Technische Universität München
Institut für Informatik
Benjamin Uekermann, Benjamin Rüth

Summer Term 2018
Worksheet 1

Praktikum Wissenschaftliches Rechnen
Computational Fluid Dynamics

Worksheet 1
Navier-Stokes Equations

Deadline: May 1st 2018, 12:00
don't forget to also sign up for the examination!

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. Air is an example for a compressible medium, that is 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 to be incompressible. The viscosity describes fluid-internal friction. Air is characterized by a very small viscosity, water has a somewhat larger one, and honey possesses a very large viscosity. If 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. These flows are called *laminar* and not *turbulent*. *Non-stationary* flows vary over 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 Equations

Non-stationary (incompressible) viscous fluid flow is described by the Navier-Stokes equations. For simplicity, we restrict our considerations 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.

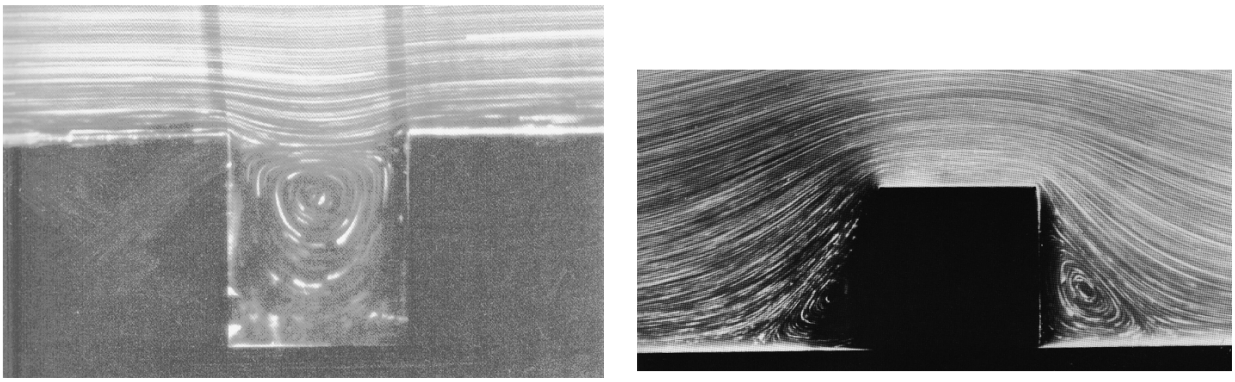

Figure 1: Left: Flow over a hollow space (driven cavity). Right: Flow over a step.

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

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

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

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 such as gravity.

The dimensionless quantity Re is called *Reynolds number*; it characterizes the fluid flow and defines whether the flow is laminar or turbulent. The Reynolds number depends on the (kinematic) viscosity ν, the characteristic length scale L_c of the scenario, and the characteristic velocity u_c of the fluid,

$$Re = \frac{u_c L_c}{\nu}. \tag{4}$$

In order to solve this system of partial differential equations, both initial and boundary conditions are required.

Initial conditions:

At the initial time $t = 0$, initial conditions $u = u_0(x, y)$ and $v = v_0(x, y)$ are prescribed throughout the entire computational domain. The velocity field needs to satisfy the continuity equation (3).

Boundary conditions:

Besides, supplementary conditions holding at all four boundaries of the region at all times are required, yielding 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 $w_\perp$ and the component parallel to the boundary (in tangential

direction) as $w_\parallel$. The derivatives in the normal direction will be denoted as $\partial w_\perp/\partial \mathbf{n}$ and $\partial w_\parallel/\partial \mathbf{n}$, respectively.

Then, for the vertical boundary, we have

$$w_\perp = u, \qquad w_\parallel = v, \qquad \frac{\partial w_\perp}{\partial \mathbf{n}} = \frac{\partial u}{\partial x}, \qquad \frac{\partial w_\parallel}{\partial \mathbf{n}} = \frac{\partial v}{\partial x},$$

and for the horizontal boundary

$$w_\perp = v, \qquad w_\parallel = u, \qquad \frac{\partial w_\perp}{\partial \mathbf{n}} = \frac{\partial v}{\partial y}, \qquad \frac{\partial w_\parallel}{\partial \mathbf{n}} = \frac{\partial u}{\partial y}.$$

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

1. No-slip condition: $w_\perp(x, y) = 0$, $w_\parallel(x, y) = 0$.
   (the fluid velocity vanishes at the boundary)
2. Free-slip condition: $w_\perp(x, y) = 0$, $\partial w_\parallel(x, y)/\partial \mathbf{n} = 0$.
   (there is no friction at the boundary)
3. Inflow condition: $w_\perp(x, y) = w_\perp^0$, $w_\parallel(x, y) = w_\parallel^0$.
   (the velocity components $w_\perp^0, w_\parallel^0$ are prescribed)
4. Outflow condition: $\partial w_\perp(x, y)/\partial \mathbf{n} = 0$, $\partial w_\parallel(x, y)/\partial \mathbf{n} = 0$.

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.

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

This is required in order to satisfy 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 Karman vortex street. Please mind the importance of boundary conditions in each of these cases.

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 finite difference method based on a *staggered Cartesian grid*, that is the unknown variables u, v, and p are located at different positions on our 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 the assignment of the index (i, j) to the values of u and v.

As a consequence, the boundary of our rectangular domain does not contain values of all unknown functions (vertical boundary lines contain only u-values, whereas level lines only contain v-values;

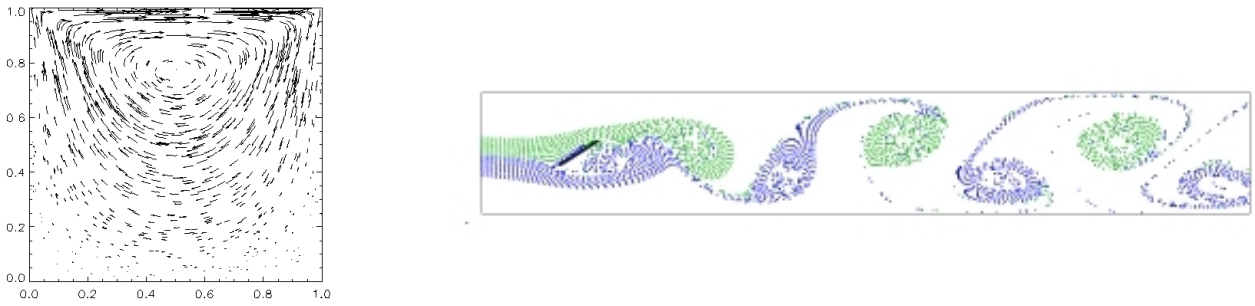

Figure 2: Left: Driven cavity: No-slip conditions at the lower, left, and right boundary, moving wall with $u = 1, v = 0$ at the upper boundary. Right: Karman vortex street: No-slip conditions at the upper and lower boundary, $u = 1, v = 0$ at the left boundary, outflow conditions at the right boundary, no-slip conditions around the obstacle.

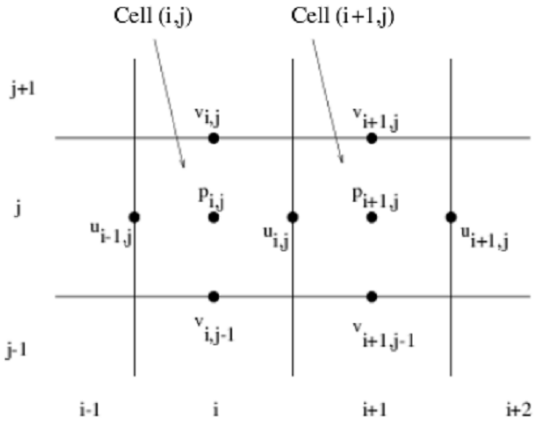

Figure 3: Staggered Grid for u, v, and p.

p-values are never located on boundary lines). For this reason, one more boundary strip of grid cells is introduced (see Figure 4), so that the boundary conditions may be satisfied by averaging the neighboring grid points on either side (see below).

Furthermore, the time interval $[0, t_{\text{end}}]$ is discretized. We consider the times $0 = t_1 < t_2 < \cdots < t_N = t_{\text{end}}$, i.e. values of u, v and p will be considered only at times t_n, $n = 0, 1, \ldots, N$. These times cannot be chosen arbitrarily but must be adjusted to the current state of the flow and the chosen spatial discretization according to stability constraints. We shall discuss the time-step control later in this worksheet.

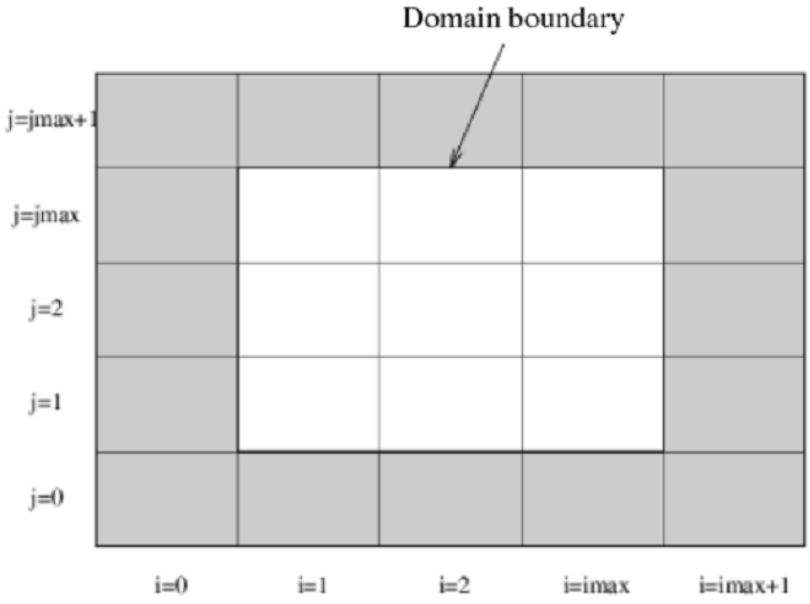

Figure 4: Domain with boundary cells.

3.2. Spatial Discretization

The Navier-Stokes equations are discretized in space as follows. The momentum equation (1) is evaluated at the vertical edge midpoints, the momentum equation (2) at the horizontal edge midpoints, and the continuity equation (3) at the cell midpoints. We replace the differential expressions from equation (1) and approximate them at the midpoint of the right edge of cell (i, j), $i = 1, \ldots, imax-1$, $j = 1 \ldots, jmax$, using

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

For the expressions in equation (2), we set at the midpoint of the upper edge of cell (i,j), $i = 1,\ldots,imax$, $j = 1,\ldots,jmax-1$

$$
\begin{aligned}
\left[\frac{\partial(uv)}{\partial x}\right]_{i,j} &:= \frac{1}{\delta x}\left(\frac{(u_{i,j}+u_{i,j+1})}{2}\frac{(v_{i,j}+v_{i+1,j})}{2}-\frac{(u_{i-1,j}+u_{i-1,j+1})}{2}\frac{(v_{i-1,j}+v_{i,j})}{2}\right)+\\
&\quad \alpha\frac{1}{\delta x}\left(\frac{|u_{i,j}+u_{i,j+1}|}{2}\frac{(v_{i,j}-v_{i+1,j})}{2}-\frac{|u_{i-1,j}+u_{i-1,j+1}|}{2}\frac{(v_{i-1,j}-v_{i,j})}{2}\right),\\
\left[\frac{\partial(v^2)}{\partial y}\right]_{i,j} &:= \frac{1}{\delta y}\left(\left(\frac{v_{i,j}+v_{i,j+1}}{2}\right)^2-\left(\frac{v_{i,j-1}+v_{i,j}}{2}\right)^2\right)+\\
&\quad \alpha\frac{1}{\delta y}\left(\frac{|v_{i,j}+v_{i,j+1}|}{2}\frac{(v_{i,j}-v_{i,j+1})}{2}-\frac{|v_{i,j-1}+v_{i,j}|}{2}\frac{(v_{i,j-1}-v_{i,j})}{2}\right), \qquad (6)\\
\left[\frac{\partial^2 v}{\partial x^2}\right]_{i,j} &:= \frac{v_{i+1,j}-2v_{i,j}+v_{i-1,j}}{(\delta x)^2},\\
\left[\frac{\partial^2 v}{\partial y^2}\right]_{i,j} &:= \frac{v_{i,j+1}-2v_{i,j}+v_{i,j-1}}{(\delta y)^2}, \qquad \left[\frac{\partial p}{\partial y}\right]_{i,j} := \frac{p_{i,j+1}-p_{i,j}}{\delta y}.
\end{aligned}
$$

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

$$
\left[\frac{\partial u}{\partial x}\right]_{i,j} := \frac{u_{i,j}-u_{i-1,j}}{\delta x}, \qquad \left[\frac{\partial v}{\partial y}\right]_{i,j} := \frac{v_{i,j}-v_{i,j-1}}{\delta y}. \qquad (7)
$$

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, we apply the *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 maximum of $|u\,\delta t/\delta x|$ and $|v\,\delta t/\delta y|$.

3.3. Time Discretization

The discrete momentum equations

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

$$u_{i,j}^{(n+1)} = F_{i,j}^{(n)} - \frac{\delta t}{\delta x}(p_{i+1,j}^{(n+1)} - p_{i,j}^{(n+1)}) \tag{8}$$
$$i = 1, \ldots, imax - 1, \quad j = 1, \ldots, jmax;$$
$$v_{i,j}^{(n+1)} = G_{i,j}^{(n)} - \frac{\delta t}{\delta y}(p_{i,j+1}^{(n+1)} - p_{i,j}^{(n+1)}) \tag{9}$$
$$i = 1, \ldots, imax, \quad j = 1, \ldots, jmax - 1.$$

with the terms $F_{i,j}^{(n)}$ and $G_{i,j}^{(n)}$ containing the discretized differential expressions of the momentum equations (1) and (2) evaluated at time step n:

$$F_{i,j} := u_{i,j} + \delta t \left(\frac{1}{Re} \left(\left[\frac{\partial^2 u}{\partial x^2}\right]_{i,j} + \left[\frac{\partial^2 u}{\partial y^2}\right]_{i,j} \right) - \left[\frac{\partial (u^2)}{\partial x}\right]_{i,j} - \left[\frac{\partial (uv)}{\partial y}\right]_{i,j} + g_x \right) \tag{10}$$
$$i = 1, \ldots, imax - 1, \quad j = 1, \ldots, jmax;$$
$$G_{i,j} := v_{i,j} + \delta t \left(\frac{1}{Re} \left(\left[\frac{\partial^2 v}{\partial x^2}\right]_{i,j} + \left[\frac{\partial^2 v}{\partial y^2}\right]_{i,j} \right) - \left[\frac{\partial (uv)}{\partial x}\right]_{i,j} - \left[\frac{\partial (v^2)}{\partial y}\right]_{i,j} + g_y \right) \tag{11}$$
$$i = 1, \ldots, imax, \quad j = 1, \ldots, jmax - 1.$$

The pressure equation

Equations (8) and (9) 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 $p^{(n+1)}$ is still unknown. Besides, we did not make use of 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

$$\frac{p_{i+1,j}^{(n+1)} - 2p_{i,j}^{(n+1)} + p_{i-1,j}^{(n+1)}}{(\delta x)^2} + \frac{p_{i,j+1}^{(n+1)} - 2p_{i,j}^{(n+1)} + p_{i,j-1}^{(n+1)}}{(\delta y)^2} =$$
$$\frac{1}{\delta t} \left(\frac{F_{i,j}^{(n)} - F_{i-1,j}^{(n)}}{\delta x} + \frac{G_{i,j}^{(n)} - G_{i,j-1}^{(n)}}{\delta y} \right), \tag{12}$$
$$i = 1, \ldots, imax, \quad j = 1, \ldots, jmax,$$

which corresponds to the well-known discrete form of the 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 uniqueness of the solution, we also need boundary values $p_{i,j}$ ($i \in \{0, imax+1\}, j \in \{0, jmax+1\}$), $F_{i,j}$ ($i \in \{0, imax\}$) and $G_{i,j}$ ($j \in \{0, jmax\}$), see later sections of this worksheet.

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 step sizes δx, δy, and δt:

$$\frac{2}{Re}\,\delta t < \frac{(\delta x)^2(\delta y)^2}{(\delta x)^2 + (\delta y)^2}, \qquad |u_{max}|\,\delta t < \delta x, \qquad |v_{max}|\,\delta t < \delta y. \tag{13}$$

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 step size control based on the stability conditions from above. This is implemented by choosing δt for the next time step in such a way that each of the three conditions (18) is satisfied:

$$\delta t := \tau \min\left(\frac{Re}{2}\left(\frac{1}{\delta x^2} + \frac{1}{\delta y^2}\right)^{-1}, \frac{\delta x}{|u|_{\max}}, \frac{\delta y}{|v|_{\max}}\right). \tag{14}$$

The coefficient $\tau \in]0, 1]$ is a safety factor. This step size control ensures, however, only the stability of the method. In order to specify the accuracy, the step size control may be based on particular error estimation procedures.

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 require the values of u and v that are lying on the domain's boundary or even outside of the domain. Here, we shall restrict ourselves, at first, to no-slip conditions, i.e. zero velocities on the boundary. Thus, we can set the following values for the velocities lying right on the boundary:

$$u_{0,j} = 0,\ u_{imax,j} = 0,\ j = 1, \ldots, jmax \qquad v_{i,0} = 0,\ v_{i,jmax} = 0,\ i = 1, \ldots, imax. \tag{15}$$

Furthermore, since no v-values lie on the vertical boundaries (cf. 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:

$$\begin{aligned} v_{0,j} &= -v_{1,j}, \quad v_{imax+1,j} = -v_{imax,j}, \quad j = 1, \ldots, jmax, \\ u_{i,0} &= -u_{i,1}, \quad u_{i,jmax+1} = -u_{i,jmax}, \quad i = 1, \ldots, imax. \end{aligned} \tag{16}$$

Boundary conditions for the pressure p

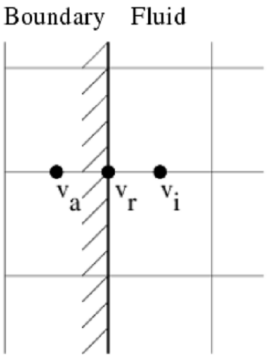

Figure 5: v-velocities around the left vertical boundary. Since no v-velocity lies on the vertical boundary, we use v_a and v_i to ensure the boundary condition: $v_r := \frac{v_a+v_i}{2} = 0 \quad \Rightarrow v_a = -v_i$.

The boundary values for the pressure are derived from the discretized momentum equation, resulting in discrete *Neumann conditions*:

$$\begin{array}{llll} p_{0,j} = p_{1,j}, & p_{imax+1,j} = p_{imax,j}, & j = 1, \ldots, jmax; \\ p_{i,0} = p_{i,1}, & p_{i,jmax+1} = p_{i,jmax}, & i = 1, \ldots, imax \end{array} \tag{17}$$

and

$$\begin{array}{llll} F_{0,j} = u_{0,j}, & F_{imax,j} = u_{imax,j}, & j = 1, \ldots, jmax, \\ G_{i,0} = v_{i,0}, & G_{i,jmax} = v_{i,jmax}, & i = 1, \ldots, imax. \end{array} \tag{18}$$

4. Solving the Linear System – SOR

If we look at the result of our discretization, we see that in each time step we have to solve the Poisson equation (12) 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, such as Gaussian elimination, lead to high computational costs for large problems, it is more appropriate to use iterative procedures. An example for an iterative method is the Gauss-Seidel method, in which, starting from some initial value, all the cells are successively traversed in each cycle, and the pressure at cell (i, j) 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, where the iteration step is given by the following loop over all cells:

$it = 1, \ldots, itmax$
  $i = 1, \ldots, imax$

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

$$p_{i,j}^{it+1} := (1-\omega)\, p_{i,j}^{it} + \frac{\omega}{2(\frac{1}{(\delta x)^2} + \frac{1}{(\delta y)^2})} \left(\frac{p_{i+1,j}^{it} + p_{i-1,j}^{it+1}}{(\delta x)^2} + \frac{p_{i,j+1}^{it} + p_{i,j-1}^{it+1}}{(\delta y)^2} - rs_{i,j} \right) \tag{19}$$

The upper indices it and $it+1$ represent the current and next iteration step. Note that the old pressure value p^{it} is immediately overwritten by the updated value p^{it+1}, i.e. there is no saved copy of the pressure field.

The quantity $rs_{i,j}$ is the right-hand side of the pressure equation (11) for the cell (i,j) 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 becomes identical to Gauss-Seidel. The iteration process should either stop once the maximum number of iterations, $itmax$, is reached or when the absolute residual

$$res := \left(\sum_{i=1}^{imax} \sum_{j=1}^{jmax} \left(\frac{p_{i+1,j} - 2p_{i,j} + p_{i-1,j}}{(\delta x)^2} + \frac{p_{i,j+1} - 2p_{i,j} + p_{i,j-1}}{(\delta y)^2} - rs_{i,j} \right)^2 / (imax \cdot jmax) \right)^{1/2} \tag{20}$$

drops below a predefined tolerance value ε.

As a starting value for the iteration process to calculate the pressure $p^{(n+1)}$, the pressure values of the previous time step n are used.

A method calculating one iteration step of the SOR solver and the corresponding residual is provided and can thus be directly used for the solution of the pressure equation. Note that the parameters (e.g. the threshold of the residual norm) still can be adjusted in order to steer the quality of the solution.

5. The Algorithm

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

```
Read the problem parameters
Set t := 0, n := 0
Assign initial values to u, v, p
While t < t_end
    Select δt according to (14)
    Set boundary values for u and v according to (15),(16)
    Compute F^(n) and G^(n) according to (10),(11),(18)
    Compute the right-hand side rs of the pressure equation (12)
    Set it := 0
    While it < itmax and res > ε
       Perform a SOR iteration according to (19) using the
          provided function and retrieve the residual res
       it := it + 1
    Compute u^(n+1) and v^(n+1) according to (8),(9)
    Output of u, v, p values for visualization, if necessary
    t := t + δt
    n := n + 1
Output of u, v, p for visualization
```

6. Problem Parameters and Data Structures

In this Practical Course, the algorithm described above is to 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`, should 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 $\Omega = [0, \texttt{xlength}] \times [0, \texttt{ylength}]$.
`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
`int n`	current time iteration step

- 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 (5))

- 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 should 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, but rather only as much memory as needed at each program start. One possible way to do this is to use the function `matrix(...)` from the file `helper.c`. 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 single algorithmic 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 step size dt for the next time step is calculated according to (14). In case of negative `tau`, the step size 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 (15), (16).
4. `void calculate_fg(Re,GX,GY,alpha,dt,dx,dy,imax,jmax,U,V,F,G)`
   Computation of `F` and `G` according to (10) and (11). Here the formulas (18) 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 (12).

6. `void calculate_uv(dt,dx,dy,imax,jmax,U,V,F,G,P)`
   The new velocities are computed according to (8) and (9).

For all the operations defined above, you find the declaration in the header files `init.h`, *uvp.h* and *boundary_val.h*.

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

1. `void sor(omg,dx,dy,imax,jmax,P,RS,*res)`
   SOR iteration for the Poisson equation (with respect to pressure) according to (19). Besides, the routine also sets the boundary values for `P` according to (17), prior to each iteration step. The residual (20) is stored in `*res`.
2. `double **matrix( nrl, nrh, ncl, nch )`
   Reserves memory for an array (matrix) of size `[nrl,nrh]x[ncl,nch]`.
3. `void free_matrix( m, nrl, nrh, ncl, nch )`
   Frees memory allocated for an array `m`
4. `void init_matrix( m, nrl, nrh, ncl, nch, a)`
   Initializes an array `m` of size `[nrl,nrh]x[ncl,nch]` with the value `a`.
5. `imatrix`, `free_imatrix`, `init_imatrix`
   Procedures for `integer` arrays with the same functions as `matrix`, `free_matrix`, and `init_matrix`.
6. `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 as `floats`.
7. `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.
8. `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`).
9. `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`).
10. `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,dt_value)`
    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 is 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 -Werror
.c.o:  ; $(CC) -c $(CFLAGS) $<
OBJ = helper.o init.o boundary_val.o uvp.o main.o visual.o

all: $(OBJ)
        $(CC) $(CFLAGS) -o sim $(OBJ) -lm

%.o: %.c
$(CC) -c $(CFLAGS) $*.c -o $*.o

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
visual.o  : helper.h
main.o    : helper.h init.h boundary_val.h uvp.h visual.h
```

Explanations of the individual commands:

- `CC = gcc` and `CFLAGS = -Wall -pedantic -Werror` 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 whereas `-Werror` turns all warnings into errors reported at compile time. 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) $<` causes the *transformation* of .c files into .o files (object files). Here `$(CC)` implies that the C compiler defined by the macro `CC` is inserted in this place.
- `OBJ = helper.o init.o boundary_val.o uvp.o main.o visual.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 be rebuilt.

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

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 write data into vtk-files by 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 click "Open". You should already see some data visualization in the right part of the GUI.
- In order to show velocity data (such as arrows), 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 click "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 click "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).

For further information or tutorials please see http://www.paraview.org/Wiki/The_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.
2. 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 also be downloaded from the Course website such that the simulation parameters 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?

3. Visualize the results with the methods described in Section 9.
4. Examine the SOR solver's behaviour depending on ω in the range $\omega = [0..2]$.
5. Select the time steps manually. Find out, by repeated trials, for which 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 $imax, jmax$. Keep the size of the time step δt constant for this task.
7. What happens if the Reynolds number is increased (Re = 100, 500, 2000, 10000)? Please use the adaptive time stepping here (and throughout the rest of the course).

Please hand in your solution that is your code including makefile *before* the given deadline (see first page)! The answers/ observations to the questions above do not have to be handed in; nevertheless, they can be part of the exam!

A. Administration

A.1. Moodle

All electronic resources for this course will be provided in the eLearning platform moodle (www.moodle.tum.de). To access the course's webpage you have to be registered via tumonline. So please register.

Finding the course in moodle

Log into moodle at www.moodle.tum.de with your tumonline username and password. Your personal homepage shoul appear and with it all the courses you registered for, i.e. all courses that have a moodle webpage. If you do not see your homepage, please click on *My home*. There you should see

Praktikum - Scientific Computing: CFD (IN2106, IN2186, IN2182, IN8904, IN4085), which leads to this course's moodle webpage.

The course moodle page

This page basically contains all resources for the course, which are all worksheets, assignments, feedbacks, announcements and (in the end) grades.

There are a few things to do for you concerning the first assignment:

Choose your Lab Group In the top center frame of the course page, you can see a little icon depicting a group with the link *Find your lab group!* behind it. Clicking that link will lead you to another moodle page, where you can assign yourself to a lab group. You can change your decision afterwards but note that there is a deadline where no changes can be made anymore. Also note, that we will assign groups, if the requirements concerning the group size are not met.

Get the worksheet In the center frame with the heading *Navier-Stokes Equations* you will find a link *Worksheet 1 - Navier-Stokes-Equations* with a pdf symbol in front. This link leads to this worksheet.

Assignment 1 - Navier-Stokes Equations Here you will have to upload your code for the first assignment, which is stated on this worksheet. Only one member of each group has to upload the code. You can only upload a single file, which should be a zip-compressed folder containing your code. And *ONLY* the code, if not stated otherwise. Note, that the upload is only possible before the deadline, which means you cannot upload anything after the passing of the deadline. The code will be reviewed by us and it will be also part of the final grade.

Get a time slot for the review This link will appear rather soon and you will be able to choose a time slot for the review session, where we will ask you (team-wise) some questions concerning the theory of the assignment. We will also ask you some questions concerning the code. This review is mandatory for every team member since your performance in all review sessions will yield the final grade.

Check the timeline Please check the timeline of the course regularly under the respective link. Note, that there might be changes to that schedule, due to some unforeseen events. All of the dates should also appear in your moodle calendar on the right-hand side.

A.2. Grading

You will be graded consecutively during the course. We will have four worksheets and a project which will be graded individually. Each grade in the worksheet is depending on the submitted code and your performance in the review session. Note, that you will be graded individually and not as group. If you fail one of the five parts of the lab course you fail the complete lab course.

The project will have a higher weight on the final grade, since you will have more time for it. For the project you will also give a final presentation of your results, which will also influence the grade.