Rayleigh-Bénard Convection Codebase User Manual

Version 0.0

Oliver Brown

September 15, 2025

Contents

Chapter 1

Introduction

This document is designed for a math and computer science student, who has knowledge of both basic Python programming, and the math behind Rayleigh-Bénard Convection. The purpose of this document is to orient an undergraduate student with David Goluskin's research project, which a few students have already worked on. Users are encouraged to read through the entire document (and the suggested links), ask Dr. Goluskin for help understanding anything unclear, and as a last resort, email Oliver Brown.

The purpose of this research project is to explore a variety of fluids that have not been extensively simulated or experimentaly observed, with hopes of either proving or disproving current theories regarding Rayleigh-Bénard Convection (RBC). These fluids are largely similar to liquid metals. The simulations that we're running are venturing into the space of superheated liquid metals, but before they turn into the materials and states present in the cores of stars. These materials are obviously of practical astrophysical importance, but the mathematical properties of the simulations ran, the logistics of the simulations, and the visualized results will help to answer research questions in fluid dynamics, scientific computing, all the while providing high quality visualizations for science outreach.

To begin, we should talk about the setup of the problem. Rayleigh-Bénard Convection is a specific model of fluid dynamics in which we have a container of fluid that is being heated at the bottom and cooled at the top. The experiment setup (temperature difference, gravitational force, height difference from top to bottom, etc.) can all be put into one dimensionless parameter; the Rayleigh number. The liquid characteristics, in conjunction with the ralyeigh number, can be characterized with another dimensionless parameter; the Prandtl number. The Prandtl number defines the ratio of "kinematic viscosity" to "heat diffusivity". Essentially, it tells you whether velocity at one point affects it's neighbouring fluid more or less than temperature affects its neighbouring fluid. Thus a very low Prandtl number implies that the heat diffuses much faster than the velocity, i.e. the velocity will play a much larger role in the fluid's movement than the temperature. This yields extremely fast moving fluids, and that speed will impact not only the larger fluid in general, but also very small scale regions, creating a delicate and intricate structure.

Dimension also plays a large role in our simulations. Three spatial dimensions (along with time) is the world in which we live in, however, slices of these 3D boxes (aka 2D planes) are also important in researching RBC. The majority of the time I will not specify whether we are discussing 2 or 3 dimensions, as the majority of the time it does not matter. When referring to the vertical direction, I am referring to the direction along which we have the temperature difference, which will always be denoted z or x_3. When referring to the horizontal direction, I am referring to the other directions. For 3D, this is the combination of the x and y directions, and for 2D it is just the x direction (there is no y). The horizontal directions will always be periodic. This can be weird to think about, but it's like in Pac-Man, how when you go off the side of the screen, you instantly appear on the other side of the screen. It's as if you took a sheet of paper and rolled it into a circle; anytime you draw a line going in a straight line horizontally and it reaches the end of the page, it instantly jumps to the other side.

This can all be quite confusing; it might be useful to just think of a rectangular container of water that is being heated evenly along the bottom while being kept at a constant room temperature on the top, as to not get lost in the jargon. Thinking of that water setup will be perfectly alright for this user manual.

The math behind the fluid dynamics models is necessarily a bit complicated. Instead of deriving the Navier-Stokes equations for general fluid motion, and then considering the forces from the buoyancy of heating the liquid to arrive at the Boussinesq approximation, I will state the partial differential equations (PDEs) that govern the fluid flow here. $\mathbf{x} = (x, y, z)$ denotes the position in 3D space, that is, $\mathbf{x} \in \mathbb{R}^3$. $\mathbf{u} = (u, v, w)$ denotes the velocity field of the fluid in 3D space, so $\mathbf{u}$ is a vector field. It takes vector inputs (position coordinates) and outputs vectors (the velocity of the fluid at that position). T, or temperature, is a scalar field that gives the temperature of the fluid at a specific point in space. Numerical subscripts indicate components of a vector, and t subscripts denote time derivatives. So x_3 is the third component (aka z) of the $\mathbf{x}$ vector, whereas $\mathbf{u}_t$ indicates the time derivative of the velocity field. Here are the governing PDEs:

$$\mathbf{u}_t + \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla p + Pr\Delta\mathbf{u} + Pr Ra\, T\hat{x}_3$$

$$T_t + \mathbf{u} \cdot \nabla T = \Delta T$$

In a potentially jarring bit of mathematical simplification, we will impose the conditions that the top of the box is at position $\frac{1}{2}$ cooled to $-\frac{1}{2}$, while the bottom is at position $-\frac{1}{2}$ and will be heated to $\frac{1}{2}$. This positioning and negative temperature may be unnatural to think about, but by non-dimensionalizing our PDEs, everything works out alright.

The math behind this model is not really what this user manual is meant to elucidate, so I will now skip to the research that we are doing using these mathematical models.

Chapter 2

Nusselt Number and Scaling Laws

There are many open questions in fluid dynamics and RBC, but the one that we are exploring with these simulations is how the Nusselt number (defined below) scales with change in Rayleigh and Prandtl numbers.

The Nusselt number is a measure of how well a certain experiment (this takes into account the Rayleigh and Prandtl number) transports heat. You can imagine that a slow moving fluid, i.e. one that is not being heated much at all, probably does not transport that much heat as compared to something at a rolling boil. The more the fluid moves hot liquid up and the cold down, the higher the Nusselt number.

The lowest possible heat transport is via a stationary fluid; all the heat transport is diffusion, as the liquid is not being heated enough to move. This is a slower movement of heat than a pot of boiling water, which clearly gets the hot liquid up to the top very quickly. Note that boiling is not something that we are taking into account, as we are not concerned with phase changes, but it is a very clear indicator that the heat is being moved quickly. Convection rolls are perhaps a better example, as they show the heat being moved around efficiently without phase changes, but they are less often observed. The Nusselt number is then calculated as a ratio of heat transport from a version of the experiment that is stationary as compared to the actual experiment:

$$\text{Nusselt} = \frac{\text{Heat transport of the actual experiment}}{\text{Heat transport of the fluid without movement}} \geq 1$$

The Nusselt number cannot be less than 1, as the heat transport can never be less than if the fluid was stationary. Intuitively, any movement *should* increase the transport of the heat.

There are multiple equivalent ways to calculate the Nusselt number: One can measure the total amount that the fluid is spinning at each point in the simulation, or measure the product of the temperature and the vertical velocity at each point, or a few other ways, and then (for any of these measurements) take a spatial average of these quantities, and then take a time average of these quantites. These averages will give us the Nusselt number, which we can then compute at many different Rayleigh-Prandtl number combinations to discover how the Nusselt number changes with a change in parameters.

Many "grand unified theories of Nusselt scaling" have been proposed and revised as more and more experiments are completed. This research project aims to dig into a region of parameter space (combination of Rayleigh and Prandtl number) that has little good quality data for. This area consists of low Prandtl (10^0 - 10^{-4}) and high Rayleigh (10^5 - 10^7) numbers. Using the data from the simulations we are running, certain theories can either be disproven and suggestions can be made, or the theory will be shown to fit simulation data, bolstering confidence in that mathematical model.

It's useful to note that there are other things besides the Nusselt number that we are interested in, such as the Reynolds number, and generating high quality movies of the fluid moving.

Chapter 3

Implementation

Now we can get into the core of the research project; actually running the simulations! That means this section is probably the most important one for understanding how to complete the typical tasks required for this research project.

3.1 The HPCs

To begin, a Digital Research Alliance of Canada (DRAC) account must be created, and access must be given to the project. This account is necessary for access to the supercomputers Trillium, Fir, Rorqual, and Nibi. Each supercomputer is specialized for a different purpose, but all of them fall under the "large parallel computations" umbrella. There are many other supercomputers available, and new ones may appear within the next few years. With this specific research project, it is very important to utilize all the resources available, as these simulations take huge amounts of core-hours to run.

I won't go too far into the details of these computers, as the DRAC has extensive documentation on how to use them. Please read the articles Getting started, Running jobs, the article for whichever supercomputer you wish to use, and then Getting help before continuing with this user manual.

Once on the supercomputer, the proper area for saving simulation data must be chosen. The correct area should be somewhere with a high read/write speed, and hopefully optimized for many small files. This is usually the scratch directory. When the correct directory has been determined, the github repository must be cloned into the directory, as it contains all of the scripts required for running the simulations. As well as the github repo, another folder must be created to house all of the simulation data, and the simulation data folder must adhere to the following structure.

3.2 Folder Structure

Each simulation is characterized by 4 variables. The Rayleigh number, which takes values as $Ra \in [10^5, 10^6, 10^7]$, the Prandtl number, which takes values as $Pr \in [10^0, 10^{-0.5}, 10^{-1}, ..., 10^{-3.5}, 10^{-4}]$, the aspect ratio (AR) of the box of fluid, which is a float defining how many times larger the horizontal dimensions are than the vertical (2 for all of our cases, but research has been done with ARs of 0.5 to 65), and the resolution of the fluid (extensive effort has been put in to discover proper resolution scaling, and resolution checks were completed to ensure we were properly resolved). More on resolution checks in a later section.

Using these 4 variables, we can create a folder structure that stores all the information required for each simulation, freeing the file names to be pithy. The first two defining variables are chosen to span the high Rayleigh - low Prandtl parameter space, the AR is set constant to 2 (but experiments were done with higher ARs, so it remains a variable portion of the organization), and the last defining variable, the resolution, is chosen for the specific combination of the first 3 parameters. That is, the resolution is a function of the Rayleigh number, Prandtl number, and aspect ratio. Increasing Rayleigh requires increasing the resolution in all directions (vertical and horizontal, aka the x, y, and z directions). Decreasing the Prandtl number requires increasing the resolution in all directions. Increasing the AR requires increasing the resolution, but *only* in the horizontal directions (x and y). A more detailed analysis on resolution and how to choose the proper number is given later.

To wrap-up this section, here is a text tree of how the folder structure should look (only Ra$= 10^5$ for brevity). Note that pr3e-x is equivalent to Prandtl $= 10^{-(x-0.5)}$, but due to the linux filesystem, we should not have additional periods in the filename.

```
rb_data
└── ra1e5
    ├── pr1e0
    │   └── 48_Gam2
    ├── pr1e-1
    │   └── 80_Gam2
    ├── pr1e-2
    │   └── 112_Gam2
    ├── pr1e-3
    │   └── 160_Gam2
    ├── pr1e-4
    │   └── 208_Gam2
    ├── pr3e-1
    │   └── 48_Gam2
    ├── pr3e-2
    │   └── 96_Gam2
    ├── pr3e-3
    │   └── 128_Gam2
    └── pr3e-4
        └── 176_Gam2
```

Chapter 4

Python

This section is dedicated to the Python script (and the package powering said script) that runs the fluid simulation. This package is called Dedalus. Please read the documentation before continuing. Many other packages are required to run Dedalus, such as mpi4py, fftw-mpi, hdf5-mpi, numpy, scipy, and various others. However, the environments required to run Dedalus on the supercomputers have already been set up, and instructions to recreate them are below. There are many tutorials and examples on using Dedalus on their website.

We are currently in the process of updating the scripts, as this project began around 2020 but was put on hold for a while, and in that time there have been significant changes to computer architecture and supercomputer design. This has led to increasing difficulty continuing with Dedalus version 2. So, although this guide will help understand the Dedalus 2 Python script, it will be updated in the future to have the version 3 information. Now we will go through the Python script (for 2D, and then discuss modifications for 3D). To start, here are the imports:

```
from docopt import docopt # Collecting simulation parameters
import numpy as np
from mpi4py import MPI # pyright: ignore
import time
from pathlib import Path

from dedalus import public as de # Dedalus import
from dedalus.extras import flow_tools # For variable timestepping

import logging
logger = logging.getLogger(__name__)

from dedalus.tools.config import config
config['logging']['stdout_level'] = 'debug' # Output more information in
```

This should all be fairly straightforward. Next we have the gathering of the simulation parameters.

```
args = docopt(__doc__)

# Fix non-dimensional height of the layer at 1
```

```
Lx, Lz = (np.float64(args['--Gamma']), 1)

# Rayleigh and Prandtl number
Pr = 10**np.float64(args['--Pr_exp'])
Ra = np.float64(args['--Ra'])

# Resolution for the simulation
nx, nz = (int(Lx*int(args['--res'])), int(Lz*int(args['--res'])))

# Problem and solver params
stepper = str(args['--stepper'])
flow_bc = "no-slip"
arg_dt = np.float64(args['--dt'])
stop_sim_time = np.float64(args['--sim_time'])
cfl = args['--cfl']

# File parameters
basepath = Path(str(args['--basepath']))
snapshots = args['--snapshots']
restart_index = int(args['--index'])

# Iterations between saves
state_iters = 5000
analysis_iters = 25
field_analysis_iters = 200
snapshots_iters = 200
message_num_iters = 500

if Ra/Pr >= 1e10:
    # The timestep goes very low when Ra big and Pr low; saves
    # should be less frequent.
    message_num_iters *= 10
    state_iters *= 10
    snapshots_iters *= 2
```

This should also be fairly straightforward. Here are a few parts to note:

stepper decides which timestepper should be used; there are a few options available, and they typically have trade offs associated with choosing one over another. The most common one that will be used for "less expensive" cases is RK222, whereas RK443 should be used when the resolution needs get quite high. The "cost" of a simulation will be discussed later.

flow_bc tells us what we want our boundary conditions to be. The only one that should be used is no-slip, but I assume there was some testing done previously, so it was good to be able to vary it easily. Do not change this from no-slip though.

cfl is a parameter that, when True, enables variable timestepping. CFL stands for Courant-Friedrich-Lewy, who were three mathematicians that created the conditions determining optimal timestepping for each step. There are many parameters here that can be tweaked, but the main thing to note is that when properly resolved, the parameters should be fine to be default.

The **_iters* variables determine how often certain quantities should be saved. For example, *state_iters* set to 5000 implies that every 5000 timesteps we save the entirety of the governing variables, so that we can restart the simulation from a fairly recent checkpoint. We don't to miss a large chunk of simulation because we didn't want to save the state often enough, but we also don't want to slow down the simulation by saving big chunks of data all the time. This is why there is a check to determine whereabouts in parameter space we are; if $\frac{Ra}{Pr} \geq 10^{10}$, the timestep goes very small, so we have to do many many iterations to get a substantial change from the previous checkpoint to the next, so we increase the number of timesteps required before checkpointing. The same logic holds for the other variables. *message_num_iters* is solely for cleanliness of the text output for the simulation; it determines how often we should print some information on how the flow is progressing.

Moving on further, we have code to set up the Dedalus portion:

```python
# Create bases and domain
x_basis = de.Fourier('x', nx, interval=(0, Lx), dealias=3/2)
z_basis = de.Chebyshev('z', nz, interval=(-Lz/2, Lz/2), dealias=3/2)
domain = de.Domain([x_basis, z_basis], grid_dtype=np.float64)

# Problem setup: 2D Boussinesq equations; non-dimensionalized by the
# buoyancy time and written in terms of the vorticity.
problem = de.IVP(domain, variables=['p','T','u','w','Tz','oy'])

# Boundary conditions
problem.meta['p','T','u','w']['z']['dirichlet'] = True

# Parameters
problem.parameters['P'] = (Ra * Pr)**(-1/2)
problem.parameters['R'] = (Ra / Pr)**(-1/2)
problem.parameters['F'] = F = 1
problem.parameters['kappa_xz'] = 1/(Lx*Lz)
problem.parameters['kappa_x'] = 1/Lx


# Equations describing the PDE, and first order reductions
problem.add_equation("dx(u) + dz(w) = 0")
problem.add_equation(
    "dt(T) - P*(d(T,x=2) + dz(Tz)) - F*w  = -(u*dx(T) + w*Tz)"
)
problem.add_equation(
    "dt(u) - R*dz(oy) + dx(p) = -oy*w"
)
problem.add_equation("
    dt(w) + R*dx(oy) + dz(p) - T = oy*u"
)
problem.add_equation(
    "Tz - dz(T) = 0"
)
problem.add_equation(
    "oy + dx(w) - dz(u) = 0"
)


problem.add_bc("left(T) = 0")
problem.add_bc("right(T) = 0")

if flow_bc == "no-slip":
    problem.add_bc("left(u) = 0")
    problem.add_bc("right(u) = 0")

if flow_bc == "stress-free":
    problem.add_bc("left(oy) = 0")
    problem.add_bc("right(oy) = 0")

problem.add_bc("left(w) = 0")
problem.add_bc(
    "right(w) = 0", condition="(nx != 0)"
)
problem.add_bc(
    "right(p) = 0", condition="(nx == 0)"
)


# Build solver; timestepper is passed as a CLA
solver=None
if stepper == 'RK222':
    solver = problem.build_solver(de.timesteppers.RK222)
elif stepper == 'CNAB2':
    solver = problem.build_solver(de.timesteppers.CNAB2)
elif stepper == 'MCNAB2':
    solver = problem.build_solver(de.timesteppers.MCNAB2)
elif stepper == 'SBDF4':
    solver = problem.build_solver(de.timesteppers.SBDF4)
else:
    solver = problem.build_solver(de.timesteppers.RK443)
logger.info(f'Solver {stepper} built')

solver.stop_sim_time = stop_sim_time
```

Again, once the Dedalus documentation has been read and understood, this should be fairly simple. Moving on now to the section on deciding whether a simulation should be started fresh or restarted from a checkpoint. If there does not exist a restart file, we start the simulation from scratch. This is done by setting the velocity to 0 everywhere, adding a tiny perturbation to the temperature field at all points (recommended, I'm not sure why), and then letting the temperature difference do the work as the simulation progresses.

If there was a restart file we load it, and then check if the timestep that was previously used is larger than what was provided as an argument to the script. If the timestep provided was smaller, use that one.

```python
restart_path = basepath / "restart/restart.h5"
if not restart_path.exists():
    # Start simulation from a static state;
    # initial conditions are parallel friendly
    logger.info("Restart path not found.")
    z = domain.grid(1)
    T = solver.state['T']
    Tz = solver.state['Tz']

    # Random perturbations, initialized globally for same
    # results in parallel
    gshape = domain.dist.grid_layout.global_shape(scales=1)
    slices = domain.dist.grid_layout.slices(scales=1)
    rand = np.random.RandomState(seed=42)
    noise = rand.standard_normal(gshape)[slices] # pyright: ignore

    # Linear background + perturbations damped at walls
    zb, zt = z_basis.interval
    pert =  1e-3 * noise * (zt - z) * (z - zb)
    T['g'] = F*pert
    T.differentiate('z', out=Tz)

    # Timestepping and output
    dt = arg_dt
    fh_mode = 'overwrite'

else:
# restart simulation from a given initial condition, specified
# as an index in a .h5 file
    # Restart
    write, last_dt = solver.load_state(restart_path, restart_index)

    # Timestepping and output - if the last sim was using a
    # smaller timestep, it's a good idea to use that
    # instead of the user specified.
    if cfl:
        dt = min(last_dt, arg_dt)
    else:
        dt = arg_dt

    fh_mode = 'append'
```

The next section is the file saving section; this is where we specify which quantities should be saved, at what frequency, and where.

```
# For movie making magic; use command line argument
# --snapshots to activate
if snapshots:
    snapshots_file = basepath / "snapshots"
    snapshots_file.mkdir(exist_ok=True)

    snapshots = solver.evaluator.add_file_handler(
        snapshots_file,
        iter=snapshots_iters,
        max_writes=1000,
        mode=fh_mode
    )
    snapshots.add_task("T-(z-1/2)", name = 'temp')
    snapshots.add_task("oy", name='vorticity')

# States saved as checkpoints for restarting. Can adjust iter
# as necessary.
# Infrequent saves are better when the simulation runs quickly
# (speed reasons).
state_file = basepath / 'state'
state_file.mkdir(exist_ok=True)
state = solver.evaluator.add_file_handler(
    state_file,
    iter=state_iters,
    max_writes=1000,
    mode=fh_mode
)
state.add_system(solver.state)

# For Nu and Re calculations - data saved at 1 point in
# space FREQUENTLY
analysis = basepath / 'analysis'
analysis.mkdir(exist_ok=True)
analysis = solver.evaluator.add_file_handler(
    analysis,
    iter=analysis_iters,
    max_writes=1000,
    mode=fh_mode
)

analysis.add_task('R/P', name='Pr')
analysis.add_task('1/(P*R)', name='Ra')

# For calculating Nu
analysis.add_task(
    "kappa_xz * integ_z(integ_x( (w*(T+1/2-z))))", name='avg_wT'
)
```

```
analysis.add_task(
    "kappa_xz*integ_z(integ_x((dx(u)**2)+dz(u)**2+dx(w)**2+dz(w)**2))",
    name='avg_grad_u_sq'
)
analysis.add_task(
    "kappa_xz*integ_z(integ_x((dx(T+1/2-z))**2+(dz(T+1/2-z)**2) ))",
    name='avg_grad_T_sq'
)
analysis.add_task(
    "kappa_xz * integ_z( integ_x( (oy**2) ))", name='avg_oy_sq'
)

# For calculating Re and plotting profiles
analysis.add_task(
    "kappa_xz*integ_z(integ_x( u**2 + w**2 ))",
    name='avg_K'
)
analysis.add_task("kappa_x*integ_x( T )", name='avg_T')
analysis.add_task("kappa_x*integ_x( u**2 )", name='avg_u_sq')
analysis.add_task("kappa_x*integ_x( w**2 )", name='avg_w_sq')
analysis.add_task("kappa_x*integ_x( T**2 )", name='avg_T_sq')


# For analysis - data saved at 1 point in space INFREQUENTLY
field_analysis_file = basepath / 'field_analysis'
field_analysis_file.mkdir(exist_ok=True)
field_analysis = solver.evaluator.add_file_handler(
    field_analysis_file,
    iter=field_analysis_iters,
    max_writes=1000,
    mode=fh_mode
)

field_analysis.add_task('R/P', name='Pr')
field_analysis.add_task('1/(P*R)', name='Ra')

field_analysis.add_task("interp(u, x=0, z=0)", name='u')
field_analysis.add_task("interp(w, x=0, z=0)", name='w')
field_analysis.add_task("interp(T, x=0, z=0)", name='T')
field_analysis.add_task(
    "interp(oy*w+1/2*dx(u**2+w**2), x=0, z=0)",
    name='u.grad_u'
)
field_analysis.add_task(
    "interp(-oy*u+1/2*dz(u**2+w**2), x=0, z=0)",
    name='u.grad_w'
)
field_analysis.add_task(
    "interp(u*dx(T)+w*Tz, x=0, z=0)",
    name='u.grad_T'
)
field_analysis.add_task(
    "interp(dx(p)-1/2*dx(u**2+w**2), x=0, z=0)",
    name='p_x
')
field_analysis.add_task(
    "interp(dz(p)-1/2*dz(u**2+w**2), x=0, z=0)", name='p_z')
field_analysis.add_task("interp(d(u, x=2), x=0, z=0)", name='u_xx')
field_analysis.add_task("interp(d(u, z=2), x=0, z=0)", name='u_zz')
field_analysis.add_task("interp(d(w, x=2), x=0, z=0)", name='w_xx')
field_analysis.add_task("interp(d(w, z=2), x=0, z=0)", name='w_zz')
field_analysis.add_task("interp(d(T, x=2), x=0, z=0)", name='T_xx')
field_analysis.add_task("interp(dz(Tz), x=0, z=0)", name='T_zz')
```

For the snapshots section: This is where we save the temperature field and the vorticity (amount that the fluid is spinning) in order to create visualizations. The *max_writes* parameter specifies how many times the quantities should be saved in one file before creating another. This is to avoid massive files that are unwieldly to handle. The *iter* parameter specifies how many timesteps should pass in between the quantities being saved (for snapshots, 200 iterations typically works well). The files are saved in a folder titled "snapshots" in the simulation folder. Each file is labelled *snapshots_s{%d}*, where {%d} is just some integer numbering the files. The *s* after the underscore stands for set.

It's now a good time to talk about the format that the data is stored in: HDF5. When the simulation is running, it creates the sets as a folder, and in each of these folders is one file per processor used. Once the simulation is finished, the snapshots directory will look something like this:

```
snapshots
├── snapshots_s1
│   ├── snapshots_s1_p0.h5
│   ├── snapshots_s1_p1.h5
│   ├── snapshots_s1_p2.h5
│   └── snapshots_s1_p3.h5
└── snapshots_s2
    ├── snapshots_s2_p0.h5
    ├── snapshots_s2_p1.h5
    ├── snapshots_s2_p2.h5
    └── snapshots_s2_p3.h5
```

This is an example if the simulation was run using 4 processors (normally it is somewhere around or above 400), and where we surpassed the *max_writes* number of writes, but only once.

After running the simulation, these set folders of processor files need to be merged into single set files before we can use the data. This is covered in the Workflow section.

The state files are HDF5 files that contain everything required to completely restart a simulation from where it is currently; they are the checkpoints. They hold the velocity, temperature, and pressure information at every grid point in the simulation space. They get very large for 3D, so it is important to keep their max_writes down to 10-25.

The analysis files are what we primarily care about; these are files containing the data for us to calculate the Nusselt number, and the Reynolds number.

The field_analysis files do not seem to have a purpose anywhere - I think they were being saved "just in case" we wanted anything from them, but they are just taking up space currently.

Now for the final section; moving forward in the simulation! There are 2 possible routes, one using variable timestepping and one using a constant timestep. The only time it is reasonable to not use a variable timestep is for creating movies, but while just running the simulation it is unwise to use a constant step.

```
logger.info('Starting loop.')
logger.info("-" * 80)
start_time = time.time()
message_num_iters = 500
start_iter = solver.iteration

# Variable timestepping:
if cfl:
CFL = flow_tools.CFL(
        solver,
        initial_dt=dt,
        cadence=2,
        safety=0.15,
        max_dt=0.1,
        max_change=1.01,
        threshold=0.005
)
CFL.add_velocities(('u', 'w'))
dt = CFL.compute_dt()
dts = []
while solver.proceed:
        solver.step(dt)
        if (solver.iteration-1) % message_num_iters == 0:
        l_dts = len(dts)
        if l_dts > 0:
                logger.info(f"The timestep has changed {l_dts}//
                 time(s) over the last {message_num_iters}//
                 iterations.")
                logger.info(f"The average timestep was//
                 {np.mean(dts):.4e}.")
                logger.info(f"The minimum was {np.min(dts):.4e}.")
                logger.info(f"The maximum was {np.max(dts):.4e}.")
        elif (solver.iteration-1) == 0:
                pass
        else:
                logger.info(f"The timestep did not change over//
                 the last {message_num_iters} iterations.")

        logger.info(f"Iteration: {solver.iteration},//
         Time: {solver.sim_time}, dt: {dt}")
        logger.info("-" * 80)


        dts = []
        new_dt = CFL.compute_dt()

        if new_dt != dt:
        dts.append(new_dt)
        dt = new_dt

# Constant timesteps:
else:
while solver.proceed:
        solver.step(dt)
        if (solver.iteration-1) % message_num_iters == 0:
        logger.info(f"Iteration: {solver.iteration},//
         Time: {solver.sim_time}, dt: {dt}")
```

Moving downwards, we should start with CFL. The parameters are discussed in the dedalus docs, so please read those. The general procedure is the following loop, and the only difference between using CFL and not is in the calculating of the timestep:

```
Calculate the timestep
Move forward one timestep
Display timestep statistics if it has been message_num_iters
Check that we have not reached the stop time specified
Go back to beginning of loop
```

That's all for running the simulation! The last bit is displaying some helpful statistics from the run.

```

# Total time and iterations
end_time = time.time()
end_iter = solver.iteration
total_iter = end_iter-start_iter
total_time = end_time-start_time

# Degrees of freedom for speed calculation
dof = nx * nz * len(problem.variables)

# Write info to job_sim.out file
rt = (total_time)/60/60*domain.dist.comm_cart.size
speed =  (dof * total_iter) / (domain.dist.comm_cart.size * total_time)
logger.info('Iterations: %i' %solver.iteration)
logger.info('Sim end time: %f' %solver.sim_time)
logger.info('Run time: %.2f sec' %(total_time))
logger.info('Run time: %f cpu-hr' %(rt))
logger.info('Speed: %f' %(speed))
```

The speed statistic is essentially calculating the number of calculations done per second, or some multiple of it. The higher the better. Somewhere between 400,000 and 1,000,000 is expected. Congrats, you've made it through the hardest part!

Chapter 5

Workflow

Now that we have our folders organized and understand what the simulation will save, we can move into a simulation folder and begin a run. Found in the github repository, there is a script run_sim.sh, which must be moved into the simulation folder, and then can be customized for your specific simulation. Here is the customizable portion of the script, and then we will go over the actual simulation portion.

```
#!/bin/bash

#SBATCH --output=job_sim.out
#SBATCH --time=02:59:00
#SBATCH --nodes=1
#SBATCH --ntasks=96
#SBATCH --mem=0
#SBATCH --account=def-goluskin
#SBATCH --job-name=r1e7_pr1e-4


##################################################################
# User specified parameters
START_TIME=0
# Rayleigh number
RA=1e7
# Exponent of 10 for Pr. (ie if Pr=0.1 -> PR_EXP=-1)
PR_EXP=-4
# Vertical resolution
RES=200
# Timestep- if using fixed timestep this matters. Otherwise just
# leave sufficiently small that the simulation won't blow up in
#  25 iterations
DT=0.0002
# Dimensionless time to run the simulation for
SIM_TIME=200
# Method for timestepping. Can be RK222, RK443, CNAB2, MCNAB2, SBDF4
STEPPER=RK222
# Aspect ratio
GAM=2
```

```
# Use an initial condition from a previous run? 1=yes, 0=no
IC=1


# When to start averaging - post process
AVG_TIME=0
# Exponent of 10 for minimum y axis on power spectra
POWER_YMIN=-12
# Exponent of 10 for maximum y axis on power spectra
POWER_YMIN=0

#################################################################


# Path to all python scripts for simulations; change as needed
PATH_TO_SCRIPTS="$SCRATCH/rbc_scripts"
SCRIPTS_2D="$SCRATCH/rbc_scripts/2d"
PATH_TO_ENV="$SCRATCH/dedalus"


#################################################################
```

The *#SBATCH* commands should be familiar from the DRAC articles, so let's look at the *User specified parameters* section.

START_TIME determines what time the simulation should begin running from; this is only relevant if you are restarting a simulation. It often comes up if there is a point in the simulation which runs into extreme numerical instability, or there is a section which you would like to rerun for visualization. *START_TIME=0* implies that the most recent time should be used.

RA determines the Rayleigh number. Simple as that.

PR_EXP determines the Prandtl number. The difference in Rayleigh being the exact number and Prandtl being saved as the exponent of 10 is a bit strange to me, but I never bothered to change it. If you want Pr = 1, *PR_EXP* should be set to 0.

RES is the vertical resolution, i.e. how many chebyshev nodes there are. The horizontal resolutions are calculated using this resolution, and the AR. This is an important variable, as simulation data with different resolutions **cannot** be mixed. That is, if you start a simulation using a resolution of 192 but decide that it's underresolved and begin a new simulation using a resolution of 256, you cannot mix the timeseries of any average quantities. However, you **can** start a new simulation using a previous simulation's state even if they have differing resolutions. This is very commonly used, and is referred to as bootstrapping (not to be confused with the statistics term). It is also recommended that the initial transient of a simulation should be ran at a lower resolution than what is required to be properly resolved, and after settling down a little, increasing the resolution to be properly resolved. This allows for less core hours, and shorter wall-times.

DT is the initial timestep used for a simulation. If you have selected CFL, this can be any small number, as it should level itself out quickly.

SIM_TIME is the total number of time units you wish to move forward with the simulation. If you are starting fresh and give *SIM_TIME* = 50, you will end the simulation at time $t = 50$. If you are restarting a simulation from time 100 and give a *SIM_TIME* = 50, you will end at $t = 150$.

STEPPER determines which timestepper to use; this should be a string, and you can choose from the options listed. A higher order stepper will use more memory and be slower, with the benefit of accuracy, and potentially a larger timestep.

GAM is short for gamma, or Γ, and is the aspect ratio. A gamma of 2 means that the horizontal size will be twice the vertical, and the horizontal resolution will also be double the vertical.

IC stands for **I**nitial **C**ondition, and it will either set up the simulation folder to handle all of the first time run tasks if *IC*=0, or it will assume everything is set up and there is a restart file.

The next few options, *AVG_TIME*, *POWER_YMIN*, and *POWER_YMAX* are for the post processing of the simulation data, so we won't worry about them yet.

PATH_TO_SCRIPTS and the other 2 variables are the path to the cloned github repository, the path to the Python scripts inside the github repository, and the path to the Python environment, respectively.

Now for the simulation portion, as following the workflow diagram:

```bash
# Load the required modules
module load StdEnv/2020
ml python/3.10.2
ml mpi4py/3.1.3
ml fftw-mpi/3.3.8
ml hdf5-mpi/1.12.1
ml scipy-stack/2023b

source $PATH_TO_ENV/bin/activate;

# Dedalus performance tip!
export OMP_NUM_THREADS=1
export NUMEXPR_MAX_THREADS=1


# specify desired time for initial condition
if [ $IC -eq 1 ]; then
  IC_ARRAY=($(python3 $PATH_TO_SCRIPTS/initial_condition.py \\
   $START_TIME --file=$PWD/restart/restart.h5))
  TOTAL_TIME=$(echo "$SIM_TIME+${IC_ARRAY[1]}" | bc)
  IND=${IC_ARRAY[0]}

else
  echo "Not running with a prior simulation's initial conditions"
  #rm -rf {analysis,state,preliminary_outputs}
  #rm -rf {snapshots,outputs,field_analysis}
  rm -rf restart

  TOTAL_TIME=$SIM_TIME
  IND=-1
fi
```

```
srun python3 $SCRIPTS_2D/rayleigh_benard_script.py \\
 --Ra=$RA --Pr_exp=$PR_EXP --res=$RES --dt=$DT \\
 --sim_time=$TOTAL_TIME --stepper=$STEPPER \\
 --Gamma=$GAM --index=$IND --basepath=$PWD --cfl --snapshots
```

First up, the *module load* commands are to put the proper software onto the compute nodes, as the nodes do not have Python installed on them to begin. Note that StdEnv/2020 is pretty old software for very new hardware, which is why Dedalus 3 is currently being implemented. The required modules for Dedalus are Python and mpi4py - the others could be pip installed in the Python environment, and I haven't done a test if there's a noticeable difference in runtimes, but I suspect that loading the modules and then pip installing whatever is left is going to be quickest. The module versions are also added, so that the simulations are exactly the same every time. Please read the DRAC article Python before continuing.

The Python environment is then loaded, and it is required to have been created prior to running any scripts. Here is a short aside to creating the environment:

- Load the required modules (we just saw which they are)
- Run the command *virtualenv –no-download ENVIRONMENT_NAME*
- Run the command *source ENVIRONMENT_NAME*
- Now we're in the Python environment. We need to install the required packages. Run the following commands:
  - pip install –no-index –upgrade pip
  - pip install –no-index dedalus

The Python environment will then be set up and ready for simulations, which is what the *source $PATH_TO_ENV* line does.

The *export OMP_NUM_THREADS* and proceeding line are tips as posted on the Dedalus website, essentially it is telling the computer to not try to do too much multiprocessing at once.

Next up is determining where to start the simulation from. The first option in that if/else check deals with starting from a checkpoint, and the else deals with starting fresh. The script *initial_condition.py* looks at the restart file, determines which write in the file is closest to the *START_TIME* variable, and then gives back the time and the index of the write. This all works very well and should not be changed.

Now that we have determined all the parameters for the simulation, we run it with the *srun python3 rayleigh_benard_script.py* command.

Inside each simulation folder (that is, the final level of the Rayleigh/Prandtl/resolution_AR organization), there only need to be a few shell scripts to begin running a simulation. All the shell scripts are contained inside the github repo, and there is a handy script sim_ready.sh which will copy all the required scripts into wherever sim_ready.sh is ran. So, for example, to run a simulation at from the example directory tree, we would move into ra