Paul Sabatier University

University of Exeter

INTERNSHIP REPORT

The effects of rotation and stratification in simulations of turbulent convection

Author :
Enguerran Vidal

Internship Supervisor :
Matthew Browning

25th of February - 22nd of May 2020

Contents

Introduction

As part of the Parcours Spécial Physics Bachelor dispensed by the Paul Sabatier University located in Toulouse, France, a three months long internship is scheduled for all third years during their second semester. Being attracted by the possibility of studying and working abroad, I had the chance to have my internship be in the Department of Physics and Astronomy of the University of Exeter in the United Kingdom. Its planning has not been without troubles however, as Brexit has been a looming threat that could have ruined the entire venture as were the recent news coming from China as the early stages of the COVID-19 pandemic were unfolding in mid-January. Finally arriving in late February, I began my internship under the supervision of the Professor Matthew Browning and began to work on the effects of rotation and stratification on thermal convection, mostly focusing on the Boussinesq and anelastic approximations for incompressible and compressible fluids respectively. The resulting sets of equations were solved using regular Python, then using the Dedalus open source code, in 2D boxes containing a fluid heated from below and cooled from above. The main focus of the internship was to create my own codes and study the different ways to simulate thermal convection. Later in the internship, I was sadly forced to leave the United Kingdom as France was beginning to close its borders and issuing a national lock-down in mid-March amidst the spread of the COVID-19 virus in Europe. I was therefore fearing the possibility of being stuck abroad after the internship. Thankfully however, I was able to continue my work remotely after returning to France, while maintaining contact with my internship supervisor.

The Internship Setting

2.1 The University

The University of Exeter is a public research university primarily located in Exeter, Devon in South West England in the United Kingdom. It was founded in 1955 although most of its predecessors, notably the St Luke's College or the Exeter School of Science were established throughout the 19th Century.

The university is composed of four campuses : Streatham and St Luke's which are located in Exeter and Truro and Penryn in the city of Cornwall. However, the majority of administrative buildings and institutions are located on the Streatham Campus. Named "University of the Year" by the Sunday Times in 2013, its research effort is focused on a wide range of interdisciplinary themes, including extrasolar planets, genomics, climate change and medical history to name a few.

Within the university, there are around 70 research centers and institutes, making it one of the leading university of the United Kingdom.

2.2 The Astrophysics Group

The Department of Physics and Astronomy is a part of the College of Engineering, Mathematics and Physical Sciences, containing different teams responsible for the many sectors of research, such as the Astrophysics group in which the internship took place. The Astrophysics Group makes use of a number of on-campus facilities in addition to the relationships it possesses with multiple renowned international observatories. Two major on-campus assets can be named :

- The University of Exeter High Performance Computing Facility can be used by the Astrophysics Group to run highly demanding numerical simulations. It was launched in 2017, composed of more than 200 compute nodes.
- The team also maintains an observatory on the Streatham campus that is remotely operated through a computer-controlled mount and dome, hosting a 14 inches wide Schmidt-Cassegrain telescope. However it is more frequently used as an undergraduate teaching tool.

The Physics Building located on the Streatham campus hosts the offices of most of the Department, the Astrophysics Group being mainly on the 4th and 5th floor of the building.

The team is taking part of one the main focus of the University : Extrasolar planets, but it also focuses more broadly on Stellar Physics, Stars and Planets Formation as well as the Physics of Interstellar Medium.

Fluid Modeling in a 2D Box

3.1 The Boussinesq Approximation

The goal of this project is to come up with a good model for the study of the thermal convection of a fluid trapped in a 2D box. It is therefore needed to use a continuity equation, a momentum conservation equation as well as an energy conservation equation since the study of thermal convection implies the need to track energy inside the box.

A good model to start from is an incompressible viscous fluid ($\rho = cst$ throughout the fluid, with ρ being its density). However, it is preferred to use the Boussinesq approximation to simulate thermal convection in a simple manner. Basically, the density would now be $\rho = \rho_0(1 - \alpha(T - T_0))$ with α being the coefficient of thermal expansion. With this slight density variation, we arrive to these equations found in Gary A. Glatzmaier's book on convection modelling [3] :

$$\rho_0 \left[\frac{\partial \vec{u}}{\partial t} + (\vec{u}.\nabla)\vec{u} \right] = -\nabla P + \mu \nabla^2 \vec{u} + \rho_0 \alpha \vec{g}(T - T_0) \tag{3.1}$$

$$\nabla.\vec{u} = 0 \tag{3.2}$$

$$\frac{\partial T}{\partial t} + (\vec{u}.\nabla)T = \kappa \nabla^2 \vec{T} \tag{3.3}$$

Where $\vec{u} = (u, v)$ is the fluid velocity field, u and v being its horizontal and vertical components respectively, T is the temperature field, P is the Pressure field, μ is the fluid's dynamic viscosity, κ is its thermal diffusivity and $\vec{g} = g\hat{\mathbf{z}}$ the vertical downward gravity vector field.

To go even further, it is possible to get the dimensionless versions of (3.1) and (3.3). Rhe 2D Box will be set to have a length of L and a depth of D. It helps to define an aspect ratio $a = L/D$ as well as the temperature drop across the depth ΔT. The resulting dimensionless equations are :

$$\frac{\partial \vec{u}}{\partial t} + (\vec{u}.\nabla)\vec{u} = -\nabla P + Pr \nabla^2 \vec{u} + RaPrT\hat{\mathbf{z}} \tag{3.4}$$

$$\nabla.\vec{u} = 0 \tag{3.5}$$

$$\frac{\partial T}{\partial t} + (\vec{u}.\nabla)T = \nabla^2 \vec{T} \tag{3.6}$$

Ra and Pr, being the Rayleigh and Prandlt numbers respectively, are defined like so :

$$Ra = \frac{\alpha g \Delta T D^3}{\nu \kappa} \qquad Pr = \frac{\nu}{\kappa} \tag{3.7}$$

With $\nu = \mu/\rho_0$ the cinematic viscosity coefficient and a the aspect ratio $a = L/D$. This shifts focus from dimensions and values of constants to a couple of dimensionless numbers which is helpful as only these two numbers control the final behaviour of the fluid.

We can also use a Fourier expansion spectral method. In order to do that, we first define the vorticity ω and stream function ψ :

$$\vec{\omega} = \omega\hat{\mathbf{y}} = \nabla \times \vec{u}, \quad \vec{u} = \nabla \times (\psi\hat{\mathbf{y}}) = -\frac{\partial \psi}{\partial z}\hat{\mathbf{x}} + \frac{\partial \psi}{\partial x}\hat{\mathbf{z}} \tag{3.8}$$

Then, we expand T, ψ and ω in both sines and cosines in the x direction and change (3.4), (3.5) and (3.6) as explained in much broader details in [3] . It results in those equations:

$$\frac{\partial T_n}{\partial t} + [(\vec{u}.\nabla)T]_n = \left(\frac{\partial^2 T_n}{\partial z^2} - (\frac{n\pi}{a})^2 T_n\right) \tag{3.9}$$

$$\frac{\partial \omega_n}{\partial t} + [(\vec{u}.\nabla)\omega]_n = RaPr(\frac{n\pi}{a})T_n + Pr\left(\frac{\partial^2 \omega_n}{\partial z^2} - (\frac{n\pi}{a})^2 \omega_n\right) \tag{3.10}$$

$$\omega_n = -\left(\frac{\partial^2 \psi_n}{\partial z^2} - (\frac{n\pi}{a})^2 \psi_n\right) \tag{3.11}$$

3.2 The Anelastic Approximation

This part was done in accordance with Laura K. Currie and Steven M. Tobias study [1]

One thing that the Boussinesq approximation does not account for is a change of density along the z direction as it is observed often in nature, so called "stratification" in our report title. To be able to study its effect, it would be preferred the density would not remain the same throughout the fluid. The anelastic approximation is regularly used for that purpose. We find these equations by decomposing the pressure, temperature and pressure in a reference state and a perturbation which value fluctuates around the reference state.

$$\rho = \rho_0(\bar{\rho} + \epsilon\rho'), \quad P = P_0(\bar{P} + \epsilon P'), \quad T = T_0(\bar{T} + \epsilon T') \tag{3.12}$$

It is preferred to replace the temperature by the entropy in the heat conservation equation in our case and to consider an ideal gas so that $P = R\rho T$ with R being the universal gas constant. If we get rid of the primes to note the fluctuations it gives us the dimensionless anelastic equations :

$$\left[\frac{\partial \vec{u}}{\partial t} + (\vec{u}.\nabla)\vec{u}\right] = -\nabla\left(\frac{P}{\bar{\rho}}\right) + RaPrS\hat{\mathbf{z}} - Ta^{\frac{1}{2}}Pr\vec{\Omega} \times \vec{u} + \frac{Pr}{\bar{\rho}}\nabla.\boldsymbol{\zeta} \tag{3.13}$$

$$\nabla.(\bar{\rho}\vec{u}) = 0 \tag{3.14}$$

$$\bar{\rho}\bar{T}\left[\frac{\partial S}{\partial t} + (\vec{u}.\nabla)S\right] = \nabla.[\bar{T}\nabla S] - \frac{\theta S}{\bar{\rho}Ra}\frac{\zeta^2}{2} \tag{3.15}$$

$\boldsymbol{\zeta}$ is the stress tensor defined by $\zeta_{ij} = \bar{\rho}[\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} - \frac{2}{3}(\nabla.\vec{u})\delta_i j]$ with $\boldsymbol{\zeta}^2 = \boldsymbol{\zeta} : \boldsymbol{\zeta} = \zeta_{ij}\zeta_{ij}$. θ is the dimensionless temperature difference between across the fluid layer, we also take a 3D velocity vector $\vec{u} = (u, v, w)$ and $\vec{\Omega} = (0, \cos(\phi), \sin(\phi))$ is the rotation vector with ϕ being the latitude. The three relevant dimensionless numbers are defined like so :

$$Ra = \frac{gd^3\epsilon}{\kappa\nu}, \quad Ta = \frac{4\Omega^2 d^4}{\nu^2}, \quad Pr = \frac{\nu}{\kappa} \tag{3.16}$$

These are the Rayleigh, Taylor and Prandlt numbers respectively. The reference state is then given by considering it as time-independent and polytopric. To quantify the rotation, we can use the Taylor number but for the stratification, we define $N_\rho = \ln(1+\theta)^{-m}$ the number of density scale heights in the layer. For example if $N_\rho = 0$, we can reduce (3.13)-(3.15) to the Boussinesq equations.

Building Our Own CFD Solver

4.1 First Code : Building a CFD solver from scratch

The resulting code can be found in A.1

Our first task is to write down a Python code that acts as a Computational Fluid Dynamic solver. Its job is to apply the dimensionless equations resulting from the Boussinesq approximation onto a set of fields in order to calculate the ones at the next time step. In the process of doing the first program (using (3.5), (3.5) and (3.5) in 3.1), we quickly face a few hurdles.

How to represent the fields and where to place their values ? The simplest way would be a so called collocated grid, meaning orthogonal and rectangular ,of size $m \times n$ on which we put the pressure P and the components of the velocity u and v at the same places at the intersections of the grid. However, such a layout results in high speed anomalies from odd-even decoupling, more information about this problem can be found in section 1.1.4.2 on the Visual Room website [9].

The easy solution is a staggered grid, using the Marker and Cell method : we define the pressure and velocities on different, separate grids. The layout is composed of cells and the speeds are located on the midpoints of their edges while the pressures are located on their centers as can be seen on 4.1. By using the finite difference method (section 1.1.2.1 [9]), the formulas for the derivatives become those in section 1.1.4.2 [9].

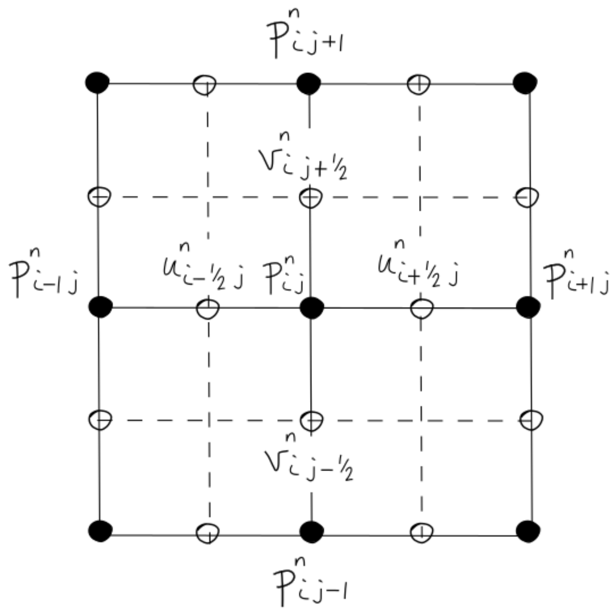


Figure 4.1: Marker and Cell representation from [9]

How to use the velocities to access the pressures using (3.5) ? As we have seen in 3.1, we have no equation linking directly the pressure with the velocity and no way of applying the incompressibility condition present in (3.5) by just using this equation. We therefore need to come up with another equation by applying (3.5) on the divergence of (3.4), giving a Poisson equation. This method is described in section 1.1.4.1 of [9] and in the 10th course of the Lorena A. Barba, CFD in Python course [4] for a fully incompressible fluid. In our Boussinesq 2D case, by using the same method, it gives us :

$$\left[\frac{\partial^2 P}{\partial x^2} + \frac{\partial^2 P}{\partial y^2}\right] = Ra\frac{\partial T}{\partial y} - \frac{1}{Pr}\left[\left(\frac{\partial u}{\partial x}\right)^2 + \frac{\partial u}{\partial y}\frac{\partial v}{\partial x} + \left(\frac{\partial v}{\partial y}\right)^2\right] \tag{4.1}$$

To calculate the pressure field, we build a loop in which the pressure field is updated by using (4.1), this process is described in the 10th course of [4]. At each iteration, the pressure will be more compliant with (3.5), which is why, in our `Poisson_compute` Python function, we prefer to apply it at least 50 times to assure stability. It calls the `B_calculation` whose job is to calculate the Right-hand term in (4.1).

How to calculate the next time step ? To get from a time t to a time $t + dt$, we use an explicit Euler integration method. It is often unstable and tends to diverge from the true solution if given a dt too big. However, in our 2D Boussinesq case, the time-step value has to be controlled so that it does not exceed a certain value and risks to create numerical instability. We therefore define the time step to be $dt = \alpha \Delta z / |v|_{max}$ with α being a safety coefficient added to keep the time-step low. The calculation of dt is handled by `dT_Calculation`.

4.2 Second Code : Spectral Method

The resulting code can be found in A.2

To create our spectral method solver, the Princeton Series book [3] is a great guide by using the first 4 chapters. This time, we apply (3.9),(3.9) and (3.9) on a $nz \times N$ collocated grid where nz is the vertical resolution and N is the truncation level. The Navier stokes equations will now apply on the amplitudes of the Fourier expansions of T, ψ and ω. The resulting code can be found in the appendices, although two main problems have to be mentioned :

- The convective terms in (3.9) and (3.9) use the fluid's velocities even though they are not variables used anymore. How to calculate those terms without having to reconstruct the velocities from the stream function ? Again, Gary A. Glatzmaier book gives formulas in the form of a Galerkin method in section 4.2 [3]. This results in the `Temp_convective_term` and `Curl_convective_term` Python functions in
- How to solve the Poisson equation for the stream function ? In 3.1, 3.11 is a Poisson equation would let us get the stream function from ψ. In section 2.5 of [3], 3.11 is transformed into a tridiagonal matrix problem. We use a method called a Thomas solver, described in [6]. It consists of a LU decomposition in which the lower diagonal is reduced and then the matrix becomes an easy problem to solve. This is the method we use in the `Thomas_solver` Python function in ,

We can clearly predict that this code will be incredibly slow. That low speed would come from the `Temp_convective_term` and `Curl_convective_term` functions responsible for calculating the non-linear convective terms.

Study of Thermal Convection using Dedalus

5.1 What is Dedalus ?

Dedalus is a Python framework developed to solve a broad range of partial differential equations in N-dimensional domains. It possesses a range of key features that make it useful in solving computational fluid dynamics :

- A symbolic equation entry, meaning it accepts nearly any systems of equations by just having to write it in plain text. The same process can be done to apply boundary conditions. However, non linear terms have to be put on the Right-hand side.
- A spectral domain discretization, meaning that Dedalus solves over domains that can be represented by the direct product of spectral bases. The first N-1 need to be discretized in separable bases and the last in a coupled base.
- A quick way to specify analysis tasks saved in HDF5 files.
- An implicit-explicit Runge-Kutta timestepping method.
- The use of fast Fourier transformations through the FFTW module.

Judging by those assets, Dedalus seems to be the wisest choice for modeling any computational fluid dynamics problem. However, it requires to be installed on a machine and a Windows version does not exist for now. I then tried the VirtualBox software from my supervisor's advice. It lets us create a virtual machine on which I put Ubuntu 18.06, an OS with an available Dedalus installation script that uses Miniconda. The procedure and scripts can be found on their website in the documentation [2]. However, the VM did not provide enough memory and CPU-usage for solving CFD problems. I then found a second hand computer that I repaired and installed Dedalus on, therefore becoming my main work machine on which I ran calculations from then on.

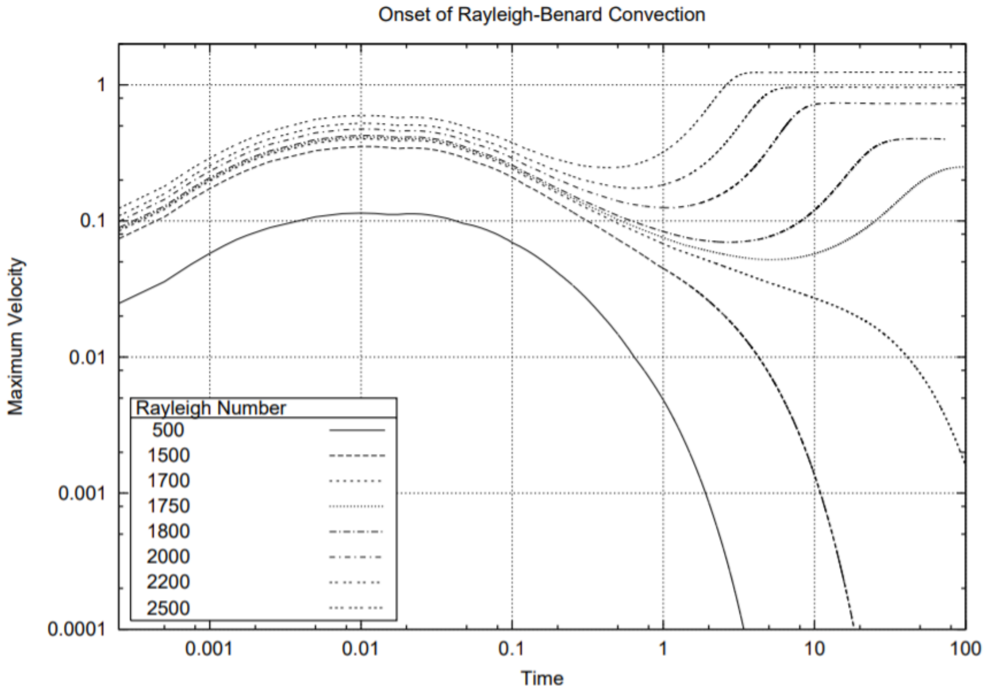
After this bit of trouble I was able to write down my first solver that can be found in Annex A.4. To really apply Dedalus in the best way, I took inspiration from a program solving Boussinesq approximation equations with an immense Prandlt number by Shane Alexander McQuarrie [7]. It helped in configuring the equations entries in plain text as well as the boundary conditions.

5.2 Studying the effects of stratification and rotation

To observe the effects of stratification and rotation on thermal convection in a quantitative manner, our attention needs to be focused on the critical Rayleigh number that we will note Ra_{crit} from now on. This is a key value to assert the behavior of a fluid : if $Ra < Ra_{crit}$, we do not have convection in the fluid and if $Ra > Ra_{crit}$, we have convection in the fluid. Therefore, we can look at the effects stratification and rotation have on this value in order to quantify their magnitudes.

h

Figure 5.1: The maximum velocity as a function of time found in chapter 7 of [8]



But how do we find the Ra_{crit} in a fixed case ? Chapter 7 of S.E. Norris' thesis on "A Parallel Navier Stokes Solver for Natural Convection and Free Surface Flow" [8] gives us a clue. It is described the curve obtained by plotting the maximum velocity over the dimensionless viscous times has a different aspect if the fluid is convective and if it is not. As on Figure (*****), we can observe that both cases follow a " noise " bump at first but they quickly diverge from one another. The convective cases will see their curve rise in an exponential fashion and, id given enough time, will reach a " plateau ". On the other hand, the non-convective cases will see their curves plummet. To find Ra_{crit}, we would only have to test a few values of Ra, find between which values the aspect changes and narrow down Ra_{crit} by dichotomy. Using this method, we have build a procedure to follow :

- Firstly, we would need to look at the non-rotating Boussinesq case ($N_\rho = 0$ and $Ta = 0$) and find its Ra_{crit} to act as a reference point.
- Secondly, we look at a certain value of Ta and fix $N_\rho = 0$ and find Ra_{crit} each time. If enough values are found, this would let us plot $Ra_{crit}(\Omega)$ where Ω is the rotation rate as described in Section 3.2.
- Thirdly, we look at a certain value of N_ρ and fix $Ta = 0$ and find Ra_{crit} each time. It would let us access to $Ra_{crit}(N_\rho)$ with enough values.

To achieve this plan, we were given a code from my supervisor. It was developped by Simon R. W. Lance using his convection notes [5] where he described the anelastic approximation in a different way. Instead of maximum velocity, we prefer to track the kinetic energy of the layer which formula is describe in [5]. I modified the code to fit ours need, it can be found in Annex A.5

Results

6.1 Boussinesq Approximation Observations

Sadly, the two first codes I did (Annex A.1 and A.2) did not give any substantial results. In part due to their incredible slowness, results could not be accessed without letting my computer do calculations for a few days. The Second program slowness is especially surprising, the double sums present in the Galerkin method formulas and the back and forth transformations of fields from spectral to spatial are indeed calculations-heavy algorithms, hence why Dedalus is needed.

In regard of our third Python program, even though it is supposed to be the same (minus the Dedalus assets listed in 5.1) as our second spectral code, it is much faster and easier to manage, giving us the ability to showcase some temperature heatmap results :

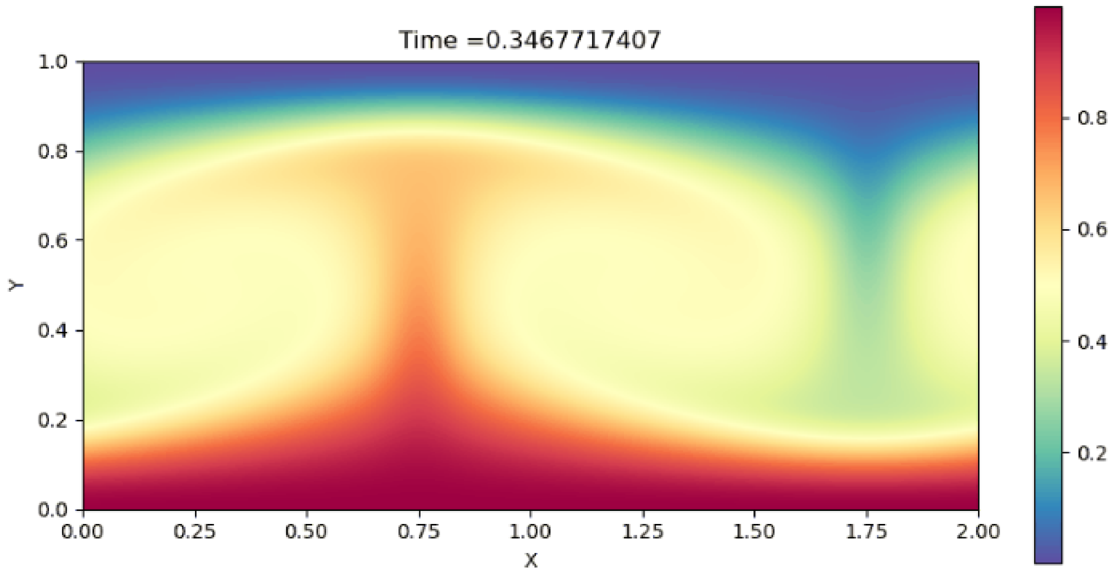


Figure 6.1: Third Program output for $Ra = 10^5$ and $Pr = 1$

We were able to quickly muster the simulations of a few Rayleigh numbers after fixing the Prandlt number at $Pr = 1$. We can observe as said in 5.2 that two aspects of the flow can be found depending on the value of Ra. One with a small $Ra = 10^2$ where no convection happens. However for a $Ra = 10^5$, we can observe symmetric patterns of thermal convection. The pattern is made of two "chimneys ", one ascending, carrying warm fluid as on the other hand a cold one descends (see figure 6.1 right above).

When it comes to the Fourth program, before doing any calculations, we decide to fix any values we can to only study the effects of Ta and N_ρ on thermal convection. Therefore, we fix the box aspect ratio to be of 2:1 and its resolution to be of 192 by 96. The Prandlt number is set to remain at 1 and we fix the latitude at 45° N ($Lat = \pi/4$ in Annex A.5). To see the convection patterns, two tests are made at $Ra = 10^2$ and $Ra = 10^4$, the resulting entropy heatmaps can be found in A.1 in Annex A.8) The kinetic energies are plotted over the viscous times in 6.2, we can clearly deduce the critical Rayleigh to be approximately around 600 which slightly aligns with theory, placing it at around 650.

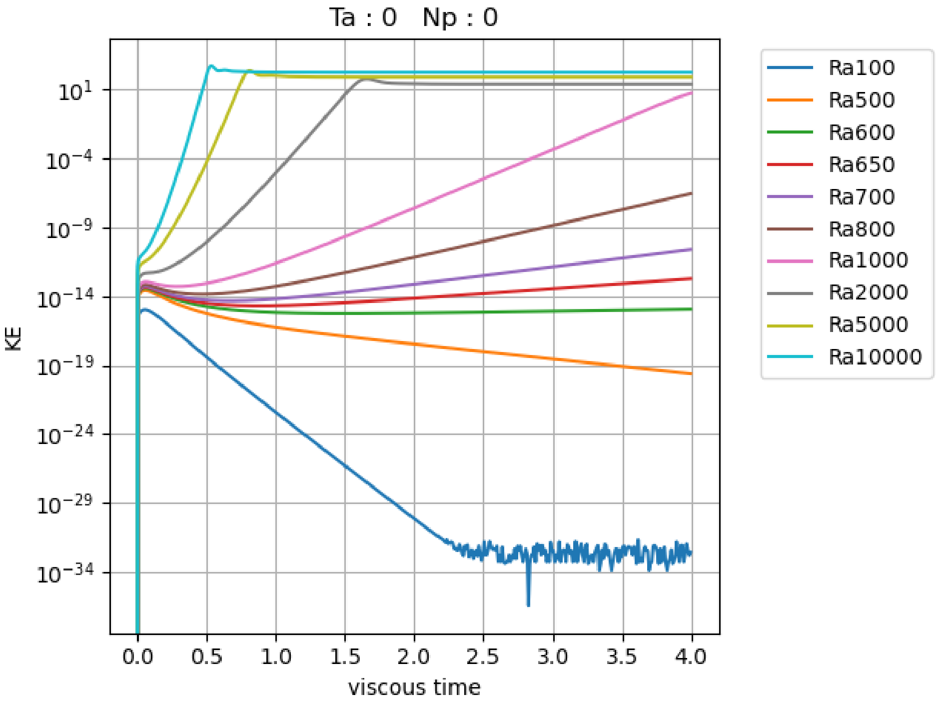


Figure 6.2: Kinetic energies over time for the Boussinesq case

6.2 The effects of rotation

the kinetic energies are plotted over the viscous times for two Taylors numbers of 10^3 and 10^4 in the figures 6.3 and 6.4.

We can see that for $Ta = 10^3$, we have a Ra_{crit} of around 1150 and of 4000 for $Ta = 10^3$, therefore, we can conclude that rotation tends to stabilize the fluid, making it harder to convect, explaining the need for a bigger Rayleigh number. The effects of rotation can also be observed in A.1 in Annex A.8 where we can see that its pattern is tilted at 45°. If we had enough time and could put Taylor into even higher numbers, we could have observed Taylor columns, however the computation of such a Taylor number would be impossible due to our scarce machine resources (a simple 2 GB RAM computer with a duo-core).

6.3 The effects of stratification

The case of stratification has been more laborious, our fourth program struggled immensely to produce any results whatsoever. However, we still managed to get a few values. Firstly the kinetic energies plotted over time that suggest a Ra_{crit} of around 300 (see 6.5).

The effects of stratification on the entropy heatmap aspects are far more visible however. As we can observe in A.1 in Annex A.8, the symmetry is broken as the high entropy region (red) wins over the low entropy region (blue). We could have made more conclusions if not for the slow pace at which stratification simulations were calculating.

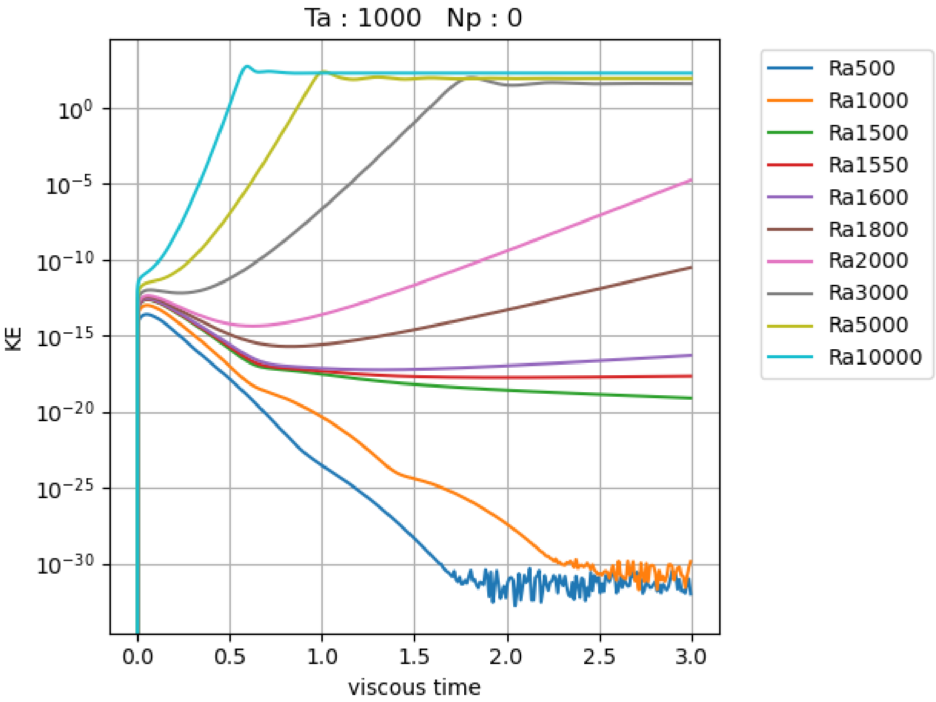


Figure 6.3: Kinetic energies over time for $Ta = 10^3$

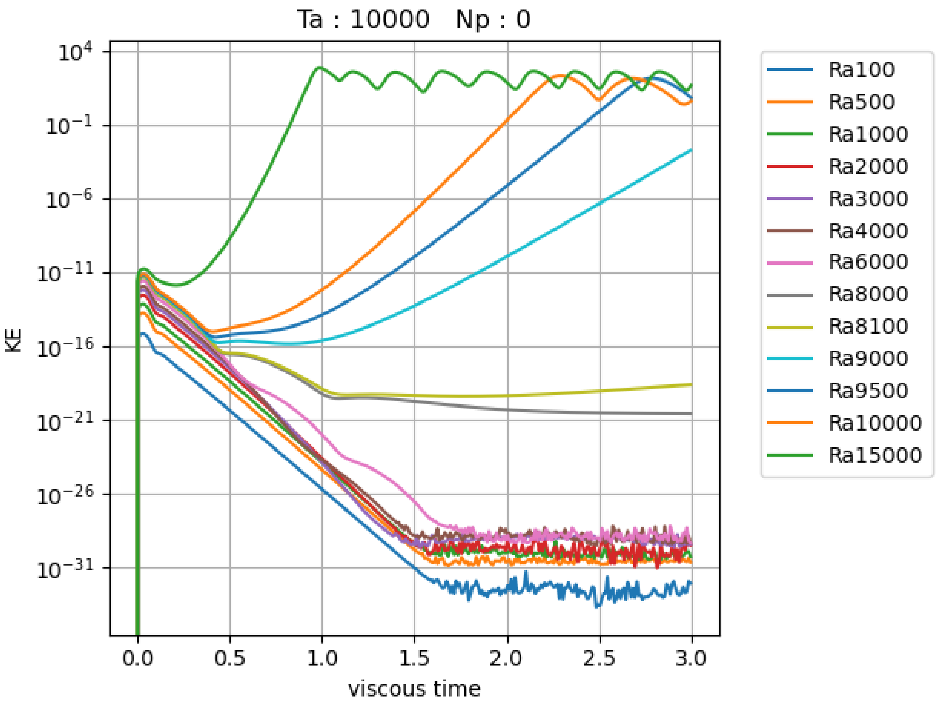


Figure 6.4: Kinetic energies over time for $Ta = 10^4$

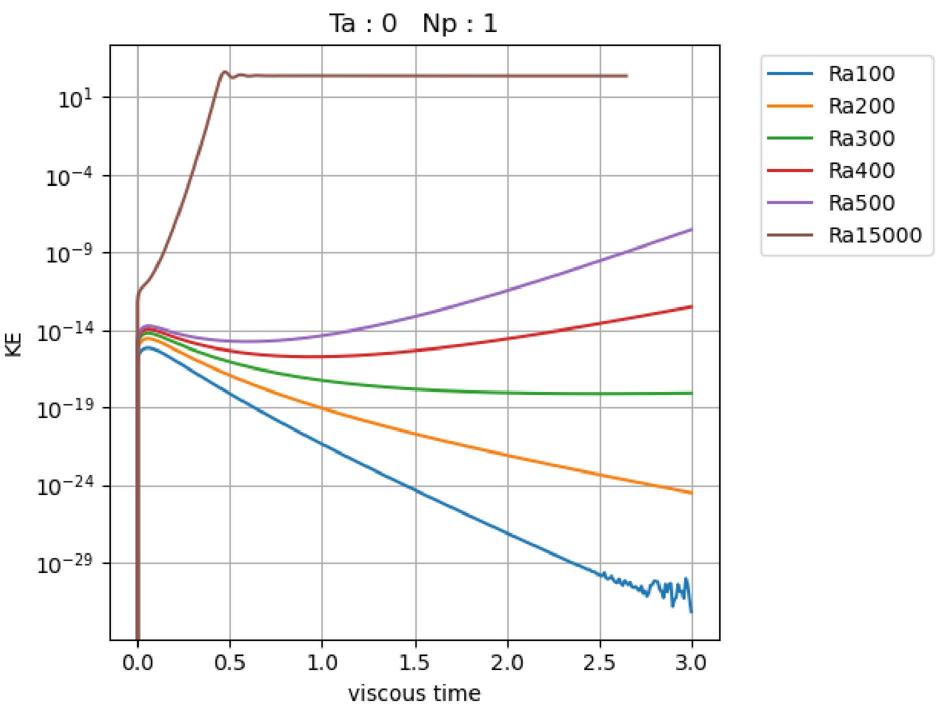


Figure 6.5: Kinetic energies over time for $N_\rho = 1$

Conclusion

The simulation of thermal convection through Navier-Stokes equations solving as been a real challenge for me because of the amount of code needed, calculations times and rigor. It has also been a challenge to pursue the completion of an internship abroad, in the United Kingdom in the middle of the Brexit process and the Covid pandemic. The biggest challenge was the mastering of the Dedalus open-source code, learning a whole new way of thinking about solving differential equations has been a delight and will surely help me in the pursuit of my career goals in Astrophysics and the space industry. The results I have produced have mostly been in accordance with the ones described by Professor Matthew Browning, my supervisor, who has been a great help throughout this internship. However, the lack of time and the fact I could not do that much calculations given my scarce computational resources gave me a hard time producing many results that could have helped me in making more conclusions and conjectures.

A few regrets still tarnish this great venture. the Covid pandemic cut short my time at the University of Exeter and I would have gladly spent time with the amazing Astrophysics group at the Physics Building. This has been a great opportunity that I would gladly redo if the opportunity shows up.

Acknowledgments

I would like to thank the Professor Matthew Browning for accepting me as his intern, the Astrophysics group for their warm welcome and their enthusiasm as well as the University of Exeter for their marvelous campus. I would also like to thank Professor Laurene Jouve at the University Paul Sabatier in Toulouse for helping me contact my internship supervisor and Professor Florence Pettinari-Sturmel for her help throughout the pandemic, making sure me and the other students were safe and sound.

References

[1] Laura K. Currie and Steven M. Tobias. *Generation of shear flows and vortices in rotating anelastic convection*. 2020.

[2] *Dedalus Project Documentation*. URL: `https://dedalus-project.readthedocs.io`.

[3] Gary A. Glatzmaier. *Introduction to Modeling Convection in Planets and Stars*. Princeton Series in Astrophysics. Princeton University Press, 2014.

[4] Lorena A. Barba group. *CFD Python : 12 steps to Navier-Stokes*. URL: `https://lorenabarba.com/blog/cfd-python-12-steps-to-navier-stokes/`.

[5] Simon R. W. Lance. *Compressible convection notes*. 2019.

[6] W. T. Lee. *Tridiagonal Matrices : Thomas Algorithm*.

[7] Shane Alexander McQuarrie. *Data Assimilation in the Boussinesq Approximation for Mantle Convection*. 2018.

[8] S.E. Norris. "A Parallel Navier Stokes Solver for Natural Convection and Free Surface Flow". 2000.

[9] *The Visual Room - Computational Fluid Dynamics Theory*. URL: `https://thevisualroom.com/01_barba_theory/barba_cfd_theory.html`.

Appendix

A.1 First Python program described in section 4.1

```python
# Program_1_DimlessBoussi_Regular.py

# IMPORTS --------------------------------------------------
# We first import the necessary libraries like mentionned.

import numpy as np
import matplotlib.pyplot as plt
import imageio
import time
import pickle

from functions_pickle import*

# CLASSES --------------------------------------------------
# We create a Python Object class modelizing the fluid and managing
# the calculations and storing of the resusts as a GIF montage.

class DimLess_Boussinesq_Box_2D():
    '''
    The DimLess_Boussinesq_Box_2D class uses the dimensionless Boussinesq equation to simulate thermal convection
    in a 2D box containing a fluid heated from below and cooled from above. The class uses a Marker and Cell
    staggered grid to calculate the derivatives from the finite difference method.
    The integration at each time step is done via an explicit Euler method.


    Fonctions :

    - __init__ : initializes the object
    - initialize_numbers : initializes the Prandtl and Rayleigh numbers
    - initialize_grid : initializes the staggered grid coordinates and parameters
    - initialize_fields : initializes the fields and gives them their initial values
    - Speeds_compute : calculates the next step speeds
    - Temp_compute : calculates the next step temperatures
    - B_calculation : calculates the RHS of the Poisson Pressure equations
    - Poisson_compute : calculates the pressure field by using a Poisson equation solver
    - dT_Calculation : calculates the next step time step to avoid program unstability
    - UV : calculates the speeds at the center of the MAC grid cells
    - RUN_Iterations : runs the simulation and stores the results in .pickle files
    - Post_processing : creates a GIF montage of the snapshots taken by the RUN_Iterations function

    '''
    def __init__(self):
```

```
        ########## Variables Fields
        self.U=None
        self.V=None
        self.P=None
        self.T=None

    def initialize_numbers(self,Prandtl_number=1,Rayleih_number=1800):
        self.Pr=Prandtl_number
        self.Ra=Rayleih_number

    def initialize_grid(self,grid_height=1,grid_width=2,nx=10,ny=20):
        # Grid Variables
        self.nx=nx
        self.ny=ny
        self.dx=grid_width/nx
        self.dy=grid_height/ny
        self.grid_height=grid_height
        self.grid_width=grid_width
        self.aspect_ratio=grid_width/grid_height
        # Grid Coordinates
        self.Cell_x=np.linspace(0+self.dx/2,grid_width-self.dx/2,num=nx)
        self.Cell_y=np.linspace(0+self.dy/2,grid_height-self.dy/2,num=ny)
        self.Cell_y=np.flip(self.Cell_y)
        self.Vertice_Vertical_x=np.linspace(0,grid_width,num=nx+1)
        self.Vertice_Vertical_y=np.linspace(0+self.dy/2,grid_height-self.dy/2,num=ny)
        self.Vertice_Vertical_y=np.flip(self.Vertice_Vertical_y)
        self.Vertice_Horizontal_x=np.linspace(0+self.dx/2,grid_width-self.dx/2,num=nx)
        self.Vertice_Horizontal_y=np.linspace(0,grid_height,num=ny+1)
        self.Vertice_Horizontal_y=np.flip(self.Vertice_Horizontal_y)
        self.Cell_X,self.Cell_Y=np.meshgrid(self.Cell_x,self.Cell_y)
        # Grid Meshgrids
        self.Vertice_Vertical_X,self.Vertice_Vertical_Y=np.meshgrid(self.Vertice_Vertical_x,self.Vertice_Vertical_y)
        self.Vertice_Horizontal_X,self.Vertice_Horizontal_Y=np.meshgrid(self.Vertice_Horizontal_x,self.Vertice_Horizontal_y)

    def initialize_fields(self,bottom_P=0,delta=0.05,template='closed_box'):
        # Speeds Initialization
        self.template=template
        self.U=delta*np.random.uniform(low=-1.0,high=1.0,size=self.Vertice_Vertical_X.shape)
        self.V=delta*np.random.uniform(low=-1.0,high=1.0,size=self.Vertice_Horizontal_X.shape)
        if self.template=='closed_box':
            self.U[0,:]=0
            self.U[:,0]=0
            self.U[:,-1]=0
            self.U[-1,:]=0
            self.V[0,:]=0
            self.V[-1,:]=0
            self.V[:,0]=0
            self.V[:,-1]=0
        #self.T=np.sin(np.pi*self.Cell_Y)
        self.T=np.zeros_like(self.Cell_Y)
        self.T[0,:]=0
        self.T[-1,:]=1
```

```python
        self.P=np.zeros_like(self.Cell_Y)

    def Speeds_compute(self,dt):
        if self.template=='closed_box':
            V=0.25*(self.V[1:-2,0:-1]+self.V[1:-2,1:]+self.V[2:-1,0:-1]+self.V[2:-1,1:])
            U=0.25*(self.U[0:-1,1:-2]+self.U[0:-1,2:-1]+self.U[1:,1:-2]+self.U[1:,2:-1])
            T_V=0.5*(self.T[0:-1,1:-1]+self.T[1:,1:-1])
            self.new_U=np.empty_like(self.U)
            self.new_V=np.empty_like(self.V)
            self.new_U[1:-1,1:-1]=self.U[1:-1,1:-1]+dt*(-self.U[1:-1,1:-1]*(self.U[1:-1,2:]-self.U[1:-1,0:-2])/(2*self.dx)-V*(self.U[0:-2,1:-1]-self.U[2:,1:-1])/(2*self.dy)-(self.P[1:-1,0:-1]-self.P[1:-1,1:])/(2*self.dx)+self.Pr*(self.U[1:-1,2:]+self.U[1:-1,0:-2]-2*self.U[1:-1,1:-1])/(self.dx**2)+self.Pr*(self.U[0:-2,1:-1]+self.U[2:,1:-1]-2*self.U[1:-1,1:-1])/(self.dy**2))
            self.new_V[1:-1,1:-1]=self.V[1:-1,1:-1]+dt*(-self.V[1:-1,1:-1]*(self.V[1:-1,2:]-self.V[1:-1,0:-2])/(2*self.dx)-U*(self.V[0:-2,1:-1]-self.V[2:,1:-1])/(2*self.dy)-(self.P[0:-1,1:-1]-self.P[1:,1:-1])/(2*self.dy)+self.Pr*self.Ra*T_V+self.Pr*(self.V[1:-1,2:]+self.V[1:-1,0:-2]-2*self.V[1:-1,1:-1])/(self.dx**2)+self.Pr*(self.V[0:-2,1:-1]+self.V[2:,1:-1]-2*self.V[1:-1,1:-1])/(self.dy**2))
            self.new_U[0,:]=0
            self.new_U[:,0]=0
            self.new_U[:,-1]=0
            self.new_U[-1,:]=0
            self.new_V[0,:]=0
            self.new_V[-1,:]=0
            self.new_V[:,0]=0
            self.new_V[:,-1]=0

    def Temp_compute(self,dt):
        if self.template=='closed_box':
            U=0.5*(self.U[1:-1,1:-2]+self.U[1:-1,2:-1])
            V=0.5*(self.V[1:-2,1:-1]+self.V[2:-1,1:-1])
            VL=0.5*(self.V[1:-2,0]+self.V[2:-1,0])
            VR=0.5*(self.V[1:-2,-1]+self.V[2:-1,-1])
            self.new_T=np.empty_like(self.T)
            self.new_T[1:-1,1:-1]=self.T[1:-1,1:-1]+dt*(-U*(self.T[1:-1,2:]-self.T[1:-1,0:-2])/(2*self.dx)-V*(self.T[0:-2,1:-1]-self.T[2:,1:-1])/(2*self.dy)+(self.T[1:-1,2:]+self.T[1:-1,0:-2]-2*self.T[1:-1,1:-1])/(self.dx**2)+(self.T[0:-2, 1:-1]+self.T[2:,1:-1]-2*self.T[1:-1,1:-1])/(self.dy**2))
            self.new_T[1:-1,0]=self.T[1:-1:,0]+dt*(-VL*(self.T[0:-2,0]-self.T[2:,0])/(2*self.dy)+(2*self.T[1:-1,1]-2*self.T[1:-1,0])/(self.dx**2)+(self.T[0:-2,0]+self.T[2:,0]-2*self.T[1:-1,0])/(self.dy**2))
            self.new_T[1:-1,-1]=self.T[1:-1,-1]+dt*(-VR*(self.T[0:-2,-1]-self.T[2:,-1])/(2*self.dy)+(2*self.T[1:-1,-2]-2*self.T[1:-1,-1])/(self.dx**2)+(self.T[0:-2,-1]+self.T[2:,-1]-2*self.T[1:-1,-1])/(self.dy**2))
            self.new_T[0,:]=0
            self.new_T[-1,:]=1

    def B_calculation(self,dt):
        if self.template=='closed_box':
            VL=0.5*(self.V[1:-2,0:-2]+self.V[2:-1,0:-2])
            VR=0.5*(self.V[1:-2,2:]+self.V[2:-1,2:])
            UT=0.5*(self.U[0:-2,1:-2]+self.U[0:-2,2:-1])
            UB=0.5*(self.U[2:,1:-2]+self.U[2:,2:-1])
            B=self.Ra*self.Pr*(self.T[0:-2,1:-1]-self.T[2:,1:-1])/(2*self.dy)
```

```
        B=B-(((self.U[1:-1,2:-1]-self.U[1:-1,1:-2])/self.dx)**2+((self.V[1:-2,1:-1]-self.V[2:-1,1:-1])/self.dy)**2+((VR-VL)/(2*self.dx))*((UT-UB)/(2*self.dy)))
        return B

    def Poisson_compute(self,nit,dt):
        if self.template=='closed_box':
            Pn=np.empty_like(self.P)
            Pn=self.P.copy()
            B=self.B_calculation(dt)
            for i in range(nit):
                Pn=self.P.copy()
                self.P[1:-1,1:-1]=(((Pn[1:-1,2:]+Pn[1:-1,0:-2])*self.dy**2+(Pn[2:,1:-1]+Pn[0:-2, 1:-1])*self.dx**2)/(2*(self.dx**2+self.dy**2))-self.dx**2*self.dy**2/(2*(self.dx**2+self.dy**2))*B)
                self.P[:,-1]=self.P[:,-2]
                self.P[0,:]=self.P[1,:]
                self.P[:,0]=self.P[:,1]
                self.P[-1,:]=self.P[-2,:]

    def dT_Calculation(self):
        max_u=np.amax(np.absolute(self.U))
        max_v=np.amax(np.absolute(self.V))
        if max_u>max_v:
            return self.safety_coeff*(self.dx**2/max_u)
        else:
            return self.safety_coeff*(self.dy**2/max_v)

    def UV(self):
        U=0.5*(self.U[:,0:-1]+self.U[:,1:])
        V=0.5*(self.V[0:-1,:]+self.V[1:,:])
        U=0.5*U/np.sqrt(U**2+V**2)
        V=0.5*V/np.sqrt(U**2+V**2)
        return U,V

    def RUN_Iterations(self,n_iterations=500,n_poisson=50,safety_coefficient=0.01):
        t=0
        dt=0
        self.safety_coeff=safety_coefficient
        # Clearing the pickle files
        clear_file('temperatures.pkl')
        clear_file('times.pkl')
        # Saving the initial Fourier arrays
        save_unique_array('temperatures.pkl',self.T)
        save_unique_array('times.pkl',t)
        calc_time0=time.time()
        percent_5=int(n_iterations*5/100)
        print("######### BEGINNING CALCULATIONS #########")
        for i in range(n_iterations):
            if i%percent_5==0:
                printProgressBar(i+1,n_iterations,prefix='Progress:',suffix='Complete',length=20,time0=calc_time0)
            t=t+dt
            dt=self.dT_Calculation()
            self.Poisson_compute(n_poisson,dt)
            self.Temp_compute(dt)
            self.Speeds_compute(dt)
```

```
            self.U=self.new_U
            self.V=self.new_V
            self.T=self.new_T
            save_unique_array('temperatures.pkl',self.T)
            save_unique_array('times.pkl',t)
        printProgressBar(n_iterations,n_iterations,prefix='Progress:',suffix='Complete',length=20,time0=calc_time0)
        print("######### CALCULATIONS FINISHED #########")
        print(" Final calculations duration = ",round(time.time()-calc_time0,4)," seconds")

    def Post_processing(self,plot=False,save=True,skip=1,gif_fps=25):
        Temps=load_files('temperatures.pkl',frequency=skip)
        Ts=load_files('times.pkl',frequency=skip)
        n=len(Temps)
        print("######### BEGINNING POST-PROCESSING #########")
        Tmax=np.amax(Temps[0])
        Tmin=np.amin(Temps[0])
        fig,ax=plt.subplots(figsize=(10,5))
        heatmap=ax.imshow(Temps[0],cmap='Spectral_r',extent=[0,self.grid_width,0,self.grid_height],vmin=Tmin,vmax=Tmax)
        ax.set(xlabel='X',ylabel='Y',title='Time ='+str(round(Ts[0],10)))
        plt.colorbar(heatmap)
        if plot==True:
            fig.show()
            plt.pause(5)
        if save==True:
            Images=[]
        for i in range(1,n):
            heatmap.set_data(Temps[i])
            ax.set(xlabel='X',ylabel='Y',title='Time ='+str(round(Ts[i],10)))
            if save==True and plot==True:
                plt.pause(0.01)
                plt.draw()
                image=np.frombuffer(fig.canvas.tostring_rgb(), dtype='uint8')
                image=image.reshape(fig.canvas.get_width_height()[::-1]+(3,))
                Images.append(image)
            if plot==True and save==False:
                plt.pause(0.01)
                plt.draw()
            if plot==False and save==True:
                fig.canvas.draw()
                image=np.frombuffer(fig.canvas.tostring_rgb(), dtype='uint8')
                image=image.reshape(fig.canvas.get_width_height()[::-1]+(3,))
                Images.append(image)
        if save==True:
            print("######### GIF CREATION #########")
            imageio.mimsave('sim.gif',Images,fps=gif_fps)
            print("######### GIF FINISHED #########")
```

A.2 Second Python program described in section 4.2

```
# Program_2_DimlessBoussi_Spectral.py

# IMPORTS ------------------------------------------------
# We first import the necessary libraries like mentionned.
```

```python

import numpy as np
import matplotlib.pyplot as plt
import imageio
import time
import pickle

from functions_pickle import*

# CLASSES ---------------------------------------------------
# We create a Python Object class modelizing the fluid and managing
# the calculations and storing of the resusts as a GIF montage.

class Spectral_DimLess_Boussinesq_Box_2D():
    '''
    The DimLess_Boussinesq_Box_2D class uses the dimensionless Boussinesq equation to simulate thermal convection
    in a 2D box containing a fluid heated from below and cooled from above using a pseudo spectral
    Fourier expansion method.
    The integration at each time step is done via an explicit Euler method.


    Fonctions :

    - __init__ : initializes the object
    - initialize_numbers : initializes the Prandtl and Rayleigh numbers
    - initialize_fields : initializes the grid, the fields and gives them their initial values
    - Curl_convective_term : calculates the Curl equation convective term
    - Temp_convective_term : calculates the Temperature equation convective term
    - Thomas_solver : solves the tridiagonal matrix from the Poisson equation
    - Curl_compute : calculates the next step curls
    - Temp_compute : calculates the RHS of the Poisson Pressure equations
    - Poisson_compute : calculates the pressure field by using a Poisson equation solver
    - Physical_space_calculation : allows us to calculate the spatial fields from the Fourier amplitudes
    - Time_step : calculates the next step time step to avoid program unstability
    - Velocity_calculation : calculates the speeds
    - RUN_Iterations : runs the simulation and stores the results in .pickle files
    - Post_processing : creates a GIF montage of the snapshots taken by the RUN_Iterations function

    '''
    def __init__(self):
        #Variables Fields
        self.f_Psi=None
        self.f_Omega=None
        self.f_Temp=None

    def initialize_numbers(self,Prandtl_number=1,Rayleih_number=1800):
        self.Pr=Prandtl_number
        self.Ra=Rayleih_number

    def initialize_fields(self,grid_width=1,grid_height=1,nx=100,nz=100, Fourier_limit=30):
```

```python
        self.grid_width=grid_width
        self.grid_height=grid_height
        self.nx=nx
        self.nz=nz
        self.dx=grid_width/nx
        self.dz=grid_height/nz
        self.aspect_ratio=grid_width/grid_height
        self.x=np.linspace(0,grid_width,num=nx)
        self.z=np.linspace(0,grid_height,num=nz)
        self.z=np.flip(self.z)
        self.grid_X,self.grid_Z=np.meshgrid(self.x,self.z)
        self.Nn=Fourier_limit
        # Creation of the Fourier indices matrix
        self.n=np.linspace(0,self.Nn,num=self.Nn+1)
        self.f_N,self.fourier_Z=np.meshgrid(self.n,self.z)
        # Creation of the Fourier Temperatures Amplitudes
        T=np.sin(np.pi*self.z)
        self.f_Temp=np.zeros_like(self.f_N)
        for i in range(nz):
            self.f_Temp[i][0]=T[i]
        self.f_Temp[0][0]=0
        self.f_Temp[-1][0]=1
        # Creation of the Curl and Stream Function Fourier Amplitudes
        self.f_Curl=np.zeros_like(self.f_N)
        self.f_Psi=np.zeros_like(self.f_N)
        # Creation of the Poisson Tridiagonal Matrix
        self.Poisson_Sup=[]
        self.Poisson_Dia=[]
        self.Poisson_Sub=[]
        for i in range(self.Nn+1):
            sup=np.full(self.nz-1,-1/(self.dz**2))
            sub=np.full(self.nz-1,-1/(self.dz**2))
            dia=np.full(self.nz,(i*np.pi/self.aspect_ratio)**2+2/(self.dz**2))
            # adjustments to the limit values
            sup[0]=0
            dia[0]=1
            sub[-1]=0
            dia[-1]=1                # adding them to the lists
            self.Poisson_Sup.append(sup)
            self.Poisson_Dia.append(dia)
            self.Poisson_Sub.append(sub)
        assert len(self.Poisson_Sup)==self.Nn+1
        assert len(self.Poisson_Sub)==self.Nn+1
        assert len(self.Poisson_Dia)==self.Nn+1

    def Curl_convective_term(self,n):
        C=np.zeros_like(self.f_Curl[1:-1,0])
        for n_p in range(1,self.Nn+1):
            for n_pp in range(1,self.Nn+1):
                C=C+((-n_p*self.f_Curl[1:-1,n_p]*(self.f_Psi[0:-2,n_pp]-self.f_Psi[2:,n_pp])/(2*self.dz)+n_pp*self.f_Psi[1:-1,n_pp]*(self.f_Curl[0:-2,n_p]-self.f_Curl[2:,n_p])/(2*self.dz))*Kronecker(n_pp+n_p,n)-(-n_p*self.f_Curl[1:-1,n_p]*(self.f_Psi[0:-2,n_pp]-self.f_Psi[2:,n_pp])/(2*self.dz)+n_pp*self.f_Psi[1:-1,n_pp]*(self.f_Curl[0:-2,n_p]-self.f_Curl[2:,n_p])/(2*self.dz))*(Kronecker(n_pp-n_p,n)-Kronecker(n_p-n_pp,n)))
        return C*(-np.pi/(2*self.aspect_ratio))

    def Temp_convective_term(self,n):
```

```python
        C=np.zeros_like(self.f_Temp[1:-1,0])
        C=-(n*np.pi/self.aspect_ratio)*self.f_Psi[1:-1,n]*(self.f_Temp[0:-2,0]-self.f_Temp[2:,0])/(2*self.dz)
        S=np.zeros_like(self.f_Temp[1:-1,0])
        if n==0:
            for n_p in range(1,self.Nn+1):
                for n_pp in range(1,self.Nn+1):
                    S=S+((-n_p*self.f_Curl[1:-1,n_p]*(self.f_Psi[0:-2,n_pp]-self.f_Psi[2:,n_pp])/(2*self.dz)+n_pp*self.f_Psi[1:-1,n_pp]*(self.f_Curl[0:-2,n_p]-self.f_Curl[2:,n_p])/(2*self.dz))*Kronecker(n_pp+n_p,0)-(-n_p*self.f_Curl[1:-1,n_p]*(self.f_Psi[0:-2,n_pp]-self.f_Psi[2:,n_pp])/(2*self.dz)+n_pp*self.f_Psi[1:-1,n_pp]*(self.f_Curl[0:-2,n_p]-self.f_Curl[2:,n_p])/(2*self.dz))*(Kronecker(n_pp-n_p,0)))
            S=S*(-np.pi/(2*self.aspect_ratio))
            return C+S

        else:
            for n_p in range(1,self.Nn+1):
                for n_pp in range(1,self.Nn+1):
                    S=S+((-n_p*self.f_Temp[1:-1,n_p]*(self.f_Psi[0:-2,n_pp]-self.f_Psi[2:,n_pp])/(2*self.dz)+n_pp*self.f_Psi[1:-1,n_pp]*(self.f_Temp[0:-2,n_p]-self.f_Temp[2:,n_p])/(2*self.dz))*Kronecker(n_pp+n_p,n)-(-n_p*self.f_Temp[1:-1,n_p]*(self.f_Psi[0:-2,n_pp]-self.f_Psi[2:,n_pp])/(2*self.dz)+n_pp*self.f_Psi[1:-1,n_pp]*(self.f_Temp[0:-2,n_p]-self.f_Temp[2:,n_p])/(2*self.dz))*(Kronecker(n_pp-n_p,n)+Kronecker(n_p-n_pp,n)))
            S=S*(-np.pi/(2*self.aspect_ratio))
            return C+S

    def Thomas_solver(self,a,b,c,d):
        n=len(d)
        ac,bc,cc,dc=map(np.array,(a,b,c,d))
        for i in range(1, n):
            mc=ac[i-1]/bc[i-1]
            bc[i]=bc[i]-mc*cc[i-1]
            dc[i]=dc[i]-mc*dc[i-1]
        xc=bc
        xc[-1]=dc[-1]/bc[-1]
        for i in range(n-2,-1,-1):
            xc[i]=(dc[i]-cc[i]*xc[i+1])/bc[i]
        return xc

    def Curl_compute(self,dt):
        self.new_f_Curl=np.empty_like(self.f_Curl)
        for i in range(1,self.Nn+1):
            C=self.Curl_convective_term(i)
            self.new_f_Curl[1:-1,i]=self.f_Curl[1:-1,i]+dt*(C+self.Ra*self.Pr*(i*np.pi/self.aspect_ratio)*self.f_Temp[1:-1,i]+self.Pr*(((self.f_Curl[0:-2,i]+self.f_Curl[2:,i]-2*self.f_Curl[1:-1,i])/(self.dz**2))-(i*np.pi/self.aspect_ratio)**2*self.f_Curl[1:-1,i]))
        self.new_f_Curl[0,:]=0
        self.new_f_Curl[-1,:]=0
        self.new_f_Curl[:,0]=0

    def Temp_compute(self,dt):
        self.new_f_Temp=np.empty_like(self.f_Curl)
        for i in range(self.Nn+1):
            C=self.Temp_convective_term(i)
            self.new_f_Temp[1:-1,i]=self.f_Temp[1:-1,i]+dt*(C+(((self.f_Temp
```

```
[0:-2,i]+self.f_Temp[2:,i]-2*self.f_Temp[1:-1,i])/(self.dz**2))-(i*np.pi/self.
aspect_ratio)**2*self.f_Temp[1:-1,i]))
        self.new_f_Temp[0,:]=0
        self.new_f_Temp[-1,1:]=0
        self.new_f_Temp[-1][0]=1

    def Poisson_compute(self):
        self.new_f_Psi=np.empty_like(self.f_Psi)
        for i in range(1,self.Nn+1):
            Curl=self.f_Curl[1:-1,i]
            Curl=Curl.T
            Sup=self.Poisson_Sup[i]
            Dia=self.Poisson_Dia[i]
            Sub=self.Poisson_Sub[i]
            Sol=self.Thomas_solver(Sub,Dia,Sup,Curl)
            self.new_f_Psi[:,i]=Sol
        self.new_f_Psi[0,:]=0
        self.new_f_Psi[-1,:]=0
        self.new_f_Psi[:,0]=0

    def Physical_space_calculation(self,variable='temp'):
        if variable=='temp':
            self.Temp=np.zeros_like(self.grid_X)
            for i in range(self.Nn+1):
                for j in range(self.nz):
                    self.Temp[j,:]=self.Temp[j,:]+self.f_Temp[j][i]*np.cos(i*np.pi*self.grid_X[j,:]/self.aspect_ratio)
        if variable=='psi':
            self.Psi=np.zeros_like(self.grid_X)
            for i in range(self.Nn+1):
                for j in range(self.nz):
                    self.Psi[j,:]=self.Psi[j,:]+self.f_Psi[j][i]*np.sin(i*np.pi*self.grid_X[j,:]/self.aspect_ratio)
        if variable=='curl':
            self.Curl=np.zeros_like(self.grid_X)
            for i in range(self.Nn+1):
                for j in range(self.nz):
                    self.Curl[j,:]=self.Curl[j,:]+self.f_Curl[j][i]*np.sin(i*np.pi*self.grid_X[j,:]/self.aspect_ratio)

    def Velocity_calculation(self):
        U=np.zeros_like(self.Psi)
        V=np.zeros_like(self.Psi)
        U[1:-1,1:-1]=-(self.Psi[0:-2,1:-1]-self.Psi[2:,1:-1])/(2*self.dz)
        V[1:-1,1:-1]=(self.Psi[1:-1,2:]-self.Psi[1:-1,0:-2])/(2*self.dx)
        return U,V

    def Time_step(self,V):
        v=np.amax(np.absolute(V))
        if self.Pr<1:
            if v!=0:
                dt1=(self.dz**2)/4
                dt2=(self.dz**2)/(4*v)
                return self.safety_coeff*min(dt1,dt2)
            else:
                return self.safety_coeff*(self.dz**2)/4
        elif self.Pr>=1:
            if v!=0:
```

```
                dt1=(self.dz**2)/4*self.Pr
                dt2=(self.dz**2)/(4*v)
                return self.safety_coeff*min(dt1,dt2)
            else:
                return self.safety_coeff*(self.dz**2)/4*self.Pr

    def RUN_Iterations(self,n_iterations=500,safety_coefficient=0.01):
        t=0
        dt=0
        self.safety_coeff=safety_coefficient
        # Clearing the pickle files
        clear_file('temperatures.pkl')
        clear_file('streams.pkl')
        clear_file('times.pkl')
        # Saving the initial Fourier arrays
        save_unique_array('temperatures.pkl',self.f_Temp)
        save_unique_array('streams.pkl',self.f_Psi)
        save_unique_array('times.pkl',t)
        # Calculation of the space initial arrays
        self.Physical_space_calculation(variable='psi')
        # Calculation of the initial velocities
        U,V=self.Velocity_calculation()
        calc_time0=time.time()
        percent_5=int(n_iterations*5/100)
        print("######### BEGINNING CALCULATIONS #########")
        for i in range(n_iterations):
            if i%percent_5==0:
                printProgressBar(i+1,n_iterations,prefix='Progress:',suffix='
    Complete',length=20,time0=calc_time0)
            dt=self.Time_step(V)
            t=t+dt
            self.Temp_compute(dt)
            self.Curl_compute(dt)
            self.Poisson_compute()
            # Updating the spectral arrays
            self.f_Psi=self.new_f_Psi
            self.f_Curl=self.new_f_Curl
            self.f_Temp=self.new_f_Temp
            # Extracting the Velocity field
            self.Physical_space_calculation(variable='psi')
            U,V=self.Velocity_calculation()
            save_unique_array('temperatures.pkl',self.f_Temp)
            save_unique_array('streams.pkl',self.f_Psi)
            save_unique_array('times.pkl',t)
        print("######### CALCULATIONS FINISHED #########")

    def Post_processing(self,plot=False,save=True,skip=1,gif_fps=25):
        f_Temps=load_files('temperatures.pkl',frequency=skip)
        #f_Psis=load_files('streams.pkl',frequency=skip)
        Ts=load_files('times.pkl',frequency=skip)
        n=len(f_Temps)
        Temps=[]
        Psis=[]
        print("######### BEGINNING POST-PROCESSING #########")
        for i in range(n):
            self.f_Temp=f_Temps[i]
            self.Physical_space_calculation(variable='temp')
            Temps.append(self.Temp)
```

```
            #self.f_Psi=f_Psi[i]
            #self.Physical_space_calculation(variable='psi')
            #Psis.append(self.Psi)
        print("######### POST-PROCESSING FINISHED #########")
        Tmax=np.amax(Temps[0])
        Tmin=np.amin(Temps[0])
        fig,ax=plt.subplots(figsize=(10,5))
        heatmap=ax.imshow(Temps[0],cmap='Spectral_r',extent=[0,self.grid_width,0,self.grid_height],vmin=Tmin,vmax=Tmax)
        #arrows=ax.quiver(self.grid_X[1::15,1::15],self.grid_Z[1::15,1::15],U[1::15,1::15],V[1::15,1::15])
        ax.set(xlabel='X',ylabel='Y',title='Time ='+str(round(Ts[0],10)))
        plt.colorbar(heatmap)
        if plot==True:
            fig.show()
            plt.pause(5)
        if save==True:
            Images=[]
        for i in range(1,n):
            heatmap.set_data(Temps[i])
            ax.set(xlabel='X',ylabel='Y',title='Time ='+str(round(Ts[i],10)))
            if save==True and plot==True:
                plt.pause(0.01)
                plt.draw()
                image=np.frombuffer(fig.canvas.tostring_rgb(), dtype='uint8')
                image=image.reshape(fig.canvas.get_width_height()[::-1]+(3,))
                Images.append(image)
            if plot==True and save==False:
                plt.pause(0.01)
                plt.draw()
            if plot==False and save==True:
                fig.canvas.draw()
                image=np.frombuffer(fig.canvas.tostring_rgb(), dtype='uint8')
                image=image.reshape(fig.canvas.get_width_height()[::-1]+(3,))
                Images.append(image)
        if save==True:
            print("######### GIF CREATION #########")
            imageio.mimsave('sim.gif',Images,fps=gif_fps)
            print("######### GIF FINISHED #########")
```

A.3 Python File containing the functions used in A.1 and A.2

```
#functions_pickle.py

# IMPORTS --------------------------------------------------
# We first import the necessary libraries like mentionned.

import numpy as np
import time
import pickle

# FUNCTIONS -------------------------------------------

def printProgressBar(iteration,total,prefix='',suffix='',decimals=1,length=100,
```

```
    fill='#',time0=0):
    ''' Allows us to print a progress bar for the simulation '''
    percent=("{0:."+str(decimals)+"f}").format(100*(iteration/float(total)))
    filledLength=int(length*iteration // total)
    bar=fill*filledLength+'-'*(length - filledLength)
    print('\r%s |%s| %s%% %s' % (prefix,bar,percent,suffix)+" "+str(time.time()-time0)+" "+str(iteration))
    # Print New Line on Complete
    if iteration==total:
        print()

def Kronecker(i,j):
    ''' Replaces the Kronecker parameter '''
    if i==j:
        return 1
    else:
        return 0

def save_unique_array(file_name,data):
    ''' Allows us to save an array or value in a .pickle file '''
    file=open(file_name,'ab')
    pickle.dump(data,file,pickle.HIGHEST_PROTOCOL)
    file.close()

def load_files(file_name,frequency=2):
    ''' Allows us to load every array or value in a .pickle file '''
    data=[]
    file=open(file_name,'rb')
    i=0
    while True:
        try:
            if i==0 or i%frequency==0:
                data.append(pickle.load(file))
            else:
                trash=pickle.load(file)
            i=i+1
        except EOFError:
            break
    return data

def clear_file(file_name):
    ''' Allows us to clear every array or value in a .pickle file '''
    file=open(file_name,'wb')
    file.close()
```

A.4 Third Python program mentioned in section 5.1

```
# Program_3_DimlessBoussi_Dedalus.py

# IMPORTS ------------------------------------------------
# We first import the necessary libraries like mentionned.

import numpy as np
import matplotlib.pyplot as plt
from dedalus import public as de
from dedalus.extras import flow_tools
```

```
import time
import imageio

from functions_txt import*

# CLASSES --------------------------------------------------
# We create a Python Object class modelizing the fluid and managing
# the calculations and storing of the resusts as a GIF montage.

class Dedalus_Boussinesq():
    '''
    The Dedalus_Boussinesq class uses the Dedalus open source to simulate thermal
     convection in
    a 2Dbox containing a fluid cooled from above and heated from below. It uses
   the curl and
    the stream function as well as the temperature as its variables.


    Fonctions :

    - __init__ : initializes the object
    - problem_setup : initializes the problem parameters, equatiosn and boundary
   equations
    - RUN : runs the simulation
    - post_processing : creates a GIF montage of the snapshots taken by the
   RUN_Iterations function

    '''
    def __init__(self):
        self.domain=None
        self.solver=None
        self.problem=None

    def problem_setup(self,L=2.,nx=192,nz=96,Prandtl_number=1.,Rayleih_number=1e4
   ,bc_type='no_slip'):
        self.L=float(L)
        self.nx=int(nx)
        self.nz=int(nz)
        x_basis = de.Fourier('x', int(nx), interval=(0, L), dealias=3/2)
        z_basis = de.Chebyshev('z',int(nz), interval=(0, 1), dealias=3/2)
        self.saving_shape=(int(nx*3/2),int(nz*3/2))
        self.domain = de.Domain([x_basis, z_basis], grid_dtype=np.float64)
        self.problem = de.IVP(self.domain, variables=['T','Tz','psi','psiz','curl
   ','curlz'])
        self.problem.parameters['L'] = L
        self.problem.parameters['nx'] = nx
        self.problem.parameters['nz'] = nz
        self.Pr=float(Prandtl_number)
        self.Ra=float(Rayleih_number)
        self.problem.parameters['Ra'] = self.Ra
        self.problem.parameters['Pr'] = self.Pr
        # Stream function relation to the speed
        self.problem.substitutions['u'] = "-dz(psi)"
        self.problem.substitutions['v'] = "dx(psi)"
        # Derivatives values relation to the main values
        self.problem.add_equation("psiz - dz(psi) = 0")
        self.problem.add_equation("curlz - dz(curl) = 0")
        self.problem.add_equation("Tz - dz(T) = 0")
```

```python
        self.problem.add_equation("curl + dx(dx(psi)) + dz(psiz) = 0")
        self.problem.add_equation("dt(curl)+Ra*Pr*dx(T)-Pr*(dx(dx(curl))+dz(curlz))=-(u*dx(curl)+v*curlz)")
        self.problem.add_equation("dt(T)-dx(dx(T))-dz(Tz)=-(u*dx(T)+v*Tz)")
        self.problem.add_bc("left(T) = 1")
        self.problem.add_bc("right(T) = 0")
        self.problem.add_bc("left(psi) = 0")
        self.problem.add_bc("right(psi) = 0")
        if bc_type not in ['no_slip','free_slip']:
            raise ValueError("Boundary Conditions must be 'no_slip' or 'free_slip'")
        else:
            if bc_type=='no_slip':
                self.problem.add_bc("left(psiz) = 0")
                self.problem.add_bc("right(psiz) = 0")
            if bc_type=='free_slip':
                self.problem.add_bc("left(dz(psiz)) = 0")
                self.problem.add_bc("right(dz(psiz)) = 0")

    def RUN(self,scheme=de.timesteppers.RK443,adding=False,sim_time=2,wall_time=np.inf,tight=False,save=20):
        self.solver = self.problem.build_solver(scheme)
        if adding:
            t=load_last_value('times.txt')
            temp=load_last_array('temperatures.txt',shape=self.saving_shape)
            Variables=load_arrays('variables.txt',frequency=1,shape=self.saving_shape)
            T=self.solver.state['T']
            Psi=self.solver.state['psi']
            Curl=self.solver.state['curl']
            T['g']=temp
            T.differentiate('z', out=self.solver.state['Tz'])
            Psi['g']=Variables[0]
            Psi.differentiate('z', out=self.solver.state['psiz'])
            Curl['g']=Variables[1]
            Curl.differentiate('z', out=self.solver.state['curlz'])
        else:
            t=0
            print("Clearing old data ...")
            clear_file('temperatures.txt')
            clear_file('times.txt')
            clear_file('variables.txt')
            # Initial conditions ----------------------------------------------------------
            print("Initializing Values ...")
            eps = 1e-4
            k = 3.117
            x,z = self.problem.domain.grids(scales=1)
            T=self.solver.state['T']
            T['g']=1-z+eps*np.sin(k*x)*np.sin(2*np.pi*z)
            T.differentiate('z', out=self.solver.state['Tz'])
        # Stopping Parameters ---------------------------------------------------------
        self.solver.stop_sim_time = sim_time # Length of simulation.
        self.solver.stop_wall_time = wall_time # Real time allowed to compute.
        self.solver.stop_iteration = np.inf # Maximum iterations allowed.
        # Control Flow ----------------------------------------------------------------
```

```
        dt = 1e-4
        if tight:
            cfl = flow_tools.CFL(self.solver,initial_dt=dt,cadence=1,
                                 safety=1,max_change=1.5,
                                 min_change=0.01,max_dt=0.01,
                                 min_dt=1e-10)
        else:
            cfl = flow_tools.CFL(self.solver, initial_dt=dt, cadence=10,
                                 safety=1,max_change=1.5,
                                 min_change=0.5,max_dt=0.01,
                                 min_dt=1e-6)
        cfl.add_velocities(('u', 'v'))
        # Flow properties (print during run; not recorded in the records files)
        flow = flow_tools.GlobalFlowProperty(self.solver, cadence=1)
        flow.add_property("sqrt(u **2 + v **2) / Ra", name='Re' )
        # MAIN COMPUTATION LOOP -------------------------------------------------------
        try:
            print("####### BEGINNING CALCULATIONS #######")
            print("Starting main loop")
            start_time = time.time()
            while self.solver.ok:
                # Recompute time step and iterate.
                dt = self.solver.step(cfl.compute_dt())
                t=t+dt
                if self.solver.iteration % 10 == 0:
                    info = "Iteration {:>5d}, Time: {:.7f}, dt: {:.2e}".format(self.solver.iteration, self.solver.sim_time, dt)
                    Re  = flow.max("Re")
                    info += ", Max Re = {:f}".format(Re)
                    print(info)
                    if np.isnan(Re):
                        raise ValueError("Reynolds number went to infinity!!"
                                         "\nRe = {}".format(Re))
                if save:
                    if self.solver.iteration % save == 0:
                        T=self.solver.state['T']
                        append_unique_array('temperatures.txt',T['g'])
                        append_unique_value('times.txt',t)
        except BaseException as e:
            print("Exception raised, triggering end of main loop.")
            raise
        finally:
            print("####### CALCULATIONS FINISHED #######")
            total_time = time.time() - start_time
            print("Iterations: {:d}".format(self.solver.iteration))
            print("Sim end time: {:.3e}".format(self.solver.sim_time))
            print("Run time: {:.3e} sec".format(total_time))
            print("END OF SIMULATION\n")
            T=self.solver.state['T']
            Psi=self.solver.state['psi']
            Curl=self.solver.state['curl']
            append_unique_array('temperatures.txt',T['g'])
            append_unique_value('times.txt',t)
            append_unique_array('variables.txt',Psi['g'])
            append_unique_array('variables.txt',Curl['g'])
```

```python
    def post_processing(self,plot=False,save=True,skip=1,gif_fps=25):
        Temps=load_arrays('temperatures.txt',frequency=skip,shape=self.
    saving_shape)
        Ts=load_values('times.txt',frequency=skip)
        n=len(Temps)
        print("####### BEGINNING POST-PROCESSING #######")
        Tmax=np.amax(Temps[0])
        Tmin=np.amin(Temps[0])
        fig,ax=plt.subplots(figsize=(10,5))
        heatmap=ax.imshow(np.flip(Temps[0].T,0),cmap='Spectral_r',extent=[0,self.
    L,0,1.],vmin=Tmin,vmax=Tmax)
        ax.set(xlabel='X',ylabel='Y',title='Time ='+str(round(Ts[0],10)))
        plt.colorbar(heatmap)
        if plot==True:
            fig.show()
            plt.pause(3)
        if save==True:
            Images=[]
        for i in range(1,n):
            heatmap.set_data(np.flip(Temps[i].T,0))
            ax.set(xlabel='X',ylabel='Y',title='Time ='+str(round(Ts[i],10)))
            if save==True and plot==True:
                plt.pause(0.01)
                plt.draw()
                image=np.frombuffer(fig.canvas.tostring_rgb(), dtype='uint8')
                image=image.reshape(fig.canvas.get_width_height()[::-1]+(3,))
                Images.append(image)
            if plot==True and save==False:
                plt.pause(0.01)
                plt.draw()
            if plot==False and save==True:
                fig.canvas.draw()
                image=np.frombuffer(fig.canvas.tostring_rgb(), dtype='uint8')
                image=image.reshape(fig.canvas.get_width_height()[::-1]+(3,))
                Images.append(image)
        if save==True:
            print("####### GIF CREATION #######")
            imageio.mimsave('sim.gif',Images,fps=gif_fps)
            print("####### GIF FINISHED #######")
```

A.5 Fourth Python program used to get the main results in sections 6.2 and 6.3

```python
# Program_4_DimlessAnelastic_Dedalus.py

# IMPORTS -------------------------------------------------
# We first import the necessary libraries like mentionned.

import numpy as np
from mpi4py import MPI
import time
import matplotlib.pyplot as plt
import sys
import imageio
import os
```

```python

import h5py

from dedalus import public as de
from dedalus.extras import flow_tools
import pathlib

from functions_txt import*

import logging
logger = logging.getLogger(__name__)

# FUNCTIONS -------------------------------------------
#################### ENTROPY AND KE STUDY ##############################

def Dedalus_Inelastic_S(Ra,Pr,Np,Ta,end_time,snaps=True):
    ''' Builds a problem with its equations, parameters and boundary conditions and solves it, saving values in a h5py '''
    m=1.5
    theta = 1-np.exp(-Np/m)
    Ly, Lz = 2,1
    Ny, Nz = 192,96
    Lat=np.pi/4
    initial_timestep = 1.5e-4           # Initial timestep
    snapshot_skip=10
    analysis_freq = 1.5e-3              # Frequency analysis files are outputted
    end_sim_time = end_time                  # Stop time in simulations units
    end_wall_time = np.inf              # Stop time in wall time
    end_iterations = np.inf             # Stop time in iterations
    max_dt=0.005
    save_direc = "raw_data/"
    pathlib.Path(save_direc).mkdir(parents=True, exist_ok=True)
    # Create bases and domain
    y_basis = de.Fourier('y', Ny, interval=(0, Ly), dealias=3/2)   # Fourier basis in the x
    z_basis = de.Chebyshev('z', Nz, interval=(0, Lz), dealias=3/2) # Chebyshev basis in the z
    domain = de.Domain([y_basis, z_basis], grid_dtype=np.float64)  # Defining our domain
    z = domain.grid(1, scales=1)                                   # accessing the z values
    # 2D Anelastic hydrodynamics
    problem = de.IVP(domain, variables=['p', 's', 'u', 'v', 'w', 'sz', 'uz', 'vz', 'wz'])
    problem.meta['p','s','u','w']['z']['dirichlet'] = True
    # Defining model parameters
    problem.parameters['Ly'] = Ly
    problem.parameters['Lz'] = Lz
    problem.parameters['Ra'] = Ra
    problem.parameters['Pr'] = Pr
    problem.parameters['Ta'] = Ta
    problem.parameters['Lat'] = Lat
    problem.parameters['m'] = m
    problem.parameters['theta'] = theta
    problem.parameters['X'] = Ra/Pr
    problem.parameters['Y'] = (Pr*Pr*theta) / Ra
    problem.parameters['T'] = Ta**(1/2)
    Sfilename='S-Ra:'+str(Ra)+'-Pr:'+str(Pr)+'-Ta:'+str(Ta)+'-Np:'+str(Np)
```

```
    KEfilename='KE-Ra:'+str(Ra)+'-Pr:'+str(Pr)+'-Ta:'+str(Ta)+'-Np:'+str(Np)
    # Non-constant coeffiecents
    rho_ref = domain.new_field(name='rho_ref')
    rho_ref['g'] = (1-theta*z)**m
    rho_ref.meta['y']['constant'] = True
    problem.parameters['rho_ref'] = rho_ref         # Background state for rho
    T_ref = domain.new_field(name='T_ref')
    T_ref['g'] = 1-theta*z
    T_ref.meta['y']['constant'] = True
    problem.parameters['T_ref'] = T_ref             # Background state for T
    dz_rho_ref = domain.new_field(name='dz_rho_ref')
    dz_rho_ref['g'] = -theta*m*((1-theta*z)**(m-1))
    dz_rho_ref.meta['y']['constant'] = True
    problem.parameters['dz_rho_ref'] = dz_rho_ref   # z-derivative of rho_ref
    # Defining d/dz of s, u, and w for reducing our equations to first order
    problem.add_equation("sz - dz(s) = 0")
    problem.add_equation("uz - dz(u) = 0")
    problem.add_equation("vz - dz(v) = 0")
    problem.add_equation("wz - dz(w) = 0")
    # mass continuity with rho_ref and dz(rho_ref) expanded analytically
    problem.add_equation("  (1-theta*z)*(dy(v) + wz) - theta*m*w = 0 ")
    # x-component of the momentum equation
    problem.add_equation("  rho_ref*( dt(u) - dy(dy(u)) - dz(uz) + T*(w*cos(Lat) - v*sin(Lat)) ) - dz_rho_ref*uz = -rho_ref*( v*dy(u) + w*uz ) ")
    # y-component of the momentum equation
    problem.add_equation("  rho_ref*( dt(v) - (4/3)*dy(dy(v)) - dz(vz) - (1/3)*dy(wz) + T*u*sin(Lat) ) + dy(p) - dz_rho_ref*(vz + dy(w)) = -rho_ref*( v*dy(v) + w*vz )")
    # z-component of the momentum equation
    problem.add_equation("  rho_ref*T_ref*( dt(w) - X*s - dy(dy(w)) - (4/3)*dz(wz) - (1/3)*dy(vz) - T*u*cos(Lat) ) + T_ref*dz(p) + theta*m*p + (2/3)*theta*m*rho_ref*( 2*wz - dy(v) ) = -rho_ref*T_ref*( v*dy(w) + w*wz )")
    # entropy diffusion equation
    problem.add_equation("  T_ref*( Pr*dt(s) - dy(dy(s)) - dz(sz) ) + theta*(m+1)*sz = -Pr*T_ref*( v*dy(s) + w*sz ) + 2*Y*( dy(v)*dy(v) + wz*wz + vz*dy(w) - (1/3)*(dy(v) + wz)*(dy(v) + wz) + (1/2)*(dy(u)*dy(u) + uz*uz + vz*vz + dy(w)*dy(w)) )")
    # Flux equations for use in analysis outputs
    problem.add_bc("left(w) = 0")              # Impermeable bottom boundary
    problem.add_bc("right(w) = 0", condition="(ny != 0)")   # Impermeable top boundary
    problem.add_bc("right(p) = 0", condition="(ny == 0)")   # Required for equations to be well-posed - see https://bit.ly/2nPVWIg for a related discussion
    problem.add_bc("left(uz) = 0")             # Stress-free bottom boundary
    problem.add_bc("right(uz) = 0")            # Stress-free top boundary
    problem.add_bc("left(vz) = 0")
    problem.add_bc("right(vz) = 0")
    problem.add_bc("right(s) = 0")             # Fixed entropy at upper boundary, arbitarily set to 0
    problem.add_bc("left(sz) = -1")            # Fixed  flux at bottom boundary, F = F_cond
    # Build solver
    solver = problem.build_solver(de.timesteppers.RK222)
    logger.info('Solver built')
    # Initial conditions
    x = domain.grid(0)
    z = domain.grid(1)
```

```
    s = solver.state['s']
    w = solver.state['w']
    sz = solver.state['sz']
    # 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]
    # Linear background + perturbations damped at walls
    zb, zt = z_basis.interval
    pert =  1e-5*noise*(zt - z)*(z - zb)
    s['g'] = pert
    s.differentiate('z', out=sz)
    dt = initial_timestep # Initial timestep
    # Integration parameters --- Note if these are all set to np.inf, simulation will perpetually run.
    solver.stop_sim_time = end_sim_time
    solver.stop_wall_time = end_wall_time
    solver.stop_iteration = end_iterations
    # CFL criterion
    CFL = flow_tools.CFL(solver,initial_dt=dt,cadence=1,safety=1,max_change=1.5,min_change=0.01,max_dt=0.01,min_dt=1e-10)
    CFL.add_velocities(('v', 'w'))
    # Flow properties
    flow = flow_tools.GlobalFlowProperty(solver,cadence=10)
    flow.add_property("sqrt(u*u + v*v + w*w)",name='Re')
    # Analysis tasks
    analysis = solver.evaluator.add_file_handler(save_direc +'analysis',sim_dt=analysis_freq,max_writes=5000)
    analysis.add_task("s", layout='g', name='entropy')
    # Flux decomposition - Internal energy equation
    analysis.add_task("integ( integ( sqrt(u*u + v*v + w*w) , 'y')/Ly, 'z')/Lz", layout='g', name='Re') # Mean Reynolds number
    analysis.add_task(" integ( (integ(0.5*(u*u + v*v + w*w)*rho_ref,'y')/Ly), 'z')/Lz", layout='g', name='KE') # Mean KE
    # Creating a parameter file
    run_parameters = solver.evaluator.add_file_handler(save_direc+'run_parameters',wall_dt=1e20,max_writes=1)
    run_parameters.add_task(Ly,name="Ly")
    run_parameters.add_task(Lz,name="Lz")
    run_parameters.add_task(Ra,name="Ra")
    run_parameters.add_task(Pr,name="Pr")
    run_parameters.add_task(Np,name="Np")
    run_parameters.add_task(m,name="m")
    run_parameters.add_task(Ny,name="Ny")
    run_parameters.add_task(Nz,name="Nz")
    run_parameters.add_task("z",layout='g',name="z_grid")
    run_parameters.add_task(analysis_freq,name="ana_freq")
    run_parameters.add_task(max_dt,name="max_dt")
    try: # Main loop
        logger.info('Starting loop')
        start_time = time.time()
        while solver.ok:
            dt = CFL.compute_dt()
            dt = solver.step(dt)
            time.sleep(0.02)
            if (solver.iteration) == 1:
                # Prints various parameters to terminal upon starting the
```

```python
simulation
            logger.info('Parameter values imported form run_param_file.py:')
            logger.info('Ly = {}, Lz = {}; (Resolution of {},{})'.format(Ly, Lz, Ny, Nz))
            logger.info('Ra = {}, Pr = {}, Np = {}'.format(Ra, Pr, Np))
            logger.info('Analysis files outputted every {}'.format(analysis_freq))
            if end_sim_time != np.inf:
                logger.info('Simulation finishes at sim_time = {}'.format(end_sim_time))
            elif end_wall_time != np.inf:
                logger.info('Simulation finishes at wall_time = {}'.format(end_wall_time))
            elif end_iterations != np.inf:
                logger.info('Simulation finishes at iteration {}'.format(end_iterations))
            else:
                logger.info('No clear end point defined. Simulation may run perpetually.')
        if (solver.iteration-1) % 10 == 0:
            # Prints progress information include maximum Reynolds number every 10 iterations
            logger.info('Iteration: %i, Time: %e, dt: %e' %(solver.iteration, solver.sim_time, dt))
            logger.info('Max Re = %f' %flow.max('Re'))
except:
    logger.error('Exception raised, triggering end of main loop.')
    raise
finally:
    # Prints concluding information upon reaching the end of the simulation.
    end_time = time.time()
    if snaps==True:
        with open(Sfilename,'w') as file:
            file.write(str(gshape[0])+' '+str(gshape[1]))
    logger.info('Iterations: %i' %solver.iteration)
    logger.info('Sim end time: %f' %solver.sim_time)
    logger.info('Run time: %.2f sec' %(end_time-start_time))
    logger.info('Run time: %f cpu-hr' %((end_time-start_time)/60/60*domain.dist.comm_cart.size))
    with h5py.File("raw_data/analysis/analysis_s1/analysis_s1_p0.h5", mode='r') as file:
        times=file['scales']['sim_time'][:]
        data=file['tasks']['entropy'][:,:,:]
        ke=file['tasks']['KE'][:]
        ke=np.squeeze(ke)
        times=np.squeeze(times)
        n=times.shape[0]
        if snaps==True:
            for i in range(n):
                if i%snapshot_skip==0:
                    append_unique_value(Sfilename,times[i])
                    append_unique_array(Sfilename,data[i])
        save_fct_txt(times,ke,KEfilename)
```

A.6 Python File containing the functions used in A.4 and A.5

```python
# functions_txt.py

# IMPORTS ---------------------------------------------------
# We first import the necessary libraries like mentionned.

import numpy as np

# FUNCTIONS ------------------------------------------

def n_lines(filename):
    ''' Gives the number of lines contained in a txt file '''
    with open(filename, 'r') as reader:
        line = reader.readline()
        i=0
        while line != '':  # The EOF char is an empty string
            i=i+1
            line = reader.readline()
    return i

def save_fct_txt(X,Y,filename):
    ''' Saves two arrays X and Y in lines as : |X| |Y|'''
    n=X.shape[0]
    with open(filename, 'w') as adder:
        for i in range(n):
            string=str(X[i])+' '+str(Y[i])
            if i==0:
                adder.write(string)
            else:
                adder.write('\n'+string)

def read_state_file(filename):
    ''' Extract two arrays X and Y in lines as : |X| |Y|'''
    arrays=[]
    values=[]
    with open(filename, 'r') as reader:
        line = reader.readline()
        print(line)
        line=line.split()
        shape=(int(line[0]),int(line[1]))
        line = reader.readline()
        i=0
        while line != '':
            if i%2==0:
                values.append(line)
            else:
                arrays.append(line)
            i=i+1
            line=reader.readline()
    n=len(values)
    for i in range(n):
        values[i]=float(values[i])
        arrays[i]=string2array(arrays[i],shape)
    return values,arrays
```

```python

def string2array(string,shape):
    ''' transforms a string of floats into an array'''
    string=string[1:-1]
    array=np.fromstring(string, dtype=np.float64, sep=' ')
    array=array.reshape(shape)
    return array

def array2string(array):
    ''' transforms an array into a string of floats '''
    n=len(array)
    string=' '
    for i in range(n):
        string=string+str(array[i])+' '
    return string

def append_unique_value(filename,data):
    ''' writes a float value into a txt file '''
    n=n_lines(filename)
    with open(filename, 'a') as adder:
        if n==0:
            adder.write(str(data))
        else:
            adder.write('\n'+str(data))

def append_unique_array(filename,data):
    ''' writes an array into a txt file '''
    n=n_lines(filename)
    data=data.flatten()
    string=array2string(data)
    with open(filename, 'a') as adder:
        if n==0:
            adder.write(string)
        else:
            adder.write('\n'+string)

def load_last_array(filename,shape):
    '''Loads the last array contained into a txt file '''
    with open(filename, 'r') as reader:
        line = reader.readline()
        i=0
        while line != '':
            i=i+1
            last_line=line
            line = reader.readline()
    array=string2array(last_line,shape)
    return array

def load_last_value(filename):
    '''Loads the last float value contained into a txt file '''
    with open(filename, 'r') as reader:
        line = reader.readline()
        i=0
        while line != '':
            i=i+1
            last_line=line
            line = reader.readline()
    last_line=float(last_line)
```

```
    return last_line

def load_arrays(filename,frequency,shape):
    '''Loads the arrays contained into a txt file '''
    arrays=[]
    with open(filename, 'r') as reader:
        line = reader.readline()
        i=0
        while line != '':
            if i%frequency==0:
                arrays.append(line)
            i=i+1
            line=reader.readline()
    n=len(arrays)
    for i in range(n):
        arrays[i]=string2array(arrays[i],shape)
    return arrays

def load_values(filename,frequency):
    '''Loads the float values contained into a txt file '''
    values=[]
    with open(filename, 'r') as reader:
        line = reader.readline()
        i=0
        while line != '':
            if i%frequency==0:
                values.append(line)
            i=i+1
            line=reader.readline()
    n=len(values)
    for i in range(n):
        values[i]=float(values[i])
    return values

def clear_file(filename):
    ''' Allows us to clear every array or value in a txt file '''
    f=open(filename, 'w')
    f.close()
```

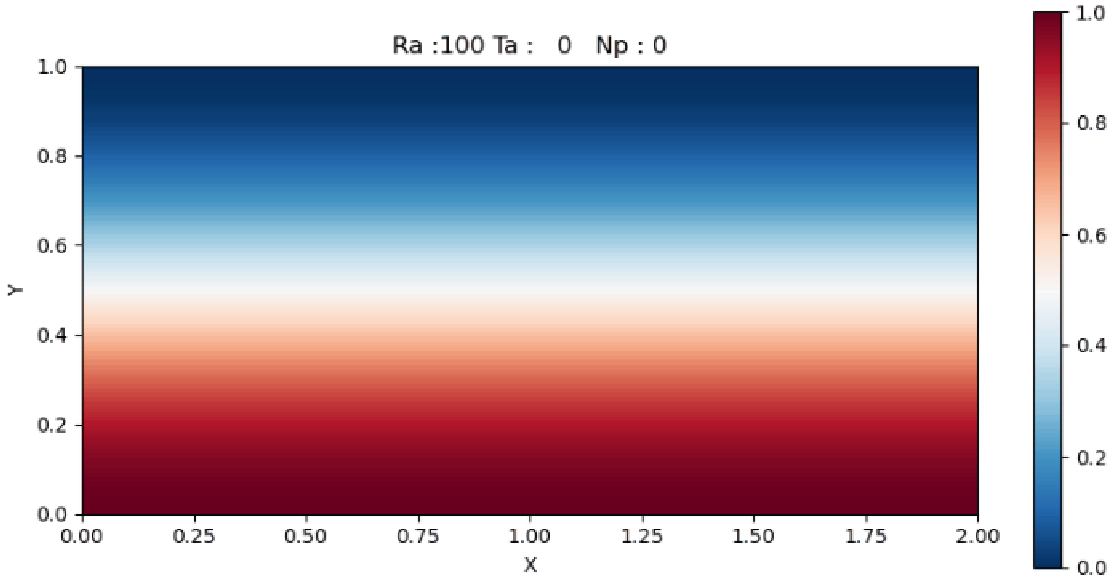
A.7 Python File containing the functions used to plot figures and creates GIFS of the field changes

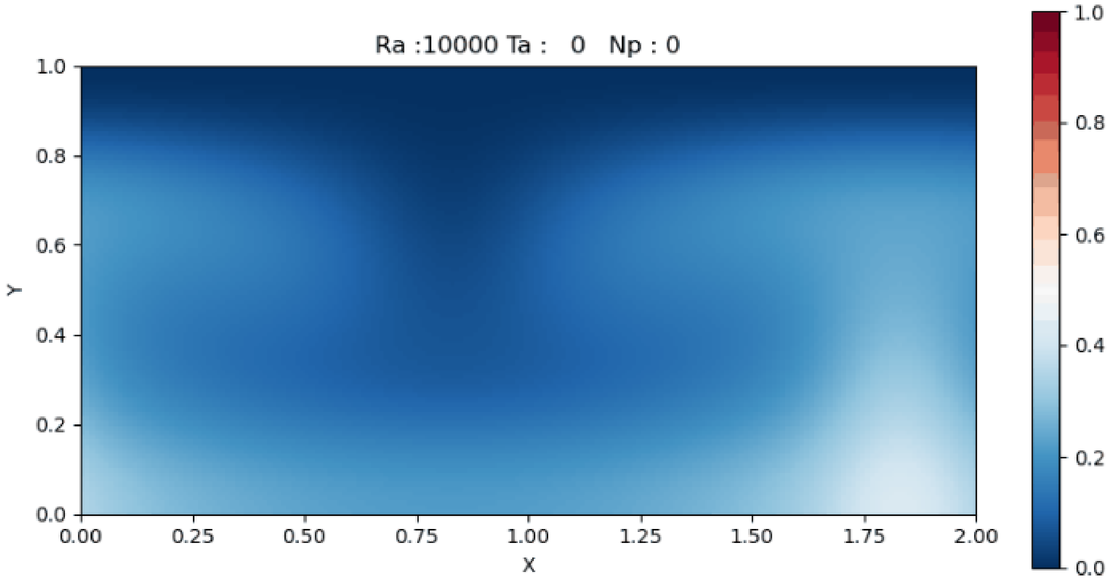
```
# plot_files.py

# IMPORTS --------------------------------------------------
# We first import the necessary libraries like mentionned.


import numpy as np
import matplotlib.pyplot as plt

# FUNCTIONS -------------------------------------------

def plot_convection_file(Ra,Pr,Ta,Np):
    ''' Extracts KEs from a specific Ta, Pr , Np and Ra'''
```

```
    file='KE-Ra:'+str(Ra)+'-Pr:'+str(Pr)+'-Ta:'+str(Ta)+'-Np:'+str(Np)
    with open(file,'r') as f:
        lines=f.readlines()
    n=len(lines)
    X=[]
    Y=[]
    for i in range(n):
        lines[i]=lines[i].split(' ')
        X.append(float(lines[i][0]))
        Y.append(float(lines[i][1]))
    plt.plot(X, Y, label="KE")
    plt.title('Ra='+str(Ra))
    plt.yscale("log")
    plt.legend(loc='upper right', fontsize=10)
    plt.show()


def plot_multiple_Ras(Ras,Pr,Ta,Np):
    ''' Extracts muliples KEs from a specific Ta, Pr and Np from a list of Rayleigh numbers and plots them'''
    n_files=len(Ras)
    X=[]
    Y=[]
    for j in range(n_files):
        file='KE-Ra:'+str(Ras[j])+'-Pr:'+str(Pr)+'-Ta:'+str(Ta)+'-Np:'+str(Np)
        with open(file,'r') as f:
            lines=f.readlines()
        n=len(lines)
        Xn=[]
        Yn=[]
        for i in range(n):
            lines[i]=lines[i].split(' ')
            Xn.append(float(lines[i][0]))
            Yn.append(float(lines[i][1]))
        X.append(Xn)
        Y.append(Yn)
    for k in range(n_files):
        plt.plot(X[k],Y[k],label="Ra"+str(Ras[k]))
    plt.grid()
    plt.yscale("log")
    plt.xscale("log")
    plt.xlabel("viscous time")
    plt.ylabel("KE")
    plt.legend(loc='upper right', fontsize=10)
    plt.show()

def post_processing_S(Ra,Pr,Np,Ta,plot=False,save=True,gif_fps=25):
    ''' Extracts the entropies and times from a given file and outputs a GIF of the field changes '''
    filename='S-Ra:'+str(Ra)+'-Pr:'+str(Pr)+'-Ta:'+str(Ta)+'-Np:'+str(Np)
    if save==True:
        gif_name='GIF-Ra:'+str(Ra)+'-Pr:'+str(Pr)+'-Ta:'+str(Ta)+'-Np:'+str(Np)+'.gif'
    Ts,Ss=read_state_file(filename)
    n=len(Ts)
    print("########## BEGINNING POST-PROCESSING ##########")
    Smax=1
    Smin=0
```

```
    fig,ax=plt.subplots(figsize=(10,5))
    heatmap=ax.imshow(np.flip(Ss[0].T,0),cmap='RdBu_r',extent=[0,2,0,1.],vmin=Smin,vmax=Smax)
    ax.set(xlabel='X',ylabel='Y')
    plt.colorbar(heatmap)
    if plot==True:
        fig.show()
        plt.pause(3)
    if save==True:
        Images=[]
    for i in range(1,n):
        heatmap.set_data(np.flip(Ss[i].T,0))
        ax.set(xlabel='X',ylabel='Y')
        if save==True and plot==True:
            plt.pause(0.01)
            plt.draw()
            image=np.frombuffer(fig.canvas.tostring_rgb(), dtype='uint8')
            image=image.reshape(fig.canvas.get_width_height()[::-1]+(3,))
            Images.append(image)
        if plot==True and save==False:
            plt.pause(0.01)
            plt.draw()
        if plot==False and save==True:
            fig.canvas.draw()
            image=np.frombuffer(fig.canvas.tostring_rgb(), dtype='uint8')
            image=image.reshape(fig.canvas.get_width_height()[::-1]+(3,))
            Images.append(image)
    if save==True:
        print("########## GIF CREATION ##########")
        imageio.mimsave(gif_name,Images,fps=gif_fps)
        print("########## GIF FINISHED ##########")
```

A.8 Entropy heatmaps resulting from 6.3





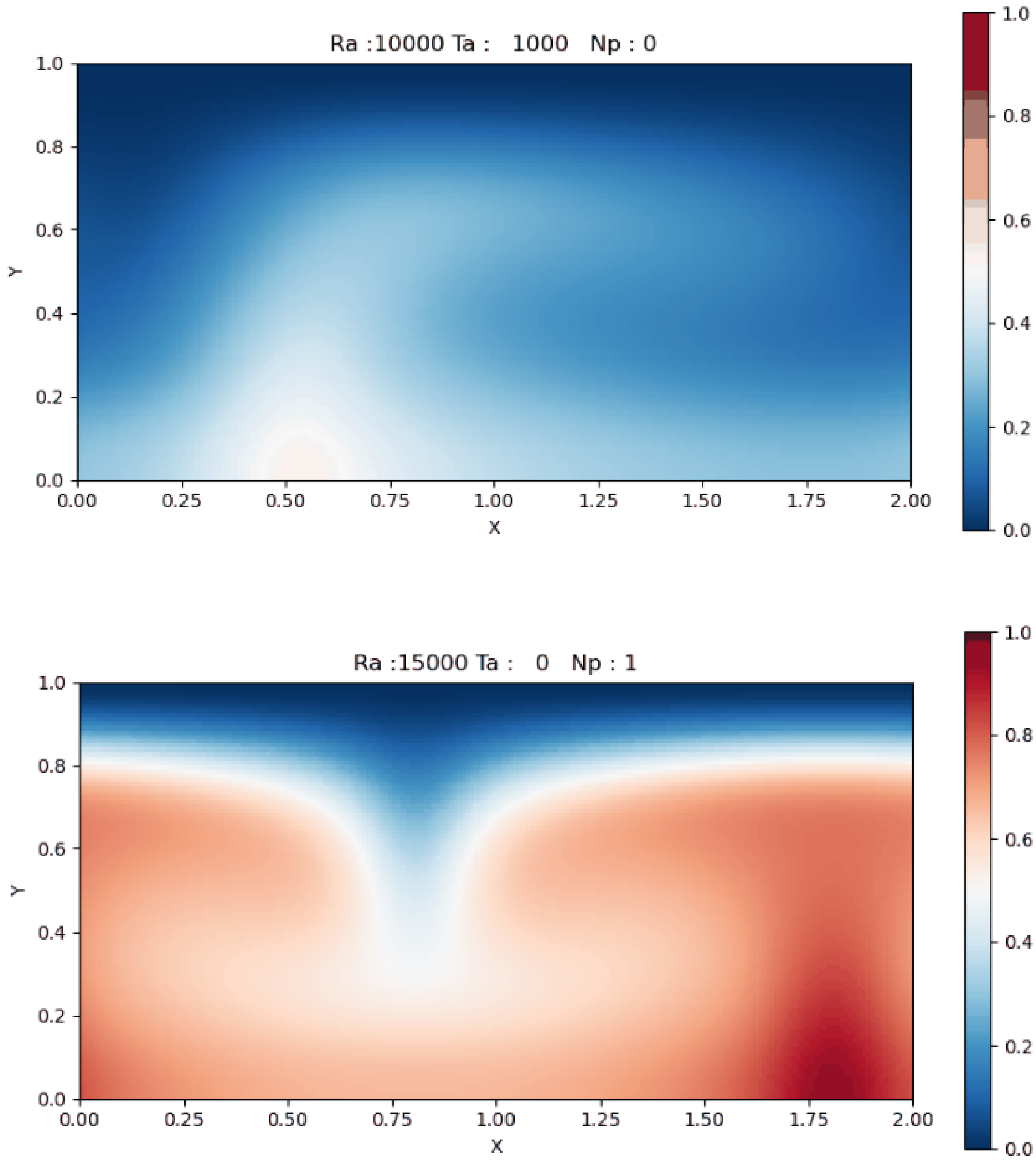


Figure A.1: Entropy heatmaps done using the results of 6.3. The parameters are those described in 6.1 and the numbers used are mentioned in their respective titles