The School of Mathematics

THE UNIVERSITY of EDINBURGH

Modelling Geophysical Fluids Using Dedalus

by

Tristan Pang

Dissertation Presented for the Degree of
MSc in Computational Applied Mathematics

August 2023

Supervised by
Dr James R. Maddison

Abstract

Partial differential equations (PDEs) are an essential part of mathematics and can be applied in a diverse range of places, including geophysical problems. Spectral methods are an alternative to the traditional finite element and finite difference methods for numerically solving PDEs. Spectral methods rely on expanding the solution out in terms of basis functions. It is a global method, as opposed to the finite element and finite difference methods, which rely on local information on a specified grid. Spectral methods offer the benefit of yielding smooth solutions but require a regular domain.

Dedalus is a Python package that implements spectral methods in a human-readable manner. In this dissertation, we start by testing Dedalus on time-independent models. We then build up towards time-dependent models, and simulate a simple advection problem. We analyse the errors in each of these examples and draw conclusions about the best method for our main model.

These pieces are then put together to simulate the Stommel–Munk problem on a circular domain. This time-dependent PDE is a simplified model of a shallow ocean based on the vorticity equation and forced by the wind with Stommel's drag and Munk's viscosity damping. It captures ocean-like behaviour, including westward intensification and the formation of eddy currents.

Acknowledgements

I would like to express my deepest appreciation to my supervisor, Dr James Maddison, for his guidance, support and encouragement throughout this project and the Computational Applied Mathematics MSc programme. His overall insights in this field have been inspiring and have helped me smoothly transition to a new area of research.

I am also grateful for the financial support from the University of Edinburgh School of Mathematics Scholarship.

Many thanks to my academic mentors and role models, Prof. Steven Galbraith and Prof. Victor Flynn, for their invaluable patience and advice. Whenever I am at an academic hurdle, they always point me in the right direction.

This endeavour would not have been possible without my parents' unconditional support. Their belief in me is my greatest source of motivation.

Own Work Declaration

I confirm that this dissertation is my own work except where explicitly indicated in the text.

Contents

List of Figures

1 Introduction

There is widespread consensus that climate change has a global impact [16] and has been attributed to human behaviour [7]. June 2023 has been the hottest since temperature records started [11]. This heat has caused wildfires [34] and heat exhaustion [11]. Understanding climate patterns is a crucial step to alleviating the effects of climate change.

The flow of water in our oceans is one such area of study. For example, a recent study suggests that the Atlantic meridional overturning circulation (AMOC), a large-scale flow of water in the Atlantic Ocean, could collapse in the future due to an influx of freshwater from melting icecaps due to greenhouse gas emissions [10]. They reach this conclusion by fitting real-world data to their stochastic model. Crossing such a tipping point would have significant implications for Earth and could lead to extreme weather events [16].

Another important part of the Atlantic Ocean is the Gulf Stream, a flow of warm water from North America to Europe, contributing to warmth in Western Europe [35]. Modelling ocean flows such as the Gulf Stream allows us to understand the formation of such flows and their effects. In 2011, NASA published an animation of the world's ocean currents [29], which includes the image in Figure 1. This image shows westward intensification, where intense current forms on the western boundary of a body of water. The standard model for this is Stommel's wind-driven ocean model [30], which describes this behaviour. Munk proposed a modification of the Stommel model, in which friction, described using the Laplacian of the stream function, is replaced by the viscosity, the Laplacian of the vorticity [23].

Combining these two models yields the Stommel–Munk model [31]. This model is described by the time-dependent partial differential equation (PDE) in Equation 28 on a circular domain with free slip boundary conditions. We assume a sufficiently shallow ocean to use a two-dimensional model. By forcing the vorticity equation with a sinusoidal wind and implementing the Stommel drag and Munk viscosity damping, we observe the formation of eddy currents by examining the vorticity over time.

This dissertation aims to numerically implement the Stommel–Munk model PDE in Dedalus, a Python package for spectral methods.

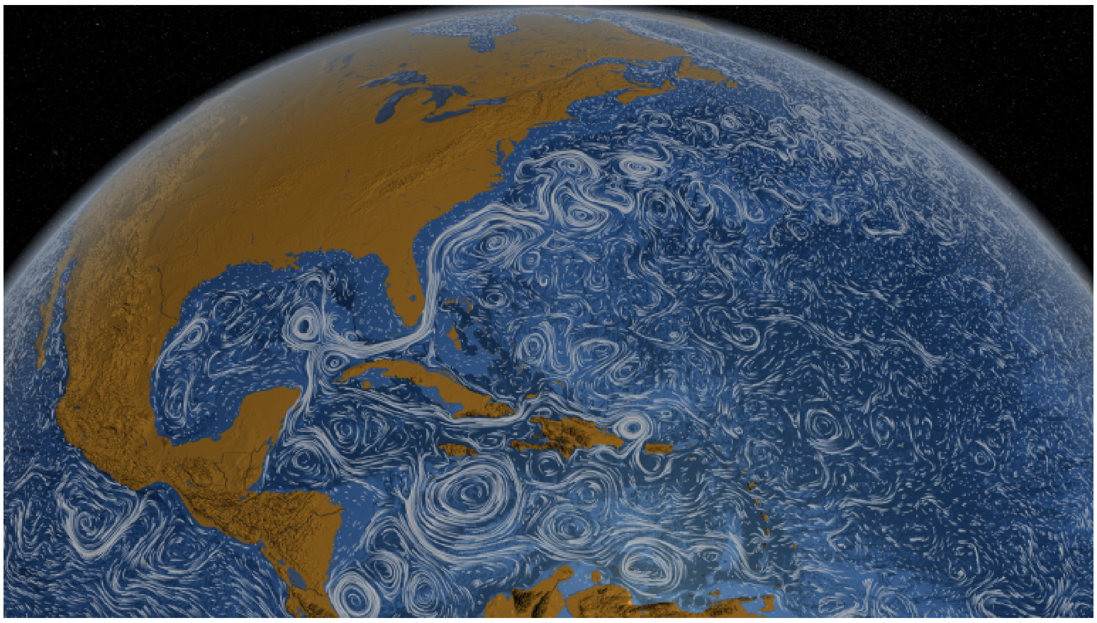

Figure 1: NASA's visualisation of the Gulf Stream using a mix of real ocean data and numerical models. The image was created by NASA's Scientific Visualization Studio [29].

1.1 Solving PDEs

Consider the 2-dimensional Poisson equation with forcing $f(x, y)$ on a square domain, a prototypical elliptic PDE [19]. Let the domain be $\Omega = \{(x, y) : 0 \leq x, y \leq 1\}$ and impose Dirichlet boundary conditions:

$$\begin{cases} \nabla^2 \psi = f(x, y) & \text{in } \Omega, \\ \quad \psi = 0 & \text{on } \partial\Omega. \end{cases} \tag{1}$$

An elementary method for numerically solving the Laplace Equation demonstrated in [19] uses a finite difference method via a 5-point stencil for the Laplacian. The problem is discretised in space with grid spacing h (in both coordinate directions) by replacing the derivatives with centred finite differences, namely

$$\nabla^2 \psi(x_i, y_j) \approx \frac{1}{h^2}(\psi_{i-1,j} - 2\psi_{ij} + \psi_{i+1,j}) + \frac{1}{h^2}(\psi_{i,j-1} - 2\psi_{ij} + \psi_{i,j+1}), \tag{2}$$

where $x_i = ih$, $y_j = jh$ and $\psi_{i,j} = \psi(x_i, y_j)$. These approximations can be gathered to form a sparse matrix system of dimension $m^2 \times m^2$ with $m^2 = \frac{1}{h^2}$ (where h is nicely chosen so that m is an integer):

$$\begin{gathered} L\boldsymbol{\psi} = \mathbf{f}, \quad \boldsymbol{\psi} = \begin{pmatrix} \boldsymbol{\psi}^1 \\ \vdots \\ \boldsymbol{\psi}^m \end{pmatrix}, \boldsymbol{\psi}^i = \begin{pmatrix} \psi_{1i} \\ \vdots \\ \psi_{mi} \end{pmatrix}, \quad \mathbf{f} = \begin{pmatrix} \mathbf{f}^1 \\ \vdots \\ \mathbf{f}^m \end{pmatrix}, \quad \mathbf{f}^i = \begin{pmatrix} f(x_1, y_i) \\ \vdots \\ f(x_m, y_i) \end{pmatrix}, \\ L = \frac{1}{h^2} \begin{pmatrix} T & I & & & \\ I & T & I & & \\ & \ddots & \ddots & \ddots & \\ & & & I & T \end{pmatrix}, \quad T = \begin{pmatrix} -4 & 1 & & & \\ 1 & -4 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & & 1 & -4 \end{pmatrix}, \end{gathered} \tag{3}$$

so that L is a tridiagonal block matrix (I is the $m \times m$ identity matrix) and T is an $m \times m$ tridiagonal matrix.

This method can become cumbersome to create and manipulate for small grid spacing ($h \to 0$) as the dimension m^2 scales inversely with h^2. Although this scheme is second-order accurate, the condition number of the matrix L is $O\left(\frac{1}{h^2}\right)$ and so L is very ill-conditioned as $h \to 0$ [19]. Thus a small perturbation can lead to drastic errors [9].

Computational framework solves the tedious issue of manually creating these large systems – human-readable code reflecting the PDE in Equation 1 can be interpreted by a computer, which generates the relevant matrix systems required to solve the PDE. Examples of such frameworks include Devito [18], which employs finite difference methods; FEniCS [28] and Firedrake [26], which employ finite element methods; and Dedalus [5], which employs spectral methods. These different methods for solving PDEs aim to improve on finite differencing.

The finite element method (FEM) is a natural extension of finite difference [17]. Finite difference discretises derivatives using Taylor-like expansions as in Equation 2, whereas FEM creates piecewise functions on a mesh [17]. The PDE can be discretised using these piecewise functions, and a matrix system can be formed using the Galerkin method [22] (Section 2.2.2).

On the other hand, spectral methods rely on a set of basis functions (similar to the piecewise functions of FEM) and aim to solve PDEs by finding coefficients that express the solution in terms of these basis functions [5]. The solution of a PDE can be written as

$$\psi(x) = \sum_{n=0}^{\infty} a_0 \varphi_n(x),$$

where φ_n are the chosen basis functions. For example, if $\varphi_n(x) = x^n$, the sum is a Taylor expansion of ψ.

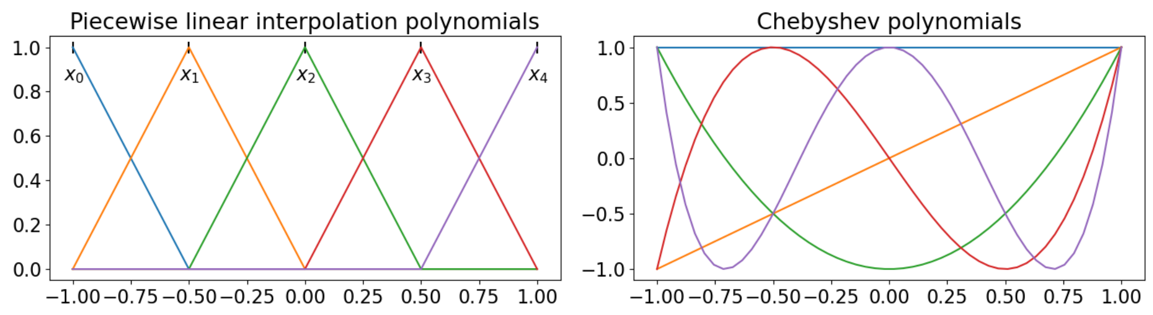

Figure 2: First five piecewise linear interpolation polynomials on the grid points $\{-1, -0.5, 0, 0.5, 1\}$ for FEM and first five Chebyshev polynomials for the spectral method.

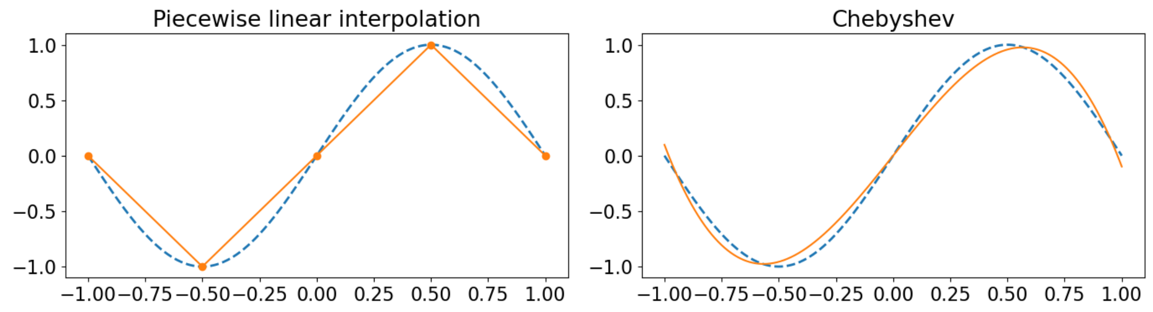

Figure 3: Approximations of $\sin(\pi x)$ on the interval $[-1, 1]$ when written in terms of the first five piecewise linear interpolation basis functions for FEM and first five Chebyshev basis functions for the spectral method. In Section 3.8, we see that 20 Chebyshev polynomials is sufficient to approximate $\sin(\pi x)$ with machine precision errors.

Boyd [4] explains the difference between these methods: FEM makes a grid on the problem domain, then chooses local basis functions to approximate the solution; whereas spectral methods do not need a grid, and use global basis functions. The advantages of FEM are that they create a sparse matrix problem and have easily customisable domain shapes. These come at the cost of a lower accuracy. Spectral methods have better accuracy at the cost of requiring a very smooth and regular domain [4].

For example, a typical set of basis functions for FEM over the grid points $\{x_0, \dots, x_m\}$ of the interval $[x_0, x_m] = [-1, 1]$ are the piecewise linear interpolation Lagrange polynomials [17], given by

$$L_n(x) = \prod_{i \neq n,\, 0 \leq i \leq m} \frac{x - x_i}{x_n - x_i}.$$

Typical basis functions over $[-1, 1]$ for the spectral method are the Chebyshev polynomials [5] given by (Section 2.1.3)

$$T_n(x) = \cos(n \arccos(x)).$$

These two sets of basis functions are plotted in Figure 2. Expressing a function in a basis typically involves integration via the least squares method (Equation 4). Since the FEM basis functions have compact support, the integration is local, whereas the spectral basis requires global integration. This makes integration for FEM quicker and results in a sparse mass matrix (Equation 12), whereas the spectral method gives dense mass matrices. A disadvantage of the FEM basis is that functions in the function space are usually not smooth. In contrast, spectral methods yield infinitely differentiable functions, as seen in Figure 3. This impacts accuracy [4] and can make spectral methods a favourable choice.

In this dissertation, we solve PDEs with spectral methods using Dedalus in Python [5]. The main result will be solving the Stommel–Munk problem, a time-dependent PDE, on a disc.

1.2 Outline of dissertation

This dissertation is split into sections that build up towards implementing the Stommel–Munk problem in Dedalus. In Section 2, we introduce the core concepts of spectral methods by following [4], including discussions on basis choices and imposing boundary conditions. We then consider a simple example in Section 3, where we solve a Poisson equation on a square domain by hand to demonstrate a spectral method, as well as a thorough discussion on the Dedalus implementation of this problem.

To overcome the limitations of Dedalus, we implement a circular domain using polar coordinates in Section 4. We also discuss the accuracy of Dedalus.

In Section 5, we introduce a time dependence, and discuss methods for timestepping in initial value problems (IVPs) by following [19]. This theory is used in Section 6, where we solve an advection equation on a disc using Dedalus. We create a time-dependent PDE solver class that can be used to create subclasses to solve various different IVPs. The full code is presented in Appendix C.

In section 7, we derive the Stommel–Munk problem following [31]. This is then solved in Dedalus using the IVP class defined in Section 6.

Throughout this dissertation, we assume that all domains are sufficiently nice (e.g. open, bounded and connected), and that functions are sufficiently smooth. A summary of notation used can be found in Appendix B.

2 Spectral methods

In this section, we present a brief overview on spectral methods based on Chapters 1-4, 6 and 21 of Boyd's book [4] and supplemented by [5] and [20, Kang, Suh. Spectral Methods].

Let Ω be an open, bounded, connected and sufficiently regular domain in a Hilbert space. Define a suitable inner product between two functions u, v over a domain Ω by

$$\langle u, v\rangle = \int_\Omega \rho(x)\overline{u(x)}v(x)\,\mathrm{d}x,$$

where $\bar{\cdot}$ is complex conjugation and ρ is a given weight function that is positive everywhere and satisfies [15, Süli. Numerical Solution of PDEs]

$$\int_\Omega \rho(x)|x|^i\,\mathrm{d}x < \infty, \quad i \in \mathbb{Z}_{\geq 0}.$$

The inner product $\langle\cdot,\cdot\rangle$ induces the weighted L^2-norm

$$\|u\|^2 = \langle u, u\rangle = \int_\Omega \rho(x)|u(x)|^2\,\mathrm{d}x.$$

Let $\mathfrak{B} = \{\varphi_n(x)\}_{n\in\mathbb{N}}$ be a complete orthogonal basis of the function space of Ω satisfying given boundary conditions. Without loss of generality, it is possible to scale this basis to be normal. Orthonormality ensures that for the given inner product,

$$\langle \varphi_n, \varphi_m\rangle = \delta(m-n) = \begin{cases} 1 & \text{if } n = m, \\ 0 & \text{if } n \neq m, \end{cases}$$

where δ is the Kronecker delta function ($\delta(0) = 1$ and is 0 everywhere else).

The goal of spectral methods is to approximate a function f using the orthogonal basis $\mathfrak{B}$ by computing coefficients $f_n^\varphi := \langle\varphi_n, f\rangle$ so that the N-th order approximation of f with respect to $\mathfrak{B}$ given by [5]

$$f_N(x) := \sum_{n=0}^{N} f_n^\varphi \varphi_n(x) \tag{4}$$

converges to f, i.e. $\lim_{N\to\infty} f_N(x) = f(x)$ (where convergence is taken in an appropriate norm).

Let $Lu(x) = f(x)$ be a differential equation with operator L on the domain Ω (e.g. for Poisson in Equation 1, $L = \nabla^2$ and $u = \psi$). To measure the residual of the coefficient choices u_i^φ, define the residual function as [4]

$$R(x, u_1^\varphi, \ldots, u_N^\varphi) := Lu_N(x) - f(x).$$

Since this residual is 0 precisely when $Lu_N(x) = f(x)$, we aim to minimise this magnitude of the residual when varying the coefficients of u. Different spectral methods choose different minimisation strategies [4].

Although not necessary, spectral methods work best when the true solution is smooth [4]. In the next sections, we outline which basis to choose, how to ensure boundary conditions are satisfied and how to find the optimal coefficients of u.

2.1 Basis choice

There are many possible choices for the basis $\mathfrak{B}$ to optimise the rate of convergence of f_N. The simplest choice is the power series basis $\{x^n : n \in \mathbb{Z}_{\geq 0}\}$, but this is not orthogonal and does not have good numerical conditioning [4]. Another is a Fourier series basis $\{\exp(inx) : n \in \mathbb{Z}\}$. The Chebyshev polynomial basis $\{\cos(n \arccos(x)) : n \in \mathbb{Z}_{\geq 0}\}$ is the gold standard of spectral

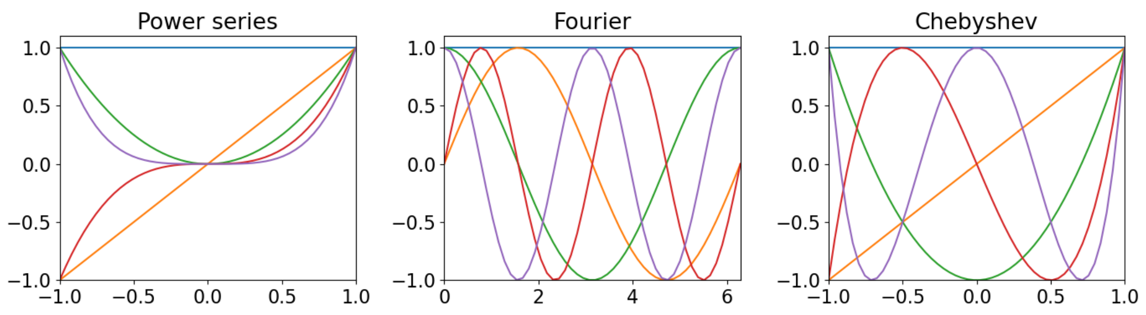

Figure 4: First few basis functions for the power, Fourier and Chebyshev bases.

methods and is almost always the best choice [4]. The standard Fourier basis is useful for periodic boundary conditions [5]. Figure 4 shows the first few functions for each of these bases.

2.1.1 Power series basis

A smooth function f can be written as a power series

$$f(x) = \sum_{n=0}^{\infty} a_n x^n.$$

The coefficients may be found in the usual way by performing a Taylor expansion. Alternatively, given an inner product, we may solve a mass matrix system as in Equation 12.

2.1.2 Fourier series basis

The 2π-periodic Fourier series basis functions are trigonometric polynomials of the form $\exp(inx)$ for $n \in \mathbb{Z}$ [4]. Then any function f can be written as

$$f(x) = \sum_{n=-\infty}^{\infty} c_n \exp(inx), \quad c_n = \frac{1}{2\pi}\int_{-\pi}^{\pi} f(x)\exp(-inx)\,\mathrm{d}x.$$

This basis is clearly orthogonal in the L^2 inner product with as $\frac{1}{2\pi}\int_{-\pi}^{\pi}\exp(inx - imx) = 0$ if $n \neq m$, and 1 if $n = m$.

By using the identities

$$\cos(x) = \frac{1}{2}\left(\exp(ix) + \exp(-ix)\right), \quad \text{and} \quad \sin(x) = \frac{1}{2i}\left(\exp(ix) - \exp(-ix)\right),$$

the Fourier series of a function f can also be written explicitly with sines and cosines [4] as

$$f(x) = a_0 + \sum_{n=1}^{\infty} a_n \cos(nx) + \sum_{n=1}^{\infty} b_n \sin(nx), \tag{5}$$

where the coefficients are given by

$$\begin{aligned} a_0 &= \frac{1}{2\pi}\int_{-\pi}^{\pi} f(x)\,\mathrm{d}x, \\ a_n &= \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\cos(nx)\,\mathrm{d}x, \\ b_n &= \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\sin(nx)\,\mathrm{d}x. \end{aligned}$$

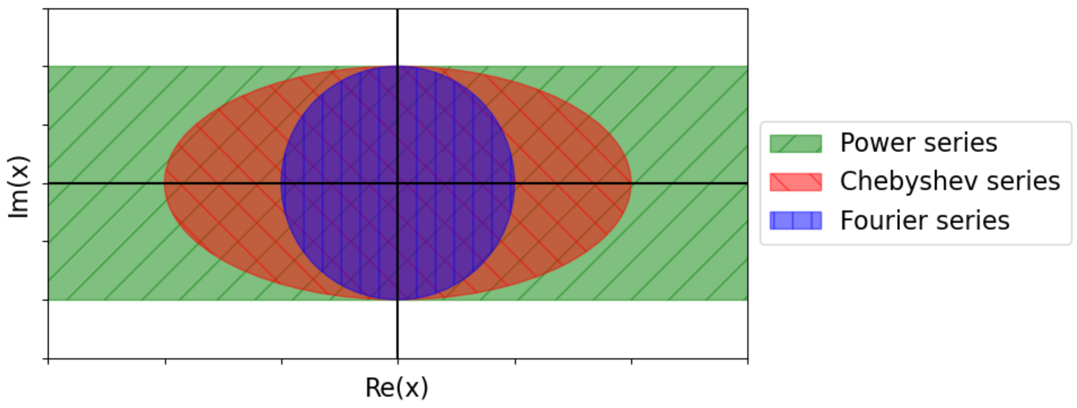

Figure 5: Domain of convergence of various bases on the complex plane. Figure adapted from [4, Fig 2.12].

2.1.3 Chebyshev basis

The Chebyshev basis are trigonometric polynomials $T_n(x) = \cos(n \arccos(x))$ for $n \in \mathbb{Z}_{\geq 0}$ on the interval $[-1, 1]$, which geometrically is the projection of $\cos(nx)$ from the cylinder to the plane [5]. The basis is orthogonal with respect to the weighted L^2 inner product with $\rho(x) = \frac{1}{\sqrt{1-x^2}}$ [4]. The Chebyshev series expansion of a function f is

$$f(x) = \sum_{n=0}^{\infty} f_n^{\varphi} T_n(x), \quad f_n^{\varphi} = \int_{-1}^{1} \frac{f_n^{\varphi}(x) T_n(x)}{\sqrt{1-x^2}} \, \mathrm{d}x.$$

Explicitly, the first few Chebyshev polynomials are

$$\begin{aligned} T_0(x) &= 1, \\ T_1(x) &= x, \\ T_2(x) &= 2x^2 - 1, \\ T_3(x) &= 4x^3 - 3x, \\ T_4(x) &= 8x^4 - 8x^2 + 1. \end{aligned}$$

2.1.4 Convergence

The domain of convergence of an infinite sum $f(x) = \sum f_n^{\varphi} \varphi_n(x)$ are the values of x (in the complex plane) such that the sum is finite. A theorem from Boyd [4, Theorem 2, Section 2.7] states the following typical convergence domains with a suitable inner product.

- Power series converges in a disc.
- Fourier series converges in an infinite strip symmetric about the real-axis.
- The Chebyshev polynomials converge in an ellipse with foci on the real axis at ± 1.

From Figure 5, we can clearly see that the domain of convergence of the power series basis is typically the smallest, and thus is not a good choice. Fourier should be chosen for problems with periodic boundary conditions, and Chebyshev for any other case [4]. Aside from having good convergence, the orthogonality of the Chebyshev and Fourier bases also simplifies calculations as seen in Section 3.2.

2.2 Method of weighted residuals

Now that we can choose a basis $\mathfrak{B} = \{\varphi_i\}$, we can solve a PDE with no boundary conditions by choosing the coefficients u_i^{φ} so that the residual is minimised. This can be done using the method of weighted residuals [20, Kang, Suh. Spectral Methods]. For all $i \in \{0, 1, \dots, N\}$, define test functions w_i such that

$$\langle w_i, R(x, u_1^\varphi, \dots, u_N^\varphi) \rangle = 0. \tag{6}$$

There are different choices we can make for the test functions, the main traditional methods being collocation and Galerkin.

2.2.1 Collocation method

The collocation method (also known as pseudospectral [4]) uses the Kronecker delta $w_i(x) := \delta(x - x_i)$, where the x_i are chosen and are called the collocation points. Substituting the weights into Equation 6 gives the constraints

$$Lu_N^\varphi(x_i) = f(x_i).$$

Thus this method minimises the residuals by minimising at the collocation points. The collocation method is effective in many situations and so is the most common polynomial spectral method [5].

Whilst these conditions are easy to enforce, the dense matrices produced are complicated to work with [5].

2.2.2 Galerkin method

The Galerkin method uses the basis functions as the test functions $w_i(x) := \varphi_i(x)$. Substituting into Equation 6,

$$\langle \varphi_i, Lu_N - f \rangle = \langle \varphi_i, Lu_N \rangle - \langle \varphi_i, f \rangle = 0. \tag{7}$$

Fix an $N + 1$ as the number of basis functions to use computationally and suppose that the operator L is linear. Then Equation 7 becomes

$$\langle \varphi_i, Lu_N \rangle = \sum_{n=0}^{N} u_n^\varphi \langle \varphi_i, L\varphi_n \rangle = \langle \varphi_i, f \rangle.$$

The Galerkin method reduces to an $(N + 1) \times (N + 1)$ matrix problem [4]:

$$L\mathbf{u} = \mathbf{f}, \quad L_{ij} = \langle \varphi_i, L\varphi_j \rangle, \quad u_i = u_i^\varphi, \quad f_i = \langle \varphi_i, f \rangle, \quad i = 0, \dots, N.$$

The problem then can be solved by evaluating the entries of L and f by computing the inner products, then solving the matrix system via usual methods (e.g. LU factorisation if N is small). The solution can then be found as $u_N = \sum_{n=0}^{N} u_n^\varphi \varphi_n$.

The hardest part of the Galerkin method is the integration when computing the inner products, and the accuracy is slightly worse than collocation [4].

2.3 Imposing boundary conditions

Suppose we now have a PDE with the Dirichlet boundary condition $\partial\Omega = 0$. There are two ways to ensure that the solution of this PDE satisfies these boundary conditions [4]. Either impose additional constraints

$$\sum_{n=0}^{\infty} u_n^\varphi \varphi_n(\partial\Omega) = 0,$$

or choose the basis functions φ_n carefully so that they satisfy the boundary conditions. The first is the tau method, and the latter is basis recombination [4].

2.3.1 Basis recombination

An example of basis recombination is modifying the Chebyshev basis $\{T_n\}$ in Section 2.1.3 to get a new non-orthogonal basis $\{\varphi_n\}$ satisfying $\varphi_n(-1) = \varphi_n(1) = 0$ where [4]

$$\varphi_{2n} = T_{2n} - T_0, \quad \varphi_{2n+1} = T_{2n+1} - T_1, \quad n \geq 1.$$

See Section 3.2 for a worked example. The disadvantage of basis recombination is that, in general, a suitable new basis φ_n is hard to find, especially when the boundary conditions are complicated, and in general will not be orthogonal. Non-orthogonal bases lead to dense mass matrices (Equation 12).

2.3.2 Lifting

Suppose the boundary conditions of a PDE $Lu(x) = f(x)$ were not homogeneous (that is, not zero everywhere on $\partial\Omega$). Then let $\hat{u}$ be any smooth function satisfying the boundary conditions. Define a new problem in v using the substitution $u = v + \hat{u}$ to get the PDE

$$Lv(x) = f(x) - L\hat{u}(x)$$

with homogeneous boundary conditions (i.e. $\partial\Omega = 0$). We can then use basis recombination to get an appropriate basis.

2.3.3 Tau method

The tau method is an alternative to basis recombination [4], and is the method used by Dedalus [5]. The tau method uses the test functions $w_i = T_i$ where T_i are the Chebyshev polynomials from Section 2.1.3. So, the tau method is a modified Galerkin method and coincides when the chosen spectral basis $\mathfrak{B}$ is Chebyshev [4].

The idea of the tau method is to solve a slightly perturbed differential equation

$$Lv(x) + \tau\varepsilon(x) = f(x),$$

where $\varepsilon(x)$ is a specified perturbation [5]. A usual choice is $\varepsilon = T_N$, the highest degree basis function used in the approximation. The hope is that if this perturbation is small, then the perturbed solution v is a good approximation to the true solution u.

Consider the differential equation

$$u_x - u = 0, \quad u(0) = 1, \quad 0 \leq x \leq 1.$$

Following [4, Chapter 21], the residual function is a polynomial of degree N (with $N+1$ coefficients), but the extra boundary conditions imposes one extra constraint, so that there are $N+2$ equations with only $N+1$ variables (coefficients of u_N). Thus, we perturb the system to get

$$v_x + v = \tau T_N(x), \quad v(-1) = 1.$$

The Galerkin method in Section 2.2.2, determines the first N rows of an $(N+1) \times (N+1)$ system. The final row arises from the tau factor. When solved, this system determines the coefficients of v independent of τ. When using one tau term (i.e. one boundary condition), the matrix equation can be solved in $O(N^2)$ time by using canonical polynomials (much less than $O(N^3)$ of normal matrix inversion) [25][4]. The matrix arrising from the tau method is dense, but the method can be modified to produce sparse matrices [5].

Adding more boundary conditions involves more tau factors. For each boundary condition, one extra row is added to the Galerkin matrix system.

2.4 Computing errors

Now that we know how to solve a PDE using spectral methods, we can evaluate its effectiveness. Suppose a problem over a domain Ω has solution $u(\mathbf{x})$. Let $\hat{u}(\mathbf{x})$ be an approximation of this solution (as a result of a spectral method or otherwise). We can evaluate the effectiveness of this approximation by computing the error

$$R(\mathbf{x}) = U(\mathbf{x}) - u(\mathbf{x}).$$

The magnitude of error can be computed using any appropriate function norm [19], for example, the L^2-norm

$$\|R\|_2 = \left(\int_\Omega |R(\mathbf{x})|^2 \, \mathrm{d}\mathbf{x} \right)^{\frac{1}{2}} = \left(\int_\Omega |U(\mathbf{x}) - u(\mathbf{x})|^2 \, \mathrm{d}\mathbf{x} \right)^{\frac{1}{2}}. \tag{8}$$

2.5 Dedalus implementation

Given a discretised linear boundary value problem (LBVP) arising from Section 2.3.3 with an appropriate amount of tau factors (an example is given in Section 3.5), Dedalus solves the LBVP as follows [5]:

1. The LBVP is written in the form $L\mathbf{x} = F$, where L is a matrix of operators on the variables $\mathbf{x}$ and $L\mathbf{x}$ is strictly linear. Each row of the matrix formula is an equation (either part of the defining PDEs or a boundary condition). The number of (linearly independent) equations must be the same as the number of problem variables.
2. Dedalus breaks the problem down into smaller problems called pencils $L_p\mathbf{x}_p = F_p$ (see the example in Section 3.2).
3. For each pencil matrix equation, Dedalus converts the dense matrix L_p into a sparse and banded matrix $\tilde{L}_p = P_p^L L_p P^R$ corresponding to the equivalent matrix problem $\tilde{L}_p\tilde{\mathbf{x}_p} = \tilde{F}_p$, where $\tilde{\mathbf{x}_p} = \left(P^R\right)^{-1} \mathbf{x}_p$ and $\tilde{F}_p = P_P^L F_p$. This is done by applying the matrices P_p^L and P^R called the left and right preconditioning matrices respectively [5].
4. This sparse system can be efficiently solved for $\tilde{\mathbf{x}}_p$.
5. Applying the right preconditioning matrix to $\tilde{\mathbf{x}}_p = \left(P^R\right)^{-1} \mathbf{x}_p$ yields $\mathbf{x}_p$.
6. The solution $\mathbf{x}$ can be retrieved by evaluating the pencil.

More details and specific examples are in subsequent sections.

3 Poisson equation in Dedalus

In this section, we outline how spectral methods and Dedalus works with an explicit example. We build and solve the following simple Poisson equation as a toy example. Let $\Omega = \{(x, y) : -1 \leq x, y \leq 1\}$ be a square and define the problem

$$\begin{cases} \nabla^2\psi(x,y) = -2\pi^2\sin(\pi x)\sin(\pi y) & \text{in } \Omega, \\ \psi \text{ periodic} & \text{along the } x\text{-axis}, \\ \psi(x,-1) = \psi(x,1) = 0 & \text{along the } y\text{-axis}, \end{cases} \tag{9}$$

so that the boundary conditions are Dirichlet at the horizontal boundaries and periodic at the vertical boundaries. The exact solution is

$$\psi(x,y) = \sin(\pi x)\sin(\pi y).$$

We will approximate a solution by hand using spectral methods following Section 2 with basis recombination, and mention why the tau method may be more favourable. Dedalus code snippets using the tau method will then be provided. We will check the relationship between the error and the number of basis functions, then compare the spectral method with the 5-point stencil finite differencing method mentioned in Section 1.1.

The code in this section is adapted from the `poisson.py` example in the Dedalus documentation [6]. To use Dedalus on Windows, see Appendix C.1. The necessary packages are in Appendix C.2.

3.1 Creating the bases

Since the x-coordinates are periodic, we use the Fourier basis along the x-axis. For the y-axis, we use the Chebyshev basis. Explicitly, the x basis functions written in their real form in Equation 5 (normalised and rescaled to have a period of 2) are

$$\xi_0(x) = \frac{1}{\sqrt{2}}, \quad \xi_n(x) = \sin(n\pi x), \quad \xi_{-n}(x) = \cos(n\pi x), \quad n \in \mathbb{Z}_{>0}.$$

And the y basis functions are the Chebyshev polynomials from Section 2.1.3,

$$T_0(y) = 1, \quad T_1(y) = y, \quad T_2(y) = 2y^2 - 1, \quad T_3(y) = 4y^3 - 3y, \ldots.$$

Note that the x basis is orthonormal with respect to the unweighted L^2 inner product, and the y basis is orthogonal with respect to the weighted L^2 inner product with $\rho(y) = \frac{1}{\sqrt{1-y^2}}$.

Numerically, we use 256 basis functions in each direction.

```python
Nx, Ny = 256, 256
```

We now define our coordinate system, and the x and y bases each on the interval $[-1, 1]$. All data is stored in double precision (`np.float64`).

```python
coords = d3.CartesianCoordinates('x', 'y')
dist = d3.Distributor(coords, dtype=np.float64)
xbasis = d3.RealFourier(coords['x'], size=Nx, bounds=(-1, 1))
ybasis = d3.Chebyshev(coords['y'], size=Ny, bounds=(-1, 1))
```

Next, create a field for ψ, the function we are solving for.

```python
psi = dist.Field(name='psi', bases=(xbasis, ybasis))
```

3.2 Solving using basis recombination

We now demonstrate how to use spectral methods with basis recombination by hand. To construct a suitable basis for the y basis, note that for all $n \in \mathbb{Z}_{\geq 0}$, $T_{2n}(\pm 1) = 1$ and $T_{2n+1}(\pm 1) = \pm 1$

[4]. Thus let the new y basis functions be

$$\varphi_{2n+1} = T_{2n+1} - T_1, \quad \varphi_{2n} = T_{2n} - T_0, \quad n \in \mathbb{Z}_{\geq 1}.$$

Note that this basis for functions satisfying $f(1) = f(-1) = 0$ is neither orthogonal with the standard unweighted L^2 inner product nor the weighted inner product with $\rho(y) = (1-y^2)^{-\frac{1}{2}}$. For simplicity, we use the unweighted L^2 inner product throughout this section. Note that if the boundary conditions were nonzero, we would have to use lifting from Section 2.3.2 to force zero boundary conditions.

Here, we use three basis functions in each direction for our calculations. Explicitly,

$$\xi_0(x) = \frac{1}{\sqrt{2}}, \tag{10a}$$

$$\xi_1(x) = \sin(\pi x), \tag{10b}$$

$$\xi_{-1}(x) = \cos(\pi x), \tag{10c}$$

$$\varphi_2(y) = 2y^2 - 2, \tag{10d}$$

$$\varphi_3(y) = 4y^3 - 4y, \tag{10e}$$

$$\varphi_4(y) = 8y^4 - 8y^2. \tag{10f}$$

Now, to expand the forcing term $f(x,y) = -2\pi^2 \sin(\pi x)\sin(\pi y)$ of Equation 9 in terms of our basis functions, we need the coefficients of $\xi_i \varphi_j$ for every combination of i, j. For simplicity, we fully decouple the two bases so that we can multiply the coefficients of $\sin(\pi x)$ and $\sin(\pi y)$ in the two bases. This is valid since we can separate

$$\langle f, \xi_i\varphi_j\rangle = -2\pi^2 \int_\Omega \sin(\pi x)\sin(\pi y)\xi_i(x)\varphi_j(y) = -2\pi^2 \int_{-1}^{1} \sin(\pi x)\xi_i(x)\,\mathrm{d}x \int_{-1}^{1} \sin(\pi y)\varphi_j(y)\,\mathrm{d}y.$$

Clearly, the x component of f is

$$\sin(\pi x) = \xi_1(x). \tag{11}$$

Since $\{\varphi_i\}$ are not orthogonal, we cannot simply determine the coefficients of Equation 4 using $\langle \varphi_n, g\rangle$, where $g(y) = \sin(\pi y)$. Instead, we require a solution to the matrix equation (using the three basis functions of Equation 10),

$$M\mathbf{a} = \mathbf{b}, \quad M = \begin{pmatrix} \langle\varphi_2,\varphi_2\rangle & \langle\varphi_2,\varphi_3\rangle & \langle\varphi_2,\varphi_4\rangle \\ \langle\varphi_3,\varphi_2\rangle & \langle\varphi_3,\varphi_3\rangle & \langle\varphi_3,\varphi_4\rangle \\ \langle\varphi_4,\varphi_2\rangle & \langle\varphi_4,\varphi_3\rangle & \langle\varphi_4,\varphi_4\rangle \end{pmatrix}, \quad \mathbf{a} = \begin{pmatrix} g_2^\varphi \\ g_3^\varphi \\ g_4^\varphi \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} \langle\varphi_2, g\rangle \\ \langle\varphi_3, g\rangle \\ \langle\varphi_4, g\rangle \end{pmatrix}. \tag{12}$$

The matrix M with entries $M_{ij} = \langle\varphi_i, \varphi_j\rangle$ is called the mass matrix [4]. Note that if the basis is orthonormal, then the mass matrix M is the identity, and if the basis is orthogonal, then M is diagonal. Thus it is desirable to have an orthogonal/orthonormal basis.

Working out the inner products, Equation 12 becomes

$$\begin{pmatrix} \frac{64}{15} & 0 & \frac{256}{105} \\ 0 & \frac{256}{105} & 0 \\ \frac{256}{105} & 0 & \frac{1024}{315} \end{pmatrix} \begin{pmatrix} g_2^\varphi \\ g_3^\varphi \\ g_4^\varphi \end{pmatrix} = \begin{pmatrix} 0 \\ -\frac{48}{\pi^3} \\ 0 \end{pmatrix}.$$

We can either solve the matrix system directly (by inverting the mass matrix, Gaussian elimination or otherwise); or we could orthogonalise M using Gram-Schmidt [21] or by performing a Cholesky decomposition [4]. As the dimension of M is small here, we choose the solve the matrix system directly (although, in practice M is dense and so orthogonalising can be more useful and faster [4]). Thus the vector of coefficients $\mathbf{a}$ is

$$\mathbf{a} = \begin{pmatrix} g_2^{\varphi} \\ g_3^{\varphi} \\ g_4^{\varphi} \end{pmatrix} = \begin{pmatrix} 0 \\ -\frac{315}{16\pi^3} \\ 0 \end{pmatrix}.$$

Note that the even coefficients are always zero as g is an odd function. So,

$$g(y) = \sin(\pi y) \approx -\frac{315}{16\pi^3}\varphi_3(y).$$

Thus f expressed in terms of the six basis functions in Equation 10 is

$$f(x,y) \approx \frac{315}{8\pi}\xi_1(x)\varphi_3(y).$$

It remains to solve the problem using the Galerkin method from Section 2.2.2. Every combination $\xi_i\varphi_j$ of Equation 10 gives a total of 9 coupled basis functions. Thus we have the 9×9 matrix equation $L\mathbf{u} = \mathbf{f}$, where L is a matrix with entries $\langle \xi_i\varphi_j, \nabla^2\xi_k\varphi_m\rangle$ (with the unweighted L^2 inner product), $\mathbf{u}$ is a vector of coefficients to be found, and $\mathbf{f}$ are the coefficients of the forcing term f written in terms of the 9 basis functions. We use the lexicographical ordering $\xi_0\varphi_2, \xi_0\varphi_3, \xi_0\varphi_4, \xi_1\varphi_2, \ldots, \xi_{-1}\varphi_4$. The second derivatives of the basis functions are

$$\begin{aligned}
\xi_0''(x) &= 0,\\
\xi_1''(x) &= -\pi^2\sin(\pi x) = -\pi^2\xi_1,\\
\xi_{-1}''(x) &= -\pi^2\cos(\pi x) = -\pi^2\xi_{-1},\\
\varphi_2''(y) &= 4,\\
\varphi_3''(y) &= 24y,\\
\varphi_4''(y) &= 96y^2 - 16.
\end{aligned}$$

Since the basis $\{\xi_i\}$ is orthogonal, many of the inner products of the matrix L will be zero. Note that for $\xi_{\pm 1}$,

$$\langle \xi_{\pm1}\varphi_j, \nabla^2\xi_{\pm1}\varphi_m\rangle = \int_\Omega \xi_{\pm1}^2\varphi_j\left(-\pi^2\varphi_m + \varphi_m''\right) = \int_{-1}^1 \varphi_j\left(-\pi^2\varphi_m + \varphi_m''\right)\,\mathrm{d}y.$$

Evaluating the inner products gives

$$L = \begin{pmatrix} A & 0 & 0 \\ 0 & B & 0 \\ 0 & 0 & B \end{pmatrix},$$

$$A = \begin{pmatrix} -\frac{32}{3} & 0 & -\frac{128}{15} \\ 0 & -\frac{128}{5} & 0 \\ -\frac{128}{15} & 0 & -\frac{5632}{105} \end{pmatrix}, \quad B = \begin{pmatrix} -\frac{32}{15}(5+2\pi^2) & 0 & -\frac{128}{105}(7+2\pi^2) \\ 0 & -\frac{128}{105}(21+2\pi^2) & 0 \\ -\frac{128}{105}(7+2\pi^2) & 0 & -\frac{512}{315}(33+2\pi^2) \end{pmatrix}.$$

This is a block diagonal matrix composed of three blocks (arising from the orthogonality of the x basis functions and their nice second derivatives). Note that the diagonal A, B, B are the pencils of L, and we can solve each of these three pencil matrix equations independently. Since the vector $\mathbf{f}$ corresponding to the forcing term f only has a nonzero entry for $\xi_1\varphi_3$, only the second block of L corresponding to ξ_1 gives nonzero coefficients for $\mathbf{u}$, i.e. we solve

$$B\begin{pmatrix} u_{12} \\ u_{13} \\ u_{14} \end{pmatrix} = \begin{pmatrix} 0 \\ \frac{315}{8\pi} \\ 0 \end{pmatrix}.$$

Upon solving this system, we find that $u_{12} = u_{14} = 0$ and $u_{13} = -\frac{33075}{1024\pi(21+2\pi^2)}$, which yields the solution

$$u(x, y) = -\frac{33075}{1024\pi(21 + 2\pi^2)}\xi_1(x)\varphi_3(y).$$

This has an error of around 0.3739. Increasing the number of basis functions decreases the error, as in Section 3.8.

In general, given a non-orthogonal basis, to find a basis representation for a function requires solving a system with the mass matrix by inversion or otherwise. The mass matrix is typically dense, and so this can be computationally costly. Instead, Dedalus chooses to work with orthogonal matrices (so that the mass matrix is diagonal) and uses the tau method to enforce boundary conditions. This increases the size of the Galerkin method matrix slightly as per Section 2.3.3 [5].

3.3 Creating the forcing term

To implement the forcing term $f(x, y) = -2\pi^2 \sin(\pi x) \sin(\pi y)$ in Dedalus, we first define and retrieve values at collocation points stored as `x, y` from `dist.local_grids`. These are used to define f, which Dedalus numerically converts to coefficients of $\xi_i T_j$ in a similar style to Section 3.2, except instead of using a dense mass matrix, it is diagonal as a consequence of the orthogonal basis.

```python
x, y = dist.local_grids(xbasis, ybasis)
f = dist.Field(bases=(xbasis, ybasis))
f['g'] = -2 * np.pi**2 * np.sin(np.pi*x) * np.sin(np.pi*y)
```

3.4 Implementing the tau method

Since there are two boundary conditions for the x-basis, we require two tau factors as in Section 2.3.3.

```python
tau_0 = dist.Field(name='tau_0', bases=xbasis)
tau_1 = dist.Field(name='tau_1', bases=xbasis)
```

A lift function is defined to "lift" the tau factor to the derivative basis $\tau_j U_i(y)$ (where $i = -1, -2$ can be used to select the last two derivative basis functions in the y direction) [6], where U_i is a Chebyshev polynomial of the second kind [4]

$$T_n'(y) = nU_{n-1}(y).$$

Dedalus uses U_{n-1} instead of T_n to improve on the usual dense tau matrix (Section 2.3.3) by making it sparse [5].

```python
def lift(tau, i):
    lift_basis = ybasis.derivative_basis()
    return d3.Lift(tau, lift_basis, i)
```

3.5 Creating the problem

We can now define the LBVP in Dedalus. Let $\hat{\psi}$ and $\hat{f}$ be the discretised forms of ψ and $\hat{f}$ (i.e. expanded in the spectral basis as in Section 2.2.2), so the problem is now

$$\nabla^2\hat{\psi}(x, y) + \tau_0 T_{-1}(y) + \tau_1 T_{-2}(y) = \hat{f}(x, y).$$

This will be solved by forming the matrix in Section 2.3.3, which looks similar to the Galerkin matrix in Section 3.2, except with two extra rows and columns for the two tau factors.[1]

[1]Using `locals()` to import local variables into a class in Python is usually discouraged, but is the suggested method in Dedalus [6].

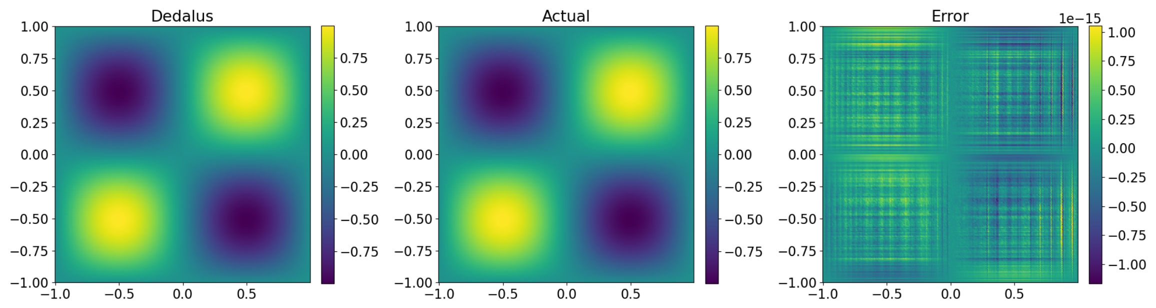

Figure 6: A plot of a solution to the Poisson equation on a square as computed by Dedalus, alongside the true solution and the error.

```python
problem = d3.LBVP([psi, tau_0, tau_1], namespace=locals())
problem.add_equation("lap(psi) + lift(tau_0, -1) + lift(tau_1,-2)
    = f")
problem.add_equation("psi(y=0) = 0")
problem.add_equation("psi(y=1) = 0")
```

3.6 Solving the problem

We now pass the problem into Dedalus' solver which solves the linear system as in Section 2.5.

```python
solver = problem.build_solver()
solver.solve()

x = xbasis.global_grid()
y = ybasis.global_grid()
psi_g = psi.allgather_data('g')
```

3.7 Computing the error

We create a Dedalus field for the actual function. As in Section 3.3, this field is defined on a grid, which Dedalus expresses in terms of the basis functions.

```python
xx, yy = np.meshgrid(x, y)
actual = np.sin(np.pi * xx) * np.sin(np.pi * yy)
actual_field = dist.Field(bases=(xbasis, ybasis))
actual_field['g'] = actual.T
```

The error field is the difference between the computed and actual fields.

```python
R = psi - actual_field
```

3.8 Results

We now plot the results using the a plotting functions defined in Appendix C.3.

```python
fig, axs = plt.subplots(1, 3, figsize=(20, 6))
im_plot(xx, yy, psi_g.T, ax=axs[0], title='Dedalus')
im_plot(xx, yy, actual, ax=axs[1], title='Actual')
im_plot(xx, yy, R.evaluate()['g'].T, ax=axs[2], title='Error')
plt.show()
```

Dedalus gives the results in Figure 6. The errors are very close to machine precision. To compute the L^2-norm of this error as in Equation 8, we may use Dedalus' built in integration `d3.integ`.

```python
np.sqrt(d3.integ(R**2)).evaluate()['g']
```

This gives $\|R\|_2 = 4.93\times10^{-16}$, which has the same order as machine precision $\varepsilon_m = 2.22\times10^{-16}$.

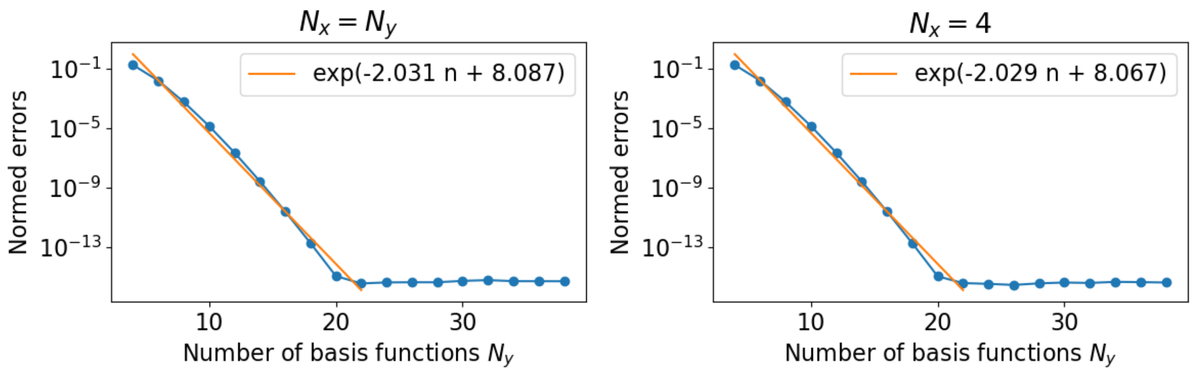

Figure 7: A plot of errors to the Poisson equation on a square when varying the number of basis functions N_y between 4 and 38. The first plot has $N_x = N_y$, and the second holds $N_x = 4$ constant.

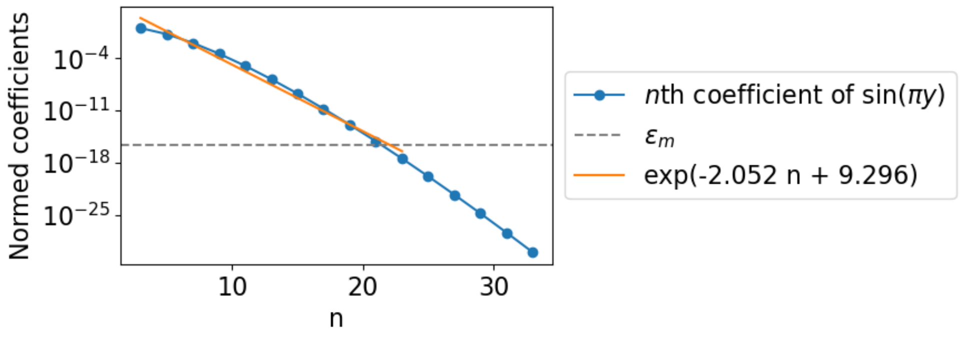

Figure 8: Plot of the normed spectral coefficients of $\sin(\pi y)$ in the Chebyshev basis.

```
np.finfo(np.float64).eps
```

By rerunning the code with varying values for the number of basis functions $N_x = N_y$, we see that around 20 basis functions for this problem is actually sufficient to achieve machine precision errors, as in Figure 7. The errors approximately decrease log-linearly before reaching machine epsilon. Holding $N_x = 4$ constant and varying N_y also yields a very similar error plot. This is because the x component of the solution coincides with a Fourier x-basis function as in Equation 11.

The errors where $N_r \geq 20$ fluctuate around ε_m, and this is called the round-off plateau [4]. The shape of this curve before the round-off plateau approximately follows the largest coefficient of the true solution when written in the basis. Note that sin is odd, so all even coefficients are zero. The odd coefficients are given by [27]

$$\langle \sin(\pi y), T_n(y)\rangle = \int_{-1}^{1} \frac{\sin(\pi y) T_n(y)}{\sqrt{1-y^2}}\,\mathrm{d}y = 2\int_{0}^{1} \frac{\sin(\pi y) T_n(y)}{\sqrt{1-y^2}}\,\mathrm{d}y = \pi J_n(\pi),$$

where J_n is the n-th Bessel function (see Section 4.3). These coefficients are plotted in Figure 8. We see that the coefficients reach machine precision at around $n = 20$, which is the same as the numerical experiment in Figure 7. The log linear fit of the first 20 coefficients is also similar in these two figures. We say that the coefficients experience supergeometric convergence with rate approximately 2 [4].

Note also that for $N_y = 3$ and $N_x = 4$, the error is 0.3852, which is of the same order as basis recombination in Section 3.2.

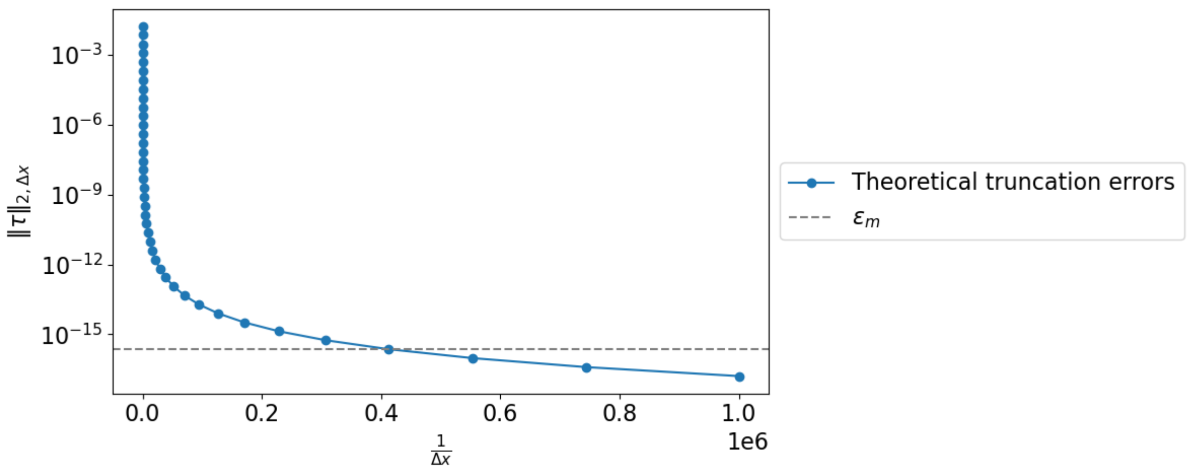

Figure 9: Theoretical truncation errors of the 5-point Laplacian finite difference method on the Poisson equation (Equation 9) versus $\frac{1}{\Delta x}$.

3.9 Comparison to finite difference

Using the 5-point Laplacian method in Equation 3, we get a theoretical local truncation error of [19]

$$\tau_{ij} = \frac{1}{12}(\Delta x)^2(\psi_{xxxx} + \psi_{yyyy}) + O((\Delta x)^4) = \frac{\pi^4}{6}(\Delta x)^2\psi + O((\Delta x)^4),$$

where Δx is the chosen grid spacing for finite differencing and τ_{ij} is the error at $(i\Delta x, j\Delta x)$. Thus the weighted normed truncation error in the L^2 norm is

$$\|\tau_{ij}\|_{2,\Delta x} = \Delta x\|\tau_{ij}\|_2 = \frac{\pi^4}{6}(\Delta x)^3\left(\int \psi^2\right)^{\frac{1}{2}} + O((\Delta x)^5) = \frac{\pi^4}{6}(\Delta x)^3 + O((\Delta x)^5).$$

Thus, to achieve machine precision, we would require $\Delta x = 2.5 \times 10^{-6}$ as in Figure 9. This corresponds to 4×10^5 grid points along both axis. This corresponds to a much larger matrix than the required $20^2 \times 20^2$ Galerkin matrix from spectral methods required for machine precision errors, but the advantage of finite difference is that no integration (numerical or otherwise) needs to be performed.

4 Circular domains in Dedalus

In Section 3, we solved the Poisson equation in a box with periodic boundary conditions along one axis (Equation 9). But unfortunately, Dedalus cannot solve a problem with non-periodic boundary conditions along the whole boundary of the box [6]. Instead, we must rely on Dedalus' ability to solve problems in polar coordinates, where we can specify boundary conditions along the boundary of a circle [5].

In this section, we will construct a Python class to solve problems in a circular domain in a similar style to Section 3. We shall use this framework to solve a Poisson equation with a Bessel solution.

4.1 Poisson equation on a circle

Let $\Omega = \{(x, y) : x^2 + y^2 < L_r\}$, i.e. Ω is the open disc of radius L_r, and the boundary $\partial\Omega$ is the unit circle. Consider the Poisson equation

$$\begin{cases} \nabla^2\psi = q & \text{in } \Omega, \\ \psi = 0 & \text{on } \partial\Omega, \end{cases} \tag{14}$$

where $q(x, y)$ is a smooth forcing function. We use the usual polar substitutions arising from $x = r\cos\varphi$ and $y = r\sin\varphi$. The Laplacian becomes [2]

$$\nabla^2\psi = \psi_{xx} + \psi_{yy} = \psi_{rr} + \frac{1}{r}\psi_r + \frac{1}{r^2}\psi_{\varphi\varphi},$$

and the gradient is

$$\nabla = \begin{pmatrix} \partial_r \\ \frac{1}{r}\partial_\varphi \end{pmatrix}.$$

4.2 Dedalus implementation

We now create a Dedalus solver for Equation 14 given an arbitrary forcing function q and domain radius L_r.

As in Section 3, we present Dedalus code alongside explanations, though we switch to object orientated programming. Class definitions can be found in Appendices C.3 and C.5. The full code is in Appendix C.6.

We first create an abstract base class called `DedalusSolver`, which implements the method `make_space` which defines the space Dedalus will work in, as well as useful plotting methods. A subclass with methods `make_problem` and `solve_problem` must be created to make the problem in Dedalus, then instruct Dedalus to solve. For this section, the subclass `Poisson_Circ` implements these methods.

4.2.1 Creating the space

Within the `DedalusSolver` in Appendix C.5, the method `make_space` creates the basis and domain. Given the radius of Ω as `Lr` and dealias factor `dealias` (see Section 5.6; for now, we use a dealias of 1 which has no effect), we can define the coordinate system and basis.

```python
coords = d3.PolarCoordinates('phi', 'r')
dist = d3.Distributor(coords, dtype=dtype)
disk = d3.DiskBasis(coords, shape=(Nphi, Nr), radius=Lr, dealias=
    dealias, dtype=dtype)
edge = disk.edge
```

Note that contrary to usual conventions, Dedalus has the azimuth as the first coordinate then the radius as the second coordinate. The number of basis functions are `Nphi` and `Nr` in the azimuthal and radial directions respectively. The basis for a disc in polar coordinates is `d3.DiskBasis`, which is a Fourier type basis along the azimuth (since it is 2π periodic) [6]. The radial basis is built from Jacobi polynomials which have similar properties to Bessel functions (Section 4.3) [32]. The disc basis allows for boundary conditions at $r = L_r$ [32].

Now, we define our unknown field `psi`, and collocation grid points `phi, r`.

```
phi, r = dist.local_grids(disk)
psi = dist.Field(name='psi', bases=disk)
```

4.2.2 Creating the problem

As in Section 3, we implement the problem Equation 14 in Dedalus. This is done in the `make_problem` method. The forcing term `q_func` as a function of `phi` and `r` will be given externally later. Recall that mesh values are given to fields using `['g']`.

```
q = dist.Field(bases=disk)
q['g'] = q_func(phi, r, Lr=Lr)  # to be given later
```

Since we have one boundary condition, we need one tau factor.

```
tau_psi = dist.Field(name='tau_psi', bases=edge)
```

The following lift function lifts the tau factor to a multiple of the last function in the derivative basis (Section 3.4).

```
def lift(A):
    lift_basis = disk.derivative_basis(2)
    return d3.Lift(A, lift_basis, -1)
```

Let $\hat{\psi}$ and $\hat{q}$ be discretisation of ψ and q. We can then create Equation 14 with one tau factor. This creates a system with two equations and two unknowns.

$$\begin{cases} \nabla^2\hat{\psi} + \text{lift}_{-1}(\tau_u) = \hat{q} & \text{in } \Omega, \\ \psi = 0 & \text{on } \partial\Omega. \end{cases}$$

```
problem = d3.LBVP([psi, tau_psi], namespace=locals())
problem.add_equation("lap(psi) + lift(tau_psi) = q")
problem.add_equation("psi(r=Lr) = 0")
```

4.2.3 Solving the problem

Finally, we can pass to Dedalus using the method `solve_problem` as in Section 2.5.

```
solver = problem.build_solver()
solver.solve()

phi, r = dist.local_grids(disk)
psig = psi.allgather_data('g')
```

4.3 Bessel function solution

Using the `Poisson_Circ` class, we solve the Equation 14 such that the forcing term q is chosen so that ψ can be described radially by a Bessel function.

A Bessel function $J_n(r)$ is the solutions to the differential equation [2]

$$r^2 J_n'' + r J_n' + (r^2 - n^2) J_n = 0, \quad r > 0,$$

where $n \geq 0$ denotes the order of the Bessel function. When n is an integer, the Bessel function can be written as [2]

$$J_n(r) = \sum_{k=0}^{\infty} \frac{(-1)^k}{k!(k+n)!} \left(\frac{r}{2}\right)^{2k+n}. \tag{15}$$

If n is not an integer, then we can replace $(k+n)!$ with the analytic continuation of the factorial $\Gamma(k+n+1)$ (called the gamma function). The derivative of a Bessel function is [2]

$$J_n'(r) = \frac{1}{2}\left(J_{n-1}(r) - J_{n+1}(r)\right). \tag{16}$$

Since we wish to impose zero Dirichlet boundary conditions at $r = L_R$ (Equation 14), we consider the Bessel function $J_n\left(\frac{\alpha}{r}L_r\right)$, where α is a root of J_n. It can be shown that

$$\psi(r,\varphi) = \sin(n\varphi)J_n\left(\frac{\alpha}{L_r}r\right) \tag{17}$$

describes the surface of a vibrating circular drum of radius L_r [2], and is a smooth function that satisfies $\psi(L_r,\varphi) = 0$. Furthermore, ψ has rotational symmetry around the origin. For ψ to be a solution of the Poisson Equation, we calculate the forcing term using Equation 16,

$$\begin{aligned} q(r,\varphi) = \nabla^2\psi(r,\varphi) =& \frac{\alpha^2\sin(n\varphi)}{4L_r^2}\left(J_{n-2}\left(\frac{\alpha}{L_r}r\right) - 2J_n\left(\frac{\alpha}{L_r}r\right) + J_{n+2}\left(\frac{\alpha}{L_r}r\right)\right) \\ &+ \frac{\alpha\sin(n\varphi)}{2L_r r}\left(J_{n-1}\left(\frac{\alpha}{L_r}r\right) - J_{n+1}\left(\frac{\alpha}{L_r}r\right)\right) \\ &- \frac{n^2\sin(n\varphi)}{r^2}J_n\left(\frac{\alpha}{L_r}r\right). \end{aligned} \tag{18}$$

Thus, $\psi(r,\varphi) = \sin(n\varphi)J_n\left(\frac{\alpha}{L_r}r\right)$ is a solution to

$$\begin{cases} \nabla^2\psi = q & \text{in } \Omega, \\ \psi = 0 & \text{on } \partial\Omega, \end{cases}$$

where the forcing q is as in Equation 18.

4.3.1 Dedalus implementation

We wish to solve the Equation 4.3 with forcing Equation 18. Using the classes defined in Section 4.2, it remains to define the forcing term.

```python
def bessel_q(phi, r, n=3, Lr=1):
    r = r/Lr   # Scale r
    a = sc.jn_zeros(n, 1)[0]
    q = (((a**2)/4) * np.sin(n*phi)) \
        * (sc.jn(n-2, a*r) - 2*sc.jn(n, a*r) + sc.jn(n+2, a*r))\
        + ((a/(2*r)) * np.sin(n*phi)) \
        * (sc.jn(n-1, a*r) - sc.jn(n+1, a*r)) \
        - (((n**2)/(r**2)) * np.sin(n*phi)) * sc.jn(n, a*r)
    return q/Lr**2
```

We can then call the `Poisson_Circ` class. We choose a Bessel function of order 3 on a circle of radius 1 with 256 basis functions in each direction.

```python
n = 3  # Bessel order
Nphi, Nr = 256, 256
Lr = 1

bessel_lap = Poisson_Circ(Nphi, Nr, partial(bessel_q, n=n), Lr=Lr)
```

We also store the true solution (Equation 17) in the class.

```python
def bessel(phi, r, n, Lr=1):
    r = r/Lr   # Scale r
    a = sc.jn_zeros(n, 1)[0]
    z = np.sin(n*phi) * sc.jn(n, a*r)
```

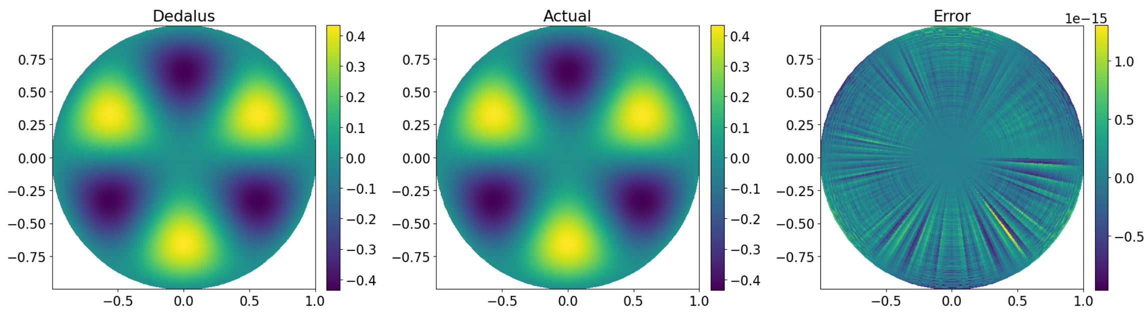

Figure 10: A plot of the Bessel solution to the Poisson equation as computed by Dedalus, alongside the actual solution and the error.

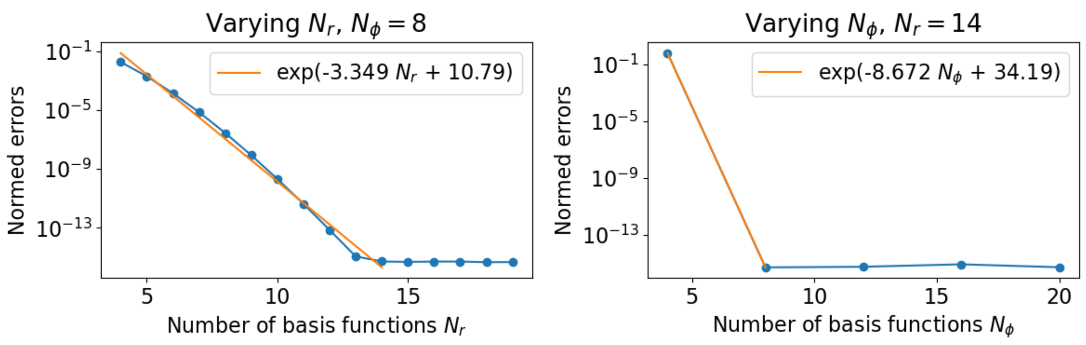

Figure 11: Plots of normed errors when varying the number of basis functions for the Poisson equation on a circular domain. The first plot varies N_r, holding $N_\varphi = 8$ constant, and the second varies N_φ holding $N_r = 14$ constant.

```
    return z


bessel_lap.actual(partial(bessel, n=n))
```

4.3.2 Results

The method `compute_error()` computes the errors as in Section 3.7, then plots them as in Section 3.8. This results in Figure 10 and a normed error of 3.74×10^{-16}, which is close to machine precision.

As in Section 3.8, we can also investigate the effects of varying the basis sizes by using the `error_lists` method, which repeatedly runs the spectral method for different `Nr` and `Nphi`. Note that `Nphi` must be a multiple of 4. The argument of `error_plots` specifies the number of data points to consider for the orange log linear best fit line.

```
bessel_lap.Nr, bessel_lap.Nphi = 15, 8

Nr_list = np.arange(4, 20, 1, dtype=int)
Nphi_list = np.arange(4, 24, 4, dtype=int)

bessel_lap.error_lists(Nphi_list, Nr_list)
bessel_lap.error_plots([11, 2])
```

Figure 11 shows that the errors eventually reach machine precision after $N_r = 14$ and $N_\varphi = 8$. A smaller required number of azimuthal basis functions is expected due to the rotational symmetry of the problem. As in Section 3.8, we see evidence of log-linear behaviour before the round-off plateaus.

5 Initial value problems

In the previous sections, we looked at time-independent PDEs. We now look at PDEs that evolve over time given an initial condition. This is called an initial value problem (IVP). This section follows Chapters 5 to 8 of the book by LeVeque [19].

Consider an IVP $u'(t) = f(u,t)$ with initial condition $u(0) = u_0$. A solution in a neighbourhood of $t = 0$ exists and is unique if f is Lipschitz continuous over the domain Ω and continuous over time [19]. We shall assume that the problem, domain and solution are sufficiently regular. We first look at common methods of timestepping, then discuss their stability. We then see how dealiasing can improve numerical results.

5.1 Forward Euler

Consider the IVP $\partial_t u(t) = f(u,t)$. A simple timestepper is forward Euler – replacing the time derivative in $\partial_t u(t)$ with the forward difference [19]

$$\frac{U^{n+1} - U^n}{\Delta t} = f(U^n, n\Delta t), \quad n \in \mathbb{Z}_{\geq 0},$$

where U^n is a discrete approximation of $u(n\Delta t)$ for a suitably chosen timestep Δt. We can then form a linear system (similar to Equation 3) with rows

$$U^{n+1} = U^n + \Delta t f(U^n), \quad n \in \mathbb{Z}_{\geq 0},$$

to solve for the vector $\left(U^0 \ \cdots \ U^{n+1}\right)^T$.

Other schemes include backward Euler (backwards difference approximation for $\partial_t u$), the trapezoidal Crank-Nicolson method (average of forward and backward Euler), the leapfrog method (a 2-step method) and the general Runge–Kutta methods [19].

One-step methods (e.g. forward Euler) are advantageous over multi-step methods (e.g. leapfrog) as they are self-starting and the timestep can be easily changed any time during simulation [19].

5.2 Linear multi-step methods

Linear multi-step methods (LMMs) is a class of timesteppers used by Dedalus to discretise along time. The forward Euler method is a member of this class. We first define LMMs, then give some examples and investigate their stability.

An LMM has the form

$$\sum_{j=0}^{r} \alpha_j U^{n+j} = \Delta t \sum_{j=0}^{r} \beta_j f(U^{n+j}, (n+j)\Delta t), \quad n \in \mathbb{Z}_{\geq 0}. \tag{19}$$

So U^{n+r} is computed from the r previous steps $U^{n+r-1}, \ldots, U^n$. If $\beta_r = 0$, then the method is explicit, otherwise it is implicit [19].

The Adam-Bashford methods are explicit LMMs with $\beta_r = 0$, $\alpha_r = 1, \alpha_{r-1} = -1$ and $\alpha_j = 0$ for $j < r - 1$ [19], i.e.

$$U^{n+r} = U^{n+r-1} + \Delta t \sum_{j=0}^{r} \beta_j f(U^{n+j}, (n+j)\Delta t), \quad n \in \mathbb{Z}_{\geq 0}.$$

The β_j coefficients are chosen to maximise the order of accuracy, namely order r when the scheme is explicit. This can be done by looking at the local truncation errors [19].

The forward Euler scheme is an explicit one-step Adam-Bashford method with $\beta_0 = 1$ (sometimes called AB1 [6]). The Crank-Nicolson scheme is an implicit one-step LMM Adam-Moulton method with $\beta_0 = \beta_1 = \frac{1}{2}$ [19]. The Adam-Moulton methods are a generalisation of Adam-Bashford, where β_r is allowed to be nonzero.

5.3 Absolute stability

Absolute stability is a property of a PDE and timestepper that yields information on the suitability of the timestepper for the problem [19]. We wish for the method to be stable.

Consider the von Neumann problem

$$u'(t) = \lambda u(t), \quad u(0) = 1, \quad t \geq 0,$$

for a parameter $\lambda \in \mathbb{C}$. Forward Euler over the time dimension with timestep Δt gives

$$U^{n+1} = (1 + \lambda\Delta t)U_n, \quad n \in \mathbb{Z}_{\geq 0}.$$

This recursion is a geometric sequence with ratio $1 + \lambda\Delta t$, and so we expect convergence when $|1+\lambda\Delta t| \leq 1$. When $|1+\lambda\Delta t| \leq 1$, we say that the method is absolutely stable; otherwise, it is unstable. Thus, we have absolute stability when the value of $\lambda\Delta t$ is inside the circle of radius 1 centred at -1 in the complex plane as in Figure 12.

For a general linear PDE, the problem can be represented by a matrix, and λ will instead be a (possibly complex) eigenvalue of the matrix [19] as discussed in Section 5.4. Figure 12 shows the stability regions of $\lambda\Delta t$ in the complex plane for common one-step methods.

In general, an LMM for the von Neumann problem is

$$\sum_{j=0}^{r} \alpha_j U^{n+j} = \Delta t \sum_{j=0}^{r} \beta_j \lambda U^{n+j}, \quad n \in \mathbb{Z}_{\geq 0},$$

which gives

$$\sum_{j=0}^{r} (\alpha_j - \lambda\Delta t \beta_j) U^{n+j} = 0, \quad n \in \mathbb{Z}_{\geq 0}.$$

This is a difference equation, and has a solution (when given initial conditions $U^0, \ldots, U^{r-1}$ for the first r steps) of the form [19]

$$U^n = c_1\zeta_1^n + c_2\zeta_2^n + \cdots + c_r\zeta_r^n,$$

where the c_i are arbitrary constants and ζ_i are the roots of the characteristic polynomial (often called the stability polynomial)

$$\pi(\zeta; \lambda\Delta t) := \sum_{j=0}^{r} (\alpha_j - \lambda\Delta t\beta_j)\zeta^j = (\alpha_r - \lambda\Delta t\beta_r) \prod_{j=1}^{r} (\zeta - \zeta_j).$$

The region of stability for this LMM is the set of points $\lambda\Delta t$ on the complex plane such that the roots of π satisfy

$$\begin{cases} |\zeta_j| \leq 1 & \text{for } j = 1, 2, \ldots, r, \\ |\zeta_j| < 1 & \text{if } \zeta_j \text{ is a repeated root of } \pi. \end{cases}$$

For Euler's method, $\pi(\zeta : \lambda\Delta t) = \zeta - (1 + \lambda\Delta t)$, which retrieves the same circle as before. The characteristic polynomials for common scheme are in LeVeque [19]. Their stability regions are shown in Figure 12. We see that implicit timesteppers have unbounded stability regions, while explicit methods do.

In general, stability regions are not as regular as the circle of forward Euler. Furthermore, high degree polynomials do not usually have analytic solutions. [14, Parshall. The development of abstract algebra]. To numerically find the region of stability from the characteristic polynomial, we mesh a grid on the complex plane for $\lambda\Delta t$. Then for each $\lambda\Delta t$, we can find $\max|\zeta_j|$ using

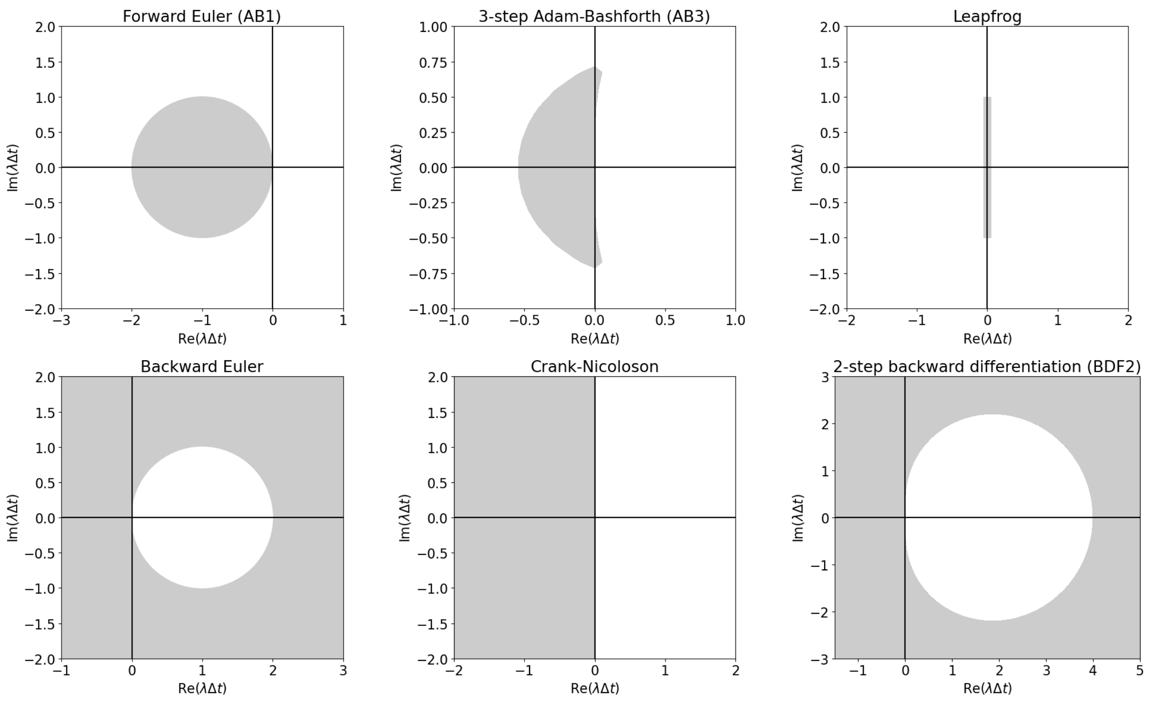

Figure 12: Stability regions for common time schemes. The first row shows explicit schemes, and the second has implicit schemes. Figure adapted from [19, Fig 7.1], [19, Fig 7.2] and [19, Fig 8.5] using code from [24].

numerical methods (e.g. `np.roots`), and test whether this is ≤ 1 or not, corresponding to being inside the region or outside respectively [24].

The stability polynomial of a consistent LMM has a root $\zeta_1 = 1$. So when $\lambda \Delta t$ is near the origin,

$$\zeta_1(\lambda\Delta t) = e^{\lambda\Delta t} + O\left((\lambda\Delta t)^{p+1}\right), \quad \text{as } \lambda\Delta t \to 0,$$

if the scheme is p-th order accurate [19].

5.4 Systems of PDEs

Let $u'(t) = Au(t)$ where $u(t) \in \mathbb{R}^m$ and $A \in \mathbb{R}^{m\times m}$ a matrix. Suppose that A is diagonalisable with eigenvalues λ_j and corresponding linearly independent eigenvectors $\mathbf{r}_j$ for $j = 1, \ldots, m$. Write $A = R\Lambda R^{-1}$ where Λ is the diagonal matrix of eigenvalues, and R the matrix of eigenvectors. Then we can perform a change of variable $v(t) = R^{-1}u(t)$ to get the decoupled system [19]

$$v'(t) = \Lambda v(t).$$

We can then perform our chosen time method to each independent equation separately on the eigenvalues λ_j. For absolute stability, each separate equation needs to be stable, namely $\Delta t\lambda_j$ needs to be inside the region of stability for the scheme for all $j = 1, \ldots, m$; i.e. $\text{argmax}_j |\Delta t \lambda_j|$ must be in the stability region.

5.5 Stiffness

A problem $\partial_t u(t) = f(u,t)$ is called stiff if the solution is perturbed, the perturbed data varies rapidly; concretely, $\partial_u f(t,u) \gg \partial_t(u)$ [19]. For a system of equations, the stiffness ratio is defined to be $\frac{\max|\lambda_j|}{\min|\lambda_j|}$ over the eigenvalues λ_j of the Jacobian $\partial_u f(t,u)$. Schemes with bounded regions of stability (such as forward Euler or explicit Adam-Bashford methods) are not suited for stiff problems, whereas those with unbounded regions of stability (such as Crank-Nicolson and backward Euler) are appropriate for stiff problems [19].

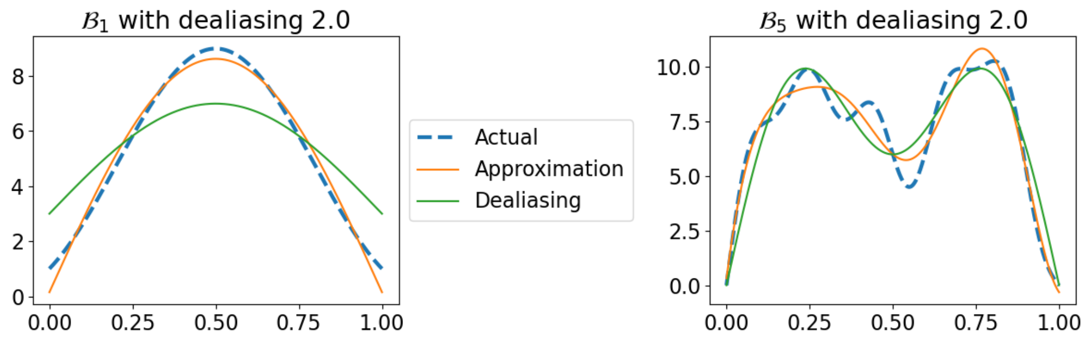

Figure 13: Plots of dealiasing effects. The first plot shows $\psi = (1 + 2\sin(\pi x)^2$ with basis $\mathcal{B}_1$ with dealiasing scale $s = 2$. The dotted curve is the actual function, the orange curve is the least squares approximation with $\mathfrak{B}_1$ without dealiasing, and the green curve is the least squares approximation with dealiasing (i.e. use $\mathfrak{B}_2$, then truncate to $\mathfrak{B}_1$). The second shows $\psi = 10\sin(\pi x) + 4\sin(3\pi x) + 6\sin(6\pi x) + \sin(10\pi x)$ with basis $\mathcal{B}_5$ (i.e. Equation 20 with $n \leq 5$) and dealiasing $s = 2$. Code can be found in Appendix C.10.

5.6 Dealiasing

When viewing a video of a fan or helicopter blades, illusions can occur (Chapter 11 of Boyd [4]). Since a video has a finite number of frames per second (e.g. 30 FPS), the blades may appear to move in the opposite direction, seem stationary, or move slower than they truly are. This is called aliasing [4], in which discrete sampling causes errors. An object rotating faster than 30 revolutions per second in a 30 FPS video will have their frequency aliased to a frequency less than or equal to 30 revolutions per second.

In the case of spectral methods, truncating the number of basis functions used in Equation 4 can introduce discretisation errors [4] as lower degree basis functions cannot resolve high degree behaviour. This causes numerical issues in nonlinear equations [5]. One way to fix these errors is to increase the number of basis functions. But this is not always practical, especially as this increases the computational cost. The second way is to introduce a dealias scaling.

The dealias s transforms the truncated N basis functions to a scaled set of basis functions of size sN [5]. For each intermediate calculation when implementing a spectral method, we use sN basis functions, but only keep the lowest N [4]. As a rule-of-thumb, $s \geq \frac{3}{2}$ is a good choice [5]. Dealiasing should be used when the computed solution is only marginally resolved. But if the solution is poorly resolved, then dealiasing will not help, and the number of basis functions must be increased [4].

For example, consider the following Fourier type basis $\mathcal{B}$ on the interval $[0, 1]$:

$$\varphi_0(x) = 1, \quad \varphi_n(x) = \sin(n\pi x), \quad \varphi_{-n}(x) = \cos(n\pi x), \quad n \in \mathbb{Z}_{>0}. \tag{20}$$

Note that this basis is not orthogonal with respect to the usual L^2 inner product. Consider the zeroth order differential equation

$$\psi(x) = (1 + 2\sin(\pi x))^2 .$$

Then ψ can be expressed exactly in the basis as

$$\psi(x) = 3 + 4\sin(\pi x) - 2\cos(2\pi x) = 3\varphi_0(x) + 4\varphi_1(x) - 2\varphi_{-2}(x).$$

Suppose we use the subset $\mathcal{B}_1 = \{\varphi_0, \varphi_1, \varphi_{-1}\}$ as our spectral basis. Then ψ cannot be expressed exactly. Since the basis is not orthogonal, we form the mass matrix M (Equation 12) to write

ψ in terms of the basis $\mathcal{B}_1$:

$$M = \begin{pmatrix} \langle \varphi_0, \varphi_0 \rangle & \langle \varphi_0, \varphi_1 \rangle & \langle \varphi_0, \varphi_{-1} \rangle \\ \langle \varphi_1, \varphi_0 \rangle & \langle \varphi_1, \varphi_1 \rangle & \langle \varphi_1, \varphi_{-1} \rangle \\ \langle \varphi_{-1}, \varphi_0 \rangle & \langle \varphi_{-1}, \varphi_1 \rangle & \langle \varphi_{-1}, \varphi_{-1} \rangle \end{pmatrix} = \begin{pmatrix} 1 & \frac{2}{\pi} & 0 \\ \frac{2}{\pi} & \frac{1}{2} & 0 \\ 0 & 0 & \frac{1}{2} \end{pmatrix}.$$

So, we get the matrix system

$$M\mathbf{a} = \mathbf{b}, \quad \mathbf{a} = \begin{pmatrix} a_0 \\ a_1 \\ a_{-1} \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} \langle \varphi_0, \psi \rangle \\ \langle \varphi_1, \psi \rangle \\ \langle \varphi_{-1}, \psi \rangle \end{pmatrix} = \begin{pmatrix} 3 + \frac{8}{\pi} \\ 2 + \frac{22}{3\pi} \\ 0 \end{pmatrix}.$$

Solving the matrix system yields $(a_0, a_1, a_{-1}) = (0.147, 8.481, 0)$. This causes the second order mode in ψ to be approximated as a first order mode, leading to the coefficient of φ_1 being larger than expected. Using dealiasing $s = 2$ (so truncating from $\{a_0, a_1, a_2, a_{-1}, a_{-2}\}$) yields the coefficients $(3, 4, 0)$, which captures the behaviour of the lower order modes correctly. This is shown in Figure 13 along with a higher order example.

Note that for a zeroth order differential equations on an orthogonal spectral basis (e.g. the usual Fourier basis on the interval as in Section 2.1.2), dealiasing does not do anything as the mass matrix is always diagonal, and adding more basis functions adds an equation independent to existing rows of the mass matrix equation. Thus in this case, truncation yields the same result as the original mass matrix equation. For higher order differential equations with an orthogonal spectral basis, dealiasing may have an effect as the Galerkin matrix (Section 2.2.2) does not have to be diagonal, so adding extra rows may change the least squares results.

5.7 Dedalus implementation

We now put everything together and discuss how Dedalus uses this theory. Consider an IVP of the form

$$M\partial_t \mathbf{u} + L\mathbf{u} = f(\mathbf{u}, t),$$

where M and L are matrices, $\mathbf{u}$ a vector and $L\mathbf{u}$ is a linear stiff term (i.e. the stiffness ratio of L is small). Then a r-step LMM with timestep Δt for this problem has a similar form to Equation 19 [33]:

$$\sum_{j=0}^{r} \alpha_j M U^{n+j} + \Delta t \sum_{j=0}^{r} \beta_j L U^{n+j} = \Delta t \sum_{j=0}^{r} \gamma_j F^{n+j}, \quad n \in \mathbb{Z}_{\geq 0},$$

where $F^{n+j} = f(U^{n+j}, (n+j)\Delta t)$. By splitting the stiff term away from f we may apply an explicit scheme to the non-stiff side, and a implicit method for the stiff side. This is called an implicit-explicit multistep method (IMEX) [5].

Using $\gamma_r = 0$ (arising from the explicit method) and rearranging yields

$$\left(\frac{1}{\Delta t} a_0 M + b_0 L\right) U^n = \sum_{j=1}^{r} \left(c_j F^{n-j} - \frac{1}{\Delta t} a_j M U^{n-j} - b_j L U^{n-j}\right),$$

where $a_j = \alpha_{s-j}$, $b_j = \beta_{r-j}$ and $c_j = \gamma_{r-j}$ [6].

Dedalus implements the `MultistepIMEX` class which requires the coefficients a_j, b_j, c_j for $j = 0, \ldots, r$ to be chosen. Note that we always have $c_0 = 0$. By studying the local truncation error, constraints can be placed on these coefficients to achieve a specific p-th order accurate scheme [33].

For example, if $r = 1$, $\mathbf{a} = [a_0, a_1] = [1, -1]$, $\mathbf{b} = [\frac{1}{2}, \frac{1}{2}]$, $\mathbf{c} = [0, 1]$, we apply second order Crank-Nicolson to the implicit term, and first order AB1 to the explicit term. This is a first order one-step scheme called CNAB1 [6].

The SBDF2 scheme is a second order two-step scheme [6] with coefficients $\mathbf{a} = [\frac{3}{2}, -2, \frac{1}{2}]$, $\mathbf{b} = [1, 0, 0]$ and $\mathbf{c} = [0, 2, -1]$. This is second order BDF2 (2-step backwards differentiation [19]) implicitly and second order extrapolation explicitly [33]. The SBDF3 scheme is similar, except is is instead third order accurate [6].

The CNLF2 scheme is a second order scheme that uses Crank-Nicolson implicitly and leapfrog explicitly. It has coefficients $\mathbf{a} = [\frac{1}{2}, 0, -\frac{1}{2}]$, $\mathbf{b} = [\frac{1}{2}, 0, \frac{1}{2}]$ and $\mathbf{c} = [0, 1, 0]$ [6].

These schemes can be modified to accept a varying timestep Δt_n by making the coefficients a_j, b_j, c_j a function of the timestep ratio $\omega_n = \frac{\Delta t_n}{\Delta t_{n-1}}$, as implemented in the Dedalus documentation [6]. If the timestep is to change, it must be done so slowly to avoid numerical issues [5].

6 Solid body rotation in Dedalus

In this section, we demonstrate the implementation of IVPs in Dedalus by solving an advection equation with a solid body rotation solution. We will also compare the effects of a changing timestep on stability, and the effectiveness of different timesteppers. The `time_PDE` class defined in this section will be used to solve the main Stommel–Munk problem in Section 7.

Let the domain Ω be a disc of radius L_r. Two-dimensional advection can be described as the PDE

$$\partial_t \psi + \frac{1}{2}\nabla^{\perp} r^2 \cdot \nabla \psi = 0. \tag{21}$$

The second term can be rewritten in polar coordinates as

$$\frac{1}{2}\nabla^{\perp} r^2 \cdot \nabla \psi = \frac{1}{2}\begin{pmatrix} 0 \\ 2r \end{pmatrix} \cdot \begin{pmatrix} \partial_r \psi \\ \frac{1}{r}\partial_\varphi \psi \end{pmatrix} = \partial_\varphi \psi.$$

Thus the problem can be reduced to

$$\partial_t \psi = -\partial_\varphi \psi. \tag{22}$$

Angular velocity is defined as $\partial_t \varphi$ [8]. The chain rule gives $\partial_t \psi = \partial_\varphi \psi \cdot \partial_t \varphi$, and so the angular velocity of ψ in Equation 21 is

$$\partial_t \varphi = \frac{\partial_t \psi}{\partial_\varphi \psi} = -1.$$

Thus, we expect the body u to rotate anti-clockwise at a angular speed of 1 radian per time unit.

As in Section 3, selected Dedalus code will be presented alongside explanations. Note that as in Section 4, object orientated programming is used. The full code can be found in Appendix C.8 and class definitions in Appendices C.3, C.5 and C.7.

The subclass `time_PDE` of `DedalusSolver` is a solver for any time-dependent PDE and implements the method `solve_problem` which timesteps a solver. There are also methods for plotting and animating. As in Section 4.2.1, a disc basis is created with resolution `Nphi` and `Nr` within the method `make_space`. A subclass of `rotation_PDE` of `time_PDE` will implement problem specific details for solid body rotation. We will also use this class in Section 7 to implement our main problem.

6.1 Class initialisation

The `time_PDE` takes the following initialisation values.

```python
def __init__(self,
                Nphi,
                Nr,
                initial_func,
                *,
                Lr=1,
                dealias=2,
                timestepper=d3.SBDF2,
                stop_sim_time=np.pi/2,
                timestep=0.1,
                local=True,
                save_every=1,
                scales=1,
                save_name=None,
                import_previous=False,
                variable_name=None,
                **kwargs):
```

- `Nphi` and `Nr` are the number of azimuthal and radial basis vectors.
- `initial_func` is a function describing the initial condition $\psi(t = 0)$.
- `Lr` is the radius of the circular domain Ω.
- `dealias` is the dealising factor.
- `timestepper` is the timestepper for the simulation.
- `stop_sim_time` is the end time for the simulation.
- `timestep` is the timestep.
- `local` determines whether the output is stored in RAM, or to the drive.
- `save_every` takes a snapshot every specified number of timesteps.
- `save_name` file name to save as if `local=False`. If set to `None`, it defaults to a timestamp.
- `scales` is the snapshot mesh scale. This does not affect the numerics. A higher number gives more resolution in post-processing [5].
- If `import_previous` is true, then instead of running a simulation, it imports a previously run simulation from `save_name`.
- `variable_name` is the name of the field of interest.
- `kwargs` is a dictionary of parameters that will all be saved to the class (use with caution to avoid overwriting).

6.2 Creating the problem

We begin by creating a field for ψ and time t.

```
psi = dist.Field(name='psi', bases=disk)
t = dist.Field()
```

Next, we create a vector field for $\partial_\varphi \psi$.

```
phi, r = dist.local_grids(disk)
u = dist.VectorField(coords, bases=disk)
u['g'][0] = r
u['g'][1] = 0
```

Alternatively, we could get Dedalus to work with $\nabla^\perp$ directly via `skew(grad)`, but in this case it is faster (both computationally, and code writing) to work directly with coordinates. The problem can now be simply implemented as Equation 22.

```
problem = d3.IVP([psi], time=t, namespace=locals())
problem.add_equation("dt(psi) = - u @ grad(psi)")
```

No boundary conditions are used, and thus no tau factors are needed.

The initial conditions are set by defining `psi['g']` using the initial function from initialisation.

```
q['g'] = self.initial_func(phi, r)
```

6.3 Solving the problem in time

The following instructs Dedalus to build a solver.

```
solver = problem.build_solver(self.timestepper)
solver.stop_sim_time = stop_sim_time
```

We then create a loop with step size `timestep` for Dedalus to run over. Here, we log a progress report every 100 timesteps and save a snapshot every `save_every` timesteps.

```python
sim_dt = save_every * timestep
q.change_scales(scales=self.scales)
q_list = [np.copy(q['g'])]
t_list = [solver.sim_time]

while solver.proceed:
    solver.step(timestep)

    if solver.iteration % 100 == 0:
        logger.info('Iteration=%i, Time=%e, dt=%e'
                       % (solver.iteration, solver.sim_time,
                          timestep))

    if solver.iteration % iteration == 0:
        q.change_scales(scales=self.scales)
        q_list.append(np.copy(q['g']))
        t_list.append(solver.sim_time)
```

The line `solver.step(timestep)` tells Dedalus to take one timestep [6]. This single timestep is solved as a time-dependent problem. Snapshots are saved to the lists `q_list` and snapshot times are saved to `t_list`. The resolution of a snapshot can be increased with larger `self.scales` (the default is 1). This scaling works by increasing the number of collocation points Dedalus uses in post-processing. Since spectral methods do not depend on a grid in space and only relies on a set of basis functions, we are free to evaluate the resulting liner combination of the basis functions over any reasonable grid.

6.4 Solving the advection equation

The problem is solved by running the following code after defining the relevant classes. We use the degree 3 Bessel function as the initial condition on the unit circle from Section 4.3

```python
Lr = 1


def initial_func(phi,r):
    return bessel_q(phi, r, n=3, Lr=Lr)
```

We use 2^7 basis functions along both axis (chosen for good resolution and reasonable computational time). We also double the collocation points along both axis for more resolution.

```python
Nphi, Nr = 2**7, 2**7
scales = 2
```

Since we expect ψ to rotate about the origin every 2π time units, we choose a run time of 30π and snapshot every $\frac{\pi}{2}$ time units. For this problem, a timestep of $\frac{\pi}{400}$ is sufficient as shall be seen in Section 6.5. We wish to keep all timesteps rational multiples of π for easier error analysis later.

```python
time_step = np.pi/400
stop_sim_time = 30*np.pi + time_step
save_every = 200
```

For the timestepper, we use the third order timestepper SBDF3. Other timesteppers are investigated in Section 6.5.2.

```python
timestepper = d3.SBDF3
```

With the above parameters, we call the `rotation_PDE` class and save the snapshots to the drive (see Appendix C.7.1 for more details).

```python
rotation_bessel = rotation_PDE(Nphi, Nr,
                               initial_func,
                               Lr=Lr,
```

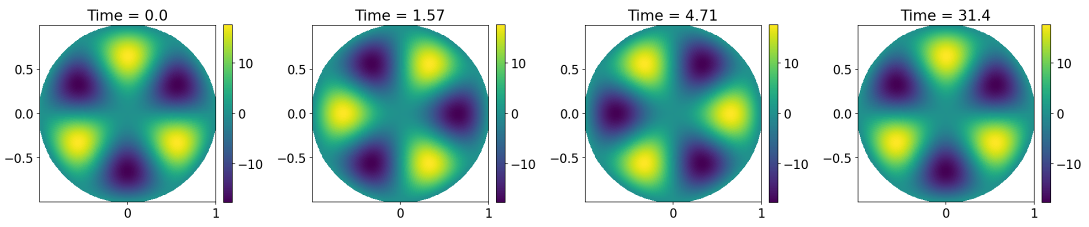

Figure 14: Snapshots of the solution to solid body rotation with initial condition given by the third order Bessel function. The timestepper used is SBDF3 with a timestep of $\frac{\pi}{400}$.

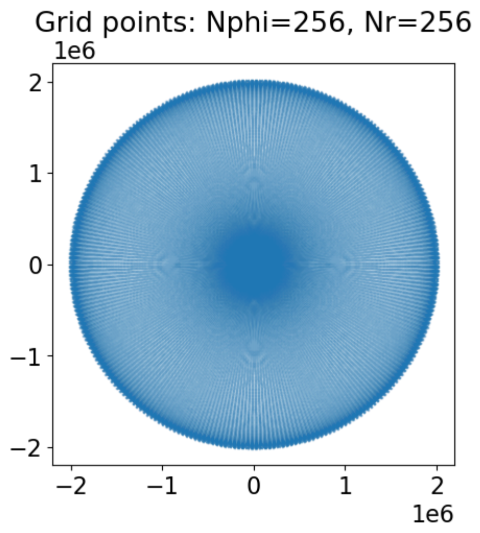

Figure 15: Grid points of the disc basis with 256 radial and 256 azimuthal basis functions. We see more grid points along the boundary and in the centre.

```
                                                stop_sim_time=stop_sim_time,
                                                timestep=time_step,
                                                timestepper=timestepper,
                                                local=False,
                                                save_every=save_every,
                                                scales=scales)
```

6.5 Results

Using the `animate` method, we can create an animation of ψ's evolution over time. This method uses `matplotlib.animation.FuncAnimation` as in Appendix C.7. Instead of embedding an animation into this dissertation, we may use the `time_plot` method which plots static snapshots at specific indices normalised to the interval $[0, 1]$.

```
rotation_bessel.time_plot([0, 0.07, 0.17, 1])
```

These plots are shown in Figure 14. We can see clear evidence of rotation in the first 2π seconds. When viewing the animation, it rotates a total of 15 times in the interval $[0, 30\pi]$, as expected.

To explicitly evaluate the accuracy of the simulation, we can compare multiples of $t = 2\pi$ to the initial condition. Note that there will be some rounding errors arising from the fact that π is irrational. Dedalus can return field field evaluations at collocation points using the `'g'` selector, or the coefficients of the spectral basis using `'c'`. We choose to directly store the grid points for ease of visualisation.

For simplicity, instead of evaluating the integrals in Section 2.4 to compute the errors, we estimate the integral over the grid of collocation points following a similar method to the weighted two-norm in LeVeque [19]. Here, our grid has size 256×256. Although the grid spacing is not

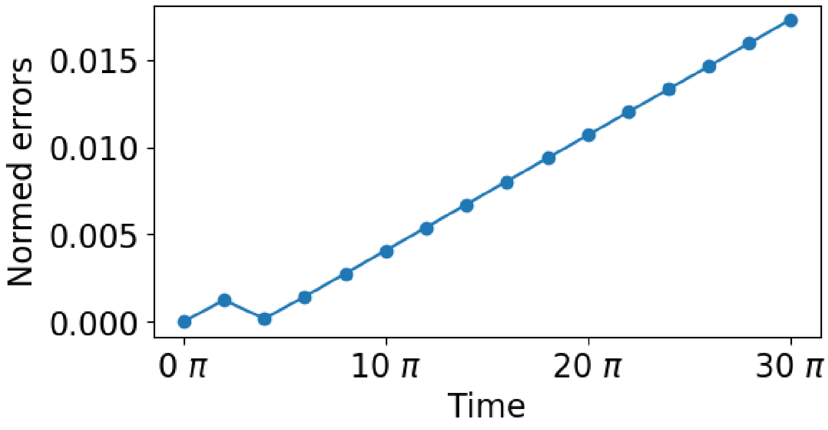

Figure 16: Norm of errors of the solution to solid body rotation at multiples of $t = 2\pi$ for SBDF3 and a timestep of $\frac{\pi}{400}$.

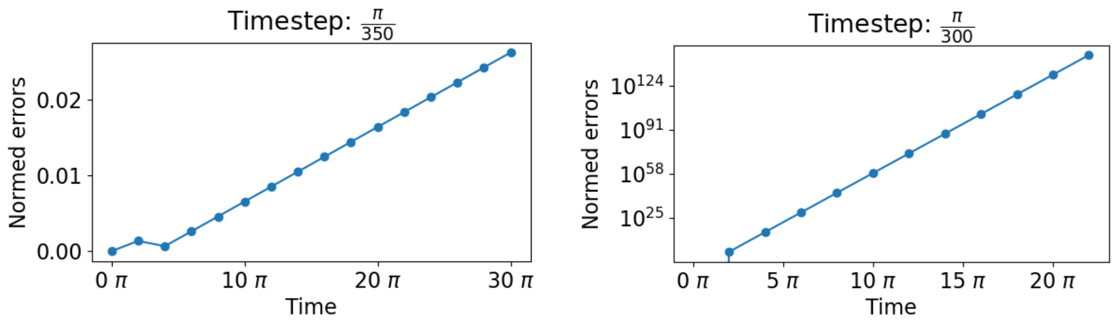

Figure 17: Norm of errors of the solution to solid body rotation at multiples of $t = 2\pi$ for SBDF3 and a timestep of $\frac{\pi}{350}$ and $\frac{\pi}{300}$.

uniform as in Figure 15, for simplicity we perform a uniform weighting over the grid points. If we wanted to compare $\psi(t = 0)$ and $\psi(t = 2\pi)$, we evaluate

$$\|\psi(t=0) - \psi(t=2\pi)\|_2 = \left(\frac{1}{256^2} \sum_{r,\varphi} |\psi(t=0,r,\varphi) - \psi(t=2\pi,r,\varphi)|^2 \right)^{\frac{1}{2}},$$

where the sum is over all collocation points (r, φ). This can be viewed as an unweighted average of the squared difference of the grid points.

The code in Appendix C.8.1 implements this, and returns the plot in Figure 16. The errors are reasonably small, but increase slightly over time. This may partially be due to Python's handling of floats.

6.5.1 Comparison of different timesteps

If we increase the slightly timestep, we would expect the errors to increase slightly. But if the timestep increases too much, we would get exponential growth due to numerical instability, since we have a nonzero forcing term on the right side of the implementation of Equation 22. This means Dedalus uses an explicit method which would put us outside the region of absolute stability.

In Figure 17, we see that the normed errors linearly increase hen the timestep is $\frac{\pi}{350}$ with a slightly greater slope than when the timestep was $\frac{\pi}{400}$. We see exponential growth when the timestep is $\frac{\pi}{300}$.

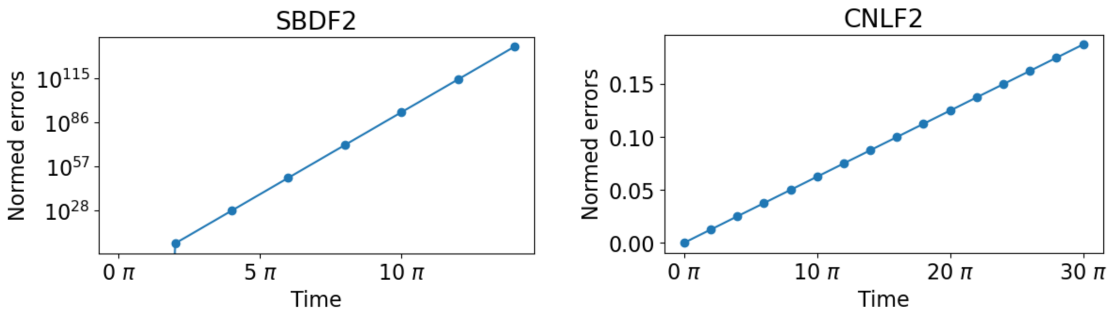

Figure 18: Norm of errors of the solution to solid body rotation at multiples of $t = 2\pi$ with a timestep of $\frac{\pi}{400}$ and timesteppers SBDF2 and CNLF2.

6.5.2 Experimental comparison of timesteppers

In Figure 18, we use a timestep of $\frac{\pi}{400}$ for various timesteppers. We see exponential growth for SBDF2, which is expected since it is only an order 2 method (lower than the order of SBDF3). The second order scheme CNLF2 is stable for the timestep $\frac{\pi}{400}$, but the slope of the error line is an order of magnitude larger than SBDF3.

Thus, we see that SBDF3 with a timestep of $\frac{\pi}{400}$ is a reasonably good choice of timestepper for the solid body rotation problem. We will use SBDF3 to solve the Stommel–Munk problem in the next section.

7 The Stommel–Munk problem

In 1948, Stommel proposed an ocean model describing westward intensification of wind driven currents [30]. This behaviour causes more streamlines on the west side of a domain, compared to the east boundary. This occurs due to the Coriolis effect (Section 7.1). Westward intensification is evident in NASA's visualisation of the Gulf Stream in Figure 1. We describe Stommel's original model in Section 7.3.1, and discuss the effects of wind forcing.

In response to Stommel's model, Munk published a paper in 1950, changing Stommel's drag term to a harmonic viscosity [23], which better explains observed real-world data.

We first follow Vallis' [31] derivation of the Stommel–Munk model from Chapters 2, 4, 5 and 19. Then we use Dedalus to simulate this model using all the framework built in the previous sections.

7.1 Coriolis effect

The Coriolis force is a quasi-force on a moving object in a rotating inertial frame [32], for example, water moving on a rotating planet. Let $\mathbf{r}(\mathbf{x}, t)$ be the position of a moving object inside a rotating inertial frame. Then we can split $\Delta\mathbf{r}$ into its relative and rotational components respectively by following Vallis [31]

$$(\Delta\mathbf{r})_I = (\Delta\mathbf{r})_R + (\Delta\mathbf{r})_{\text{rot}},$$

where $(\Delta\mathbf{r})_R$ is the movement within the frame, and $(\Delta\mathbf{r})_{\text{rot}}$ is due to the rotation of the inertial frame. Using the fact that $(\Delta\mathbf{r})_{\text{rot}} = \boldsymbol{\omega} \times \mathbf{r}\delta t$ [31], where $\boldsymbol{\omega}$ is the constant angular velocity, we get

$$(\partial_t \mathbf{r})_I = (\partial_t \mathbf{r})_R + \boldsymbol{\omega} \times \mathbf{r}. \tag{23}$$

Let $\mathbf{v}_I := (\partial_t \mathbf{r})_I$ be the inertial velocity and $\mathbf{v}_R := (\partial_t \mathbf{r})_R$ be the relative velocity. Then Equation 23 becomes $\mathbf{v}_I = \mathbf{v}_R + \boldsymbol{\omega} \times \mathbf{r}$. Substituting $\mathbf{v}_R$ for $\mathbf{r}$ yields

$$(\partial_t \mathbf{v}_R)_I = (\partial_t \mathbf{v}_R)_R + \boldsymbol{\omega} \times \mathbf{v}_R,$$

thus

$$(\partial_t(\mathbf{v}_I - \boldsymbol{\omega} \times \mathbf{r}))_I = (\partial_t \mathbf{v}_R)_R + \boldsymbol{\omega} \times \mathbf{v}_R.$$

It follows that [31]

$$(\partial_t \mathbf{v}_R)_R = (\partial_t \mathbf{v}_I)_I + \underbrace{(-2\boldsymbol{\omega} \times \mathbf{v}_R)}_{\text{Coriolis force}} + \underbrace{(-\boldsymbol{\omega} \times (\boldsymbol{\omega} \times \mathbf{r}))}_{\text{centrifugal force}}.$$

The Coriolis effect acts on moving objects perpendicularly to their direction of travel [31]. Let $f := 2\omega \sin y$ be the Coriolis parameter (proportional to the norm of the Coriolis force), where $\omega = |\boldsymbol{\omega}|$. Since this magnitude varies with latitude, we can approximate the Coriolis parameter by performing a Taylor expansion

$$f = f_0 + \beta y,$$

where $f_0 := 2\omega \sin y_0$ (at the expansion point) and $\beta := \partial_y f = \frac{2\omega}{L_r} \cos y$ is the Rossby parameter (L_r is the radius of domain). This is called the β-plane approximation [31].

7.2 Vorticity equation

From the general two-dimensional vorticity equation in a rotating frame ([31] Equation 4.66), if we assume that the flow is inviscid and incompressible, we have

$$D_t(\zeta + f) = 0,$$

where ζ is the vorticity, $D_t = \partial_t + \nabla^\perp \psi \cdot \nabla$ is the material derivative (ψ is the stream function) and f is the Coriolis parameter. By assuming that f is constant in time and using the β-plane approximation, we get

$$\partial_t\zeta + \nabla^\perp\psi \cdot \nabla(\zeta + \beta y) = 0, \tag{24}$$

as $\nabla f = (0, \beta)^T$. This is known as the two-dimensional β-plane vorticity equation [31].

7.3 The model

Consider a large circular body of water such that its radius L_r is much larger than its depth. We also assume the ocean has a flat bottom, so that the depth of water is a constant H, and that any waves on the surface do not significantly contribute to the depth. Then we can use a shallow water approximation, and treat this as a two dimensional problem [31]. So Section 7.2 can be applied.

7.3.1 Stommel model

By forcing the vorticity equation (Equation 24) with wind-stress curl and linear drag, we obtain the time-dependent Stommel problem [31]

$$\partial_t\zeta + \nabla^\perp\psi \cdot \nabla(\zeta + \beta y) = \underbrace{Q(y)}_{\text{wind forcing}} + \underbrace{(-r_0\nabla^2\psi)}_{\text{drag}}, \tag{25}$$

where $Q(y)$ is a latitude dependent wind forcing term, and r_0 is a damping coefficient due to frictional dissipation.

By ignoring the $D_t\zeta$ term in Equation 25, we obtain Stommel's original time-independent model [30]

$$r_0\nabla^2\psi + \nabla^\perp\psi \cdot \nabla(\beta y) = Q(y). \tag{26}$$

The term $\nabla^\perp\psi \cdot \nabla(\beta y) = \beta\partial_x\psi$ is known as planetary vorticity [23].

7.3.2 Munk Model

A modification to the Stommel problem is to replace the friction term $-r_0\nabla^2\psi$ with a harmonic viscosity term $\nu\nabla^2\zeta$ [23],

$$\partial_t\zeta + \nabla^\perp\psi \cdot \nabla(\zeta + \beta y) = \underbrace{Q(y)}_{\text{wind forcing}} + \underbrace{\nu\nabla^2\zeta}_{\text{viscoscity}}. \tag{27}$$

This change takes into account the fact that currents vanish at greater depths [23].

7.3.3 Stommel–Munk model

The Stommel–Munk model uses both the Stommel friction and the Munk viscosity for damping. Combining Equations 25 and 27, we get

$$\partial_t\zeta + \nabla^\perp\psi \cdot \nabla(\zeta + \beta y) = \underbrace{Q(y)}_{\text{wind forcing}} + \underbrace{(-r_0\nabla^2\psi)}_{\text{drag}} + \underbrace{\nu\nabla^2\zeta}_{\text{viscoscity}}$$

We need two boundary conditions on $\partial\Omega$ to solve this problem uniquely [31]. The first will be $\psi = 0$, which ensures no normal flow at the boundary. The second is zero-vorticity $\zeta = 0$, called free slip [31]. This helps to keep momentum inside the domain, and so the viscosity ν can create eddy currents [31].

While Stommel and Munk's original formulations consider a rectangular domain, we use a circular domain for our Dedalus implementation (see Section 4), i.e. Ω is a circle of radius L_r.

Including the boundary conditions yields the full Stommel–Munk problem.

$$\begin{cases} \partial_t \zeta + r_0 \nabla^2 \psi - \nu \nabla^2 \zeta = -\nabla^\perp \psi \cdot \nabla(\zeta + \beta y) + Q(y) & \text{on } \Omega, \\ \nabla^2 \psi - \zeta = 0 & \text{on } \Omega, \\ \psi = 0 & \text{on } \partial\Omega, \\ \zeta = 0 & \text{on } \partial\Omega, \end{cases} \tag{28}$$

where $y = r \sin \varphi$. We wish to solve for the stream function ψ and vorticity ζ over time. The parameters are wind forcing $Q(y)$, Rossby parameter β, viscosity ν and drag r_0. These are given explicit values in Section 7.3.5.

7.3.4 Initial conditions

For the initial conditions of the Stommel–Munk problem, we need to set a small smooth non-zero function for both ψ and ζ, such that they satisfy Equation 28. To do this, we choose a function ζ that satisfies $\zeta(r = L_r) = 0$. Then we solve for ψ in we

$$\begin{cases} \nabla^2 \psi = \zeta & \text{in } \Omega, \\ \psi = 0 & \text{on } \partial\Omega. \end{cases}$$

Following the method of solving a Poisson equation over a circular domain in Section 4, we can use Dedalus to solve this PDE using one tau factor. The following function is implemented as a method of the subclass `stommel_PDE` used to solve the Stommel–Munk problem (Section 7.5).

```python
def initial_condition(self, psi, zeta):
    # Tau method
    tau = self.dist.Field(name='tau_zeta', bases=self.edge)

    def lift(A):
        lift_basis = self.disk.derivative_basis()
        return d3.Lift(A, lift_basis, -1)

    # problem
    ic_problem = d3.LBVP([psi, tau], namespace=locals())
    ic_problem.add_equation("lap(psi) + lift(tau) = zeta")
    ic_problem.add_equation("psi(r=self.Lr) = 0")

    # Solver
    solver = ic_problem.build_solver()
    solver.solve()
    return psi
```

From Section 4.3, we already know that the smooth function

$$\zeta(r, \varphi) = \sin(2\varphi) J_2 \left(\frac{\alpha}{L_r} r \right),$$

where α is a root of the second Bessel function J_2, satisfies the boundary condition $\zeta(r = L_r)$ (by construction). Then the above Dedalus code finds ψ from this choice of ζ at $t = 0$.

7.3.5 Parameter choices

Our parameters use standard SI convention for units [3], as opposed to CGS convention in [30] and [23]. These parameters are summarised in Appendix B. Ocean depth H, wind forcing strength F, wind forcing shift Q_{shift} and viscosity ν are the parameters we tune in Section 7.6.

Domain size For the radius of the circular domain, we use $L_r = 2 \times 10^6$ m. This is of the same order of magnitude as the rectangular domain from [30]. We use a constant ocean depth

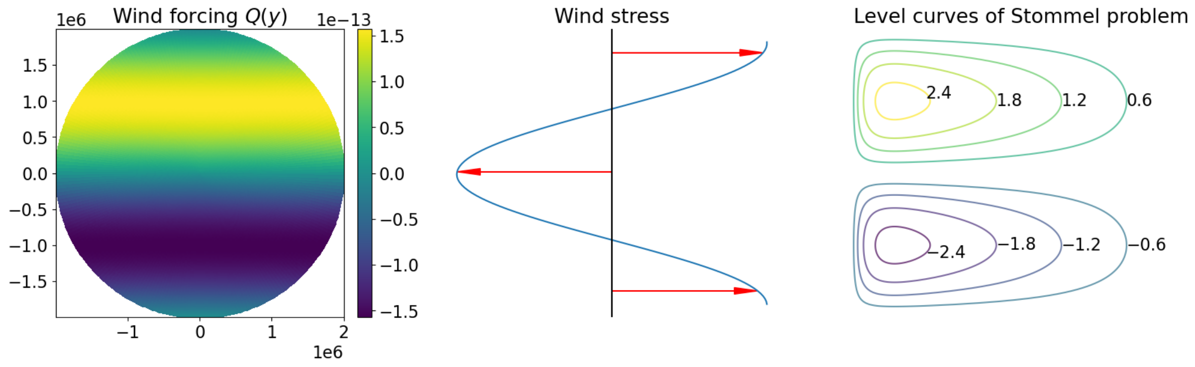

Figure 19: The first plot shows the wind forcing $Q(y)$ (Equation 30) with parameters $L_r = 2 \times 10^6\,\mathrm{m}$, $Q_{\text{shift}} = 1\%$, $F = 0.1\,\mathrm{N\,m^{-2}}$ and $H = 500\,\mathrm{m}$ and $\rho_0 = 10^3\,\mathrm{kg\,m^{-3}}$. The second plot shows the wind stress (Equation 29). The final plot shows analytic solutions to the time-independent Stommel problem on a rectangular domain (Equation 31).

of $H = 500\,\mathrm{m}$. The Gulf Stream within the Atlantic Ocean has similar dimensions [35].

Wind forcing The oceans experience wind forcing based on latitude [30]. For simplicity, we do not use true observed data of the wind stress (which can be complicated as in Figure 2 of [23]). Instead, we will use a sinusoidal stress [30]

$$-\frac{F}{\rho_0 L_r H} \cos\left(\frac{\pi y}{L_r}\right),$$

where F is a wind strength parameter. The density of water is around $\rho_0 = 10^3\,\mathrm{kg\,m^{-3}}$. To break symmetry (see Section 7.6.2), we shift the wind forcing northwards by a small amount $Q_{\text{shift}} L_r$ to get

$$-\frac{F}{\rho_0 L_r H} \cos\left(\frac{\pi(y + Q_{\text{shift}} L_r)}{L_r}\right). \tag{29}$$

The derivative of the wind stress is

$$Q(y) = \frac{F\pi}{\rho_0 L_r H} \sin\left(\frac{\pi(y + Q_{\text{shift}} L_r)}{L_r}\right), \tag{30}$$

We use a wind strength $F = 0.1\,\mathrm{N\,m^{-2}}$ and $Q_{\text{shift}} = 1\%$. Both H and F are slightly different to Stommel's parameter choices [30] to weaken the effects of wind forcing, because spectral methods can handle higher resolutions with lower damping. These parameters are tweaked so that the jets appearing in Section 7.6 travel approximately half way across the domain. If $\max |Q| = \frac{F\pi}{\rho_0 L_r H}$ is too big, then the wind forces the system excessively, causing the jets to crash into the eastern boundary and causing unrealistic chaotic behaviour as discussed in Section 7.6.2.

The wind forcing is plotted in Figure 19 with concrete values for the parameters.

Coriolis parameters The Rossby parameter is $\beta = \frac{2\omega}{L_r} \cos y$, where $\omega = 7.25 \times 10^{-5}\,\mathrm{rad\,s^{-1}}$ is the angular speed of Earth's rotation [1, Earth Rotation. Dickey]. So, we use $\beta = 2 \times 10^{-11}\,\mathrm{s^{-1}\,m^{-1}}$.

Damping We use $r_0 = 2 \times 10^{-7}\,\mathrm{N}$ for drag, which is the same as Stommel's value [30] (who uses dynes instead of Newtons). For viscosity, Munk [23] uses $5 \times 10^3\,\mathrm{m^2\,s^{-1}}$. The smaller this viscosity is, the more eddy currents appear. Thus, we use $\nu = 80\,\mathrm{m^2\,s^{-1}}$, which spectral methods can resolve with a reasonable amount of basis functions.

Simulation time We use a total simulation time T of 1 year with snapshots every week and a timestep Δt of 6 minutes. Increasing the timestep may cause numerical problems (see Section 5.5 and Section 6.5.1). This simulation time encompasses the behaviour in Section 7.4 and takes a total of around 2.5 hours to run with 256 basis functions along both axis as in Section 7.6.1.

7.4 Expected behaviour

Given a small initial condition as in Section 7.3.4, all terms in Equation 28 are small, except Q. So, when t is small, Q dominates. Thus $\zeta \sim Qt$, i.e. vorticity grows linearly. As $\zeta = \nabla^2\psi$ grows, the terms in Equation 28 continue to grow, but $\nabla^2\zeta$ will grow the slowest. Thus, at the beginning of the simulation, we may ignore $-\nu\nabla^2\zeta$, and be left with the Stommel problem (Equation 25).

The Stommel problem here has a solution similar to the result in the lower pane of [31, Fig. 19.6] – two vortices (called gyres) form in the north and south of the domain. These gyres move to the western boundary, causing westward intensification. This can be analytically observed by splitting ψ into two parts – an interior term, and a boundary correction term [31]. Inside the interior, the drag term $-r_0\nabla^2\psi$ is negligible, so we can consider the effects of wind forcing only. This yields a dimensionless solution of the time-independent Stommel problem in a rectangular domain of the form [31, Section 19.1.2]

$$\psi(x, y) = \underbrace{(1-x)\pi\sin\pi y}_{\text{interior}} - \underbrace{e^{-\frac{x}{\varepsilon}}\pi\sin\pi y}_{\text{boundary}}, \tag{31}$$

where ε is a constant depending on parameters. Increasing magnitudes of the level curves of this shows ψ's evolution over time as in the third plot of Figure 19, where westward intensification of the level sets can be observed.

After the initial westward intensification, $-\nu\nabla^2\zeta$ becomes non-negligible, and we must consider the whole Stommel–Munk problem. We wish for eventual formation of eddy currents from the Rossby waves to create a simulation that resembles the Earth's oceans in Figure 1.

7.5 Dedalus implementation

As in Section 6, we will use the `time_PDE` class from Section 6.1. We will need to create a subclass `stommel_PDE` specific to the Stommel–Monk problem, implementing the PDE in Equation 28. Then, this will be solved with the method outlined in Section 6.3.

7.5.1 Creating the problem

We first create fields for two variables to solve for on the disc, the stream function ψ and vorticity ζ.

```python
psi = dist.Field(name='psi', bases=disk)
zeta = dist.Field(name='zeta', bases=disk)
```

We then create a field for $y = r\sin\varphi$.

```python
phi, r = dist.local_grids(disk)
y = dist.Field(name='y', bases=disk)
y['g'] = r * np.sin(phi)
```

We also define the wind forcing Q from Equation 30.

```python
Q = dist.Field(name='Q', bases=disk)
Q['g'] = (F * np.pi / (rho0 * Lr * H)) * np.sin(np.pi * r * np.sin(phi
    ) /Lr)
```

Since there are two boundary conditions, we need two tau factors.

```python
tau_zeta = dist.Field(name='tau_zeta', bases=edge)
tau_psi = dist.Field(name='tau_psi', bases=edge)

def lift(A, i=-1):
     lift_basis = disk.derivative_basis(2)
     return d3.Lift(A, lift_basis, i)
```

We can now implement the Stommel–Munk problem (Equation 28) in Dedalus.

```python
problem = d3.IVP([zeta, psi, tau_zeta, tau_psi], time=t, namespace=
    locals())
problem.add_equation("dt(zeta)  + r0 * lap(psi) - nu * lap(zeta) +
    lift(tau_zeta, -2) \
                     = -skew(grad(psi)) @ grad(zeta + beta * y)  + Q")
problem.add_equation("lap(psi) - zeta + lift(tau_psi, -1) = 0")
problem.add_equation("psi(r=Lr) = 0")
problem.add_equation("zeta(r=Lr) = 0")
```

We then set the ζ initial condition using the Bessel function defined in Section 4.3.1. The ψ initial condition is set by solving a PDE as in Section 7.3.4.

```python
def zeta_init(phi, r):
    phi_mesh, r_mesh = np.meshgrid(phi, r)
    return 1e-16 * partial(bessel, n=n, Lr=Lr)(phi_mesh, r_mesh).T


zeta['g'] = zeta_init(phi, r)
psi = self.initial_condition(psi, zeta)
```

Then we set the parameters based on Section 7.3.5.

```python
Lr = 2e6
constants = {
    'F' : 0.1,
    'H' : 500,
    'r0' : 2e-7,
    'beta' : 2e-11,
    'nu' : 80,
    'rho0' : 1000,
    'Q_shift' : 0.01}
```

For the basis functions, we use 256 along both the azimuthal and radial components of the disc basis.

```python
Nphi, Nr = 256, 256
```

Since SBDF3 was a reasonable choice for the solid body rotation problem in Section 6, we use it for the Stommel–Monk problem, along with a dealias scale of 2 (i.e. double the number of basis functions before truncating).

```python
timestepper = d3.SBDF3
dealias = 2
```

We choose a timestep of 6 minutes for a period of 1 year. Snapshots will happen every week with collocation resolution increased threefold.

```python
time_step = 6 * 60
stop_sim_time = 60 * 60 * 24 * 365

save_every = (60 * 60 * 24 * 7) // time_step
scale = 3
```

We can then run the solver.

```python
stommel_bessel = stommel_PDE(Nphi, Nr,
                             initial_func,
```

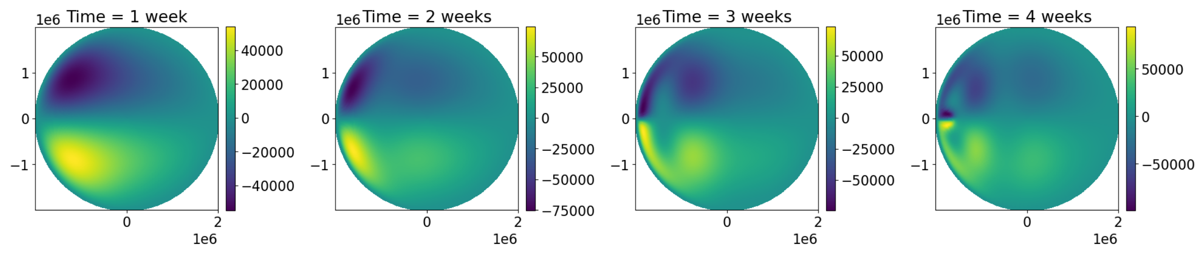

Figure 20: Early behaviour of the stream function ψ of the Stommel–Munk problem with parameters $Q_{\text{shift}} = 1\%$, $\nu = 80\,\text{m}^2\,\text{s}$, $F = 0.1\,\text{N}\,\text{m}^{-2}$ and $H = 500\,\text{m}$ as given in Appendix B.3 and explained in Section 7.3.5.

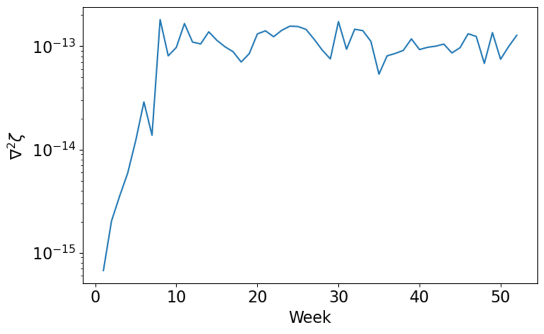

Figure 21: Plot of the unweighted average magnitude of the Munk viscosity term $\nabla^2\zeta$ for the Stommel–Munk problem with parameters $Q_{\text{shift}} = 1\%$, $\nu = 80\,\text{m}^2\,\text{s}$, $F = 0.1\,\text{N}\,\text{m}^{-2}$ and $H = 500\,\text{m}$.

```
                                        dealias=dealias,
                                        stop_sim_time=stop_sim_time,
                                        timestep=time_step,
                                        timestepper=timestepper,
                                        Lr=Lr,
                                        scales=scale,
                                        local=False,
                                        save_every=save_every,
                                        **constants)
```

7.6 Results

As in Section 6.5, we can view snapshots using the `time_plot` method and animations using the `animate` method. In Figure 29, we see the evolution of the stream function ψ in Equation 28 over the course of a year. The vorticity ζ can be seen in Figure 30. More frames for various parameters can be viewed in Section A. Note that the plot of vorticity ζ conveys more visual information that the stream function ψ, as variations of ψ are small.

In the first two weeks in Figure 20, we see the Stommel term dominating, causing westward intensification of the north and south gyres as discussed in Section 7.4. This is similar to the expected behaviour in the third plot if Figure 19.

After the initial westward intensification, waves form as in weeks 2-4 of Figure 20. We see that the Munk viscosity term $\nabla^2\zeta$ increases log-linearly until around week 10 as in Figure 21. This confirms that for small time, we may ignore the Munk term.

Two vortices also form at week 4, and travel directly east along the centre of the domain. Since the wind forcing is shifted slightly north, the vortices whip northwards as seen in week 8 of Figure 22. At around week 14, eddy currents start peeling off from the main vortices.

Figure 23 shows a high resolution snapshot at week 36. The main jet reaches around a quarter of the domain, and many eddies have been formed. There are small waves in the east, but most

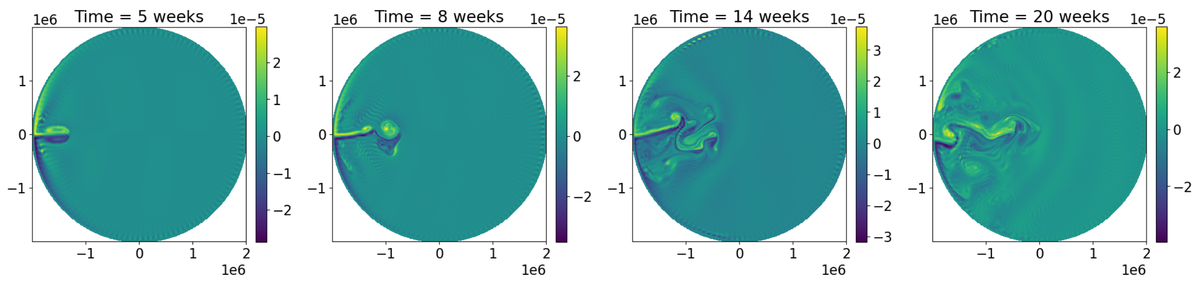

Figure 22: Plots of vorticity ζ of the Stommel–Munk problem showing formation of eddy currents. With parameters $Q_{\text{shift}} = 1\%$, $\nu = 80\,\mathrm{m}^2\,\mathrm{s}$, $F = 0.1\,\mathrm{N\,m}^{-2}$ and $H = 500\,\mathrm{m}$.

activity is still in the west. This behaviour is characteristic of the Gulf Stream as in Figure 1.

In years 2 and 3, similar behaviour is observed as in Figure 31. Over time, vortices are formed and move around near the western border. The problem remains stable

The pixellation of the vorticity ζ in Figure 23 along the boundary $\partial\Omega$ may be a sign of numerical instabilities caused by having a small ν and free slip boundary conditions. This does not seem to have a large effect on the overall stability of the simulation with the current timestep and simulation length. Increasing the number of basis functions should decrease this pixellation.

7.6.1 Performance

To run the Stommel–Munk problem for 1 year with a time step of 6 minutes, it takes Dedalus approximately 2.5 hours of computation time on an Intel i5 processor and 8GB RAM with no threading (as suggested in documentation [6]) and without utilising a GPU. Saving both the stream function and vorticity every week (for a total of 52 snapshots) at a resolution of 768×768 takes approximately 500 MB of disc space. The scaling for both computation time and storage space is approximately linear – a three year simulation takes 8 hours and 1.5GB of storage.

Possible speedups are discussed in Section 8.1.1.

7.6.2 Varying wind forcing

When $Q_{\text{shift}} = 0$, the wind forcing is symmetric across the horizontal axis. In this case, we expect ψ and ζ to be symmetric, as in Figure 32. The formed eddies are also symmetric as in Figure 33. This is because all terms in Equation 28 have symmetry.

The value $\max|Q| = \frac{F\pi}{\rho_0 L_r H}$ determines the strength of wind forcing in the Stommel–Munk problem (Equation 28). The parameters for the depth H and wind strength F can be freely changed to alter $\max|Q|$. Figure 24 shows the created jets at 19 weeks for different values of $\max|Q|$ in the symmetric case ($Q_{\text{shift}} = 0$). We see that halving $\max|Q|$ causes the jets to travel less in those 19 weeks. This distance travelled approximately halves, thus their speed is halved when $\max|Q|$ is halved.

In Figure 25 when wind forcing is large, we see the jets crashing into the eastern border. This causes the formation of two vortices which destroy the jets and cause chaotic behaviour as in Figure 34. This is not the behaviour of the Gulf Stream we want to model (Figure 1), but may be characteristic of fluids in a smaller body of fluid or higher wind strengths.

7.6.3 Varying viscosity

Varying viscosity ν does not have much effect at the beginning since the viscosity forcing term is negligible. We would expect that a higher viscosity makes it harder for eddies to form. In Figure 26, we see more detailed eddies for $\nu = 40$ compared with $\nu = 80$. The lengths of the two main vortex jets are approximately the same.

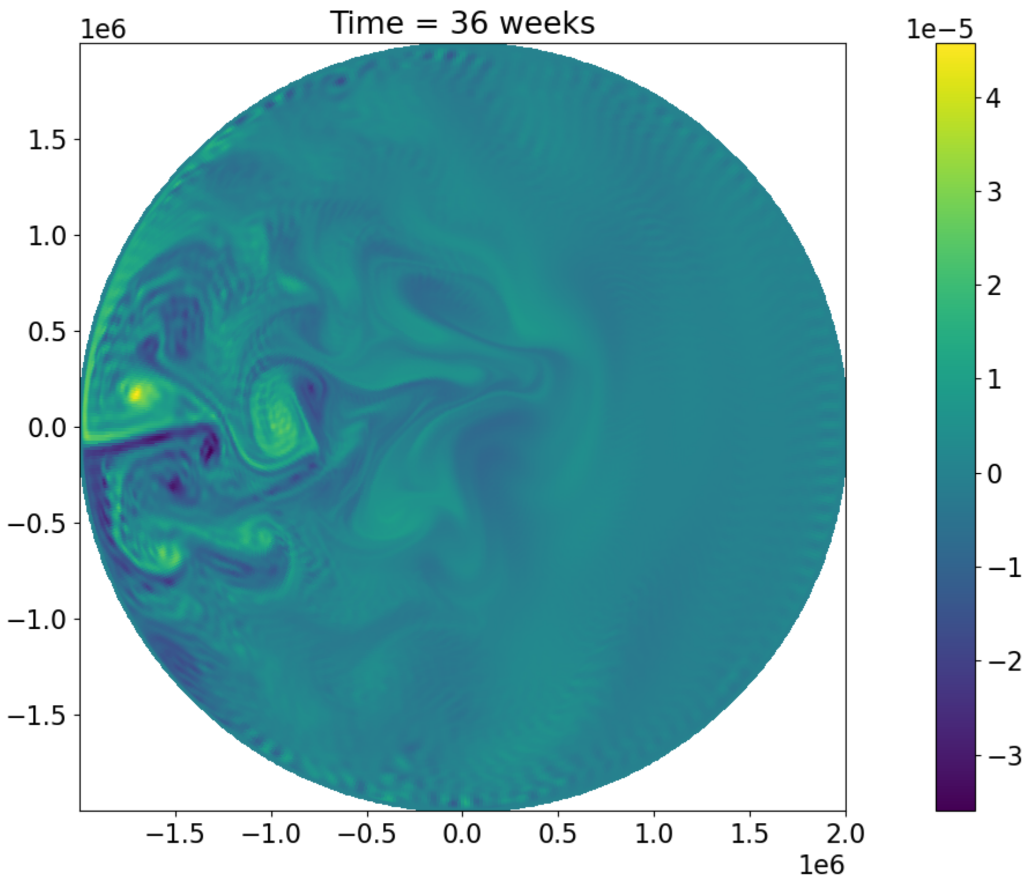

Figure 23: High resolution plot of the vorticity ζ of the Stommel–Munk problem, showing ocean-like behaviour with parameters $Q_{\text{shift}} = 1\%$, $\nu = 80\,\text{m}^2\,\text{s}$, $F = 0.1\,\text{N}\,\text{m}^{-2}$ and $H = 500\,\text{m}$.

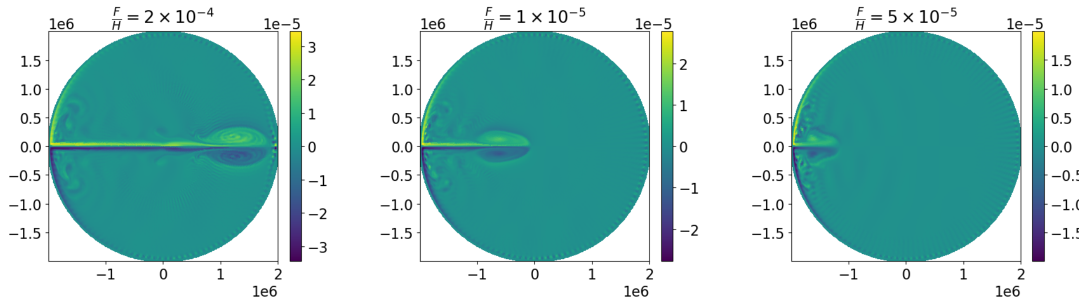

Figure 24: Plot of vorticity ζ at 19 weeks for different wind forcing strengths $\max |Q| \propto \frac{F}{H}$ with $Q_{\text{shift}} = 0\%$ and $\nu = 80$. The jets move further for larger strengths.

Smaller values of ν would require more resolution through more basis functions. In Figure 27, we see that we cannot resolve the eddies with $N_r = N_\varphi = 256$ number of basis functions. Increasing the number of basis functions requires more computational time.

If ν is too large, Dedalus experiences numerically difficulties and the Munk viscosity term explodes. A further line of work would be to tune the parameters and step size to investigate a large viscosity, and to compare these results to standard literature, where we expect the system to settle in a steady state [31].

The finite element method can simulate the Stommel–Munk model with a large viscosity $\nu = 2000\,\text{m}^2\,\text{s}^{-1}$ [13]. The FEM simulation results in results similar to ours at small timesteps, including westward intensification. The benefit of FEM is that the domain can be very complex as a spacial mesh can cover any shape, for example the Mediterranean Sea in [13].

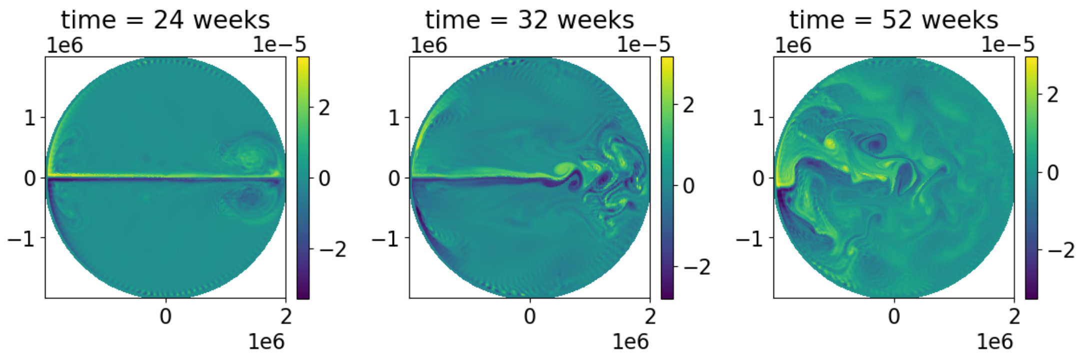

Figure 25: Plots of vorticity ζ for $\frac{F}{H} = 2 \times 10^{-4}$ at 24, 32 and 52 weeks with $Q_{\text{shift}} = 0$ and $\nu = 80$. The jets crash into the eastern boundary and cause chaotic behaviour.

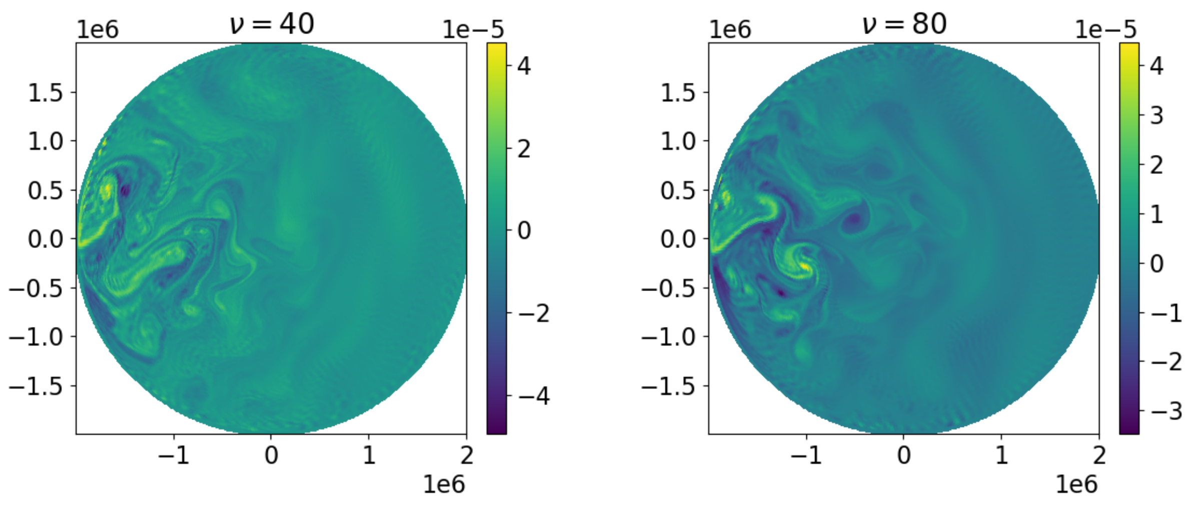

Figure 26: Plots of vorticity ζ for different values of viscosity ν with parameters $Q_{\text{shift}} = 1\%$, $F = 0.1\,\mathrm{N\,m^{-2}}$ and $H = 500\,\mathrm{m}$. There are more eddies for the lower viscosity.

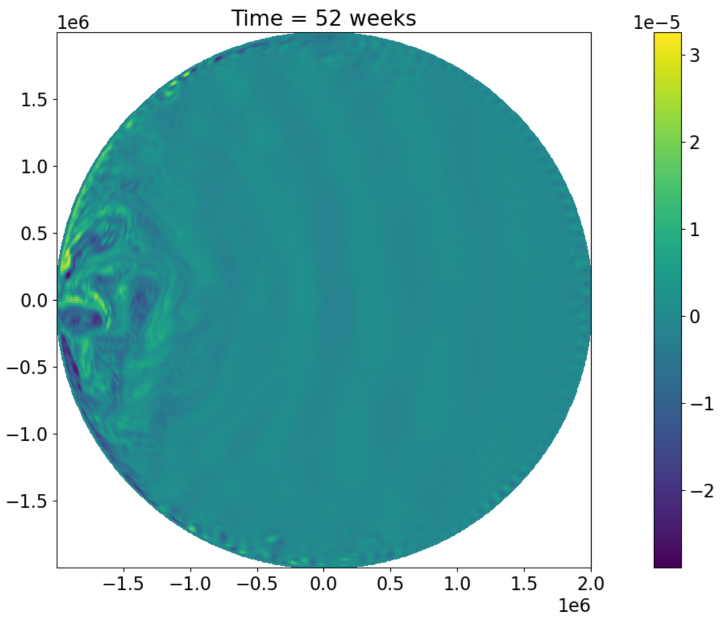

Figure 27: Plot of vorticity ζ for $\nu = 20\,\mathrm{m^2\,s}$ showing unresolved details with parameters $Q_{\text{shift}} = 1\%$, $F = 0.05\,\mathrm{N\,m^{-2}}$ and $H = 500\,\mathrm{m}$.

8 Conclusion

8.1 Future work

Obvious future work would include investigating the effects of a wider range of damping coefficients on the Stommel–Munk problem, i.e. making viscosity ν small and increasing the number of basis functions; making ν large and decreasing timestep; and varying the drag r_0. A longer simulation time of 50 to 100 years would also be beneficial to examine long term behaviour. To achieve the resolution and to replicate the behaviour of NASA's Gulf Stream in Figure 1, we need to modify our main parameters in Section 7.3.5 to a lower viscosity, higher resolution and accuracy, and greater wind forcing; and increase the number of basis functions and decreasing the timestep. Besides these tweaks, there are a few other extensions that could be implemented.

8.1.1 GPU

One of the biggest issues of simulating the Stommel–Munk model was computation time. Aside from using better hardware or accessing a supercomputer, we could utilise parallel computing and GPUs. This is not currently supported by Dedalus, but there have been plans to implement such speedups[2].

Another helpful feature would be to be able to fully save a state, instead of just snapshots. This would help to stop and restart a simulation, as well as experimenting with a changing timestep mid-simulation.

8.1.2 Conformal mappings

So far, all simulations have used a circular domain due to limitations of setting boundary conditions in Dedalus. To reflect real-world domains, it may be more appropriate to model rectangular patches of the ocean. Thus, we wish to map between a circle and a rectangular domain. We wish for this map to be bijective, smooth, conformal (angle preserving) and boundary preserving. Although a smooth map is sufficient, conformal mappings can lead to nicer mathematics. Without loss of generality, we can consider circle to square to circle maps, as a suitable map from a square to a rectangle is stretching.

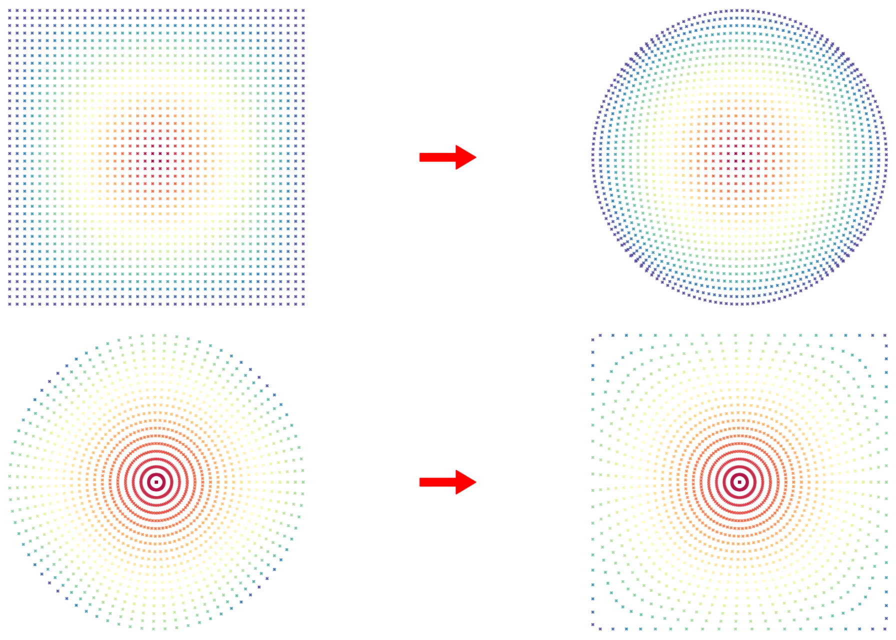

Figure 28: The elliptical grid mapping as in [12].

[2]See the discussion in `https://groups.google.com/g/dedalus-dev/c/rj2gK1VMjx8`.

The review paper by Fong [12] presents the Schwarz-Chrisoffel conformal mapping. They also suggest the elliptical grid mapping (which utilises trigonometric relations to convert between Cartesian and polar) which is not conformal, but is an easy to implement map as in Figure 28. The elliptical grid map keeps the radial component fairly even, but distorts angles and stretches the boundaries causing a fish-eye effect [12].

Given a PDE on a square, we apply the change of coordinates arising from our chosen map. To resolve issues with the collocation grid, we may need to evaluate fields at the preimage of Dedalus's circular grid points. This results in a PDE on a circular domain, and we may apply boundary conditions on the whole boundary $\partial\Omega$ as in Section 4. Dedalus can then work with this PDE and the result can be mapped back to the square.

8.1.3 Multilayer models

In our formulation of the Stommel–Munk model (Section 7), we assumed a think layer of ocean of constant depth H. This is called a rigid lid approximation. But, the ocean is not two-dimensional. Although we assume a shallow water system, in reality, there is mixing between different water depths, and the height of water may change. If we two shallow water models together, we can model more complex behaviour [31].

To correctly implement the boundary of the layers, we introduce the deformation radius $L_D = \frac{\sqrt{gH}}{f}$, which depends on the layer depth H, Coriolis parameter f and perceived gravitational acceleration g (called reduced gravity) [31]. Features with a length greater than the deformation radius experience rotational effects. In our formulation of the Stommel–Munk problem, we took $L_D = \infty$, which is called the short wave limit, where dispersion of water happens as in a non-rotating frame [31]. A finite deformation radius is added to the vorticity equation (Equation 7.2) by adding $-\frac{1}{L_D^2}\psi$ to get the shallow water quasi-geostrophic potential vorticity [31]

$$D_t\left(\nabla^2\psi + f - \frac{1}{L_D^2}\psi\right) = 0. \tag{32}$$

For a two layer model with depths H_1 and H_2, we have two stream functions ψ_1 and ψ_2 satisfying a modified Equation 32 depending on different deformation radii. The modified vorticity equations involve interaction terms between ψ_1 and ψ_2 via the deformation radius term as in [31, Equation 5.85]. Since the interface between the two layers experience both gravity from above, and buoyancy from below, we must account for reduced gravity in L_D.

If we continue stacking layers, in the limit, each layer will have an infinitesimal thickness. This represents continuous stratification [31].

8.2 Final remarks

In this dissertation, we have defined spectral methods for solving a PDE and have demonstrated its core ideas by solving a Poisson equation on a box in Section 3 using basis recombination and the Galerkin method. Dedalus is the computational Python package used for spectral methods, and we have demonstrated Dedalus' use on the same Poisson equation example using the tau method, which is a modification of the Galerkin method that allows for boundary conditions. We found that spectral methods solve this toy example well and that only around 20 basis functions are needed to achieve machine precision. This results in a much smaller matrix than finite difference, which would require a grid spacing of $\Delta x = \frac{1}{4\times10^5}$ to achieve a similar precision. This experiment suggests that spectral methods could be better than finite difference in terms of accuracy as long as we have a sufficiently regular domain (namely, a circle or rectangle).

In order to impose boundary conditions over the whole boundary, we moved to a circular domain in Section 4. We implemented a class that can be modified to solve different PDEs. The precision was found to be similar to Section 3 for the Poisson equation with a Bessel solution. We found that $N_\varphi = 8$ and $N_r = 14$ were sufficient for machine precision. The Bessel function would deem itself useful later as we needed a smooth nonzero function to initialise the main Stommel–Munk problem.

After exploring the mathematical theory and Dedalus implementation of time-dependent PDEs in Section 5, we solved an advection equation with a solid body rotation solution in Section 6. A Python class was created to implement this problem in Dedalus so that general problems can be easily created. Comparing different timesteppers, we found that the third order method SBDF3 had the lowest errors. This was the timestepper used for our main problem.

Using all the theory and examples on solving PDEs spectrally, we derived our main model – the Stommel–Munk problem in Section 7. This is a simplified model of the ocean, taking into account the Coriolis effect, wind forcing, linear drag and viscosity. Using free slip boundary conditions, we implemented this model in Dedalus with concrete parameter choices.

At the beginning of the simulation, the Stommel–Munk problem can be reduced to the Stommel problem, and westward intensification of a north gyre and a south gyre is observed. As the Munk term grows, waves start forming to the east of these shrinking gyres. These gyres then formed two vortices that travelled along the horizontal axis and emitted eddy currents as in Figure 23.

Our results were similar to that of literature, and we observed ocean-like behaviour given a low viscosity. We also found that a non-symmetric wind forcing was essential to achieve non-symmetric results. The best results used the following parameters: a small northwards shift $Q_{\text{shift}} = 1\%$ to break symmetry; a small viscosity $\nu = 80\,\text{m}^2\,\text{s}$ for the formation of eddy currents; and wind forcing strength $F = 0.1\,\text{N}\,\text{m}^{-2}$ and water depth $H = 500\,\text{m}$ to give a reasonable wind effect.

Overall, the spectral method is a good alternative to the traditional finite difference and finite element methods, as it offers better accuracies, yields smooth results, and is not reliant on a grid in the spacial dimensions. When implemented in Dedalus, spectral methods can reliably model complicated behaviour, such as the Stommel–Munk model of the ocean resembling the Gulf Stream.

References

[1] T. J. Ahrens, editor. **Global earth physics: a handbook of physical constants**. Number 1 in AGU reference shelf. American Geophysical Union, Washington, D.C, 1995.

[2] N. H. Asmar. **Partial differential equations with Fourier series and boundary value problems**. Pearson Prentice Hall, Upper Saddle River, N. j, 2nd ed edition, 2005.

[3] BIPM. **Le Système international d'unités / The International System of Units ('The SI Brochure')**. Bureau international des poids et mesures, ninth edition, 2019.

[4] J. P. Boyd. **Chebyshev and Fourier spectral methods**. Courier Corporation, 2001.

[5] K. J. Burns, G. M. Vasil, J. S. Oishi, D. Lecoanet, and B. P. Brown. Dedalus: A Flexible Framework for Numerical Simulations with Spectral Methods. **Physical Review Research**, 2(2):023068, Apr. 2020.

[6] K. J. Burns, G. M. Vasil, J. S. Oishi, D. Lecoanet, and B. P. Brown. Dedalus project documentation, 2022. Accessed: 30 July 2023.

[7] I. P. O. C. Change. Climate change 2007: The physical science basis. **Agenda**, 6(07):333, 2007.

[8] B. Crowell. Conceptual Physics. **Fullerton College**, Aug. 2020.

[9] B. N. Datta. **Numerical linear algebra and applications**. Society for Industrial and Applied Mathematics, Philadelphia, 2nd ed edition, 2010.

[10] P. Ditlevsen and S. Ditlevsen. Warning of a forthcoming collapse of the Atlantic meridional overturning circulation. **Nature Communications**, 14(1):4254, July 2023.

[11] D. Erdenesanaa. June Was Earth's Hottest on Record. August May Bring More of the Same. **The New York times**, July 2023.

[12] C. Fong. Mappings for Squaring the Circular Disc. **Seoul ICM**, 2014.

[13] E. L. Foster, T. Iliescu, and Z. Wang. A Finite element discretization of the streamfunction formulation of the stationary quasi-geostrophic equations of the ocean. **Computer Methods in Applied Mechanics and Engineering**, 261-262:105–117, July 2013.

[14] T. Gowers, J. Barrow-Green, and I. Leader. **The Princeton companion to mathematics**. Princeton University Press, 2010.

[15] N. J. Higham and M. R. Dennis, editors. **The Princeton companion to applied mathematics**. Princeton University Press, Princeton, 2015.

[16] L. Kemp, C. Xu, J. Depledge, K. L. Ebi, G. Gibbins, T. A. Kohler, J. Rockström, M. Scheffer, H. J. Schellnhuber, W. Steffen, and T. M. Lenton. Climate Endgame: Exploring catastrophic climate change scenarios. **Proceedings of the National Academy of Sciences**, 119(34):e2108146119, Aug. 2022.

[17] S. W. Key and R. D. Krieg. Comparison of Finite-Element and Finite-Difference Methods. In **Numerical and Computer Methods in Structural Mechanics**, pages 337–352. Elsevier, 1973.

[18] N. Kukreja, M. Louboutin, F. Vieira, F. Luporini, M. Lange, and G. Gorman. Devito: Automated Fast Finite Difference Computation. In **2016 Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)**, pages 11–19, Salt Lake, UT, USA, Nov. 2016. IEEE.

[19] R. J. LeVeque. **Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems**. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2007.

[20] D. Li, editor. **Encyclopedia of Microfluidics and Nanofluidics**. Springer US, Boston, MA, 2008.

[21] P. Livermore. A compendium of Galerkin orthogonal polynomials. **Scripps Institution of Oceanography Technical Report**, 2009.

[22] D. L. Logan. **A first course in the finite element method**. Thomson, United States, 4th ed edition, 2007.

[23] W. H. Munk. On the wind-driven ocean circulation. **Journal of Atmospheric Sciences**, 7(2):80–93, 1950.

[24] J. Niesen. Stability region for BDF1.svg. **Wikimedia Commons, the free media repository**, 2020. Accessed: 02 Aug. 2023.

[25] E. L. Ortiz. The Tau Method. **SIAM Journal on Numerical Analysis**, 6(3):480–492, 1969.

[26] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. Mcrae, G.-T. Bercea, G. R. Markall, and P. H. J. Kelly. Firedrake: Automating the Finite Element Method by Composing Abstractions. **ACM Transactions on Mathematical Software**, 43(3):1–27, Sept. 2017.

[27] G. Rawitscher, V. Dos Santos Filho, and T. C. Peixoto. **An Introductory Guide to Computational Methods for the Solution of Physics Problems: With Emphasis on Spectral Methods**. Springer International Publishing, Cham, 2018.

[28] M. W. Scroggs, J. S. Dokken, C. N. Richardson, and G. N. Wells. Construction of Arbitrary Order Finite Element Degree-of-Freedom Maps on Polygonal and Polyhedral Cell Meshes. **ACM Transactions on Mathematical Software**, 48(2):1–23, June 2022.

[29] G. Shirah, H. Mitchell, V. Weeks, D. Menemenlis, and H. Zhang. Perpetual Ocean. **NASA/Goddard Space Flight Center Scientific Visualization Studio**, Aug. 2011.

[30] H. Stommel. The westward intensification of wind-driven ocean currents. **Transactions, American Geophysical Union**, 29(2):202, 1948.

[31] G. K. Vallis. **Atmospheric and Oceanic Fluid Dynamics: Fundamentals and Large-Scale Circulation**. Cambridge University Press, 2 edition, June 2017.

[32] G. M. Vasil, K. J. Burns, D. Lecoanet, S. Olver, B. P. Brown, and J. S. Oishi. Tensor calculus in polar coordinates using Jacobi polynomials. **Journal of Computational Physics**, 325:53–73, Nov. 2016.

[33] D. Wang and S. J. Ruuth. Variable step-size implicit-explicit linear multistep methods for time-dependent partial differential equations. **Journal of Computational Mathematics**, pages 838–855, 2008.

[34] R. Zhong. Where major wildfires have struck around the world so far in 2023. **The New York times**, Aug. 2023.

[35] I. S. Zonn, A. G. Kostianoy, and A. V. Semenov. Gulf Stream. In **The Western Arctic Seas Encyclopedia**. Springer International Publishing, Cham, 2017.

Appendices

A Extra figures

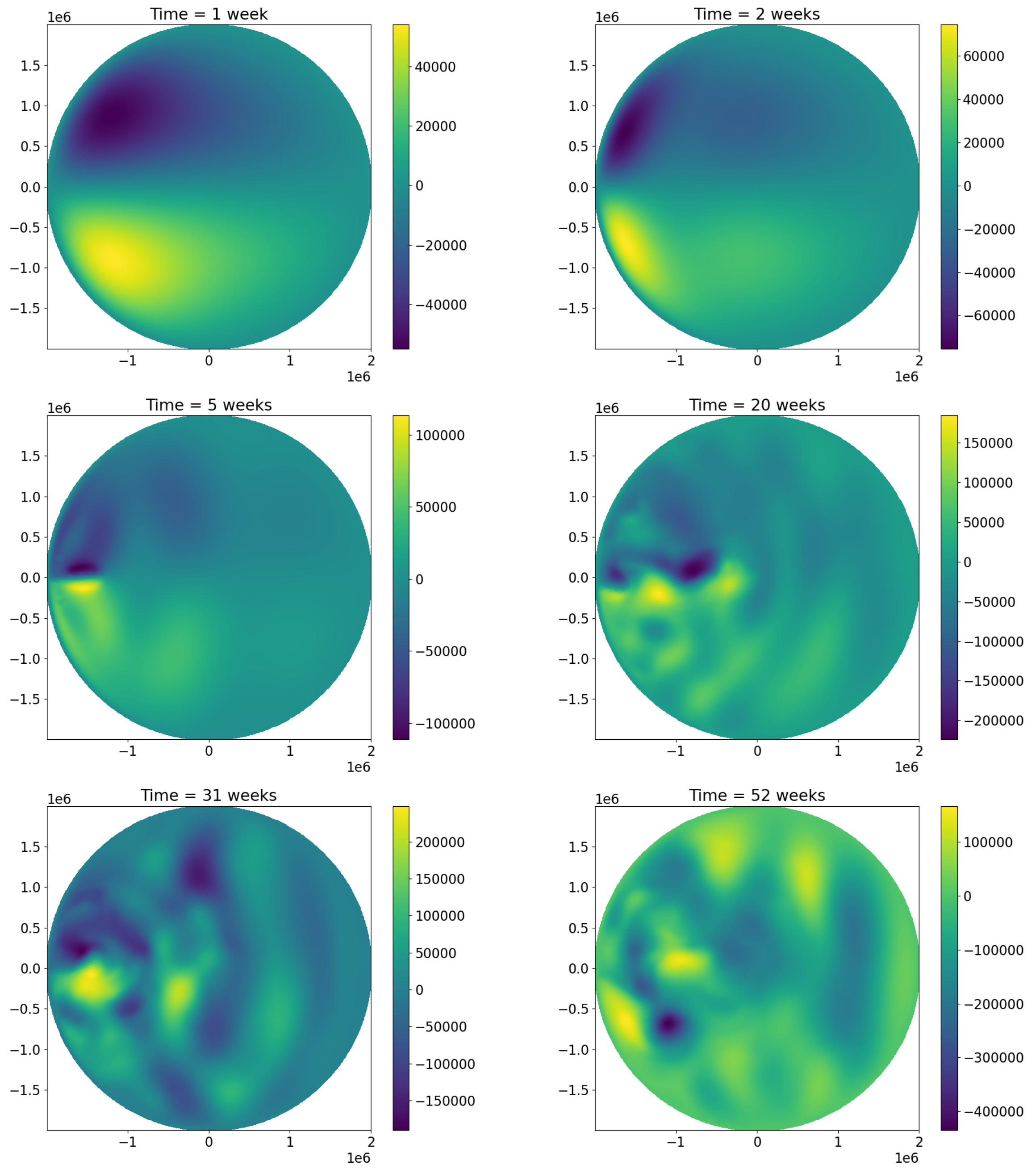

Figure 29: Plots of the Stommel–Munk model stream function at 1, 2, 5, 20, 31, 52 weeks with parameters $Q_{\text{shift}} = 1\%$, $\nu = 80\,\mathrm{m}^2\,\mathrm{s}$, $F = 0.1\,\mathrm{N\,m}^{-2}$ and $H = 500\,\mathrm{m}$ as explained in Section 7.3.5. Westward intensification of the north and south gyres is observed in the first two weeks. Then waves start appearing to the east of the gyres and vortices start to form and travel along the horizontal axis.

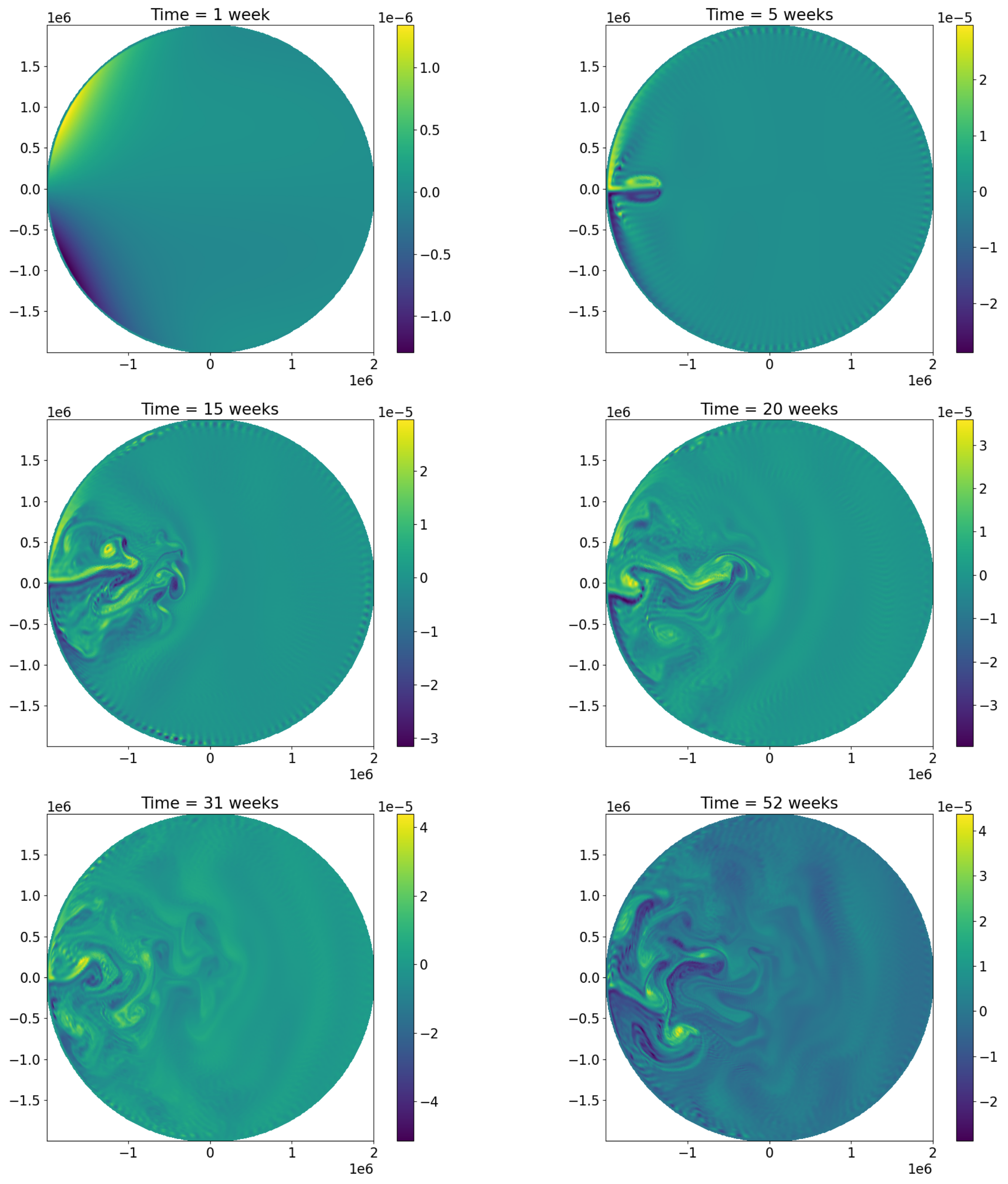

Figure 30: Plots of the Stommel–Munk model vorticity at 1, 5, 15, 20, 31, 52 weeks with parameters $Q_{\text{shift}} = 1\%$, $\nu = 80\,\text{m}^2\,\text{s}$, $F = 0.1\,\text{N}\,\text{m}^{-2}$ and $H = 500\,\text{m}$ as explained in Section 7.3.5. We see vortices travelling along the horizontal axis, and eddies being emitted. Most activity is concentrated in the west.

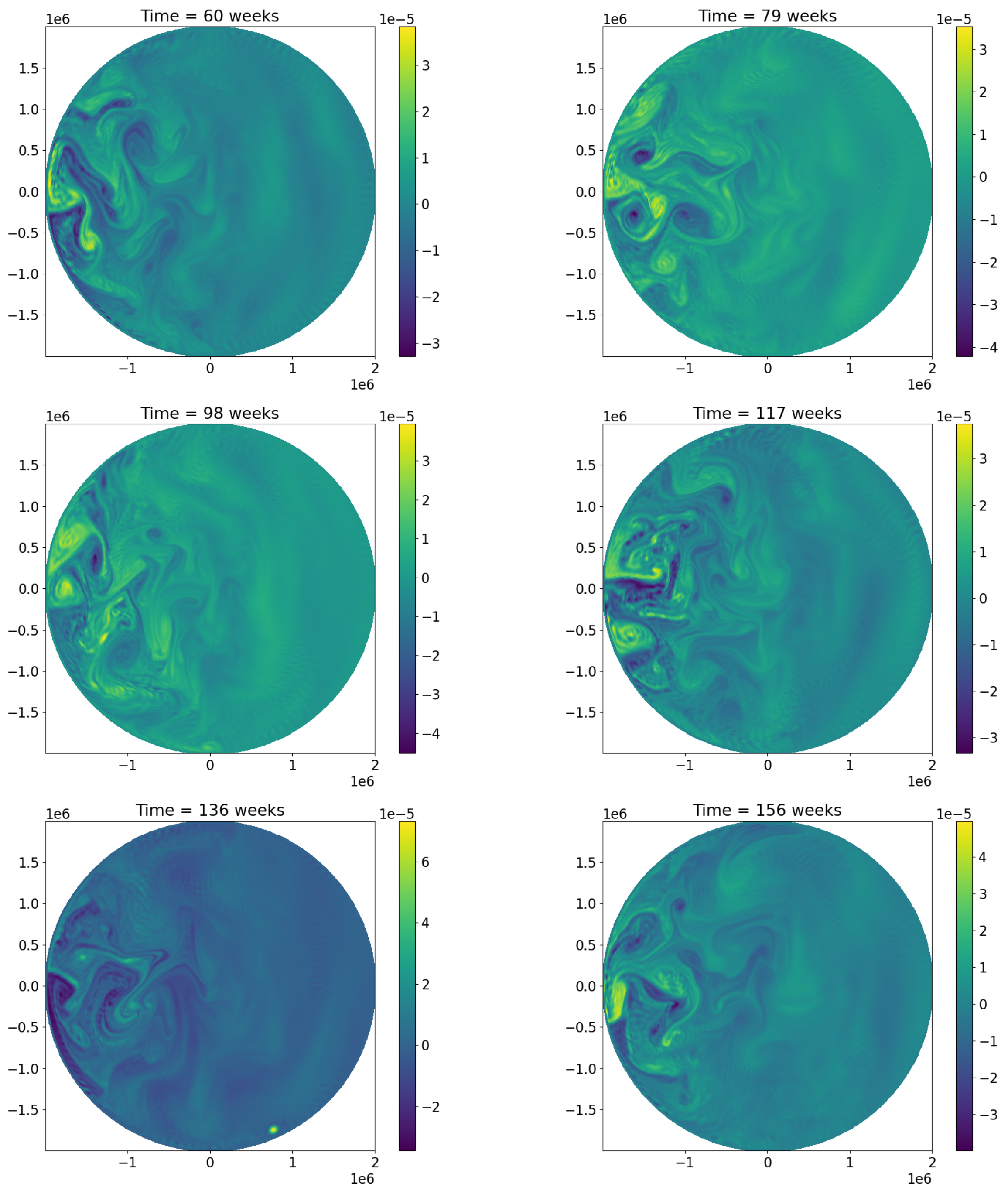

Figure 31: Plots of the Stommel–Munk model vorticity in the second and third years with parameters $Q_{\text{shift}} = 1\%$, $\nu = 80\,\mathrm{m}^2\,\mathrm{s}$, $F = 0.1\,\mathrm{N\,m}^{-2}$ and $H = 500\,\mathrm{m}$ as explained in Section 7.3.5. Over three years, eddies continue to form in the west. The problem remains stable.

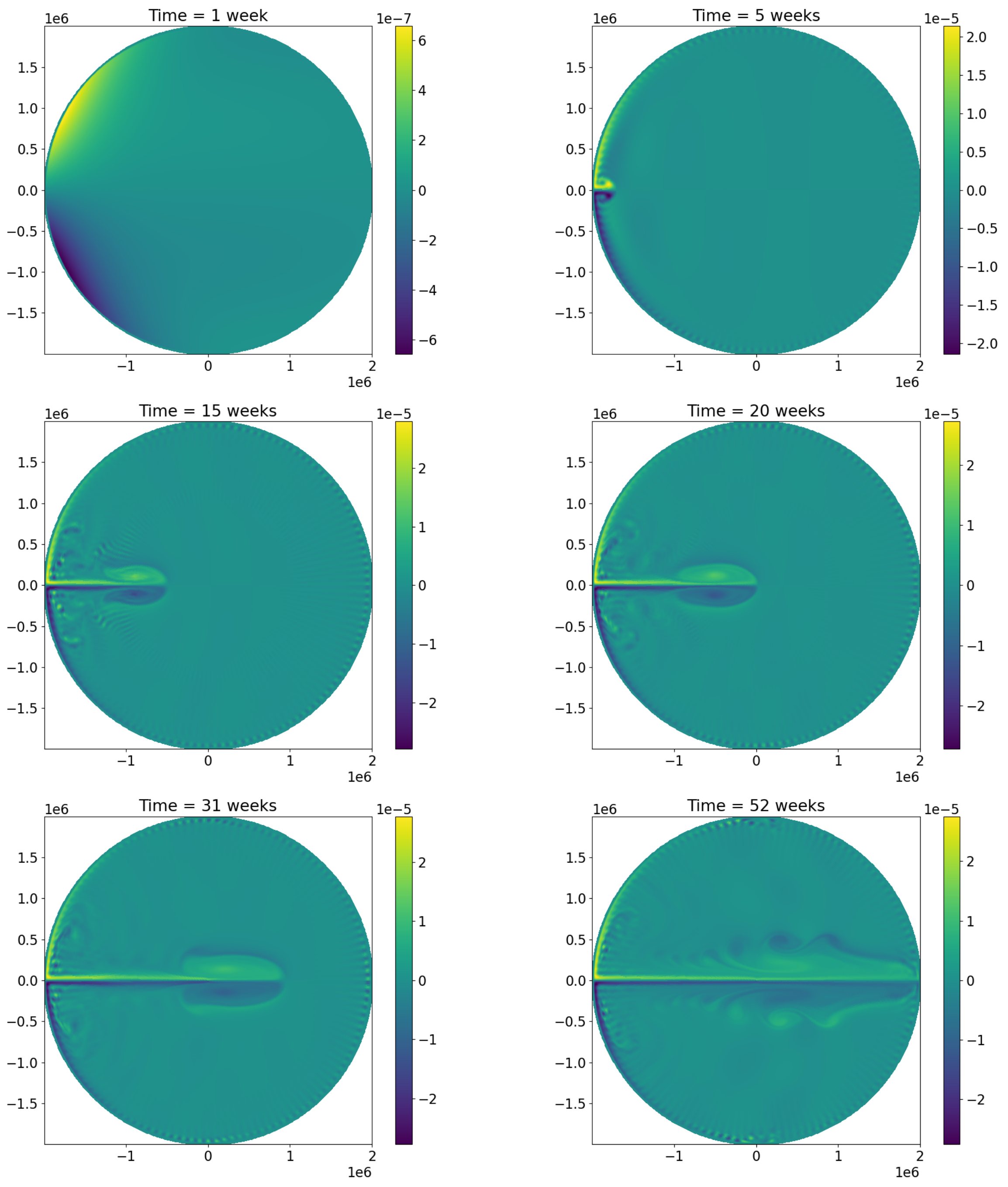

Figure 32: Plots of the symmetric Stommel–Munk model vorticity at 1, 5, 15, 20, 31, 52 weeks with parameters $Q_{\text{shift}} = 0\%$, $\nu = 80\,\text{m}^2\,\text{s}$, $F = 0.1\,\text{N}\,\text{m}^{-2}$ and $H = 10^3\,\text{m}$. The central jets remain true to the horizontal, and the formed eddies are symmetric.

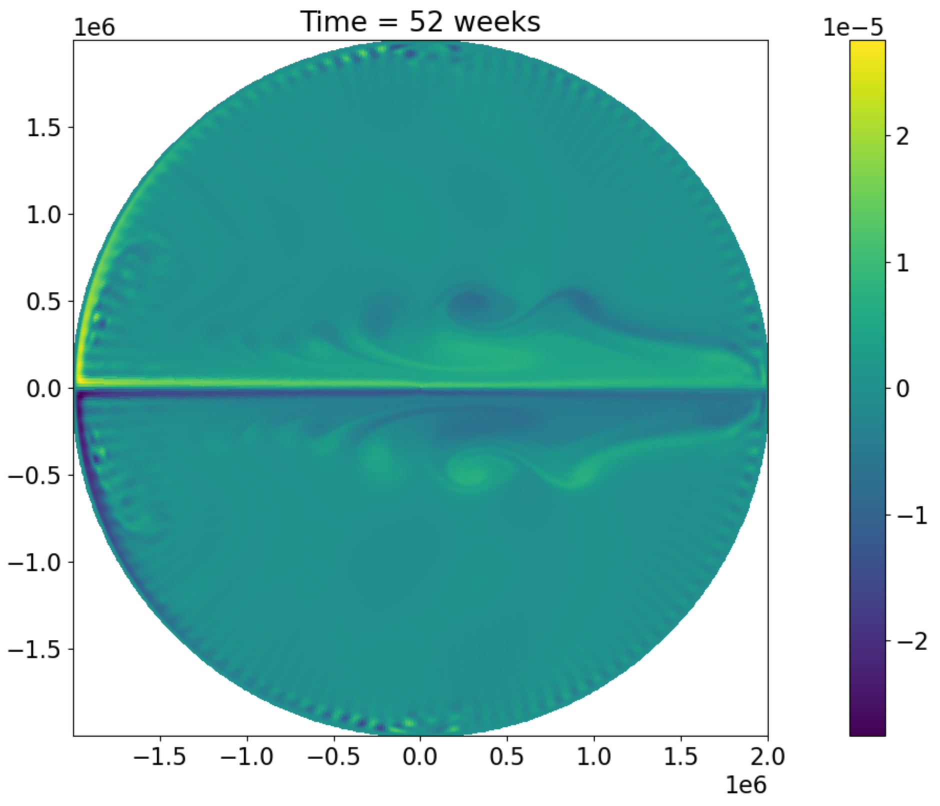

Figure 33: High resolution plot of eddies in the symmetric Stommel–Munk problem with parameters $Q_{\text{shift}} = 0\%$, $\nu = 80\,\text{m}^2\,\text{s}$, $F = 0.1\,\text{N}\,\text{m}^{-2}$ and $H = 10^3\,\text{m}$. The eddies are symmetric and peel off the main jet smoothly.

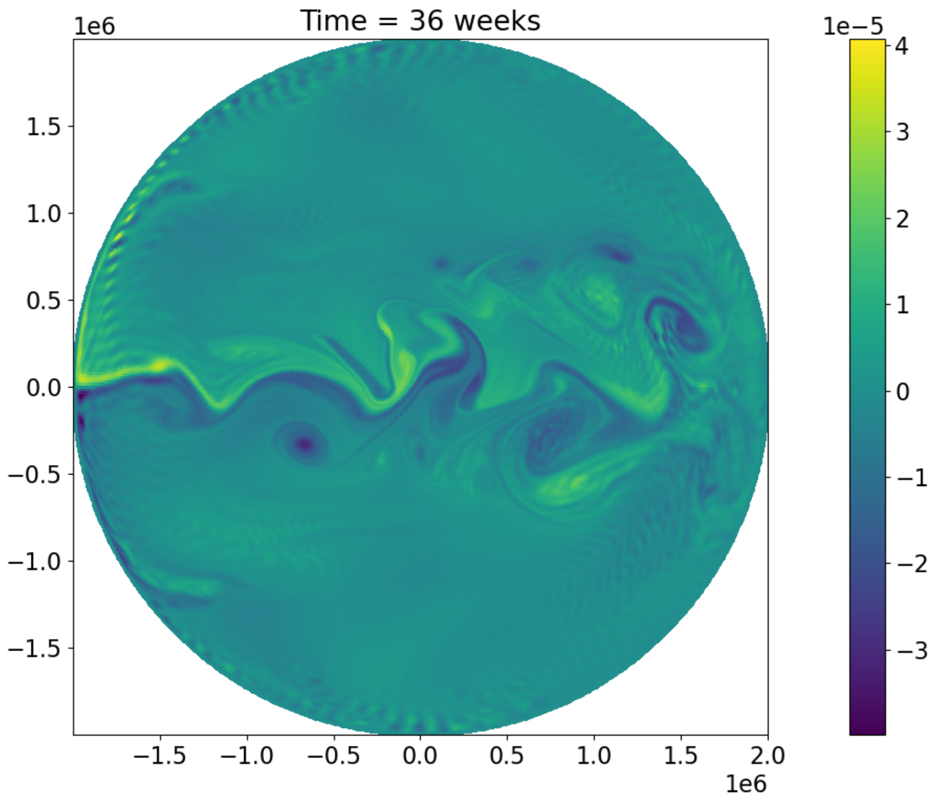

Figure 34: High resolution plot of vorticity showing the collapse of the main vortex jets a few weeks after they hit the eastern boundary in the Stommel–Munk problem with parameters $Q_{\text{shift}} = 0\%$, $\nu = 80\,\text{m}^2\,\text{s}$, $F = 0.1\,\text{N}\,\text{m}^{-2}$ and $H = 500\,\text{m}$.

B Nomenclature

B.1 Variable naming conventions

Symbol	Description
t	Time
r, φ	Radial and azimuthal components of polar coordinates
x, y	Components of Cartesian coordinates
Ω	Open, bounded and connected domain
$\partial\Omega$	Boundary of domain
Δt	Timestep
ψ	Stream function
ζ	Vorticity

B.2 Operators and functions

Symbol	Description	Definition
∂_x	Partial derivative with respect to x	
f'	Total derivative of f	
∇	Gradient	$\begin{pmatrix}\partial_x & \partial_y\end{pmatrix}^T$
$\nabla^\perp$	Orthogonal gradient	$\begin{pmatrix}-\partial_y & \partial_x\end{pmatrix}^T$
$\nabla\cdot$	Divergence	$\partial_x + \partial_y$
∇^2	Laplacian	$\partial_{xx} + \partial_{yy} = \partial_{rr} + \frac{1}{r}\partial_r + \frac{1}{r^2}\partial_{\varphi\varphi}$
D_t	Material derivative	$\partial_t + \nabla^\perp\psi \cdot \nabla$
$\delta(x)$	Kronecker delta	$\delta(x) = 0$ if $x \neq 0$ and $\delta(0) = 1$
$T_n(x)$	n-th Chebyshev polynomial	$\cos(n \arccos(x))$ for $n \in \mathbb{Z}_{\geq 0}$
$U_n(x)$	Chebyshev polynomial of the second kind	$U_{n-1}(x) = \frac{1}{n}T_n'(x)$
J_n	n-th Bessel function	Equation 15

B.3 Parameters

The parameter values are the ones used in the main Stommel–Munk simulation as discussed in Section 7.3.5.

B.3.1 Constant parameters

Symbol	Description	Value used in simulation
L_r	Radius of modelled ocean	$2 \times 10^6\,\mathrm{m}$
β	Rossby parameter	$2 \times 10^{-11}\,\mathrm{m}^{-1}\,\mathrm{s}^{-1}$
r_0	Drag damping	$2 \times 10^{-7}\,\mathrm{N}$
ρ_0	Density of water	$10^3\,\mathrm{kg}\,\mathrm{m}^{-3}$

B.3.2 Tweaked parameters

Symbol	Description	Value used in main simulation
ν	Viscosity damping	$80\,\mathrm{m}^2\,\mathrm{s}$
H	Depth of modelled ocean	$500\,\mathrm{m}$
F	Wind forcing strength	$0.1\,\mathrm{N}\,\mathrm{m}^{-2}$
Q_{shift}	Wind forcing vertical shift	1%

B.3.3 Simulation parameters

Symbol	Description	Value used in simulation
T	Simulation time	1 year
Δt	Simulation timestep	6 minutes
N_r, N_φ	Number of basis functions	256, 256
	Timestepper	SBDF3
	Snapshot period	Every week
s	Dealiasing scale	2
	Mesh scale	3

C Code

C.1 Using Dedalus on Windows

1. Install WSL `https://learn.microsoft.com/en-us/windows/wsl/install`.
2. Access Linux files in Windows path `\\wsl.localhost\`.
3. Open WSL using `wsl`.
4. Install Conda `https://github.com/conda-forge/miniforge`.
5. Install Dedalus v3 `https://dedalus-project.readthedocs.io/en/latest/pages/installation.html` [6].
6. Install other packages via Conda (e.g. `jupyter` and `matplotlib`).
7. Run `jupyter notebook` and open link in any browser (back in Windows).
8. Make sure you are running commands in the right Conda environment (check environment name via `conda info --envs` then activate using `conda activate dedalus3`).
9. Use `wsl --shutdown` to shutdown and stop RAM usage. Alternatively, to force shutdown use `taskkill /F /im wslservice.exe`.

C.2 Packages used

```python
# Calculations
import numpy as np
import dedalus
import dedalus.public as d3
import scipy.special as sc

# General
import logging
import time
import warnings
import copy
from functools import partial

# File handling
import h5py
import json

# Plotting
import matplotlib.pyplot as plt
import matplotlib.animation
from IPython.display import HTML
from IPython.display import clear_output

matplotlib.rcParams.update({'font.size': 16})
print('Dedalus v', dedalus.__version__)
logger = logging.getLogger(__name__)
```

The versions of these packages are as follows.

- dedalus 3.0.0a
- numpy 1.24.3
- matplotlib 3.7.1
- scipy 1.10.1
- json 2.0.9
- logging 0.5.1.2
- h5py 3.8.0
- IPython 8.14.0

C.3 Helper functions

```python
def im_plot(x_vals, y_vals, z, ax=None, filename=None, title=None, cax=None):
    ''' Make plot of z(x_vals, y_vals) from input arrays.
    Inputs: phi = azimuth (radians)
            r = radius
            z = function value as array (ensure dimension is phi-by-r)
            ax = ax to plot on
            title = plot title (=filename if title=None)
            cax = colourbar axis (False=disable)'''

    if ax is None:
        ax_set = False
        fig = plt.figure(figsize=(5, 5))
        ax = fig.gca()
    else:
        ax_set = True

    if title is not None:
        ax.title.set_text(title)
    else:
        ax.title.set_text(filename)

    with warnings.catch_warnings():
        warnings.filterwarnings('ignore', message=("The input coordinates"))
        im = ax.pcolormesh(x_vals, y_vals, z, edgecolors='face')

    ax.set_aspect('equal')

    if cax is None:
        plt.colorbar(im, ax=ax)
    elif cax is False:
        pass
    else:
        plt.colorbar(im, cax=cax)

    if ax_set is False:
        if filename is not None:
            fig.savefig(filename+'.png', dpi=fig.dpi, bbox_inches='tight')
        plt.show()
    return im


def polar_plot(phi, r, z, ax=None, filename=None, title=None, cax=None):
    ''' Make polar plot of z(phi,r) from input arrays.
    Inputs: phi = azimuth (radians)
            r = radius
            z = function value as array (ensure dimension is phi-by-r)
            ax = ax to plot on
            title = plot title (=filename if title=None)
            cax = colourbar axis (False=disable)'''

    phi_mesh, r_mesh = np.meshgrid(phi, r)
    x_vals = r_mesh * np.cos(phi_mesh)
    y_vals = r_mesh * np.sin(phi_mesh)

    im = im_plot(x_vals, y_vals, z, ax=ax,
                 filename=filename, title=title, cax=cax)
    return im


def animate(plot_func, t_list, pause=0):
    '''Make animation based on a plot function and time list'''
    for k, t in enumerate(t_list):
        clear_output(wait=True)
        plot_func(k, t)
        plt.pause(pause)
        plt.show()


def func_on_mesh(psi, phi, r):
    '''Return psi(phi, r) on meshed grid'''
    phi_mesh, r_mesh = np.meshgrid(phi, r)
    return psi(phi_mesh, r_mesh)
```

C.4 Poisson on a square

```python
def lap_square(Nx, Ny):
    '''
    Solve Laplace Equation on a square
        lap(u) = q;
        u(x=0) = u(x=1) = 0, BC.
        u(y=0) = u(y=1), periodic BC

    n = num basis fns
    '''

    # Bases
    coords = d3.CartesianCoordinates('x', 'y')
    dist = d3.Distributor(coords, dtype=np.float64)
    xbasis = d3.RealFourier(coords['x'], size=Nx, bounds=(-1, 1))
    ybasis = d3.Chebyshev(coords['y'], size=Ny, bounds=(-1, 1))

    # Fields
    psi = dist.Field(name='psi', bases=(xbasis, ybasis))

    # Forcing
    x, y = dist.local_grids(xbasis, ybasis)
    f = dist.Field(bases=(xbasis, ybasis))
    f['g'] = -2*np.pi**2 * np.sin(np.pi*x) * np.sin(np.pi*y)

    # Tau method
    tau_0 = dist.Field(name='tau_0', bases=xbasis)
    tau_1 = dist.Field(name='tau_1', bases=xbasis)

    def lift(tau, i):
        lift_basis = ybasis.derivative_basis()
        return d3.Lift(tau, lift_basis, i)

    # Problem
    problem = d3.LBVP([psi, tau_0, tau_1], namespace=locals())
    problem.add_equation("lap(psi) + lift(tau_0, -1) + lift(tau_1,-2) = f")
    problem.add_equation("psi(y=-1) = 0")
    problem.add_equation("psi(y=1) = 0")

    # Solver
    solver = problem.build_solver()
    solver.solve()

    # Gather global data
    x = xbasis.global_grid()
    y = ybasis.global_grid()
    psi_g = psi.allgather_data('g')

    clear_output(wait=True)

    # Plots and errors ###

    xx, yy = np.meshgrid(x, y)
    actual = np.sin(np.pi * xx) * np.sin(np.pi * yy)
    actual_field = dist.Field(bases=(xbasis, ybasis))
    actual_field['g'] = actual.T

    R = psi - actual_field

    fig, axs = plt.subplots(1, 3, figsize=(20, 6))
    im_plot(xx, yy, psi_g.T, ax=axs[0], title='Dedalus')
    im_plot(xx, yy, actual, ax=axs[1], title='Actual')
    im_plot(xx, yy, R.evaluate()['g'].T, ax=axs[2], title='Error')
    fig.tight_layout()
    plt.show()

    filename = 'Poisson on square'
    fig.savefig(filename+'.png', dpi=fig.dpi, bbox_inches='tight')

    err = np.sqrt(d3.integ(R**2)).evaluate()['g']
    print('Error:', err)
    print('Machine eps:', np.finfo(np.float64).eps)
    return err[0, 0]

```

```
lap_square(256, 256)
```

C.5 Basic polar solver

```
class DedalusSolver:
    '''
    Basic class for solver.

    IMPORTANT: implement make_problem and
    solve_problem in subclass.

    Useful methods:
    run = make_space + make_problem + solve_problem
          (__init__ executes run once)
    actual = set actual func
    compute_error = graph overview of error for saved run
    error_lists + error_plots = graph error for varying N
    '''
    def __init__(self, Nphi, Nr, Lr=1, *, dealias=1):
        # Export vars
        self.Nphi = Nphi
        self.Nr = Nr
        self.Lr = Lr
        self.dealias = dealias

        # Run
        self.make_space()

    def make_space(self):
        '''Setup Dedalus basis and field'''
        # Import vars
        Nphi = self.Nphi
        Nr = self. Nr
        Lr = self.Lr
        dealias = self.dealias

        # Parameters
        dtype = np.float64

        # Bases
        coords = d3.PolarCoordinates('phi', 'r')
        dist = d3.Distributor(coords, dtype=dtype)
        disk = d3.DiskBasis(coords, shape=(Nphi, Nr), radius=Lr,
                            dealias=dealias, dtype=dtype)  # Circular domain
        edge = disk.edge
        phi, r = dist.local_grids(disk)

        # Field
        u = dist.Field(name='u', bases=disk)

        # Export vars
        self.coords = coords
        self.dist = dist
        self.disk = disk
        self.edge = edge
        self.u = u
        self.phi = phi
        self.r = r

    def run(self, local=True, save_every=None, save_name=None):
        '''Fully run problem from scratch'''
        self.make_space()
        self.make_problem()

        time_0 = time.perf_counter()  # Start timer
        self.solve_problem(local=local, save_every=save_every, save_name=save_name)
        time_tot = time.perf_counter() - time_0

        self.time = time_tot

    def actual(self, actual_func):
        '''Return array of values of actual function on
        input of actual_func = type func'''
        # Import vars
        phi = self.phi
```

```python
        r = self.r
        Lr = self.Lr

        self.actual_func = actual_func
        scaled_func = partial(actual_func, Lr=Lr)
        return func_on_mesh(scaled_func, phi, r)

    def plot(self, z, ax=None, filename=None, title=None, cax=None):
        '''Generic plot via polar_plot. Input z'''
        # Import vars
        phi = self.phi
        r = self.r
        polar_plot(phi, r, z, ax=ax, filename=filename, title=title, cax=cax)

    def plot_result(self, ax=None, filename=None, title=None, cax=None):
        '''Plot of results using polar_plot'''
        z = self.ug.T
        self.plot(z, ax, filename, title, cax)

    def plot_actual(self, ax=None, filename=None, title=None, cax=None):
        '''Plot of actual using polar_plot'''
        z = self.actual(self.actual_func)
        self.plot(z, ax, filename, title, cax)

    def integral_error(self):
        '''Compute sqrt(int(computed-actual)^2)'''
        # Import vars
        disk = self.disk
        actual_func = self.actual_func
        phi = self.phi
        r = self.r
        u = self.u  # field
        dist = self.dist

        # Actual field
        actual = dist.Field(name='actual', bases=disk)
        actual['g'] = actual_func(phi, r)

        # Compute errors
        error = np.sqrt(d3.integ((u - actual)**2)).evaluate()['g']
        return error[0, 0]

    def compute_error(self, make_plots=True, filename=None):
        '''Plots and computes error of given functions by
        comparing with actual_func on mesh. Assumes problem
        is presolved.'''
        # Import vars
        Nr = self.Nr
        Nphi = self.Nphi
        ug = self.ug
        actual = self.actual(self.actual_func)

        errors = actual - ug.T
        weighted_norm = self.integral_error()

        if make_plots is True:
            print('Error:', weighted_norm)
            # Plotting
            fig, axs = plt.subplots(1, 3, figsize=(20, 6))

            for ax in axs:
                ax.ticklabel_format(style='sci')

            self.plot_result(ax=axs[0], title='Dedalus')
            self.plot_actual(ax=axs[1], title='Actual')
            self.plot(errors, ax=axs[2], title='Error')
            plt.tight_layout()
            plt.show()

            if filename is not None:
                fig.savefig(filename+'.png', dpi=fig.dpi,
                            bbox_inches='tight')

        # Export vars
        self.error = weighted_norm

    def error_lists(self, Nphi_list, Nr_list):
        '''Solve problem with various different Nphi and Nr
        (input as numpy lists, nb needs Nphi = 0 mod 4).
        Outputs lists of errors and times.'''
        # Import vars
        Nphi = self.Nphi
        Nr = self.Nr
        loop_solver = copy.copy(self)

        # Loop Nr
        loop_solver.Nphi = Nphi
        Nr_error_list = []
        Nr_times_list = []

        for Nr_loop in Nr_list:
            print('Nr =', Nr_loop)

            loop_solver.Nr = Nr_loop
            loop_solver.run()
            loop_solver.compute_error(make_plots=False)

            Nr_error_list.append(loop_solver.error)
            Nr_times_list.append(loop_solver.time)

        # Loop Nphi
        loop_solver.Nr = Nr
        Nphi_error_list = []
        Nphi_times_list = []

        for Nphi_loop in Nphi_list:
            print('Nphi =', Nphi_loop)

            loop_solver.Nphi = Nphi_loop
            loop_solver.run()
            loop_solver.compute_error(make_plots=False)

            Nphi_error_list.append(loop_solver.error)
            Nphi_times_list.append(loop_solver.time)

        print('Done!')

        # Export vars
        self.Nr_list = Nr_list
        self.Nphi_list = Nphi_list
        self.Nr_error_list = Nr_error_list
        self.Nr_times_list = Nr_times_list
        self.Nphi_error_list = Nphi_error_list
        self.Nphi_times_list = Nphi_times_list

    def error_plots(self, truncs=[20, 20]):
        '''Plots error and time lists from self.error_lists
           trunc = point to truncate best fit'''
        # Import vars
        Nr_list = self.Nr_list
        Nphi_list = self.Nphi_list
        Nr_error_list = self.Nr_error_list
        Nr_times_list = self.Nr_times_list
        Nphi_error_list = self.Nphi_error_list
        Nphi_times_list = self.Nphi_times_list

        fig, axs = plt.subplots(1, 2, figsize=(12, 4))

        ax = axs[0]
        ax.plot(Nr_list, Nr_error_list, '-o')
        ax.set(xlabel="Number of basis functions $N_r$",
               ylabel="Normed errors")
        ax.set_yscale('log')
        ax.set_title(f"Varying $N_r$, $N_\phi={self.Nphi}$")

        trunc = truncs[0]
        coeffs = np.polyfit(Nr_list[:trunc], np.log(Nr_error_list[:trunc]), 1)
        poly = str(np.poly1d(coeffs, variable=r'$N_r$'))[2:]
        ax.plot(Nr_list[:trunc], np.exp(np.poly1d(coeffs)(Nr_list[:trunc])),
                label=f'exp({poly})')
        ax.legend()

        ax = axs[1]
        ax.plot(Nphi_list, Nphi_error_list, '-o')
        ax.set(xlabel=r"Number of basis functions $N_\phi$",
               ylabel="Normed errors")
        ax.set_yscale('log')
        ax.set_title(f"Varying $N_\phi$, $N_r={self.Nr}$")

        trunc = truncs[1]
        if trunc is not None:
            coeffs = np.polyfit(Nphi_list[:trunc],
                                np.log(Nphi_error_list[:trunc]), 1)
            poly = str(np.poly1d(coeffs, variable=r'$N_\phi$'))[2:]
            ax.plot(Nphi_list[:trunc],
                    np.exp(np.poly1d(coeffs)(Nphi_list[:trunc])),
                    label=f'exp({poly})')
            ax.legend()

        fig.tight_layout()
        plt.show()
```

C.6 Poisson on a circle

```
class Poisson_Circ(DedalusSolver):
    '''
    Subclass of `DedalusSolver`. Adds methods
    `make_problem` and `solve_problem`.

    Solve Laplace Equation on Unit Circle
        lap(u) - u/Ld^2 = q;
        u(r=1) = 0, BC.

    Input:
    Nphi, Nr = Grid spacing
    q_func(phi, r, Ld) = q

    'ug' outputs np.array. 'u' outputs dedalus object

    Returns phi, r, u
    '''
    def __init__(self, Nphi, Nr, q_func, *, Lr=1, dealias=1):
        # Export vars
        self.Nphi = Nphi
        self.Nr = Nr
        self.q_func = q_func
        self.Ld = np.inf
        self.Lr = Lr
        self.dealias = dealias

        # Run
        self.run()

    def make_problem(self):
        '''Make problem with Laplace equation'''
        # Import vars
        q_func = self.q_func
        Ld = self.Ld
        dist = self.dist
        disk = self.disk
        edge = self.edge
        coords = self.coords
        u = self.u
        Lr = self.Lr

        # Forcing
        phi, r = dist.local_grids(disk)
        q = dist.Field(bases=disk)
        q['g'] = q_func(phi, r, Lr=Lr)  # as input

        # Tau method
        tau_u = dist.Field(name='tau_u', bases=edge)

        def lift(A):
            lift_basis = disk.derivative_basis()
            return d3.Lift(A, lift_basis, -1)

        # Problem
        problem = d3.LBVP([u, tau_u], namespace=locals())
        problem.add_equation("lap(u) - u/(Ld**2) + lift(tau_u) = q")
        problem.add_equation("u(r=Lr) = 0")

        # Export vars
        self.problem = problem

    def solve_problem(self, local=None, save_every=None, save_name=None):
        '''Solve Laplace equation'''
        # Import vars
        dist = self.dist
        disk = self.disk
        problem = self.problem
        u = self.u

        # Solver
        solver = problem.build_solver()
        solver.solve()

        # Gather global data
        phi, r = dist.local_grids(disk)
        ug = u.allgather_data('g')
        print('Done!')

        clear_output(wait=True)

        # Export vars
        self.u = u
        self.ug = ug
        self.phi = phi
        self.r = r
```

C.6.1 Bessel solution

```
def bessel_q(phi, r, n=3, Lr=1):
    '''q arrising from bessel'''
    r = r/Lr  # Scale r
    a = sc.jn_zeros(n, 1)[0]
    q = (((a**2)/4) * np.sin(n*phi)) \
        * (sc.jn(n-2, a*r) - 2*sc.jn(n, a*r) + sc.jn(n+2, a*r)) \
        + ((a/(2*r)) * np.sin(n*phi)) * (sc.jn(n-1, a*r) - sc.jn(n+1, a*r)) \
        - (((n**2)/(r**2)) * np.sin(n*phi)) * sc.jn(n, a*r)
    return q/Lr**2


def bessel(phi, r, n, Lr=1):
    '''Return Bessel values'''
    r = r/Lr  # Scale r
    a = sc.jn_zeros(n, 1)[0]
    z = np.sin(n*phi) * sc.jn(n, a*r)
    return z


# Params
n = 3  # Bessel order
Nphi, Nr = 2**8, 2**8
Lr = 1

bessel_lap = Poisson_Circ(Nphi, Nr, partial(bessel_q, n=n), Lr=Lr)
bessel_lap.actual(partial(bessel, n=n))
bessel_lap.compute_error()
```

```
bessel_lap.Nr, bessel_lap.Nphi = 14, 8

Nr_list = np.arange(4, 20, 1, dtype=int)
Nphi_list = np.arange(4, 24, 4, dtype=int)

bessel_lap.error_lists(Nphi_list, Nr_list)
bessel_lap.error_plots([11, 2])  # Truncate line of best fit
```

C.7 Time-dependent polar solver

```python
class time_PDE(DedalusSolver):
    '''
    Subclass of `DedalusSolver`.

    IMPORTANT: create make_problem manually in subclass.

    Useful methods:
    time_plot = plot PDE at specific times
    animate = create video over all time
    solve_problem
    '''
    def __init__(self, Nphi, Nr, initial_func, *,
                 Lr=1, dealias=2,
                 timestepper=d3.SBDF2, stop_sim_time=np.pi/2, timestep=0.1,
                 local=True, save_every=1, save_name=None, scales=1,
                 import_previous=False, variable_name=None, **kwargs):
        # Export vars
        self.Nphi = Nphi
        self.Nr = Nr
        self.initial_func = initial_func
        self.Lr = Lr
        self.dealias = dealias
        self.timestepper = timestepper
        self.stop_sim_time = stop_sim_time
        self.timestep = timestep
        self.local = local
        self.save_every = save_every
        self.save_name = save_name
        self.import_previous = import_previous
        self.variable_name = variable_name
        self.scales = scales
        self.__dict__.update(kwargs)

        # Run
        if import_previous is False:
            self.run(local=local, save_every=save_every, save_name=save_name)
        else:
            with open(f'{save_name}/params.json') as json_file:
                data = json.load(json_file)
                self.__dict__.update(data)

    def solve_problem(self, *, local=True, save_every=None, save_name=None):
        '''Solve PDE with given timestepper
        local = True : outputs to variable q_list
        local = False: outputs to file=filename (default classname + time)
        sim_dt: output every sim_dt time
        '''
        # Import vars
        dist = self.dist
        disk = self.disk
        problem = self.problem
        q = self.q
        t = self.t
        timestep = self.timestep
        stop_sim_time = self.stop_sim_time
        save_every = self.save_every

        time_str = time.strftime("%Y-%m-%d %H:%M:%S", time.localtime())
        if save_name is None:
            save_name = 'Snapshots ' + time_str + ' ' + self.__class__.__name__

        sim_dt = save_every * timestep

        # Solver
        solver = problem.build_solver(self.timestepper)
        solver.stop_sim_time = stop_sim_time

        # Main loop (external)
        if local is False:
            snapshots = solver.evaluator.add_file_handler(save_name,
                                                          sim_dt=sim_dt)
            snapshots.add_tasks(solver.state, layout='g', scales=self.scales)

            while solver.proceed:
                solver.step(timestep)
                if solver.iteration % 100 == 0:
                    logger.info('Iteration=%i, Time=%e, dt=%e'
                                % (solver.iteration, solver.sim_time,
                                   timestep))
            end_time = time.strftime("%Y-%m-%d %H:%M:%S", time.localtime())
            logger.info('Done!')
            clear_output(wait=True)

            # Export vars
            self.sim_ind_list = range(int(stop_sim_time // sim_dt))
            self.save_name = save_name
            self.execution_time = time_str
            self.execution_end_time = end_time
            self.import_previous = True

            with open(f'{save_name}/params.json', 'w') as file:
                json.dump(self.__dict__, file, default=str)

        # Main loop (local)
        if local is True:
            q.change_scales(scales=self.scales)
            q_list = [np.copy(q['g'])]
            t_list = [solver.sim_time]
            while solver.proceed:
                solver.step(timestep)
                if solver.iteration % 100 == 0:
                    logger.info('Iteration=%i, Time=%e, dt=%e'
                                % (solver.iteration, solver.sim_time,
                                   timestep))
                if solver.iteration % save_every == 0:
                    q.change_scales(scales=self.scales)
                    q_list.append(np.copy(q['g']))
                    t_list.append(solver.sim_time)

            logger.info('Done!')
            clear_output(wait=True)

            # Export vars
            self.sim_ind_list = range(int(stop_sim_time // sim_dt))
            self.q_list = q_list
            self.t_list = t_list

    def time_plot(self, plot_t_list=[0, .5, 1], filename=None):
        '''Plots PDE over time. `plot_t_list` is list
        of normalised time in [0,1] to be plotted.'''
        if self.import_previous is True:
            var_name = self.variable_name
            time_plot_file(self.save_name, plot_t_list=plot_t_list,
                           variable_name=var_name, filename=filename)
        else:
            # Import vars
            Nr = self.Nr
            Nphi = self.Nphi
            q_list = self.q_list
            t_list = self.t_list

            # Plotting
            plot_length = len(plot_t_list)
            fig, axs = plt.subplots(1, plot_length, figsize=(20, 6))

            for i, t in enumerate(plot_t_list):
                t_ind = int(t * (len(t_list) - 1))
                self.plot(q_list[t_ind].T, ax=axs[i],
                          title=f'Time = {t_list[t_ind]:.3}')
            fig.tight_layout()
            plt.show()

            if filename is not None:
                fig.savefig(filename+'.png', dpi=fig.dpi,
                            bbox_inches='tight')

    def animate(self, frames=None):
        '''Animate PDE over time with global animate function'''
        if self.import_previous is True:
            var_name = self.variable_name
            animate_file(self.save_name, variable_name=var_name, frames=frames)
        else:
            q_list = self.q_list
            t_list = self.t_list
            t_steps = len(t_list)

            if frames is None:
                frames = range(t_steps)

            def plot_func(k):
                self.plot(q_list[k].T, ax=ax[0], title=
                          f'Time: {t_list[k]:.3f}', cax=ax[1])

            fig, ax = plt.subplots(1, 3, width_ratios=[10, 1, 1],
                                   figsize=(6, 5))
            ax[2].axis('off')
            ani2 = matplotlib.animation.FuncAnimation(fig, plot_func,
                                                      frames=frames)
            plt.close()

            video = HTML(ani2.to_jshtml())
            display(video)
```

C.7.1 Writing to drive

We are storing data as float64 objects. These take 8 bytes each (this can be checked by running `np.float64(0.0).itemsize`). With a collocation grid of 256×256, each snapshot takes over 0.5 MB. For long periods of time, the memory usage for the collection of all snapshots increases, and it becomes unreasonable to store all values to RAM. Dedalus implements a file handler which can regular write snapshots to HDF5 files [6]. If the attribute `local` is `False`, the snapshots will not be saved in lists like above, but instead in HDF5 files using the following alternative solver loop instead of Section 6.3.

```
snapshots = solver.evaluator.add_file_handler(save_name,
                                              sim_dt=sim_dt)
snapshots.add_tasks(solver.state, layout='g', scales=self.scales)

while solver.proceed:
    solver.step(timestep)
    if solver.iteration % 100 == 0:
        logger.info('Iteration=%i, Time=%e, dt=%e'
                    % (solver.iteration, solver.sim_time,
                       timestep))
```

We no longer need to manually save snapshots inside the loop. Instead `solver.evaluator.add_file_handler` handles the saving of snapshots every `sim_dt` time automatically for us. The method `snapshots.add_tasks(solver.state)` tells the file handler to save all fields in the snapshot. Dedalus creates a folder with name `save_name`, which within stores files with extension `_s1.h5`. There are options to split the save into multiple different files by passing `iter=10` into `solver.evaluator.add_file_handler` (which stores 10 snapshots in each file) [6].

C.7.2 Reading from drive

```
def load_snapshot(save_name, file_num=1, index=0,
                  variable_name='q'):
    with h5py.File(f"{save_name}/{save_name}_s{file_num}.h5",
                   mode='r') as f:
        # Load dataset
        q = f['tasks'][variable_name]
        t_list = q.dims[0][0]
        phi_list = q.dims[1][0]
        r_list = q.dims[2][0]

        return phi_list[:], r_list[:], q[index], t_list[index]
```

This function takes in a folder name `save_name` and file number (defaults to 1). It loads the snapshot of field `variable_name` with index given by `index`. This is done via `f['tasks'][variable_name]`. The coordinates φ and r and the time at the chosen index are also returned.

C.8 Solid body rotation

```python
class rotation_PDE(time_PDE):
    def make_problem(self):
        '''Make PDE problem:
        Advection Equation on Unit Circle
        dt(psi) + 0.5 * gradperp(r^2) @ grad(psi) = 0;
        where gradperp(.) = -skew(grad(.)) in d3'''
        # Import vars
        dist = self.dist
        disk = self.disk
        edge = self.edge
        coords = self.coords
        Lr = self.Lr

        # Overwrite fields from make_space
        psi = dist.Field(name='psi', bases=disk)
        t = dist.Field()

        # Substitutions
        phi, r = dist.local_grids(disk)

        u = dist.VectorField(coords, bases=disk)
        u['g'][0] = r
        u['g'][1] = 0

        # Problem
        problem = d3.IVP([psi], time=t, namespace=locals())
        problem.add_equation("dt(psi) = - u @ grad(psi)")

        # Initial conditions
        psi['g'] = self.initial_func(phi, r)

        # Export vars
        self.q = psi
        self.t = t
        self.problem = problem
        self.variable_name = 'psi'


# Params
Nphi, Nr = 2**7, 2**7  # Grid spacing
n = 3  # initial bessel
time_step = np.pi/400  # Time step
stop_sim_time = 30*np.pi + time_step  # Simulation length
Lr = 1
timestepper = d3.SBDF3
save_every = 200
scales = 2


def initial_func(phi, r):
    return bessel_q(phi, r, n=n, Lr=Lr)


rotation_bessel = rotation_PDE(Nphi, Nr, initial_func, Lr=Lr,
                               stop_sim_time=stop_sim_time, timestep=time_step,
                               timestepper=timestepper,
                               local=False, save_every=save_every, scales=scales)
rotation_bessel.time_plot()
```

C.8.1 Computing errors

```python
save_name = 'Insert file name here'

Nphi, Nr = 2**7, 2**7  # Grid spacing
scales = 2
```

```

# Create field for error processing
coords = d3.PolarCoordinates('phi', 'r')
dist = d3.Distributor(coords, dtype=np.float64)
disk = d3.DiskBasis(coords, shape=(Nphi, Nr), radius=Lr,
                    dealias=2, dtype=np.float64)  # Circular domain
psi_init = dist.Field(name='u', bases=disk)
psi = dist.Field(name='u', bases=disk)
psi_init.change_scales(2)
psi.change_scales(2)

# Load first frame
phi, r, psi_initg, t = load_snapshot(save_name, variable_name='psi', index=0)
psi_init['g'] = psi_initg

max_i = int(stop_sim_time//(2*np.pi))
sim_dt = save_every * time_step
i_step = int((2*np.pi + .5) // sim_dt)
num_points = len(psi_initg.ravel())
print(psi_initg.shape)


errors = []
times = []

for i in range(0, max_i+1):
    phi, r, psig, t = load_snapshot(save_name, variable_name='psi', index=i*i_step)
    psi['g'] = psig
    E = psi_init - psi
    err = np.sqrt(d3.integ(E**2)).evaluate()['g']  # Integral error
    errors.append(err[0, 0])
    times.append(t)

fig, ax = plt.subplots(figsize=(6, 3))

ax.plot(np.array(times)/np.pi, errors, '-o')
ax.set_xlabel('Time')
ax.xaxis.set_major_formatter(plt.FormatStrFormatter('%g $\pi$'))
ax.set_ylabel('Normed errors')
plt.show()
```

C.9 Stommel–Munk problem

```
class stommel_PDE(time_PDE):
    def make_problem(self):
        '''Make PDE problem:
        Stommel Equation on Unit Circle
        dt(u) + 0.5 * gradperp(r^2) @ grad(u) = Q - r_0 lap(r^2);
        where gradperp(.) = skew(grad(.)) in d3'''
        # Import vars
        dist = self.dist
        disk = self.disk
        edge = self.edge
        coords = self.coords
        Lr = self.Lr
        r0 = self.r0
        F = self.F
        H = self.H
        beta = self.beta
        nu = self.nu
        rho0 = self.rho0
        Q_shift = self.Q_shift

        # Overwrite fields from make_space
        t = dist.Field()

        # Substitutions
        psi = dist.Field(name='psi', bases=disk)
        zeta = dist.Field(name='zeta', bases=disk)

        phi, r = dist.local_grids(disk)
        y = dist.Field(name='y', bases=disk)
        y['g'] = r * np.sin(phi)
```

```python

        Q = dist.Field(name='Q', bases=disk)
        Q['g'] = (F * np.pi / (rho0 * Lr * H)) *\
                 np.sin(np.pi * (r * np.sin(phi) + Lr*Q_shift) /Lr)


        # Tau method
        tau_zeta = dist.Field(name='tau_zeta', bases=edge)
        tau_psi = dist.Field(name='tau_psi', bases=edge)

        def lift(A, i=-1):
            lift_basis = disk.derivative_basis()
            return d3.Lift(A, lift_basis, i)

        # Problem
        problem = d3.IVP([zeta, psi, tau_zeta, tau_psi], time=t, namespace=locals())
        problem.add_equation("dt(zeta)  + r0 * lap(psi) - nu * lap(zeta) + lift(tau_zeta, -2) \
                             = -skew(grad(psi)) @ grad(zeta + beta * y)  + Q")
        problem.add_equation("lap(psi) - zeta + lift(tau_psi, -1) = 0")
        problem.add_equation("psi(r=Lr) = 0")
        problem.add_equation("zeta(r=Lr) = 0")

        # Initial conditions
        psi_init, zeta_init, t_init = self.initial_func
        zeta['g'] = zeta_init(phi, r)
        psi = self.initial_condition(psi, zeta)
        t['g'] = t_init

        # Export vars
        self.q = psi
        self.psi = psi
        self.zeta = zeta
        self.Q = Q
        self.y = y
        self.tau_zeta = tau_zeta
        self.tau_psi = tau_psi
        self.t = t
        self.problem = problem
        self.variable_name = 'psi'

    def initial_condition(self, psi, zeta):
        '''Set initial conds'''
        # Tau method
        tau = self.dist.Field(name='tau_zeta', bases=self.edge)

        def lift(A):
            lift_basis = self.disk.derivative_basis()
            return d3.Lift(A, lift_basis, -1)

        # problem
        ic_problem = d3.LBVP([psi, tau], namespace=locals())
        ic_problem.add_equation("lap(psi) + lift(tau) = zeta")
        ic_problem.add_equation("psi(r=self.Lr) = 0")

        # Solver
        solver = ic_problem.build_solver()
        solver.solve()
        return psi

# Params
Nphi, Nr = 256, 256  # Grid spacing
n = 2  # initial bessel
time_step = 6 * 60  # Time step
timestepper = d3.SBDF3
stop_sim_time = 60 * 60 * 24 * 365 * 3  # Simulation length
save_every = 10 * 24 * 7
Lr = 2e6
dealias = 2
scale = 3


# Params
def zeta_init(phi, r):
    phi_mesh, r_mesh = np.meshgrid(phi, r)
    return 1e-16 * partial(bessel, n=n, Lr=Lr)(phi_mesh, r_mesh).T
```

```python


constants = {
    'F' : 0.1,
    'H' : 500,
    'r0' : 2e-7,
    'beta' : 2e-11,
    'nu' : 80,
    'rho0' : 1000,
    'Q_shift' : 0.01}
initial_func = [None, zeta_init, 0]


stommel_bessel = stommel_PDE(Nphi, Nr,
                             initial_func,
                             dealias=dealias,
                             stop_sim_time=stop_sim_time,
                             timestep=time_step,
                             timestepper=timestepper,
                             Lr=Lr,
                             scales=scale,
                             local=False,
                             save_every=save_every,
                             **constants)
stommel_bessel.time_plot()
```

C.9.1 Evolution of vorticity Laplacian

```python
save_name = 'Insert file name here'
var_name = 'zeta'
inds = np.linspace(1, 52, num=52, dtype=int)

coords = d3.PolarCoordinates('phi', 'r')
dist = d3.Distributor(coords, dtype=np.float64)
disk = d3.DiskBasis(coords, shape=(256, 256), radius=2e6,
                    dealias=2, dtype=np.float64)  # Circular domain
edge = disk.edge
phi, r = dist.local_grids(disk)
q = dist.Field(name='q', bases=disk)

ave = []
for i in inds:
    phi, r, qg, t = load_snapshot(save_name, variable_name=var_name, index=i)
    q.change_scales(3)
    q['g'] = qg
    p = d3.lap(q).evaluate()  # p = lap(vorticity)
    p.change_scales(3)
    ave.append(np.average(np.abs(p['g'].T)))  # take unweighted average of p

fig, ax = plt.subplots(1, 1, figsize=(8, 5))
ax.plot(inds, ave)
ax.set_yscale('log')
fig.tight_layout()
plt.show()
```

C.10 Dealiasing

```python
def dealiasing_eg(Ns, psi, *, x=np.linspace(0, 1, num=2000), ax=None, legend=True):
    '''
    Ns = [N1, N2] = [number of basis fn, number of basis fn to dealias]
    psi = fn to approx
    x = domain
    '''
    pi = np.pi
    cos = np.cos
    sin = np.sin

    # Generate basis
    Bs = []
    for N in Ns:
        B = [np.ones_like(x)]
        for i in range(N):
            B.append(sin(pi*x*(i+1)))
            B.append(cos(pi*x*(i+1)))
        Bs.append(B)

    # Generate coeffs
    As = []
    for B in Bs:
        M = np.zeros((len(B), len(B)))
        b = np.zeros(len(B))

        for i, f in enumerate(B):
            for j, g in enumerate(B):
                M[i, j] = np.trapz(f*g, x)  # mass matrix
            b[i] = np.trapz(f*psi, x)  # rhs

        a = np.linalg.solve(M, b)  # Ma = b
        As.append(a)

    approx = As[0]@Bs[0]
    dealiasing = As[1][:len(Bs[0])]@Bs[1][:len(Bs[0])]

    # Plot
    if ax is None:
        ax = plt
    ax.plot(x, psi, '--', label='Actual', lw='3')
    ax.plot(x, approx, label='Approximation')
    s = Ns[1] / Ns[0]
    ax.plot(x, dealiasing, label='Dealiasing')
    ax.set_title('$\mathcal{B}$'+f'$_{{{Ns[0]}}}$ with dealiasing ${s:.1f}$')
    if legend:
        ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))
    return As


pi = np.pi
sin = np.sin
fig, axs = plt.subplots(1, 2, figsize=(12, 4))
dealiasing_eg([1, 2], (1 + 2*sin(pi*x))**2, ax=axs[0])
dealiasing_eg([5, 10], 10*sin(pi*x) + 4*sin(pi*3*x) + sin(pi*6*x) + sin(pi*10*x), ax=
    axs[1], legend=False)
fig.tight_layout()
plt.show()
```