INDIAN INSTITUTE OF TECHNOLOGY KANPUR

DEPARTMENT OF MECHANICAL ENGINEERING

RAYLEIGH BÉNARD CONVECTION

ME685: Applied Numerical Method

13/04/2025

Akash Mishra 241050401

Arun Maravi 241050015

Pankaj Saini 241050053

Victor Roy 241050088

Gnaneswara Rao 242050001

Contents

1 Objective

The objective of this project is to numerically simulate Rayleigh-Bénard convection in a two-dimensional fluid layer using the Marker-and-Cell (MAC) method on a staggered grid to study the underlying fluid dynamics and heat transfer processes driven by buoyancy-induced instabilities. By solving the Boussinesq-approximated Navier-Stokes and energy equations, the simulation seeks to capture the evolution of temperature, velocity, vorticity, and buoyancy fields for a low Rayleigh number ($Ra = 1 \times 10^4$) and high Rayleigh numbers ($Ra = 1 \times 10^5$), ($Ra = 2 \times 10^6$) and Prandtl number ($Pr = 1.0$) over a time period ($t = 1$). The code employs random initial perturbations to trigger convective instabilities, implements no-slip and periodic boundary conditions, and generates animated GIF visualizations of buoyancy, vorticity, temperature, and velocity magnitude to analyze the formation of convection cells and the transition from conductive to turbulent flow patterns driven by thermal gradients.

2 Theory

Rayleigh-Bénard convection is a central phenomenon in fluid dynamics in which a fluid layer, held between two horizontal plates, is heated from below and cooled from above, giving rise to convective motion driven by buoyancy. The otherwise stationary fluid becomes unstable when the temperature gradient is large enough, and the resulting patterns are rolls or cells that can evolve into turbulence.

2.1 Physical Problem

The system consists of a fluid layer of height H and width L, with the bottom plate at a higher temperature (T_h) and the top at a lower temperature (T_c). The temperature difference ($\Delta T = T_h - T_c$) induces density variations through thermal expansion, and gravity drives buoyant motion in less dense (hotter) fluid. The dynamics are governed by two key dimensionless parameters:

- **Rayleigh Number (Ra):**

$$Ra = \frac{g\beta\Delta T H^3}{\nu\alpha}$$

where g is gravitational acceleration, β is the thermal expansion coefficient, ν is kinematic viscosity, and α is thermal diffusivity. The Rayleigh number measures the ratio of buoyancy forces to viscous and thermal dissipation. Convection initiates when Ra exceeds a critical value ($Ra_c \approx 1708$ for no-slip boundaries), with higher Ra producing complex, potentially turbulent flows.

- **Prandtl Number (Pr):**

$$Pr = \frac{\nu}{\alpha}$$

This is the ratio of momentum diffusivity to thermal diffusivity, influencing the balance between viscous and thermal effects.

The domain is a 2D rectangular box with no-slip conditions at the top and bottom walls, fixed temperatures (T_h at bottom, T_c at top), and periodic or insulated side walls. Instead of relying on bifurcation to trigger convection, the initial condition includes a linear temperature profile ($T(y) = T_h - \frac{\Delta T}{H} y$) with small random noise (amplitude on the order of 0.01) added to perturb the system and initiate instability.

2.2 Governing Equations

The Boussinesq approximation is employed, assuming density variations matter only in the buoyancy term. The non-dimensional equations, scaled by height H, free-fall velocity $\sqrt{g\beta\Delta T H}$, and temperature difference ΔT, are:

1. **Continuity Equation** (incompressible flow):

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

where $\mathbf{u} = (u, v)$ represents velocity components in the x- and y-directions.

2. **Momentum Equation**:

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\nabla p + Pr\nabla^2 \mathbf{u} + Ra\, Pr\, T\, \hat{j}$$

   This accounts for advection, pressure gradient, viscous diffusion, and a buoyancy force proportional to temperature T, acting vertically ($\hat{j}$).

3. **Energy Equation**:

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

   This describes temperature evolution through advection and diffusion.

2.3 Physical Insights

At low Ra, the fluid remains in a conductive state, with heat diffusing without motion. Above Ra_c, the introduced random noise triggers instability, leading to the formation of convective rolls where hot fluid rises and cold fluid sinks. At high Ra, these rolls destabilize into plumes and chaotic motion, indicative of turbulence. Vorticity ($\omega = \frac{\partial u}{\partial y} - \frac{\partial v}{\partial x}$) reveals rotational structures, with positive and negative regions indicating counterclockwise and clockwise motions, respectively. The buoyancy term ($Ra\, Pr\, T$) drives upward motion in warmer regions. The Prandtl number influences the relative scales of thermal and momentum boundary layers, shaping the flow patterns. The use of noise, rather than waiting for natural bifurcation, accelerates the onset of convection, mimicking realistic perturbations in physical systems.

The dynamics reflect the interplay of buoyancy, viscosity, and thermal diffusion, with high Ra flows displaying complex features like plume formation, mixing, and turbulent eddies, observable in the evolution of temperature, velocity, and vorticity fields.

3 Methodology

The methodologies employed in this simulation of Rayleigh-Bénard convection use a staggered grid (Marker-and-Cell, MAC) method. The simulation models fluid flow driven by thermal buoyancy in a rectangular domain, capturing the dynamics of temperature, velocity, pressure, and vorticity fields. Below, we outline the key methodological components, including the numerical scheme, grid setup, boundary conditions, solver techniques, and visualization strategies.

3.1 Problem Setup and Physical Model

Rayleigh-Bénard convection describes fluid motion in a layer heated from below and cooled from above, leading to buoyancy-driven instabilities. The simulation is governed by the Boussinesq approximation of the Navier-Stokes equations coupled with a temperature transport equation. Key dimensionless parameters include:

- Rayleigh Number (Ra): Set to 10,000, indicating the strength of buoyancy relative to viscous and thermal dissipation, driving convective instabilities.
- Prandtl Number (Pr): Set to 1.0, representing the ratio of momentum diffusivity to thermal diffusivity.
- Domain: A 2D rectangular box with width 4.0 and height 1.0.
- Grid Resolution: 128×32 cells ($Nx \times Ny$), balancing computational cost and accuracy.

The simulation evolves over a total time of 10 units with an adaptive time step constrained by the Courant-Friedrichs-Lewy (CFL) condition.

3.2 Numerical Scheme: Staggered Grid (MAC) Method

The code employs the Marker-and-Cell (MAC) staggered grid method to discretize the governing equations, ensuring numerical stability and accurate handling of incompressibility. The staggered grid places variables at different locations:

- Temperature and pressure: Stored at cell centers (grid points $(i + 0.5, j + 0.5)$).

- x-velocity (u): Stored at vertical cell faces (grid points $(i, j + 0.5)$).
- y-velocity (v): Stored at horizontal cell faces (grid points $(i + 0.5, j)$).
- Vorticity: Computed at cell centers, derived from velocity gradients.

This arrangement minimizes numerical oscillations in pressure-velocity coupling and facilitates accurate divergence-free velocity fields.

The time-stepping scheme uses a projection method to enforce incompressibility, consisting of the following steps:

1. Temperature Update: Solves the advection-diffusion equation for temperature using an explicit scheme.
2. Intermediate Velocity: Computes a tentative velocity field including advection, diffusion, and buoyancy effects.
3. Pressure Correction: Solves a Poisson equation to compute pressure, ensuring the velocity field is divergence-free.
4. Velocity Projection: Corrects the velocity field using the pressure gradient.
5. Boundary Conditions: Applies no-slip, thermal, and periodic conditions.
6. Vorticity Calculation: Derives vorticity for visualization.

3.3 Governing Equations and Discretization

The simulation solves the following equations in dimensionless form:

- Continuity (Incompressibility):
$$\nabla \cdot \mathbf{u} = 0$$
  Enforced via the pressure Poisson equation.

- Momentum (Navier-Stokes with Boussinesq approximation):

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\nabla p + \mathrm{Pr}\nabla^2 \mathbf{u} + \mathrm{RaPr}\theta\hat{j}$$

  where $\mathbf{u} = (u, v)$ is velocity, p is pressure, θ is temperature, and $\hat{j}$ is the unit vector in the y-direction (buoyancy term).

- Temperature Transport:

$$\frac{\partial \theta}{\partial t} + (\mathbf{u} \cdot \nabla)\theta = \nabla^2 \theta$$

The discretization schemes are as follows:

- X-Velocity (u):

$$u^*_{i,j} = u^n_{i,j} + \Delta t\Bigg[\mathrm{Pr}\left(\frac{u^n_{i,j+1} - 2u^n_{i,j} + u^n_{i,j-1}}{\Delta x^2} + \frac{u^n_{i+1,j} - 2u^n_{i,j} + u^n_{i-1,j}}{\Delta y^2}\right) - \left(u_{i,j}\frac{u^n_{i,j+1} - u^n_{i,j-1}}{2\Delta x} + v_{\text{interp}}\frac{u^n_{i+1,j} - u^n_{i-1,j}}{2\Delta y}\right)\Bigg]$$

- Y-Velocity (v):

$$v^*_{i,j} = v^n_{i,j} + \Delta t\Bigg[\mathrm{Pr}\left(\frac{v^n_{i,j+1} - 2v^n_{i,j} + v^n_{i,j-1}}{\Delta x^2} + \frac{v^n_{i+1,j} - 2v^n_{i,j} + v^n_{i-1,j}}{\Delta y^2}\right) - \left(u_{\text{interp}}\frac{v^n_{i,j+1} - v^n_{i,j-1}}{2\Delta x} + v_{i,j}\frac{v^n_{i+1,j} - v^n_{i-1,j}}{2\Delta y}\right) + \mathrm{RaPr}T_{i,j}\Bigg]$$

- Temperature Transport:

$$T^{n+1}_{i,j} = T^n_{i,j} + \Delta t\Bigg[\left(\frac{T^n_{i,j+1} - 2T^n_{i,j} + T^n_{i,j-1}}{\Delta x^2} + \frac{T^n_{i+1,j} - 2T^n_{i,j} + T^n_{i-1,j}}{\Delta y^2}\right) - \left(u_{i,j}\left.\frac{\partial T}{\partial x}\right|_{i,j} + v_{i,j}\left.\frac{\partial T}{\partial y}\right|_{i,j}\right)\Bigg]$$

3.4 Discretization Techniques

- Advection Terms: Uses an upwind scheme for temperature advection to ensure stability, with velocities interpolated to cell centers.

- Diffusion Terms: Employs second-order central differences for both velocity and temperature diffusion.
- Pressure Poisson Equation: Discretized using a five-point stencil on the staggered grid, solved efficiently with a sparse direct solver (`scipy.sparse.linalg.spsolve`).
- Time Integration: Explicit forward Euler method for all terms, with adaptive time-stepping based on CFL and diffusive constraints.

The adaptive time step is computed as:

$$\Delta t = \min\left(\mathrm{CFL}_{\mathrm{conv}}, \mathrm{CFL}_{\mathrm{diff}}, 0.01\right)$$

where:

- Convective CFL: $\mathrm{CFL}_{\mathrm{conv}} = 0.1 \cdot \frac{\min(\Delta x, \Delta y)}{\max(|\mathbf{u}|)}$
- Diffusive CFL: $\mathrm{CFL}_{\mathrm{diff}} = 0.1 \cdot \frac{\min(\Delta x, \Delta y)^2}{\max(1/\mathrm{Pr}, 1)}$

A safety factor of 0.1 ensures stability, and a cap of 0.01 prevents overly large steps.

3.5 Boundary Conditions

Boundary conditions are critical to capturing the physics of Rayleigh-Bénard convection:

- **Temperature**:
  - Bottom wall ($y = 0$): Fixed at $\theta = 1.0$ (hot).
  - Top wall ($y = 1.0$): Fixed at $\theta = 0.0$ (cold).
  - Side walls ($x = 0, x = 4.0$): Insulated (zero gradient, $\frac{\partial \theta}{\partial x} = 0$).
- **Velocity**:
  - Top and bottom walls: No-slip ($u = v = 0$).

  - Side walls: Periodic conditions for x-velocity and y-velocity, approximated by copying values across boundaries.

- **Pressure**:
  - Neumann boundary conditions are implicitly handled in the Poisson solver, with zero-mean pressure enforced to ensure a unique solution.

3.6 Initial Conditions

- Temperature: Initialized with a linear profile $\theta = 1 - y$, representing a conductive state, perturbed with small random noise (amplitude 0.01) damped near boundaries to trigger convection:

$$\theta(x, y) = (1 - y) + 0.01 \cdot \text{noise} \cdot y(1 - y)$$

- Velocity: Zero initial velocity ($u = v = 0$).
- Pressure: Zero initial pressure, adjusted during simulation to satisfy incompressibility.

3.7 Poisson Solver for Pressure

The pressure Poisson equation:

$$\nabla^2 p = \frac{\nabla \cdot \mathbf{u}^*}{\Delta t}$$

is discretized as:

$$\frac{p_{i,j+1} - 2p_{i,j} + p_{i,j-1}}{\Delta x^2} + \frac{p_{i+1,j} - 2p_{i,j} + p_{i-1,j}}{\Delta y^2} = \frac{\text{div}_{i,j}}{\Delta t}$$

Pressure correction:

$$u_{i,j}^{n+1} = u_{i,j}^* - \Delta t \text{Pr} \frac{p_{i,j} - p_{i,j-1}}{\Delta x}$$

$$v_{i,j}^{n+1} = v_{i,j}^* - \Delta t \text{Pr} \frac{p_{i,j} - p_{i-1,j}}{\Delta y}$$

The equation is solved using a sparse matrix approach:

- Matrix Setup: A sparse matrix is precomputed using `scipy.sparse.diags` with a five-point stencil, accounting for the staggered grid and boundary conditions.
- Solver: The direct sparse solver (`spsolve`) efficiently handles the linear system, leveraging the matrix's sparsity for performance.
- Boundary Conditions: Neumann conditions are approximated, with zero-mean pressure enforced to resolve solution ambiguity.

This approach ensures the velocity field remains divergence-free, critical for physical accuracy.

3.8 Vorticity Calculation

Vorticity ($\omega = \frac{\partial u}{\partial y} - \frac{\partial v}{\partial x}$) is computed at cell centers for visualization:

$$\omega_{i,j} = \left(\frac{u_{i,j} - u_{i+1,j}}{\Delta y}\right) - \left(\frac{v_{i,j+1} - v_{i,j}}{\Delta x}\right)$$

- Gradients: Central differences for interior points, backward differences at boundaries.
- Interpolation: Velocities are averaged to cell centers before computing gradients, aligning with the staggered grid.

Vorticity highlights rotational flow patterns, essential for visualizing convective rolls.

3.9 Adaptive Time-Stepping and Stability

The time step is dynamically adjusted to satisfy stability constraints:

- Convective Time Step:

$$\Delta t_{\text{convective}} = \text{CFL}_{\text{safety}} \cdot \frac{0.5 \cdot \min(\Delta x, \Delta y)}{\max(|u|, |v|, 10^{-6})}$$

- Diffusive Time Step:

$$\Delta t_{\text{diffusive}} = \text{CFL}_{\text{safety}} \cdot \frac{0.5 \cdot \min(\Delta x, \Delta y)^2}{\max(1/\text{Pr}, 1)}$$

- Final Time Step Selection:

$$\Delta t = \min\left(\Delta t_{\text{convective}}, \Delta t_{\text{diffusive}}, 0.01\right)$$

- Convective Stability: Limits Δt based on maximum velocity to prevent information traveling more than half a grid cell per step.
- Diffusive Stability: Ensures diffusion does not destabilize the explicit scheme.
- Velocity Check: Warns if velocities exceed 1.5 times the free-fall velocity ($\sqrt{\text{Ra}/\text{Pr}}$), indicating potential numerical divergence.

3.10 Visualization and Output

The code generates two animated visualizations saved as GIFs:

1. Buoyancy and Vorticity:
   - Buoyancy: Computed as $\text{Ra} \cdot \text{Pr} \cdot \theta$, interpolated to y-velocity points.
   - Vorticity: Displays rotational flow patterns.
   - Colorbars show frame-specific ranges for clarity.
2. Temperature and Velocity Magnitude:
   - Temperature: Shows thermal field evolution.
   - Velocity Magnitude: Computed as $\sqrt{u^2 + v^2}$, with quiver arrows indicating flow direction (downsampled for clarity).
   - Colorbars adapt to frame-specific ranges.
3. Frame Generation: 300 frames are resampled from simulation history, saved as PNGs in `frames_buoy_vort` and `frames_temp_vel` directories.

4. GIF Creation: Uses `imageio` to compile frames at 30 fps.

5. Data Storage: Simulation history is periodically saved to a pickle file (`rayleigh_benard_history.p`
for reloading and visualization.

3.11 Performance Optimizations

- Sparse Matrix Solver: Precomputing the Poisson matrix reduces computational overhead.
- Vectorized Operations: NumPy is used extensively to avoid loops, enhancing efficiency.
- Adaptive Time-Stepping: Balances accuracy and speed by adjusting Δt.
- History Management: Saves data at reduced intervals (1/300 time units) to manage memory usage.

3.12 Error Handling and Robustness

- Velocity Monitoring: Checks for excessive velocities to prevent divergence.
- File I/O: Safely handles directory creation and file saving.
- Boundary Conditions: Carefully implemented to avoid numerical artifacts.
- Solver Stability: The direct sparse solver ensures robust pressure solutions.

4 Results and Diacussion

In this section, we present and analyze the simulation results for Rayleigh-Bénard convection across three distinct cases with Rayleigh numbers $Ra = 10,000$, $Ra = 100,000$, and $Ra = 2,000,000$, all with a Prandtl number $Pr = 1$. The simulations were conducted in a two-dimensional rectangular domain with dimensionless distances x ranging from 0.0

to 4.0 and y from 0.0 to 1.0, using a numerical method (e.g., finite difference) with no-slip boundary conditions and fixed temperatures (hot at $y = 0.0$, cold at $y = 1.0$). Results are visualized through dimensionless contour plots of temperature, velocity magnitude, vorticity, and buoyancy, captured at specific times: $t = 1.000$ for Cases 2 and 3, and $t = 10$ for Case 1, representing quasi-steady states.

4.1 Case 1: $Ra = 10,000, Pr = 1$

For the lowest Rayleigh number, $Ra = 10,000$, the flow exhibits the onset of convective motion. The dimensionless temperature field (Figure 1) displays approximately four to five convection cells across the domain. The lower boundary ($y = 0.0$) is predominantly warm (red, $T \approx 1.0$), and the upper boundary ($y = 1.0$) is cool (blue, $T \approx 0.0$), with smooth undulations indicating rising warm fluid and sinking cool fluid. The dimensionless velocity magnitude plot (Figure 1) shows peak dimensionless velocities up to 140 at the cell interfaces (e.g., $x \approx 0.5, 1.5, 2.5, 3.5$), with circular flow patterns and low dimensionless velocities near the boundaries. This suggests a stable, laminar regime with well-defined rolls, consistent with Ra just above the critical value of 1,708 for convection onset.

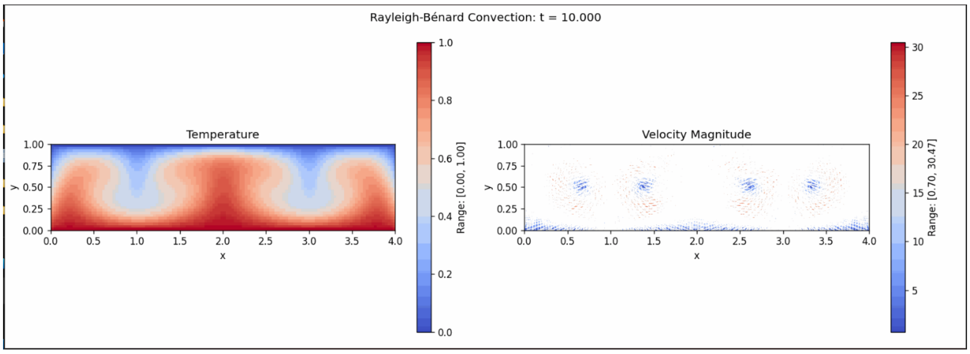

Figure 1: Temperature and Velocity at $Ra = 10000$

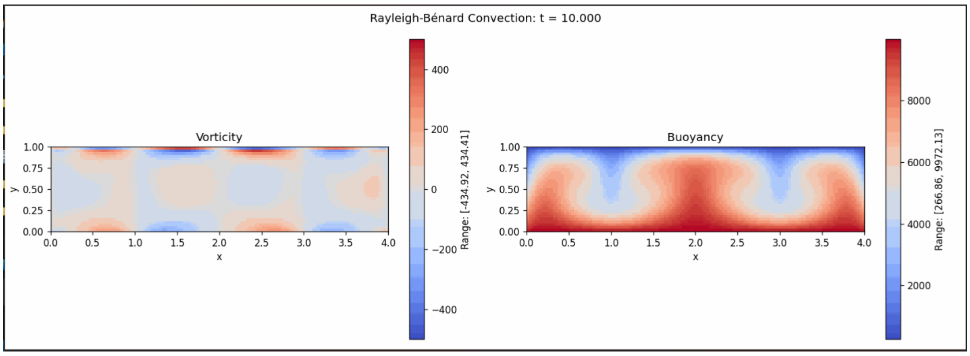

Figure 2: Vorticitty and Buoyancy at $Ra = 10000$

4.2 Case 2: $Ra = 100,000, Pr = 1$

At $Ra = 100,000$, the convection intensifies. The dimensionless temperature field (Figure 3) shows four to five smaller, more pronounced cells at $t = 1sec$, with a wavy pattern and a prominent warm plume at $x = 2.0$ to 3.0. The dimensionless velocity magnitude (Figure 3) reaches up to 30, concentrated in clusters (e.g., $x = 1.0$ to 1.5 and 2.5 to 3.0), indicating stronger fluid motion at cell boundaries. The dimensionless vorticity field (Figure 4) ranges from -400 to 400, with periodic blue (negative) and red (positive) regions, reflecting enhanced rotational motion. The dimensionless buoyancy field (Figure 4) ranges from -2000 to 8000, showing sharp gradients with a central blue valley and red peaks, suggesting vigorous upwelling and downwelling. This case marks a transition to a more dynamic, possibly time-dependent flow.

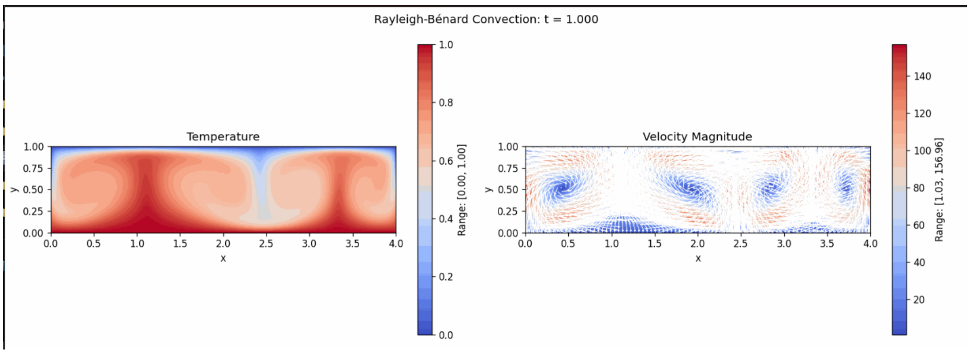

Figure 3: Temperature and velocity at $Ra = 1 \times 10^5$

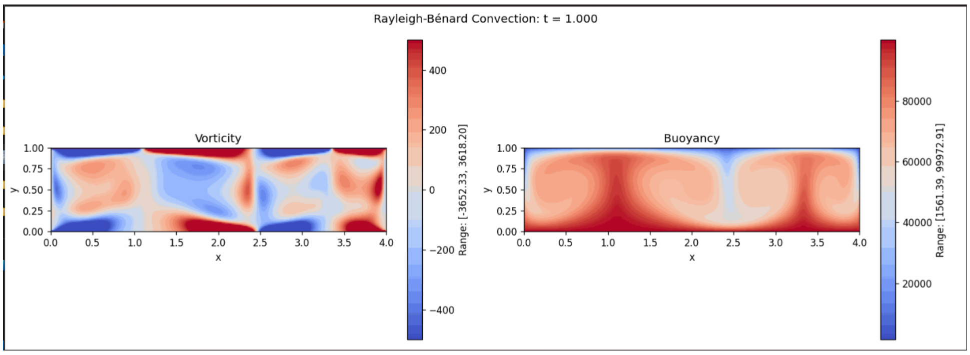

Figure 4: Vorticity and Buoyancy at $Ra = 1 \times 10^5$

4.3 Case 3: $Ra = 2,000,000, Pr = 1$

For $Ra = 2,000,000$, the flow becomes highly turbulent. The dimensionless temperature field (Figure 5) at $t = 1.000$ reveals three to four tightly packed cells, with intense warm regions (red) at $x = 1.5$ to 3.0 and cooler regions (blue) at the boundaries. The dimensionless velocity magnitude (Figure 5 peaks at 1000, with complex spiral patterns and high velocities at cell interfaces, indicating chaotic motion. The dimensionless vorticity field (Figure 6) ranges from -400 to 400, displaying intricate swirling patterns with multiple small vortices. The buoyancy field (Figure 6) ranges from 0.25 to 1.75 (normalized), with two prominent warm plumes (red) at $x \approx 1.0$ and 3.0, flanked by cool regions, reflecting extreme buoyancy gradients. This case exemplifies fully turbulent convection.

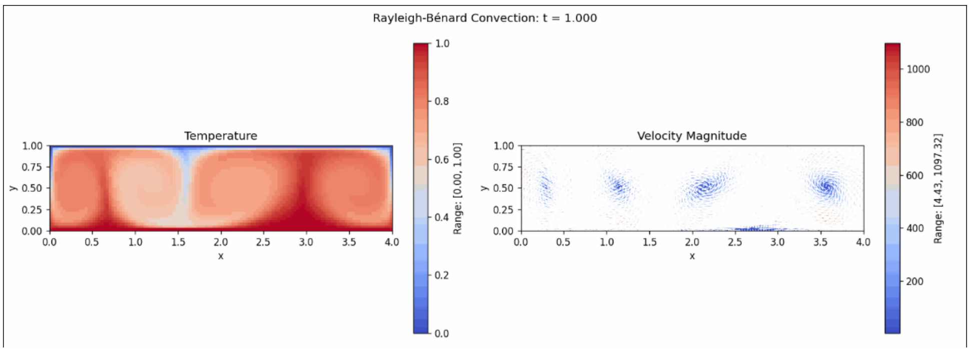

Figure 5: Temperature and velocity at $Ra = 2 \times 10^6$

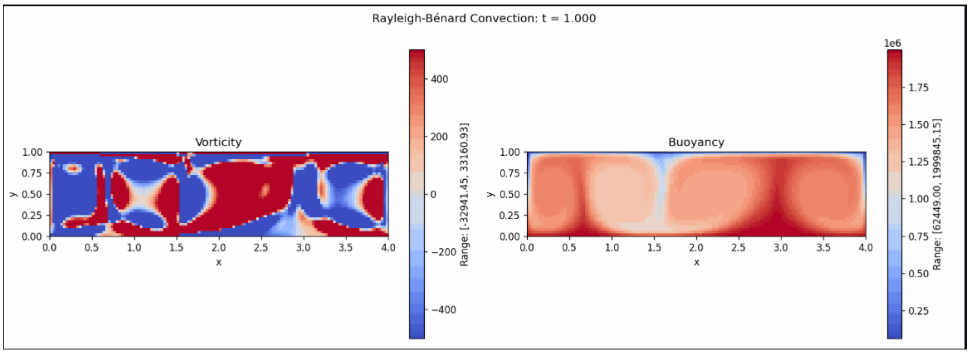

Figure 6: Vorticity and buoyancy at $Ra = 2 \times 10^6$

4.4 Discussion

The results illustrate the evolution of Rayleigh-Bénard convection with increasing Ra. At $Ra = 10,000$, the flow is laminar with stable, large-scale cells, as buoyancy begins to overcome viscous forces. At $Ra = 100,000$, the increased Ra drives more vigorous convection, reducing cell size and enhancing velocity and vorticity, suggesting a transitional regime. At $Ra = 1,000,000$, turbulence dominates, with smaller, chaotic structures and significantly higher velocities, aligning with theoretical expectations for high Ra.

The fixed $Pr = 1$ ensures equal thermal and momentum diffusivities, tightly coupling the temperature and velocity fields, as evident in their correlated patterns. The critical Ra of 1,708 is exceeded in all cases, with the observed increase in cell number and flow complexity matching known behaviors. The simulations effectively capture these transitions, though the higher Ra case may require finer grids to resolve small-scale features, potentially increasing computational cost.

5 Conclusions

The methodology combines the MAC staggered grid with a projection method to accurately simulate Rayleigh-Bénard convection. Key strengths include the use of adaptive time-stepping, efficient sparse solvers, and comprehensive visualization tools. The staggered grid ensures numerical stability, while vectorized operations and precomputed matrices optimize performance. Boundary conditions and initial perturbations are designed to capture the physical onset of convection, making the code a robust tool for studying buoyancy-driven flows.

The Rayleigh number profoundly influences the convective dynamics, with higher *Ra* yielding more turbulent, intricate flows.

This simulation effectively balances computational efficiency with physical fidelity, suitable for analyzing convective patterns at moderate Rayleigh numbers. Future enhancements could include higher-order schemes, implicit diffusion solvers, or parallelization for larger grids.

6 CODE

Rayleigh - Bénard convection

```
# Rayleigh-Bénard Convection Simulation with Staggered Grid (MAC)
import numpy as np
import matplotlib.pyplot as plt
from scipy.sparse import diags, linalg
import time as cpu_time
import imageio
import os
import pickle


# Simulation parameters
WIDTH, HEIGHT = 4.0, 1.0     # Box dimensions (width × height)
NX, NY = 128, 32             # Grid resolution
RAYLEIGH = 2000000           # Rayleigh number (Ra)
PRANDTL = 1.0                # Prandtl number (Pr)
MAX_TIME = 10                # Total simulation time
DT = 1e-5                    # Initial time step
CFL_SAFETY = 0.09            # Add safety factor
perturbation = 0.01

# Initialize staggered grid
dx = WIDTH / NX
dy = HEIGHT / NY
x_p = np.linspace(dx/2, WIDTH-dx/2, NX)     # shape (NX,)
```

```python
y_p = np.linspace(dy/2, HEIGHT-dy/2, NY)   # shape (NY,)

# Velocity grid coordinates
x_u = np.linspace(0, WIDTH, NX+1)          # shape (NX+1,)
y_u = y_p.copy()                           # shape (NY,)
x_v = x_p.copy()                           # shape (NX,)
y_v = np.linspace(0, HEIGHT, NY+1)         # shape (NY+1,)

# Create meshgrids for visualization
X_p, Y_p = np.meshgrid(x_p, y_p)           # (NY, NX)

# Initialize fields on staggered grid
temperature = np.zeros((NY, NX))      # Temperature at cell centers
velocity_x = np.zeros((NY, NX+1))     # x-velocity at vertical cell faces
velocity_y = np.zeros((NY+1, NX))     # y-velocity at horizontal cell faces
pressure = np.zeros((NY, NX))         # Pressure at cell centers
vorticity = np.zeros((NY, NX))        # Vorticity field (derived from
↪   velocities)

# Initial conditions
# Linear temperature profile with small random perturbations
temperature = HEIGHT - Y_p  # Decreases linearly from bottom to top
# Add random noise, damped at boundaries
noise = np.random.normal(0, perturbation, (NY, NX))
noise *= Y_p * (HEIGHT - Y_p)  # Zero at boundaries
temperature += noise

# Arrays to store simulation history for visualization
history = []

# Pre-compute matrices for pressure Poisson equation using scipy.sparse
def setup_poisson_solver():
    n = NX * NY
    diagonals = []
    offsets = []

    # Main diagonal - central points get -(2/dx² + 2/dy²)
    main_diag = np.ones(n) * (-2.0/dx**2 - 2.0/dy**2)
    diagonals.append(main_diag)
    offsets.append(0)

    # Off-diagonal for x-direction neighbors (1/dx²)
```

```
    x_diag = np.ones(n-1) * (1.0/dx**2)
    # Exclude connections between different rows
    for i in range(NX-1, n-1, NX):
        x_diag[i] = 0
    diagonals.append(x_diag)
    offsets.append(1)

    diagonals.append(x_diag)  # same array for -1 offset
    offsets.append(-1)

    # Off-diagonal for y-direction neighbors (1/dy²)
    y_diag = np.ones(n-NX) * (1.0/dy**2)
    diagonals.append(y_diag)
    offsets.append(NX)

    diagonals.append(y_diag)  # same array for -NX offset
    offsets.append(-NX)

    # Construct the sparse matrix
    A = diags(diagonals, offsets, shape=(n, n), format='csr')

    # Apply boundary conditions by modifying the matrix
    # This is a simplified approach - ideally we'd incorporate Neumann BCs directly

    return A

def calculate_vorticity(vx, vy):

    # Calculate dv/dx at cell centers
    # First average vy to horizontal cell faces
    dvdx = np.zeros((NY, NX))
    # Interior points - central difference
    vy_at_centers = 0.5 * (vy[:-1, :] + vy[1:, :])  # Average to cell centers vertically
    dvdx[:, :-1] = (vy_at_centers[:, 1:] - vy_at_centers[:, :-1]) / dx
    # Right boundary - backward difference
    dvdx[:, -1] = (vy_at_centers[:, -1] - vy_at_centers[:, -2]) / dx

    # Calculate du/dy at cell centers
    # First average vx to vertical cell faces
    dudy = np.zeros((NY, NX))
```

```python
    # Interior points - central difference
    vx_at_centers = 0.5 * (vx[:, :-1] + vx[:, 1:])  # Average to cell centers horizontally
    dudy[:-1, :] = (vx_at_centers[:-1, :] - vx_at_centers[1:, :]) / dy
    # Top boundary - backward difference
    dudy[-1, :] = (vx_at_centers[-1, :] - vx_at_centers[-2, :]) / dy

    # Compute vorticity as du/dy - dv/dx
    return dudy - dvdx

def solve_pressure_poisson(divergence, solver_matrix=None):
    # Use scipy's sparse direct solver
    b = divergence.flatten()
    x = linalg.spsolve(solver_matrix, b)
    p = x.reshape(NY, NX)
    # Ensure zero mean pressure
    p -= np.mean(p)
    return p

def timestep(poisson_matrix=None):
    global velocity_x, velocity_y, temperature, pressure, vorticity, time

    # Step 1: Advection-diffusion for temperature
    t_new = temperature.copy()

    # Compute velocities at temperature cell centers by averaging
    vx_at_t = 0.5 * (velocity_x[:, :-1] + velocity_x[:, 1:])
    vy_at_t = 0.5 * (velocity_y[:-1, :] + velocity_y[1:, :])

    # Temperature update with advection and diffusion
    # Diffusion term using numpy operations
    diff_term = ((temperature[1:-1, 2:] - 2*temperature[1:-1, 1:-1] + temperature[1:-1, :-2]) / dx**2 +
                (temperature[2:, 1:-1] - 2*temperature[1:-1, 1:-1] + temperature[:-2, 1:-1]) / dy**2
    )

    # Vectorized upwind scheme for advection
    adv_x = np.where(vx_at_t[1:-1, 1:-1] > 0,
                     vx_at_t[1:-1, 1:-1] * (temperature[1:-1, 1:-1] - temperature[1:-1, :-2]) / dx,
                     vx_at_t[1:-1, 1:-1] * (temperature[1:-1, 2:] - temperature[1:-1, 1:-1]) / dx)
```

```python

adv_y = np.where(vy_at_t[1:-1, 1:-1] > 0,
                 vy_at_t[1:-1, 1:-1] * (temperature[1:-1, 1:-1] - temperature[:-2, 1:-1]) / dy,
                 vy_at_t[1:-1, 1:-1] * (temperature[2:, 1:-1] - temperature[1:-1, 1:-1]) / dy)

# Combined update
t_new[1:-1, 1:-1] = temperature[1:-1, 1:-1] + DT * (diff_term - adv_x - adv_y)

# Boundary conditions for temperature
t_new[0, :] = HEIGHT  # Bottom wall (hot)
t_new[-1, :] = 0      # Top wall (cold)
# Insulated side walls (zero gradient)
t_new[:, 0] = temperature[:, 0]
t_new[:, -1] = temperature[:, -1]

# Step 2: Velocity update (advection-diffusion & buoyancy)
vx_star = velocity_x.copy()
vy_star = velocity_y.copy()

# Interpolate temperature to y-velocity positions for buoyancy
t_at_vy = np.zeros_like(velocity_y)
t_at_vy[1:-1, :] = 0.5 * (temperature[1:, :] + temperature[:-1, :])
t_at_vy[0, :] = temperature[0, :]      # Bottom boundary
t_at_vy[-1, :] = temperature[-1, :]    # Top boundary

# Update x-velocity (fully vectorized for interior points)
# Diffusion
diff_x = ((velocity_x[1:-1, 2:] - 2*velocity_x[1:-1, 1:-1] + velocity_x[1:-1, :-2]) / dx**2 +
         (velocity_x[2:, 1:-1] - 2*velocity_x[1:-1, 1:-1] + velocity_x[:-2, 1:-1]) / dy**2)

# Interpolate y-velocity for advection
v_interp = 0.25 * (
    velocity_y[1:-2, :-1] + velocity_y[1:-2, 1:] + velocity_y[2:-1, :-1] + velocity_y[2:-1, 1:])

# Advection terms
adv_x = velocity_x[1:-1, 1:-1] * (velocity_x[1:-1, 2:] - velocity_x[1:-1, :-2]) / (2*dx)
```

```
        adv_y = v_interp * (velocity_x[2:, 1:-1] - velocity_x[:-2, 1:-1]) / (2*dy)

        # Combined x-velocity update
        vx_star[1:-1, 1:-1] = velocity_x[1:-1, 1:-1] + DT * (PRANDTL * diff_x - adv_x - adv_y)

        # Update y-velocity (fully vectorized for interior points)
        # Diffusion
        diff_y = ((velocity_y[1:-1, 2:] - 2*velocity_y[1:-1, 1:-1] + velocity_y[1:-1, :-2]) / dx**2 +
                 (velocity_y[2:, 1:-1] - 2*velocity_y[1:-1, 1:-1] + velocity_y[:-2, 1:-1]) / dy**2)

        # Interpolate x-velocity for advection
        u_interp = (0.25 * (velocity_x[:-1, 1:-2] +
                            velocity_x[:-1, 2:-1] + velocity_x[ 1:, 1:-2] + velocity_x[ 1:, 2:-1]))

        # Advection terms
        adv_x = u_interp * (velocity_y[1:-1, 2:] - velocity_y[1:-1, :-2]) / (2*dx)
        adv_y = velocity_y[1:-1, 1:-1] * (velocity_y[2:, 1:-1] - velocity_y[:-2, 1:-1]) / (2*dy)

        # Buoyancy term
        buoyancy = RAYLEIGH * PRANDTL * t_at_vy[1:-1, 1:-1]

        # Combined y-velocity update with buoyancy
        vy_star[1:-1, 1:-1] = velocity_y[1:-1, 1:-1] + DT * (PRANDTL * diff_y - adv_x - adv_y + buoyancy)

        # Calculate divergence for pressure correction
        div = np.zeros_like(pressure)
        div = ((vx_star[:, 1:] - vx_star[:, :-1]) / dx +
            (vy_star[1:, :] - vy_star[:-1, :]) / dy)

        # Solve pressure Poisson equation
        pressure = solve_pressure_poisson(div / DT, poisson_matrix)

        # Project velocities (vectorized)
        vx_star[1:-1, 1:-1] -= DT * PRANDTL* (pressure[1:-1, 1:] - pressure[1:-1, :-1]) / dx
```

```
    vy_star[1:-1, 1:-1] -= DT * PRANDTL * (pressure[1:, 1:-1] - pressure[:-1, 1:-1]) / dy

    # Step 6: Update fields for next timestep
    velocity_x = vx_star
    velocity_y = vy_star

    # Step 7: Apply boundary conditions
    # No-slip boundary conditions for top/bottom walls only
    # x-velocity
    velocity_x[0, :] = 0    # Bottom wall
    velocity_x[-1, :] = 0   # Top wall
    # Periodic boundary conditions for side walls
    velocity_x[:, 0] = 0   # Left boundary copies second-to-last column
    velocity_x[:, -1] = 0   # Right boundary copies second column

    # y-velocity
    velocity_y[0, :] = 0    # Bottom wall
    velocity_y[-1, :] = 0   # Top wall
    # Periodic boundary conditions for side walls
    velocity_y[:, 0] = velocity_y[:, 1]   # Left boundary copies last column
    velocity_y[:, -1] = velocity_y[:, -2]   # Right boundary copies first column

    temperature = t_new

    # Calculate vorticity for visualization
    vorticity = calculate_vorticity(velocity_x, velocity_y)

    # Update time
    time += DT

    # Adjust time step based on CFL condition
    max_vx = np.max(np.abs(velocity_x))
    max_vy = np.max(np.abs(velocity_y))
    max_vel = max(max_vx, max_vy, 1e-6)

    Uff = np.sqrt(RAYLEIGH/PRANDTL)
    if max_vel > 1.5*Uff:
        print(f"Warning: velocity {max_vel:.1f} exceeds 1.2Uff ({1.5*Uff:.1f}) possible divergence!")

```

```
    convective_dt = CFL_SAFETY * 0.5 * min(dx, dy) / max_vel
    diffusive_dt = CFL_SAFETY * 0.5 * min(dx, dy)**2 / max(1/PRANDTL, 1)
    return min(convective_dt, diffusive_dt, 0.01), max_vel

def run_simulation():
    global time, DT
    time = 0.0

    # Setup Poisson solver matrix (significant performance boost)
    print("Setting up pressure Poisson solver...")
    poisson_matrix = setup_poisson_solver()

    # Storage for visualization
    save_interval = 1 / 300  # Increased to reduce memory usage and speed up simulation
    next_save = save_interval
    save_step = 1

    print("Starting simulation...")
    start_time = cpu_time.time()
    iteration = 0
    max_vel_old = 0.0  # Initialize max velocity for timestep adjustment

    while time < MAX_TIME:
        iteration += 1
        DT, max_vel = timestep(poisson_matrix)
        max_vel_old = max_vel

        # Save data for visualization at regular intervals
        if time >= next_save:
            # Interpolate velocities to cell centers for visualization
            vx_centered = 0.5 * (velocity_x[:, :-1] + velocity_x[:, 1:])
            vy_centered = 0.5 * (velocity_y[:-1, :] + velocity_y[1:, :])

            history.append({
                'time': time,
                'temperature': temperature.copy(),
                'vorticity': vorticity.copy(),
                'velocity_x': vx_centered.copy(),
                'velocity_y': vy_centered.copy()
            })
            next_save += save_interval
```

```python

            # Print progress
            max_velocity = max(np.max(np.abs(velocity_x)), np.max(np.abs(velocity_y)))
            elapsed = cpu_time.time() - start_time
            print(f"Progress: {time/MAX_TIME*100:.1f}% ({time:.4f}/{MAX_TIME}), RT: {elapsed:.1f}s, ETA: {elapsed/max(time,0.001)*(MAX_TIME-time):.1f}s, Max Vel: {max_velocity}")
        if time >= save_step:
            print("Saving simulation history to file...")
            with open('rayleigh_benard_history.pkl', 'wb') as f:
                pickle.dump(history, f)
            print("History saved successfully to 'rayleigh_benard_history.pkl'")
            create_visualization()
            save_step += 1

    print(f"Simulation completed in {cpu_time.time() - start_time:.2f} seconds. Total frames: {len(history)}")

def create_visualization():
    # Function to compute buoyancy from temperature
    def compute_buoyancy(T):
        T_at_vy = np.zeros((NY+1, NX))
        T_at_vy[1:-1, :] = 0.5 * (T[:-1, :] + T[1:, :])
        T_at_vy[0, :] = T[0, :]
        T_at_vy[-1, :] = T[-1, :]
        return RAYLEIGH * PRANDTL * T_at_vy[1:-1, :]

    # Output settings
    N_FRAMES = 300  # Fixed number of frames
    fps = 30  # Frame rate for GIF

    # Resample history to exactly N_FRAMES
    if len(history) > 1:
        orig_times = np.array([f['time'] for f in history])
        target_times = np.linspace(orig_times[0], orig_times[-1], N_FRAMES)
        idxs = np.searchsorted(orig_times, target_times, side='right') - 1
        idxs = np.clip(idxs, 0, len(history) - 1)
        print(f"Resampling {len(history)} frames to {N_FRAMES} for visualization")
```

```
    else:
        print("Warning: Not enough frames in history for animation")
        idxs = [0] * N_FRAMES

    # Create directories for frames
    os.makedirs("frames_buoy_vort", exist_ok=True)
    os.makedirs("frames_temp_vel", exist_ok=True)

    # Save individual frames for both animations
    for frame_idx in range(N_FRAMES):
        h = history[idxs[frame_idx]]

        # Animation 1: Buoyancy and Vorticity
        fig1, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))

        # Compute frame-specific ranges for vorticity
        vmin_vort, vmax_vort = np.min(h['vorticity']), np.max(h['vorticity'])

        # Plot vorticity with frame-specific colorbar
        im1 = ax1.imshow(h['vorticity'], cmap='coolwarm', origin='lower',
                  extent=[0, WIDTH, 0, HEIGHT], vmin=-500, vmax=500)
        ax1.set_title('Vorticity')
        ax1.set_xlabel('x')
        ax1.set_ylabel('y')
        cbar1 = fig1.colorbar(im1, ax=ax1, orientation='vertical')
        cbar1.set_label(f"Range: [{vmin_vort:.2f}, {vmax_vort:.2f}]")

        # Compute buoyancy and its range for this frame
        buoyancy = compute_buoyancy(h['temperature'])
        vmin_buoy, vmax_buoy = np.min(buoyancy), np.max(buoyancy)

        # Plot buoyancy with frame-specific colorbar
        im2 = ax2.imshow(buoyancy, cmap='coolwarm', origin='lower',
                  extent=[0, WIDTH, 0, HEIGHT], vmin=vmin_buoy,
                  ↪ vmax=vmax_buoy)
        ax2.set_title('Buoyancy')
        ax2.set_xlabel('x')
        ax2.set_ylabel('y')
        cbar2 = fig1.colorbar(im2, ax=ax2, orientation='vertical')
        cbar2.set_label(f"Range: [{vmin_buoy:.2f}, {vmax_buoy:.2f}]")

        fig1.suptitle(f"Rayleigh-Bénard Convection: t = {h['time']:.3f}")
```

```
        plt.tight_layout()

        fname1 = f"frames_buoy_vort/frame_{frame_idx:04d}.png"
        plt.savefig(fname1, dpi=120)
        plt.close(fig1)

        # Animation 2: Temperature and Velocity Magnitude
        fig2, (ax3, ax4) = plt.subplots(1, 2, figsize=(14, 5))

        # Compute frame-specific ranges for temperature
        vmin_temp, vmax_temp = np.min(h['temperature']), np.max(h['temperature'])

        # Plot temperature with frame-specific colorbar
        im3 = ax3.imshow(h['temperature'], cmap='coolwarm', origin='lower',
                    extent=[0, WIDTH, 0, HEIGHT], vmin=vmin_temp, vmax=vmax_temp)
        ax3.set_title('Temperature')
        ax3.set_xlabel('x')
        ax3.set_ylabel('y')
        cbar3 = fig2.colorbar(im3, ax=ax3, orientation='vertical')
        cbar3.set_label(f"Range: [{vmin_temp:.2f}, {vmax_temp:.2f}]")

        # Compute velocity magnitude and its range for this frame
        vel_mag = np.sqrt(h['velocity_x']**2 + h['velocity_y']**2)
        vmin_vel, vmax_vel = np.min(vel_mag), np.max(vel_mag)

        # Plot velocity magnitude with frame-specific colorbar
        im4 = ax4.imshow(vel_mag, cmap='coolwarm', origin='lower',
                    extent=[0, WIDTH, 0, HEIGHT], vmin=vmin_vel, vmax=vmax_vel)
        ax4.set_title('Velocity Magnitude')
        ax4.set_xlabel('x')
        ax4.set_ylabel('y')
        cbar4 = fig2.colorbar(im4, ax=ax4, orientation='vertical')
        cbar4.set_label(f"Range: [{vmin_vel:.2f}, {vmax_vel:.2f}]")

        # Add velocity field arrows (downsampled for clarity)
        X, Y = np.meshgrid(np.linspace(0, WIDTH, NX), np.linspace(0, HEIGHT, NY))
        skip = max(1, min(NX, NY) // 20)  # Adjust skip factor based on grid size
        ax4.quiver(X[::skip, ::skip], Y[::skip, ::skip],
```

```python
                 h['velocity_x'][::skip, ::skip], h['velocity_y'][::skip, ::skip],
                 color='white', scale=max(vmax_vel*3, 0.1), width=0.002)

        fig2.suptitle(f"Rayleigh-Bénard Convection: t = {h['time']:.3f}")
        plt.tight_layout()

        fname2 = f"frames_temp_vel/frame_{frame_idx:04d}.png"
        plt.savefig(fname2, dpi=120)
        plt.close(fig2)

        if frame_idx % 10 == 0:
            print(f"Saved frame {frame_idx}/{N_FRAMES}")

    print("Frames saved to 'frames_buoy_vort' and 'frames_temp_vel' directories")

    # Create GIF from buoyancy/vorticity frames
    print("Creating buoyancy/vorticity GIF animation...")
    with imageio.get_writer('rayleigh_benard_buoy_vort.gif', fps=fps, mode='I') as writer:
        for frame_idx in range(N_FRAMES):
            fname = f"frames_buoy_vort/frame_{frame_idx:04d}.png"
            if os.path.exists(fname):
                image = imageio.imread(fname)
                writer.append_data(image)
    print(f"Buoyancy/vorticity animation saved as GIF ({N_FRAMES} frames @ {fps}fps)")

    # Create GIF from temperature/velocity frames
    print("Creating temperature/velocity GIF animation...")
    with imageio.get_writer('rayleigh_benard_temp_vel.gif', fps=fps, mode='I') as writer:
        for frame_idx in range(N_FRAMES):
            fname = f"frames_temp_vel/frame_{frame_idx:04d}.png"
            if os.path.exists(fname):
                image = imageio.imread(fname)
                writer.append_data(image)
    print(f"Temperature/velocity animation saved as GIF ({N_FRAMES} frames @ {fps}fps)")
```

```python
def load_and_visualize():
    import pickle

    print("Loading simulation history from file...")
    with open('rayleigh_benard_history(Ra=1e5).pkl', 'rb') as f:
        global history
        history = pickle.load(f)

    print(f"Loaded {len(history)} frames from history file")
    create_visualization()
    print("Visualization complete!")

print(f"Simulation parameters:\n GRID SIZE: {NX} x {NY}\n RAYLEIGH NUMBER: {RAYLEIGH}\n PRANDTL NUMBER: {PRANDTL}\n MAX TIME: {MAX_TIME}\n Safety factor: {CFL_SAFETY}\n Perturbation: {perturbation}")

# Modify the main block to add an option to load from file
if __name__ == "__main__":
    import sys
    if len(sys.argv) > 1 and sys.argv[1] == '--load':
        load_and_visualize()
    else:
        run_simulation()
```

7 References

- **Woods Hole Oceanographic Institution.** *Rayleigh–Bénard Convection – WHOI Report.* https://www.whoi.edu/fileserver.do?id=21393&pt=10&p=17277
- **Rahman, A., Ali, M., Ahmed, K.K.** *A Review on Rayleigh–Bénard Convection*, The Scientific World Journal, 2014. https://pmc.ncbi.nlm.nih.gov/articles/PMC4058157/pdf/TSWJ2014-786102.pdf
- **Wikipedia.** *Rayleigh–Bénard Convection.* https://en.wikipedia.org/wiki/Rayleigh%E2%80%93B%C3%A9nard_convection
- **Charles University.** *Notes on Rayleigh–Bénard Convection.* https://geo.mff.cuni.cz/~matyska/AMC_21.pdf
- **ScienceDirect.** *Rayleigh–Bénard Convection Overview.* https://www.sciencedirect.com/topics/physics-and-astronomy/rayleigh-benard-convection
- **Wikipedia.** *Marker-and-Cell Method.* https://en.wikipedia.org/wiki/Marker-and-cell_method
- **D. H. Rothman, MIT, October 24, 2022.** *Rayleigh–Bénard Study PDF.* Lecture notes for 12.006J/18.353J/2.050J, Nonlinear Dynamics: Chaos