three-dimensional wavefield-boundary interaction with background shear. The integrated v^2 initially decays from the white-noise perturbed initial conditions, but as the flow evolves the billows at the bottom boundary undergo three-dimensional undulations, causing an increase in v^2. The dot is placed at $t = 750$, which is pictured in figure 4.14.

Initially, the evolution of the flow is governed by the two-dimensional process. Three-dimensional effects, analyzed with the proxy term $\int v(t)^2 \mathrm{d}V$ and plotted in figure 4.13 do not become significant until after $t = 600$ seconds into the run. After that time, however, the vorticity in the bottom boundary layer begins three-dimensional modulation (figure 4.14). Three dimensional effects are initially unimportant in the pycnocline, but instead they are concentrated in a small region at the bottom boundary.

Looking at this feature in detail (figure 4.15) through the bottom stresses $\frac{\partial u}{\partial z}$ and $\frac{\partial v}{\partial z}$ shows the form of the instability. The streamwise velocity organizes into high and low-speed streaks, and in regions of high spanwise velocity these streaks become more chaotic.

Further work will focus on analysis of the three-dimensional form of these instabilities, including how their eventual development may affect the wave motion in the pycnocline. The modification of motion in the bottom boundary layer will also impact any mixing, including sediment transport.

Additionally, the parameters for these cases were chosen somewhat arbitrarily, and there is a great deal of freedom in selecting them. The difference between figures 4.8, 4.10, and 4.11 is stark, suggesting that there may be a threshold for the onset of instability. Additionally, at

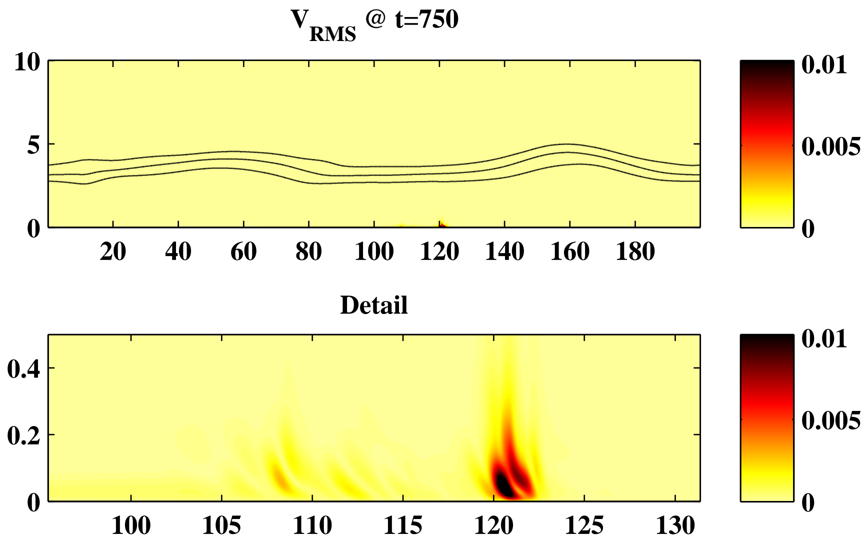

Figure 4.14: Root mean square spanwise velocity $(\langle v^2 \rangle_y)^{1/2}$ of the three-dimensional interaction at time $t = 750$, a local maximum of $\int v^2 \mathrm{d}V$. At top, the full field (x-z plane) with contours visualizing the pycnocline. The spanwise velocity is localized to a small region of the bottom, viewed in detail in the bottom panel.

ocean scales a constant background flow is less realistic than an oscillating, tidal flow; this may produce qualitatively different behaviour.

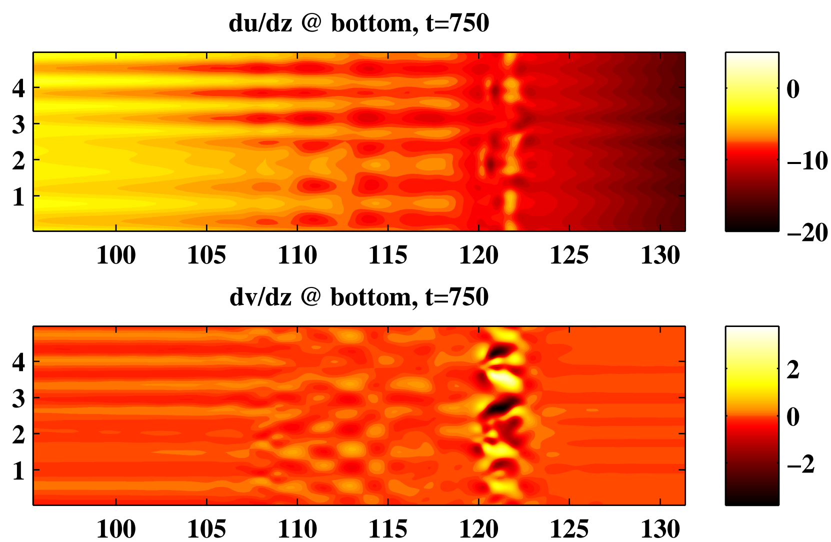

Figure 4.15: Bottom stress $\frac{\partial u}{\partial z}$ (top) and $\frac{\partial v}{\partial z}$ (bottom) for the region shown in detail in figure 4.14 at time $t = 750$. $\frac{\partial u}{\partial z}$ is dominated by the background and wave-induced currents, but still has significant modulation from the three-dimensional character of the instability. $\frac{\partial v}{\partial z}$ is controlled entirely by three-dimensionalization.

4.4 Future work

The numerical code that is the focus of this work is already applicable to a wide range of cases, but with further work it can be extended for greater generality and improved performance. In some cases, especially for topography extensions (section 4.4.1) the modifications needed may be extensive, but the same basic algorithms developed in Chapter 2 will still apply.

4.4.1 Two-dimensional topography and the no-slip cube

The largest and most obvious extension for this code is to allow full, two-dimensional boundary mapping. Currently, the boundaries of the domain can only be mapped in one dimension, in the xz plane. They must be constant in the spanwise (y) direction, because the code relies on separating the flow variables with a Fourier (trigonometric) expansion in y. This is not a mathematical requirement for the algorithm, but it allows the coding simplification of specializing the multigrid algorithm of section 2.4.3 to two-dimensional grids like that in figure 2.22.

Allowing grid-mapping along the y direction breaks this assumption. Worse, even without such a mapping simply allowing the dimension to be expanded in terms of Chebyshev polynomials also breaks the assumption. In both cases, the problem arises because the resulting expansion (mapped trigonometric functions for the former case, polynomials for the latter) are no longer eigenfunctions of Laplace's equation in the domain.

Making this extension would not require much change to the forward operators. Computing gradients and divergence would remain essentially unchanged from the mapping method in (2.72) and (2.76), with additional terms reflecting any mapping along the y dimension. The multigrid method, however, would need to be replaced wholesale.

The underlying algorithm used in this work, that of line relaxation and x-semicoarsening, can effectively be extended. This would give plane relaxation with x-semicoarsening, where subproblems on entire yz planes are solved for at each relaxation step. In general, these yz planes would still have anisotropy, and they would have to be solved recursively with a two-dimensional multigrid method like that of section 2.4.3. For a grid of size $N_x \times N_y \times N_z$ points, each plane relaxation would take $O(N_y N_z)$ operations, giving a total asymptotic workload of $O(N_x N_y N_z)$ operations (in serial) by the scaling arguments in section 2.4.3.

It may instead be advisable to approach the three-dimensional problem in its entirety, with a conditional-semicoarsening method like that in Larsson et al. [2005]. Instead of using line (in two dimensions) or plane (in three dimensions) relaxation, this method uses simpler pointwise relaxation but removes only a subset of grid lines (planes) at each recursive step. As a result, the coarser grids will retain more than $\frac{1}{2}$ of the finer grid's points in each dimension.

The semicoarsening method reduces the grid size by a factor of only two at each level both for two and three dimensions; this net reduction would be matched by a three-dimensional con-

ditional coarsening method if the number of points was reduced by approximately 20% per dimension per level.

4.4.2 General boundary conditions

A problem somewhat easier to correct with the code is allowing for suitably general boundary conditions. Currently, the implemented boundary conditions are useful but inflexible. Dimensions expanded in Chebyshev polynomials do not have assumed symmetry, but the code is specialized for no-slip boundary conditions (zero velocity) with viscosity and no normal flow without viscosity, where boundary velocity is enforced through the pressure condition. Tracers do not need boundary conditions without diffusivity[8], but with diffusivity the code makes the assumption that there should be no flux through the boundary.

This assumption is generally realistic, but it precludes accurate treatment of heat transfer. Flow between two plates held at separate temperatures, for example, requires a temperature field that takes one fixed value at the bottom boundary and a second fixed value at the top. The simplest expression for temperature in the flow would require satisfying separate, nonzero Dirichlet-type boundary conditions, which is currently not possible with this code.

The underlying algorithms, however, are flexible enough to incorporate general boundary conditions. Much of the underlying code, in fact, was written with the assumption of general, Robin-type ($u + \hat{\mathbf{n}} \cdot \nabla u = f$ on δD) boundary conditions. The primary difficulty in translating this to the case-specific code is in developing a clean, simple interface with the case-specific code. The case-specific code has so far operated on the full two or three-dimensional fields, whereas the boundaries are a dimension reduced.

Infinite domains and outflow conditions

A more complicated instance of general boundary conditions is that of infinite domains. Domains like that of section 4.2 are fairly typical, where the region is effectively infinite in extent but only activity within a relatively narrow region is of interest.

Direct implementation of outflow boundary conditions (see Tsynkov [1998] for a review) is problematic, both numerically and physically. Numerically, general boundary conditions don't have convenient symmetries, so the infinite direction (most likely x) would have to be expanded with Chebyshev polynomials. This approach concentrates the grid points near the boundaries, *away* from the region of interest. Physically, these outflow boundary conditions must account for the full spectrum of waves allowed by stratification [Jensen, 1998], which makes simple, efficient implementation difficult.

[8] Without diffusivity, the evolution equations for tracers are wholly explicit. With this set of velocity conditions, there is no inflow from outside the domain, so there is no need to set upwind boundary conditions on tracers.

A second approach is to use sponge layer methods, relaxing flow parameters to prescribed far-field conditions over a finite region. This is the approach used in section 4.2, and there it works extremely well because the background flow is faster than the linear long-wave speed – small amplitude disturbances created at the sponge layer cannot propagate upstream into the middle of the domain. Slower (or oscillating) background flows would require more care in design of a sponge layer.

This approach still allows simple trigonometric expansions in the spanwise direction; section 4.2 uses the periodic, Fourier expansion. Applications with a shelf-type topography, however, cannot use a periodic expansion. The left and right boundaries would have drastically different far-field conditions, and the numerical shock caused by identifying the boundaries with each other would cause severe Gibbs-type oscillations. Instead, assuming unaccelerated flow in the far-field would give a natural cosine-based expansion for both velocities and pressure.

Ordinarily, assuming a cosine basis for all of these variables would be problematic. After all, $\frac{\partial p}{\partial x}$ would be best expressed with a sine basis. The key source of this problem is the boundary, where a cosine expansion has zero derivative at the boundary, whereas a sine expansion (the derivative of pressure) would possibly have a nonzero derivative. This is fortunately not a problem with outflow boundaries because a sponge layer will damp both the flow and its derivatives exponentially.

A third possible approach to outflow boundaries would be to directly discretize the infinite domain by applying a singular mapping to the grid, as described in Boyd [2001]. Using the mapping $x = L\tan(\theta)$ lets θ, discretized on $-\frac{\pi}{2}$ to $\frac{\pi}{2}$, substitute for the full, infinite domain in x, with a scale factor of L. This approach is perhaps the most elegant solution, but it has several possible flaws:

- Firstly, infinite domains reduce exponential convergence of spectral methods to sub-exponential, with the logarithm of the error proportional to $-N^{1/2}$. Physically, this arises because increasing the number of points splits the improvement between better-resolving the central region and better-resolving the far-field behaviour.

- Secondly, the derivative of the mapping is infinite at the endpoints. This work currently takes the derivative of the mapping numerically (in the computational coordinates) to find the coefficients for (2.72), and the singular transformation would generate inaccurate values. Using an infinite mapping would require analytically specifying these derivatives, which is not currently supported.

- Thirdly, an infinite grid still does not eliminate the need for damping or sponge layers. For example, a solitary wave propagating downstream will propagate infinitely, without changing form, while the grid spacing becomes progressively coarser. At some finite distance from the central region, the wave would no longer be well-resolved and would cause oscil-

lations which could contaminate the rest of the domain. Filtering of the type used in this work would not by itself resolve matters, since the filter kernels are not strictly positive.

4.4.3 Lagrangian particles

One limitation of the Eulerian tracers implemented in this code is that it is impossible to trace the identity of a specific fluid parcel over time. A tracer may be able to show, for example, that fluid from the boundary is advected into the middle of the domain, but the exact paths of fluid parcels are unrecoverable.

An alternative approach is to implement tracer particles that are advected with the fluid velocity. That is, for a particle with current location (x_j, z_j) (in two dimensions):

$$\begin{aligned} \frac{\partial x_j}{\partial t} &= u(x_j, z_j, t) \\ \frac{\partial z_j}{\partial t} &= w(x_j, z_j, t). \end{aligned} \tag{4.6}$$

For consistency, these ordinary differential equations could be integrated with the same explicit timestepping scheme as is used for the advective terms. From an analytical standpoint, this is not a difficult problem, only requiring correction if a particle is advected through the boundary.[9]

In implementation, there are two significant issues. The first is that the velocity lookups involved in (4.6) will not align with the grid. Direct, off-grid summation of the spectral expansion requires $O(N_x N_z)$ operations for a $N_x \times N_z$ grid, a prohibitive cost for any more than a handful of particles. Low-order interpolation is much simpler with a fixed computational cost per particle (independent of grid size), but then the computed particle paths would have discretization errors much larger than the errors in the velocity fields. This may be acceptable, however, if the particles are integrated only for short times or if the particle paths are used for a qualitative analysis.

A compromise is to use a specialized off-grid method such as that developed in Boyd [1992]. As a preprocessing step, this method extends the field to a uniform grid of three times as many points, by padding the spectral representation with zeros. Then, the grid points in the neighbourhood of the interpolation location are used for interpolation, either directly by a moderate-order Lagrange interpolation polynomial or indirectly by applying the Euler sum acceleration to the cardinal-function contributions of the nearest points[10]. With either approach, the preprocessing step takes $O(N_x N_z \log(N_x N_z))$ operations, and then each off-grid point uses a wide but local

[9]This should not happen often with temporally-smooth velocities. The velocity-based CFL condition on the maximum timestep ensures that the product of the timestep with local velocity is less than one grid cell in length.

[10]This sum acceleration procedure notes that $f(x_j) = \sum_{k=0}^{N} f(x_k) C(x_j - x_k)$, for x_k as each grid point, x_j as the off-grid point, and $C(x)$ as the cardinal function. This sum is slowly converging to its true value, but re-ordering it so that the points nearest x_j are evaluated first allows the Euler sum acceleration formula to greatly improve the convergence.

neighbourhood. This compromise would be most effective for moderate numbers of particles (much greater than $\log(N_x N_z)$), where the per-particle workload would be overwhelming with direct summation.

The second issue of implementation is parallelism. Thus far with this code, the grid itself has been split among processors. Individual particles, however, do not respect arbitrary divisions on the grid, and indeed in general can move anywhere within the computational domain. With local-neighbourhood interpolation for the fluid velocity at a particle's location, the neighbourhood may split between two processors. Interpolation would then require additional communication of boundary points between processors.

4.4.4 Sediment tracers

So far in this work, tracers have been passively advected by the fluid, without independent dynamics. This is sensible for tracers that are properties of the fluid such as heat, are in solution like salt, or are abstract ideas like "fluid that was initially near the boundary."

This assumption breaks down when the tracer represents a trait such as sediment load, simulations of which [Kneller and Buckee, 2000] are important in quantifying mixing of nutrients and pollutants into water column from the bottom boundary. Sediments have comparatively large particle sizes, and their motion is not governed solely by passive advection. Individual particles have a complex interaction with background flows [Burton and Eaton, 2005], but a reasonable approximation is to assume that at first order they settle out of the water column with a velocity that depends on particle size [Davis and Acrivos, 1985], with more complicated interactions for finite sediment loads.

This settling velocity is currently not allowed in this work, which assumes that each tracer's advective motion is identical to that of the fluid parcels underneath. A simple modification to account for a settling velocity would be to have a complicated `forcing` function, modified to take the gradient of the tracer and apply a $(\vec{u}_s \cdot \nabla S)$ term to the sediment forcing function (where $\vec{u}_s$ is the local settling velocity and S is the sediment concentration). A more elegant solution would modify the Navier-Stokes integration to allow for explicitly-specified background velocities.

Neither approach would have a significance performance impact.

4.4.5 Anelastic equations

So far, this model has been developed with the implicit assumption of water as the underlying fluid. Water is effectively incompressible, but this assumption does not hold for the atmosphere, where the compressibility of atmospheric density with depth (hydrostatic pressure) is significant.

Including this effect while still excluding sound waves gives the anelastic equations, which in two dimensions are (Ogura and Phillips [1962], with form from Fulton [1993]):

$$\frac{\partial u}{\partial t} + \vec{u}\cdot\nabla u = -\bar{\rho}^{-1}\frac{\partial p}{\partial x} + \nu\nabla^2 u \tag{4.7a}$$

$$\frac{\partial w}{\partial t} + \vec{u}\cdot\nabla w = -\bar{\rho}^{-1}\frac{\partial p}{\partial z} + \nu\nabla^2 w + \frac{g}{\bar{\theta}}\left(\theta - \bar{\theta}\right) \tag{4.7b}$$

$$\frac{\partial \theta}{\partial t} + \vec{u}\cdot\nabla\theta = \nu\nabla^2\theta \tag{4.7c}$$

$$\frac{\partial \bar{\rho} u}{\partial x} + \frac{\partial \bar{\rho} w}{\partial z} = 0 \tag{4.7d}$$

where u and w are the horizontal and vertical velocities respectively, θ is the potential temperature ($T\rho_0^{\kappa}(\bar{p}+p)^{-\kappa}$, where κ is the ratio of the gas constant and specific heat at constant pressure), and the bar variables are the vertical background profiles given by:

$$\bar{p}(z) = \rho_0\left(\frac{\bar{T}}{T_0}\right)^{1/\kappa} \tag{4.8a}$$

$$\bar{T}(z) = T_0 - \frac{gz}{c_p} \tag{4.8b}$$

$$\bar{\rho}(z) = \tfrac{\bar{p}}{R_d\bar{T}} \tag{4.8c}$$

$$\bar{\theta}(z) = T_0, \tag{4.8d}$$

where (4.8a) is the background pressure, (4.8b) is the background temperature, (4.8c) is the background density, and (4.8d) is the background potential temperature. T_0 and p_0 are reference temperature and pressure respectively, R_d is the ideal gas constant, and c_p is the specific heat at constant pressure.

These equations have much the same form as the incompressible Navier-Stokes equations, and they can be solved through a very similar projection method (Fulton [1993] does this for a mixed Fourier-Chebyshev expansion on an unmapped grid). The Poisson equation to compute pressure is modified, but it can still be solved in the same manner.

4.4.6 Numerical & IO optimization

The efficient scalings discussed in Chapter 2, while encouraging, are not the complete picture for performance. This code has not yet been fully analyzed for implementation bottlenecks, and speed improvements in the numerical computation are almost certainly possible. Several optimizations are obvious, although possibly cumbersome:

- The Blitz++ array library allows writing very simple array expressions, but they are not necessarily fully optimized at compile-time. Most notably, the general, runtime memory allocation used by the library tends to prohibit compilers from automatically vectorizing otherwise eligible loops [Cummings and Hilscher, 2011]. Hand-optimizing some of the inner loops, particularly in derivative computations, may significantly improve performance.

- The array-transposing required in parallel to take derivatives in the first (x) dimension is not a computational step. With proper code reorganization and the use of nonblocking MPI communication [Message Passing Interface Forum, 2008], it would be possible to compute the other two components of gradient (y and z) while waiting for the transposition.

- Currently, array output is serialized and written to disk for postprocessing (generally with MATLAB). Output is infrequent enough that this does not have a significant performance impact during the simulation, but the format on disk of a single, large array is cumbersome for analysis. Using a dedicated scientific output library like HDF5 [Folk et al., 1999] would allow both direct, parallel output and more convenient analysis for large datasets. A particularly useful feature allowed by HDF5 is subvolume access, where subvolumes can be loaded independently of the larger dataset. This allows small regions to be visualized without wasting memory on loading the entire dataset.

4.4.7 Available potential energy

Energy analysis of internal waves requires both the kinetic and potential energies. The kinetic energy is well-defined, but the potential energy must be defined with respect to a background stratification. The available potential energy [Lamb, 2008] gives the energy that can be converted to kinetic energy; this uses the background profile given by rearranging the density field to that with the minimum potential energy.

Rearranging the density field effectively sorts it, so that the profile is horizontally uniform and stably stratified. Then, the available potential energy over the domain is given by:

$$E = \int (\rho(x,z,t) - \bar{\rho}(z)) gz \mathrm{d}V, \tag{4.9}$$

where $\bar{\rho}(z)$ is the reference, sorted profile[11]. The local available potential energy density is also available, given by the expression (from Lamb [2008]):

$$E_a(x,z,t) = g \int_z^{z^*(\rho(x,z,t))} (\bar{\rho}(s) - \rho(x,z,t)) \, \mathrm{d}s, \tag{4.10}$$

[11]This is distinct from the background profile of equation (4.8c), for the anelastic equations. In this section, $\bar{\rho}$ refers to the sorted density profile.

where z^* is the height in the reference density profile of the fluid parcel with density $\rho(x,z,t)$ – it is the functional inverse of the reference density profile. In a numerical simulation, (4.10) is computable with numerical integration given $\bar{\rho}$.

Sorting the density profile to find $\bar{\rho}$ is ordinarily not a problem, but it becomes an issue with parallelization. In general, the implementation cannot assume that a full, three-dimensional field will fit in local memory on a single node, so the sorting must be undertaken in parallel. An appropriate algorithm for this sorting is the partitioned parallel radix sort [Lee et al., 2002]. This sorting algorithm effectives creates an approximate histogram of the data (in this case the density field), assigns buckets on that histogram to individual processors in order to ensure rough load balancing, and then uses a serial sorting algorithm on each processor to fully sort the density field. This algorithm requires only one large-scale communication step, and the computational costs are dominated by the per-processor sorting step.

Although implementation of this algorithm and specialization for floating-point density data would be a fairly complicated module, these scaling results are encouraging. Computation of the available potential energy and its local density should be possible at runtime.

4.5 Conclusions

The methods presented here and implemented in the associated numerical code represent an accurate, reasonably efficient method of simulating the incompressible Navier-Stokes equations in three dimensions. The total computational work scales as $O(N\log(N))$ for a domain with a total of N points, and this is maintained even when the x and z dimensions are coupled through coordinate mapping.

The time-discretization of the Navier-Stokes equations gives an implicit equation for computing pressure and additional implicit equations for viscous effects on velocities. Efficiently solving these equations with a spectral collocation method requires a preconditioner to avoid prohibitively expensive matrix inversion. This work developed a finite-difference preconditioning with a flexible GMRES algorithm, and the finite-difference preconditioner itself was solved through a geometric multigrid iteration. These algorithms were also developed with care for parallelization, and the resulting code efficiently parallelizes with the MPI message passing library.

This code can also be further extended to allow for greater simulation flexibility. The underlying numerical algorithms, currently restricted to two-dimensional coordinate mappings, can be extended to allow for full, three-dimensional mapping (with two-dimensional $h(x,y)$ topography). Other fairly complicated but significant improvements would allow for simulation of Lagrangian-frame particles and calculation of the available potential energy density during processing, rather than as an expensive postprocessing step. Additionally, simpler modifications would allow for simulation of sediment loading and the anelastic equations for atmospheric dynamics.

Appendix A

Source listings

A.1 Minimal code

Discussed in section 3.1, this is an example of the minimal, baseline framework required for this code to function correctly. It defines and initializes the user-supplied class, initializes the timetsepping, and steps to the trivial final time of 0.

Listing A.1: The minimal case

```
/* A minimal example case for the fluids model. When compiled and run, this
   exits cleanly after performing no useful computation. However, this also
   shows the basic required structure for any (useful) case for this
   underlying model. */

// Required headers
#include <blitz/array.h> // Blitz++ array library
#include "../TArray.hpp" // Custom extensions to the library to support FFTs
#include "../NSIntegrator.hpp" // Time-integrator for the Navier-Stokes equations
#include <mpi.h> // MPI parallel library
#include "../BaseCase.hpp" // Support file that contains default implementations of several functions

using namespace std;
using namespace NSIntegrator;

class minimal : public BaseCase {
  public:
    /* Set up a 100 x 1 x 100 grid */
    int size_x() const { return 100; }
    int size_y() const { return 1; }
```

```
      int size_z() const { return 100; }

      /* Set all boundaries to be periodic */
      DIMTYPE type_default() const { return PERIODIC; }
25
      /* The grid corresponds to a 1 (x 1) x 1 physical space */
      double length_x() const { return 1; }
      double length_y() const { return 1; }
      double length_z() const { return 1; }
30
      /* Use no tracer variables */
      int numtracers() const { return 0; }

      /* Start at t=0 */
35    double init_time() const { return 0; }

      /* Initialize velocities at the start of the run. For this simple
        case, initialize all velocities to 0 */
      void init_vels(DTArray & u, DTArray & v, DTArray & w) {
40      u = 0; // Use the Blitz++ syntax for simple initialization
        v = 0; // of an entire (2D or 3D) array with a single line
        w = 0; // of code.
        return;
      }
45
      /* The analysis routines are called at each timestep, since it's
        impossible to predict in advance just what will be interesting. For
        now, this function will do nothing. */
      void vel_analysis(double t, DTArray & u, DTArray & v, DTArray & w) {
50      return;
      }
   };

   /* The ``main'' routine */
55 int main(int argc, char ** argv) {
     /* Initialize MPI. This is required even for single-processor runs,
       since the inner routines assume some degree of parallelization,
       even if it is trivial. */
     MPI_Init(&argc, &argv);
60   minimal mycode; // Create an instantiated object of the above class
     /// Create a flow-evolver that takes its settings from the above class
     FluidEvolve<minimal> do_nothing(&mycode);
     // Initialize the flow
```

```
    do_nothing.initialize();
65  // Run to a final time of 1.
    do_nothing.do_run(1);
    MPI_Finalize(); // Cleanly exit MPI
    return 0; // End the program
}
```

A.2 Kelvin-Helmholtz Billows

Discussed in section 3.2, this code looks at the linear growth (early-time) of Kelvin-Helmholtz billows on the shear unstable profile given by the background density and velocity profiles:

$$\rho(z) = \rho_0 \frac{0.01}{2} \tanh\left(\frac{z-0.5}{0.1}\right) \tag{A.1}$$

$$u(z) = 0.1 \cdot 1.808 \tanh\left(\frac{z-0.5}{0.1}\right), \tag{A.2}$$

in a domain of total depth 1 ($0 \leq z \leq 1$). This gives a minimum Richardson number of 0.15 at the centre of the picnocline. From numerical evaluation of the Taylor-Goldstein equation [Kundu and Cohen, 2004], the most unstable wavenumber is approximately $k = 2.38434$, with wavelength approximately 2.64. This code initializes a domain of 4 times that wavelength and initializes with a density perturbation of the most unstable wavelength.

Listing A.2: Kelvin-Helmholtz billows

```
/* A sample case, for illustrating Kelvin-Helmholtz billows */

// Required headers
#include <blitz/array.h> // Blitz++ array library
5  #include "../TArray.hpp" // Custom extensions to the library to support FFTs
#include "../NSIntegrator.hpp" // Time-integrator for the Navier-Stokes equations
#include <mpi.h> // MPI parallel library
#include "../BaseCase.hpp" // Support file that contains default implementations of several functions

10 using namespace std;
using namespace NSIntegrator;

// Tensor variables for indexing
blitz::firstIndex ii;
15 blitz::secondIndex jj;
blitz::thirdIndex kk;
```

```
// Physical constants
const double g = 9.81;
const double rho_0 = 1; // Units of kg / L

// Pysical parameters
const double pertur_k = 2.38434; // Wavelength of most unstable perturbation
const double LENGTH_X = 8*M_PI/pertur_k; // 4 times the most unstable wavelength
const double LENGTH_Z = 1; // depth 1
const double delta_rho = 0.01; // Top to bottom density difference
const double RI = 0.15; // Richardson number at the centre of the pycnocline
const double dz_rho = 0.1; // Transition length for rho
const double dz_u = 0.1; // Transition length for u

const double N2_max = g*delta_rho/2/dz_rho; // Maximum N2
const double delta_u = 2*dz_u*sqrt(N2_max/RI); // Top-to-bottom shear

// Numerical parameters
int NZ = 0; // Number of vertical points. Number of horizontal points
int NX = 0; // will be calculated based on this.
const double plot_interval = 5; // Time between field writes
const double final_time = 200.0;


class helmholtz : public BaseCase {
  public:
    Array<double,1> xx, zz; // One-dimensional grid arrays
    // Variables to set the plot sequence number and time of the last writeout
    int plot_number; double last_plot;

    // Resolution in X, Y (1), and Z
    int size_x() const { return NX; }
    int size_y() const { return 1; }
    int size_z() const { return NZ; }

    /* Set periodic in x, free slip in z */
    DIMTYPE type_z() const {return FREE_SLIP;}
    DIMTYPE type_default() const { return PERIODIC; }

    /* The grid size is governed through the #defines above */
    double length_x() const { return LENGTH_X; }
    double length_y() const { return 1; }
    double length_z() const { return LENGTH_Z; }
```

```
/* Use one actively-modified tracer */
int numActive() const { return 1; }

// Use 0 viscosity and diffusivity
double get_visco() const { return 0; }
double get_diffusivity(int t_num) const { return 0; }

/* Start at t=0 */
double init_time() const { return 0; }

/* Modify the timestep if necessary in order to land evenly on a plot time */
double check_timestep (double intime, double now) {
  // Firstly, the buoyancy frequency provides a timescale that is not
  // accounted for with the velocity-based CFL condition.
  if (intime > 0.5/sqrt(N2_max)) {
    intime = 0.5/sqrt(N2_max);
  }
  // Now, calculate how many timesteps remain until the next writeout
  double until_plot = last_plot + plot_interval - now;
  int steps = ceil(until_plot / intime);
  // And calculate where we will actually be after (steps) timesteps
  // of the current size
  double true_fintime = steps*intime;

  // If that's close enough to the real writeout time, that's fine.
  if (fabs(until_plot - true_fintime) < 1e-6) {
    return intime;
  } else {
    // Otherwise, square up the timeteps. This will always shrink the timestep.
    return (until_plot / steps);
  }
}



// Initialize velocities at the beginning of the run
void init_vels(DTArray & u, DTArray & v, DTArray & w) {
  // Use the background shear profile for u
  u = 0.5*delta_u*tanh((zz(kk)-0.5)/dz_u);
  v = 0; // The other velocities are initially zero
  w = 0;
  // Also, write out the (zero) initial velocities and proper M-file readers
```

```
  write_reader(u,"u",true);
  write_reader(w,"w",true);
  write_array(u,"u",0);
  write_array(w,"w",0);
  return;
}

/* Initialze the temperature perturbation to a small value */
void init_tracer(int t_num, DTArray & rhoprime) {

  /* We want to write out a grid in order to make plots later,
    so let's re-use rhoprime to that end */

  // Assign the x-array to the two-dimensional grid
  rhoprime = xx(ii) + 0*kk;
  write_array(rhoprime,"xgrid"); write_reader(rhoprime,"xgrid",false);

  // Assign the z-array to the two-dimensional grid
  rhoprime = 0*ii + zz(kk);
  write_array(rhoprime,"zgrid"); write_reader(rhoprime,"zgrid",false);

  rhoprime = (cos(pertur_k*xx(ii)))*pow(cosh((zz(kk)-0.5)/dz_rho),-2)
          *delta_rho*1e-8;
  write_array(rhoprime,"rho",0); write_reader(rhoprime,"rho",true);
}

// Forcing in the momentum equations
void vel_forcing(double t, DTArray & u_f, DTArray & v_f, DTArray & w_f,
    vector<DTArray *> & tracers) {
  u_f = 0; v_f = 0;
  w_f = -g*((*tracers[0]))/rho_0;
}
// Forcing of the perturbation density
void tracer_forcing(double t, const DTArray & u, const DTArray & v,
    const DTArray & w, vector<DTArray *> & tracers_f) {
  /* Since the perturbation density is a perturbation, its forcing is
    proportional to the background density gradient and the w-velocity */
  *tracers_f[0] = w(ii,jj,kk)*0.5*delta_rho/dz_rho*pow(cosh((zz(kk)-0.5)/dz_rho),-2);
}


/* The analysis routines are called at each timestep, since it's
  impossible to predict in advance just what will be interesting.
```

```
    This function will write out volume average data and flow fields. */
  void analysis(double time, DTArray & u, DTArray & v, DTArray & w,
      vector<DTArray *> & tracer, DTArray & pressure) {
    /* If it is very close to the plot time, write data fields to disk */
150 if ((time - last_plot - plot_interval) > -1e-6) {
      plot_number++;
      if (master()) fprintf(stderr,"*");
      write_array(u,"u",plot_number);
      write_array(w,"w",plot_number);
155   write_array(*tracer[0],"rho",plot_number);
      last_plot = last_plot + plot_interval;
    }
    // Also, calculate and write out useful information: maximum u, w, and t'
    double max_u = psmax(max(abs(u)));
160 double max_w = psmax(max(abs(w)));
    double max_t = psmax(max(abs(*tracer[0])));
    // Energetics: mean(u^2), mean(w^2), and mean(rho*h)
    double usq = pssum(sum(u*u))/(NX*NZ);
    double wsq = pssum(sum(w*w))/(NX*NZ);
165 double rhogh = pssum(sum(*tracer[0]*g*zz(kk)))/(NX*NZ);
    if (master()) fprintf(stderr,"%.2f: %.2g %.2g %.2g\n",time,max_u,max_w,max_t);
    if (master()) {
      FILE * vels_output = fopen("velocity_output.txt","a");
      if (vels_output == 0) {
170     fprintf(stderr,"Unable to open velocity_output.txt for writing\n");
        exit(1);
      }
      fprintf(vels_output,"%.16g %.16g %.16g %.16g %.16g %.16g %.16g\n",
          time,max_u,max_w,max_t,usq,wsq,rhogh);
175   fclose(vels_output);
    }
  }

  helmholtz():
180 xx(split_range(NX)), zz(NZ)
  { // Initialize the local variables
    plot_number = 0;
    last_plot = 0;
    // Create one-dimensional arrays for the coordinates
185 xx = LENGTH_X*(-0.5 + (ii + 0.5)/NX);
    zz = LENGTH_Z*((ii+0.5)/NZ);
  }
```

```
};
190
/* The ‘‘main’’ routine */
int main(int argc, char ** argv) {
  /* Initialize MPI. This is required even for single−processor runs,
     since the inner routines assume some degree of parallelization,
195  even if it is trivial. */
  MPI_Init(&argc, &argv);
  if (argc > 1) { // Check command line arguments
    NZ = atoi(argv[1]); // Read in number of vertical points, if specified
  } else {
200 NZ = 256;
  }
  NX = rint(NZ*LENGTH_X/LENGTH_Z);
  if (master()) {
    fprintf(stderr,"Using a grid of %d x %d points\n",NX,NZ);
205 }
  helmholtz mycode; // Create an instantiated object of the above class
  /// Create a flow−evolver that takes its settings from the above class
  FluidEvolve<helmholtz> do_helmholtz(&mycode);

210 do_helmholtz.initialize();
  do_helmholtz.do_run(final_time);
  MPI_Finalize(); // Cleanly exit MPI
  return 0; // End the program
}
```

A.3 Dipole-wall interaction

Discussed in section 3.4, this code replicates the run of Clercx and Bruneau [2006] and Kramer et al. [2007] simulating the collision of a translating dipole with a no-slip boundary, at a Reynolds number of 1250. The initial conditions are two shielded monopoles at $(0.1,0)$ (positive) and $(-0.1,0)$ (negative) on a 2×2 grid, each having vorticity of the form:

$$\omega = 299.5284 \cdot \left(1-10^2r^2\right)\exp\left(-10^2r^2\right). \tag{A.3}$$

The integration is carried out at a grid-size specified on the command-line (defaulting to 64×64), until the final time of 1.6, which is long enough for primary and secondary interactions with the boundary. The fields are written to disk every 0.05 time-units for comparisons with Clercx and Bruneau [2006], and the kinetic energy ($0.5\int u^2+w^2dV$) and enstrophy ($0.5\int \omega^2dV$) are computed and written to disk on a per-timestep basis.

Listing A.3: The dipole-wall interaction

```
/* Copy of test case used in MATLAB code, collision of an initially
   shielded vortex dipole with a no-slip boundary */

#include "../Par_util.hpp"
#include <mpi.h>
#include "../BaseCase.hpp"
#include "../TArray.hpp"
#include "../NSIntegrator.hpp"
#include "../T_util.hpp"
#include <stdio.h>
#include <stdlib.h>
#include <random/normal.h>
#include <vector>
#include "../Science.hpp"

using namespace std;
using namespace TArrayn;
using namespace NSIntegrator;
using namespace ranlib;

blitz::firstIndex ii;
blitz::secondIndex jj;
blitz::thirdIndex kk;

#define D1_X -.1
#define D1_Z 0
#define D2_X .1
#define D2_Z 0
#define RAD2 0.01

double times_record[100], ke_record[100], enst_record[100], ke2d_record[100];
int myrank = -1;

class userControl : public BaseCase {
  public:
    /* Grid sizes and plot statistics */
    int szx, szy, szz, plotnum, itercount, lastplot, last_writeout;
    double plot_interval, nextplot;

    // Gradient operator
    Grad * gradient_op;
```

```
  // 1D grid arrays
  Array<double,1> xx, yy, zz;

  // Array for vorticity computation
  DTArray vorticity;

  // A no-slip box, with periodic spanwise
  DIMTYPE type_x() const { return NO_SLIP; }
  DIMTYPE type_y() const { return PERIODIC; }
  DIMTYPE type_z() const { return NO_SLIP; }

  // Use a grid-scale Rynolds-number of 1250
  double get_visco() const {
    return 1.0/1250;
  }

  // Give a 2x1x2 box
  double length_x() const { return 2; }
  double length_y() const { return 1; }
  double length_z() const { return 2; }

  int size_x() const { return szx; }
  int size_y() const { return szy; }
  int size_z() const { return szz; }

  double check_timestep(double intime, double now) {
    if (intime < 1e-9) {
      if (master()) fprintf(stderr,"Tiny timestep, aborting\n");
      return -1;
    } else if (itercount < 100 && intime > .01) {
      intime = .01;
    }
    // Now, calculate how many timesteps remain until the next writeout
    double until_plot = nextplot - now;
    int steps = ceil(until_plot / intime);
    // And calculate where we will actually be after (steps) timesteps
    // of the current size
    double true_fintime = steps*intime;

    // If that's close enough to the real writeout time, that's fine.
    if (fabs(until_plot-true_fintime) < 1e-6) {
      return intime;
```

```
85      } else {
          // Otherwise, square up the timeteps. This will always shrink the timestep.
          return (until_plot / steps);
        }
      }
90
      // Record the gradient−taking object. This is given by the NSIntegrator
      // code, and it reflects the boundary types and any Jacobian−transform
      void set_grad(Grad * in_grad) {
        gradient_op = in_grad;
95    }

      void compute_vorticity(DTArray & u, DTArray & w) {
        // Compute vorticity
        gradient_op−>setup_array(&u,CHEBY,FOURIER,CHEBY);
100     // Put du/dz in vorticity
        gradient_op−>get_dz(&vorticity,false);
        // Invert that to get −du/dz
        vorticity = vorticity*(−1);
        // And add dw/dx
105     gradient_op−>setup_array(&w,CHEBY,FOURIER,CHEBY);
        gradient_op−>get_dx(&vorticity,true);
      }


110   void analysis(double time, DTArray & u, DTArray & v, DTArray & w,
          vector<DTArray *> tracer, DTArray & pressure) {
        /* Write out velocities */
        bool plotted = false;
        itercount++;
115
        // Compute the 2D vorticity
        compute_vorticity(u,w);

        double enst, ke, ke2d;
120     // Compute (twice) enstrophy, sum(vort^2)
        enst = enst_record[itercount−last_writeout−1] =
          pssum(sum((*get_quad_x())(ii)*(*get_quad_y())(jj)*
               (*get_quad_z())(kk)*vorticity*vorticity));

125     // And KE sum(u^2+w^2). It needs to be divided by 2 for true energy.
        ke = ke_record[itercount−last_writeout−1] = pssum(sum(
             (*get_quad_x())(ii)*(*get_quad_y())(jj)*(*get_quad_z())(kk)*
```

```
          (pow(u(ii,jj,kk),2)+pow(v(ii,jj,kk),2)+pow(w(ii,jj,kk),2))));

130    // KE in the spanwise mean of velocity. For a 2D run (szy=1), this will
       // be identical to the full KE.
       ke2d = ke2d_record[itercount−last_writeout−1] = pssum(sum(
             (∗get_quad_x())(ii)∗(∗get_quad_z())(jj)∗
             (pow(mean(u(ii,kk,jj),kk),2)+
135           pow(mean(v(ii,kk,jj),kk),2)+
              pow(mean(w(ii,kk,jj),kk),2))))∗(length_y());

       // The current time.
       times_record[itercount−last_writeout−1] = time;
140

       if ((time − nextplot) > −1e−5∗plot_interval) {
         plotted = true;
         nextplot += plot_interval;
145      if (master()) fprintf(stdout,"*");
         plotnum++;
         // Write u, v, w, and vorticity to disk
         write_array(u,"u",plotnum);
         write_array(v,"v",plotnum);
150      write_array(w,"w",plotnum);
         write_array(vorticity,"vort",plotnum);
         lastplot = itercount;
         if (master()) {
           // And save the current timestep for reference
155        FILE ∗ plottimes = fopen("plot_times.txt","a");
           assert(plottimes);
           fprintf(plottimes,"%.10g\n",time);
           fclose(plottimes);
         }
160    }
       if ((itercount − lastplot)%1 == 0 || plotted) {
         double mu = psmax(max(abs(u))),
              mv = psmax(max(abs(v))),
              mw = psmax(max(abs(w)));
165      // Diagnostic information −− write out maximum (absolute) velocities
         // along with enstrophy and KE on a per−timestep basis. This allows
         // for very finely−detailed comparisons between runs.
         if (master())
           fprintf(stdout,"%f␣[%d]␣(%.4g,␣%.4g,␣%.4g)␣--␣(%g,␣%g,␣%g)\n",
170            time, itercount, mu, mv, mw,enst,ke,ke2d);
```

```
    if (master()) {
      FILE * en_record = fopen("energy_record.txt","a");
      assert(en_record);
      for (int i = 0; i < (itercount-last_writeout); i++) {
175     fprintf(en_record,"%.9g %.9g %.9g %.9g\n",times_record[i],
            ke_record[i],enst_record[i],ke2d_record[i]);
      }
      fclose(en_record);
    }
180 last_writeout = itercount;
  }
}

void init_vels(DTArray & u, DTArray & v, DTArray & w) {
185 xx = -length_z()/2*cos(M_PI*ii/(szx-1));
  yy = -(length_y()/2) + length_y()*(ii+0.5)/szy; // unused in 2D
  zz = -(length_z()/2)*cos(M_PI*ii/(szz-1));
  Array<double,3> grid(alloc_lbound(szx,szy,szz),
              alloc_extent(szx,szy,szz),
190           alloc_storage(szx,szy,szz));
  /* Vortex strength is set according to Clercx(2006) and Kramer(2007),
     which normalizes initial KE to 2 and initial enstrophy to 800 */
  u = 299.5284/2*(
    +(zz(kk)-D1_Z)*exp(-(pow(xx(ii)-D1_X,2)+pow(zz(kk)-D1_Z,2))/RAD2) -
195  (zz(kk)-D2_Z)*exp(-(pow(xx(ii)-D2_X,2)+pow(zz(kk)-D2_Z,2))/RAD2));
  w = 299.5284/2*(
    -(xx(ii)-D1_X)*exp(-(pow(xx(ii)-D1_X,2)+pow(zz(kk)-D1_Z,2))/RAD2) +
     (xx(ii)-D2_X)*exp(-(pow(xx(ii)-D2_X,2)+pow(zz(kk)-D2_Z,2))/RAD2));
  v = 0;
200 // Write out the initial arrays: velocity...
  write_array(u,"u",0);
  write_reader(u,"u",true);
  write_array(v,"v",0);
  write_reader(v,"v",true);
205 write_array(w,"w",0);
  write_reader(w,"w",true);
  // ... and vorticity
  compute_vorticity(u,w);
  write_array(vorticity,"vort",0);
210 write_reader(vorticity,"vort",true);
  grid = xx(ii) + 0*kk;
  write_array(grid,"xgrid"); write_reader(grid,"xgrid",false);
  grid = yy(jj) + 0*kk;
```

```
      write_array(grid,"ygrid"); write_reader(grid,"ygrid",false);
215   grid = zz(kk);
      write_array(grid,"zgrid"); write_reader(grid,"zgrid",false);
    }
    // Once initialized, this is freely−evolving flow. No forcing is necessary
    void passive_forcing(double t, const DTArray & u, DTArray & u_f,
220     const DTArray & v, DTArray & v_f,
        const DTArray & w, DTArray & w_f) {
      u_f = 0;
      v_f = 0;
      w_f = 0;
225 }
    userControl(int s):
      // Setup a 2D run, of size S x 1 x S
      szx(s), szy(1), szz(s),
      // Write out fields every 0.005 timeunits for detailed graphics
230   plotnum(0), plot_interval(.005),
      nextplot(plot_interval), lastplot(0),
      itercount(0),last_writeout(0),
      // Initalize arrays for 1D grid coordinates
      xx(split_range(szx)), yy(szy), zz(szz),
235   // Initalize the array for vorticity computation
      vorticity(alloc_lbound(szx,szy,szz),
          alloc_extent(szx,szy,szz),
          alloc_storage(szx,szy,szz)) {
        compute_quadweights(szx,szy,szz,
240         length_x(),length_y(),length_z(),
            type_x(),type_y(),type_z());
        if (master()) {
          printf("Using␣array␣size␣%d\n",s);
        }
245   }
};

int main(int argc, char ** argv) {
  MPI_Init(&argc, &argv);
250 MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
  // Read in the grid size from the command−line
  int grid_size = 0;
  if (argc > 1) grid_size = atoi(argv[1]);
  // And use a sensible default if it's not given or invalid
255 if (grid_size <= 0) grid_size = 64;
  userControl mycode(grid_size);
```

```
    FluidEvolve<userControl> ppois(&mycode);
    ppois.initialize();
    ppois.do_run(1.4);
260 MPI_Finalize();
    return 0;
}
```

A.4 Internal wave generation by topograhpy

Discussed in section 3.6, this code models the production of internal waves in a linearly stratified basin via generation from a topological feature. The basin depth is 5km and the domain length is 400km, with a hill in the middle of the domain given by:

$$h(x) = 1500\text{m} \cdot \exp\left(\left(\frac{x-200\text{km}}{12\text{km}}\right)^2\right). \tag{A.4}$$

This hill forms the bottom boundary of the domain. The background stratification is set so that the buoyancy frequency is 10^{-3}s, and the domain is forced at the M2 tidal period of $44,712$s. The forcing also includes the standard Coriolis force, with $f = \frac{1}{2}10^{-4}\text{s}^{-1}$.

Listing A.4: Internal wave generation

```
1  /* Generation of internal waves by topographical forcing */

   // Required headers
   #include <blitz/array.h> // Blitz++ array library
   #include "../TArray.hpp" // Custom extensions to the library to support FFTs
6  #include "../NSIntegrator.hpp" // Time-integrator for the Navier-Stokes equations
   #include <mpi.h> // MPI parallel library
   #include "../BaseCase.hpp" // Support file that contains default implementations of several functions

   using namespace std;
11 using namespace NSIntegrator;

   // Tensor variables for indexing
   blitz::firstIndex ii;
   blitz::secondIndex jj;
16 blitz::thirdIndex kk;

   // Physical constants
   const double g = 9.81;
   const double rho_0 = 1028; // Units of kg / m^3
```

```
// Pysical parameters
const double ROT_F = 0.5e-4; // 1/s, mid-latitude
const double LENGTH_X = 4e5; // 400km
const double LENGTH_Z = 5000; // Water depth
const double N_max = 1e-3; // Linear stratification buoyancy frequency

const double S0 = 35; // Baseline salinity
const double beta = 7.6e-4; // Density change from salt content

// Hill parameters
const double H_HEIGHT = 1500; // Hill height
const double H_LENGTH = 12e3; // Hill length

// Tide parameters
const double TIDE_PERIOD = 44712;
const double TIDE_M2 = 2*M_PI/TIDE_PERIOD; // M2 tidal frequency
const double TIDE_STRENGTH = 0.01; // Desired maximum tidal current

// Numerical parameters
int NZ = 0; // Number of vertical points. Number of horizontal points
int NX = 0; // will be calculated based on this.
const double plot_interval = TIDE_PERIOD/32; // Time between field writes
const double final_time = 4*TIDE_PERIOD;


class mapiw : public BaseCase {
  public:
    // Variables to set the plot sequence number and time of the last writeout
    DTArray * xgrid, * zgrid;
    Array<double,1> hill;
    int plot_number; double last_plot;

    // Resolution in X, Y (1), and Z
    int size_x() const { return NX; }
    int size_y() const { return 1; }
    int size_z() const { return NZ; }

    /* Set periodic in x, no-slip (Chebyshev) in z */
    DIMTYPE type_z() const {return NO_SLIP;}
    DIMTYPE type_default() const { return PERIODIC; }

    /* The grid size is governed through the #defines above */
```

```
double length_x() const { return LENGTH_X; }
double length_y() const { return 1; }
double length_z() const { return LENGTH_Z; }

bool is_mapped() const {return true;}
void do_mapping(DTArray & xg, DTArray & yg, DTArray & zg) {
  xgrid = alloc_array(NX,1,NZ);
  zgrid = alloc_array(NX,1,NZ);
  Array<double,1> xx(split_range(NX)), zz(NZ);
  // Use periodic coordinates in horizontal
  xx = (ii+0.5)/NX; // x-coordinate
  zz = cos(ii*M_PI/(NZ-1)); // Chebyshev in vertical

  xg = LENGTH_X*xx(ii) + 0*jj + 0*kk;
  *xgrid = xg;

  hill = H_HEIGHT*exp(-pow(LENGTH_X*(xx(ii)-1/2)/H_LENGTH,2));
  zg = -LENGTH_Z/2+LENGTH_Z/2*zz(kk) + 0.5*(1-zz(kk))*
       hill(ii);;
  *zgrid = zg;

  yg = 0;

  write_array(xg,"xgrid");
  write_reader(xg,"xgrid",false);
  write_array(zg,"zgrid");
  write_reader(zg,"zgrid",false);
}
/* Use one actively-modified tracer */
int numActive() const { return 1; }

// Use 0 viscosity and diffusivity
double get_visco() const { return 0; }
double get_diffusivity(int t_num) const { return 0; }

/* Start at t=0 */
double init_time() const { return 0; }

/* Modify the timestep if necessary in order to land evenly on a plot time */
double check_timestep (double intime, double now) {
  // Firstly, the buoyancy frequency provides a timescale that is not
  // accounted for with the velocity-based CFL condition.
  if (intime > 0.5/N_max) {
```

```
    intime = 0.5/N_max;
  }
  // Now, calculate how many timesteps remain until the next writeout
  double until_plot = last_plot + plot_interval − now;
111 int steps = ceil(until_plot / intime);
  // And calculate where we will actually be after (steps) timesteps
  // of the current size
  double true_fintime = steps∗intime;

116 // If that's close enough to the real writeout time, that's fine.
  if (fabs(until_plot − true_fintime) < 1e−6) {
    return intime;
  } else {
    // Otherwise, square up the timeteps. This will always shrink the timestep.
121   return (until_plot / steps);
  }
}


126 /∗ Initialize velocities at the start of the run. For this simple
  case, initialize all velocities to 0 ∗/
void init_vels(DTArray & u, DTArray & v, DTArray & w) {
  u = 0;
  v = −TIDE_STRENGTH∗ROT_F/TIDE_M2∗LENGTH_Z/(LENGTH_Z−hill(ii));
131 w = 0;
  // Also, write out the (zero) initial velocities and proper M−file readers
  write_reader(u,"u",true);
  write_reader(v,"v",true);
  write_reader(w,"w",true);
136 write_array(u,"u",0);
  write_array(v,"v",0);
  write_array(w,"w",0);
  return;
}
141
void init_tracer(int t_num, DTArray & salt) {
  // The primary constituent of density is salt, so that is
  // initialize here

146 salt = S0 + (N_max∗N_max)/(−beta∗g)∗(∗zgrid)(ii,jj,kk);
  write_array(salt,"s",0); write_reader(salt,"s",true);
}
```

```
    // Forcing must be done generally, since both rotation and density are
151 // involved
    void forcing(double t, const DTArray & u, DTArray & u_f,
        const DTArray & v, DTArray & v_f, const DTArray & w,
        DTArray & w_f, vector<DTArray *> & tracers,
        vector<DTArray *> & tracers_f) {
156   // Rotation couples u and v, plus a source term for the tide
      u_f = -ROT_F*v + cos(t*TIDE_M2)*TIDE_STRENGTH*
        (TIDE_M2-ROT_F*ROT_F/TIDE_M2*LENGTH_Z/(LENGTH_Z-hill(ii)));
      v_f = ROT_F*u;
      w_f = -g*beta*(*tracers[0]-S0);
161   // And since the salt content is expressed as total content rather
      // than perturbation, no forcing is necessary.
      *tracers_f[0] = 0;
    }

166 /* Basic analysis, to write out the field periodically */
    void analysis(double time, DTArray & u, DTArray & v, DTArray & w,
        vector<DTArray *> & tracer, DTArray & pressure) {
      /* If it is very close to the plot time, write data fields to disk */
      if ((time - last_plot - plot_interval) > -1e-6) {
171     plot_number++;
        if (master()) fprintf(stderr,"*");
        write_array(u,"u",plot_number);
        write_array(v,"v",plot_number);
        write_array(w,"w",plot_number);
176     write_array(*tracer[0],"s",plot_number);
        last_plot = last_plot + plot_interval;
      }
      // Also, calculate and write out useful information: maximum u, w, and t'
      double max_u = psmax(max(abs(u)));
181   double max_w = psmax(max(abs(w)));
      double max_t = psmax(max(abs(*tracer[0])));
      if (master()) fprintf(stderr,"%.2f: %.2g %.2g %.2g\n",time,max_u,max_w,max_t);
    }

186 mapiw(): hill(split_range(NX))
    { // Initialize the local variables
      plot_number = 0;
      last_plot = 0;
      // Create one-dimensional arrays for the coordinates
191 }
```

```
};

/* The ``main'' routine */
196 int main(int argc, char ** argv) {
  /* Initialize MPI. This is required even for single-processor runs,
     since the inner routines assume some degree of parallelization,
     even if it is trivial. */
  MPI_Init(&argc, &argv);
201 if (argc > 3) { // Check command line arguments
    NX = atoi(argv[1]); // Read in number of horizontal points, if specified
    NZ = atoi(argv[2]); // and vertical
  }
  if (NX <= 0) {
206   NX = 2048;
  }
  if (NZ <= 0) {
    NZ = 128;
  }
211 if (master()) {
    fprintf(stderr,"Using a grid of %d x %d points\n",NX,NZ);
  }
  mapiw mycode; // Create an instantiated object of the above class
  /// Create a flow-evolver that takes its settings from the above class
216 FluidEvolve<mapiw> do_mapiw(&mycode);
  // Run to a final time of 1.
  do_mapiw.initialize();
  do_mapiw.do_run(final_time);
  MPI_Finalize(); // Cleanly exit MPI
221 return 0; // End the program
}
```

References

A. Adcroft, J.-M. Campin, S. Dutkiewicz, C. Evangelinos, D. Ferreira, G. Forget, B. Fox-Kemper, P. Heimbach, C. Hill, E. Hill, H. Hill, O. Jahn, M. Losch, J. Marshall, G. Maze, D. Menemenlis, and A. Molod. The Massachusetts Institute of Technology general circulation model, 2011. URL http://mitgcm.org. 4

P. Aghsaee, L. Boegman, and K. G. Lamb. Breaking of shoaling internal solitary waves. *Journal of Fluid Mechanics*, 659:289–317, 2010. 3

U. M. Ascher and L. R. Petzold. *Computer methods for ordinary differential equations and differential-algebraic equations*. Society for Industrial and Applied Mathematics, 1998. 20

M. F. Barad and O. B. Fringer. Simulations of shear instabilities in interfacial gravity waves. *Journal of Fluid Mechanics*, 644:61–95, 2010. 124

M. R. Bate, I. A. Bonnell, and V. Bromm. The formation of a star cluster: predicting the properties of stars and brown dwarfs. *Monthly Notices of the Royal Astronomical Society*, 339(3): 577–599, March 2003. 29

J. B. Bell, P. Colella, and L. H. Howell. An efficient second-order projection method for viscous incompressible flow. In *Proceedings of the 10th AIAA Computational Fluid Dynamics Conference*, pages 789–794, 1991. 11

H. bin Zubair, S. P. MacLachlan, and C. W. Oosterlee. A geometric multigrid method based on L-shaped coarsening for PDEs on stretched grids. *Numerical Linear Algebra with Applications*, 17(6):871–894, 2010. 71

J. P. Boyd. A fast algorithm for Chebyshev, Fourier, and sinc interpolation onto an irregular grid. *Journal of Computational Phyiscs*, 103:243–257, 1992. 140

J. P. Boyd. *Cheyshev and Fourier spectral methods*. Dover publications, 2001. ix, 25, 35, 37, 39, 45, 46, 48, 49, 54, 57, 61, 62, 73, 74, 109, 139

W. L. Briggs. *A multigrid tutorial*. Society for Industrial and Applied Mathematics, 1987. 64, 68, 69

T. M. Burton and J. K. Eaton. Fully resolved simulations of particle-turbulence interaction. *Journal of Fluid Mechanics*, 545:67–111, 2005. 141

C. Canuto, M. Y. Hussaini, A. Quarteroni, and T. A. Zang. *Spectral methods in fluid dynamics*. Springer-Verlag, 1988. 61, 62

M. Carr and P. A. Davies. Boundary layer flow beneath an internal solitary wave of elevation. *Physics of Fluids*, 22, 2010. 127

A. R. Cieslik, R. A. D Akkermans, L. P. J Kamp, H. J. H. Clercx, and G. J. F van Heijst. Dipole-wall collision in a shallow fluid. *European Journal of Mechanics - B/Fluids*, 28(3):397–404, 2009. 121

H. J. H. Clercx and C.-H. Bruneau. The normal and oblique collision of a dipole with a no-slip boundary. *Computers & Fluids*, 35:245–279, 2006. 6, 101, 102, 104, 107, 108, 109, 121, 152

J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965. 37

J. Cummings and P Hilscher. [Blitz-support] vectorization issue and data alignment. Blitz-support e-mail discussion list, January 2011. URL `http://sourceforge.net/mailarchive/forum.php?forum_name=blitz-support`. 143

B. Cushman-Roisin and J.-M. Beckers. *Introduction to Geophysical Fluid Dynamics: Physical and Numerical Aspects*. Academic Press, 2011. URL `http://engineering.dartmouth.edu/~cushman/books/GFD.html`. 3, 113

R. H. Davis and A. Acrivos. Sedimentation of noncolloidal particles at low Reynolds numbers. *Annual Review of Fluid Mechanics*, 17:91–118, 1985. 141

T. A. Davis. Algorithm 832: UMFPACK V4.3 — an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software*, 30(2):196–199, June 2004a. 51, 73

T. A. Davis. A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software*, 30(2):165–195, June 2004b. 73

P. J. Diamessis and L. G. Redekopp. Numerical investigation of solitary internal wave-induced global instability in shallow water benthic boundary layers. *Journal of physical oceanography*, 36:784–812, 2006. 127

P. J. Diamessis, J. A. Domaradzki, and J. S. Hesthaven. A spectral multidomain penalty method model for the simulation of high Reynolds number localized incompressible stratified turbulence. *Journal of Computational Physics*, 202:298–322, 2005. 4

C. C. Douglas. A review of numerous parallel multigrid methods. In Greg Astfalk, editor, *Applications on Advanced Architecture Computers*. Society for Industrial and Applied Mathematics, 1996. 72

M. Folk, R. E. McGrath, and N. Yeager. HDF: an update and future directions. In *Geoscience and remote sensing symposium, 1999. IGARSS '99 proceedings. IEEE 1999 International*, volume 1, pages 273–275, 1999. 143

O.B. Fringer, M. Gerritsen, and R. L. Street. An unstructured-grid, finite-volume, nonhydrostatic parallel coastal ocean simulator. *Ocean Modelling*, 14:139–173, 2006. 4, 53

S. R. Fulton. A semi-implicit spectral method for the anelastic equations. *Journal of Computational Physics*, 106:299–305, 1993. 142

P. Godon and G. Shaviv. A two-dimensional time dependent Chebyshev method of collocation for the study of astrophysical flows. *Computer Methods in Applied Mechanics and Engineering*, 110:171–194, 1993. 110

J. Hesthaven, S. Gotlieb, and D. Gottlieb. *Spectral Methods for Time-Dependent Problems*. Cambridge University Press, 2007. 111

T. G. Jensen. Open boundary conditions in stratified ocean models. *Journal of Marine Systems*, 16:297–322, October 1998. 138

G. E. Karniadakis and S. J. Sherwin. *Spectral/hp element methods for computational fluid dynamics*. Oxford University Press, 2005. 4, 29

G. E. Karnidakis, M. Israeli, and S. A. Orszag. High-order splitting methods for the incompressible Navier-Stokes equations. *Journal of Computational Physics*, 97:414–443, 1991. 8, 12, 13

G. H. Keetels, H. J. H. Clercx, and G. J. F. van Heijst. Fourier spectral solver for the incompressible Navier-Stokes equations with volume-penalization. volume 4487 of *Lecture Notes in Computer Science*, pages 898–905. Springer, 2007. 5, 53

B. Kneller and C. Buckee. The structure and fluid mechanics of turbidity currents: a review of some recent studies and their geological implications. *Sedimentoogy*, 47:62–94, February 2000. 141

W. Kramer, H. J. H. Clercx, and G. J. F. van Heijst. Vorticity dynamics of a dipole colliding with a no-slip wall. *Physics of Fluids*, 19, 2007. doi: 10.1063/1.2814345. 6, 102, 104, 107, 108, 109, 121, 123, 152

P. K. Kundu and I. M. Cohen. *Fluid mechanics*. Elsevier, 2004. 1, 3, 9, 41, 44, 89, 90, 119, 147

K. G. Lamb. Numerical experiments of internal wave generation by strong tidal flow across a finite amplitude bank edge. *Journal of Geophyiscal Research*, 99(C1):843–864, 1994. xii, 3, 4, 10, 53, 118, 119, 125

K. G. Lamb. A numerical investigation of solitary internal waves with trapped cores formed via shoaling. *Journal of Fluid Mechanics*, 451:109–144, 2002. 124

K. G. Lamb. On the calculation of the available potential energy of an isolated perturbation in a density-stratified fluid. *Journal of Fluid Mechanics*, 597:415–427, 2008. 143

K. G. Lamb and D. Farmer. Instabilities in an internal solitary-like wave on the Oregon Shelf. *Journal of Physical Oceanography*, 41:67–87, 2011. 123

H. P. Langtangen, K.-A. Mardal, and R. Winther. Numerical methods for incompressible viscous flow. *Advances in Water Resources*, 25:1125–1146, 2002. 9

F. Laporte and A. Corjon. Direct numerical simulations of the elliptic instability of a vortex pair. *Physics of Fluids*, 12:1016–1031, 2000. 121

J. Larsson, F. S. Lien, and E. Yee. Conditional semicoarsening multigrid algorithm for the Poisson equation on anisotropic grids. *Journal of Computational Physics*, 208:368–383, 2005. 66, 71, 137

S.-J. Lee, M. Jeon, D. Kim, and A. Sohn. Partitioned parallel radix sort. *Journal of Parallel and Distributed Computing*, 62(4):656–668, 2002. 144

T. Leweke and C. H. K. Williamson. Cooperative elliptic instability of a vortex pair. *Journal of Fluid Mechanics*, 360:85–119, 1998. 121, 123

Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 2.1*. High Performance Computing Center Stuttgart, 2008. URL `http://www.mpi-forum.org/docs/mpi21-report.pdf`. 86, 143

P. Moin and K. Mahesh. Direct numerical simulation: a tool in turbulence research. *Annual Review of Fluid Mechanics*, 30:539–578, 1998. 4

J. J. Monaghan. An introduction to SPH. *Computer Physics Communications*, 48(1):89–96, January 1988. 29

Y. Ogura and N. A. Phillips. Scale analysis of deep and shallow convection in the atmosphere. *Journal of Atmospheric Sciences*, 19(2):173–179, March 1962. 142

S. A. Orszag. On the elimination of aliasing in finite difference schemes by filtering high-wavenumber components. *Jorunal of the Atmospheric Sciences*, 28:1074, 1971. 110

S. A. Orszag, M. Israeli, and M. O. Deville. Boundary conditions for incompressible flows. *Journal of Scientific Computing*, 1(1):75–111, 1986. 12

R. Peyret. *Spectral methods for incompressible viscous flow*. Springer-Verlag, 2002. 4, 9

D. Rosenberg, A. Fournier, P. Fischer, and A. Pouquet. Geophysical-astrophysical spectral-element adaptive refinement (GASpAR): Object-oriented *h*-adaptive fluid dynamics simulation. *Journal of Computational Physics*, 215(1):59–80, June 2006. 4

Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM Journal on Scientific Computing*, 14(2):461–469, 1993. 5

W. D. Smyth, J. D. Nash, and J. N. Moum. Differential diffusion in breaking Kelvin-Helmholtz billows. *Journal of Physical Oceanography*, 35:1004–1022, 2005. 89, 126

N. Soontiens, C. Subich, and M. Stastna. Numerical simulation of supercritical trapped internal waves over topography. *Physics of Fluids*, 22(11), 2010. 61

A. Staniforth and J. Côté. Semi-Lagrangian integration schemes for atmospheric models – a review. *Monthly Weather Review*, 119:2206–2223, 1991. 8

M. Stastna and K. G. Lamb. Vortex shedding and sediment resuspension associated with the interaction of an internal solitary wave and the bottom boundary layer. *Geophysical Research Letters*, 29(11), 2002. 132

L. N. Trefethen. *Spectral Methods in* MATLAB. Society for Industrial and Applied Mathematics, 2000. 32, 34, 49, 50, 65, 105

L. N. Trefethen and D. Bau, III. *Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, 1997. 57, 58, 59, 60

U. Trottenberg, C. Oosterlee, and A. Schüller. *Multigrid*. Academic Press, 2001. 64, 65, 67, 68, 69, 70, 71, 74, 75, 77, 79

W. Tsai and L. Hung. Three-dimensional modeling of small-scale processes in the upper boundary layer bounded by a dynamic ocean surface. *Journal of Geophysical Research*, 112, 2007. 5

S. V. Tsynkov. Numerical solution of problems on unbounded domains. a review. *Applied Numerical Mathematics*, 27:465–532, 1998. 138

T. Veldhuizen. Arrays in Blitz++. In Denis Caromel, Rodney Oldehoeft, and Marydell Tholburn, editors, *Computing in Object-Oriented Parallel Environments*, volume 1505 of *Lecture Notes in Computer Science*, pages 501–501. Springer, 1998. 87

T. Veldhuizen. Blitz++ user's guide: A C++ class library for scientific computing, 2006. URL http//www.oonumerics.org/blitz/docs/blitz.pdf. 87

D. Wang and S. J. Ruuth. Variable step-size implicit-explicit linear multistep methods for time-dependent partial differential equations. *Journal of Computational Mathematics*, 26(6):838–855, 2008. 22, 23

A. Warn-Varnas, J. Hawkins, K. G. Lamb, S. Piacsek, S. Chin-Bing, D. King, and G. Burgos. Solitary wave generaton dynamics at Luzon Strait. *Ocean Modelling*, 31:9–27, 2010. 3

K. B. Winters, J. A. MacKinnon, and B. Mills. A spectral model for process studies of rotating, density-stratified flows. *Journal of Atmospheric and Oceanic Technology*, 21:69–94, January 2004. 4, 40, 53, 63

:wq