Project SG2212/SG3114

Development/assessment of a 2D Navier–Stokes solver

due March 24 2024, 23:59

1 Introduction

Your task for this project is to develop a simulation code that solves the incompressible Navier–Stokes equations in a rectangular domain and to perform a comparison with the open-source finite-volume code OpenFOAM. For your help you can download Matlab and Python templates on the course homepage. A tutorial on how to install and use OpenFOAM was given during the lectures and is available on Canvas[1].

1.1 Simulation code

The code will include the fundamental parts of a typical CFD software; parts that you worked on in the homework problems:

- solution of the incompressible Navier–Stokes equations (and the scalar transport equation for PhD students)
- second-order finite differences on a staggered grid;
- projection method with explicit Euler time discretisation;
- sparse matrices for the Poisson equation for the pressure;
- Dirichlet and Neumann boundary conditions on a rectangular domain in two dimensions (2D);
- direct and iterative solvers.

Since this code is used for educational purposes it does not include some of the advanced features which are commonly found in professional CFD software such as:

- 3D formulation;
- implicit time stepping for the viscous term;
- CFL condition for time stepping;
- higher order time integration methods;
- advanced high order (spectral) spatial discretisation methods;
- complex geometries, multiblock distributions, mappings *etc.*

In order to demonstrate the capabilities of the code we will consider two physical flow cases: the lid-driven cavity and the Rayleigh–Bénard convection (RB is compulsory for PhD students only), which are introduced in sections 2.1 and 3.

[1] `https://canvas.kth.se/courses/31845`

1.2 Introduction to the numerical discretisation

The Navier–Stokes equations for an incompressible fluid can be written in nondimensional form as:

$$\frac{\partial u_i}{\partial t} + \frac{\partial (u_i u_j)}{\partial x_j} = -\frac{\partial p}{\partial x_i} + \frac{1}{Re}\frac{\partial^2 u_i}{\partial x_j \partial x_j} + f_i, \tag{1a}$$

$$\frac{\partial u_i}{\partial x_i} = 0. \tag{1b}$$

Here u_i is the velocity component in the direction x_i, p is the pressure, f_i is a volume force and Re is the Reynolds number. Note that the summation convention has been used.

The corresponding nondimensional transport equation for a scalar θ, such as the temperature field or the concentration of a pollutant, reads:

$$\frac{\partial \theta}{\partial t} + \frac{\partial (u_j \theta)}{\partial x_j} = \frac{1}{Pe}\frac{\partial^2 \theta}{\partial x_j \partial x_j}, \tag{2}$$

where $Pe = PrRe$ is the Péclet number and Pr the Prandtl number. In the case of a so-called *active scalar*, the Boussinesq approximation can be used to model the influence of variable density onto the momentum equation by setting $f_2 = \theta Ri$, where Ri is the Richardson number. Here we assume that the gravity acts in the $y = x_2$ direction, hence this force acts along x_2.

After temporal and spatial discretisation, having chosen the explicit Euler method for the time derivative, the system of equations (1) becomes

$$\frac{\underline{u}^{n+1} - \underline{u}^n}{\Delta t} + \underline{\underline{N}}(\underline{u}^n, \underline{u}^n) - \frac{1}{Re}\underline{\underline{L}}\,\underline{u}^n + \underline{\underline{G}}\,\underline{p}^{n+1} = \underline{f}^n, \tag{3a}$$

$$\underline{\underline{D}}\,\underline{u}^{n+1} = 0, \tag{3b}$$

where $\underline{u}$ denotes the velocity *vector*, $\underline{\underline{N}}(\cdot,\cdot)$ represents the discretised nonlinear advection operator, $\underline{\underline{L}}$ the discretised laplacian operator, $\underline{\underline{G}}$ the discretised gradient, $\underline{\underline{D}}$ the discretised divergence, $\underline{f}$ the discretised forcing terms including the boundary conditions, and Δt the time step. Note that the pressure field is treated implicitly, i.e. using the values at time step $n+1$.

An efficient method for the solution of the incompressible discretised equations (3) is the projection method, which consists of the following steps:

(i) in the first half-step an intermediate, non divergence-free, velocity field $\underline{u}^*$ is computed by solving the following equation convection-diffusion:

$$\frac{\underline{u}^* - \underline{u}^n}{\Delta t} + \underline{\underline{N}}(\underline{u}^n, \underline{u}^n) - \frac{1}{Re}\underline{\underline{L}}\,\underline{u}^n = \underline{f}^n. \tag{4}$$

(ii) the second half-step consists in the solution of the following problem with both velocity and pressure as unknowns:

$$\frac{\underline{u}^{n+1} - \underline{u}^*}{\Delta t} + \underline{\underline{G}}\,\underline{p}^{n+1} = 0, \tag{5a}$$

$$\underline{\underline{D}}\,\underline{u}^{n+1} = 0. \tag{5b}$$

Applying the divergence operator $\underline{\underline{D}}$ to equation (5a) and using the divergence-free condition (5b), the following Poisson equation for the pressure at step $n+1$ is obtained:

$$\underline{\underline{D}}\,\underline{\underline{G}}\,\underline{p}^{n+1} = \underline{\underline{L}}\,\underline{p}^{n+1} = \frac{1}{\Delta t}\,\underline{\underline{D}}\,\underline{u}^{*}. \tag{6}$$

The pressure computed from this step can be inserted in equation (5a) to obtain $\underline{u}^{n+1}$:

$$\underline{u}^{n+1} = \underline{u}^{*} - \Delta t\,\underline{\underline{G}}\,\underline{p}^{n+1}. \tag{7}$$

The solution of the scalar transport equation (2) is simpler and can be done in a straight-forward way using explicit time integration (velocity field from the previous time step).

2 Problem description

2.1 The 2D lid-driven cavity

Consider the incompressible flow in a two-dimensional rectangular domain $\Omega := [0, l_x] \times [0, l_y]$. The side and lower walls ($\Gamma_\mathrm{l} := \{0\} \times [0, l_y]$, $\Gamma_\mathrm{r} := \{l_x\} \times [0, l_y]$ and $\Gamma_\mathrm{b} := [0, l_x] \times \{0\}$) are all solid and still, requiring no-slip boundary conditions. The top wall ($\Gamma_\mathrm{t} := [0, l_x] \times \{l_y\}$) is moving with a constant speed $u = u_{\mathbf{lid}}$ in the positive x-direction. This flow case is a common test problem for Navier–Stokes solvers, and usually called the *lid-driven cavity*. A sketch is given in Figure 1.

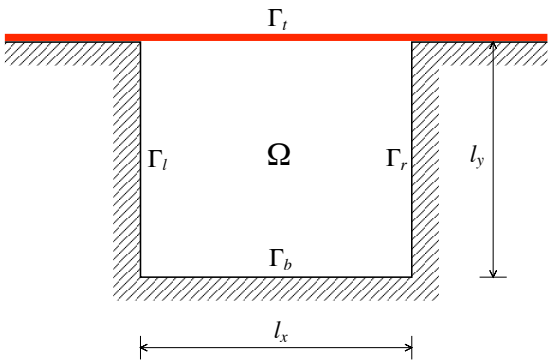

Figure 1: Sketch of the lid-driven cavity. The moving top wall is indicated in red.

The continuous velocity-pressure formulation for the lid-driven cavity problem becomes:

$$\frac{\partial u}{\partial t}+\frac{\partial p}{\partial x}=-\frac{\partial(u^2)}{\partial x}-\frac{\partial(uv)}{\partial y}+\frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right), \quad (x,y)\in\Omega,\ t>0 \tag{8a}$$

$$\frac{\partial v}{\partial t}+\frac{\partial p}{\partial y}=-\frac{\partial(uv)}{\partial x}-\frac{\partial(v^2)}{\partial y}+\frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2}+\frac{\partial^2 v}{\partial y^2}\right), \quad (x,y)\in\Omega,\ t>0, \tag{8b}$$

$$\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}=0, \quad (x,y)\in\Omega,\ t>0, \tag{8c}$$

$$u=v=0, \quad (x,y)\in\Gamma_\mathrm{l}\cup\Gamma_\mathrm{r}\cup\Gamma_\mathrm{b},\ t>0, \tag{8d}$$

$$u=1,\ v=0, \quad (x,y)\in\Gamma_t,\ t>0, \tag{8e}$$

$$\frac{\partial p}{\partial n}=0, \quad (x,y)\in\Gamma_t\cup\Gamma_\mathrm{l}\cup\Gamma_\mathrm{r}\cup\Gamma_\mathrm{b},\ t>0, \tag{8f}$$

$$u=0,\ v=0 \quad (x,y)\in\Omega,\ t=0. \tag{8g}$$

where n in the Neumann boundary conditions for pressure is the normal direction to the boundary. Note that the pressure is only defined up to an additive constant (only the gradient of the pressure enters the Navier–Stokes equations), hence as Neumann boundary conditions are used the solution for pressure will not be unique. The finite differences differential matrix for the Laplacian becomes singular, and there is no numerical solution to the problem. One workaround for this is to set the pressure explicity in one point of the domain. In the Matlab template that you are given this is done by imposing a Dirichlet boundary condition $p=0$ in one of the corner points.

The initial conditions $u=0$ and $v=0$ indicates that the flow is initially at rest.

For the PhD part of the project the advection of a passive scalar should also be considered, it is governed by

$$\frac{\partial\theta}{\partial t}=-\frac{\partial(u\theta)}{\partial x}-\frac{\partial(v\theta)}{\partial y}+\frac{1}{Pe}\left(\frac{\partial^2\theta}{\partial x^2}+\frac{\partial^2\theta}{\partial y^2}\right), \quad (x,y)\in\Omega,\ t>0\ , \tag{9a}$$

$$\text{(adiabatic)}\quad \frac{\partial\theta}{\partial x}=0, \quad (x,y)\in\Gamma_\mathrm{l}\cup\Gamma_\mathrm{l},\ t>0, \tag{9b}$$

$$\theta=0, \quad (x,y)\in\Gamma_\mathrm{b},\ t>0, \tag{9c}$$

$$\theta=1, \quad (x,y)\in\Gamma_\mathrm{t},\ t>0, \tag{9d}$$

$$\theta=y/l_y, \quad (x,y)\in\Omega,\ t=0. \tag{9e}$$

with $Pe=Re\,Pr$. Choose $Pr=0.71$ which is a good approximation for air. The initial condition $\theta=y/l_y$ assumes a linear profile in the vertical direction.

Your task is to write a Matlab/Python code which solves the Navier–Stokes equations for the flow case described above using the projection method on a staggered grid. The implementation of the scalar equation is only compulsive for PhD students (course SG3114). Run your code for the flow cases A–D given below and answer questions Q1–Q6.

Flow cases:

A) **Matlab/Python code:** Run the code for $Re=25$ and $l_x=l_y=1$. Use a total of $N_x=N_y=30$ grid nodes and a time step $\Delta t=0.001$. You need to integrate up to the point that the solution becomes steady.

For low values of Re, the flow field is nearly symmetric with respect to the vertical midline $x = l_x/2$ (see figure 2) with very little transient behaviour (the stability requirements of the code are discussed in Q2).

B) **Matlab/Python code:** Run the code for $Re = 250$, $l_x = l_y = 1$, $N_x = N_y = 50$, and $\Delta t = 0.001$.

For this moderate value of Re the flow becomes asymmetric due to the action of inertial forces but it is still steady (see figure 3). This can be clearly seen by means of a time series recorded in the middle of the domain (see question Q4).

C) **Matlab/Python code:** Run the code for $Re = 5000$ or even higher, $l_x = l_y = 1$, $N_x = N_y = 100$, $\Delta t = 0.001$.

Note that increasing the Reynolds number will give rise to thinner boundary layers which require more grid points to be properly resolved. You can experiment with different combinations of Re, N_x, N_y and Δt and report how the results (and the stability) are affected (see Q2).

D) **OpenFOAM:** Repeat Case C using OpenFOAM, with same domain size and resolution.

You can start from the test case "cavity" discussed during the tutorial, which is located in *OpenFOAM/OpenFOAM-7/tutorials/incompressible/icoFoam/cavity/cavity*. Make sure that the resolution and Reynolds number are matching. Remember: space resolution and domain size are specified in the input file *blockMeshDict* in the folder *system*, while the viscosity is specified in the input file *transportProperties* in the folder *constant*. (*Hint:* What is the actual size of the domain in the test case? If in doubt about the domain size, use ParaView to visualize the mesh!).

Questions:

Q1) Discuss why the `kron` operator (Kronecker tensor product, introduced in Section A.4) returns the matrix for the second derivative in the two-dimensional case.

Q2) For case A, determine "experimentally" the maximum time step Δt for which the simulation is stable. Can you relate that maximum time step to the grid resolution and Reynolds number by considering stability conditions for both convective and diffusive terms? (*Hint:* you must consider a 2D problem!). Which of the two is more strict for cases A–C?
For PhD students: Choosing $Pr = 0.71$, is the scalar equation less stable than the momentum equations? Why?

Q3) The boundary conditions for $\underline{u}$ on the top and bottom boundaries as well as for $\underline{v}$ on the left and right boundaries should be imposed by choosing the correct values for the ghost nodes. Write the expressions for $\underline{u}_{i+\frac{1}{2},\,0}$ and $\underline{v}_{0,\,j+\frac{1}{2}}$ at the ghost nodes.

The values for the pressure at the ghost nodes $\underline{p}_{0,\,j}$ and $\underline{p}_{i,\,0}$ should also be chosen such that the Neumann boundary condition at the boundaries is satisfied. Write the expression for the pressure values on the ghost nodes and for the discretised Laplace operator $\underline{\underline{L}}\,\underline{p}_{1,\,j}$ for $j = 2, \cdots, N_y - 1$ (remember to include the boundary conditions!).

Q4) Place a probe at the domain centre and compare the time history of the horizontal velocity(u) for flow cases A–C. Try to estimate the time it takes for each run to establish a steady solution. How does this time correlate with the Reynolds number?

Q5) Compare the velocity profile along the centre plane that you obtained with OpenFOAM and your Matlab/Python code (use the filter "plot-over-line" in ParaView, save the data and import them in Matlab/Python to compare the profiles).

Q6) In your Matlab/Python code, implement a moving lower wall using appropriate boundary conditions. Try at least one case at $Re = 100$ with the lower wall moving with the same velocity as the upper one. Include the plot of the final velocity field, and discuss the symmetry properties of the resulting flow.

You should hand in your Matlab/Python code and a report with answers to questions Q1–Q6 including a discussion and plots illustrating the flow development for the four cases A–D above. To have an idea of how of the flow behaves you can look at figure 2 for case A ($Re = 20$), figure 3 for case B ($Re = 200$) and figure 4 for case C ($Re = 4000$).

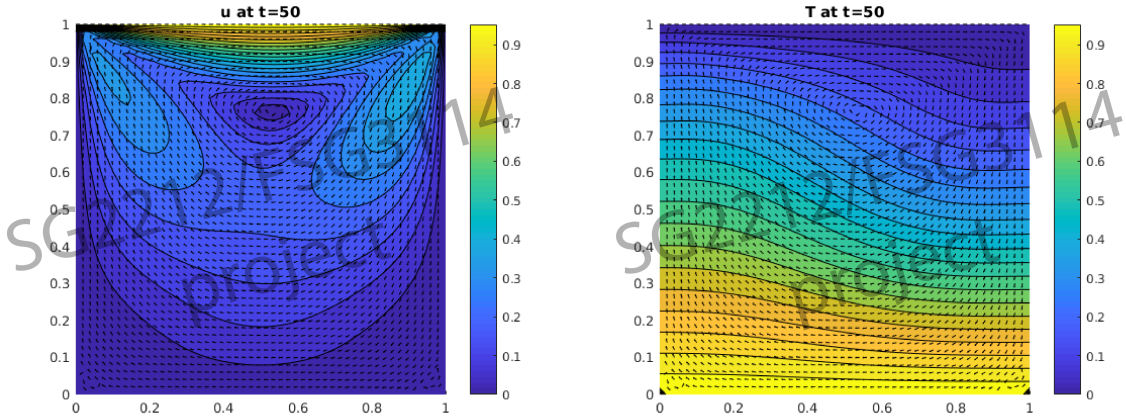

Figure 2: Simulation results for the lid-driven cavity, case A ($Re = 20$). Velocity field *(left)* and temperature field *(right)* at $t = 50$.

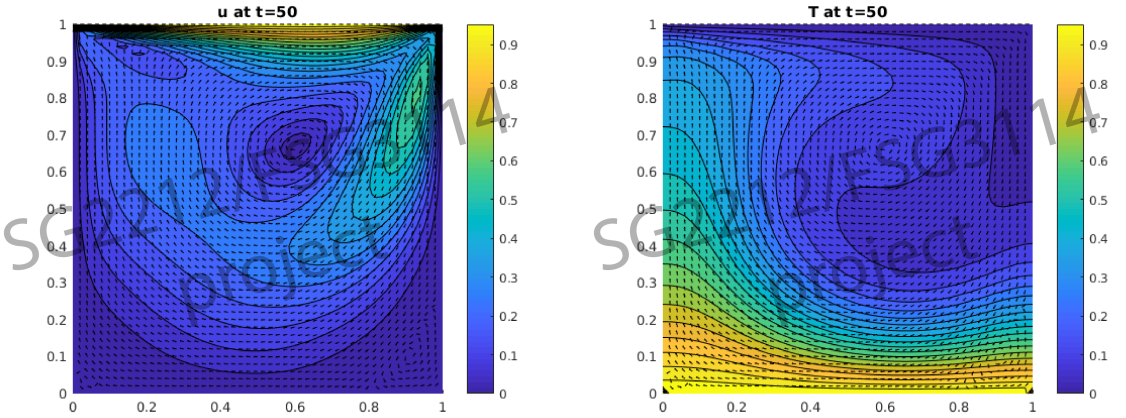

Figure 3: Simulation results for the lid-driven cavity, case B ($Re = 200$). Velocity field *(left)* and temperature field *(right)* at $t = 50$.

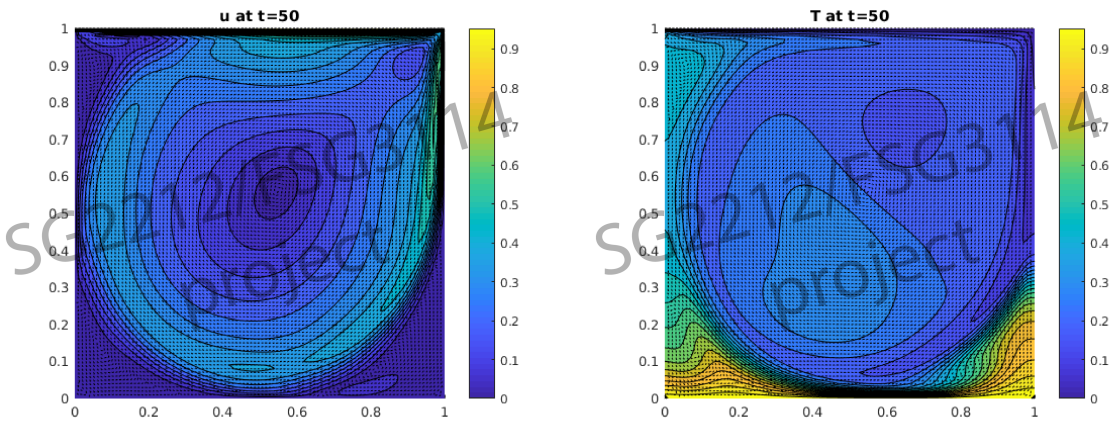

Figure 4: Simulation results for the lid-driven cavity, case C ($Re = 4000$). Velocity field *(left)* and temperature field *(right)* at $t = 50$.

3 Task 2: Rayleigh-Bénard problem (only for PhD students SG3114)

The flow induced by small density variations due to temperature differences between two walls is called Rayleigh-Bénard convection. The control parameter is the so-called Rayleigh number Ra which describes the ratio of buoyancy to viscous forces acting on the flow. The Rayleigh number is defined as

$$Ra = \frac{g\alpha\Delta T}{\nu U_\kappa/h^2} = \frac{\text{buoyancy}}{\text{viscosity}} \ , \tag{10}$$

with α being the thermal expansion coefficient, g the gravitational acceleration, h the height and ΔT the temperature difference. The convective velocity scale is

$$U_\kappa = \kappa/h \ , \tag{11}$$

with the thermal diffusivity κ. Depending on Ra, various flow regimes can be studied, ranging from stable flow configuration for low Ra, two-dimensional convection rolls (*i.e.* large counter-rotating vortices, see below), hexagonal convection cells, and finally for high Ra fully turbulent flow. The major relevance for us to study this flow is that using comparably simple tools from linear stability analysis the changeover from stable to unstable (*i.e.* amplifying) flow can be exactly determined. To numerically check this so-called critical Rayleigh number Ra_c provides a very useful validation of the numerical code.

If the flow field is subjected to a temperature gradient between the two walls, density variation in the flow may be generated. These density variations, due to buoyancy forces, induce motion in the fluid. The corresponding force in the momentum equations can be modelled by means of the Boussinesq approximation. In order to tackle this flow problem, the governing equations (1) are thus coupled to the temperature equation (2) using the forcing term f_2. Then, a slightly different non-dimensionalisation has to be employed, essentially based on the convective velocity scale instead of the lid velocity. The only non-dimensional numbers in the equations are the Rayleigh number Ra and the Prandtl number $Pr = \nu/\kappa$. For a two-dimensional flow the final non-dimensional equations can be written as

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \ , \tag{12}$$

$$\frac{\partial u}{\partial t} + \frac{\partial P}{\partial x} = -\frac{\partial (u^2)}{\partial x} - \frac{\partial (uv)}{\partial y} + Pr\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \ , \tag{13}$$

$$\frac{\partial v}{\partial t} + \frac{\partial P}{\partial y} = -\frac{\partial (uv)}{\partial x} - \frac{\partial (v^2)}{\partial y} + Pr\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) + f_2 \ , \tag{14}$$

$$\frac{\partial \theta}{\partial t} = -\frac{\partial (u\theta)}{\partial x} - \frac{\partial (v\theta)}{\partial y} + \left(\frac{\partial^2 \theta}{\partial x^2} + \frac{\partial^2 \theta}{\partial y^2}\right) \ . \tag{15}$$

The forcing in the normal direction, caused by buoyancy forces, is

$$f_2 = RaPr\theta \ . \tag{16}$$

The original formulation of the Rayleigh-Bénard convection problem is considered as the flow between two infinite parallel walls, see the sketch in Fig. 6. However, for the present project, we consider a two-dimensional cavity where the top and bottom walls are kept at different

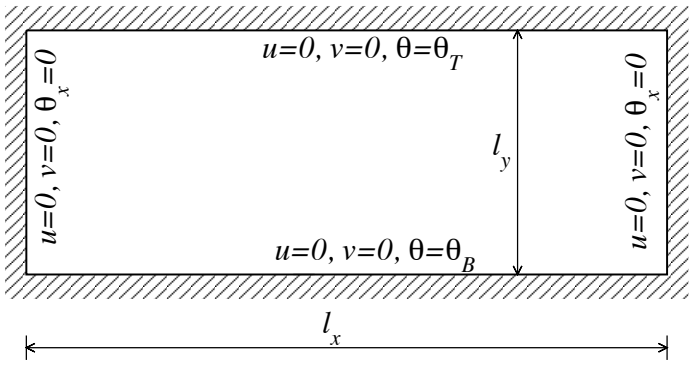

Figure 5: Closed cavity with non-isothermal walls.

temperatures and the side-walls are assumed to be adiabatic ($\frac{\partial\theta}{\partial n} = 0$). A sketch of the geometry is given in Figure 5. Here, the boundary conditions are

$$x = 0: \qquad u = v = 0, \qquad \frac{\partial\theta}{\partial x} = 0 \ , \tag{17}$$

$$x = l_x: \qquad u = v = 0, \qquad \frac{\partial\theta}{\partial x} = 0 \ , \tag{18}$$

$$y = 0: \qquad u = v = 0, \qquad \theta = \theta_B \ , \tag{19}$$

$$y = l_y: \qquad u = v = 0, \qquad \theta = \theta_T \ . \tag{20}$$

Initial conditions are at $t = 0$ non-moving fluid, *i.e.* $u(t = 0) = v(t = 0) = 0$. For the scalar, assume a linear profile in the vertical direction,

$$\theta(x, y, t = 0) = \theta_B + \frac{y}{l_y}(\theta_T - \theta_B) \ . \tag{21}$$

For cases where $\theta_T > \theta_B$, the density is decreasing with increasing height. This corresponds to a state with is called *stable stratification*, which means that the flow is stable and all disturbances will decay eventually. However, if $\theta_T < \theta_B$, corresponding to *unstable stratification*, for large enough values of Ra the flow becomes unstable. A stability diagram for this case is shown in Fig. 6. It can be seen that the first instability appears for $Ra_c = 1708$ with a wavenumber $K_c = 3.12$, which corresponds to a wavelength of $\lambda_c = 2.01$.

Your task is to include the forcing f_2 and the energy equation into the code developed in Task 1 and run it for cases A and B given below. Your report should discuss the results for these cases together with answers to questions Q1–Q3.

For all runs, choose the Prandtl number $Pr = 0.71$, the box size $l_x = 10$, $l_y = 1$, and the resolution $N_x = 200$, $N_y = 20$. The domain thus becomes elongated in the x-direction, imitating an infinite domain. As initial conditions random small-scale noise of moderate amplitude ($\pm 0.1(\theta_B - \theta_T)$) should be used in order to trigger the instability. The time step Δt should be chosen such that the time integration remains stable.

Flow cases:

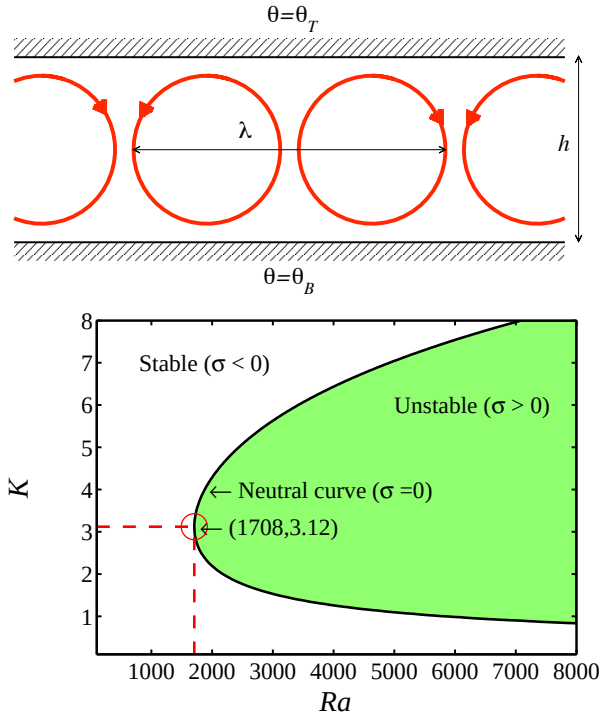

Figure 6: *Top:* Sketch of the flow structure (convection cells). *Bottom:* Stable and unstable (shaded) regions for Rayleigh-Bénard problem. Here, $\lambda = 2\pi/K$ is the wavelength of the unstable perturbation. The critical Rayleigh number is $Ra_c = 1708$ with associated spatial wavenumber $K = 2\pi/\lambda_c = 3.12$ ($\lambda_c = 2.01$).

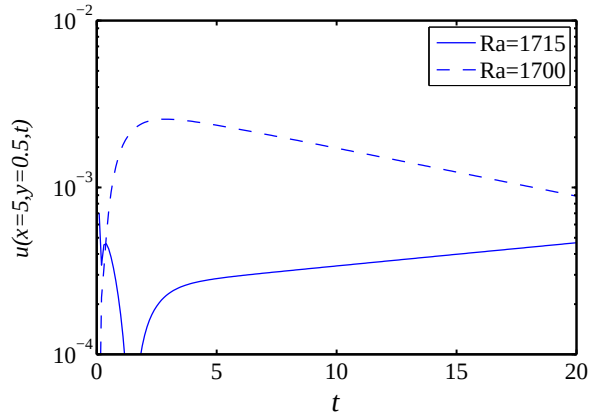

Figure 7: Temporal evolution of the probe signals for two Rayleigh numbers close to the neutrally stable state ($Ra_c = 1708$). It can clearly be seen that the supercritical trajectory leads to exponential growth, whereas the subcritical case eventually will exponentially decay.

A) Run the code for $Ra = 200, 2000, 60000$, $\theta_T = 1$ and $\theta_B = 0$.

These parameters correspond to stable flow. You should observe that initial disturbances will decay due to viscosity. For the case of $Ra = 60000$ the appropriate time step must

be selected.

B) Run the code for $\theta_T = 0$ and $\theta_B = 1$.

Here, if the value of Rayleigh number is large enough ($Ra > 1708$) flow instability will be observed. Run the code for different values of Ra and try to find "experimentally" the critical value Ra_c. To determine the stability of a specific run, consider the time evolution of a probe in the flow, see Fig. 7. If you observe exponential growth (solid line in Fig. 7), you are in the unstable regime, otherwise the flow is stable.

Questions:

Q1) Give the definitions of the non-dimensional parameters Pr and Ra both in terms of mathematical symbols and words (physical meaning). Why do the equations not contain any Reynolds number?

Q2) Estimate the wavelength of the instability at $Ra = 1715$. What boundary conditions should be employed (both for velocity and temperature) to exactly reproduce the critical Rayleigh number?

Q3) Based on time signals such as the example given in Figure 7, try to estimate the critical Rayleigh number for your specific discretisation (see description for run B).

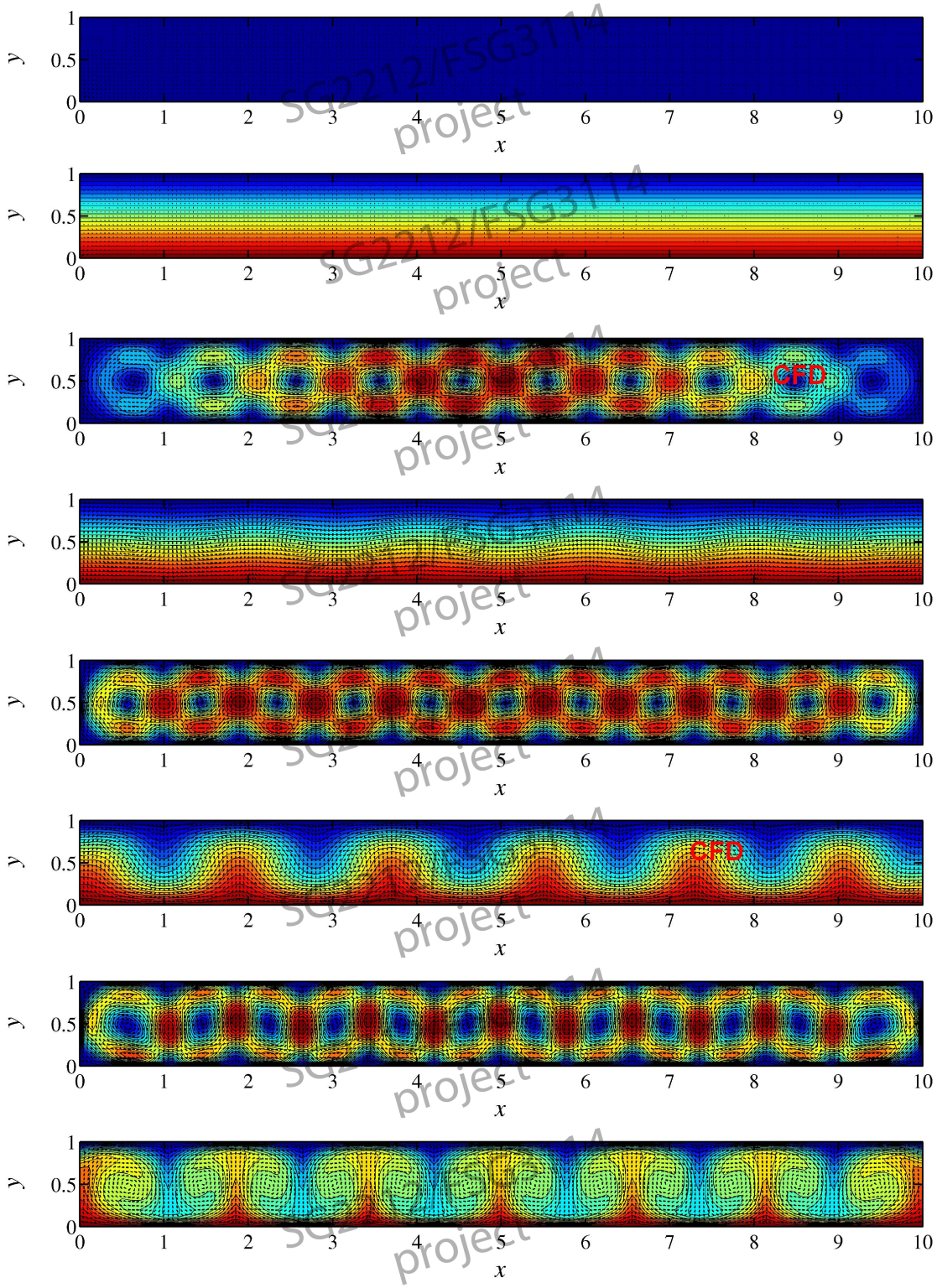

Figure 8: Stationary flow patterns for large times corresponding to $Ra = 1700$, $Ra = 1715$, $Ra = 2700$ and $Ra = 50000$ (from top to bottom). Both the velocity distribution and the temperature field are shown for each Rayleigh number.

A Solution procedure and implementation

A.1 Grid

In order to avoid spurious checkerboard solution the equations are discretized on a staggered grid. Figure 9 shows the numerical domain where the pressure p is defined in cell centres and the velocities u and v are defined in the centre of the vertical and horizontal cell faces, respectively. Table 1 gives the number of unknowns for each of the flow variables to solve for. Here, N_x and

Table 1: Resolution for the various solution variables.

Variable	Interiour resolution	Boundary conditions included
P	$N_x \times N_y$	$(N_x + 2) \times (N_y + 2)$
U	$(N_x - 1) \times N_y$	$(N_x + 1) \times (N_y + 2)$
V	$N_x \times (N_y - 1)$	$(N_x + 2) \times (N_y + 1)$

N_y are the number of cells in the x and y directions, respectively. The coordinates of the grid points (cell corners) are given as

$$X_i = i\frac{l_x}{N_x} \ , \quad i = 0, \cdots, N_x \ , \tag{22}$$

$$Y_j = j\frac{l_y}{N_y} \ , \quad j = 0, \cdots, N_y \ , \tag{23}$$

where l_x and l_y are the lengths in x and y directions. Note that number of grid points (including the boundary points) is $N_x + 1$ and $N_y + 1$, respectively.

A.2 Boundary conditions

A.2.1 Velocity

Boundary conditions for both velocity components are given all around the rectangular domain. In the following, these vectors are denoted U_S for U at the lower boundary (south), V_W for V on the left boundary (west), *etc.* Thus, on each edge a total of $(N_x + 1)$ or $(N_y + 1)$ boundary values are specified on the cell corners. The Dirichlet boundary condition for U on the left and the right boundaries of computational domain can be directly applied as the velocity nodes lie on those boundaries, see Fig. 9. The same is valid for V on the top and the bottom boundaries. However, the boundary conditions for U on the top and bottom boundaries should be imposed by choosing the correct values for the dummy cells (cells outside the domain) in such a way that linear interpolation over the boundary will give the correct value at the boundary. The formulation of the boundary conditions is asked in Q3.

A.2.2 Pressure

As it was shown during the lecture, we have the choice of specifying homogeneous Neumann boundary conditions for the pressure. In other words the normal derivative at the boundaries

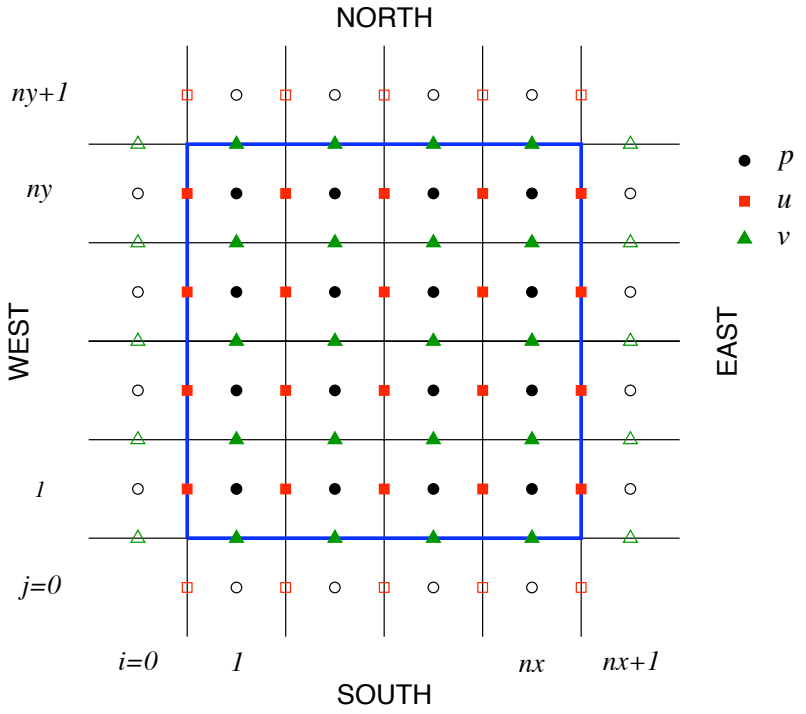

Figure 9: Sketch of the staggered grid. Circles are pressure nodes, squares U-velocity, and triangles V-velocity. Filled symbols correspond to interiour points, whereas open symbols represent the dummy values. The domain boundary is indicated by the thick blue line, on which also the boundary conditions are given.

should vanish for the pressure, *i.e.*,

$$\frac{\partial P}{\partial n} = 0 \ . \tag{24}$$

Again, the value for the pressure dummy cells has to be chosen appropriately to fulfill this condition. The discrete formulation of this boundary condition should also be given as part of Q3.

A.3 Data structure

All variables are stored as matrices in MATLAB/Python. Note that the first index in MATLAB denotes the row and the second index the column, which means that the direction of increasing x is from top to bottom in the matrices. Similarly, increasing the y direction is from left to right in the matrix. Therefore, the storage of variables in the matrices corresponds to the *transpose* of the "physical" domain.

We introduce matrices `U` and `V` which only contain the unknown velocity values at the inner cells, and not the (known) boundary values. Thus, the size for the matrix `U` containing the u velocity is $(N_x - 1) \times N_y$. Similarly, `V` containing the v velocity has $N_x \times (N_y - 1)$ elements.

At time $t = 0$ the velocities can be initiated with zeros (*i.e.* fluid at rest) in MATLAB as

```
1 U=zeros(nx-1,ny);
```

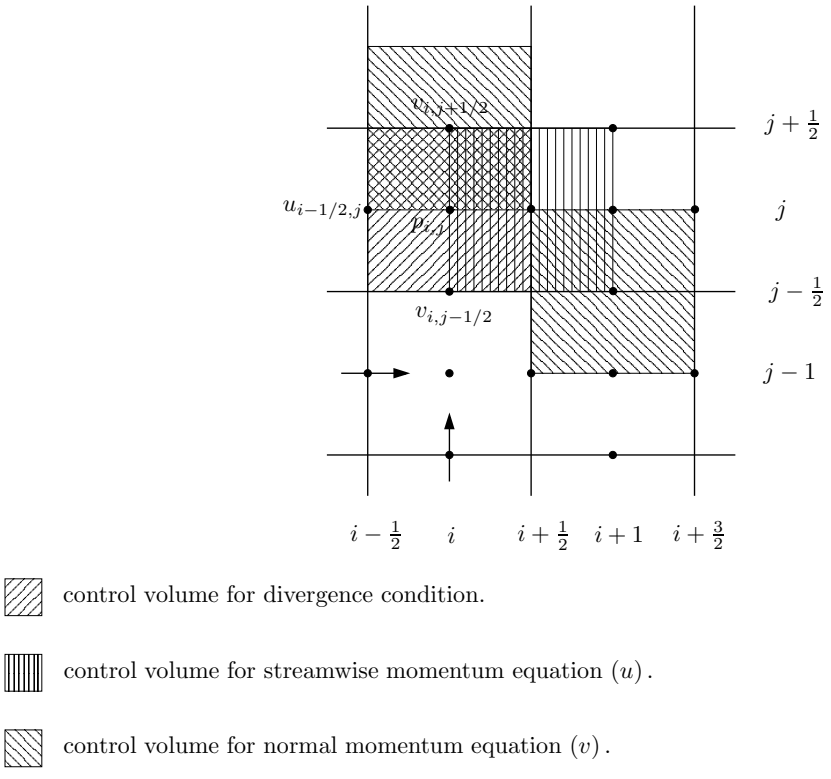

Figure 10: Sketch of the staggered grid. Open symbols indicate dummy cells situated outside the computational domain.

```
V=zeros(nx,ny-1);
```

or in Python:

```
U = np.zeros((Nx-1,Ny))
V = np.zeros((Nx,Ny-1))
```

To perform central difference over the inner points, we need to include the boundary points in `U` and `V`. These *extended* matrices are called `Ue` and `Ve`. They can be created by adding the appropriate rows and columns to the `U` and `V` matrices. For example for the u velocity this is done as follows:

1) Add vectors `uE` and `uW` containing the boundary values at the right and left boundaries of the physical domain, respectively (transposed in MATLAB). In MATLAB code:

```
Ue=[uW ; U ; uE];
```

or in Python:

```
Ue = np.vstack((uW, U, uE))
```

2) Introduce dummy cells with values which give the correct boundary conditions on the top and bottom boundaries of the physical domain. The boundary values are stored in vectors `uN` and `uS` as described above.

   In MATLAB:

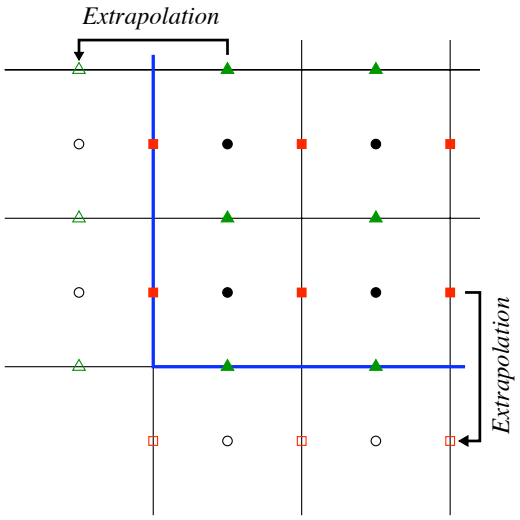

Figure 11: Boundary condition treatment of velocity

```
Ue=[2uS'-Ue(:,1) Ue 2uN'-Ue(:,end)];
```

in Python:

```
Ue = np.hstack( (2*uS-Ue[:,0,np.newaxis], Ue, ...
    2*uN-Ue[:,-1,np.newaxis]))
```

The matrix `Ve` is built in a similar way. Then, the size of the new matrices will be

`Ue`: $(N_x + 1) \times (N_y + 2)$
`Ve`: $(N_x + 2) \times (N_y + 1)$

The advection (non-linear) terms are calculated using the values of the velocity field on the boundaries of the control volumes which are different from the locations the discrete velocity field is defined. Therefore, we introduce matrices which contain an averaged value of u and v on the boundaries of the control volumes. To generate these data `Ue` and `Ve` should be averaged in the x or y direction depending on the term to be evaluated. Here, we define the *averaging* function `avg( )` such that

$$[\texttt{avg}(f)]_i = \frac{1}{2}\left(f_{i-1} + f_i\right) \ . \tag{25}$$

Note that the length of the resulting vector `avg`(f) is one element less than the length of f. Furthermore, `avg`(f) can be applied in both the x- and y-directions. In the example below, the function makes the averaging in either x or y direction.

In MATLAB:

```
function B=avg(A,idim)
  if(idim==1)
    B=1/2 [ A(2:end,:) + A(1:end-1,:) ];
  elseif(idim==2)
    B=1/2 [ A(:,2:end) + A(:,1:end-1) ];
  end;
```

In Python:

```
def avg(A,axis=0):
    if (axis==0):
        B = (A[1:,] + A[0:-1,])/2.
    elif (axis==1):
        B = (A[:,1:] + A[:,0:-1])/2.
    else:
        raise ValueError('Wrong value for axis')
    return B
```

Applying this function to average the u velocity in the y direction can be achieved as follows in MATLAB:

```
Ua=avg(Ue,2);
```

In Python:

```
Ua = avg(Ue,axis=1)
```

After averaging the size of the matrix `Ua` will be $(N_x + 1) \times (N_y + 1)$.

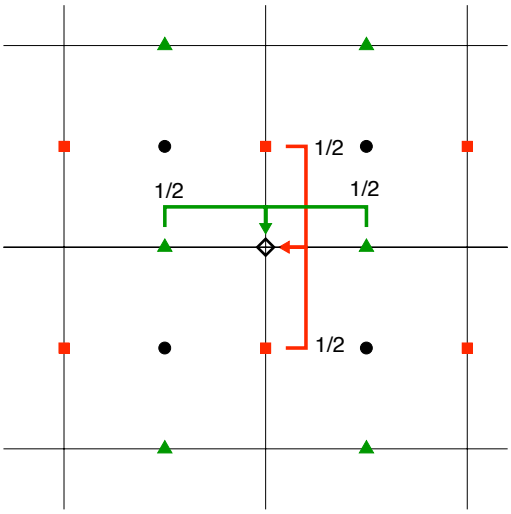

Figure 12: Averaging of velocities.

A.4 Construction of Laplace (L_p) operator

One of the most crucial steps of the code is the implementation of the Laplace operator. We need to construct that in a efficient way. To save memory and gain speed when the equations are solved we make use of *sparse* matrix storage in MATLAB. This can be done in a smart and concise but slightly more complicated way in MATLAB. Let us first define the Laplace operator for a one-dimensional case. Using central finite differences second order, with Neumann

boundary condition at the two boundaries we can define a derivative matrix with the following structure,

$$\underline{\underline{D}}_2 = \frac{1}{h^2}\begin{bmatrix} -1 & 1 & & \\ 1 & -2 & 1 & \\ & \ddots & \ddots & \ddots \\ & & 1 & -1 \end{bmatrix} . \tag{26}$$

This is implemented in the function `DD(n,h)`, available as template.

To extended this operator for a two-dimensional case, one can use the `kron` operator in MATLAB to perform a Kronecker tensor product,

$$\texttt{kron}(A,B) = \begin{bmatrix} A(1,1)\cdot B & A(1,2)\cdot B & \cdots \\ A(2,1)\cdot B & A(2,2)\cdot B & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix} . \tag{27}$$

Then one obtains the derivative matrix to compute $U_{xx} = \underline{\underline{L_P}} \cdot U$ by choosing

$$\underline{\underline{A}} = \underline{\underline{I}}_{N_y \times N_y}, \quad \underline{\underline{B}} = \underline{\underline{D}}_2 ,$$

and U_{yy} by choosing

$$\underline{\underline{A}} = \underline{\underline{D}}_2, \quad \underline{\underline{B}} = \underline{\underline{I}}_{N_x \times N_x} .$$

In MATLAB:

```
1  Lp=kron(speye(ny),DD(nx,dx))+...
```

In Python:

```
1  Lp = np.kron(sp.eye(Ny).toarray(),DD(Nx,hx).toarray()) + ...
```

The actual level of the pressure is not determined since only the pressure gradient enters the Navier–Stokes equations, *i.e.* both $P + const.$ and P satisfy the Poisson equation. Therefore, one needs to fix the value of the pressure at one node, *e.g.* $P_{1,1} = 0$. This is achieved by the following modification of the equation system in Matlab

$$L_p(1,:) = 0, \quad L_p(1,1) = 1, \quad R(1) = 0 .$$

or in Python

$$L_p[0,:] = 0, \quad L_p[0,0] = 1, \quad R[0] = 0 .$$

Ignoring this gives singular matrices. The above observation is of course a general feature: For any incompressible flow, the pressure is only determined up to a constant.

A.5 Outline of required steps

Assume that we consider an iteration that starts at $t = t^n$ with U^n, V^n and we will march towards $t = t^{n+1} = t^n + \Delta t$.

1. Compute the non-linear advection terms, namely ${NL_x}^n, {NL_x}^n$:

$$ {NL_x}^n = -((U^n)^2)_x - (U^n V^n)_y \tag{28} $$
$$ {NL_y}^n = -(U^n V^n)_x - ((V^n)^2)_y. \tag{29} $$

The term ${NL_x}^n$ is evaluated on the U velocity grid, similarly ${NL_y}^n$ on the V grid (see Fig. 10). The derivatives should be calculated based on the difference between the velocity values at neighbouring velocity nodes. Therefore, the velocities U^n and V^n need to be interpolated to the these nodes such that derivatives can be taken. For example for $\left(U^n V^n\right)_x$ we proceed as follows:

$$ ((U^n V^n)_x)_{i,j+\frac{1}{2}} = \frac{1}{h_x}\left[U_{i+\frac{1}{2},j+\frac{1}{2}} V_{i+\frac{1}{2},j+\frac{1}{2}} - U_{i-\frac{1}{2},j+\frac{1}{2}} V_{i-\frac{1}{2},j+\frac{1}{2}}\right] \tag{30} $$
$$ \text{where} \qquad U_{i+\frac{1}{2},j+\frac{1}{2}} = \frac{1}{2}\left[U_{i+\frac{1}{2},j+1} + U_{i+\frac{1}{2},j}\right] \tag{31} $$
$$ \text{and} \qquad V_{i+\frac{1}{2},j+\frac{1}{2}} = \frac{1}{2}\left[V_{i+1,j+\frac{1}{2}} + V_{i,j+\frac{1}{2}}\right]. \tag{32} $$

In MATLAB:

```
Ua=avg(Ue,2);
Va=avg(Ve,1);
UVx = 1/dx * diff( Ua.*Va, 1 , 1);
```

In Python:

```
Ua = avg(Ue,axis=1);
Va = avg(Ve,axis=0);
dUVdx = np.diff( Ua*Va, axis=0)/hx;
```

For the terms $((U^n)^2)_x$ and $((V^n)^2)_y$ the difference is taken between the pressure nodes onto which the velocity has to be interpolated first.

2. Compute the viscous diffusion terms, namely $DIFF_x^n, DIFF_y^n$

$$ DIFF_x^n = \frac{1}{Re}[U_{xx}^n + U_{yy}^n] \tag{33} $$
$$ DIFF_y^n = \frac{1}{Re}[V_{xx}^n + V_{yy}^n]\ . \tag{34} $$

For example

$$ [DIFF_x^n]_{i+\frac{1}{2},j} = \frac{1}{Re}\left(\frac{U_{i-\frac{1}{2},j} - 2U_{i+\frac{1}{2},j} + U_{i+\frac{3}{2},j}}{h_x^2} + \frac{U_{i+\frac{1}{2},j-1} - 2U_{i+\frac{1}{2},j} + U_{i+\frac{1}{2},j+1}}{h_y^2}\right). \tag{35} $$

Use the extended grid and store only interior points.

In MATLAB:

```
viscu = diff( Ue(:,2:end-1),2,1 )/dx^2 + ...
        diff( Ue(2:end-1,:),2,2 )/dy^2;
```

in Python:

```
viscu = np.diff( Ue[:,1:-1],axis=0,n=2 )/hx**2 + ...
np.diff( Ue[1:-1,:],axis=1,n=2 )/hy**2;
```

The command `diff( )` is a built-in MATLAB (or numpy) operator which takes the difference between adjacent components. The first argument is the matrix to operate on, the second is the order of the difference (here 2 for the second derivative) and the third is specifying along which matrix dimension the derivative is taken (1 for x-direction and 2 for y-direction).

3. Compute forcing terms if required, F_x^n, F_y^n.

4. Advance the solution to an intermediate time level (U^* and V^*) using explicit Euler

$$\frac{U^* - U^n}{\Delta t} = NL_x^n + DIFF_x^n + F_x^n \ , \tag{36}$$

$$\frac{V^* - V^n}{\Delta t} = NL_y^n + DIFF_y^n + F_y^n \ . \tag{37}$$

5. Solve the Poisson equation for the pressure

$$\Delta P^{n+1} = \frac{1}{\Delta t}(D_x U^* + D_y V^*) \ . \tag{38}$$

Now we need to solve the Poisson equation with homogeneous Neumann conditions for P^{n+1}. Construct the discretised Laplace operator L_p (for details on the structure of the operator L_p see Section A.4 above) and evaluate $D_x U^* + D_y V^*$ with central differences second order.

$$L_p P^{n+1} = R \quad \text{with} \quad R = \frac{1}{\Delta t}(D_x U^* + D_y V^*). \tag{39}$$

$$\Rightarrow \quad P^{n+1} = {L_p}^{-1} R \ . \tag{40}$$

Here, P^{n+1} and R are of size $N_x \times N_y$, containing the unknown pressure and the right-hand side (r.h.s.) at the cell centres, respectively.

In MATLAB:

```
  rhs = (diff([uW ; U ; uE])/dx + ...)/dt;
  rhs = reshape(rhs,nx*ny,1);
  P = Lp\rhs;
  P = reshape(P,nx,ny);
```

in Python:

```
rhs = (np.diff( np.vstack( (uW,U, uE)),axis=0)/hx + ...)/dt;
rhs = np.reshape(rhs.T,(Nx*Ny,1));
P = Lps_lu.solve(rhs)
P = np.reshape(P.T,(Ny,Nx)).T;
```

6. Compute the velocity field at time step $n+1$ according to

$$U^{n+1} = U^* - \Delta t G P^{n+1} \ . \tag{41}$$

B Matlab template

B.1 SG2212_template.m

```
 % Navier-Stokes solver,
% adapted for course SG2212
% KTH Mechanics
%
% Depends on avg.m and DD.m
%
% Code version:
% 20180222

clear all

%----------------------------------------

lid_driven_cavity=1;

if (lid_driven_cavity==1)
  % Parameters for test case I: Lid-driven cavity
  % The Richardson number is zero, i.e. passive scalar.

  Pr = 0.71;     % Prandtl number
  Re = ...;      % Reynolds number
  Ri = 0.;       % Richardson number

  dt = ...;      % time step
  Tf = 20;       % final time
  Lx = 1;        % width of box
  Ly = 1;        % height of box
  Nx = ...;      % number of cells in x
  Ny = ...;      % number of cells in y
  ig = 200;      % number of iterations between output

  % Boundary and initial conditions:
  Utop = 1.;
  Ubottom = 0.;
  % IF TEMPERATURE: Tbottom = 1.; Ttop = 0.;
  % IF TEMPERATURE: namp = 0.;
else
  % Parameters for test case II: Rayleigh-Bénard convection
  % The DNS will be stable for Ra=1705, and unstable for Ra=1715
  % (Theoretical limit for pure sinusoidal waves
  % with L=2.01h: Ra=1708)
  % Note the alternative scaling for convection problems.

  Pr = 0.71;     % Prandtl number
  Ra = ...;      % Rayleigh number
```

```
46
47    Re = 1./Pr;     % Reynolds number
48    Ri = Ra*Pr;     % Richardson number
49
50    dt = ...;   % time step
51    Tf = 20;        % final time
52    Lx = 10.;       % width of box
53    Ly = 1;         % height of box
54    Nx = ...;       % number of cells in x
55    Ny = ...;       % number of cells in y
56    ig = 200;       % number of iterations between output
57
58    % Boundary and initial conditions:
59    Utop = 0.;
60    Ubottom = 0.;
61    % IF TEMPERATURE: Tbottom = 1.; Ttop = 0.;
62    namp = 0.1;
63  end
64
65
66  %----------------------------------------
67
68  % Number of iterations
69  Nit = ...
70  % Spatial grid: Location of corners
71  x = linspace( ... );
72  y = linspace( ... );
73  % Grid spacing
74
75  dx = ...
76  dy = ...
77  % Boundary conditions:
78  uN = x*0+Utop;    vN = avg(x,2)*0;
79  uS = ...          vS = ...
80  uW = ...          vW = ...
81  uE = ...          vE = ...
82  tN = ...          tS = ...
83  % Initial conditions
84  U = ...; V = ...;
85  % linear profile for T with random noise
86  % IF TEMPERATURE: T = ... + namp*rand(Nx,Ny)
87  % Time series
88  tser = [];
89  Tser = [];
90
91  %----------------------------------------
92
93  % Compute system matrices for pressure
94  % First set homogeneous Neumann condition all around
```

```
95  % Laplace operator on cell centres: Fxx + Fyy
96  Lp = kron(speye(Ny), ... ) + kron( ... ,speye(Nx));
97  % Set one Dirichlet value to fix pressure in that point
98  Lp(1,:) = ... ; Lp(1,1) = ... ;
99  % Here you can pre-compute the LU decomposition
100 % [LLp,ULp] = lu(Lp);
101 %----------------------------------------
102
103 % Progress bar (do not replace the ... )
104 fprintf(...
105     '[         |         |         |         |         ]\n')
106
107 %----------------------------------------
108
109 % Main loop over iterations
110
111 for k = 1:Nit
112
113    % include all boundary points for u and v (linear extrapolation
114    % for ghost cells) into extended array (Ue,Ve)
115    Ue = ...
116    Ve = ...
117
118    % averaged (Ua,Va) of u and v on corners
119    Ua = ...
120    Va = ...
121
122    % construct individual parts of nonlinear terms
123    dUVdx = ...
124    dUVdy = ...
125    dU2dx = ...
126    dV2dy = ...
127
128    % treat viscosity explicitly
129    viscu = ...
130    viscv = ...
131
132    % buoyancy term
133    % IF TEMPERATURE: fy = ...
134
135    % compose final nonlinear term + explicit viscous terms
136    U = U + dt/Re*... - dt*(...);
137    V = V + dt/Re*... - dt*(...) + % IF TEMPERATURE: dt*fy;
138
139    % pressure correction, Dirichlet P=0 at (1,1)
140    rhs = (diff( ... )/dx + diff( ... )/dy)/dt;
141    rhs = reshape(rhs,Nx*Ny,1);
142    rhs(1) = ...
143    P = Lp\rhs;
```

```
144     % alternatively, you can use the pre-computed LU decompositon
145     % P = ...;
146     % or gmres
147     % P = gmres(Lp, rhs, [], tol, maxit);
148     % or as another alternative you can use GS / SOR from homework 6
149         % [PP, r] = GS_SOR(omega, Nx, Ny, hx, hy, L, f, p0, tol, maxit);
150     P = reshape(P,Nx,Ny);
151
152     % apply pressure correction
153     U = U - ...
154     V = V - ...
155
156     % Temperature equation
157     % IF TEMPERATURE: Te = ...
158     % IF TEMPERATURE: Tu = ...
159     % IF TEMPERATURE: Tv = ...
160     % IF TEMPERATURE: H = ...
161     % IF TEMPERATURE: T = T + dt*H;
162
163     %----------------------------------------
164
165     % progress bar
166     if floor(51*k/Nit)>floor(51*(k-1)/Nit), fprintf('.'), end
167
168     % plot solution if needed
169     if k==1|floor(k/ig)==k/ig
170
171       % compute divergence on cell centres
172       if (1==1)
173         div = diff([uW;U;uE])/dx + diff([vS' V vN'],1,2)/dy;
174
175         figure(1);clf; hold on;
176         contourf(avg(x,2),avg(y,2),div');colorbar
177         axis equal; axis([0 Lx 0 Ly]);
178         title(sprintf('divergence at t=%g',k*dt))
179         drawnow
180       end
181
182       % compute velocity on cell corners
183       Ua = ...
184       Va = ...
185       Len = sqrt(Ua.^2+Va.^2+eps);
186
187       figure(2);clf;hold on;
188       %contourf(avg(x,2),avg(y,2),P');colorbar
189       contourf(x,y,sqrt(Ua.^2+Va.^2)',20,'k-');colorbar
190       quiver(x,y,(Ua./Len)',(Va./Len)',.4,'k-')
191       axis equal; axis([0 Lx 0 Ly]);
192       title(sprintf('u at t=%g',k*dt))
```

```
193       drawnow
194
195       % IF TEMPERATURE: % compute temperature on cell corners
196       % IF TEMPERATURE: Ta = ...
197
198       % IF TEMPERATURE: figure(3); clf; hold on;
199       % IF TEMPERATURE: contourf(x,y,Ta',20,'k-');colorbar
200       % IF TEMPERATURE: quiver(x,y,(Ua./Len)',(Va./Len)',.4,'k-')
201       % IF TEMPERATURE: axis equal; axis([0 Lx 0 Ly]);
202       % IF TEMPERATURE: title(sprintf('T at t=%g',k*dt))
203       % IF TEMPERATURE: drawnow
204
205       % Time history
206       if (1==1)
207         figure(4); hold on;
208         tser = [tser k*dt];
209         Tser = [Tser Ue(ceil((Nx+1)/2),ceil((Ny+1)/2))];
210         plot(tser,abs(Tser))
211         title(sprintf('Probe signal at x=%g, y=%g',...
212               x(ceil((Nx+1)/2)),y(ceil((Ny+1)/2))))
213         set(gca,'yscale','log')
214         xlabel('time t');ylabel('u(t)')
215       end
216     end
217 end
218 fprintf('\n')
```

B.2 DD_template.m

```
function A = DD(n,h)
% DD(n,h)
%
% One-dimensional finite-difference derivative matrix
% of size n times n for second derivative:
%
% This function belongs to SG2212.m

A = ...
```

B.3 avg_template.m

```
function B = avg(A,idim)
% AVG(A,idim)
%
% Averaging function to go from cell centres (pressure nodes)
% to cell corners (velocity nodes) and vice versa.
% avg acts on index idim; default is idim=1.
%
% This function belongs to SG2212.m

if nargin<2, idim = 1; end

if (idim==1)
  B = (A( ... , ... )+A( ... , ... ))/2;
elseif (idim==2)
  B = (A( ... , ... )+A( ... , ... ))/2;
else
  error('avg(A,idim): idim must be 1 or 2')
end
```

C Python template

C.1 SG2212_template.py

```
# Development of a Python code to solve the two-dimensional
# Navier-Stokes equations on a rectangular domain.
# Import some relevant libraries:

%matplotlib notebook
# possible options: notebook, inline or widget
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.pylab as pylab
import matplotlib.animation
import math
import scipy.sparse as sp
import scipy.linalg as scl
from scipy.sparse.linalg import splu
params = {'legend.fontsize': 12,
          'legend.loc':'best',
          'figure.figsize': (8,5),
          'lines.markerfacecolor':'none',
          'axes.labelsize': 12,
          'axes.titlesize': 12,
          'xtick.labelsize':12,
          'ytick.labelsize':12,
          'grid.alpha':0.6}
pylab.rcParams.update(params)

# Some useful functions:
def avg(A,axis=0):
    """
    Averaging function to go from cell centres (pressure nodes)
    to cell corners (velocity nodes) and vice versa.
    avg acts on index idim; default is idim=1.
    """
    if (axis==0):
        B = (A ....+ A.....)/2.
    elif (axis==1):
        B = (A..... + A.....)/2.
    else:
        raise ValueError('Wrong value for axis')
    return B

def DD(n,h):
    """
    One-dimensional finite-difference derivative matrix
    of size n times n for second derivative:
    h^2 * f''(x_j) = -f(x_j-1) + 2*f(x_j) - f(x_j+1)
```

```
46
47      Homogeneous Neumann boundary conditions on the boundaries
48      are imposed, i.e.
49      f(x_0) = f(x_1)
50      if the wall lies between x_0 and x_1. This gives then
51      h^2 * f''(x_j) = + f(x_0) - 2*f(x_1) + f(x_2)
52                     = + f(x_1) - 2*f(x_1) + f(x_2)
53                     =              f(x_1) + f(x_2)
54
55      For n=5 and h=1 the following result is obtained:
56
57      A =
58          -1     1     0     0     0
59           1    -2     1     0     0
60           0     1    -2     1     0
61           0     0     1    -2     1
62           0     0     0     1    -1
63      """
64      data = np.concatenate( (np.array(...), np.ones( (n-2,1) ) @
65          np.array(....),np.array(....)))
66      diags = np.array(.....)
67      A = sp.spdiags(data.T, diags, n, n) / h**2
68      return A
69
70  # DD(5,1).toarray()
71
72  # Homemade version of Matlab tic and toc functions
73  def tic():
74      import time
75      global startTime_for_tictoc
76      startTime_for_tictoc = time.time()
77
78  def toc():
79      import time
80      if 'startTime_for_tictoc' in globals():
81          print("Elapsed time is " + str(time.time() -
82          startTime_for_tictoc) + " seconds.")
83      else:
84          print("Toc: start time not set")
85
86  # Simulation parameters:
87  Pr = 0.71
88  Re = ....
89  # Ri = 0.
90  dt = .....
91  Tf = 20
92  Lx = 1.
93  Ly = 1.
94  Nx = ....
```

```
Ny = .....
namp = 0.
ig = 200

# Discretisation in space and time, and definition of boundary conditions:
# number of iteratins
Nit = ....
# edge coordinates
x = np.linspace(....)
y = np.linspace(....)
# grid spacing
hx = ......
hy = ......

# boundary conditions
Utop = 1.; Ttop = 1.; Tbottom = 0.;
uN = x*0 + Utop;  uN = uN[:,np.newaxis];
vN = avg(x)*0;    vN = vN[:,np.newaxis];
uS = ....  uS = uS[:,np.newaxis];
vS = ....  vS = vS[:,np.newaxis];
uW = avg(y)*0;  uW = uW[np.newaxis,:];
vW = ....   vW = vW[np.newaxis,:];
uE = avg(y)*0;  uE = uE[np.newaxis,:];
vE = y*0;    vE = vE[np.newaxis,:];

tN = ......  tS =.....

# Pressure correction and pressure Poisson equation:
# Compute system matrices for pressure
# Laplace operator on cell centres: Fxx + Fyy
# First set homogeneous Neumann condition all around
Lp = np.kron(....).toarray(),DD(.....).toarray()) + np.kron(.....).toarray(),
        sp.eye(.....).toarray());
# Set one Dirichlet value to fix pressure in that point
Lp[:,0] = ......; Lp[0,:] =.....; Lp[0,0] = .....;
Lp_lu, Lp_piv = scl.lu_factor(Lp)
Lps = sp.csc_matrix(Lp)
Lps_lu = splu(Lps)

# Initial conditions
U = np.zeros((Nx-1,Ny))
V = np.zeros((Nx,Ny-1))
T = .... + \
    namp*(np.random.rand(Nx,Ny)-0.5);

# Main time-integration loop. Write output file "cavity.mp4" if
if (ig>0):
    metadata = dict(title='Lid-driven cavity', artist='SG2212')
    writer = matplotlib.animation.FFMpegWriter(fps=15, metadata=metadata)
```

```
144     matplotlib.use("Agg")
145     fig=plt.figure()
146     writer.setup(fig,"cavity.mp4",dpi=200)
147 
148 # progress bar
149 print('[         |         |         |         |         ]')
150 tic()
151 for k in range(Nit):
152     # print("Iteration k=%i time=%.2e" % (k,k*dt))
153 
154     # include all boundary points for u and v (linear extrapolation
155     # for ghost cells) into extended array (Ue,Ve)
156     Ue = np.vstack((uW, U, uE)); Ue = np.hstack( (2*uS-Ue[:,0,np.newaxis],
157         Ue, 2*uN-Ue[:,-1,np.newaxis]));
158     Ve = .....
159 
160     # averaged (Ua,Va) of u and v on corners
161     Ua = .....
162     Va = .....
163 
164     #  construct individual parts of nonlinear terms
165     dUVdx = np.diff(....
166     dUVdy = ....
167     Ub    = avg( Ue[:,1:-1],0);
168     Vb    = ....
169     dU2dx = np.diff( .... )/hx;
170     dV2dy = .....
171 
172     # treat viscosity explicitly
173     viscu = np.diff( .....,axis=1,n=2 )/hy**2;
174     viscv = np.diff( .....,axis=0,n=2 )/hx**2;
175 
176     # compose final nonlinear term + explicit viscous terms
177     U = U + ....
178     V = V + ....
179 
180     # pressure correction, Dirichlet P=0 at (1,1)
181     rhs = (np.diff( ....)/hx + np.diff(.....),axis=1)/hy)/dt;
182     rhs = np.reshape(.....);
183     rhs[0] = 0;
184 
185     # different ways of solving the pressure-Poisson equation:
186     P = Lps_lu.solve(....)
187 
188     P = np.reshape(....
189 
190     # apply pressure correction
191     U = U - dt*np.diff.....)/hx;
192     V = V - dt*np.diff(....)/hy;
```

```
193
194      # Temperature equation
195      ....
196
197      # do postprocessing to file
198      if (ig>0 and np.floor(k/ig)==k/ig):
199          plt.clf()
200          plt.contourf(avg(x),avg(y),T.T,levels=np.arange(0,1.05,0.05))
201          plt.gca().set_aspect(1.)
202          plt.colorbar()
203          plt.title(f'Temperature at t={k*dt:.2f}')
204          writer.grab_frame()
205
206      # update progress bar
207      if np.floor(51*k/Nit)>np.floor(51*(k-1)/Nit):
208          print('.',end='')
209
210  # finalise progress bar
211  print(' done. Iterations k=%i time=%.2f' % (k,k*dt))
212  toc()
213
214  if (ig>0):
215      writer.finish()
216
217  # Visualisation of the flow field at the end time:
218  %matplotlib notebook
219
220  Ua = np.hstack( (uS,avg(np.vstack((uW,U,uE)),1),uN));
221  Va = np.vstack((vW,avg(np.hstack((vS,V,
222                                    vN)),0),vE));
223  plt.figure()
224  plt.contourf(x,y,np.sqrt(Ua**2+Va**2).T,20)
225  plt.quiver(x,y,Ua.T,Va.T)
226  plt.gca().set_aspect(1.)
227  plt.colorbar()
228  plt.title(f'Velocity at t={k*dt:.2f}')
229  plt.show()
230
231  # compute divergence on cell centres
232  div = (np.diff( np.vstack( (uW,U, uE)),axis=0)/hx + np.diff(
233          np.hstack(( vS, V, vN)),axis=1)/hy)
234  plt.figure()
235  plt.pcolor(avg(x),avg(y),div.T,shading='nearest')
236  plt.gca().set_aspect(1.)
237  plt.colorbar()
238  plt.title(f'Divergence at t={k*dt:.2f}')
239  plt.show()
240
241  # Analysis of the pressure Poisson equation:
```

```
# Matrix structure
plt.figure()
plt.spy(Lp)
plt.show()
# Size, rank and null space:
Lp.shape
np.linalg.matrix_rank(Lp)
scl.null_space(Lp)
```