Convection in Rotating Spherical Fluid Shell

Xiaole Jiang
Bates College, xjiang@bates.edu

Convection in Rotating Spherical Fluid Shell

Xiaole Jiang 姜晓乐

Department of Physics and Astronomy, Bates College, Lewiston, ME 04240

Convection in Rotating Spherical Fluid Shell

A Senior Thesis
Presented to the Department of Physics and Astronomy
Bates College
in partial fulfillment of the requirements for the
Degree of Bachelor of Arts
by

Xiaole Jiang 姜晓乐

Lewiston, Maine
May 5, 2021

Contents

Acknowledgments

First and foremost, I'd like to thank my parents for supporting me along my path of interest and in my pursuit of physics, of course, without the opportunities they provided, I will not be here today as the person I am. I would also like to thank my thesis advisor Professor Jeffrey Oishi, for his supports along the way of this thesis, and for his taking time to explain other physics/non-physics stuff to me, in and outside of class. Professor Nathan Lundblad also supported me along at Bates, and in my attempt to develop physics as a career. I think we had some very enjoyable conversations (and I hope it is enjoyable for both parties), and I hope more is to come. Professor Catherine Whiting, who is no longer on campus, also really helped me along the way of developing my physics career. Apologies for these short and dry sentences that really does not adequately encapsulate my feelings. I guess my only hope lies in the charity of those who reads this acknowledgment. I am really feel grateful for my physics experience at Bates and enjoyed it thoroughly.

I should also like to thank the friends and friendships I had at Bates. These includes long gone seniors like the carriage house crew when I was a sophomore (special mention to the persons whose thesis I bind, who put me in their thesis acknowledgment, who I had German class with, who I played Dota with, who accidentally bumped into me at a ski resort, and who did not graduate that year :)). Abe and his associates who graduated last year, I believe I don't have to explicate too much here; and the one who graduated with a non-integer number of school years. The physics crew that graduated last year. Tyler, thank you for helping me drive and driving me from airport fresh man year, I really appreciate it. Dylan, Ben, Shane, Jacob, I really enjoyed our hang outs (among other people who currently live in 280). Two (use to be three) dudes who lives near me on my floor. Tony, who I share birthday with. And here I end my acknowledgment, for I am out of time.

Introduction

It has been recently observed that there is a change in the length of day with a period of roughly six years [**1**], while a longer period in the change of length of day is attributed to action within Earth's core, the six year period is theorized to be a product of the Earth's dynamo generation, see e.g.[**2**]. Where we can think of the Earth dynamo as a shell of rotating electrically conducting fluids, the outer core, with a non-rotating center, the solid inner core. It is conjectured that a torsional wave with a six year periodicity, i.e. an oscillation in the azimuthal velocity of the outer core that propagates radially outward could explain the change in the length of the day via conservation of angular momentum. Previous simulations have identified torsional waves in both dynamo simulations [**2**][**3**] and purely hydrodynamical simulations [**4**]. Here we explore torsional waves as a purely hydrodynamical process, modeling Boussinesq fluids in a rotating spherical shell, with no-slip boundary condition. The numerical simulations are done by solving the Navier-Stokes equation sets using spherical Dedalus, a pseudo-spectral partial differential equation solver. The schematic figure 0 nicely illustrates the geometry of the setup. I should also mention that though this problem is motivated by trying to understand Earth's dynamo, by no means are we actually modelling the Earth since a) we are not considering the magnetic field, i.e. we are not considering dynamo action, b) as of now, modeling the Earth is computationally impossible.

I start this thesis by considering the fundamentals of fluid dynamics in chapter 1, aiming at presenting the equations of motion for the simulation, and also deriving relevant quantities (Reynolds stress in this case) for data analysis. I then briefly consider how the tool that helps me solve differential equations work — chapter 2 concerns the underlying principle of Dedalus. In chapter 3 I present my simulation and some data that I have acquired, and in chapter 4 I discuss the implication of my results and potential future directions.

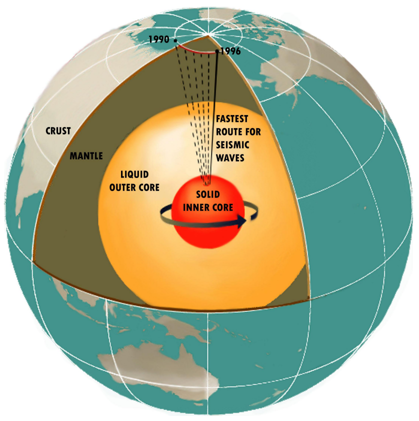

Figure 0. Image from NASA[**5**], showing the different layers of Earth core. We model the liquid outer core as a rotating spherical shell.

CHAPTER 1

Introduction to Fluid Dynamics

1. The Navier-Stokes Equation

Fluid dynamics studies macroscopic dynamics, its underlying is therefore undoubtedly Newtonian. As opposed to applying Newton's law to a single particle or a solid body, we now apply it to a fluid treated as a malleable continuum. The power of fluid dynamics rest on the fact that the fluid particles are so small that macroscopic properties, e.g. pressure, temperature, velocity of the flow, can be assigned to each point in space without ambiguity. The central governing equation of fluid dynamics is the Navier-Stokes equation (NS), given as the following:

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\frac{\nabla p}{\rho} + \nu \nabla^2 \mathbf{u} + \frac{\mathbf{F}}{\rho}. \tag{1.1}$$

Where $\mathbf{u}$ is the velocity of the fluid (a velocity field), p the pressure, ρ the density of the fluid, ν the kinematic viscosity, and $\mathbf{F}$ some external force acting on the fluid.

The left hand side is sometimes abbreviated as $\frac{D}{Dt} = \frac{\partial}{\partial t} + (\mathbf{u} \cdot \nabla)$, called the material derivative (or substantive derivative), where it describes how something changes with time as it moves along with a velocity $\mathbf{u}$. It is effectively the acceleration $d\mathbf{v}/dt$ in Newton's second law, where the coordinate system flows with the particle. In changing to a frame where the coordinate is stationary (Eulerian), the substantive derivative acquires its current form in Eq(1.1). The right hand side of Eq(1.1) is just the force on a parcel of fluid: pressure gradient, viscous force and external force. The pressure gradient can be thought of as the pressure difference on the surface of a parcel of fluid; the viscous force can be understood intutively as the fact that the faster flowing parcel will drag its slower moving parcel peers faster, and vice versa. Mathematically, the Laplacian is positive if a parcel of fluid if is slower than its peers, and vice versa. These two terms are both forces that arise purely due to fluid motion, so we also have to factor in possible external forces $\mathbf{F}$. In our case, it will just be the downward gravity $-g\hat{\mathbf{r}}$. This equation is also referred to as the momentum equation, since it concerns the conservation in momentum.

It should be noted that this is a simplified version of the NS equation, where we have constant viscosity and incompressible fluid, meaning the density of a parcel of fluid is constant. The mathematical form of incompressible fluid flow is typically captured in the continuity equation of the velocity field, which is derived from considering the conservation of mass:

$$\boldsymbol{\nabla} \cdot \mathbf{u} = 0 \tag{1.2}$$

This equation is no stranger to the physics student, as it shows up from electrodynamics to quantum mechanics.

Comparing the NS equation to the usual equations of motion that we see in Newtonian mechanics, or any other typical physics course taught here, we realize the term $u \cdot \nabla$ is unique, and indeed, this operator give rise to non-linear terms which creates much chaos in solving the equations.

2. Convection and the Boussinesq approximation

We have just considered pure fluid equations, namely the NS equation and the continuity equation. This, however, is insufficient in describing the full dynamics of our problem. Indeed, fluid flows of uniform properties and without significant body forces can be considered using only those two equations [**6**], such as simple laminar flow or flow around a sphere, but one crucial phenomena, of both practical and theoretical importance, is yet to be considered: convection. Intuitively, we know that fluid flow carries (advects) certain properties with them, such as temperature; conversely, temperature affects the density of the fluid, as with the case of air, where hot air rises and cool air falls. Moreover, we know that a candle flame have its shape precisely because of this phenomena coupled with the Earth's gravitational field; many pictures can be found on the internet that will demonstrate the otherwise spherical nature of the candle flame, e.g. [**7**]. The smoke patterns formed atop of a lit candle or a fire place is also complex and interesting precisely because of how temperature couples with density (given a gravitational field). This is the phenomenon of convection, and in this example, the temperature gradient provides an external force for fluid flow, which in turn advects the temperature. In general, temperature gradient is not necessary for convection, as the mathematics is the same for fluids with a concentration gradient, but for our problem setup, that is what we are concerned with. For our problem, we adopt what is known as the Boussinesq approximation, in which, as before, we ignore density variation unless they give rise to a gravitational force. The following derivation partly follows Tritton (1988) [**6**]. Since we now allow variation in density, we will decompose it into a reference density and a fluctuation density

$$\rho = \rho_0 + \Delta\rho.$$

and explicitly treat the force as gravitational. Since we said we will not ignore density variation due if they give rise to gravitational variation, the force term is now

$$\mathbf{F} = \rho\mathbf{g}.$$

We can write the gravitational acceleration as a potential

$$\mathbf{g} = -\nabla\Phi \implies \mathbf{F} = -\nabla(\rho_0\Phi) + \Delta\rho\mathbf{g}.$$

Recall the NS equation Eq(1.1), we substitute ρ_0 for all ρ in the equation, because we ignore density fluctuations outside of force, and put in the expression for force, we have

$$\frac{\partial\mathbf{u}}{\partial t} + (\mathbf{u}\cdot\nabla)\mathbf{u} = -\frac{\nabla P}{\rho_0} + \nu\nabla^2\mathbf{u} + \frac{\Delta\rho}{\rho_0}\mathbf{g} \tag{1.3}$$

Where $P = p + \rho_0\Phi$.

Now we will do the same for temperature, and name the reference temperature T_0:

$$\text{Temperature} = T_0 + T$$

Since the variation is small we can approximate a linear relationship between the density and temperature fluctuation

$$\frac{\Delta\rho}{\rho_0} = -\alpha\frac{T}{T_0} \tag{1.4}$$

where α is the coefficient of expansion of the fluid. This will be substituted back into Eq(1.3), and this terms is typically called the bouyancy term.

Now we need a equation for temperature as well, we do not have a internal heat source in our set up, so we could use the heat equation for our temperature variation, and change its time derivative to the substantive derivative (changing from Lagrangian to Eulerian coordinates) to account for fluid motion (advection), and have

$$\frac{\partial T}{\partial t} + (\mathbf{u} \cdot \nabla) T = \kappa \nabla^2 T \tag{1.5}$$

where κ is the thermal diffusivity. The Laplacian term is interpreted as diffusion (so if we call this temperature diffusion, we can call viscosity momentum diffusion). This equation is also sometimes called the energy equation, as it expresses conservation in energy.

Combining the three equations we have, expressing conservation of momentum, mass and energy, we have the full set of Boussinesq equations [**8**]:

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\frac{\nabla P}{\rho_0} + \nu \nabla^2 \mathbf{u} + \frac{\alpha g}{T_0} T \hat{\mathbf{r}} \tag{1.6}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{1.7}$$

$$\frac{\partial T}{\partial t} + (\mathbf{u} \cdot \nabla) T = \kappa \nabla^2 T \tag{1.8}$$

3. Rotating Fluids

Since our problem stems from considering Earth core, we also need to consider the differential rotation between the inner and outer core of the Earth. From Newtonian mechanics, we know that if we are in a rotating frame, as it is often convenient to be in such a frame, we will have non-inertial forces, also known as fictitious forces. The derivation amounts to considering how the unit vectors rotate with time, here we will simply state the form of the non-inertial forces per mass:

$$\mathbf{a}_{\text{inertial}} = \mathbf{a}_{\text{rot}} + 2\mathbf{\Omega} \times \mathbf{u}_{\text{rot}} + \mathbf{\Omega} \times (\mathbf{\Omega} \times \mathbf{r}) + \frac{\mathrm{d}\mathbf{\Omega}}{\mathrm{d}t} \times \mathbf{r} \tag{1.9}$$

where $\mathbf{\Omega}$ is the angular velocity, pointing in the direction of the axis of rotation, the subscript indicate which frame the quantity is measured in, and it should be noted that the radial vector does not change from the rotational frame to the stationary frame. The first Ω term is the Coriolis force, the second is the centrifugal force, and the third is the Euler force.

For our problem, two relevant features are: rotation is constant, so there is no Euler force, and we should notice that the centrifugal term can be written as a gradient of a scalar:

$$\mathbf{\Omega} \times (\mathbf{\Omega} \times \mathbf{r}) = -\nabla \left(\frac{1}{2} \Omega^2 r'^2 \right) \tag{1.10}$$

where Ω is the (constant) angular velocity and r' is distance measured from the axis of rotation, i.e. the radial distance in cylindrical coordinates, $r' = r \cos\theta$ where θ is the polar angle in spherical coordinates.

This means we can absorb the centrifugal term into the pressure term, like how we put the gravitational potential there. One might be worried about its effect, since at least the gravitational potential is constant, and this term is not. But notice that there is no dynamics for pressure, meaning that it is not a variable to be solved, and since incompressibility is a purely spatial differential equation, the pressure is typically interpreted as a constraint to satisfy incompressibility, indicating that no dynamics will be changed by letting the pressure absorb potentials.

Adding the Coriolis term to Eq(1.6) gives us the final form of NS equation that we will use:

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\frac{\nabla P}{\rho_0} + \nu \nabla^2 \mathbf{u} + \frac{\alpha g}{T_0} T \hat{\mathbf{r}} - 2\Omega \hat{\mathbf{z}} \times \mathbf{u} \tag{1.11}$$

where we define the rotation to be in the z direction with angular velocity Ω. This is the same dynamics for a non-rotating body with an extra force, so all velocity will be referenced in the rotating frame. Now we have all the dynamical equations that we will be solving.

4. Turbulence

So far we have only considered the equations of motion and have not talked about potential solutions to those equations. As we have alluded to earlier, the NS equations are notorious to solve partly because of their non-linear nature, and partly because partial differential equations are just hard. Nevertheless, simple solutions are possible, and I will qualitatively introduce some of them in order to contrast it with the turbulence phenomenon that we are studying.

Consider fluid flowing in a pipe, it turns out that if the pipe is narrow or the fluid is slow, the fluid will flow in a way that is "orderly," the flow is more commonly known as laminar flow. One characterization would be that given some point $\mathbf{x}_0$, the flow velocity at that point $\mathbf{u}(t, \mathbf{x}_0)$ roughly stays the same as time passes. Common examples include water flowing out of a thin pipe, or when one pours drink slowly out of a bottle. A more formal characterization of this involves the Reynolds number, a non-dimensional number the magnitude of which indicates how laminar (low Re) or turbulent (high Re) the flow is.

Of course, when the flow is not laminar, it is turbulent. Invoking the language of examining the velocity field at a point, we can say that a turbulent flow is when $\mathbf{u}(t, \mathbf{x}_0)$ behaves chaotically. Generally, a turbulent flow is one with velocity which have "rapid, irregular fluctuations in both space and time" [**6**]. The fluid motions that this thesis is concerned with are all turbulent, plots in chapter 3 will make this point evident. Similar to the Reynolds number, for our system, we can define a Rayleigh number, which typically tells us how much heat is transported via convection as opposed to diffusion. Convection, intuitively, brings turbulence to the system. Refer to chapter 3 for the precise definition of Raleigh number that we employ.

I will also take this opportunity to briefly talk about instability. A state is stable if it goes back to what it was after a small disturbance is introduced. An unstable state, on the other hand, will amplify the introduced disturbance, and quite often this will lead the transition from laminar flow to turbulent flow.

Going back to turbulent flow, since the macroscopic properties are chaotic at each point, it is more insightful for us to talk about statistical properties of the solutions rather than solutions themselves, as one could imagine that a slight change in the initial condition would result in vastly different looking solutions, despite the fact that we would expect those solutions to have some common characteristics. Considering that we are numerically solving these equations, it is unavoidable that some small numerical error will be introduced, and thus it only makes sense to consider the statistical properties of the solution. It makes sense, then, to introduce averaging methods.

Text books [**6**][**8**] defines an average as

$$Q_{\text{avg}} = \frac{1}{2s} \int_{-s}^{s} Q \, \mathrm{d}t \, . \tag{1.12}$$

where $2s$ is some time frame larger than the fluctuation time frame of Q. Operationally, however, we will define two spatial averages that we actually use in chapter 3 to process data, denoted using overbar and angle bracket respectively.

$$\overline{Q} = \frac{1}{2\pi}\int_0^{2\pi} Q\,\mathrm{d}\phi\,, \quad \langle Q\rangle = \frac{1}{h}\int_{-z}^{z} Q\,\mathrm{d}z\,. \tag{1.13}$$

The first equation is averaging over the angle ϕ, and the second is averaging over the height h along z in cylindrical coordinates.

Given some average quantity, it makes sense to talk about fluctuation. For us, we can decompose the velocity into a average component and a fluctuating component. Consider some averaging operation, $Q^* = \text{average}(Q)$, we can write

$$\mathbf{u} = \mathbf{u}^* + \mathbf{u}'.$$

Averaging the continuity equation Eq. 3.2, and using the fact that averaging and differentiation commute, we can show that

$$\boldsymbol{\nabla}\cdot\mathbf{u}^* = 0, \quad \boldsymbol{\nabla}\cdot\mathbf{u}' = 0 \tag{1.14}$$

We can perform the same operation on the NS equation, consider Eq. (1.1) without $\mathbf{F}$, we get

$$\frac{\mathrm{d}\mathbf{u}^*}{\mathrm{d}t} + (\mathbf{u}^*\cdot\nabla)\mathbf{u}^* + \overline{(\mathbf{u}'\cdot\nabla)\mathbf{u}'} = -\frac{1}{\rho}\nabla P + \nu\nabla^2\mathbf{u}^* \tag{1.15}$$

If we write the term with overbar in its vector component form, we have

$$\sum_i \overline{u_i'\frac{\partial}{\partial x_i}u_j'} = \sum_i \frac{\partial}{\partial x_i}\overline{u_i u_j} \tag{1.16}$$

This is not a straight forward step, we used a) gradient terms can be absorbed into pressure, and b) $\boldsymbol{\nabla}\cdot\mathbf{u}' = 0$. The point is so that we can now define the Reynolds stress R, which is a *tensor*, which can be represented as a matrix:

$$R = R_{ij} = \overline{u_i' u_j'} \tag{1.17}$$

The term in NS equation with overbar is then just the divergence of the Reynolds stress, also known as Reynolds force. Reynolds force can be interpreted as the force that the turbulent part of the fluid (the fluctuating part, $\mathbf{u}'$) have on the laminar part of the fluid (the mean flow $\mathbf{u}^*$).

We have covered the crucial concepts in fluid dynamics and introduced Reynold stress, which I will be using later on. We will now head to the numerical world for some understanding in numerical partial differential equation solvers.

CHAPTER 2

Spectral Methods and Dedalus

1. Introduction to Spectral Methods

Many numerical partial differential equation (PDE) solvers solve PDEs through transforming PDEs to ordinary differential equations and algebraic equations, which can then be solved through numerical matrix techniques. Different approaches achieve this differently, for example, the finite element method creates a mesh that discretizes space, and have low order interpolating functions on them, which transforms PDEs to ODEs. Spectral methods, on the other hand, does not discretize space per se. Instead, it uses the familiar analytic technique of solving PDEs — series expansion. Through expanding the solution as a series (usually in spatial coordinates), spectral methods reduces PDEs to sets of coupled ODEs which only depends on time, and can therefore be solved using simple ODEs solvers. One advantage of spectral methods, given that the solution is smooth and some natural function basis, is that it will converge exponentially, meaning it will be more precise than methods like finite difference and finite element which only converge algebraically. Spectral methods works better on regular geometries, like that of a plane, disk, cube, sphere; unlike other methods which uses arbitrary mesh, spectral methods effective uses the basis function to discretize space, and irregular geometry do not have nice discretization provided by some natural basis. Given that we are not expecting singularities for our fluid problem, and that we have a regular geometry (shell), it seems like spectral methods is the perfect fit, and we should expect exponential convergence. Below I will present the basic ideas behind spectral methods, and much of this chapter follows Boyd 2001 [**9**]. The point is to provide some foundation and justification for the reliability and accuracy of my simulations, as they are run with the pseudo-spectral solver Dedalus. For our purposes, we will use spectral and pseudo-spectral interchangeably.

Here, we will first work through a simple abstract example of applying the spectral methods following [**10**]. I should also note that the language might not be mathematically rigorous, despite my attempt to make it look formal, as this is only an example. Given some differential equation

$$H\psi(x) = f(x) \tag{2.1}$$

where H is some differential operator in x, $f(x)$ is some given smooth function, and $\psi(x)$ is the solution that we want to find. We expand $\psi(x)$ using some basis function $\phi_n(x)$ to N terms and call it $\psi_N(x)$ to indicate it is truncated at N

$$\psi_N(x) = \sum_{n=1}^{N} \hat{\psi}_n \phi_n(x) \quad \text{where} \quad N \in \mathbb{Z} \quad \text{and} \quad \langle \phi_n, \phi_m \rangle = \int_a^b \phi_n \phi_m \omega \mathrm{d}x = \delta_{nm} \tag{2.2}$$

here $\hat{\psi}_n$ is the coefficient of the basis, and we assume the basis to be orthogonal and complete where $\omega(x)$ is some weight function. I have used $\langle a, b \rangle$ to indicate the inner product of functions. The most common basis function would be the Fourier series, as the ubiquitous Fourier transform is just a inner product; other potential basis include the Hermite polynomials (solution to the quantum harmonic oscillator), the Cheybyshev polynomials, spherical harmonics, etc.

Now we can substitute our solution into our differential equation, and because we picked ϕ_n, $H\phi_n$ would just be some other known function $\Phi(x)$

$$\sum_{n=1}^{N} \hat{\psi}_n \Phi_n(x) = f(x). \tag{2.3}$$

Note that now if we pick N different x_i values to put into this equation, we will have N equations and N unknowns, with the unknowns being the coefficients $\hat{\psi}_n$. If we have some boundary condition, we can just replace some x_i with equations that enforce the boundary condition. Since these are just simultaneous equations, we can use matrices to solve them and acquire the solution to the differential equation by transforming the function $\psi(x)$ back from the coefficient space. This method of picking some points and solving for the coefficient is called the collocation method, and the points picked are the collocation points. The method is relatively simple and accurate (when the right points are picked) but have the downside of having to solve a dense matrix, which is slow. Later, we will talk about the Galerkin tau method that Dedalus uses, which gives us a sparse matrix and considerably speeds up the process.

One might be interested in the choice of basis functions and collocation points. For collocation points, there are optimal sets for each basis, and one could refer to [**9**], though I find it interesting that the density of collocation points seem to correlate to the weight function of the basis. For basis functions, it turns out there are some natural sets, for example, if the boundary of the problem is periodic then one should consider using Fourier basis, and if the problem have spherical symmetry then one should consider using spherical harmonics, and apparently Chebyshev works fine in all other cases. In general, when choosing a basis, one considers the convergence rate, whether fast Fourier transform can be employed, and other possible simplifications and tricks related to the basis that could simplify computation, e.g. recurrence relations to simplify the derivative, etc.

2. Bound on Fourier Series Error

Considering that exponential convergence is a selling point for spectral methods, I feel compelled to provide some description. However, a general proof of the exponential, or geometric, convergence of Fourier series is quite technical, and, in general, the series does not converge exponentially. Here, I will sketch an outline of a proof for exponential convergence for Fourier series of a smooth function that is periodic and whose derivatives are also periodic.

Consider some function $f(t)$, we will expand it as a Fourier series

$$f(t) = \sum_{n=-\infty}^{\infty} \hat{f}_n e^{i\omega_n t} = \sum_{|n|\leq N} \hat{f}_n e^{i\omega_n t} + \sum_{|n|>N} \hat{f}_n e^{i\omega_n t}. \tag{2.4}$$

We will call the first term on the right hand side the truncated Fourier series $f_N(t)$, which is what we have when we use the spectral methods. If we represent $f(x)$ using $f_N(x)$, it follows immediately that

$$\text{Error} = f(t) - f_N(t) = f(t) - \sum_{|n|\leq N} \hat{f}_n e^{i\omega_n t} = \sum_{|n|>N} \hat{f}_n e^{i\omega_n t}. \tag{2.5}$$

To find the magnitude, we take the absolute value

$$|\text{Error}| = \left| \sum_{|n|>N} \hat{f}_n e^{i\omega_n t} \right| \leq \sum_{|n|>N} \left| \hat{f}_n \right| \tag{2.6}$$

where the last step we used the fact that the complex exponential is bounded by 1.

We now seek a bound on $\hat{f}_n$. Recall how we can calculate $\hat{f}_n$

$$\hat{f}_n = \int f(t) e^{i\omega_n t} \mathrm{d}t. \tag{2.7}$$

Now, we will use our assumption that $f(t)$ and all its derivatives are periodic from $-\pi$ to π, and that the k-th derivative, $f^{(k)}(t)$ is integrable. Going back to $\hat{f}_n$, we will now be performing integration by parts, repeatedly differentiating $f(t)$ under the integral and integrating the exponential.

$$\hat{f}_n = \int_{-\pi}^{\pi} f(t) e^{i\omega_n t} \mathrm{d}t = -\left[\frac{f(t)}{in} e^{i\omega_n t} \right]_{-\pi}^{\pi} - \frac{i}{n} \int_{-\pi}^{\pi} f'(t) e^{i\omega_n t} \mathrm{d}t \tag{2.8}$$

The first term on the right hand side disappears because both the function and the complex exponential are periodic, and we can perform integration by parts for k times to get

$$\hat{f}_n = \left(-\frac{i}{n} \right)^k \int_{-\pi}^{\pi} f^{(k)}(t) e^{i\omega_n t} \mathrm{d}t. \tag{2.9}$$

Because complex exponential is bounded by 1, the integral is just going to be some constant, hence we can see that $\hat{f}_n = \mathcal{O}(n^{-k})$. Therefore, for a periodic analytic function, $\hat{f}_n = \mathcal{O}(e^{-n})$. In general, the more derivative f have, the faster the fourier series converge. This is also true beyond the Fourier basis. A simple example would be the Chebyshev polynomials, which is effectively just a cosine series (one need to reduce the cosine and write it in x).

$$T_n(\cos\theta) = \cos(n\theta) \tag{2.10}$$

3. The Assumption of Equal Errors

For every approximation method, we should ask, what are the errors, and are they small enough? Here I will briefly outline several form of error for the spectral methods, and state a rule of thumb "the assumption of equal errors." This is directly from a section of Boyd 2001 [**9**].

So far, we have shown that spectral methods converges exponentially for infinitely differentiable periodic functions. Alternatively, this is saying that the error made by neglecting all spectral coefficients $\hat{\psi}_n$ for $n > N$ decays exponentially. This form of error is called the *truncation error.*

The discretization error is defined as the difference between the first N terms of the exact solution and the corresponding terms as computed by spectral methods using N basis functions.

The interpolation error is the error made by approximating a function by an N-term series whose coefficients are chosen to make the approximation agree with the target function exactly at each of N collocation points, or some equivalent form thereof, for example, those present in the Galerkin Tau method.

In general, there are no ways to know the errors precisely for an unknown solution to a differential equation. Estimates can be made through different methods, for example one could look at the spectral coefficients as computed, or use model functions to estimate truncation error. In general, however, we can follow the rule of thumb "the assumption of equal errors" which states that these three error sources contributes roughly the same amount of error. It is only an empirical observation, it is not guaranteed to work, but it is comforting to know that in general, if we have exponentially converging truncation error (which we often do for smooth solutions), our other sources of error will also be equally small.

4. Dedalus

4.1. Galerkin Tau Method. As we have hinted at earlier, the collocation method is slow because it often results in a dense matrix. Dedalus uses a different method, namely the Galerkin Tau method, which when combined with clever choices of test functions result in sparse matrices. We will introduce the Galerkin Tau method by generalizing the collocation method. Recall Eq. (2.3), where we want to solve the $\hat{\psi}_n$ by solving the simultaneous equations

$$\sum_{n=1}^{N} \hat{\psi}_n H\phi_n(x_i) = f(x_i), \quad i = 1, 2, \cdots, N \tag{2.11}$$

given some set of x_i and some complete orthogonal basis $\phi_n(x)$. For simplicity, we also take $\phi_n(x)$ to satisfy the boundary conditions so we don't have to worry about it. What we are really doing is defining an error, and minimizing it according to some metric of smallness.

This can be spelled out more explicitly

$$\text{Error} = R \equiv \sum_{n=1}^{N} \hat{\psi}_n H\phi_n(x) - f(x). \tag{2.12}$$

Now let's just try to take the inner product of R with delta function $\delta(x - x_i)$

$$\langle \delta(x - x_i), R \rangle = \sum_{n=1}^{N} \hat{\psi}_n H\phi_n(x_i) - f(x_i). \tag{2.13}$$

where the inner product is the usual integral with respect to some weight $\omega(x)$.

Notice that this is the same as equation 2.11

$$\langle \delta(x - x_i), R \rangle = 0 \iff \text{Eq. } 2.11 \tag{2.14}$$

for some set of delta functions.

We have hence recovered the collocation method, but of course, instead of $\delta(x-x_i)$ functions we can use other sets of functions, $w_i(x)$, named test functions, and write that in general, the coefficient $\hat{\psi}_n$ of the solution ψ under some spectral decomposition is

$$\langle w_i(x), R \rangle = 0, \quad i = 1, 2, \cdots, N \tag{2.15}$$

and $w_i(x) = \phi_i(x)$ is the Galerkin method. Intuitively, we are specifying "error distribution". When we use a delta function as the test function, we are saying that we want no error at particular points in the domain, and if we choose other test functions, those test functions would specify the error distribution. For example, if we choose $w_i(x) = x^i$, we are minimizing the mean, variance, $\cdots$, etc. of the error function; for some other w_i, we can minimize the norm of the error, where norm is the square root of the inner product. For the Galerkin method, we are saying that we want the coefficient of the spectral expansion of the error function to be zero for $i \in \{1, 2, \cdots, N\}$. In general, this is called the mean weighted residual methods.

In order to see that this could result in a sparse matrix for some clever choice of $w_i(x)$, we need to put this equation in matrix form. We will define a column vector $\hat{\psi}_i$ whose entries are the N spectral coefficients, a matrix H_{ij} and a f_i vector which are given below

$$H_{ij} = \langle \phi_i, H\phi_j \rangle, \quad i, j = 1, 2, \cdots, N \tag{2.16}$$

$$f_i = \langle \phi_i, f \rangle, \quad i = 1, 2, \cdots, N \tag{2.17}$$

where H is the operator for the differential equation and f the RHS function.

If we look at the matrix equation

$$\sum_j H_{ij} a_j = f_i \implies \sum_j \langle \phi_i, H\phi_j \rangle \hat{\psi}_j - \langle \phi_i, f \rangle = 0 \tag{2.18}$$

and notice that

$$\sum_j \langle \phi_i, H\phi_j \rangle \hat{\psi}_j = \langle \phi_i, \sum_j \hat{\psi}_j H\phi_j \rangle = \langle \phi_j, \psi_N \rangle$$

we see that if we combine the inner product, we just have

$$\sum_j H_{ij} a_j = f_i \implies \langle \phi_j, \psi_N - f \rangle = 0 \implies \langle \phi_j, R \rangle = 0. \tag{2.19}$$

Now we see that just like the collocation method is about solving a set of simultaneous equations, so too is the Galerkin method: we aim at solving the matrix equation Eq.2.18. Effectively, we want to invert the matrix H. H is a hard matrix, as each of the element of the matrix is an integral, and in general matrix inversion is expensive. For certain cases, however, H can be easy to compute and invert.

Consider expanding $\psi(x)$ in terms of a cosine series, and $H = \frac{\mathrm{d}^2}{\mathrm{d}x^2}$. We realize that applying H on the basis, i.e. the cosine series, just gives back the basis function multiplied by some constant, and since the cosines are orthogonal to each other, the matrix H will be diagonal. Here, we found H without having to compute any integrals, and since H is diagonal, inversion is trivial.

Dedalus implements something like this, where test functions are chosen in such a way as to utilize the orthogonal relation as much as possible, even when the H operator is not as convenient as it is in my example. Thus speeding up the computation, as now we only have to solve a sparse matrix, as opposed to a dense matrix.

We have assumed that the boundary condition is satisfied through our basis expansion, but in the case that it is not, we can solve

$$H\psi(x) + \tau P(x) = f(x) \quad \text{instead of} \quad H\psi(x) = f(x) \tag{2.20}$$

where this extra variable τ provides the extra degree of freedom to match the boundary conditions, and $P(x)$ is typically a specific polynomial. We know that τ will be small because inside the boundary, $H\psi(x)$ converges to $f(x)$, so τ is forced to be small. This is the tau-method. See [**10**] for more information regarding Dedalus's methodology and implementation.

4.2. Spherical Harmonics. For my simulations the solution fields (e.g. velocity) are expanded in terms of a Cheybshev polynomial multiplied by spherical harmonics. The Cheybshev polynomial approximate radial component, and the spherical harmonics the angular component. We see spherical harmonics as the angular solution to the Laplace equation $\nabla^2\psi = 0$, and more importantly, as describing the orbits of the hydrogen atom. Even though it is not a surprise that the spherical harmonics should play a role in a problem with spherical geometry, I still want to emphasize the importance of using spherical harmonics.

There are two main problems with spherical coordinates, namely, that there is no straight forward way of imposing a rectangular grid on a sphere, and that the spherical coordinates have three coordinate singularities, one at the origin and two at the poles. By there is no rectangular grid, I meant rectangular grid that can evenly discretize surface area. The most obvious grid on a sphere would be the longitude and latitude grid, but this gives the obvious problem that grid space shrinks as one approach the poles. This means two things: a) that the physics is not resolved near the equator and over-resolved near the poles, b) worse, that the small grid space at the poles can easily lead to blow up because a smaller grid requires a smaller time step, and this hyper resolution near the poles only give way to spurious instabilities that will propagate through the sphere. One could always artificially dampen the physics at the poles or take away grid points, this may solve the blow up, but it will never help with the uneven resolution comparing the equator to points on higher altitude. The three coordinate singularities have to do with the behaviour of unit vector on the poles, and how the volume element shrink to zero at the origin. The angular unit vectors at the poles are not well defined, as there is no more east/west, and all direction are north/south depending on which pole you are at. Worse, a object flying across the pole will have a discontinuity for their polar velocity. Both problems are solved through employing spherical harmonics expansion. For scalar quantities, the second problem is not a big issue, and a normal spherical harmonics series will provide the desired properties to describe a scalar function. For vector and tensor quantities some math heavy lifting is needed, see [**11**][**12**] for a discussion on using spin-weighted spherical harmonics to perform numerical calculations. It should be noted that Dedalus uses spin-weighted spherical harmonics as opposed to the regular spherical harmonics, so all my simulations are in fact done using spin-weighted spherical harmonics.

CHAPTER 3

Numerical Experiments

We have covered aspects in both fluid dynamics and numerical methods, in this chapter we will finally be performing simulations and try to understand these highly non-linear systems.

1. Non-dimensionalization

We will first non-dimensionalize our equations using the geometry of our problem. Non-dimensionalization enables dynamical similarities to be compared, and prepares variables in a non-dimensional form for the numerical solver. According to our geometry (see figure), we define the length scale to be $D = r_{\text{out}} - r_{\text{in}}$; the time scale to be the viscous time D^2/ν, note that since we will be using $\text{Pr} = \nu/\kappa = 1$, this will also be the thermal time; and the temperature scale ΔT. Recall the set of equations we will be solving: Eq. (1.11), Eq. (1.7), Eq. (1.8), substituting in these relations, here are the equations after non-dimensionalization:

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u}\cdot\nabla)\mathbf{u} = -\nabla P + \nabla^2\mathbf{u} + \frac{\text{Ra}}{\text{Ek}}T\hat{\mathbf{r}} - \frac{2}{\text{Ek}}\hat{\mathbf{z}}\times\mathbf{u} \tag{3.1}$$

$$\nabla\cdot\mathbf{u} = 0 \tag{3.2}$$

$$\frac{\partial T}{\partial t} + (\mathbf{u}\cdot\nabla)T = \frac{1}{\text{Pr}}\nabla^2 T \tag{3.3}$$

where $\text{Ra} = g\alpha\Delta T D/\kappa\Omega$ is the modified Rayleigh number, $\text{Ek} = \nu/\Omega D^2$ is the Ekman number, $\text{Pr} = \nu/\kappa$ is the Prandtl number. For our simulations, the relevant parameters are

$$D = 1, \quad \frac{r_{\text{in}}}{r_{\text{out}}} = 0.35, \quad \text{Pr} = 1, \quad \Delta T = 1, \quad \text{Ek and Ra varies,}$$

we employ the no-slip boundary condition for the velocity field (fluids do not move near the boundary), and a non-symmetrical initial temperature distribution. A schematic is given as figure 3.1. Roughly speaking, Rayleigh number controls turbulence, the higher the Rayleigh number, the more turbulent the fluid becomes, and Ekman number controls angular velocity, the smaller the number the faster the rotation. Though it is interesting to note that in certain parameter regime rotation could suppress convection and vice versa.

In following Teed 2018, we aim at a parameter space of $\text{Ek} \sim 10^{-6}$ and a modified $\text{Ra} \sim 10^2$ (non-modified Rayleigh number $\text{Ra} \sim 10^6$). According to [**13**], this roughly corresponds to a transition from rotating convection to non-rotating convection.

2. Running the Code

As discussed in chapter 2, for this problem we use the spherical Dedalus with spherical coordinates (r, θ, ϕ) where θ is the polar angle and ϕ the azimuthal angle. Dedalus decomposes the angular part as spin-weighted spherical harmonics, and the radial part as Cheybshev polynomials. The equation is then stepped in time using the fourth order semi-implicit backward differentiation formula (SBDF). Due to the chaotic and nonlinear nature of the simulation for

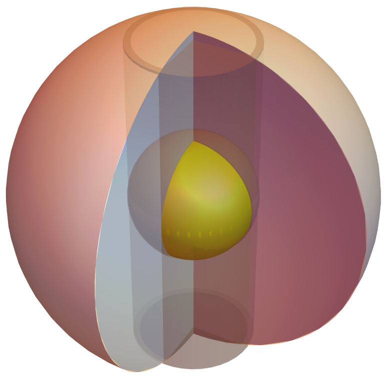

FIGURE 3.1. The setup of the simulation, the inner shell have radius $r_{\text{in}} = 7/13$ and is kept at $T = 1$ and the outer shell have radius $r_{\text{out}} = 20/13$ and is kept at $T = 0$, the cylinder along which the sphere rotates is called the tangent cylinder.

our desired parameter space, through trial and error, I found that it requires a minimum of 192 l modes from the spherical harmonics and 64 modes from the Cheybshev polynomial to resolve the physics, depending on the actual Ra and Ek parameter. Another way of putting this is that it requires $\sim 5 \times 10^6$ grid points to resolve the physical space at each time step. Time step size, in turn, is mostly limited by the Ekman number, and it is also often the bottleneck for how long it takes to run the simulation. The simulation is done on Leavitt at Bates, typically using 128 cores; the speed of the simulation is so far not limited by parallelization. The script can be found both on github and in appendix A. Here are some beautiful patterns that the fluids made (and a fail when the physical processes are not resolved and lead to blow up)!

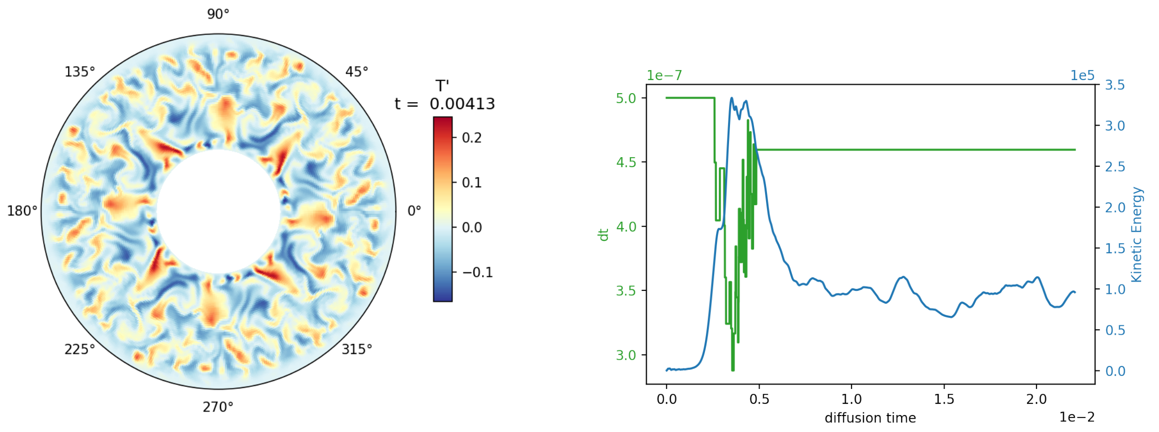

FIGURE 3.2. A simulation set that was resolved. Left shows temperature fluctuation, right shows the CFL timestep (adaptive timestep), and kinetic energy. The peak in kinetic energy corresponds to faster flow and smaller features, thus the timestep is lowered.

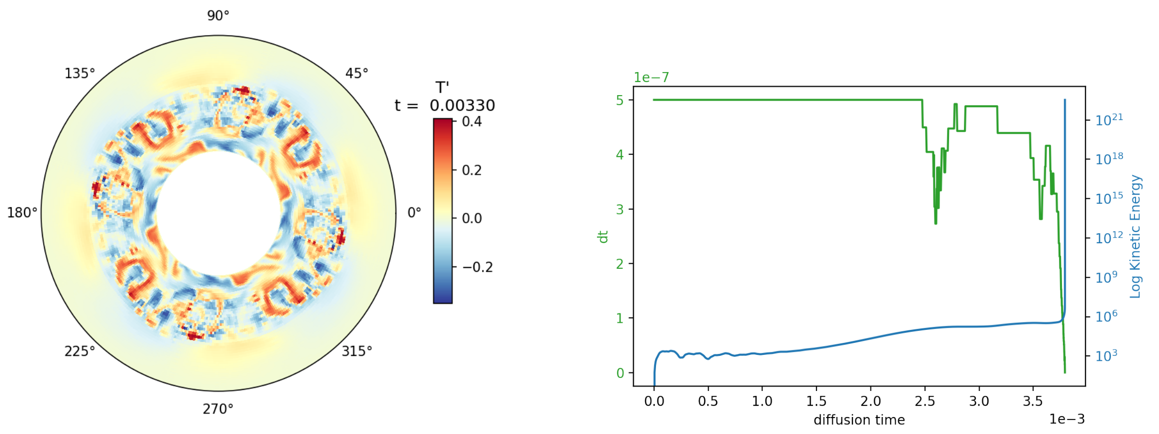

FIGURE 3.3. A simulation set that was not resolved. Plot quantities are the same as the figure above. We can see temperature chunks for the left figure, which is a bad sign, as it typically means that physics is not getting resolved. Right plot shows the blow up. The energy diverged and the timestep fell to zero trying to resolve the physics (but fails).

3. Data Analysis

The spherical Dedalus outputs data in (r, θ, ϕ). For the analysis we sometimes need the cylindrical coordinates (s, ϕ, z), a simple interpolation is made and employed to transform the data. This interpolation script and all other analysis script can be found in appendix A.

3.1. Torsional Waves. Torsional waves is an oscillation in the azimuthal fluctuating velocity that propagates radially from the tangent cylinder to the outer shell, the Earth dynamo action is thought to be the main driver for this oscillation, and for us, since we are not running a magnetohydrodynamics (MHD) simulation, we are interested in wave generation through other mechanism, e.g. Reynolds force. We decompose the velocity into a mean component and a fluctuating component, calculated as follows, following [**3**]:

$$\mathbf{u} = \tilde{\mathbf{u}} + \mathbf{u}', \quad \tilde{\mathbf{u}} = \mathbf{u} - \frac{1}{t}\int_t \mathbf{u}\,\mathrm{d}t\,, \quad \mathbf{u}' = \mathbf{u} - \tilde{\mathbf{u}} \tag{3.4}$$

In the end we take the azimuthal component since that is what we are interested in, and in order to plot it we average in the ϕ and z direction. We mentioned averaging in chapter 1 Eq. (1.13) and here we are plotting $\langle\overline{\mathbf{u}}'\rangle_\phi$. The overbar denotes averaging over ϕ and angle bracket averaging over z. The equation is reproduced here, where R is the radius of the outer shell, $R = r_{\text{out}} = 20/13$.

$$\langle\overline{\mathbf{u}}'\rangle(t, r) = \frac{1}{4\pi\sqrt{R^2 - r^2}}\int_{-\sqrt{R^2-r^2}}^{\sqrt{R^2-r^2}}\int_0^{2\pi} \mathbf{u}\,\mathrm{d}\phi\,\mathrm{d}z\,. \tag{3.5}$$

Below are four plots of $\langle\overline{\mathbf{u}}'\rangle_\phi$ for the same Ekman number and different Rayleigh number, aimed at finding torsional waves.

Looking at these plots at different Rayleigh numbers, we are seeing waves in the azimuthal fluctuating velocity, but we are not seeing torsional waves, as these waves are not propagating in

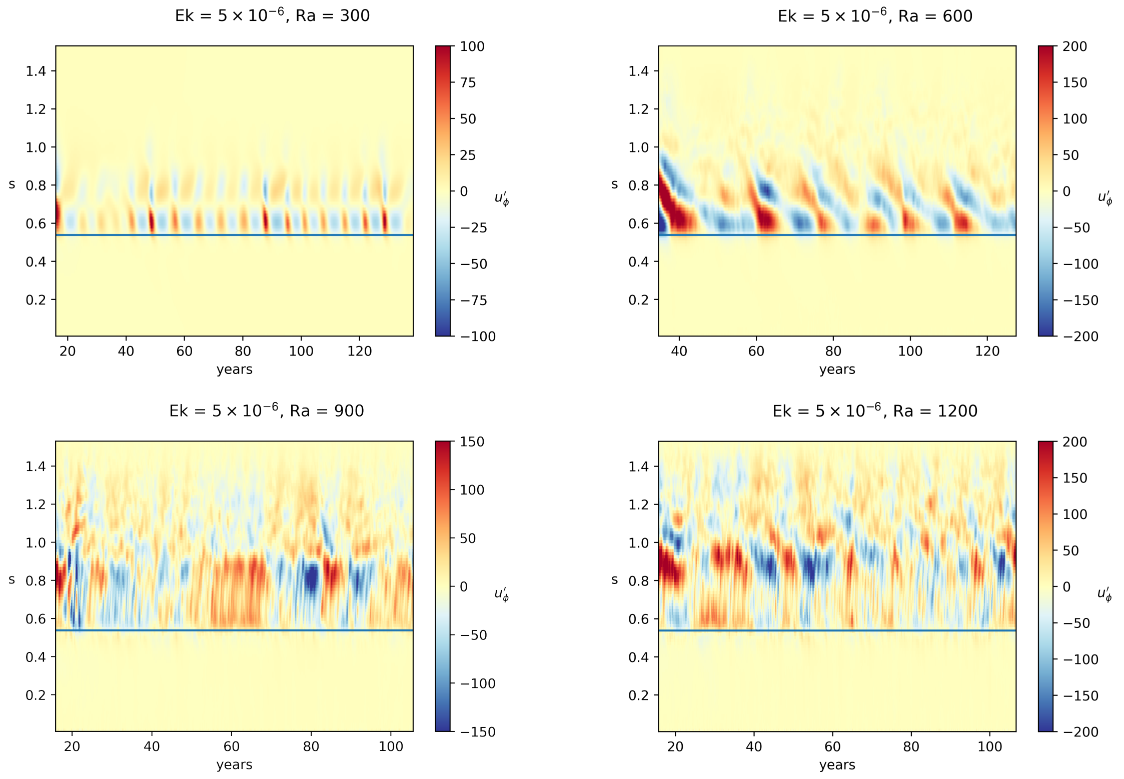

FIGURE 3.4. Azimuthal fluctuating velocity $\langle \overline{\mathbf{u}'} \rangle_\phi$ averaged over ϕ and z. Torsional wave would show up as a linear contour (a or multiple red or blue color line/lines) with positive slope. Instead, for Ra=600 we see negative slope, and others infitie slope (vertical).

the correct direction. Take the most obvious example of Ra=600, we are seeing wave propagation in the reverse direction, that is, from the outer shell to the tangent cylinder. For other cases, we are seeing oscillation in the velocity field, but no propagation (the color stripes are vertical).

3.2. Reynolds Stress. As discussed in chapter 1, Reynolds stress can be interpreted as the force that the fluctuating flow exerts on the mean flow. For our problem, the rotation gives rise to a mean flow in the ϕ direction, it then makes sense, in order to find out why there are waves propagating inwards, to calculate the Reynolds stress. Figure 3.5 are the plots that corresponds to the four simulations that I presented above, it plots the divergence of Reynolds Stress – Reynolds force, in the ϕ direction against the cylindrical radius s, which have the same range as the spherical radius r. Reynolds force is given below, following [**2**]

$$\mathcal{F}_\phi = \hat{\phi} \cdot \langle \boldsymbol{\nabla} \cdot R \rangle \quad \text{where} \quad R = R_{ij} = \overline{u_i' u_j'}. \tag{3.6}$$

From the plot we observes that there are strong driving forces near the tangent cylinder, and that as the fluid crosses the cylinder, the driving direction changes. However, we are not seeing Reynolds force driving waves across different Rayleigh number uniformly. There even seems to be some counter intuitive results for Ra=300, we see a large driving force at around $s = 0.7$, but the fluctating velocity suggest nothing is happening at $s = 0.7$.

3.3. Zonal Flow. Zonal flow are mean flows in the ϕ direction, identified by ϕ velocity. Sometimes, in spherical convection, the ϕ velocity oscillation could be relaxation oscillation.

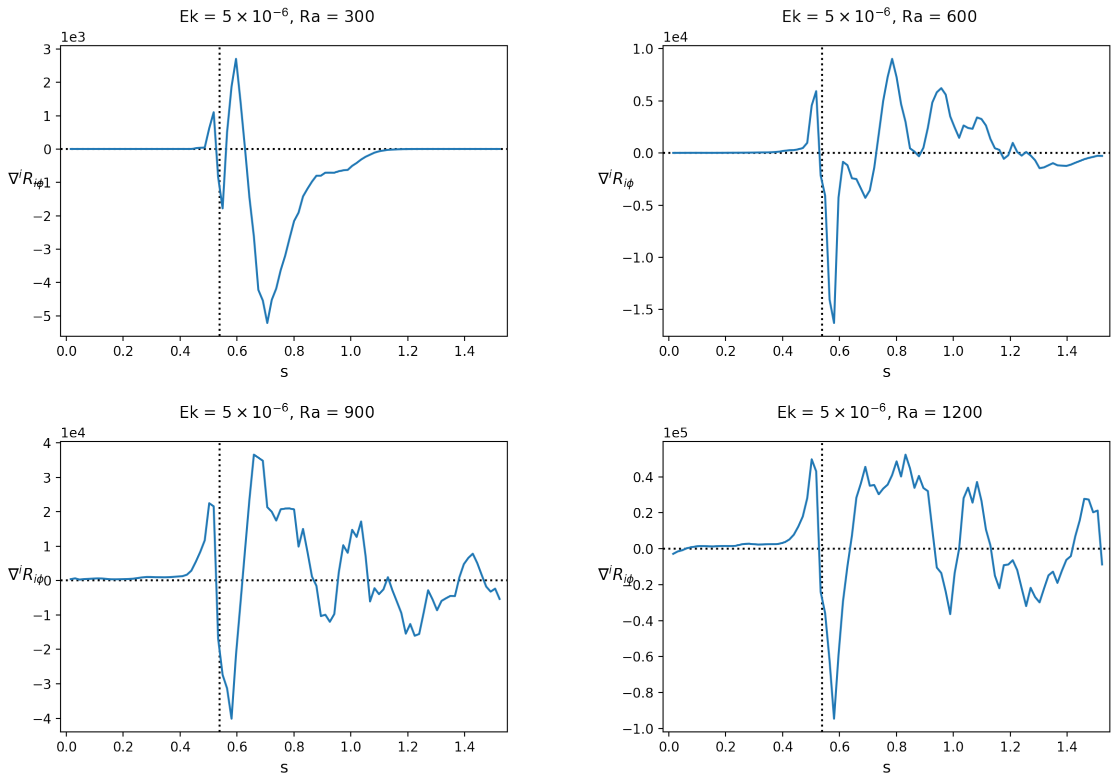

FIGURE 3.5. Reynolds Force, $\langle \boldsymbol{\nabla} \cdot R \rangle_\phi$, also averaged over time.

Roughly speaking, convection drives radial velocity as the temperature gradient is in the r direction. Rotation, i.e. Coriolis force, tilts that radial velocity, and injects energy into the zonal flow. Zonal flow grows through non-linear processes, and dominates the kinetic energy. Zonal flow slows convection, as it is purely a flow in the ϕ direction. Temperature gradient would build up at the boundary, and since there is no more radial velocity, nothing is driving the zonal flow anymore, so the zonal flow dies down and allows convection once again. Of course, this only drives the zonal flow again, hence an oscillation. By plotting the overall kinetic energy against the ϕ component, as show in figure 3.6, we can see that that is not the case, since a) there is no zonal flow build up, and b) the residual velocity is not leading zonal flow.

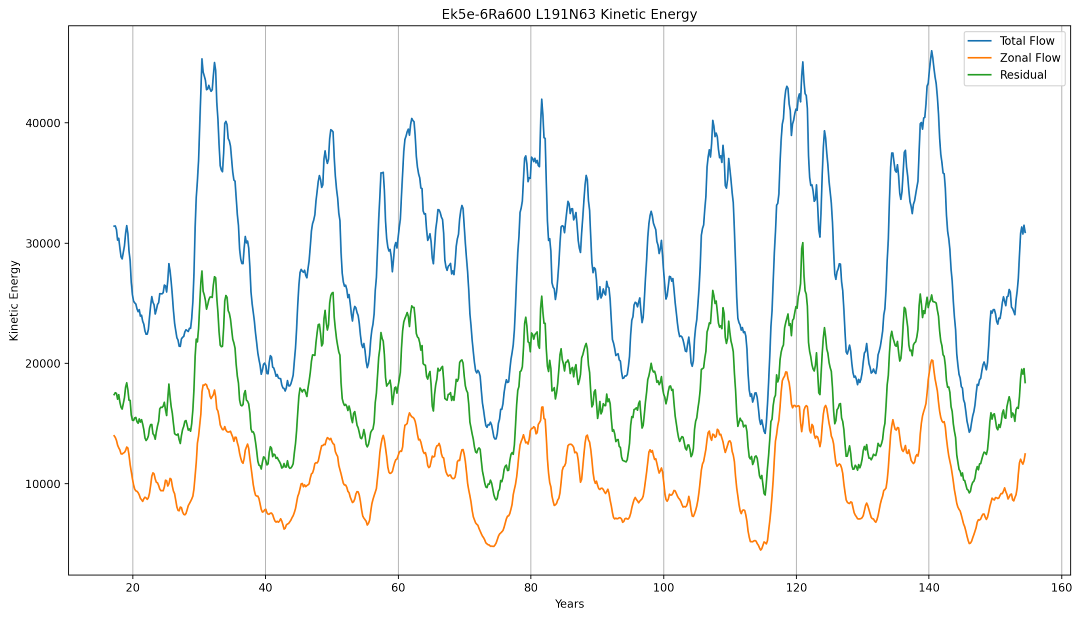

FIGURE 3.6. Zonal Flow. The curves are kinetic energy averaged over volume. Since they are in sync, and no one component dominate the other, this is not a relaxation oscillation. Though again, there seems to be periodic behaviour.

CHAPTER 4

Discussion and Future Work

Most immediately, we can try to understand the oscillatory behaviour of the azimuthal velocity. Reynolds force, as is calculated now, does not seem to provide a full explanation. One of the possibilities being that we need to account for Reynolds force differently for the north and south hemisphere. Also, according to [**14**], our parameter regime might be a transitional regime from convection being dominated by rotation to rotation having a decreasing effect on convection, in which case traditional literature could help shed light on this phenomenon.

From chapter 3, we realize that we did not find torsional oscillation, indeed, torsional oscillation is mostly seen as a magnetohydrodynamics (MHD) effect, where the magnetic field acts as a spring in storing energy and providing the torsion. Some theoretical work has been done [**15**] to show that torsional oscillation is impossible in certain spherical no-slip boundary setups. On the other hand, as mentioned in the introduction, it is shown that for some range of Prantle number and Ekman number, there could be torsional oscillation for stress-free boundary conditions. In which case we could either turn on the magnetic field to investigate torsional oscillation in MHD, or we could reformulate our boundary condition to stress free and try to produce torsional oscillation in a purely hydrodynamics context.

Another possible avenue is to pivot towards investigating convection for rotating shells, as [**14**][**16**] had done. We can explore the parameter space of rotating spherical shell, and focus on the effect that rotation have on convection.

Bibliography

[1] R. Holme and O. de Viron. Characterization and implications of intradecadal variations in length of day. 499(7457):202–204.

[2] Johannes Wicht and Ulrich R. Christensen. Torsional oscillations in dynamo simulations. *Geophysical Journal International*, 181(3):1367–1380, 06 2010.

[3] R J Teed, C A Jones, and S M Tobias. Torsional waves driven by convection and jets in Earth's liquid core. *Geophysical Journal International*, 216(1):123–129, 10 2018.

[4] Juan Sánchez, Ferran Garcia, and Marta Net. Critical torsional modes of convection in rotating fluid spheres at high taylor numbers. *Journal of Fluid Mechanics*, 791:R1, 2016.

[5] Dixon Rohr. Cut-away diagram of earth's interior, 2017. Image from NASA website.

[6] D. J. Tritton. *Physical Fluid Dynamics*. Oxford University Press, NY, 1988.

[7] Flames on earth and space (1998), 2014. Image from NASA website.

[8] Frans T.M. Nieuwstadt, Bendiks J. Boersma, and Jerry Westerweel. *Turbulence*. Springer International Publishing, 2016.

[9] John P. Boyd. *Chebyshev and Fourier Spectram Methods*. Dover, NY, 2 edition, 2001.

[10] Keaton J. Burns, Geoffrey M. Vasil, Jeffrey S. Oishi, Daniel Lecoanet, and Benjamin P. Brown. Dedalus: A flexible framework for numerical simulations with spectral methods. *Phys. Rev. Research*, 2:023068, Apr 2020.

[11] Geoffrey M. Vasil, Daniel Lecoanet, Keaton J. Burns, Jeffrey S. Oishi, and Benjamin P. Brown. Tensor calculus in spherical coordinates using jacobi polynomials. part-i: Mathematical analysis and derivations. *Journal of Computational Physics: X*, 3:100013, 2019.

[12] Daniel Lecoanet, Geoffrey M. Vasil, Keaton J. Burns, Benjamin P. Brown, and Jeffrey S. Oishi. Tensor calculus in spherical coordinates using jacobi polynomials. part-ii: Implementation and examples. *Journal of Computational Physics: X*, 3:100012, 2019.

[13] Thomas Gastine, Johannes Wicht, and Julien Aubert. Scaling regimes in spherical shell rotating convection. *Journal of Fluid Mechanics*, 808:690–732, 2016.

[14] E. DORMY, A. M. SOWARD, C. A. JONES, D. JAULT, and P. CARDIN. The onset of thermal convection in rotating spherical shells. *Journal of Fluid Mechanics*, 501:43–70, 2004.

[15] Keke Zhang, Kameng Lam, and Dali Kong. Asymptotic theory for torsional convection in rotating fluid spheres. *Journal of Fluid Mechanics*, 813:R2, 2017.

[16] Eric M. King, Stephan Stellmach, Jerome Noir, Ulrich Hansen, and Jonathan M. Aurnou. Boundary layer control of rotating convection systems. 457(7227):301–304.

Appendix A: Code

1. Run Script

```
import numpy as np
import scipy.sparse      as sparse
import dedalus.public as de
from dedalus.core import arithmetic, timesteppers, problems, solvers
from dedalus.tools.parsing import split_equation
from dedalus.extras.flow_tools import GlobalArrayReducer
import dedalus_sphere
from mpi4py import MPI
import time
#from dedalus_sphere import ball, intertwiner
#me trying to use config files
#1-0 is for a day
import os
import sys
import configparser
from configparser import ConfigParser
from pathlib import Path

import matplotlib
import matplotlib.pyplot as plt
import logging
logger = logging.getLogger(__name__)

matplotlib_logger = logging.getLogger('matplotlib')
matplotlib_logger.setLevel(logging.WARNING)

comm = MPI.COMM_WORLD
rank = comm.rank
dtype = np.float64
size = comm.size

config_file = Path(sys.argv[-1])
config = ConfigParser()
config.read(str(config_file))

logger.info('Running with the following parameters:')
logger.info(config.items('parameters'))

params = config['parameters']

# create data dir using basename of cfg file
basedir = Path('frames')
outdir = "frames_" + config_file.stem
data_dir = basedir/outdir
logger.info(data_dir)
if rank == 0:
    if not data_dir.exists():
        data_dir.mkdir(parents=True)


Lmax = params.getint('Lmax')
Nmax = params.getint('Nmax')

# right now can't run with dealiasing
L_dealias = 3/2
N_dealias = 3/2

# parameters
Ekman = params.getfloat('Ekman')
Prandtl = 1
Rayleigh = params.getint('Rayleigh')
r_inner = 7/13
r_outer = 20/13
radii = (r_inner,r_outer)

# mesh must be 2D for plotting
mesh = [params.getint('Xn'),params.getint('Yn')]

c = de.coords.SphericalCoordinates('phi', 'theta', 'r')
d = de.distributor.Distributor((c,), mesh=mesh)
b    = de.basis.SphericalShellBasis(c, (2*(Lmax+1),Lmax+1,Nmax+1), radii=radii, dealias=(L_dealias,L_dealias,N_dealia

b_inner = b.S2_basis(radius=r_inner)
b_outer = b.S2_basis(radius=r_outer)
phi, theta, r = b.local_grids((L_dealias,L_dealias,N_dealias))
phig,thetag,rg= b.global_grids((L_dealias,L_dealias,N_dealias))
theta_target = thetag[0,(Lmax+1)//2,0]

weight_theta = b.local_colatitude_weights(L_dealias)
weight_r = b.local_radial_weights(N_dealias)*r**2

u = de.field.Field(dist=d, bases=(b,), tensorsig=(c,), dtype=dtype)
u.set_scales(b.dealias)
p = de.field.Field(dist=d, bases=(b,), dtype=dtype)
p.set_scales(b.dealias)
T = de.field.Field(dist=d, bases=(b,), dtype=dtype)
T.set_scales(b.dealias)
tau_u_inner = de.field.Field(dist=d, bases=(b_inner,), tensorsig=(c,), dtype=dtype)
tau_T_inner = de.field.Field(dist=d, bases=(b_inner,), dtype=dtype)
tau_u_outer = de.field.Field(dist=d, bases=(b_outer,), tensorsig=(c,), dtype=dtype)
tau_T_outer = de.field.Field(dist=d, bases=(b_outer,), dtype=dtype)

ez = de.field.Field(dist=d, bases=(b,), tensorsig=(c,), dtype=dtype)
ez.set_scales(b.dealias)
ez['g'][1] = -np.sin(theta)
ez['g'][2] =  np.cos(theta)

r_vec = de.field.Field(dist=d, bases=(b,), tensorsig=(c,), dtype=dtype)
r_vec.set_scales(b.dealias)
r_vec['g'][2] = r/r_outer

T_inner = de.field.Field(dist=d, bases=(b_inner,), dtype=dtype)
T_inner['g'] = 1.

# initial condition
A = 0.1
```

```
x = 2*r-r_inner-r_outer
T['g'] = r_inner*r_outer/r - r_inner + 210*A/np.sqrt(17920*np.pi)*(1-3*x**2+3*x**4-x**6)*np.sin(theta)**4*np.cos(2*np

# Parameters and operators
div = lambda A: de.operators.Divergence(A, index=0)
lap = lambda A: de.operators.Laplacian(A, c)
grad = lambda A: de.operators.Gradient(A, c)
dot = lambda A, B: arithmetic.DotProduct(A, B)
curl = lambda A: de.operators.Curl(A)
cross = lambda A, B: arithmetic.CrossProduct(A, B)
ddt = lambda A: de.operators.TimeDerivative(A)

# Make grid-locked parameters to avoid unnecessary transforms
grid_r_vec = de.operators.Grid(r_vec).evaluate()
grid_ez = de.operators.Grid(ez).evaluate()

# Problem
def eq_eval(eq_str):
    return [eval(expr) for expr in split_equation(eq_str)]
problem = problems.IVP([u, p, T, tau_u_inner, tau_T_inner, tau_u_outer, tau_T_outer])
problem.add_equation(eq_eval("ddt(u) - lap(u) + grad(p) = cross(curl(u), u) + Rayleigh/Ekman*grid_r_vec*T - 2/Ekman*c
problem.add_equation(eq_eval("u = 0"), condition = "ntheta == 0")
problem.add_equation(eq_eval("div(u) = 0"), condition = "ntheta != 0")
problem.add_equation(eq_eval("p = 0"), condition = "ntheta == 0")
problem.add_equation(eq_eval("ddt(T) - lap(T)/Prandtl = - dot(u,grad(T))"))
problem.add_equation(eq_eval("u(r=7/13) = 0"), condition = "ntheta != 0")
problem.add_equation(eq_eval("tau_u_inner = 0"), condition = "ntheta == 0")
problem.add_equation(eq_eval("T(r=7/13) = T_inner"))
problem.add_equation(eq_eval("u(r=20/13) = 0"), condition = "ntheta != 0")
problem.add_equation(eq_eval("tau_u_outer = 0"), condition = "ntheta == 0")
problem.add_equation(eq_eval("T(r=20/13) = 0"))
logger.info("Problem built")


# Solver


timestepper=params.get('timestepper')
safety = params.getfloat('safety') # 0.4 should work for SBDF2
if timestepper == 'SBDF2':
        ts=timesteppers.SBDF2
        timestepper_history = [0,1]
elif timestepper == 'SBDF4':
        ts=timesteppers.SBDF4
        timestepper_history = [0,1,2,3]
else:
    ts=timesteppers.SBDF2
    timestepper_history = [0,1]
    safety = 0.4

logger.info("timestepper: {} with safety {}".format(params.get('timestepper'),params.getfloat('safety')))

solver = solvers.InitialValueSolver(problem, ts)

hermitian_cadence=100

# Add taus

# ChebyshevV
```

```
alpha_BC = (2-1/2, 2-1/2)

def C(N):
    ab = alpha_BC
    cd = (b.radial_basis.alpha[0]+2,b.radial_basis.alpha[1]+2)
    return dedalus_sphere.jacobi.coefficient_connection(N+1,ab,cd)

def BC_rows(N, num_comp):
    N_list = (np.arange(num_comp)+1)*(N + 1)
    return N_list

for subproblem in solver.subproblems:
    ell = subproblem.group[1]
    L = subproblem.left_perm.T @ subproblem.L_min
    shape = L.shape
    if dtype == np.complex128:
        if ell != 0:
            N0, N1, N2, N3, N4 = BC_rows(Nmax, 5)
            tau_columns = np.zeros((shape[0], 8))
            tau_columns[  :N0,0] = (C(Nmax))[:,-1]
            tau_columns[N0:N1,1] = (C(Nmax))[:,-1]
            tau_columns[N1:N2,2] = (C(Nmax))[:,-1]
            tau_columns[N3:N4,3] = (C(Nmax))[:,-1]
            tau_columns[  :N0,4] = (C(Nmax))[:,-2]
            tau_columns[N0:N1,5] = (C(Nmax))[:,-2]
            tau_columns[N1:N2,6] = (C(Nmax))[:,-2]
            tau_columns[N3:N4,7] = (C(Nmax))[:,-2]
            L[:,-8:] = tau_columns
        else: # ell = 0
            N0, N1, N2, N3, N4 = BC_rows(Nmax, 5)
            L[N3:N4,N4+3] = (C(Nmax))[:,-1].reshape((N0,1))
            L[N3:N4,N4+7] = (C(Nmax))[:,-2].reshape((N0,1))
    elif dtype == np.float64:
        N0, N1, N2, N3, N4 = BC_rows(Nmax, 5)*2
        if ell != 0:
            tau_columns = np.zeros((shape[0], 16))
            tau_columns[ 0:Nmax+1,0] = (C(Nmax))[:,-1]
            tau_columns[N0:N0+Nmax+1,2] = (C(Nmax))[:,-1]
            tau_columns[N1:N1+Nmax+1,4] = (C(Nmax))[:,-1]
            tau_columns[N3:N3+Nmax+1,6] = (C(Nmax))[:,-1]
            tau_columns[ 0:Nmax+1,8] = (C(Nmax))[:,-2]
            tau_columns[N0:N0+Nmax+1,10] = (C(Nmax))[:,-2]
            tau_columns[N1:N1+Nmax+1,12] = (C(Nmax))[:,-2]
            tau_columns[N3:N3+Nmax+1,14] = (C(Nmax))[:,-2]
            tau_columns[Nmax+1:2*(Nmax+1),1] = (C(Nmax))[:,-1]
            tau_columns[N0+Nmax+1:N0+2*(Nmax+1),3] = (C(Nmax))[:,-1]
            tau_columns[N1+Nmax+1:N1+2*(Nmax+1),5] = (C(Nmax))[:,-1]
            tau_columns[N3+Nmax+1:N3+2*(Nmax+1),7] = (C(Nmax))[:,-1]
            tau_columns[Nmax+1:2*(Nmax+1),9] = (C(Nmax))[:,-2]
            tau_columns[N0+Nmax+1:N0+2*(Nmax+1),11] = (C(Nmax))[:,-2]
            tau_columns[N1+Nmax+1:N1+2*(Nmax+1),13] = (C(Nmax))[:,-2]
            tau_columns[N3+Nmax+1:N3+2*(Nmax+1),15] = (C(Nmax))[:,-2]
            L[:,-16:] = tau_columns
        else: # ell = 0
            L[N3:N3+Nmax+1,N4+6] = (C(Nmax))[:,-1].reshape((N0//2,1))
            L[N3:N3+Nmax+1,N4+14] = (C(Nmax))[:,-2].reshape((N0//2,1))
            L[N3+Nmax+1:N3+2*(Nmax+1),N4+7] = (C(Nmax))[:,-1].reshape((N0//2,1))
            L[N3+Nmax+1:N3+2*(Nmax+1),N4+15] = (C(Nmax))[:,-2].reshape((N0//2,1))
    L.eliminate_zeros()
    subproblem.L_min = subproblem.left_perm @ L
    subproblem.expand_matrices(['M','L'])

reducer = GlobalArrayReducer(d.comm_cart)

vol_test = np.sum(weight_r*weight_theta+0*p['g'])*np.pi/(Lmax+1)/L_dealias
vol_test = reducer.reduce_scalar(vol_test, MPI.SUM)
vol = 4*np.pi/3*(r_outer**3-r_inner**3)
vol_correction = vol/vol_test

t = 0.

t_list = []
E_list = []

max_dt = params.getfloat('max_dt')
init_dt = params.getfloat('init_dt')
dt=init_dt

report_cadence = 10

plot_cadence = max_dt*250 #original is 100, 500
plot_num=0
dpi = 150

plot = theta_target in theta

include_data = comm.gather(plot)

var = T['g']
name = 'T'
#used to be true, just trying it out
remove_m0 = True

if plot:
    i_theta = np.argmin(np.abs(theta[0,:,0] - theta_target))
    plot_data = var[:,i_theta,:].real.copy()
    plot_rec_buf = None
else:
    plot_data = np.zeros_like(var[:,0,:].real)

plot_rec_buf = None
if rank == 0:
    rec_shape = [size,] + list(var[:,0,:].shape)
    plot_rec_buf = np.empty(rec_shape,dtype=plot_data.dtype)
comm.Gather(plot_data, plot_rec_buf, root=0)

def equator_plot(r, phi, data, index=None, pcm=None, cmap=None, title=None):
    if pcm is None:
        r_pad   = np.pad(r[0,0,:], ((0,1)), mode='constant', constant_values=(r_inner,r_outer))
        phi_pad = np.append(phi[:,0,0], 2*np.pi)
        fig, ax = plt.subplots(subplot_kw=dict(polar=True))
        r_plot, phi_plot = np.meshgrid(r_pad,phi_pad)
        pcm = ax.pcolormesh(phi_plot,r_plot,data, cmap=cmap)
        ax.set_rlim(bottom=0, top=r_outer)
        ax.set_rticks([])
        ax.set_aspect(1)
        pmin,pmax = pcm.get_clim()
        cNorm = matplotlib.colors.Normalize(vmin=pmin, vmax=pmax)
        ax_cb = fig.add_axes([0.8, 0.3, 0.03, 1-0.3*2])
        cb = fig.colorbar(pcm, cax=ax_cb, norm=cNorm, cmap=cmap)
        fig.subplots_adjust(left=0.05,right=0.85)
        if title is not None:
            ax_cb.set_title(title)
        pcm.ax_cb = ax_cb
        pcm.cb_cmap = cmap
        pcm.cb = cb
        return fig, pcm
    else:
        pcm.set_array(np.ravel(data))
        pcm.set_clim([np.min(data),np.max(data)])
        cNorm = matplotlib.colors.Normalize(vmin=np.min(data), vmax=np.max(data))
        pcm.cb.mappable.set_norm(cNorm)
        if title is not None:
            pcm.ax_cb.set_title(title)

if rank == 0:
    data = []
    for pd, id in zip(plot_rec_buf, include_data):
        if id: data.append(pd)
    data = np.array(data)
    data = np.transpose(data, axes=(1,0,2)).reshape((int(2*(Lmax+1)*L_dealias),int((Nmax+1)*N_dealias)))
    if remove_m0:
        data -= np.mean(data, axis=0)
    fig, pcm = equator_plot(rg, phig, data, title=name+"'\n t = {:8.5f}".format(0), cmap = 'RdYlBu_r')
    plt.savefig( str(data_dir)+'/%s_%04i.png' %(name, plot_num), dpi=dpi)

# timestepping loop
start_time = time.time()

#variable time step
threshold = 0.1
dr = np.gradient(r[0,0])


def calculate_dt(dt_old):
    local_freq  = np.abs(u['g'][2]/dr) + np.abs(u['g'][0]*(Lmax+1)) + np.abs(u['g'][1]*(Lmax+1))
    global_freq = reducer.global_max(local_freq)

    if global_freq == 0.:
        dt = np.inf
    else:
        dt = 1 / global_freq
        dt *= safety

    if dt > max_dt:
        dt = max_dt

    if solver.sim_time < 0.002 and dt > init_dt:
        dt = init_dt

    if dt < dt_old*(1+threshold) and dt > dt_old*(1-threshold):
        dt = dt_old
    return dt
```

```
checkpoint = solver.evaluator.add_file_handler("data_" + config_file.stem,iter=500,max_writes=2)
checkpoint.add_task(T, name='T')
checkpoint.add_task(u, name='u')

#coeffcheckpoint = solver.evaluator.add_file_handler('coeffcheckpoint',iter=1500,max_writes=5)
#coeffcheckpoint.add_task(T, name='T', layout='c')

# Integration parameters


t_end = params.getfloat('t_end') #10 #1.25
solver.stop_sim_time = t_end
#solver.stop_iteration=100

logged = False

while solver.ok:

    u.require_scales(L_dealias) #my fix, probably wrong
    dt=calculate_dt(dt)

    if solver.iteration % report_cadence == 0:
        logged = True
#         logger.info("u['g'].shape = {}".format(u['g'].shape))
#         logger.info("weight_r = {}".format(weight_r.shape))
#         logger.info("weight_theta = {}".format(weight_theta.shape))
        E0 = np.sum(vol_correction*weight_r*weight_theta*u['g']**2)
        E0 = 0.5*E0*(np.pi)/(Lmax+1)/L_dealias/vol
        E0 = reducer.reduce_scalar(E0, MPI.SUM)
        #T.require_scales(L_dealias)
        T0 = np.sum(vol_correction*weight_r*weight_theta*T['g']**2)
        T0 = 0.5*T0*(np.pi)/(Lmax+1)/L_dealias/vol
        T0 = reducer.reduce_scalar(T0, MPI.SUM)
        logger.info("iter: {:d}, dt={:e}, t={:e}, E0={:e}, T0={:e}".format(solver.iteration, dt, solver.sim_time, E0,
        t_list.append(solver.sim_time)
        E_list.append(E0)

    if solver.sim_time // plot_cadence > plot_num:

        plot_num += 1

        if logged == False:
                #logging information again, makse the plot possible
            E0 = np.sum(vol_correction*weight_r*weight_theta*u['g']**2)
            E0 = 0.5*E0*(np.pi)/(Lmax+1)/L_dealias/vol
            E0 = reducer.reduce_scalar(E0, MPI.SUM)
            T.require_scales(L_dealias)
            T0 = np.sum(vol_correction*weight_r*weight_theta*T['g']**2)
            T0 = 0.5*T0*(np.pi)/(Lmax+1)/L_dealias/vol
            T0 = reducer.reduce_scalar(T0, MPI.SUM)
            logger.info("iter: {:d}, dt={:e}, t={:e}, E0={:e}, T0={:e}".format(solver.iteration, dt, solver.sim_time,
            t_list.append(solver.sim_time)
            E_list.append(E0)

        if plot:
            plot_data = var[:,i_theta,:].real.copy()

        comm.Gather(plot_data, plot_rec_buf, root=0)
        if rank == 0:
            data = []
            for pd, id in zip(plot_rec_buf, include_data):
                if id: data.append(pd)
            data = np.array(data)
            data = np.transpose(data, axes=(1,0,2)).reshape((int(2*(Lmax+1)*L_dealias),int((Nmax+1)*N_dealias)))
            if remove_m0:
                data -= np.mean(data, axis=0)
            equator_plot(rg, phig, data, title=name+"'\n t = {:8.5f}".format(solver.sim_time), cmap='RdYlBu_r', pcm=p
            fig.savefig(str(data_dir)+'/%s_%04i.png' %(name,plot_num), dpi=dpi)

    # enforce hermitian symmetry (data should be real)
    if solver.iteration % hermitian_cadence in timestepper_history:
        for field in solver.state:
            field.require_grid_space()

    logged = False
#    logger.info("dt={:e}".format(dt))
    solver.step(dt)

end_time = time.time()
if rank==0:
    print('simulation took: %f' %(end_time-start_time))
    t_list = np.array(t_list)
    E_list = np.array(E_list)
    np.savetxt(str(config_file.stem)+'_marti_conv.dat',np.array([t_list,E_list]))
```

2. Analysis Script

2.1. Crude plot of kinetic energy and timestep. e.g. Right plot of Fig.3.2. It is originally an ipython notebook.

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import os

root=os.path.join('Z:\\','torsional_wave_topoology','python')
slurmouts=['slurm-6904.out','slurm-6913.out']
files=[]
legends=['Nmax63, t=1', 'Nmax47, t=2']

#read files
for i in range(len(slurmouts)):
    slurmouts[i]=os.path.join(root, slurmouts[i])
    files.append(open(slurmouts[i]))

#read values
E0=[]
t=[]
dt=[]
counter=0

for file in files:
    E0.append([])
    dt.append([])
    t.append([])
    for lines in file:
        if 'iter' in lines:
            i = lines.find("dt=")
            j = lines.find(" t=")
            k = lines.find("E0=")
            dt[counter].append(float(lines[3+i:15+i]))
            t[counter].append(float(lines[3+j:15+j]))
            E0[counter].append(float(lines[3+k:15+k]))
    counter+=1


plt.figure(figsize=[16,9])
for i in range(counter):
    plt.plot(t[i],E0[i],label=legends[i])
plt.xlabel("diffusion time")
plt.ylabel("E0")
plt.title("Ek5e-6Ra900, Lmax191, N63vs47")
plt.legend()
plt.savefig('E0 plot',dpi=400)
```

2.2. Cylindrical coordinate interpolation. This code returns data in a cylindrical grid interpolated from its orignal spherical grid. This can be used to both plot the fluctuating velocity, figure 3.4, and the Reynolds force, figure 3.5. It is originally an ipython notebook.

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import h5py
import os
import re
import scipy.spatial.qhull as qhull
import itertools
import time


# In[2]:


def get_file_name(mydir, head, end):
    files = []
    for file in os.listdir(mydir):
        if file.endswith(".h5"):
            files.append(os.path.join(mydir, file))
    files.sort(key=lambda f: int(re.sub('\D','', f)))

    files = files[head:end]

    return files


# In[3]:


def get_scales(file_names):
    times = []
    for i,file_name in enumerate(file_names):
        with h5py.File(file_name,"r") as df:
            dataset = df['tasks/u']
            times.append(dataset.dims[0][0][:])
            if i == 0:
                rscale = dataset.dims[3][0][:]
                thetascale = dataset.dims[2][0][:]
                phiscale = dataset.dims[1][0][:]
    times = np.array(times).ravel()
    return times, phiscale, thetascale, rscale


# In[4]:


def get_grids(r_inner, r_outer, rscale):
    start_point = 0
    end_point = r_outer
    #gridpoints = int(rscale.shape[0]/(r_outer-r_inner) * (end_point-start_point))
    gridpoints = int(64.0/(r_outer-r_inner) * (end_point-start_point))

    #finding equidistant point with the same density as the original grid along the path of interpolation
    z_desired = np.asarray(np.linspace( start_point, end_point, num = gridpoints, endpoint = False))

    s_points = z_desired[1::]
    z_points = np.sort(np.concatenate((-s_points, z_desired)))
    desired_grid = np.asarray([[[s,z] for z in z_points] for s in s_points]) #correct
    #desired_grid = np.asarray([[[s,z] for s in s_points] for z in z_points])

    #padding and mesh
    phi_pad = np.append(phiscale[:], 2*np.pi)
    s_pad   = np.append(z_desired[:], end_point)
    s_mesh, phi_mesh = np.meshgrid(z_desired, phi_pad)

    return desired_grid, s_points, z_points, s_mesh, phi_mesh


# In[5]:


#https://stackoverflow.com/questions/20915502
#/speedup-scipy-griddata-for-multiple-interpolations-between-two-irregular-grids
def interp_weights(xyz, uvw, d=2):
    #xyz is the spherical grid
    #uvw is the desired interpolation grid
    tri = qhull.Delaunay(xyz)
    simplex = tri.find_simplex(uvw)
    vertices = np.take(tri.simplices, simplex, axis=0)
    temp = np.take(tri.transform, simplex, axis=0)
    delta = uvw - temp[:, d]
    bary = np.einsum('njk,nk->nj', temp[:, :d, :], delta)
    return vertices, np.hstack((bary, 1 - bary.sum(axis=1, keepdims=True)))

def interpolate(values, vtx, wts, fill_value=np.nan):
    ret = np.einsum('nj,nj->n', np.take(values, vtx), wts)
    ret[np.any(wts < 0, axis=1)] = fill_value
    return ret


# In[6]:
```

```
def inter_prep(desired_grid, iscale, thetascale, rscale, r_inner, r_outer):

    grid_shape = (-1,) + desired_grid.shape[0:-1] #the last dim is 2 because 2d
    nanindex = np.where(np.linalg.norm(desired_grid, axis = 2) < rscale.min())
    desired_grid = desired_grid.reshape(-1,2)
    xygrid = np.asarray([[r*np.sin(theta), r*np.cos(theta)] for theta in thetascale for r in rscale])

    vtx, wts = interp_weights(xygrid, desired_grid)

    ones = np.full_like(thetascale,1)
    sph_to_cyl = np.array([[np.sin(thetascale),0*ones,-1*np.cos(thetascale)], [[0],[1],[0]]*ones,[np.cos(thetascale),

    return sph_to_cyl, vtx, wts, grid_shape, nanindex


# In[7]:


def inter_routine(file_name, sph_to_cyl, vtx, wts, grid_shape, nanindex):
    cylindrical = []
    with h5py.File(file_name,"r") as df:
        dataset = df['tasks/u']
        for data in dataset: #loops time in file
            dats = np.einsum('ajk,jckd->ackd',sph_to_cyl,data) #transforms all three vector components
            phi_comp   = np.asarray([interpolate(np.ravel(dat), vtx, wts) for dat in dats[0]]).reshape(grid_shape)
            theta_comp = np.asarray([interpolate(np.ravel(dat), vtx, wts) for dat in dats[1]]).reshape(grid_shape)
            r_comp     = np.asarray([interpolate(np.ravel(dat), vtx, wts) for dat in dats[2]]).reshape(grid_shape)

            ans = np.array([phi_comp,theta_comp,r_comp])
            ans[:,:,nanindex[0],nanindex[1]] = np.nan
            cylindrical.append(ans)
    cylindrical = np.array(cylindrical)
    cylindrical = np.einsum('tupsz->tupzs',cylindrical) #t, u, phi, z, s
    return cylindrical


# In[8]:


def F_R_phi_stress(data,phiscale,zscale, sscale):
    F_R_phi = []
    for u in data:
        u_prime = u - np.nanmean(u, axis = 1, keepdims=True) #u,phi,z,s

        T = np.einsum("i...,j...->ij...",u_prime,u_prime)
        T = np.nanmean(T, axis=(2)) #u, u, phi, z, s

        term1 = np.gradient(T[2,0], sscale, axis = 1)
        term2 = np.gradient(T[1,0], zscale, axis = 0)
        #term2 = np.einsum("ij,j -> ij", T[2,0], 1./sscale)
        #term2 = 1./sscale[np.newaxis,:]*np.gradient(T[0,0], phiscale, axis = 0)
        #term2 = np.gradient(T[1,0], zscale, axis = 0)
        #term3 = np.einsum("ij,j -> ij", dT, 1./rscale)
        #term4 = np.einsum("ij,i,j -> ij", T[1,0], 1./np.tan(thetascale), 1./rscale)
        term3 = 2./sscale[np.newaxis,:]*T[0,1]
        F_phi = term1+ term2 + term3

        F_R_phi.append(F_phi)
    return np.array(F_R_phi)


# In[162]:


mydir ="data_Ek5e-6Ra300_L191N63_init5max10e-7-u"
file_names = get_file_name(mydir, 10, 40)
r_inner = 7/13.
r_outer = 20./13.
timescale, phiscale, thetascale, rscale = get_scales(file_names)
desired_grid, sscale, zscale, _, _ = get_grids(r_inner, r_outer, rscale)
sph_to_cyl, vtx, wts, grid_shape, nanindex = inter_prep(desired_grid, phiscale, thetascale, rscale, r_inner, r_outer


# In[163]:


data_phi = []
RF_phi = []


# In[120]:


all_data = []


# In[164]:


for i,file_name in enumerate(file_names): #loops file
    data = inter_routine(file_name, sph_to_cyl, vtx, wts, grid_shape, nanindex)

    F_R_phi = F_R_phi_stress(data,phiscale,zscale,sscale)

    #data_phi.append(data[:,0])
    RF_phi.append(F_R_phi)

    #all_data.append(data)

data_phi = np.asarray(data_phi)
RF_phi = np.asarray(RF_phi)


# In[19]:


data_phi = data_phi.reshape(-1,384,195,97)


# In[165]:


RF_bar = np.nanmean(np.abs(RF_phi.reshape(-1,195,97)), axis = (0,1))


# In[166]:
```

```
plt.plot(sscale,RF_bar)
plt.axvline(r_inner,linestyle ="dotted",color = 'black')


# In[167]:


fig, ax = plt.subplots()
ax.set_xlabel('s',fontsize=12)
ax.set_ylabel(r'$\nabla^iR_{i\phi}$',rotation =0,fontsize=12 , labelpad = 15)
ax.ticklabel_format(style='sci',scilimits=(0,0))
ax.axvline(r_inner,linestyle =(0, (1, 10)),color = 'black')
ax.axhline(0,linestyle = 'dotted',color = 'black')
ax.plot(sscale, RF_bar)
fig.suptitle(r'Ek = $5\times 10^{-6}$, Ra = 300')
ax.set_xlim([-0.02, 1.55])
#fig.tight_layout()
fig.savefig('R-Ek5e-6Ra300L191N63-abs.png', dpi = 200)


# In[20]:


phi_fluct_bar= data_phi - np.nanmean(data_phi, axis = 0, keepdims=True)
phi_fluct_phiz= np.nanmean(phi_fluct_bar, axis = (1,2))


# In[38]:


fig, ax = plt.subplots()
pcm = plt.pcolormesh(timescale, sscale, phi_fluct_phiz.T,cmap='RdYlBu_r',shading='auto', vmax = 200, vmin = -200)
fig.suptitle(r'Ek = $5\times 10^{-5}$, Ra = 600')
ax.set_xlabel('years')
ax.set_ylabel('s', rotation = 0, labelpad = 10)
cb = fig.colorbar(pcm)
plt.axhline(r_inner)
cb.set_label(r'$u^\prime_\phi$', rotation=0)
#plt.savefig(mydir + '/fluct_bar_bracket_Ek5e-5Ra300.png', dpi=500)
```

2.3. Zonal flow. This code calculated the zonal flow, figure 3.6. The core calculation is rather easy, I put it in a separate file so as to not clog the main analysis file. It is originally an ipython notebook.

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import h5py
import os
import re
import time
```

```
# In[2]:


def get_file_name(mydir, head, end):
    files = []
    for file in os.listdir(mydir):
        if file.endswith(".h5"):
            files.append(os.path.join(mydir, file))
    files.sort(key=lambda f: int(re.sub('\D','', f)))

    files = files[head:end]

    return files


# In[16]:


mydir ="data_Ek5e-6Ra600_L191N63_initmax5e-7-u"
file_names = get_file_name(mydir,50,450)
timescale = []
kinetic = []
phi_kinetic = []
diff =[]
raw_data = []


# In[17]:


def vol_average(data, phi, theta, r):
    phi_bar = np.mean(data, axis=-3)
    theta_bar = -1*np.trapz(phi_bar * np.sin(theta[:,None]), theta, axis = -2) #theta is inverted
    r_bar = np.trapz(theta_bar * r**2, r, axis = -1)
    bar = r_bar/ (2/3*np.abs(r[0]**3-r[-1]**3))
    return bar


# In[21]:


for file_name in file_names:
    with h5py.File(file_name,"r") as df:
        dataset = df['tasks/u']
        rscale = dataset.dims[3][0][:]
        thetascale = dataset.dims[2][0][:]
        phiscale = dataset.dims[1][0][:]
        timescale.append(dataset.dims[0][0][:])
        us = dataset[:]

        for u in us:
            u_square = np.einsum('i...,i...->...', u, u)
            u_phi_square = u[0]**2
            u_diff = u_square - u_phi_square

            u_square_bar = vol_average(u_square, phiscale, thetascale, rscale)
            u_phi_square_bar = vol_average(u_phi_square,  phiscale, thetascale, rscale)
            u_diff_bar = vol_average(u_diff,  phiscale, thetascale, rscale)
            kinetic.append(u_square_bar)
            phi_kinetic.append(u_phi_square_bar)
            diff.append(u_diff_bar)


# In[23]:

timescale *= 700
timescale = np.array(timescale).ravel()
kinetic = np.array(kinetic)
phi_kinetic = np.array(phi_kinetic)
diff = np.array(diff)


# In[30]:


fig, ax = plt.subplots(figsize =(16,9), dpi = 100)
ax.plot(timescale, kinetic)
ax.plot(timescale, phi_kinetic)
ax.plot(timescale, diff)
ax.legend(('Total Flow', 'Zonal Flow','Residual'))
#ax.set_yscale('log')
ax.set_ylabel("Kinetic Energy")
ax.set_xlabel("Years")
ax.xaxis.grid(True)
ax.locator_params(nbins=10, axis='x')
ax.set_title("Ek5e-6Ra600 L191N63 Kinetic Energy")


# In[31]:


fig.savefig("Ek5e-6Ra600_L191N95_Kinetic_Energy", dpi = 200)
```