Technische Universität München
Institut für Informatik
Dr. Miriam Mehl
Tobias Neckel
Tobias Weinzierl

ST 2008

Praktikum Wissenschaftliches Rechnen Computational Fluid Dynamics

Worksheet 1 (deadline May 8)

1 Statement of the Problem: Incompressible Viscous Flow

In the present Practical Course, we shall deal with basic equations aimed at the description of non-stationary laminar flows of a viscous fluid. We shall restrict ourselves to *incompressible* media, such as water. The air is an example of a compressible medium, when the same mass of a fluid can fill up different volumes, depending on pressure and temperature. Water naturally is (at very high pressure) also compressible; yet, under *normal conditions* one may consider it incompressible. The viscosity describes the stickiness of the medium. Thus, air is characterized by a very small viscosity, water has a somewhat larger one, and honey possesses a very large viscosity. When the flow velocity is not too high and the viscosity is not too small, flows are observed to be relatively regular and are not mixing intensively. They are said to be *laminar* and not *turbulent*. The term *non-stationary* signifies that the flow varies with time, in contrast to stationary flows.

One can see examples of the flow of an incompressible viscous fluid in Fig. 1.

2 The Mathematical Model: The Navier-Stokes Equation

Non-stationary incompressible viscous fluids are described by the Navier-Stokes equations. For simplicity, we shall limit our consideration to the two-dimensional case and carry out our analysis in Cartesian coordinates. The quantities to be computed are

- u, the fluid velocity in x–direction,
- v, the fluid velocity in y–direction,
- p the pressure.

The Navier-Stokes equations consist of two *momentum equations*

$$\frac{\partial u}{\partial t} + \frac{\partial p}{\partial x} = \frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) - \frac{\partial (u^2)}{\partial x} - \frac{\partial (uv)}{\partial y} + g_x, \tag{1}$$

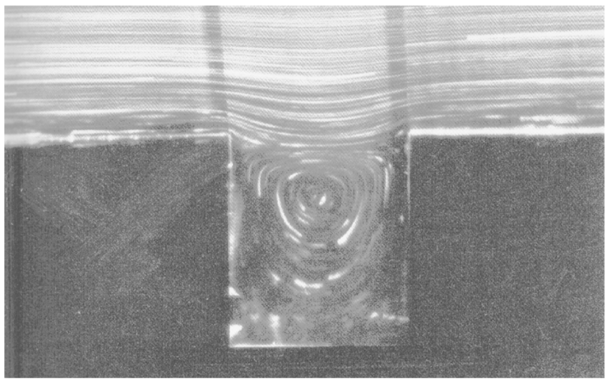
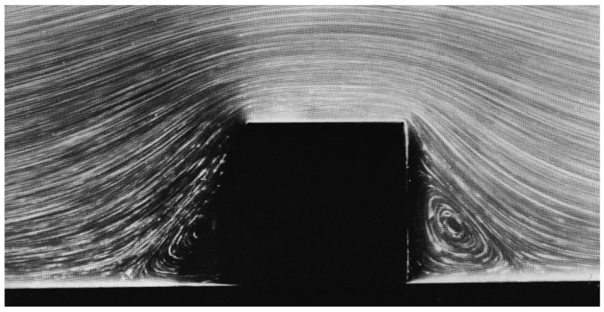

Figure 1: Left: Flow over a hollow space (driven cavity); right: flow over a step.

$$\frac{\partial v}{\partial t} + \frac{\partial p}{\partial y} = \frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) - \frac{\partial (uv)}{\partial x} - \frac{\partial (v^2)}{\partial y} + g_y \tag{2}$$

and the *continuity equation*

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0, \tag{3}$$

where g_x and g_y denote external forces, for example gravity or other body forces acting throughout the bulk of the system and producing acceleration in its parts.

The dimensionless real quantity *Re* is called the *Reynolds number*; it characterizes the fluid flow. The Reynolds number depends on viscosity, on the lengthscale of the scenario, and on average velocity of the fluid. The lower the *Re* value, the more viscous is the fluid.

Initial conditions:

At the initial moment ($t = 0$), initial conditions $u = u_0(x, y)$ and $v = v_0(x, y)$ satisfying (3) are given.

Boundary conditions:

Besides, supplementary conditions holding at all four boundaries of the region for all times are required, so that we get an *initial-boundary value problem.*

In order to formulate the boundary conditions, let us denote the velocity component perpendicular to the boundary (in normal direction) as w_1 and that parallel to the boundary (in tangential direction) as w_2 . The derivatives in the normal direction will be denoted as $\partial w_1/\partial \mathbf{n}$ and $\partial w_2/\partial \mathbf{n}$ respectively.

Then, for the vertical boundary, we have

$$w_1 = u, \qquad w_2 = v, \qquad \frac{\partial w_1}{\partial \mathbf{n}} = \frac{\partial u}{\partial x}, \qquad \frac{\partial w_2}{\partial \mathbf{n}} = \frac{\partial v}{\partial x},$$

and for the horizontal boundary

$$w_1 = v, \qquad w_2 = u, \qquad \frac{\partial w_1}{\partial \mathbf{n}} = \frac{\partial v}{\partial y}, \qquad \frac{\partial w_2}{\partial \mathbf{n}} = \frac{\partial u}{\partial y}.$$

For the points (x, y) on the rigid boundary $\Gamma := \partial\Omega$, one can formulate the following boundary conditions:

1. No-slip condition: $w_1(x, y) = 0,$ $w_2(x, y) = 0.$
   (the fluid velocity directed parallel to the boundary should vanish)
2. Free-slip condition: $w_1(x, y) = 0,$ $\partial w_2(x, y)/\partial \mathbf{n} = 0.$
   (there is no friction at the boundary)
3. Inflow condition: $w_1(x, y) = w_1^0,$ $w_2(x, y) = w_2^0.$
   (w_1^0, w_2^0 are given)
4. Outflow condition: $\partial w_1(x, y)/\partial \mathbf{n} = 0,$ $\partial w_2(x, y)/\partial \mathbf{n} = 0.$

If only the velocities and not their normal derivatives are given at the boundaries, then the surface (line) integral over the velocities normal to the boundaries should vanish, i.e.

$$\int_\Gamma \begin{pmatrix} u \\ v \end{pmatrix} \cdot \mathbf{n} ds = 0,$$

which is required by the continuity equation (3).

Fig. 2 demonstrates the results of a sample simulation. The first picture shows a driven cavity flow, the second one shows the Kármán vortex street. Please mind the importance of boundary conditions in each of these cases.

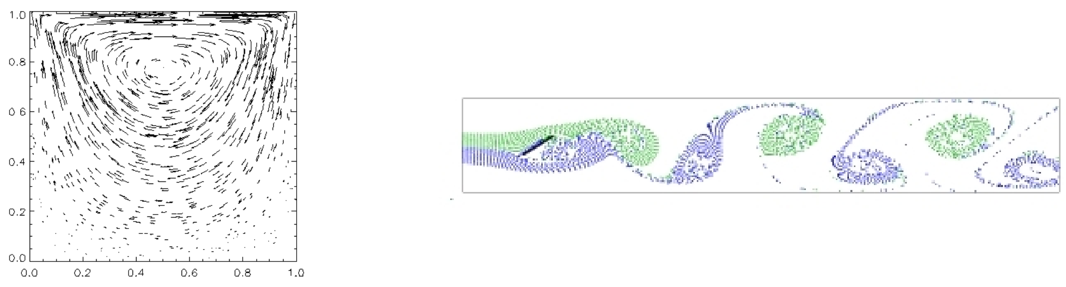

Figure 2: Left: driven cavity: no-slip conditions at the lower, left, and right boundary, $u = 1, v = 0$ at the upper boundary; right: Karman vortex street: no-slip conditions at the upper and lower boundary, $u = 1, v = 0$ at the left boundary, outflow conditions at the right boundary, no-slip conditions around the obstacle.

3 The Discretization of the Navier-Stokes Equations

3.1 The Grid

There exist a number of approaches for the discretization of the Navier-Stokes equations. We use a stable finite difference method based upon a so called staggered grid, that is the unknown variables u, v, and p lie at different positions in a reference grid. The reference grid subdivides the whole region into cells characterized by index (i,j) corresponding to the rectangle $[(i-1)\,\delta x, i\,\delta x] \times [(j-1)\,\delta y, j\,\delta y]$. The pressure p is associated to the cell's midpoint, the velocity u to the midpoint of vertical cell edges, and the velocity v to the midpoints of the horizontal cell edges. See Fig. 3 for this allocation of function values and for the assignement of the index (i,j) to the values of u and v.

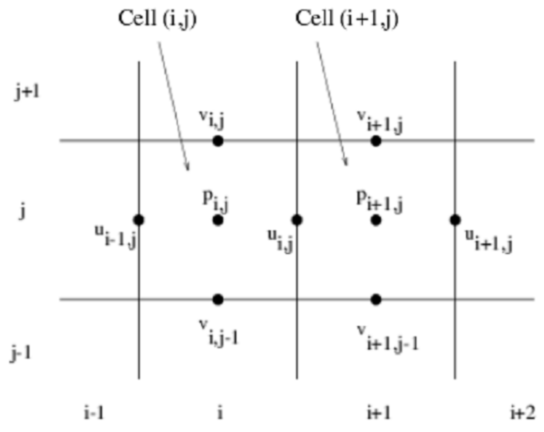

Figure 3: Staggered Grid for u, v, and p.

As a consequence, the boundary of our rectangular domain does not contain values of all unknoen functions (vertical boundary lines contain only u-values, whereas level lines only v-values; p-values are never located at boundary lines). For this reason, one more boundary strip of grid cells is introduced (see Figure 6), so that the boundary conditions may be satisfied by averaging the neighboring grid points on either side (see below).

Further, the time interval $[0, t_{end}]$ is replaced by discrete time points, $0 = t_1 < t_2 < \ldots < t_N = t_{end}$, such that values of u, v and p will be considered only at times t_n, $n = 0, 1, \ldots, N$. These time points, in contrast to spatial discretization, cannot be chosen arbitrarily but must be adjusted to the actual state of the flow. We shall discuss the time-step control below.

3.2 Spatial Discretization

Now, the Navier-Stokes equations will be discretized in the following fashion: at first, we shall handle the spatial derivatives. The momentum equation (1) will be evaluated at the vertical edge midpoints, the momentum equation (2) at the horizontal edge midpoints, and the continuity equation (3) at

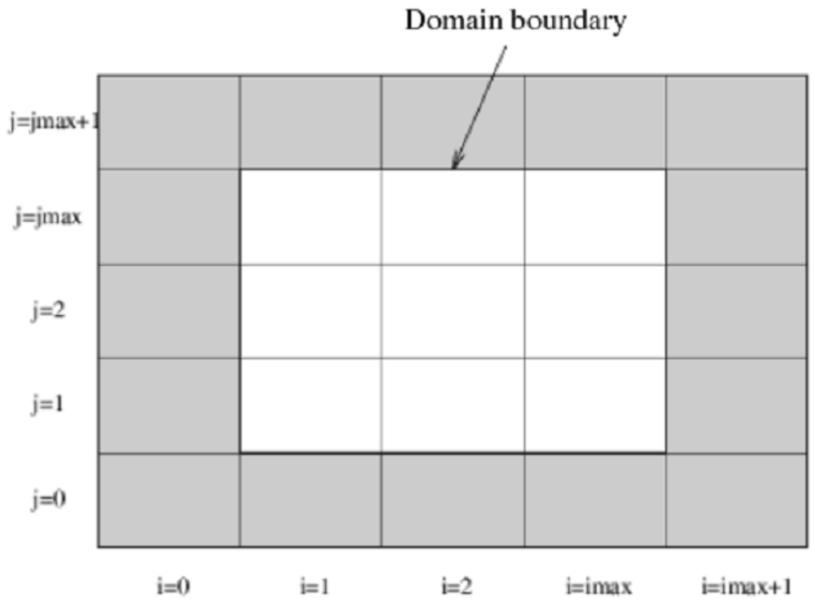

Figure 4: Domain with boundary cells.

the cell midpoints. We replace the expression in equation (1) taken at the midpoint of the right edge of cell (i, j), $i = 1, \ldots, imax - 1$, $j = 1 \ldots, jmax$, by

$$\begin{aligned}
\left[\frac{\partial(u^2)}{\partial x}\right]_{i,j} &:= \frac{1}{\delta x}\left(\left(\frac{u_{i,j}+u_{i+1,j}}{2}\right)^2 - \left(\frac{u_{i-1,j}+u_{i,j}}{2}\right)^2\right) + \\
&\quad \alpha\frac{1}{\delta x}\left(\frac{|u_{i,j}+u_{i+1,j}|}{2}\frac{(u_{i,j}-u_{i+1,j})}{2} - \frac{|u_{i-1,j}+u_{i,j}|}{2}\frac{(u_{i-1,j}-u_{i,j})}{2}\right), \\
\left[\frac{\partial(uv)}{\partial y}\right]_{i,j} &:= \frac{1}{\delta y}\left(\frac{(v_{i,j}+v_{i+1,j})}{2}\frac{(u_{i,j}+u_{i,j+1})}{2} - \frac{(v_{i,j-1}+v_{i+1,j-1})}{2}\frac{(u_{i,j-1}+u_{i,j})}{2}\right) + \\
&\quad \alpha\frac{1}{\delta y}\left(\frac{|v_{i,j}+v_{i+1,j}|}{2}\frac{(u_{i,j}-u_{i,j+1})}{2} - \frac{|v_{i,j-1}+v_{i+1,j-1}|}{2}\frac{(u_{i,j-1}-u_{i,j})}{2}\right), \qquad (4) \\
\left[\frac{\partial^2 u}{\partial x^2}\right]_{i,j} &:= \frac{u_{i+1,j}-2u_{i,j}+u_{i-1,j}}{(\delta x)^2}, \\
\left[\frac{\partial^2 u}{\partial y^2}\right]_{i,j} &:= \frac{u_{i,j+1}-2u_{i,j}+u_{i,j-1}}{(\delta y)^2}, \qquad \left[\frac{\partial p}{\partial x}\right]_{i,j} := \frac{p_{i+1,j}-p_{i,j}}{\delta x}.
\end{aligned}$$

For the expressions in equation (2), we set at the midpoint of the upper edge of cell (i, j), $i =$

$1, \ldots, imax$, $j = 1, \ldots, jmax - 1$

$$\begin{aligned}
\left[\frac{\partial(uv)}{\partial x}\right]_{i,j} &:= \frac{1}{\delta x}\left(\frac{(u_{i,j}+u_{i,j+1})}{2}\frac{(v_{i,j}+v_{i+1,j})}{2} - \frac{(u_{i-1,j}+u_{i-1,j+1})}{2}\frac{(v_{i-1,j}+v_{i,j})}{2}\right) + \\
&\quad \alpha\frac{1}{\delta x}\left(\frac{|u_{i,j}+u_{i,j+1}|}{2}\frac{(v_{i,j}-v_{i+1,j})}{2} - \frac{|u_{i-1,j}+u_{i-1,j+1}|}{2}\frac{(v_{i-1,j}-v_{i,j})}{2}\right), \\
\left[\frac{\partial(v^2)}{\partial y}\right]_{i,j} &:= \frac{1}{\delta y}\left(\left(\frac{v_{i,j}+v_{i,j+1}}{2}\right)^2 - \left(\frac{v_{i,j-1}+v_{i,j}}{2}\right)^2\right) + \\
&\quad \alpha\frac{1}{\delta y}\left(\frac{|v_{i,j}+v_{i,j+1}|}{2}\frac{(v_{i,j}-v_{i,j+1})}{2} - \frac{|v_{i,j-1}+v_{i,j}|}{2}\frac{(v_{i,j-1}-v_{i,j})}{2}\right), \qquad (5) \\
\left[\frac{\partial^2 v}{\partial x^2}\right]_{i,j} &:= \frac{v_{i+1,j}-2v_{i,j}+v_{i-1,j}}{(\delta x)^2}, \\
\left[\frac{\partial^2 v}{\partial y^2}\right]_{i,j} &:= \frac{v_{i,j+1}-2v_{i,j}+v_{i,j-1}}{(\delta y)^2}, \qquad \left[\frac{\partial p}{\partial y}\right]_{i,j} := \frac{p_{i,j+1}-p_{i,j}}{\delta y}
\end{aligned}$$

The terms in the continuity equation (3) are replaced at the midpoint of cell (i, j), $i = 1, \ldots, imax$, $j = 1, \ldots, jmax$, by

$$\left[\frac{\partial u}{\partial x}\right]_{i,j} := \frac{u_{i,j}-u_{i-1,j}}{\delta x}, \qquad \left[\frac{\partial v}{\partial y}\right]_{i,j} := \frac{v_{i,j}-v_{i,j-1}}{\delta y}. \qquad (6)$$

Here, α is a parameter with values between 0 and 1. For $\alpha = 0$, one gets the second-order approximation for differential operators, i.e. the approximation error has the accuracy $O((\delta x)^2)$ or $O((\delta y)^2)$. However, for small viscosity values, this approximation can result in oscillations in the solution. In such cases one must resort to a so-called donor-cell scheme ($\alpha = 1$), which only produces the first-order approximation. In practice, a mixture of both techniques is used, with $\alpha \in [0, 1]$. The parameter α should be selected slightly larger than the maximal value of $|u\,\delta t/\delta x|$ and $|v\,\delta t/\delta y|$.

3.3 Time Discretization

The discrete momentum equations

For the time discretization the momentum equations (1) and (2), we use the explicit Euler method:

$$\begin{aligned}
u_{i,j}^{(n+1)} &= F_{i,j}^{(n)} - \frac{\delta t}{\delta x}(p_{i+1,j}^{(n+1)} - p_{i,j}^{(n+1)}) \qquad (7) \\
&\qquad i = 1, \ldots, imax-1, \quad j = 1, \ldots, jmax; \\
v_{i,j}^{(n+1)} &= G_{i,j}^{(n)} - \frac{\delta t}{\delta y}(p_{i,j+1}^{(n+1)} - p_{i,j}^{(n+1)}) \qquad (8) \\
&\qquad i = 1, \ldots, imax, \quad j = 1, \ldots, jmax-1.
\end{aligned}$$

with the terms $F_{i,j}^{(n)}$ and $G_{i,j}^{(n)}$ containing the discretized right-hand sides of momentum equations

(1) and (2) evaluated using the a current velocities u^n and v^n: Here, the terms

$$F_{i,j} := u_{i,j} + \delta t \left(\frac{1}{Re} \left(\left[\frac{\partial^2 u}{\partial x^2}\right]_{i,j} + \left[\frac{\partial^2 u}{\partial y^2}\right]_{i,j} \right) - \left[\frac{\partial (u^2)}{\partial x}\right]_{i,j} - \left[\frac{\partial (uv)}{\partial y}\right]_{i,j} + g_x \right) \tag{9}$$
$$i = 1, \ldots, imax-1, \quad j = 1, \ldots, jmax;$$

$$G_{i,j} := v_{i,j} + \delta t \left(\frac{1}{Re} \left(\left[\frac{\partial^2 v}{\partial x^2}\right]_{i,j} + \left[\frac{\partial^2 v}{\partial y^2}\right]_{i,j} \right) - \left[\frac{\partial (uv)}{\partial x}\right]_{i,j} - \left[\frac{\partial (v^2)}{\partial y}\right]_{i,j} + g_y \right) \tag{10}$$
$$i = 1, \ldots, imax, \quad j = 1, \ldots, jmax-1.$$

The pressure equation

Equations (7) and (8) give the closed formulae to determine the new velocities $u_{i,j}^{(n+1)}$ and $v_{i,j}^{(n+1)}$ in terms of the old velocities $u_{i,j}^{(n)}$ and $v_{i,j}^{(n)}$. However, the pressure $p^{(n+1)}$ remains so far unknown. In addition, we did not make sure yet that the new velocities fulfill the continuity equation. Thus, we determine the pressure such that

$$\left[\frac{\partial u}{\partial x}\right]_{i,j}^{(n+1)} + \left[\frac{\partial v}{\partial y}\right]_{i,j}^{(n+1)} = 0.$$

This results in the pressure equation

$$\frac{p_{i+1,j}^{(n+1)} - 2p_{i,j}^{(n+1)} + p_{i-1,j}^{(n+1)}}{(\delta x)^2} + \frac{p_{i,j+1}^{(n+1)} - 2p_{i,j}^{(n+1)} + p_{i,j-1}^{(n+1)}}{(\delta y)^2} =$$
$$\frac{1}{\delta t}\left(\frac{F_{i,j}^{(n)} - F_{i-1,j}^{(n)}}{\delta x} + \frac{G_{i,j}^{(n)} - G_{i,j-1}^{(n)}}{\delta y} \right), \tag{11}$$
$$i = 1, \ldots, imax, \quad j = 1, \ldots, jmax.$$

This is the familiar form of the discretized Poisson equation for the quantity $p^{(n+1)}$

$$\frac{\partial^2 p^{(n+1)}}{\partial x^2} + \frac{\partial^2 p^{(n+1)}}{\partial y^2} = rs$$

on a domain Ω, with an arbitrary right-hand side rs. To ensure the uniqiness of the solution, we also need the boundary values $p_{i,j}$ $(i \in \{0, imax+1\}, j \in \{0, jmax+1\})$, $F_{i,j}$ $(i \in \{0, imax\})$ und $G_{i,j}$ $(j \in \{0, jmax\})$, which we can obtain from the momentum equations being considered at the boundary (see below).

The stability condition

In order to ensure the stability of the numerical algorithm and avoid oscillations, the following three stability conditions must be imposed on the stepsizes δx, δy, and δt:

$$\frac{2}{Re}\delta t < \frac{(\delta x)^2 (\delta y)^2}{(\delta x)^2 + (\delta y)^2}, \qquad |u_{max}|\,\delta t < \delta x, \qquad |v_{max}|\,\delta t < \delta y. \tag{12}$$

Here $|u_{max}|$ and $|v_{max}|$ are the maximal absolute values of the respective velocities. The latter two inequalities in (18) are called the Courant-Friedrichs-Levi (CFL) conditions.

One can use an adaptive stepsize control based on the above stability conditions. This is implemented by choosing δt for the next time step in such a way that each of the three conditions (18) is satisfied:

$$\delta t := \tau \min\left(\frac{Re}{2}\left(\frac{1}{\delta x^2}+\frac{1}{\delta y^2}\right)^{-1}, \frac{\delta x}{|u_{max}|}, \frac{\delta y}{|v_{max}|}\right). \tag{13}$$

The coefficient $\tau \in]0,1]$ is a safety factor. This stepsize control ensures, however, only the stability of the metod. In order to specify the accuracy, the stepsize control should be based on some error estimation procedure, which allows one to appraise the difference between numerical and analytical solutions.

3.4 Discrete Boundary Conditions

No-slip boundary conditions for u and v

While computing the quantities F and G for $i \in \{1, imax\}$ and $j \in \{1, jmax\}$ according to (9) and (10), one may reach the values of u and v that are lying on the domain's boundary or even outside of the domain. Here, we shall limit ourselves, at first, by no-slip conditions, when the continuous velocities must be zero everywhere at the boundary. Thus, we can set the following values for the velocities lying right on the boundary:

$$u_{0,j} = 0,\ u_{imax,j} = 0,\ j = 1, \ldots, jmax \qquad v_{i,0} = 0,\ v_{i,jmax} = 0,\ i = 1, \ldots, imax. \tag{14}$$

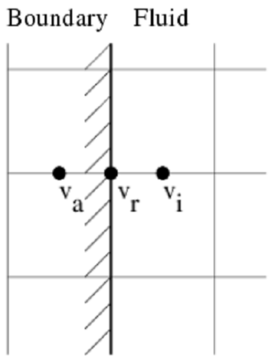

Figure 5: v-velocities around the left vertical boundary. Since no v-velocity lies on the vertical boundary, we use v_a and v_i to ensure the boundary condition: $v_r := \frac{v_a+v_i}{2} = 0 \Rightarrow v_a = -v_i$.

Furthermore, since no v-values lie on the vertical boundaries (compare Fig. 5 and no u-values lie on the horizontal boundaries, the boundary value zero is achieved by averaging the values on both sides of the boundary (see Fig. 5). Thus, we obtain the following conditions on four walls:

$$
\begin{array}{lll}
v_{0,j} = -v_{1,j}, & v_{imax+1,j} = -v_{imax,j}, & j = 1, \dots, jmax, \\
u_{i,0} = -u_{i,1}, & u_{i,jmax+1} = -u_{i,jmax}, & i = 1, \dots, imax.
\end{array} \tag{15}
$$

Boundary conditions for the pressure p

As it has been already mentioned, the boundary values for the pressure are derived from the discretized momentum equation, multiplying it by a respective normal vector. This operation gives

$$
\begin{array}{lll}
p_{0,j} = p_{1,j}, & p_{imax+1,j} = p_{imax,j}, & j = 1, \dots, jmax; \\
p_{i,0} = p_{i,1}, & p_{i,jmax+1} = p_{i,jmax}, & i = 1, \dots, imax
\end{array} \tag{16}
$$

and

$$
\begin{array}{lll}
F_{0,j} = u_{0,j}, & F_{imax,j} = u_{imax,j}, & j = 1, \dots, jmax, \\
G_{i,0} = v_{i,0}, & G_{i,jmax} = v_{i,jmax}, & i = 1, \dots, imax.
\end{array} \tag{17}
$$

4 SOR Solver

If we look at the result of our discretization, we see that in each time step we have to solve the Poisson equation (11) for the pressure represented by a large system of linear equations in the discrete formulation. For that, we can use any solution technique developed for systems of linear equations. Since the direct methods, as for example Gaussian elimination, lead to high computational costs for large problems, it is more customary to use iterative procedures. An example is the Gauss-Seidel method, in which, starting from some initial value, all the cells are successively run over in each cycle, and the pressure at (i, j) cell is adjusted in such a way that the corresponding equation should be exactly satisfied.

An improved variant is given by the SOR (successive over-relaxation) method, when the iteration step is given by the following loop over all cells:

```
$i = 1, \dots, imax$
    $j = 1, \dots, jmax$
```

$$
\begin{aligned}
p_{i,j}^{it+1} \quad := \quad & (1-\omega)\, p_{i,j}^{it} + \\
& \frac{\omega}{2(\frac{1}{(\delta x)^2} + \frac{1}{(\delta y)^2})} \left(\frac{p_{i+1,j}^{it} + p_{i-1,j}^{it+1}}{(\delta x)^2} + \frac{p_{i,j+1}^{it} + p_{i,j-1}^{it+1}}{(\delta y)^2} - rs_{i,j} \right)
\end{aligned} \tag{18}
$$

The upper indices it and $it + 1$ designate the iteration step number. Important: the old pressure value p^{it} will be right away overwritten by the updated value, p^{it+1}, i.e. there remains no saved copy of the pressure field.

The quantity $rs_{i,j}$ is the right-hand side of the pressure equation (11) for the (i, j) cell and ω is a parameter (relaxation factor), which must be chosen from the interval $]0, 2]$ (often the value $\omega = 1.7$ is used). For $\omega = 1$, the method is reduced to that of Gauss-Seidel. The iteration process stops either once the maximum number of iterations, $itmax$, is reached or when the residual

$$res := \left(\sum_{i=1}^{imax} \sum_{j=1}^{jmax} \left(\frac{p_{i+1,j} - 2p_{i,j} + p_{i-1,j}}{(\delta x)^2} + \frac{p_{i,j+1} - 2p_{i,j} + p_{i,j-1}}{(\delta y)^2} - rs_{i,j} \right)^2 / (imax \cdot jmax) \right)^{1/2} \tag{19}$$

becomes smaller than a tolerance value ε defined by the user.
As a starting value for iteration process to calculate the pressure $p^{(n+1)}$, any pressure value related to time level n can be taken.

5 The Algorithm

Summarizing the elements described above, we get the following algorithm:

```
Read the problem parameters
Set $t := 0$, $n := 0$
Assign initial values to $u, v, p$
While $t < t_{end}$
   Select $\delta t$ according to (13)
   Set boundary values for $u$ and $v$ according to (14),(15)
   Compute $F^{(n)}$ and $G^{(n)}$ according to (9),(10),(17)
   Compute the right-hand side $rs$ of the pressure equation (11)
   Set $it := 0$
   While $it < itmax$ and $res > \varepsilon$
      Set the boundary values for the pressure according to (16)
      Perform a SOR-cycle according to (18)
      Compute the residual $res$ for the pressure equation
      according to (19)
      $it := it + 1$
   Compute $u^{(n+1)}$ and $v^{(n+1)}$ according to (7),(8)
   Output of $u, v, p$ values for visualization, if necessary
   $t := t + \delta t$
   $n := n + 1$
Output of $u, v, p$ for visualization
```

6 Problem Parameters and Data Structures

In the offered Practical Course, the algorithm described above should be implemented in the C programming language. The algorithm requires the following quantities to be defined, which, with the exception of the variables `t`, `it`, `dx` and `dy`, will be provided by the input file:

- Geometry data:

`double xlength`	domain size in x–direction
`double ylength`	domain size in y–direction The computational domain is thus $\Omega = [0, \texttt{xlaenge}] \times [0, \texttt{ylaenge}]$.
`int imax`	number of interior cells in x–direction
`int jmax`	number of interior cells in y–direction
`double dx`	length δx of one cell in x–direction
`double dy`	length δy of one cell in y–direction

- Time-stepping data:

`double t`	current time value
`double t_end`	final time t_{end}
`double dt`	time step size δt
`double tau`	safety factor for time step size control τ
`double dt_value`	time interval for writing visualization data in a file

- Pressure iteration data:

`int itermax`	maximum number of pressure iterations in one time step
`int it`	SOR iteration counter
`double res`	residual norm of the pressure equation
`double eps`	accuracy criterion ε (tolerance) for pressure iteration (res < eps)
`double omg`	relaxation factor ω for SOR iteration
`double alpha`	upwind differencing factor α (see equation (4))

- Problem-dependent quantities:

`double Re`	Reynolds number *Re*
`double GX,GY`	external forces g_x, g_y, e.g. gravity
`double UI,VI,PI`	initial data for velocities and pressure

Furthermore, the following arrays will be used as data structures:

- Arrays

`double **U`	velocity in x–direction
`double **V`	velocity in y–direction
`double **P`	pressure
`double **RS`	right-hand side for pressure iteration
`double **F,**G`	F, G

Memory for these variables should be *dynamically allocated*, i.e. no fixed maximal memory size is reserved at compile time; rather only as much memory as needed at each program start. One possible way to do this is to use the function `matrix(...)` incorporated in file `helper.c`, which can be downloaded from the Course Web site (see also W.Press, B.Flannery, S.Teukolsky, and W.T.Vetterling (1990). *Numerical Recipes in C.* Cambridge: Cambridge University Press). Clean memory management requires also freeing the memory, when it is no longer needed. The procedure for freeing memory belonging to `matrix` is called `free_matrix` and can also be found in `helper.c`. Typical calls of these procedures are `P = matrix(0,imax+1,0,jmax+1);` and `free_matrix(P,0,imax+1,0,jmax+1);`

7 The Program

The following C functions corresponding to the algorithm's partial steps have to be implemented (here, for the sake of brevity, data types of the function parameters are omitted):

1. `void init_uvp(UI,VI,PI,imax,jmax,U,V,P)`
   The arrays `U,V,P` are initialized to the constant values `UI,VI` and `PI` on the whole domain.

2. `void calculate_dt(Re,tau,dt,dx,dy,imax,jmax,U,V)`
   The stepsize dt for the next time step is calculated according to (13). In case of negative `tau`, the stepsize to be read in `read_parameters` (see below) should be used.

3. `void boundaryvalues(imax,jmax,U,V)`
   The boundary values for the arrays `U` and `V` are set according to the formulas (14), (15).

4. `void calculate_fg(Re,GX,GY,alpha,dt,dx,dy,imax,jmax,U,V,F,G)`
   Computation of `F` and `G` according to (9) and (10). Here the formulas (17) must be applied at the boundary.

5. `void calculate_rs(dt,dx,dy,imax,jmax,F,G,RS)`
   Computation of the right-hand side of the pressure equation (11).

6. `void sor(omg,dx,dy,imax,jmax,P,RS,*res)`
   SOR iteration for the Poisson equation (with respect to pressure) according to (18). Besides, the routine must also set the boundary values for `P` according to (16), prior to each iteration step. The residual (19) is to be stored in `*res`.

7. `void calculate_uv(dt,dx,dy,imax,jmax,U,V,F,G,P)`
   The new velocities are computed according to (7) and (8).

For all the operations defined above you are given the declaration within the header files `init.h`, *uvp.h* and *sor.h*. A trivial Gauss–Seidel implementation, that does not take ω into account, is provided as black–box–solver.

The following procedures can be readily used (see `helper.c`, `visual.c`, and `init.c`) and downloaded from the Web site:

1. `double **matrix( nrl, nrh, ncl, nch )`
   Reserves memory for an array (matrix) of size `[nrl,nrh]x[ncl,nch]`.

2. `void free_matrix( m, nrl, nrh, ncl, nch )`
   Frees memory allocated for an array `m`

3. `void init_matrix( m, nrl, nrh, ncl, nch, a)`
   Initializes an array `m` of size `[nrl,nrh]x[ncl,nch]` with the value `a`.

4. `imatrix`, `free_imatrix`, `init_imatrix`
   Procedures for `integer` arrays with the same functions as `matrix`, `free_matrix`, and `init_matrix`.

5. `void write_matrix(char *Filename, m, nrl, nrh, ncl, nch, xlength,`
   `ylength, fFirst)`
   Writes the matrix `m` of size `[nrl,nrh]x[ncl,nch]` in file `Filename`. If `fFirst` $\neq$ `0`, then the geometric length in x- and y-directions will be written in ASCII format, followed by the array limits `nrl` .... Afterwards, the array entries are inserted in boundary format as `floats`.

6. `void write_vtkFile(szProblem, timeStepNumber, xlength, ylength, imax, jmax,`
   `dx, dy, U, V, P)`
   The function `write_vtkFile` writes all necessary visualisation data of one single time step to a vtk (paraview) file. In particular, the node coordinates, the pressure (cell-centered) and velocity (node-centered) data are written. The name of the resulting file is "szProblem.timeStepNumber.vtk". This function has to be called inside the main time loop.

7. `void write_vtkFileHeader(fp, imax, jmax, dx, dy)`
   Writes the header for the vtk visualisation file. Usually, you don't have to use this function manually (it is called automatically in `write_vtkFile`).

8. `void write_vtkPointCoordinates(fp, imax, jmax, dx, dy)`
   Writes the nodal coordinates for the vtk visualisation file. Usually, you don't have to use this function manually (it is called automatically in `write_vtkFile`).

9. `int read_parameters(char *szFile,Re,UI,VI,PI,GX,GY,t_end,xlength,`
   `ylength,dt,dx,dy,imax,jmax,alpha,omg,tau,itermax,eps)`
   The listed quantities will be read from the specified files, `dx` and `dy` will be determined from `imax` and `xlength` as well as from `jmax` and `ylength`. While writing the input file, one must stick to the following rules. Comments should start with the number sign (`#`) at the beginning of the line. Variable names stand at the line beginning and the respective variable value is separated by an arbitrary number of spaces and tabs. Variables can be declared in arbitrary order. If `tau` is negative, then `dt` is the fixed time step. Otherwise, `dt` being set in the input file has no meaning: the time step size will be calculated over and over again for each time step from the velocity array (see the function `calculate_dt`).

Finally, in the main program, the algorithm described in Sect. 5 should be implemented.

8 Guidelines for Modular Programming

In order to keep the developed program manageable and flexible and to avoid long compilation times during development, it would be useful to collect the procedures in separate modules (files), which are then linked with the main program after compilation. Thus, for example, the procedures `read_parameters` and `init_uvp` can be grouped in a file `init.c`. Once this module has been written and compiled, it then no longer needs to be recompiled each time a detail in another part of the program is modified.

For the programs developed here, the following modular structure can be offered (also with regard to possible extensions to be introduced later):

`helper.c:`	`matrix, free_matrix, write_matrix`
`init.c:`	`read_parameters, init_uvp`
`boundary_val.c:`	`boundaryvalues`
`uvp.c:`	`calculate_fg, calculate_rs, calculate_dt` and `calculate_uv`
`visual.c:`	`write_vtkFile`
`sor.c:`	holds the solver `sor`
`main.c:`	the main programm

If a function defined in a module `B.c` is needed in another module `A.c`, then the corresponding function declaration must become known to the module `A.c`. This is usually accomplished with the help of so called *Header Files*: all function declarations – headers – in a module `A.c`, which are needed in other modules, are written in a file `A.h`, the latter being then included in all calling modules using the directive `#include` "`A.h`" (preprocessor). This eliminates the necessity to explicitly list all required function declarations in each file.

In a UNIX environment, the compilation and linking processes can be automated with the help of a *makefile* that is stored in a file of the same name. Then, to compile the whole program one only needs to use the command `make`. In our case, the makefile might look as follows:

```
CC = gcc
CFLAGS = -Wall -pedantic
.c.o:  ; $(CC) -c $(CFLAGS) $*.c
OBJ = helper.o init.o boundary.o uvp.o main.o

all: $(OBJ)
        $(CC) $(CFLAGS) -o sim $(OBJ) -lm
clean:
        rm $(OBJ)
helper.o  : helper.h
init.o    : helper.h init.h
boundary_val.o:helper.h boundary_val.h
uvp.o     : helper.h uvp.h
main.o    : helper.h init.h boundary_val.h uvp.h
```

Explanations of the individual commands:

- `CC = gcc` and `CFLAGS = -Wall -pedantic` are *macro definitions*, selecting a particular compiler and possible compiler options. Here, the options (flags) `-Wall` and `-pedantic` cause the compiler to generate warnings whenever it encounters likely sources of error or incorrect code. When the option `-O` is used, the runtime-optimized program is executed. If the program should be explored by a debugger, the `-g` option must be added.
- `.SUFFIXES: .o .c` determines a general pattern of the file interdependence. Whenever a .c file is modified, the corresponding .o file must be rebuilt.

- `.c.o: ; $(CC) -c $(CFLAGS) $*.c` causes the *transformation* of .c files in the .o files (object files) by means of the command `$(CC) -c $(CFLAGS) $*.c` . Here `$(CC)` signifies, e.g., that the C compiler defined by the macro `CC` is inserted in this place.
- `OBJ = helper.o init.o boundary.o uvp.o main.o` define the objects
- `all: $(OBJ)` determines the dependencies of the `sim` program. Whenever a new .o file in `OBJ` is formed, the program must be linked together again.
- `$(CC) $(CFLAGS) -o sim $(OBJ) -lm` is the command to link together the object files listed in the `OBJ` macro. Here the option `-o sim` causes the created executable program to be named `sim` (the default name is `a.out`), and `-lm` links the mathematical library. IMPORTANT: This command must begin with a TAB!
- The remaining commands define special dependencies. For example, if `uvp.h` is modified, then both `uvp.o` and `main.o` must also be rebuilt.

In order to reduce the amount of work, the makefile can be downloaded from the Web site.

9 Visualization

The visualization of the simulated results is done in this Practical Course with the aid of the `ParaView` software package. The scope of possibilities of this software extends far beyond the needs of this Practical Course. In order to facilitate the beginner's work with this package, we shall explain here how one can proceed with the visualization of the presented worksheet.

Since the program to be created works on *structured, uniform grids*, i.e. meshes where the physical coordinates of a point are produced via a simple multiplication of the logical coordinates i and j by the mesh sizes dx and dy, it makes sense to save the calculated values in the same form as they have been stored in the corresponding arrays. The data for different time steps will be written into individual files with the form `problemName.timeStep.vtk`. These files will be read in by `ParaView` which is able to treat several of them at once to visualize time-dependent data. In this worksheet, you only need to care for writing data in this file to the extent of correctly calling the functions provided by `visual.c`.

In `ParaView`, the data treated as objects in the sense of object-oriented programming are processed in modules. The choice and functionality of available modules are quite rich, but we only need to use some basics. Here is a short tutorial to visualize flow data:

- type "paraview" in your command window to start `ParaView`.
- Choose "Open Data" from the "File" menu. Select the desired data file and push "Open". You should already see some data visualization in the right part of the GUI.
- In order to show velocity data (as arrows, e.g.), select "Glyph" from the "Filter" menu. Accept the given specification. Now you should see some default arrows for your velocity data. To change their color to a value scaled with the velocity vector length, select the "Display" tab

and select "Point GlyphVector" in the "Color by" drop down menu. To scale the size of the arrows, you have to switch back to the "Parameters" tab where you may simply specify a factor in the text field "Scale Factor" and hit "Accept" (highlighted in green now).

- To introduce streamlines, click on the loaded data (cavity100.1.vtk, e.g.) to make this one active and select "StreamTracer" from the "Filter" menu. Choose "Line" in the "Seed" drop down menu and hit "Accept". You should get some white streamlines in your visualization. In order to change the number of seeding points (and resulting lines), enter the desired value in the text field "Resolution" under "Line Widget" (default value is 20).

A list of modules shown in the upper left corner of the GUI gives you a survey of your used modules. You can download an appropriate trace file of modules, that holds exactly the steps described above, as `cavity100.1.pvs` from the Course Web site. This saved session state assumes that the output file cavity100.1.vtk exists in the same directory. You are of course encouraged to modify your visualisation and adapt it to your needs. Whenever you wish to save a certain status, simply choose "Save Session State" from the "File" menu and select an appropriate name for your session file to save. For further information or tutorials please see `http://www.psc.edu/general/software/packages/paraview/tutorial/`, e.g.

10 Problems

1. Implement the functions described in Section 7 as well as the main program, taking into account the help functions given in the files `helper.c`, `init.c` and `visual.c` and the partitioning into modules described in Section 8. Use the Gauss–Seidel black–box solver.

2. Implement a ω–SOR solver instead of the Gauss–Seidel solver and examine the solver's behaviour depending on ω.

3. As a first example, a typical problem from computational fluid dynamics, the so-called driven cavity, will be simulated. The domain is a container filled with a fluid with the container lid (a band or a ribbon) moving at a constant velocity. No-slip conditions are imposed on all four boundaries, with the exception of the upper boundary, along which the velocity u in x–direction is not set to zero, but is equal to 1, in order to simulate the moving lid. In the program, this is implemented by setting the boundary value along the upper boundary to

$$u_{i,jmax+1} = 2.0 - u_{i,jmax}, \quad i = 1, \ldots, imax$$

Adjust the input file `cavity100.dat`, which can be also downloaded from the Course Web site, in such a way as the simulation parameters would take the following values:

```
imax = 50     jmax = 50     xlength = 1.0  ylength = 1.0
dt = 0.05     t_end = 50.0  tau = 0.5      dt_value = 0.5
eps = 0.001   omg = 1.7     alpha = 0.5    itermax = 100
GX = 0.0      GY = 0.0      Re = 100
UI = 0.0      VI = 0.0      PI = 0.0
```

Perform the simulation with these input data. What do you observe?

4. Visualize the results with the methods described in Section 9.

5. Select the time steps manually. Find out, by repeated trials, for what dt the algorithm is stable.

6. Set `imax` = `jmax` = 16, 32, 64, 128, 256, ...and determine the respective end value of `U[imax/2][7*jmax/8]` of the time-stepping loop (t=100). Find the values of $imax, jmax$, for which the value of U tends to ∞ and describe how the error depends on $imax, jmax$.

7. What happens if the Reynolds number is increased (Re = 100, 500, 2000, 10000)?