University of Groningen

Computational Physics - Final Assignment

Lid-Driven Cavity Flow
using finite differences

Robert Riesebos s3220672

November 6, 2020

Contents

1 Introduction

For the final assignment of the course Computational Physics, we used the finite differences method to find an approximate solution to the lid-driven cavity flow problem, in particular the two-dimensional (2D) case. In this section, we will provide a theoretical background for the lid-driven cavity flow problem, and introduce the equations and boundary conditions that govern lid-driven cavity flow. We will also briefly discuss the computational methods that were used to approximate the solution.

After this, in the following sections, we explain the algorithm and provide pseudocode of the implementation (Section 2). Next, the results of the simulation are provided, discussed and contextualized (Section 3, Section 4). Finally, in Section 5, the report is summarized in a conclusion). At the end of the report, in Appendix A, the source code of the simulation is provided.

1.1 Lid-driven cavity flow

Lid-driven cavity flow is a well known problem in the field of fluid dynamics. It is commonly used as a benchmark problem to verify new simulation methods for in-compressible viscous flows [1].

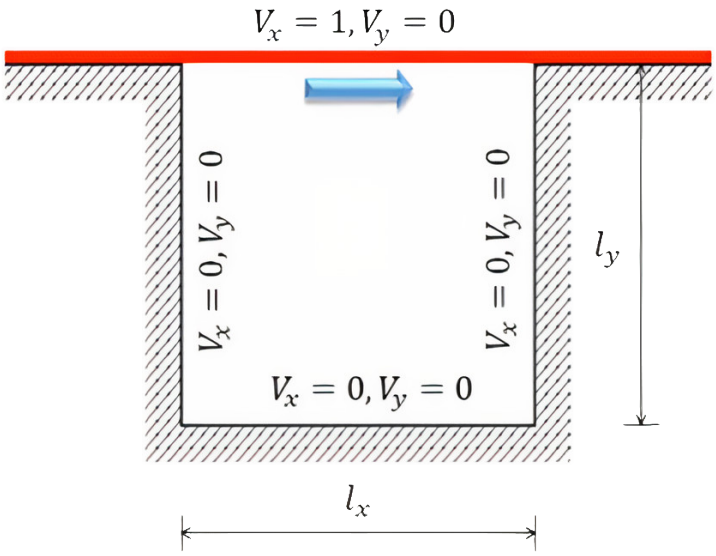


Figure 1: The 2D domain for the lid-driven cavity problem. Indicated are the velocity boundary conditions. The top wall (lid) is highlighted in red. (Credits: image provided in the assignment).

The problem can be described as follows. Consider the two-dimensional domain depicted in Figure 1, we have a closed cavity with side lengths $l_x = l_y = 1$. The top wall (the lid) is moving in the positive x-direction with a constant unit velocity (V_x). The other three walls are stationary, and the velocities in the y-direction (V_y) are zero for all the walls. The cavity is filled with an in-compressible, viscous fluid in which a flow is induced by the moving lid.

The research question is: what are the velocity and pressure profiles for different initial model parameters? The different parameters are introduced in Section 1.2.4, and the corresponding results are discussed in the results section (Section 3).

1.2 Governing equations

In this section we will introduce the equations that govern the lid-driven cavity flow, including the equations for the boundary conditions.

1.2.1 Vorticity form of the Navier-Stokes equations

Because the lid-driven cavity flow describes the motion of a viscous, in-compressible fluid, we can use the (incompressible) Navier-Stokes equations. In particular, we will use the non-dimensionalized vorticity form of the Navier-Stokes equations, represented by the following partial differential equations:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -\omega, \tag{1a}$$

$$\frac{\partial \omega}{\partial t} = -\frac{\partial u}{\partial y}\frac{\partial \omega}{\partial x} + \frac{\partial u}{\partial x}\frac{\partial \omega}{\partial y} + \frac{1}{\mathrm{Re}}\left(\frac{\partial^2 \omega}{\partial x^2} + \frac{\partial^2 \omega}{\partial y^2}\right), \tag{1b}$$

where u is the stream function and ω is the vorticity. Re is 'Reynolds number', a dimensionless number that describes the relative contributions of inertial effects versus viscous effects in the fluid flow. Reynolds number is one of the input parameters of the lid-driven cavity flow simulation.

1.2.2 Velocity and pressure field

The velocity and pressure profiles can be calculated using the stream function u and vorticity w. The velocities V in the x-direction (V_x) and in the y-direction (V_y) are related to the stream function and vorticity as follows:

$$V_x = \frac{\partial u}{\partial y}, \qquad V_y = -\frac{\partial u}{\partial x}, \tag{2}$$

$$\omega = \frac{\partial V_y}{\partial x} - \frac{\partial V_x}{\partial y}. \tag{3}$$

The pressure field is dependent on the stream function, and can be obtained by solving the following Poisson equation:

$$\frac{\partial^2 P}{\partial x^2} + \frac{\partial^2 P}{\partial y^2} = 2\left[\frac{\partial^2 u}{\partial x^2}\frac{\partial^2 u}{\partial y^2} - \left(\frac{\partial^2 u}{\partial x \partial y}\right)^2\right]. \tag{4}$$

1.2.3 Boundary conditions

As discussed before, the velocity boundary conditions (i.e. the velocities at the walls) are as follows:

$$\begin{aligned} x = 0: &\qquad V_x = V_y = 0, \\ x = l_x: &\qquad V_x = V_y = 0, \\ y = 0: &\qquad V_x = V_y = 0, \\ y = l_y: &\qquad V_x = 1, V_y = 0. \end{aligned} \tag{5}$$

These boundary conditions are for the velocities, but we are interested in the boundary conditions for the stream function and the vorticity. Since these are related to the velocity, we can use the velocities to derive them. We start with the boundaries for the stream function. Using the relation given by Equation (2), we get for the top wall:

$$V_x = \frac{\partial u}{\partial y} = 1, \qquad V_y = -\frac{\partial u}{\partial x} = 0, \tag{6}$$

and for the other walls:

$$V_x = \frac{\partial u}{\partial y} = 0, \qquad V_y = -\frac{\partial u}{\partial x} = 0. \tag{7}$$

Since the partial derivative of the stream function u with respect to y is zero for the top and bottom walls, the stream function is a constant at these boundaries. Furthermore, the partial derivative of the stream function u with respect to x is zero for the left and right walls, so at these walls the stream function is also a constant. Because the top and bottom walls meet the left and right walls at the cavity corners, the stream function is the same arbitrary constant for the whole boundary, in our case, we set the stream function to be zero at the boundary walls.

Following the fact that the stream function is constant at the boundaries, we can derive the boundary conditions for the vorticity. For the left and right walls we get:

$$\omega_{wall} = -\frac{\partial^2 u}{\partial x^2}, \tag{8}$$

and for the top and bottom wall we get:

$$\omega_{wall} = -\frac{\partial^2 u}{\partial y^2}. \tag{9}$$

1.2.4 Model parameters

There are three parameters that directly influence the simulation, plus four parameters for the velocities of the walls. These boundary velocities are discussed in Section 1.2.3, and are constant in the previously problem definition. As such we won't discuss the wall-velocity parameters again in this section.

The three input parameters that influence the simulation, and define the model are as follows:

- Reynolds number (Re): as alluded to before, this parameter describes the relative contributions of inertial effects versus viscous effects in the fluid flow.
- The grid size (N): number of grid cells in each grid dimension. The total amount of cells equals N^2.
- The time step (Δt): defines the increase in time for each iteration of the simulation. A higher time step yields more accurate results, but increases the computation time.

One important note for the model parameters is that the following criterion must hold for a stable simulation:

$$\frac{\Delta t}{\mathrm{Re}\left(\frac{1}{N}\right)^2} \leq 0.25 \tag{10}$$

1.3 Computational methods

In this section we will discuss the main computational method that are used in the simulation.

1.3.1 Finite difference method (FDM)

The main computational method that is used to approximate the solution to the vorticity form of the Navier-Stokes equations (Equation (1)) is the Finite Difference Method (FDM). As the name suggests, the finite difference method is used to approximate derivatives using finite differences. The derivative of a function function f at a point x is defined by the limit:

$$f' = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}. \tag{11}$$

There are multiple schemes to approximate the derivative using the limit given in Equation (11). In our simulation we opted to use the central difference, as opposed to the forward or backward difference, since it is more accurate. The central difference is defined as follows:

$$f' \approx \frac{f(x+h) - f(x-h)}{2h}. \tag{12}$$

We also used the forward difference method once, to approximate the time-derivative in Equation (1b). The forward difference is defined as follows:

$$f' \approx \frac{f(x+h) - f(x)}{h}. \tag{13}$$

The results of discretizating the Navier-Stokes equations using the discussed finite difference methods are given in Section 2.1.

1.3.2 Relaxation and Successive Over-Relaxation (SOR)

Another computational method that was used is iterative relaxation (Gauss – Seidel method) to approximately solve systems of linear equations. In particular, this was used to find the solution for the stream function given by Equation (1a) and the Poisson-type Equation (4) for the pressure.

The successive over-relaxation method is also implemented, but eventually it was not used because the faster convergence of the SOR method was dwarfed by the speed improvement of vectorizing the regular relaxation method (the SOR method could not be vectorized).

2 Algorithm and program structure

In this section we will explain the algorithm used to solve the lid-driven cavity flow problem. Furthermore, we will provide pseudocode and explain the program structure. We will also provide a brief discussion of the error sources.

2.1 Discretization

Before discussing the algorithm, it is important to discretize all the provided equations, to understand how these can be approximated.

2.1.1 Navier-Stokes equations

We start with the Navier-Stokes equations given in Equation (1). In order to discretize the equations, we divide the computational domain (square cavity) in N by N grid cells with discrete centers ($x_i = ih, y_j = jh$), where h is the grid cell length ($1/N$).

We start by discretizing and rewriting Equation (1a) using the central difference approximation:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -\omega$$

$$\begin{aligned}
&\approx \frac{u^n_{i+1,j} + u^n_{i-1,j} - 2u^n_{i,j}}{h^2} + \frac{u^n_{i,j+1} + u^n_{i,j-1} - 2u^n_{i,j}}{h^2} = -w^n_{i,j} \\
&\equiv \frac{u^n_{i+1,j} + u^n_{i-1,j} + u^n_{i,j+1} + u^n_{i,j-1} - 4u^n_{i,j}}{h^2} = -w^n_{i,j} \\
&\equiv u^n_{i+1,j} + u^n_{i-1,j} + u^n_{i,j+1} + u^n_{i,j-1} - 4u^n_{i,j} = -w^n_{i,j} h^2 \\
&\equiv u^n_{i+1,j} + u^n_{i-1,j} + u^n_{i,j+1} + u^n_{i,j-1} + w^n_{i,j} h^2 = 4u^n_{i,j} \\
&\equiv u^n_{i,j} = 0.25(u^n_{i+1,j} + u^n_{i-1,j} + u^n_{i,j+1} + u^n_{i,j-1} + w^n_{i,j} h^2). \qquad (14)
\end{aligned}$$

This results in a system of linear equations that can be solved by relaxation as discussed in Section 1.3.2, yielding the solution to the stream function.

Next, we will discretize Equation (1b) using the forward difference for the time-derivative and the central difference for the other partial derivatives:

$$\frac{\partial \omega}{\partial t} = -\frac{\partial u}{\partial y}\frac{\partial \omega}{\partial x} + \frac{\partial u}{\partial x}\frac{\partial \omega}{\partial y} + \frac{1}{\mathrm{Re}}\left(\frac{\partial^2 \omega}{\partial x^2} + \frac{\partial^2 \omega}{\partial y^2}\right)$$

$$\begin{aligned}
\approx \frac{\omega_{i,j}^{n+1} - \omega_{i,j}^{n}}{\Delta t} &= -\left(\frac{u_{i,j+1}^{n} - u_{i,j-1}^{n}}{2h}\right)\left(\frac{\omega_{i+1,j}^{n} - \omega_{i-1,j}^{n}}{2h}\right) \\
&\quad + \left(\frac{u_{i+1,j}^{n} - u_{i-1,j}^{n}}{2h}\right)\left(\frac{\omega_{i,j+1}^{n} - \omega_{i,j-1}^{n}}{2h}\right) \\
&\quad + \frac{1}{\mathrm{Re}}\left(\frac{\omega_{i+1,j}^{n} + \omega_{i-1,j}^{n} - 2\omega_{i,j}^{n}}{h^2} + \frac{\omega_{i,j+1}^{n} + \omega_{i,j-1}^{n} - 2\omega_{i,j}^{n}}{h^2}\right) \\
\equiv \frac{\omega_{i,j}^{n+1} - \omega_{i,j}^{n}}{\Delta t} &= -\left(\frac{u_{i,j+1}^{n} - u_{i,j-1}^{n}}{2h}\right)\left(\frac{\omega_{i+1,j}^{n} - \omega_{i-1,j}^{n}}{2h}\right) \\
&\quad + \left(\frac{u_{i+1,j}^{n} - u_{i-1,j}^{n}}{2h}\right)\left(\frac{\omega_{i,j+1}^{n} - \omega_{i,j-1}^{n}}{2h}\right) \\
&\quad + \frac{1}{\mathrm{Re}}\left(\frac{\omega_{i+1,j}^{n} + \omega_{i-1,j}^{n} + \omega_{i,j+1}^{n} + \omega_{i,j-1}^{n} - 4\omega_{i,j}^{n}}{h^2}\right) \\
\equiv \omega_{i,j}^{n+1} = \omega_{i,j}^{n} - \Delta t\Bigg[&\left(\frac{u_{i,j+1}^{n} - u_{i,j-1}^{n}}{2h}\right)\left(\frac{\omega_{i+1,j}^{n} - \omega_{i-1,j}^{n}}{2h}\right) \\
&+ \left(\frac{u_{i+1,j}^{n} - u_{i-1,j}^{n}}{2h}\right)\left(\frac{\omega_{i,j+1}^{n} - \omega_{i,j-1}^{n}}{2h}\right) \\
&+ \frac{1}{\mathrm{Re}}\left(\frac{\omega_{i+1,j}^{n} + \omega_{i-1,j}^{n} + \omega_{i,j+1}^{n} + \omega_{i,j-1}^{n} - 4\omega_{i,j}^{n}}{h^2}\right)\Bigg].
\end{aligned} \tag{15}$$

This gives us how we can compute the vorticity for the next time step ($w_{i,j}^{n+1}$), from the vorticity at the current time step ($w_{i,j}^{n}$).

Finally, we discretize the boundary conditions for the vorticity given by Equations (8) and (9). We start with the moving top wall. First, we expand the stream function using a Taylor series expansion (where the top wall is located at $j = 1$, and the first interior cell at $j = 2$):

$$u_{i,j=2} = u_{i,j=1} + \frac{\partial u_{i,j=1}}{\partial y}h + \frac{\partial^2 u_{i,j=1}}{\partial y^2}\frac{h^2}{2} + \mathcal{O}(h^3).$$

Substituting using Equation (2) $\left(\frac{\partial u}{\partial y} = V_{wall}\right)$ and Equation (9) $\left(\omega_{wall} = -\frac{\partial^2 u}{\partial y^2}\right)$, we get:

$$u_{i,j=2} = u_{i,j=1} + V_{wall}h - \omega_{wall}\frac{h^2}{2} + \mathcal{O}(h^3).$$

After re-arranging the equation we obtain the following discrete boundary for the vorticity at the top wall:

$$\omega_{top} = (u_{i,j=1} - u_{i,j=2})\frac{2}{h^2} + V_{top}\frac{2}{h} + \mathcal{O}(h). \tag{16}$$

Performing the same steps for the bottom wall ($j = N$) gives us:

$$\omega_{bottom} = (u_{i,j=N} - u_{i,j=N-1})\frac{2}{h^2} - V_{bottom}\frac{2}{h} + \mathcal{O}(h). \tag{17}$$

Now for the left and right walls the discretizations are similar, but in the derivation we use $-\frac{\partial u}{\partial x} = V_{wall}$ and Equation (8) $\left(\omega_{wall} = -\frac{\partial^2 u}{\partial x^2}\right)$ instead. This gives us:

$$\omega_{left} = (u_{i=1,j} - u_{i=2,j})\frac{2}{h^2} + V_{left}\frac{2}{h} + \mathcal{O}(h), \tag{18}$$

and

$$\omega_{right} = (u_{i=N,j} - u_{i=N-1,j})\frac{2}{h^2} - V_{right}\frac{2}{h} + \mathcal{O}(h). \tag{19}$$

If the velocity for the wall (V_{wall}) is zero, the right-most term in the equation disappears. Furthermore, because we set the stream function at the boundaries to zero, the terms for the stream function can also be excluded. This results in equations of the form:

$$\omega_{wall} = -\frac{2}{h^2}u_{i=N-1,j} + V_{wall}\frac{2}{h} + \mathcal{O}(h). \tag{20}$$

2.1.2 Velocity and pressure field

To discretize the velocities given in Equation (2) we again use the central difference. This results in the following discretizations:

$$(V_x)^n_{i,j} = \frac{u^n_{i,j+1} - u^n_{i,j-1}}{2h}, \qquad (V_y)^n_{i,j} = -\frac{u^n_{i+1,j} - u^n_{i-1,j}}{2h}. \tag{21}$$

Discretizing the pressure equation given in Equation (4) is slightly more involved, as it contains multiple second-order partial derivatives. We again use central differences. The central difference for mixed partial derivative is as follows:

$$\left(\frac{\partial^2 u}{\partial x \partial y}\right)_{i,j} = \frac{\left(\frac{\partial u}{\partial y}\right)_{i+1,j} - \left(\frac{\partial u}{\partial y}\right)_{i-1,j}}{2\Delta x} + \mathcal{O}(\Delta x^2), \qquad \Delta x = h. \tag{22}$$

Using Equation (22), we get the following discretization:

$$\frac{\partial^2 P}{\partial x^2} + \frac{\partial^2 P}{\partial y^2} = 2\left[\frac{\partial^2 u}{\partial x^2}\frac{\partial^2 u}{\partial y^2} - \left(\frac{\partial^2 u}{\partial x \partial y}\right)^2\right] \tag{23}$$

$$\tag{24}$$

$$\approx \frac{P^n_{i+1,j} + P^n_{i-1,j} + P^n_{i,j+1} + P^n_{i,j-1} - 4P^n_{i,j}}{h^2} \tag{25}$$

$$= 2\Bigg[\left(\frac{u^n_{i+1,j} + u^n_{i-1,j} - 2u^n_{i,j}}{h^2}\right)\left(\frac{u^n_{i,j+1} + u^n_{i,j-1} - 2u^n_{i,j}}{h^2}\right) \tag{26}$$

$$- \left(\frac{u^n_{i+1,j+1} - u^n_{i+1,j-1} - u^n_{i-1,j+1} + u^n_{i-1,j-1}}{4h^2}\right)^2\Bigg]. \tag{27}$$

2.2 Implementation

After discretizing the Navier-Stokes equation using the finite difference method, we are ready to discuss the implementation of the lid-driven cavity flow simulation. To do so, we make use of the pseudocode provided in Listing 1 as well as snippets from the source code provided in Appendix A.

Note: when we use the term vectorizing in the following sections, it means converting regular Python loops to ranges using `numpy` — resulting in a drastic speed-up as numpy ranges use pre-compiled, optimized C code under the hood [2].

input : *Model parameters*: Re, N, time_step and max_time,
Relaxations parameters: max_iterations, max_error
output: Approximation of the stream function (u) and vorticity (ω)

$u \leftarrow$ N $\times$ N
$\omega \leftarrow$ N $\times$ N
h = 1 / N
for *step* $\leftarrow$ 0 **to** *max_time / time_step* **do**
  t = step * time_step

  `// Step 1: find the value of the stream function through relaxation`
  for *iter* $\leftarrow$ 1 **to** *max_iterations* **do**
    u_old $\leftarrow u$
    for $i \leftarrow$ 2 **to** *N-1* **do**
      for $j \leftarrow$ 2 **to** *N-1* **do**
        $u^n_{i,j} \leftarrow 0.25(u^n_{i+1,j} + u^n_{i-1,j} + u^n_{i,j+1} + u^n_{i,j-1} + w^n_{i,j}h^2)$
      end
    end
    residual = $\sum_{i=1}^{i=N} \sum_{j=1}^{j=N} |u\_\text{old}_{i,j} - u_{i,j}|$
    if *residual* $\leq$ *max_error* **then**
      break
    end
  end

  `// Step 2: update vorticity at the boundaries`
  for $i \leftarrow$ 2 **to** *N-1* **do**
    for $j \leftarrow$ 2 **to** *N-1* **do**
      $\omega_{i,j=1} \leftarrow -\frac{2}{h^2}u_{i,j=2} + V_{top}\frac{2}{h}$
      $\omega_{i,j=N} \leftarrow -\frac{2}{h^2}u_{i,j=N-1} - V_{bottom}\frac{2}{h}$
      $\omega_{i=1,j} \leftarrow -\frac{2}{h^2}u_{i=2,j} + V_{left}\frac{2}{h}$
      $\omega_{i=N,j} \leftarrow -\frac{2}{h^2}u_{i=N-1,j} - V_{right}\frac{2}{h}$
    end
  end

  `// Step 3: update vorticity on the interior`
  for $i \leftarrow$ 2 **to** *N-1* **do**
    for $j \leftarrow$ 2 **to** *N-1* **do**

$$\omega^{n+1}_{i,j} = \omega^n_{i,j} - \Delta t\Bigg[\left(\frac{u^n_{i,j+1} - u^n_{i,j-1}}{2h}\right)\left(\frac{\omega^n_{i+1,j} - \omega^n_{i-1,j}}{2h}\right) + \left(\frac{u^n_{i+1,j} - u^n_{i-1,j}}{2h}\right)\left(\frac{\omega^n_{i,j+1} - \omega^n_{i,j-1}}{2h}\right) + \frac{1}{\text{Re}}\left(\frac{\omega^n_{i+1,j} + \omega^n_{i-1,j} + \omega^n_{i,j+1} + \omega^n_{i,j-1} - 4\omega^n_{i,j}}{h^2}\right)\Bigg]$$

    end
  end
end

Algorithm 1: Pseudocode of the lid-driven cavity flow simulation

The pseudocode in Listing 1 describes the main structure of the simulation. It shows the time-loop and the steps used to approximate the Navier-Stokes equations. These steps are described in more detail in the following sections.

2.2.1 Step 1: find the value of the stream function through relaxation

The first step in the time-loop is finding the solution to the stream function by solving the system of linear equations given by Equation (14). To do so we have to iteratively relax the system of linear equations until it converges, i.e. until the difference between the previous and current solution falls below a certain `max_error` threshold. We can either use regular relaxation as shown in the pseudocode:

```
// Step 1:  find the value of the stream function through relaxation
for iter ← 1 to max_iterations do
    u_old ← u
    for i ← 2 to N-1 do
        for j ← 2 to N-1 do
            u^n_{i,j} ← 0.25(u^n_{i+1,j} + u^n_{i-1,j} + u^n_{i,j+1} + u^n_{i,j-1} + w^n_{i,j} h^2)
        end
    end
    residual = Σ_{i=1}^{i=N} Σ_{j=1}^{j=N} |u_old_{i,j} − u_{i,j}|
    if residual ≤ max_error then
        break
    end
end
```

Algorithm 2: Solving the stream function using regular relaxation

Or successive over relaxation (SOR), for which the pseudocode would be:

```
// Step 1:  find the value of the stream function through SOR
for iter ← 1 to max_iterations do
    u_old ← u
    for i ← 2 to N-1 do
        for j ← 2 to N-1 do
            u^n_{i,j} ← 0.25 · OMEGA · (u^n_{i+1,j} + u^n_{i-1,j} + u^n_{i,j+1} + u^n_{i,j-1} + w^n_{i,j} h^2) + (1 − OMEGA) · u^n_{i,j}
        end
    end
    residual = Σ_{i=1}^{i=N} Σ_{j=1}^{j=N} |u_old_{i,j} − u_{i,j}|
    if residual ≤ max_error then
        break
    end
end
```

Algorithm 3: Solving the stream function using successive over-relaxation (SOR)

where OMEGA is the relaxation parameter (with a preferred value in the range $1 \leq \text{OMEGA} \leq 2$). This SOR method would result in faster convergence than the regular relaxation method, resulting in the outer loop to go through less iterations. However, the SOR method could not be vectorized when implementing it in Python, resulting in the following code:

Note: we removed comments present in the full source code to increase compactness.

```
for _ in range(MAX_ITERATIONS_SOR):
    sf_old = sf.copy()

    for i in range(1, N - 1):
        for j in range(1, N - 1):
            sf[i, j] = (0.25 * OMEGA * (sf[i + 1, j] + sf[i - 1, j] + sf[i, j + 1]
            ↪  + sf[i, j - 1] + h * h * vt[i, j]) + (1 - OMEGA) * sf[i, j])

    residual = np.sum(np.abs(sf_old - sf))
    if residual <= MAX_ERROR_SOR:
        break
```

The ordinary relaxation could however be vectorized, resulting in a drastic speed-up that outweighed the benefit of faster convergence that successive over-relaxation gives us. The vectorized code is shown below:

```
for _ in range(MAX_ITERATIONS):
    sf_old = sf.copy()

    sf[1:-1, 1:-1] = (0.25 * (sf[2:, 1:-1] + sf[0:-2, 1:-1] + sf[1:-1, 2:] +
    ↪  sf[1:-1, 0:-2] + h * h * vt[1:-1, 1:-1]))

    residual = np.sum(np.abs(sf_old - sf))
    if residual <= MAX_ERROR:
        break
```

For all the steps after step 1 we encountered no problems while vectorizing the code. As such the code is vectorized where possible to achieve maximum efficiency.

2.2.2 Step 2: update the vorticity at the boundaries

In step 2 we update the vorticity at the boundaries using the boundary equations we derived in Section 2.1. The pseudocode is as follows:

```
// Step 2:  update vorticity at the boundaries
for i ← 2 to N-1 do
    for j ← 2 to N-1 do
        ω_{i,j=1} ← −(2/h²) u_{i,j=2} + V_top (2/h)
        ω_{i,j=N} ← −(2/h²) u_{i,j=N−1} − V_bottom (2/h)
        ω_{i=1,j} ← −(2/h²) u_{i=2,j} + V_left (2/h)
        ω_{i=N,j} ← −(2/h²) u_{i=N−1,j} − V_right (2/h)
    end
end
```

Algorithm 4: Updating the vorticity at the boundaries

The corresponding vectorized source code is as follows:

```
# Top wall (moving lid)
vt[N - 1, 1:N - 1] = -2 * sf[N - 2, 1:N - 1] / (h * h) + VELOCITY_TOP * 2 / h
# Bottom wall
vt[0, 1:N - 1] = -2 * sf[1, 1:N - 1] / (h * h) - VELOCITY_BOTTOM * 2 / h
# Left wall
vt[1:N - 1, 0] = -2 * sf[1:N - 1, 1] / (h * h) + VELOCITY_LEFT * 2 / h
# Right wall
vt[1:N - 1, N - 1] = -2 * sf[1:N - 1, N - 2] / (h * h) - VELOCITY_RIGHT * 2 / h
```

An important thing to note is that the origin of the plots, created using `matplotlib`, is at the bottom left, while the origin for the discretized boundaries is at the top left. This is reflected by the ranges for the boundaries in the source code. The change in origin affects the velocity calculations, as discussed in Section 2.2.4.

2.2.3 Step 3: update the vorticity on the interior

The final step we have to perform in the time-loop in order to find the approximate solution to the Navier-Stokes equation at the current time, is updating the vorticity for the interior grid cells, i.e. all the grid cells apart from the boundaries. The update rule is given by Equation (15). The pseudocode for step 3 is as follows:

```
// Step 3:  update vorticity on the interior
for i ← 2 to N-1 do
    for j ← 2 to N-1 do
        ω^{n+1}_{i,j} = ω^n_{i,j} − Δt [ ((u^n_{i,j+1} − u^n_{i,j−1}) / 2h) ((ω^n_{i+1,j} − ω^n_{i−1,j}) / 2h)
                                       + ((u^n_{i+1,j} − u^n_{i−1,j}) / 2h) ((ω^n_{i,j+1} − ω^n_{i,j−1}) / 2h)
                                       + (1/Re) ((ω^n_{i+1,j} + ω^n_{i−1,j} + ω^n_{i,j+1} + ω^n_{i,j−1} − 4ω^n_{i,j}) / h²) ]
    end
end
```

Algorithm 5: Updating the vorticity on the interior

And the corresponding, vectorized Python code is:

```
# Calculate right hand side
w[1:-1, 1:-1] = (-(((sf[1:-1, 2:] - sf[1:-1, 0:-2]) * (vt[2:, 1:-1] - vt[0:-2,
↪  1:-1]) - (sf[2:, 1:-1] - sf[0:-2, 1:-1]) * (vt[1:-1, 2:] - vt[1:-1, 0:-2])) /
↪  (4 * h * h)) + (1 / RE) * ((vt[2:, 1:-1] + vt[0:-2, 1:-1] + vt[1:-1, 2:] +
↪  vt[1:-1, 0:-2] - 4 * vt[1:-1, 1:-1]) / (h * h)))

# Update interior vorticity
vt += TIME_STEP * w
```

First, the right hand side of the equation is calculated. We already know the value of the vorticity at the current time ($\omega_{i,j}^n$) so we don't compute it again. We also delay multiplying by Δt. Then we compute $\omega_{i,j}^{n+1}$ by adding the old values plus the right hand side times Δt.

2.2.4 Post-processing

Now that we know how to obtain the approximate solutions for the stream function and the vorticity, we can continue by using these solutions to compute the velocity and pressure profiles. This is done after the time-loop finished, in a post-processing step. The pseudocode detailing how to compute the velocities and pressure field is shown in Listing 6.

input : Solution to the stream function (u) and vorticity (ω)
output: Velocity and pressure profiles

velocity_x $\leftarrow$ N $\times$ N
velocity_y $\leftarrow$ N $\times$ N
velocity_total $\leftarrow$ N $\times$ N

$P \leftarrow$ N $\times$ N

`// Set velocity at the boundaries`
for $i \leftarrow 2$ **to** *N-1* **do**
  for $j \leftarrow 2$ **to** *N-1* **do**
    velocity_x$^n_{i,j=1} \leftarrow V_{top}$
    velocity_x$^n_{i,j=N} \leftarrow V_{bottom}$
    velocity_y$^n_{i=1,j} \leftarrow V_{left}$
    velocity_y$^n_{i=N,j} \leftarrow V_{right}$
  end
end

`// Calculate velocity on the interior`
for $i \leftarrow 2$ **to** *N-1* **do**
  for $j \leftarrow 2$ **to** *N-1* **do**
    velocity_x$^n_{i,j} \leftarrow \frac{u^n_{i,j+1} - u^n_{i,j-1}}{2h}$
    velocity_y$^n_{i,j} \leftarrow -\frac{u^n_{i+1,j} - u^n_{i-1,j}}{2h}$
    velocity_total $\leftarrow \sqrt{|\text{velocity\_x} + \text{velocity\_y}|}$
  end
end

`// Find the value of the pressure field through relaxation`
for *iter* $\leftarrow 1$ **to** *max_iterations* **do**
  P_old $\leftarrow P$
  for $i \leftarrow 2$ **to** *N-1* **do**
    for $j \leftarrow 2$ **to** *N-1* **do**

$$rhs^n_{i,j} = 2\left[\left(\frac{u^n_{i+1,j} + u^n_{i-1,j} - 2u^n_{i,j}}{h^2}\right)\left(\frac{u^n_{i,j+1} + u^n_{i,j-1} - 2u^n_{i,j}}{h^2}\right) - \left(\frac{u^n_{i+1,j+1} - u^n_{i+1,j-1} - u^n_{i-1,j+1} + u^n_{i-1,j-1}}{4h^2}\right)^2\right]$$

      $P^n_{i,j} \leftarrow 0.25 * (P\text{\_old}^n_{i+1,j} + P^n_{i-1,j} + P\text{\_old}^n_{i,j+1} + P^n_{i,j-1} - h^2 rhs^n_{i,j})$
    end
  end
  residual $= \sum_{i=1}^{i=N} \sum_{j=1}^{j=N} |P\text{\_old}_{i,j} - P_{i,j}|$
  if *residual* $\leq$ *max_error* **then**
    break
  end
end

Algorithm 6: Post-processing to obtain velocity and pressure profiles

The velocity calculations are quite straight-forward; the corresponding vectorized Python

code is as follows:

```
velocity_x = np.zeros((N, N), dtype='float64')
velocity_y = np.zeros((N, N), dtype='float64')

# Set horizontal velocity at the boundaries
velocity_x[N - 1, :] = VELOCITY_TOP
velocity_x[0, :] = VELOCITY_BOTTOM
velocity_y[:, 0] = VELOCITY_LEFT
velocity_y[:, N - 1] = VELOCITY_RIGHT

# Calculate velocity for the interior
# Origin is at bottom left instead of top left, so the x and y velocities are
↪  swapped
velocity_x[1:-1, 1:-1] = -(sf[2:, 1:-1] - sf[0:-2, 1:-1]) / (2 * h)
velocity_y[1:-1, 1:-1] = (sf[1:-1, 2:] - sf[1:-1, 0:-2]) / (2 * h)
velocity_total = np.sqrt(np.absolute(velocity_x + velocity_y))

return velocity_x, velocity_y, velocity_total
```

Because of the different origin, as explained before, the calculations for the horizontal and vertical velocities are swapped compared to Equation (2).

For the pressure field calculation we again have to solve a system of linear equations through relaxation. Similar to the relaxation for the stream function, it turns out that, because of vectorization, the ordinary relaxation method is faster than the successive over-relaxation (SOR) method. The code to compute the pressure field is as follows:

```
pf = np.zeros((N, N), dtype='float64')

for _ in range(MAX_ITERATIONS):
    pf_old = pf.copy()
    rhs = calculate_pressure_field(sf, h)

    pf[1:-1, 1:-1] = 0.25 * (pf_old[2:, 1:-1] + pf[0:-2, 1:-1]
                             + pf_old[1:-1, 2:] + pf[1:-1, 0:-2] - h * h *
                             ↪  rhs[1:-1, 1:-1])

    if np.sum(np.abs(pf_old - pf)) <= MAX_ERROR:
        break
```

2.2.5 Visualization

For the visualization we used `matplotlib` library. Specifically, we used `matplotlib.pyplot.contourf` for the filled contour plots and
`matplotlib.pyplot.quiver` for the velocity vector plots. The complete plotting code is provided in the source code in Appendix A.

2.3 Errors

In this section we will briefly discuss the different error sources in the lid-driven cavity flow simulation.

First when we look at the time t. For each iterations of the time-loop, we made sure to calculate the time separately, in order to prevent a round-off errors (accumulating errors due to limitations in the floating point precision). If we, for the time t, replaced the line: `t = i * TIME_STEP` with `t += TIME_STEP` we would have fallen into this trap.

Every discretized partial differential equation is also an error source. Whenever we use the central difference for the approximation we get an error of $\mathcal{O}(h^2)$. Whenever we use the forward difference we get an error of $\mathcal{O}(h)$. In this case $h = 1/N$ where N is the number of grid cells in each dimension. Increasing N, thus increasing the number of grid cells yields a finer grid causing the error decrease.

Finally, when we use successive over-relaxation or plain relaxation to solve a system of linear equations there is also an error in the solution. We can control this error by looping until the error falls below a threshold we decide.

2.4 Code optimizations

To speed up the code a couple of optimizations were made, including:

- We use `h * h` instead of `h ** 2`, as the performance of multiplication is faster (since exponentation has some overhead, even for the power of two).
- Numpy indexing (`arr[i, j]`), instead of Python indexing (`arr[i][j]`).
- Vectorization of the different loops whenever possible.

To quantify the speed difference between vectorized code and regular loops, we ran the time-loop excluding the stream function calculation 100000 times, for both the vectorized code and the unvectorized code. The test code is provided in Appendix B. The results are shown in Table 1 below.

	Vectorized code	**Unvectorized code**
Avg. run-time	0.69094	63.677

Table 1: Speed comparison between vectorized and unvectorized code

These results show that for this particular test case the vectorized code is more than 90 times as fast!

3 Results

3.1 Flow case A: $Re = 10, N = 100, \Delta t = 0.0001$

3.1.1 Bottom wall stationary

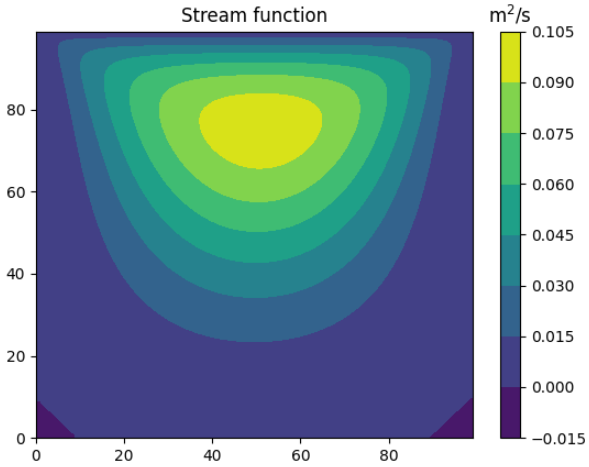


Figure 2: Stream function

We observe that the streamlines are nearly symmetrical around a vertical center line. The streamlines are closer together near the top — closer to the lid. Furthermore, there seems to be some sort of vortex in the horizontal center of the top region. In the bottom corners, there are two vortices that spin in the opposite direction of the main, larger vortex.

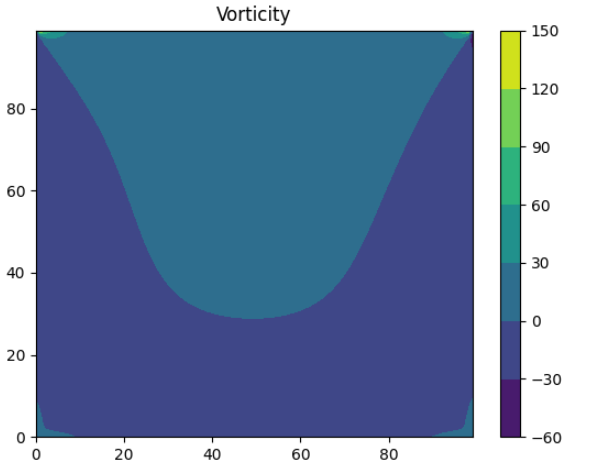


Figure 3: Vorticity

For the vorticity we see that it has a similar shape as the stream function, it kind of outlines the vortices. Both the central vortex and the two vortices in the corners are present.

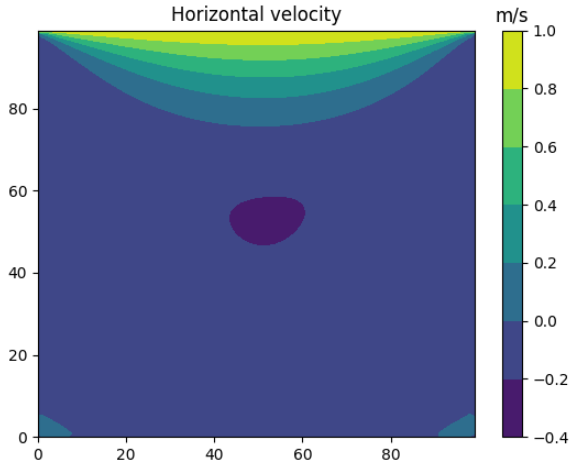


Figure 4: Horizontal velocities

The horizontal velocity is 1 m/s at the top and slowly decrease when going downwards. The contours at the top become more U-shaped when going down. In the center there is an ellipse-like shape with a negative horizontal velocity. Finally, in the bottom corners there is also a positive, horizontal velocity.

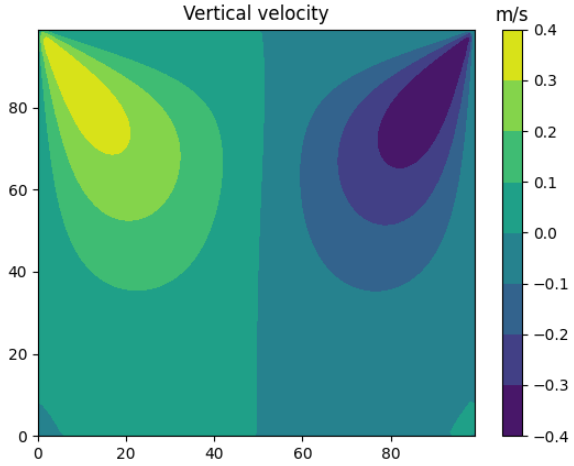


Figure 5: Vertical velocities

The vertical velocity shows two shapes with opposite velocities originating from the top corners. These seem to appear at the left and right of the main vortex shown in the stream function plot (Figure 2). The vortices in the corners also have a vertical velocity component.

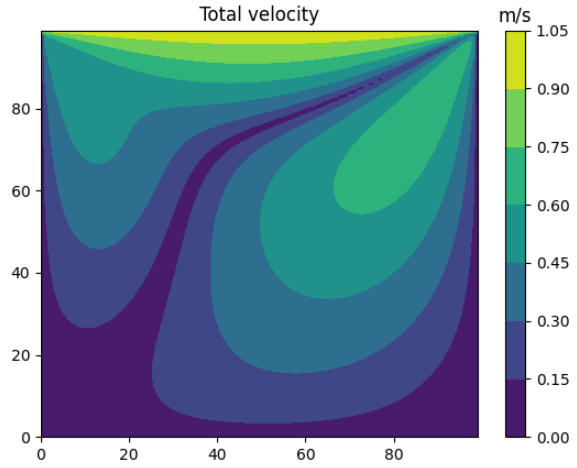


Figure 6: Total velocities

The total velocity show the combined effects of the horizontal and vertical velocity.

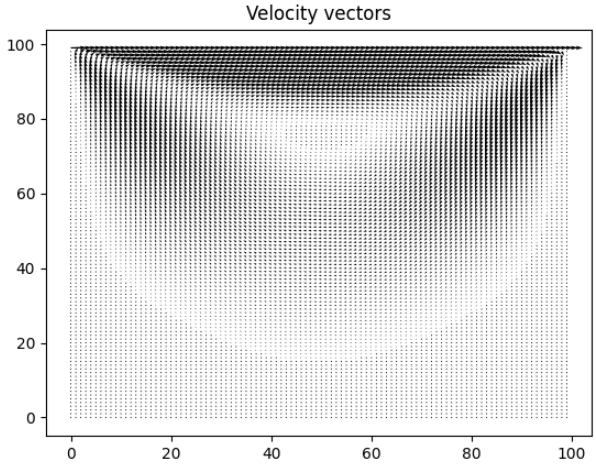


Figure 7: Velocity vector field

The velocity vectors are relatively small to read, but a pattern is still visible. After zooming in it is clear that the main vortex is rotating in clockwise direction. In this plot the smaller, secondary vortices are not directly visible.

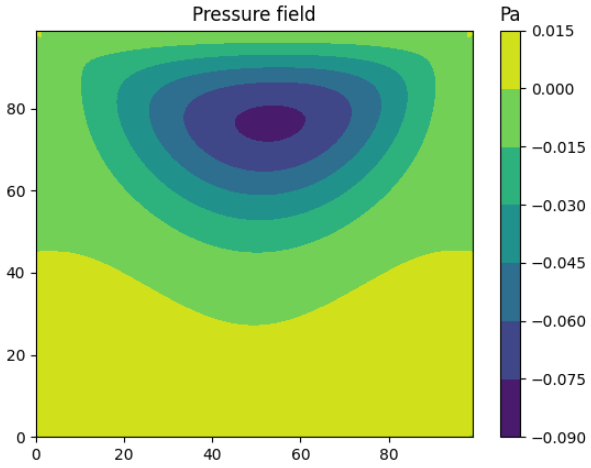


Figure 8: Pressure field

The pressure field shows a negative pressure in the center of the main vortex, decreasing when nearing the edge of the vortex. At the bottom of the cavity a positive pressure is observed, with a U-shaped indent matching the contour of the of the main vortex. After zooming in a small pressure difference can just be observed in the bottom corners at the secondary vortices.

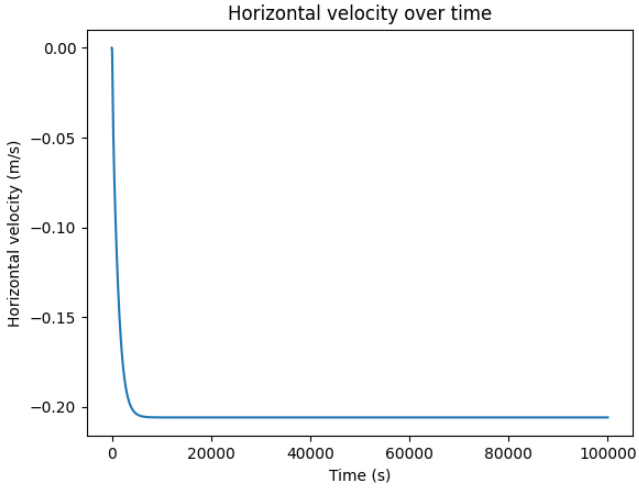


Figure 9: Horizontal velocities for the center probe

In this plot the vertical velocity at the center of the cavity is plotted over time, i.e. for each iteration of the simulation. It shows that the horizontal velocity at the center starts at zero, after which it drops to around -0.22 m/s in relatively few iterations. Then it stays around -0.22 m/s for the rest of the simulation. The end point of the graph matches Figure 12.

3.1.2 Bottom wall moving in the positive x-direction

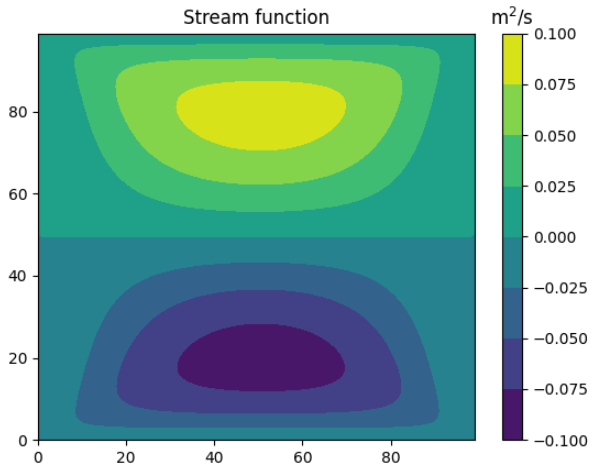


Figure 10: Stream function

Figure 10 shows the stream function plot where both the top and bottom wall move in the positive x-direction with unit speed. We observe two vortices moving in opposite directions. The graph looks to be nearly symmetric about the both the horizontal and vertical centers, only the sign differs for the horizontal symmetry.

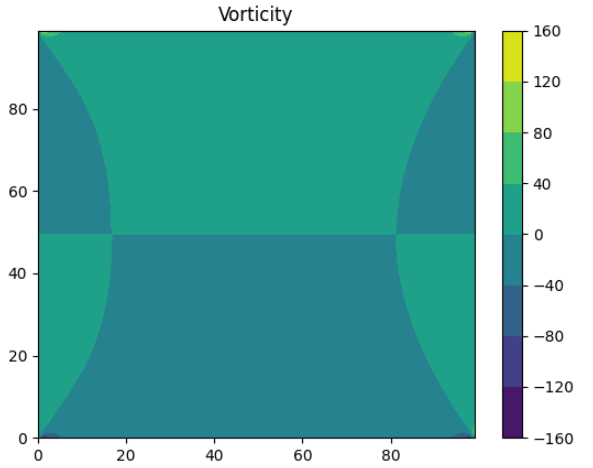


Figure 11: Vorticity

Just like the plot of the stream function, the vorticity plot looks symmetric around both center lines apart from the flipped sign. Furthermore, the colored areas of vorticity field seem to contain the two vortices.

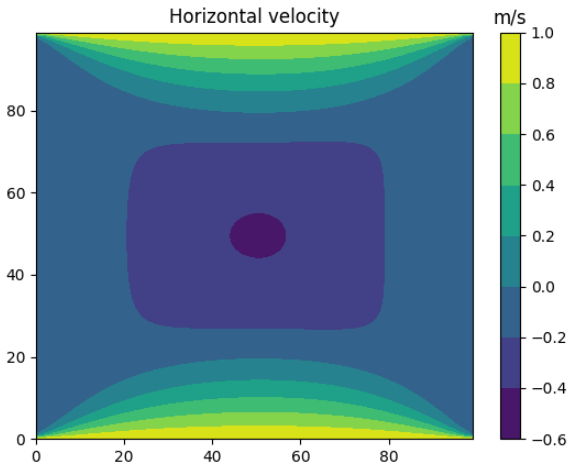


Figure 12: Horizontal velocities

We can clearly observe the horizontal velocities of the top and bottom wall. The velocities near the top wall look nearly identical to the horizontal velocities where only the top wall was moving (Figure 4). These velocities are the same at the bottom wall, and the whole plot seems to be symmetric around the horizontal and vertical centers, without flipping signs. Where the horizontal velocities in Figure 4 only have an elliptical shape in the center, Figure 12 also has an elliptical shape with a negative velocity, but it is surrounded by a rounded square that has a smaller, negative velocity.

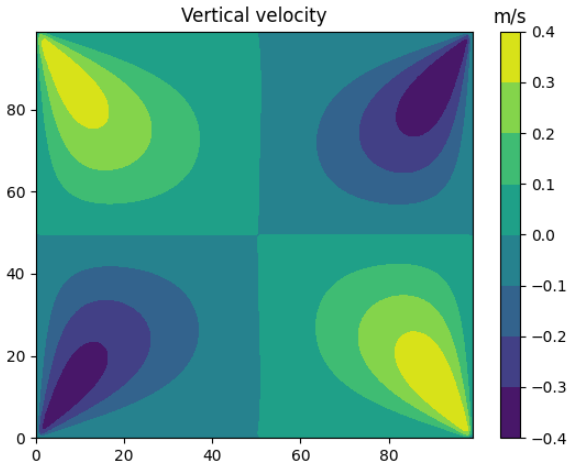


Figure 13: Vertical velocities

The vertical velocity looks similar to Figure 5; it also has the two drop-like shapes with opposite velocities at the top corners, they reach less far to the center however. These shapes are mirrored in the horizontal center line, and the sign is flipped for the bottom 'droplets'. Again these shapes surround the vortices.

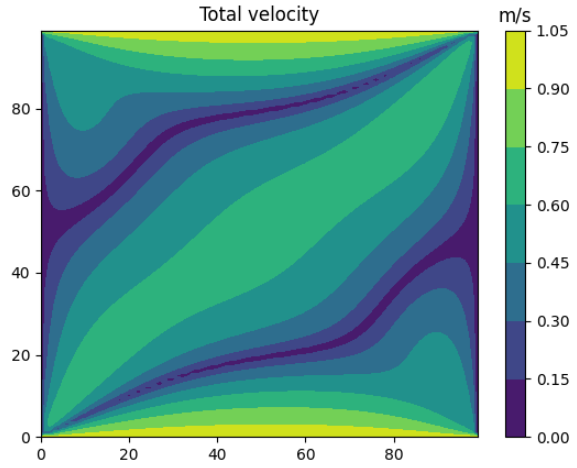


Figure 14: Total velocities

The total velocity again the combination of the horizontal and vertical velocities.

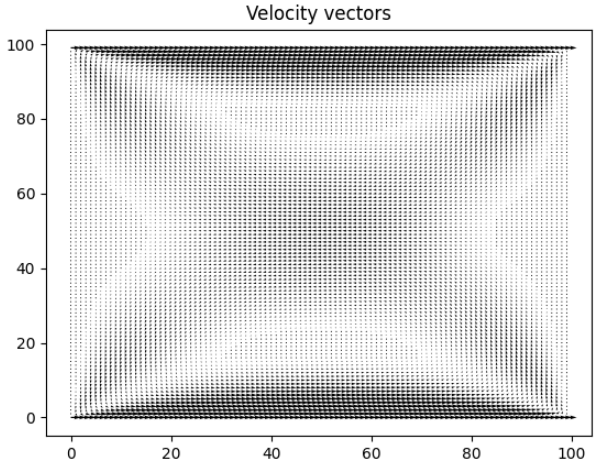


Figure 15: Velocity vector field

The velocity vectors show two vortices, where the top vortex is spinning clock-wise and the bottom vortex is spinning counter-clockwise.

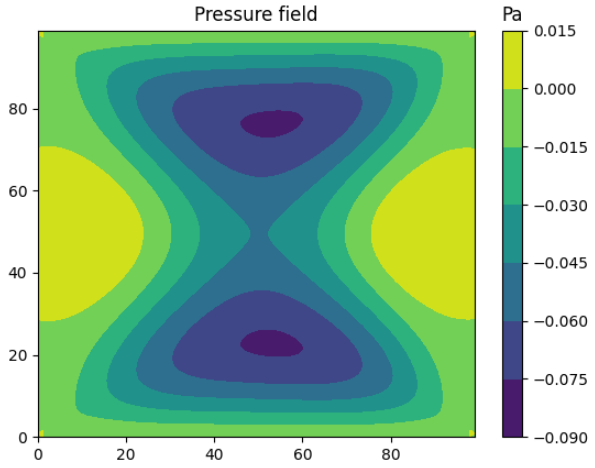


Figure 16: Pressure field

Just as before, the pressure is the smallest at the centers of the vortices. At the left and right we observe two 'bubbles' with a positive pressure. The pressure field around both vortices is shaped like an hourglass.

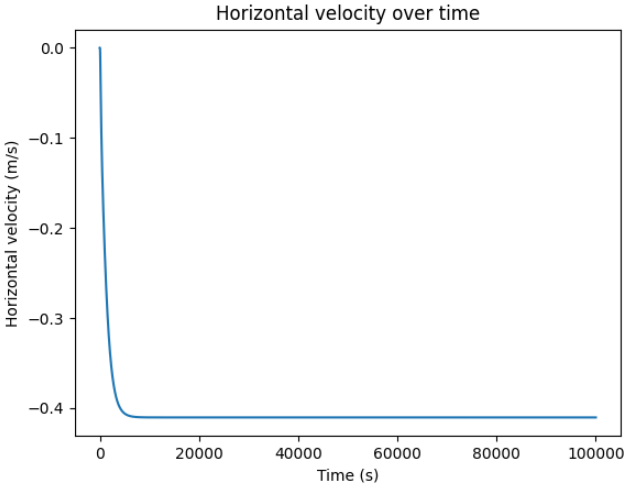


Figure 17: Horizontal velocities for the center probe

The horizontal velocity again quickly falls to a value that it stays at for the remainder of the simulation. This time the value looks to be close to -0.42 m/s, around double that of the value from Figure 9.

3.1.3 Bottom wall moving in the negative x-direction

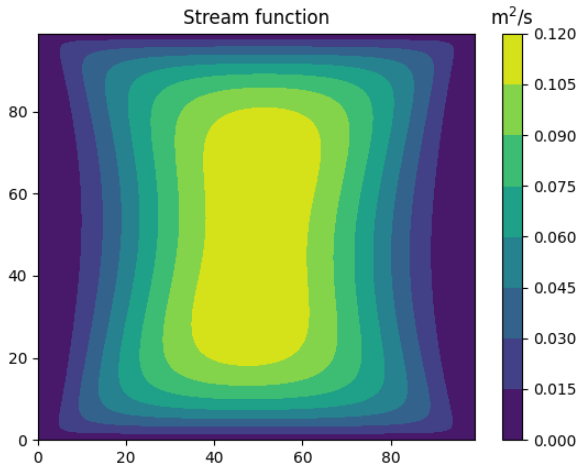


Figure 18: Stream function

For the situation where the top and bottom wall are moving in opposite directions, with the same speed, we observe one vortex. This vortex is centered horizontally and seems to be elongated vertically. It is also slightly tilted to the right. The whole plot seems to be diagonally symmetric (around the lines from bottom left to top right).

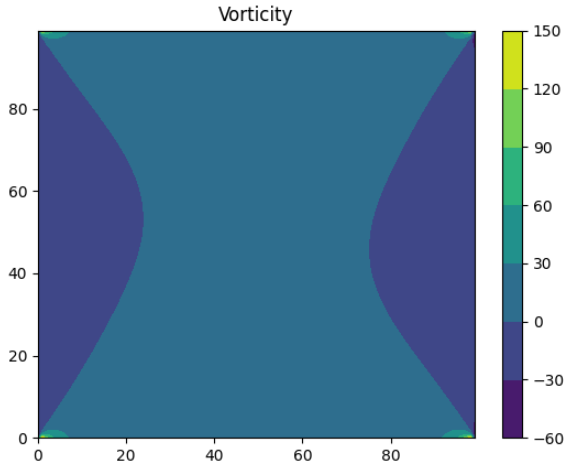


Figure 19: Vorticity

The vorticity has an hourglass shape surrounding the main vortex.

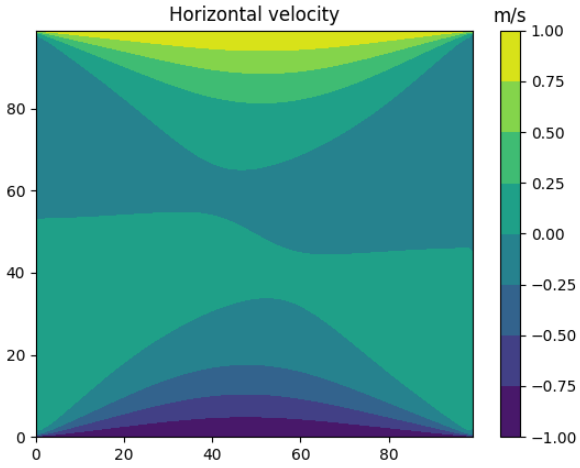


Figure 20: Horizontal velocities

The horizontal velocities again slowly decrease as the distance to the wall increases. This time there is not a clean, straight divide where the effects from the top and bottom wall meet, but more of a wave.

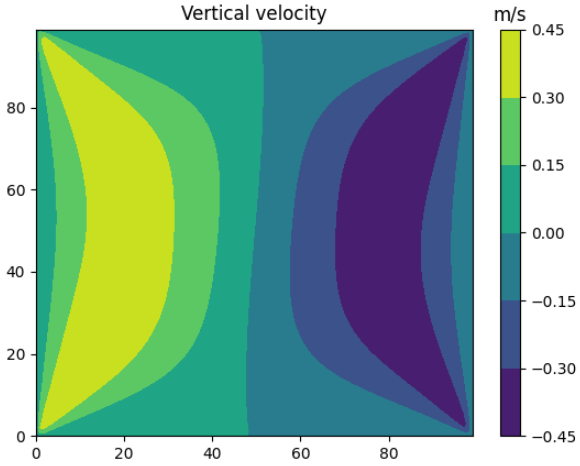
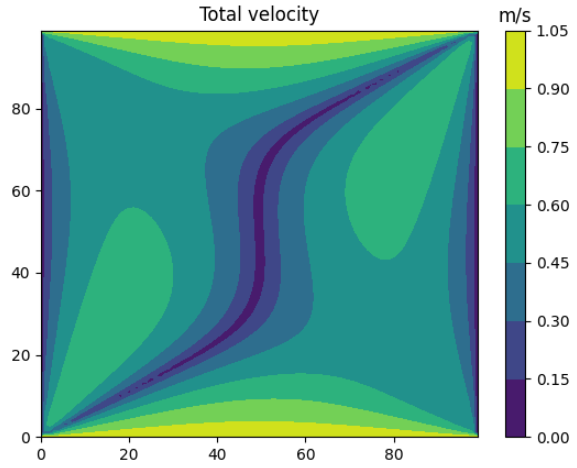


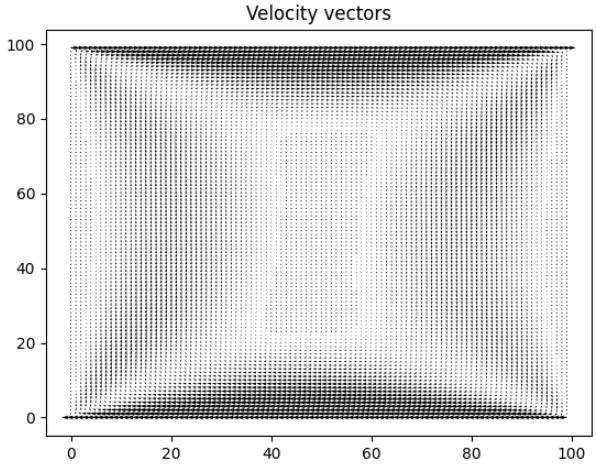
Figure 21: Vertical velocities

The vertical velocity has two banana-shaped areas where the velocity is largest/smallest. These areas start at the top corners and span all the way to the bottom corners. As with the horizontal velocity the divide where both effects meet is not a straight line.



The total velocity shows the effects of both the horizontal and vertical velocities, and is included for completeness' sake.

Figure 22: Total velocities



The velocity vectors clearly show the central, elongated vortex.

Figure 23: Velocity vector field

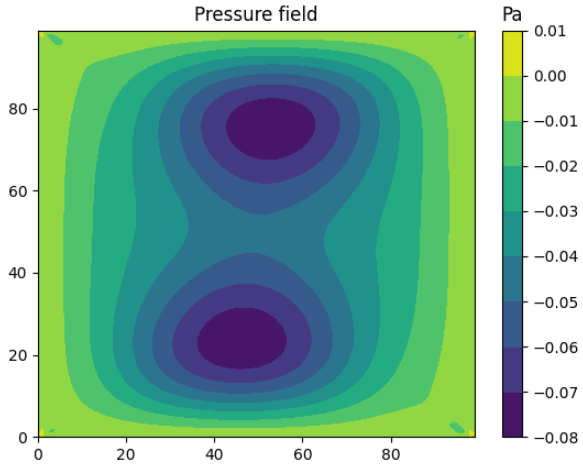


Figure 24: Pressure field

The pressure field shows two areas with a relatively large negative pressure. It looks like the main vortex might be composed of two smaller vortices that merged. There seem to be small disturbances in the corners.

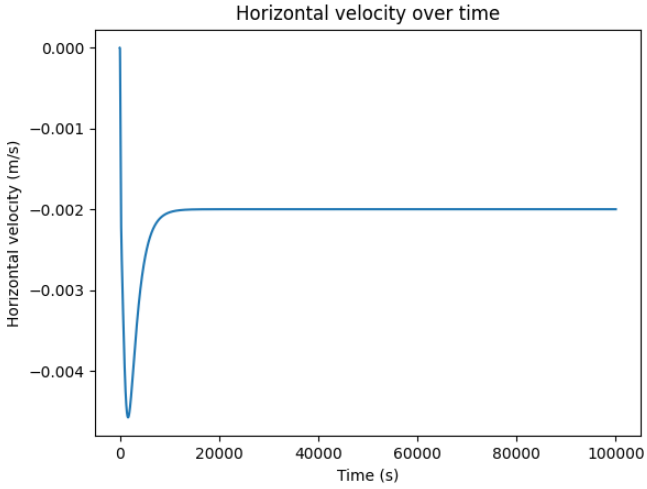


Figure 25: Horizontal velocities for the center probe

The horizontal velocity over time dips down to around 0.0047 m/s, after which it goes up again to around 0.0021 m/s where it stabilizes. The velocity at the center is significantly lower than for the previous two configurations.

3.2 Flow case B: $Re = 100, N = 200, \Delta t = 0.00001$

Note: for the remaining plots we will focus on aspects that set them apart from previously discussed plots, to prevent unnecessary repetition.

3.2.1 Bottom wall stationary

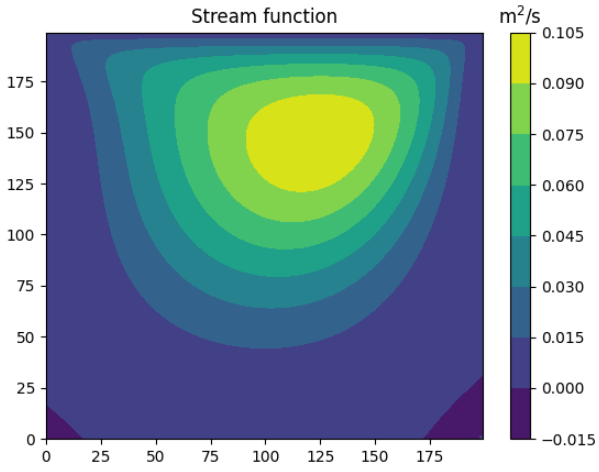


Figure 26: Stream function

The stream function seems to have lost its symmetric aspect, as the main vortex now seems to be 'pulled' to the top right corner. The secondary, smaller vortices also differ in size; the bottom right vortex is significantly larger than the bottom left vortex.

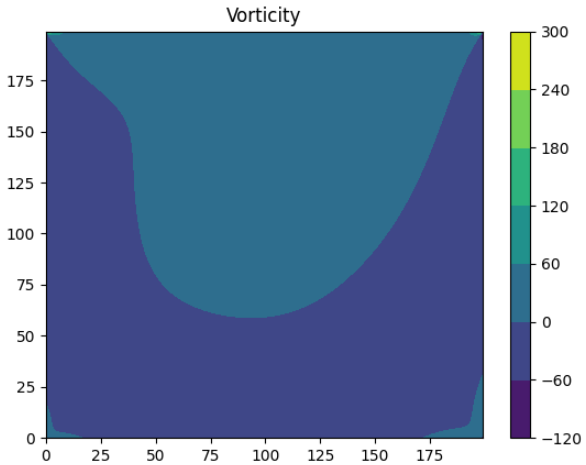


Figure 27: Vorticity

The vorticity follows the same pattern as the stream function, losing its symmetry.

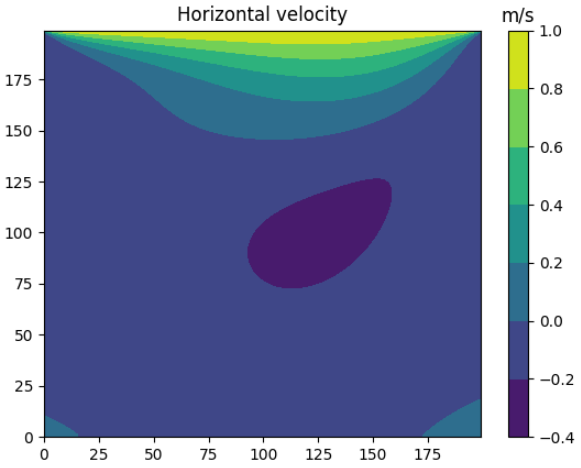


Figure 28: Horizontal velocities

As with the stream function the horizontal velocity regions are distorted to the top right.

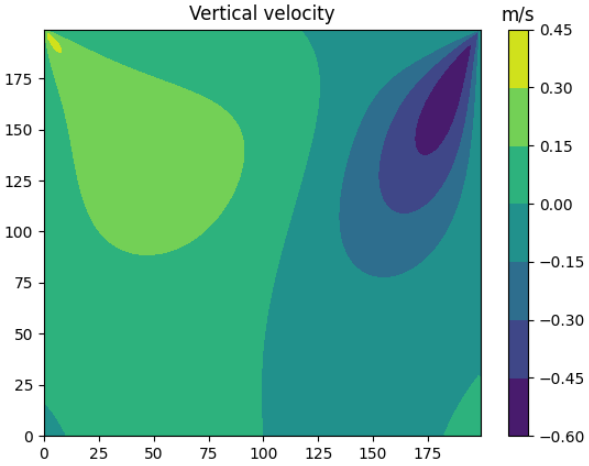


Figure 29: Vertical velocities

The horizontal velocities follow the same pattern. Interestingly the area with the largest velocity, located in the top left has almost disappeared.

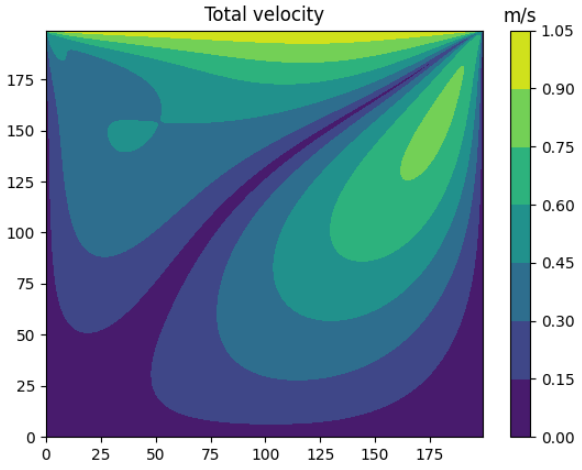


Figure 30: Total velocities

The total velocity shows the combined effects of the horizontal and vertical velocities.

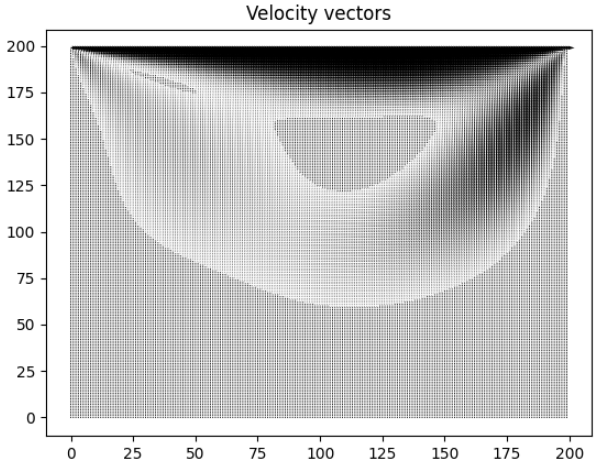


Figure 31: Velocity vector field

Due to the grid size, the velocity vectors become almost impossible to discern. However, the primary vortex is still clearly visible. Also the areas where the magnitude of the vectors is larger can be seen. There seems to be a patch with larger velocities at the top-right.

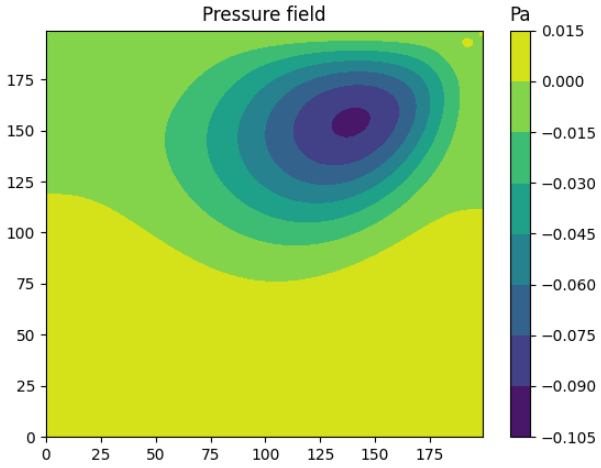


Figure 32: Pressure field

The pressure field is also skewed to the top right. In the top right corner, a small round area is visible where the pressure is larger than the surrounding area.

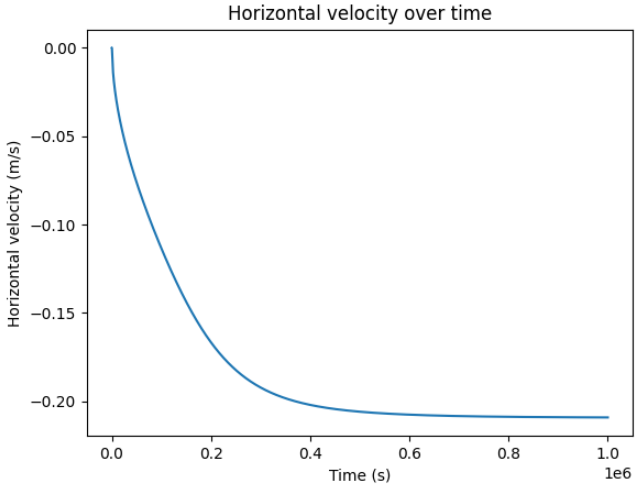


Figure 33: Horizontal velocities for the center probe

The horizontal velocity at the center takes longer to converge, but it ends up at the same value as for Figure 9; around 0.22 m/s.

3.2.2 Bottom wall moving in the positive x-direction

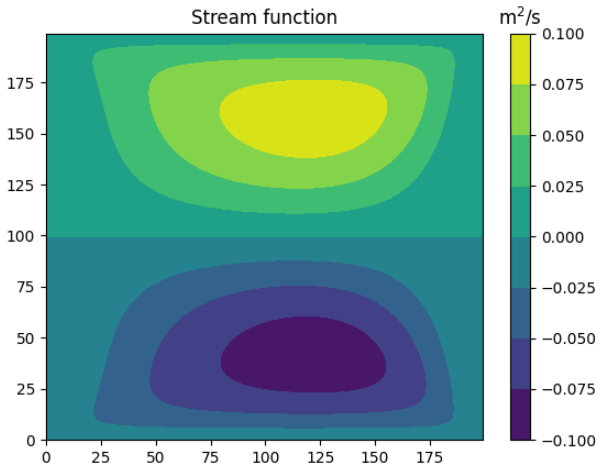


Figure 34: Stream function

Again the symmetry is lost compared to the case where $Re = 10, N = 100$. This time the plot is not skewed to the right.

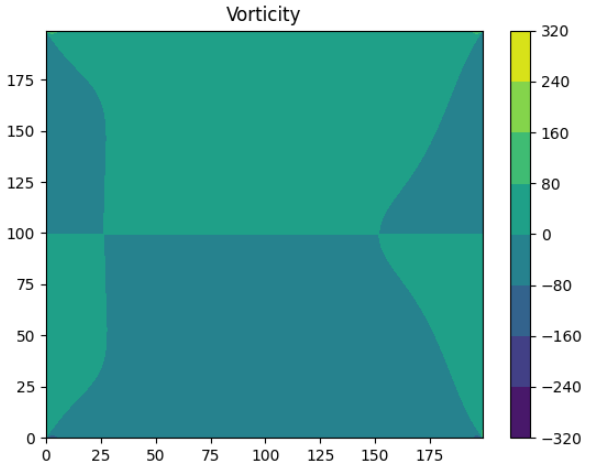


Figure 35: Vorticity

The vorticity follows the same pattern as the stream function (as they are related), and also loses its symmetry. Interestingly there appears to be a bulge in the center part of the right contour lines.

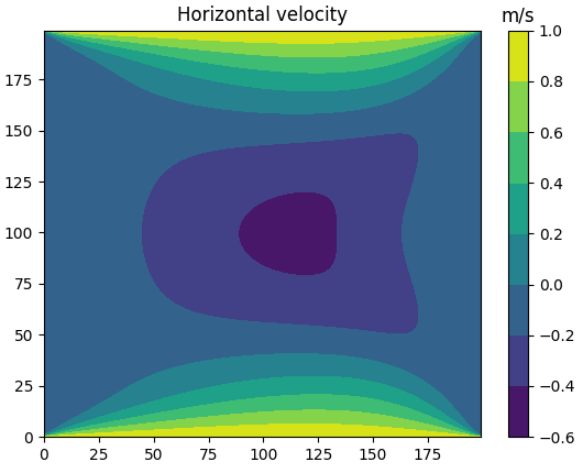


Figure 36: Horizontal velocities

The trend of the graph being skewed to the right continues for the horizontal velocities, but the bulge visible in the vorticity plot in Figure 35 is also visible here.

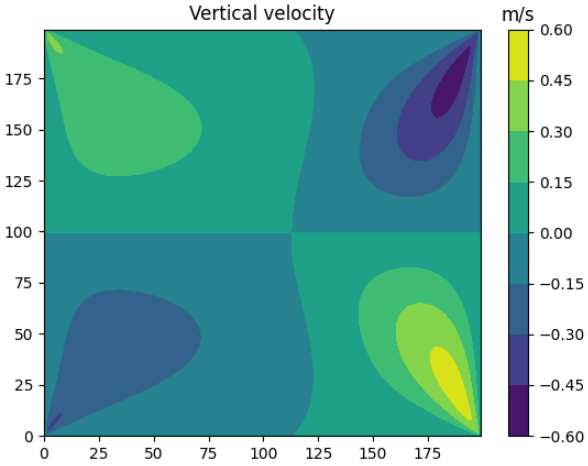


Figure 37: Vertical velocities

The vertical velocities mirror what happened in Figure 29, and are skewed to the right compared to Figure 13.

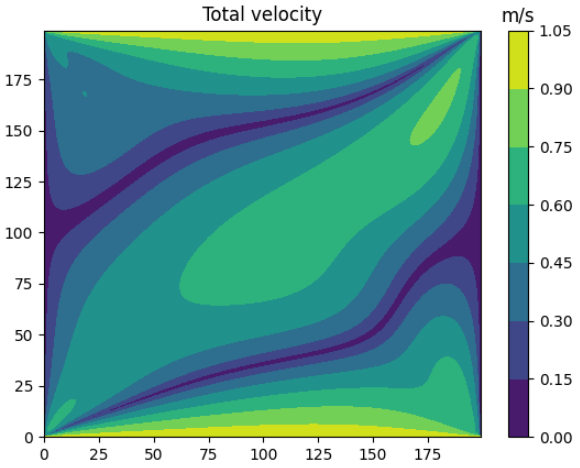


Figure 38: Total velocities

The total velocity again shows the contributions of the horizontal and vertical velocities.

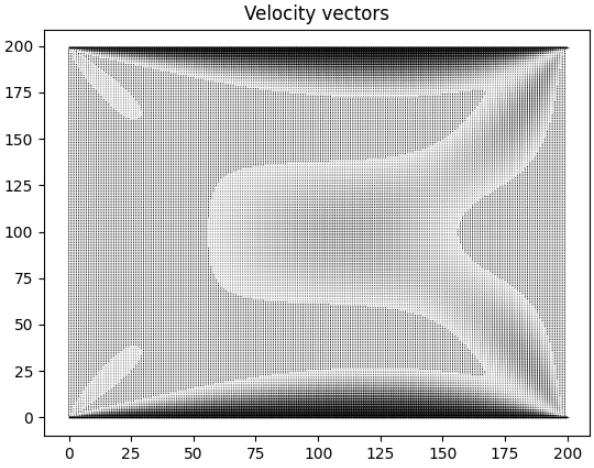


Figure 39: Velocity vector field

The velocity vectors shows regions originating from the left corners induced by the vertical velocities, a region for the central vortex from the horizontal velocities, as well as larger regions from the right corners which seem to connect to the central vortex. These right corner regions also originate from the vertical velocities. The flow in the middle with negative x-direction, induced by the moving walls seems to lose speed faster than in flow case A.

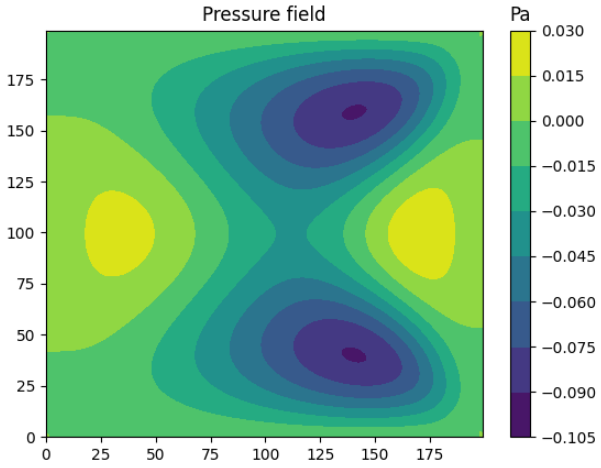


Figure 40: Pressure field

The pressure field shows two vortices that almost connect. Very close inspection also shows pressure increases in the corners.

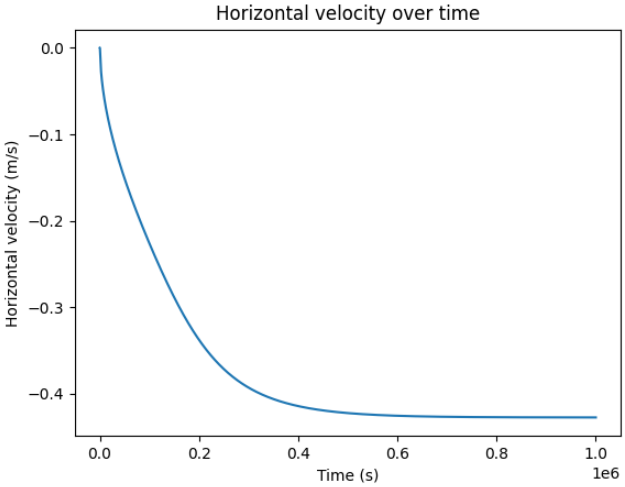


Figure 41: Horizontal velocities for the center probe

Just as before, the horizontal velocity at the center seems to converge slower than for flow case A, but it eventually reaches approximately the same value, where it stays the same for the remainder of the simulation.

3.2.3 Bottom wall moving in the negative x-direction

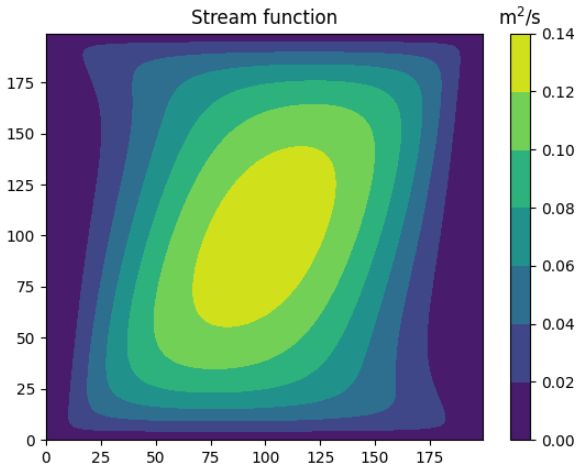


Figure 42: Stream function

The tilt to the right observed in Figure 18 is more pronounced in Figure 42, however the diagonal symmetry seems to be maintained.

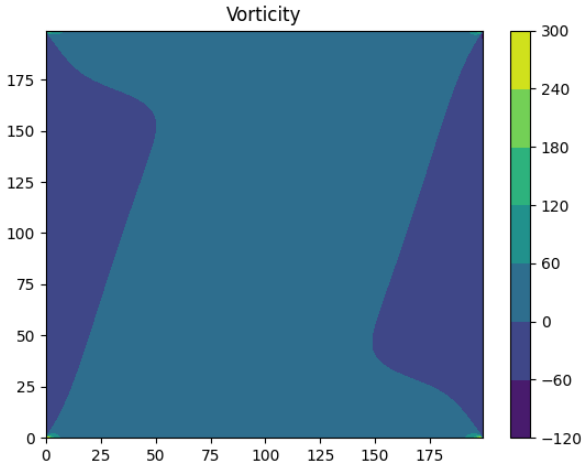


Figure 43: Vorticity

The vorticity reflects this more pronounced tilt and contours the stream function.

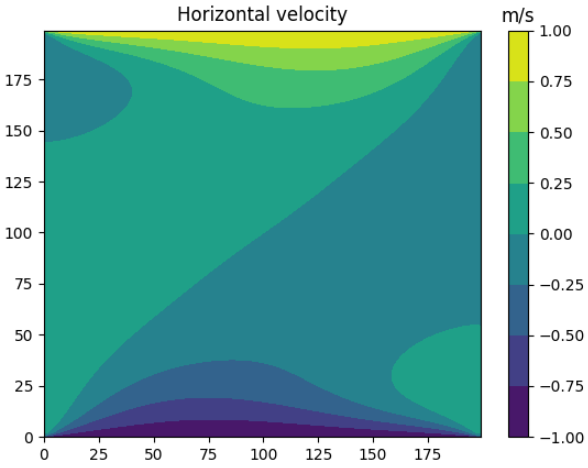


Figure 44: Horizontal velocities

The horizontal velocities close to the moving walls behave the same as before. A new phenomenon is the diagonal line dividing two areas with opposite velocities. Furthermore within these new areas are two rounded shapes where the velocity is again flipped as opposed to the surrounding.

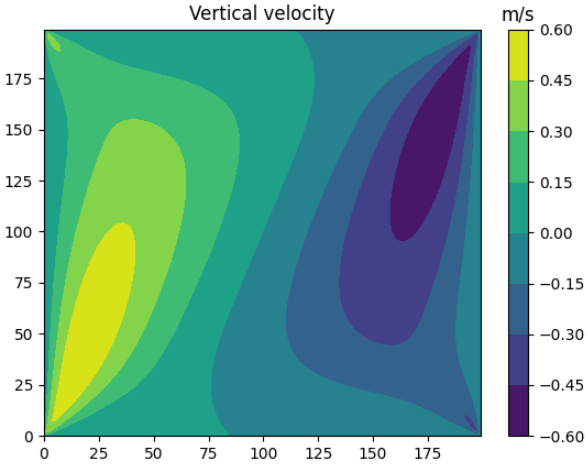


Figure 45: Vertical velocities

Just as for the horizontal velocity, the diagonal symmetry is maintained. For the vertical velocities there are now two large regions where the velocity is relatively high, emanating outwards and two small areas in the top-left and bottom right corners.

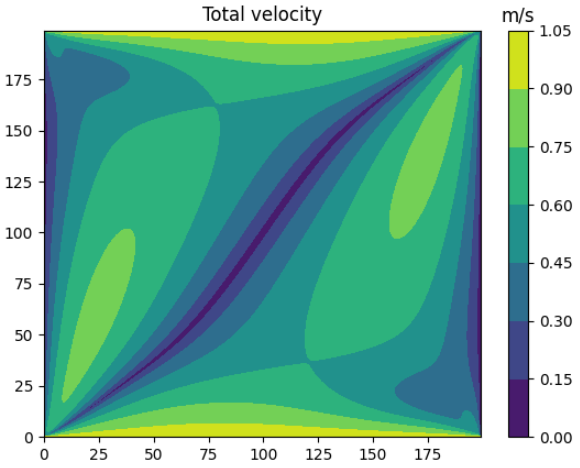


Figure 46: Total velocities

The combined contributions of the horizontal and vertical velocities are shown in Figure 46.

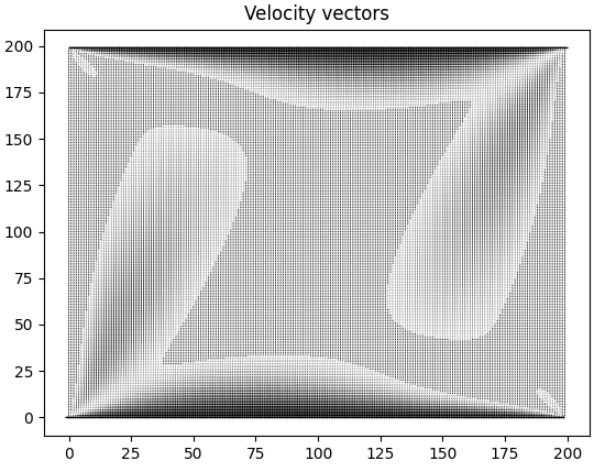


Figure 47: Velocity vector field

The velocity vector field due to the resolution is only useful to show the combination of the horizontal and vertical velocities. Similar to the total velocity plot.

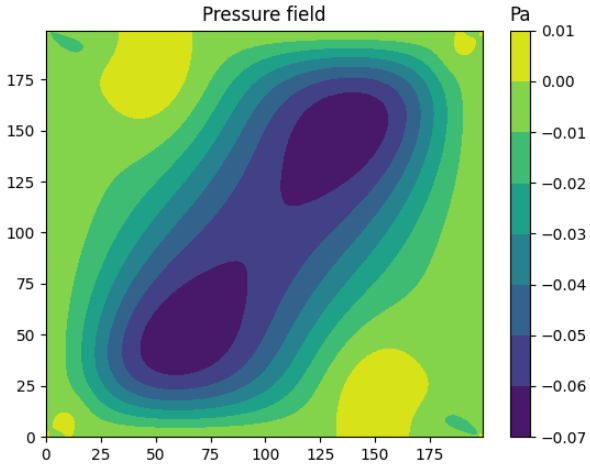


Figure 48: Pressure field

The main vortex again seems to contain two negative pressure fields. Furthermore there are multiple positive pressure fields near the corners.

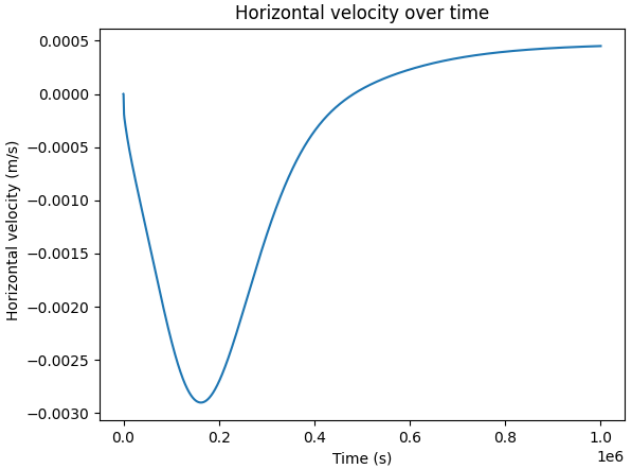


Figure 49: Horizontal velocities for the center probe

The horizontal velocity at the center seems to follow the same pattern as for flow case A. However the horizontal velocity at the last time is drastically different. It is now around 0.0004 m/s where it was negative in Figure 25.

4 Discussion

In this discussion section we will quickly interpret and try to explain the results, after which we discuss the computational method and the quality of the results.

4.1 Interpretation of the results

We will discuss the results of flow case A (Section 3.1) and flow case B (3.2) by looking at the different scenarios.

4.1.1 Bottom wall stationary

In the first scenario, the bottom wall was stationary, and only the top wall (lid) was moving in the positive x-direction (as in all scenarios). The resulting plots for flow case A showed a central vortex, with nearly symmetrical streamlines lines due to the symmetries of Equation (1) when the Reynolds number is closer to zero. When the Reynolds number is larger, as in flow case B, the symmetry is destroyed because of the increase in inertial effects — the vortices that arise are 'pulled' to the top-right corner, where the (absolute) vertical velocity is larger. Both flow cases seem to converge, but the higher Reynolds number in flow case B seems to slow down the convergence. This hints at an inverse relation between the convergence speed and Reynolds number, where higher value for Reynolds numbers imply slower convergence. Another relation that we found is that the velocities seem to be tangent to the streamlines as shown in Figure 50, below. The situation where the bottom wall is stationary is the only situation where we observed additional, secondary vortices by looking at the stream function.

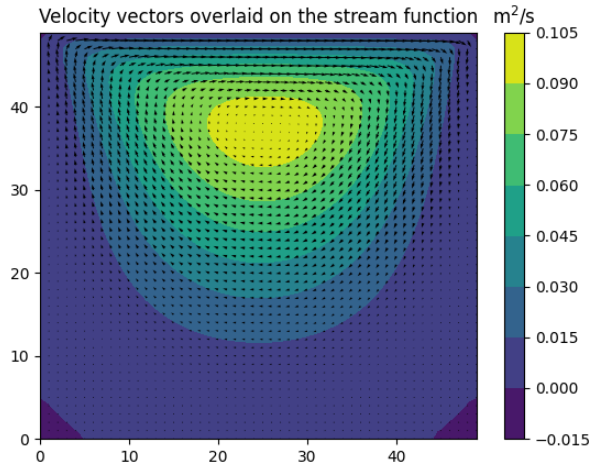


Figure 50: Velocity vectors overlaid on the stream function contours for N=50, Re=10

4.1.2 Bottom wall moving in the positive x-direction

When the bottom walls is moving identically to the top wall we get two similar vortices, where the top one moves in clockwise direction and the bottom vortex moves in counter-clockwise direction. For the higher Reynolds number the centers of the top and bottom vortices seem to move to the right, which is due to the viscosity of the fluid decreases due to high strain rates [3]. For flow case A the stream function plot is symmetric around both the vertical and horizontal center lines. Just as for the stationary bottom wall, the simulation converges, but it is slowed down by the increase in the value of Reynolds number. For flow case B the streamlines are only symmetrical around the horizontal center line. This horizontal symmetry is very intuitive as both walls move parallel to eachother.

4.1.3 Bottom wall moving in the negative x-direction

For the final scenario the bottom and top wall are moving in opposite directions. For this case, as opposed to the parallel moving walls, a single, vertical vortex emerges that occupies a significant fraction of the cavity. Similar to the previous cases, higher values for Re slow down the convergence. Furthermore due to inertial effects previously observed, the vortex tilts to the right as the walls are moving in opposite directions. Looking at the pressure fields there seem to be two smaller vortices that can not be discerned by only looking at the stream function. These smaller vortices seem to be close to merging when the Reynolds number increases. One more observation is that the stream function plot seems to be symmetric around the main diagonal for both flow cases, where for the other scenarios the symmetry (partly) breaks when Reynolds number goes up.

4.2 The algorithm

We went through the literature to compare our results. Using [4], [5] we found that the results for the scenario where only the top wall is moving agree. Furthermore, the scenarios where the bottom wall is also moving seem to be correct as well as they agree with the plots provided in [3].

As such we think that using the finite difference method is a good way to approximate the Vorticity form of the Navier-Stokes equations, and simulate the lid-driven cavity flow. As for the implementation, Python was not the optimal language as run-time is very important for larger simulations. Luckily, we were able to speed up most loops by vectorizing them using `numpy`.

5 Conclusion

To conclude, we set out to solve the two-dimensional lid-driven cavity flow problem using finite differences. To do so, we first introduced the problem and discussed the governing equations, boundary conditions and model parameters. Then we discretized the stream function and voriticity field given by the in-compressible Navier-Stokes equation (Equation (1)), as well as the boundary conditions for the simulation. To minimize the approximation error, we used the central difference scheme whenever possible. The implementation was discussed using pseudocode, as well as snippets from the source code. The main time-loop of the simulation consists of three steps:

1. Finding the solution to the stream function by solving the system of linear equations through relaxation,
2. Updating the vorticity at the boundaries,
3. Updating the vorticity on the interior.

After the main time-loop, in a post-processing step, we computed the velocity and pressure profiles. The pressure field was computed using relaxation, just like the stream function. Regular relaxation was used over successive over-relaxation (SOR), because the SOR method could not be vectorized. To speed up the simulation we employed various optimization techniques in our implementation. The most impactful optimization of which was vectorizing all possible loops, resulting in a speed-up of orders of magnitude.

In the results section we showed plots for two flow cases (i.e. sets of different model parameters) and observed symmetries present in different scenarios. These scenarios include one where only the top wall was moving, one where the top wall and bottom walls were moving at the same velocity in the same direction, and one where the top and bottom walls where moving in opposite directions (at unit velocity). We also looked at the number of visible vortices and their directions.

In the discussion we examined some of the effects observed in the different scenarios. We discussed the effects of increasing Reynolds number, and found that increasing Reynolds number resulted in slower convergence of the simulation. Furthermore, we discussed the algorithm and the implementation.

To conclude, the results seem to correspond (qualitatively) to the available literature [4], [5], [3], and we succeeded in finding the velocity and pressure profiles. The finite difference method worked out well for this problem, albeit that it does not scale well for larger grid sizes. Alternative methods such as the (integral) finite volume method can be considered to decrease the error obtained with the finite difference method, or when the grid is irregular.

A Source code

```
import numpy as np
import matplotlib.pyplot as plt
import time
import math

# Simulation parameters
N = 200
TIME_STEP = 0.00001
MAX_TIME = 1000
RE = 100

# Relaxation parameters
MAX_ITERATIONS = 10
MAX_ERROR = 10 ** -6

# Successive Over Relaxation (SOR) parameters
OMEGA = 1.8
MAX_ITERATIONS_SOR = 1000
MAX_ERROR_SOR = 0.0001

# Velocities of walls in the positive x direction
VELOCITY_TOP = 1
VELOCITY_BOTTOM = 0
VELOCITY_LEFT = VELOCITY_RIGHT = 0


def sor_stream_function(sf, vt, h):
    """
        Update stream function (sf) using Successive Over Relaxation (SOR) to solve
↪   the system of linear equations

        Args:
            sf: stream function
            vt: vorticity
            h: grid cell length, 1 / grid max length
    """

    for _ in range(MAX_ITERATIONS_SOR):
        sf_old = sf.copy()

        for i in range(1, N - 1):
            for j in range(1, N - 1):
                sf[i, j] = (0.25 * OMEGA * (sf[i + 1, j] + sf[i - 1, j] + sf[i, j +
                ↪  1] + sf[i, j - 1] + h * h * vt[i, j]) + (1 - OMEGA) * sf[i, j])
```

```
        residual = np.sum(np.abs(sf_old - sf))
        if residual <= MAX_ERROR_SOR:
            break

    return sf


def relax_stream_function(sf, vt, h):
    """
        Update stream function (sf) using ordinary relaxation to solve the system
↪   of linear equations
        This function does not use Successive Over Relaxation (SOR), which enables
↪   it to use
        numpy vectors for a significant speed-up

        Args:
            sf: stream function
            vt: vorticity
            h: grid cell length, 1 / grid max length
    """

    for _ in range(MAX_ITERATIONS):
        sf_old = sf.copy()

        sf[1:-1, 1:-1] = (0.25 * (sf[2:, 1:-1] + sf[0:-2, 1:-1] + sf[1:-1, 2:] +
        ↪  sf[1:-1, 0:-2] + h * h * vt[1:-1, 1:-1]))

        residual = np.sum(np.abs(sf_old - sf))
        if residual <= MAX_ERROR:
            break

    return sf


def update_vorticity_boundaries(sf, vt, h):
    """
        Update vorticity on the boundaries

        Args:
            sf: stream function
            vt: vorticity
            h: grid cell length, 1 / grid max length
    """

    # Top wall (moving lid)
```

```
    vt[N - 1, 1:N - 1] = -2 * sf[N - 2, 1:N - 1] / (h * h) + VELOCITY_TOP * 2 / h
    # Bottom wall
    vt[0, 1:N - 1] = -2 * sf[1, 1:N - 1] / (h * h) - VELOCITY_BOTTOM * 2 / h
    # Left wall
    vt[1:N - 1, 0] = -2 * sf[1:N - 1, 1] / (h * h) + VELOCITY_LEFT * 2 / h
    # Right wall
    vt[1:N - 1, N - 1] = -2 * sf[1:N - 1, N - 2] / (h * h) - VELOCITY_RIGHT * 2 / h

    return vt


def update_vorticity_interior(sf, vt, w, h):
    """
        Update vorticity on the interior

        Args:
            sf: stream function
            vt: vorticity
            w: old right hand side
            h: grid cell length, 1 / grid max length
    """

    # Calculate right hand side
    w[1:-1, 1:-1] = (-(((sf[1:-1, 2:] - sf[1:-1, 0:-2]) * (vt[2:, 1:-1] - vt[0:-2,
    ↪  1:-1]) - (sf[2:, 1:-1] - sf[0:-2, 1:-1]) * (vt[1:-1, 2:] - vt[1:-1, 0:-2]))
    ↪  / (4 * h * h)) + (1 / RE) * ((vt[2:, 1:-1] + vt[0:-2, 1:-1] + vt[1:-1, 2:]
    ↪  + vt[1:-1, 0:-2] - 4 * vt[1:-1, 1:-1]) / (h * h)))

    # Update interior vorticity
    vt += TIME_STEP * w

    return w, vt


def calculate_velocity(sf, h):
    velocity_x = np.zeros((N, N), dtype='float64')
    velocity_y = np.zeros((N, N), dtype='float64')

    # Set horizontal velocity at the boundaries
    velocity_x[N - 1, :] = VELOCITY_TOP
    velocity_x[0, :] = VELOCITY_BOTTOM
    velocity_y[:, 0] = VELOCITY_LEFT
    velocity_y[:, N - 1] = VELOCITY_RIGHT

    # Calculate velocity for the interior
```

```
    # Origin is at bottom left instead of top left, so the x and y velocities are
    ↪ swapped
    velocity_x[1:-1, 1:-1] = -(sf[2:, 1:-1] - sf[0:-2, 1:-1]) / (2 * h)
    velocity_y[1:-1, 1:-1] = (sf[1:-1, 2:] - sf[1:-1, 0:-2]) / (2 * h)
    velocity_total = np.sqrt(np.absolute(velocity_x + velocity_y))

    return velocity_x, velocity_y, velocity_total


def sor_pressure_field(sf, h):
    """
        Update pressure field (pf) using Successive Over Relaxation (SOR) to solve
↪   the system of linear equations

        Args:
            sf: stream function
            h: grid cell length, 1 / grid max length
    """

    pf = np.zeros((N, N), dtype='float64')

    for _ in range(MAX_ITERATIONS_SOR):
        pf_old = pf.copy()

        for i in range(1, N - 1):
            for j in range(1, N - 1):
                rhs = calculate_pressure_field(sf, h)

                pf[i, j] = (OMEGA * 0.25 * (pf_old[i + 1, j] + pf[i - 1, j] +
                ↪ pf_old[i, j + 1] + pf[i, j - 1] - h * h * rhs[i, j]) + (1 -
                ↪ OMEGA) * pf_old[i, j])

        if np.sum(np.abs(pf_old - pf)) <= MAX_ERROR_SOR:
            break

    return pf


def relax_pressure_field(sf, h):
    """
        Update pressure field (pf) using ordinary relaxation to solve the system of
↪   linear equations
        This function does not use Successive Over Relaxation (SOR), which enables
↪   it to use
        numpy vectors for a significant speed-up
```

```
        Args:
            sf: stream function
            h: grid cell length, 1 / grid max length
    """

    pf = np.zeros((N, N), dtype='float64')

    for _ in range(MAX_ITERATIONS):
        pf_old = pf.copy()
        rhs = calculate_pressure_field(sf, h)

        pf[1:-1, 1:-1] = 0.25 * (pf_old[2:, 1:-1] + pf[0:-2, 1:-1] + pf_old[1:-1,
        ↪  2:] + pf[1:-1, 0:-2] - h * h * rhs[1:-1, 1:-1])

        if np.sum(np.abs(pf_old - pf)) <= MAX_ERROR:
            break

    return pf


def calculate_pressure_field(sf, h):
    """
        Calculate pressure field (pf) by solving the Poisson equation

        Args:
            sf: stream function
            h: grid cell length, 1 / grid max length
    """

    pressure_field = np.zeros((N, N), dtype='float64')
    pressure_field[1:-1, 1:-1] = 2 * (((sf[2:, 1:-1] + sf[0:-2, 1:-1] - 2 *
    ↪  sf[1:-1, 1:-1]) / (h * h)) * ((sf[1:-1, 2:] + sf[1:-1, 0:-2] - 2 * sf[1:-1,
    ↪  1:-1]) / (h * h)) - ((sf[2:, 2:] - sf[2:, 0:-2] - sf[0:-2, 2:] + sf[0:-2,
    ↪  0:-2]) / (4 * h * h)) ** 2)

    return pressure_field


def plot_results(velocity_x, velocity_y, velocity_total, pressure_field, sf, vt,
↪  horizontal_velocities_center):
    plt.title('Horizontal velocity')
    velocity_plot = plt.contourf(velocity_x)
    clb = plt.colorbar(velocity_plot)
    clb.ax.set_title('m/s')
    plt.show()
```

```
plt.title('Vertical velocity')
velocity_plot = plt.contourf(velocity_y)
clb = plt.colorbar(velocity_plot)
clb.ax.set_title('m/s')
plt.show()

plt.title('Total velocity')
velocity_plot = plt.contourf(velocity_total)
clb = plt.colorbar(velocity_plot)
clb.ax.set_title('m/s')
plt.show()

plt.title('Velocity vectors')
plt.quiver(velocity_x, velocity_y)
plt.show()

plt.title('Pressure field')
pressure_field_plot = plt.contourf(pressure_field)
clb = plt.colorbar(pressure_field_plot)
clb.ax.set_title('Pa')
plt.show()

plt.title('Velocity vectors overlaid on the pressure field')
pressure_field_plot = plt.contourf(pressure_field)
clb = plt.colorbar(pressure_field_plot)
clb.ax.set_title('Pa')
plt.quiver(velocity_x, velocity_y, color='k')
plt.show()

plt.title('Stream function')
sf_plot = plt.contourf(sf)
clb = plt.colorbar(sf_plot)
clb.ax.set_title(r'$\mathregular{m^2/s}$')
plt.show()

plt.title('Velocity vectors overlaid on the stream function')
sf_plot = plt.contourf(sf)
clb = plt.colorbar(sf_plot)
clb.ax.set_title(r'$\mathregular{m^2/s}$')
plt.quiver(velocity_x, velocity_y, color='k')
plt.show()

plt.title('Vorticity')
vt_plot = plt.contourf(vt)
plt.colorbar(vt_plot)
plt.show()
```

```
    plt.title('Velocity vectors overlaid on the vorticity')
    vt_plot = plt.contourf(vt)
    plt.colorbar(vt_plot)
    plt.quiver(velocity_x, velocity_y, color='k')
    plt.show()

    plt.title('Horizontal velocity over time')
    plt.ylabel('Horizontal velocity (m/s)')
    plt.xlabel('Time (s)')
    plt.plot(horizontal_velocities_center)
    plt.show()


def main():
    start = time.time()

    # Initialize stream function (sf), vorticity (vt) and a matrix w
    # that is used to hold the rhs of the vorticity equation (1b)
    sf = np.zeros((N, N), dtype='float64')
    w = np.zeros((N, N), dtype='float64')
    vt = np.zeros((N, N), dtype='float64')

    # Grid cell length
    h = 1 / N

    velocity_center_old = 0
    horizontal_velocities_center = []

    max_steps = math.floor(MAX_TIME / TIME_STEP)
    for i in range(max_steps):
        # Calculate time on each iteration to prevent accumulating floating point
        ↪  precision errors
        t = i * TIME_STEP
        if i % 1000 == 0:
            print(t)
            np.save(f'saves/SF_N={N}_RE={RE}_TIME_STEP={TIME_STEP}', sf)
            np.save(f'saves/VT_N={N}_RE={RE}_TIME_STEP={TIME_STEP}', vt)
            np.save(f'saves/velocities_N={N}_RE={RE}_TIME_STEP={TIME_STEP}',
            ↪  horizontal_velocities_center)

        sf = relax_stream_function(sf, vt, h)
        vt = update_vorticity_boundaries(sf, vt, h)
        w, vt = update_vorticity_interior(sf, vt, w, h)

        # Probe center for horizontal velocity
```

```
            velocity_center = -(sf[N // 2 + 1, N // 2] - sf[N // 2 - 1, N // 2]) / (2 *
            ↪  h)
            horizontal_velocities_center.append(velocity_center)

        print(f'Elapsed time: {time.time() - start}')

        # Save results for future plotting/post-processing
        np.save(f'saves/SF_N={N}_RE={RE}_TIME_STEP={TIME_STEP}', sf)
        np.save(f'saves/VT_N={N}_RE={RE}_TIME_STEP={TIME_STEP}', vt)
        np.save(f'saves/velocities_N={N}_RE={RE}_TIME_STEP={TIME_STEP}',
        ↪  horizontal_velocities_center)

        # Compute pressure and velocity from stream function
        velocity_x, velocity_y, velocity_total = calculate_velocity(sf, h)
        pressure_field = relax_pressure_field(sf, h)

        # Plot results
        plot_results(velocity_x, velocity_y, velocity_total, pressure_field, sf, vt,
        ↪  horizontal_velocities_center)


main()
```

B Vectorization speed test code

```
import numpy as np
import time


N = 30
h = 1 / N
RE = 10
TIME_STEP = 0.002

VELOCITY_TOP = 1
VELOCITY_BOTTOM = 0
VELOCITY_LEFT = VELOCITY_RIGHT = 0


def with_loop():
    sf = np.load(f'saves/SF_N={N}_RE={RE}_TIME_STEP={TIME_STEP}.npy')
    vt = np.load(f'saves/VT_N={N}_RE={RE}_TIME_STEP={TIME_STEP}.npy')
    np.array(sf, 'float64')
    np.array(vt, 'float64')
    w = np.zeros((N, N), dtype='float64')
    for _ in range(10000):
        # Update vorticity on the boundaries
        # Top wall (moving lid)
        vt[N - 1, 1:N-1] = -2 * sf[N - 2, 1:N-1] / (h ** 2) + VELOCITY_TOP * 2 / h
        # Bottom wall
        vt[0, 1:N-1] = -2 * sf[1, 1:N-1] / (h ** 2) - VELOCITY_BOTTOM * 2 / h
        # Left wall
        vt[1:N-1, 0] = -2 * sf[1:N-1, 1] / (h ** 2) + VELOCITY_LEFT * 2 / h
        # Right wall
        vt[1:N-1, N - 1] = -2 * sf[1:N-1, N - 2] / (h ** 2) - VELOCITY_RIGHT * 2 /
        ↪  h

        # Update vorticity on the interior
        for i in range(1, N - 1):
            for j in range(1, N - 1):
                w[i][j] = (-0.25 * (((sf[i][j + 1] - sf[i][j - 1]) * (vt[i + 1][j]
                ↪  - vt[i - 1][j])
                                    + (sf[i + 1][j] - sf[i - 1][j]) * (vt[i][j + 1]
                                    ↪  - vt[i][j - 1])) / h ** 2)
                           + (1 / RE) * ((vt[i + 1][j] + vt[i - 1][j] + vt[i][j +
                           ↪  1] + vt[i][j - 1] - 4 * vt[i][j])
                                         / h ** 2))

        for i in range(1, N - 1):
```

```
            for j in range(1, N - 1):
                vt[i][j] += TIME_STEP * w[i][j]


def with_numpy():
    sf = np.load(f'saves/SF_N={N}_RE={RE}_TIME_STEP={TIME_STEP}.npy')
    vt = np.load(f'saves/VT_N={N}_RE={RE}_TIME_STEP={TIME_STEP}.npy')
    w = np.zeros((N, N))
    for _ in range(10000):
        vt[N - 1, 1:N-1] = -2 * sf[N - 2, 1:N-1] / (h ** 2) + VELOCITY_TOP * 2 / h
        # Bottom wall
        vt[0, 1:N-1] = -2 * sf[1, 1:N-1] / (h ** 2) - VELOCITY_BOTTOM * 2 / h
        # Left wall
        vt[1:N-1, 0] = -2 * sf[1:N-1, 1] / (h ** 2) + VELOCITY_LEFT * 2 / h
        # Right wall
        vt[1:N-1, N - 1] = -2 * sf[1:N-1, N - 2] / (h ** 2) - VELOCITY_RIGHT * 2 /
        ↪  h

        # Update vorticity on the interior
        w[1:N-1, 1:N-1] = (-0.25 * (((sf[1:N-1, 2:N] - sf[1:N-1, 0:N-2]) * (vt[2:N,
        ↪  1:N-1] - vt[0:N-2, 1:N-1])
                                    + (sf[2:N, 1:N-1] - sf[0:N-2, 1:N-1]) *
                                    ↪  (vt[1:N-1, 2:N] - vt[1:N-1, 0:N-2]))
                                   / h ** 2)
                           + (1 / RE) * ((vt[2:N, 1:N-1] + vt[0:N-2, 1:N-1]
                                          + vt[1:N-1, 2:N] + vt[1:N-1, 0:N-2]
                                          - 4 * vt[1:N-1, 1:N-1])
                                         / h ** 2))

        vt[1:N-1][1:N-1] = vt[1:N-1][1:N-1] + TIME_STEP * w[1:N-1][1:N-1]


def main():
    start = time.time()
    with_numpy()
    print(f'Time using numpy: {time.time() - start}')

    start = time.time()
    with_loop()
    print(f'Normal time: {time.time() - start}')


main()
```

Bibliography

[1] Ercan Erturk, Thomas Corke, and C. Gokcol. Numerical solutions of 2-d steady incompressible driven cavity flow at high reynolds numbers. *International Journal for Numerical Methods in Fluids*, 48:747 – 774, 07 2005.

[2] Numpy. *Why is numpy fast?* `https://numpy.org/devdocs/user/whatisnumpy.html#why-is-numpy-fast`.

[3] Siva Subrahmanyam Mendu and P. Das. Flow of power-law fluids in a cavity driven by the motion of two facing lids – a simulation by lattice boltzmann method. *Journal of Non-newtonian Fluid Mechanics*, 175:10–24, 2012.

[4] Mitutosi Kawaguti. Numerical solution of the navier-stokes equations for the flow in a two-dimensional cavity. *Journal of the Physical Society of Japan*, 16(11):2307–2315, 1961.

[5] Odus R. Burggraf. Analytical and numerical studies of the structure of steady separated flows. *Journal of Fluid Mechanics*, 24(1):113–151, 1966.