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

ST 2009

Praktikum Wissenschaftliches Rechnen Computational Fluid Dynamics

Worksheet 2 (deadline June 2, 12:00)

In the second part of this Lab Course, the program implemented in the first part will be extended, such that it can work on arbitrary domain geometries and apply various boundary conditions.

What we have in mind is that the program reads all physical and numerical parameters, together with the boundary conditions, from a single input file `problem.dat`. The program should then be started by the command „`sim problem`“. Changes of parameters or boundary conditions should no longer require modifications in the source code, nor recompilation of the program.

1 Other Boundary Conditions

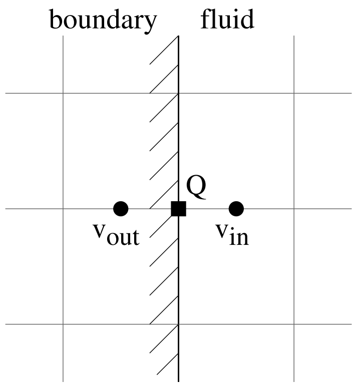
In this section, we shall treat again the boundary conditions already discussed in Worksheet 1. We shall deal with the question how reasonable boundary conditions for the velocity can be implemented. In particular, we will introduce three new types of boundary conditions: *free-slip*, *inflow*, and *outflow* conditions.

Free-slip Conditions:

Free-slip conditions model the case when the fluid can flow freely parallel to the domain boundary, but cannot cross the boundary. As a consequence, the velocity component normal to the wall should be zero, as well as the normal derivative of the velocity component parallel to the wall. In our rectangular domain that has been discretized with the help of a staggered grid, the discrete velocity components normal to the wall lie directly at the boundary. Thus, we may set (just as for the no-slip condition):

$$\begin{aligned} u_{0,j} &= 0, \quad u_{imax,j} = 0, \qquad j = 1,\ldots,jmax; \\ v_{i,0} &= 0, \quad v_{i,jmax} = 0, \qquad i = 1,\ldots,imax \end{aligned} \tag{1}$$

(assuming free-slip conditions at all four boundaries).



The normal derivative $\partial v/\partial n$ of the tangential velocity at a boundary point Q may be discretized by the expression $(v_i - v_a)/\delta x$, so that the requirement $\partial v/\partial n = 0$ leads to the condition

$$v_a = v_i.$$

We thus obtain the further boundary conditions

$$\begin{array}{llr} v_{0,j} = v_{1,j}, & v_{imax+1,j} = v_{imax,j}, & j = 1,\dots,jmax; \\ u_{i,0} = u_{i,1}, & u_{i,jmax+1} = u_{i,jmax}, & i = 1,\dots,imax \end{array} \tag{2}$$

(again, for all four boundaries).

Outflow Conditions:

For outflow boundary conditions, the normal derivatives of both velocity components are set to zero at the boundary, which means that the total velocity does not change in the direction normal to the boundary. This can be realized in the discrete grid by setting the velocity values at the boundary equal to their neighboring velocities inside the region, i.e.,

$$\begin{array}{lll} u_{0,j} = u_{1,j}, & u_{imax,j} = u_{imax-1,j}, & \multirow{2}{*}{} j = 1,\dots,jmax; \\ v_{0,j} = v_{1,j}, & v_{imax+1,j} = v_{imax,j}, & \\ u_{i,0} = u_{i,1}, & u_{i,jmax+1} = u_{i,jmax}, & i = 1,\dots,imax. \\ v_{i,0} = v_{i,1}, & v_{i,jmax} = v_{i,jmax-1}, & \end{array} \tag{3}$$

Inflow Conditions:

For inflow boundary conditions, the velocities on an inflow boundary are explicitly given. However, since these velocities take diverse values in different examples and are not always constant at the whole boundary, we cannot read these values from the input file in a simple way. Instead, we will set these boundary conditions, according to the considered problem, in a function `spec_boundary_val` that will be called each time directly after the function `boundary_val`.

Remark:

The boundary conditions for p, F and G have the same form as for no-slip conditions in Worksheet 1 (formulas (16) and (17)), regardless of whether we adopt free-slip, outflow, or inflow conditions.

Implementation:

The program now needs to be extended in a way that allows us to specify the boundary condition for each of the four sides of our domain. Therefore, we shall introduce integer parameters `wl`, `wr`, `wt` and `wb` for the left, right, upper, and lower boundary, respectively. The initial values of these parameters should be read from the input file by the function `read_parameters`.
The parameters `wl, wr, wt` and `wb` can take the values

- 1 for no-slip conditions,
- 2 for free-slip conditions, or
- 3 for outflow conditions.

For inflow conditions, using the options `wl,wr,wt` and `wb` is not sufficient, as the velocities have to be set to specific values determined by the respective problem (for example, corresponding to a parabolic or constant inflow profile). Instead, the distribution of velocities at such boundaries should be produced by a separate procedure `spec_boundary_val`. To decide which boundaries are inflow boundaries, the procedure `spec_boundary_val` uses the newly introduced variable `char* problem` that specifies the considered problem scenario.
In addition, this new procedure `spec_boundary_val` allows us to specify any kind of special boundary conditions. These can even act only on a specific part of a boundary and override the settings given by the parameters `wl, wr, wt` and `wb`.

Assignments:

1. Supplement the function `read_parameters` such that it reads initial values for the variable `problem` and for the boundary value options `wl,wr,wt` and `wb`.
2. Extend the function `boundary_values` in such a way that, for each initial distribution of boundary value parameters `wl, wr, wt` and `wb`, with the help of `if` or `switch` command, the correct boundary values for u and v are set according to the formulas above.
3. Write (in `boundary.c`) the procedure `void spec_boundary_val (char *problem, int imax, int jmax, double **U, double **V)`, where, depending on the considered problem (given by the variable `problem`), additional boundary conditions are implemented. This includes any kind of inflow conditions, and also special settings, as they are for example present in the driven cavity example (setting $u = 1$ at the upper boundary).

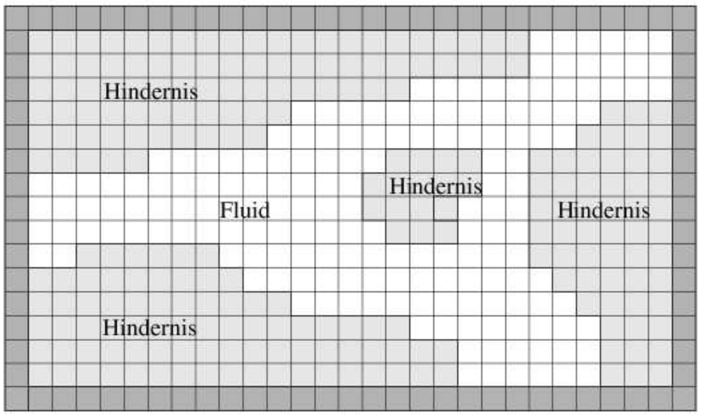


Fig. 1: Embedding of an arbitrary domain into a rectangular domain.

2 Treatment Of General Geometries

Up to now, we have described the simulation of flows in rectangular domains. A simple extension will enable us to approximately treat flows in arbitrary two-dimensional geometries. For this, we embed the flow region Ω in a rectangle $\mathcal{R}$ of the smallest possible size, which we shall cover with a grid, as described in Worksheet 1. The cells of $\mathcal{R}$ are then classified into *fluid cells* (that lie completely or mostly in Ω) and *obstacle cells* (which lie completely or mostly in the obstacle region, $\mathcal{H} := \mathcal{R} \setminus \Omega$). The discrete Navier-Stokes equations are then to be solved only in the fluid cells. Obstacle cells which share an edge (*boundary edge*) with at least one fluid cell are denoted as *boundary cells*. They play a similar role like the cells of the artificial cell layers at the outer boundaries. By the way, the cells of the outer boundary layers are also denoted as obstacle cells. A domain Ω with an arbitrary curved boundary is thus approximated by a domain $\tilde{\Omega}$ whose boundary is specified by the set of boundary edges lying on grid lines (see Figure 1).

Boundary Conditions:

In fluid cells adjacent to obstacle cells, we need the following boundary values in order to compute F and G according to (9) and (10) from Worksheet 1:

- values of the normal velocity components at the boundary edges and
- values of the tangential velocity components on edges between two boundary cells.

We will only implement no-slip conditions at the internal boundary edges. In the example shown in Figure 2, the boundary edges are marked with a square, whereas the edges between two boundary cells are marked with a circle.

Furthermore, to compute the pressure at the centers of these fluid cells, the F and G values at the boundary edges are needed, as well as the pressure values at the centers of the adjacent boundary cells (see the cross in Figure 2).

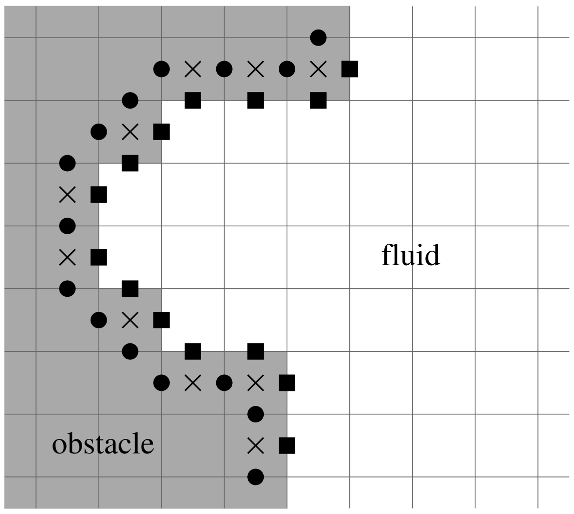


Fig.2: Required boundary values

To classify the fluid cells, we use an integer array `int **Flag`, which can be initialized as follows:

- `C_F` for a fluid cell and
- `C_B` for an obstacle cell.

The macros `C_B` and `C_F` denote integer constants that can be chosen arbitrarily.

To implement the initialization of boundary values, the set of boundary cells must be further classified: in the cell flag array we will store which of the four neighbouring cells are fluid cells. Thus, the macro value `B_O` designates e.g. a boundary cell whose right (eastern, "Ost") neighbor belongs to $\tilde{\Omega}$. Similar, `B_SW` may denote a boundary cell whose lower (southern) and left (western) neighbors belong to $\tilde{\Omega}$. We assume that only boundary cells with one neighboring fluid cell [1] (*edge cells*) or two neighboring fluid cells sharing a corner (*corner cells*) may occur. Boundary cells with two opposite or even three or four neighboring fluid cells are excluded (forbidden boundary cells), because these no longer allow uniquely defined boundary values. The accuracy of representing the geometry is thus limited by the double cell length.

In accordance to the formulas (14), (15), (16) and (17) of Worksheet 1, we set for a northern edge cell (i, j) having the flag `B_N`

$$\begin{aligned} &v_{i,j} = 0, && u_{i-1,j} = -u_{i-1,j+1}, && u_{i,j} = -u_{i,j+1}, \\ &G_{i,j} = v_{i,j}, && p_{i,j} = p_{i,j+1} \end{aligned} \tag{4}$$

or, for a western edge cell (i, j) with flag `B_W`

$$\begin{aligned} &u_{i-1,j} = 0, && v_{i,j-1} = -v_{i-1,j-1}, && v_{i,j} = -v_{i-1,j}, \\ &F_{i-1,j} = u_{i-1,j} && p_{i,j} = p_{i-1,j}. \end{aligned} \tag{5}$$

[1] We consider cells to be *neighboring* if they share a common edge, not only a corner.

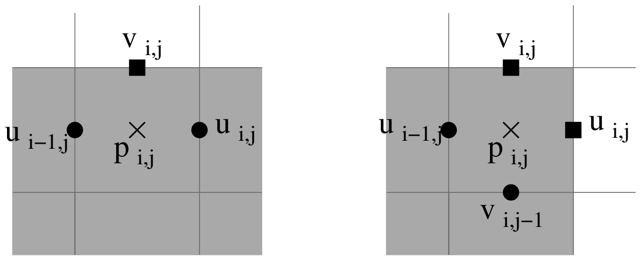


Fig. 3: Setting boundary values at obstacle cells (left: edge cell `B_N`, right: corner cell `B_NO`

The boundary values for `B_S` and `B_O` cells are set analogously.

For corner cells, we apply the condition to the normal velocity component at the two edges, and the condition for the tangent velocity component at the remaining two edges. Thus, e.g. for a cell (i, j) with flag `B_NO`

$$\begin{aligned} &u_{i,j} = 0, \qquad && u_{i-1,j} = -u_{i-1,j+1}, \qquad && F_{i,j} = u_{i,j}, \\ &v_{i,j} = 0, && v_{i,j-1} = -v_{i+1,j-1}, && G_{i,j} = v_{i,j}, \\ & && p_{i,j} = (p_{i,j+1} + p_{i+1,j})/2. && \end{aligned} \tag{6}$$

The solution of the pressure equation (Worksheet 1, formula (11)) is constrained to the fluid cells, whereas the computation of the F and G values according to (9) and (10), Worksheet 1, as well as the correction to the velocity value according to (7), (8), Worksheet 1, takes place only on edges between two fluid cells.

As a summary, we shall list once more all cell and edge definitions:

• fluid cell:	cell lying completely or mostly in Ω
• obstacle cell:	cell lying completely or mostly in $\mathcal{R} \setminus \Omega$
• boundary cell:	obstacle cell bordering on one or more fluid cells
• edge cell:	boundary cell bordering on exactly one fluid cell
• corner cell:	boundary cell bordering on two fluid cells sharing a corner
• forbidden boundary cell:	boundary cell that is neither corner, nor edge cell
• boundary edge:	edge between a fluid cell and a boundary cell

Assignments:

The integer array `int **Flag` of dimension `[0,imax+1]`x`[0,jmax+1]` is now needed as an additional data structure. Its entries are set depending on the parameter `problem`. It is convenient to use a binary representation for the possible states, assigning one bit to each cell and its four neighbors, e.g.

center	east	west	south	north

Each bit can be set to either 1, when the corresponding cell is a fluid cell, or 0, when it is an obstacle cell. The cells in the interior of the obstacle would thus carry the flag 00000, boundary cells (including forbidden ones), a flag between 00001 and 01111, fluid cells bordering on obstacle cells, a flag between 10000 and 11110, and cells in the interior of the fluid region receive a flag value of 11111. The flag `B_SO`, for example, corresponds to a bit coding of 01010 i.e. the decimal value of 10. Using the bit operators available in the C programming language, these distinctive cases can be formulated in a quite elegant fashion.

The function `int **imatrix(int nrl, int nrh, int ncl, int nch)` from `helper.c` will generate an array of integers, which is suitable to hold the flag field describing the fluid region. Based on this data structure, you should implement the following new function (in the file `init.c`):

- `void init_flag(problem,imax,jmax,Flag)`
  The array `Flag` is initialized with the flags `C_F` for fluid cells and `C_B` for obstacle cells as specified by the parameter `problem`. This must be followed by a loop over all cells where the boundary cells are marked with the appropriate flags `B_xy` depending on the direction, in which neighboring fluid cells lie.

The function `void init_flag` has to be called in the initialization part of the main program.

In addition, the following functions must be adjusted:

- In `boundary_values` and `calculate_fg`, boundary values must be assigned to the boundary cells defined by the flag array, as described above.
- In `SOR`, apart from the boundary values for the pressure at the boundary stripe, also the boundary values in the interior boundary cells should be set before each iteration step.
- In `SOR`, the iteration and the residual computation needs to be limited to the fluid cells, whereas F and G values in `calculate_fg` and velocities in `calculate_uv` are only calculated on edges separating two fluid cells. Besides, the normalization of the residual in `SOR` should be produced by dividing by the current number of fluid cells, and not by $imax \cdot jmax$.

3 Boundary Conditions For Pressure

In numerous cases, the driving force for the flow is the one of the pressure. An example is provided by air bubbles passing through a straw. Here, one produces the pressure difference between the air in the mouth and that in the environment. This results in an air flow directed against this gradient. Quite a lot of technologically important flows are produced by pressure differences.

Besides, for the numerical simulation, taking the pressure difference as the boundary condition is a great simplification. Actually, one can as well simulate each pressure-driven flow by imposing the velocity boundary conditions, and obtain the same results as by stipulating the pressure difference. However, the velocity boundary values are in most cases unknown. Often also the pressure difference must be calculated, the one that should be applied to produce a predetermined volume flow in a device.

In case we want to apply the pressure difference Δp to the considered domain, we can implement the former in our simulation by imposing Dirichlet boundary conditions for the pressure at the corresponding boundaries.

Since no points with a definite pressure value lie at the physical boundary of the domain, this value must be defined by interpolation (like for the velocities parallel to the wall). If we denote the pressure at the right boundary by p_w, then one can formulate the following relationship between the pressure p_{imax+1} at the boundary outer edge, the last value p_{imax} in the pressure array and p_w:

$$p_w = \frac{p_{imax} + p_{imax+1}}{2} \tag{7}$$

This equation, being solved with respect to p_{imax+1}, must be inserted into the system of equations for the pressure values. It replaces the old condition for pressure at the boundary, where the pressure at the outer edge was set equal to that in neighboring cells lying inside the domain.

If we specify a fixed value for the pressure at a boundary (Dirichlet condition), the corresponding velocity should obey to a Neumann boudary condition (equivalent to our *outflow* condition). To decide whether certain velocity values have to be computed at a boundary, we have already introduced the flag field `**Flag` . Hence, we also use the flag field to implement the pressure boundary conditions. Within the framework of this Lab Course, it will be sufficient to apply the pressure boundary values just for two sides of the considered domain, left and right. Therefore, we set one bit in the flag field for the case when the pressure value is set at the (boundary) cell lying at the right boundary and another bit corresponding to the left boundary.

The decision in which cells the appropriate bit for the pressure boundary conditions will be set is contained in the function `init_flag`, which also holds the other bits. The pressure values for both boundaries as well as the pressure difference (in case the pressure always remains constant at one boundary) should be read by the parameter file.

Assignments:

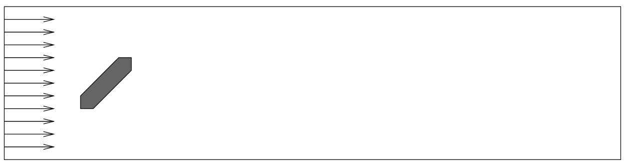
1. Make sure the initialization of appropriate bits is produced in `init_flag` at boundary cells.
2. Read the pressure difference in `read_parameters`.
3. Modify the `SOR` function in such a way that instead of the Neumann boundary conditions for the pressure the correct Dirichlet boundary conditions will be set.

Hint:
In order to read in general (that is more complex) geometries, a new version of the `helper`-files is provided. They contain a new function `read_pgm(filename)` that reads in an ASCII-encoded pgm-file. Thus, you can paint your own geometries. For details on the function, see the documentation in the code; for details on the pgm-format, see for example http://de.wikipedia.org/wiki/Portable_Graymap.

4 Example Problems

With the help of these modifications of the flow program, we are now able to compute a whole bunch of new examples. We only need to set up a new input file for each example. Besides, the problem-specific obstacle cells should be implemented in the function `init_flag` as well as the inflow conditions and further special boundary conditions in the function `spec_boundary_val`.

a) The von Karman vortex street:



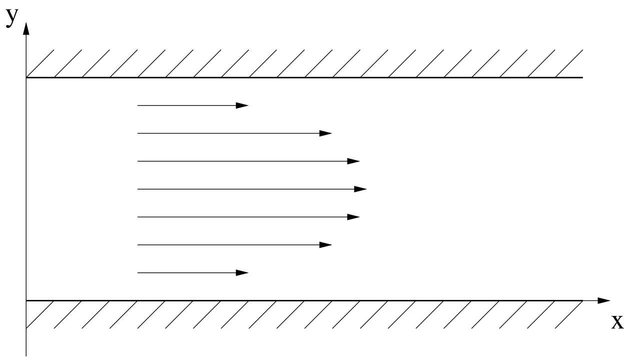
The flow in a channel hits a tilted plate. At the left boundary, the fluid inflow has a constant velocity profile ($u = 1.0$, $v = 0.0$), while at the upper and lower boundaries no-slip conditions are imposed.

The plate occupies one fifth of the channel width and is three cells thick. The distance from the left boundary is assumed to be the same as the distance to the lower and upper boundaries. Make sure that there are no forbidden obstacle cells.

Use the following values for the parameters:

imax = 100	jmax = 20	xlength = 10	ylength = 2
dt = 0.05	t_end = 20	tau = 0.5	dt_value = 2.0
eps = 0.001	omg = 1.7	alpha = 0.9	itermax = 500
GX = 0.0	GY = 0.0	Re = 10000	
UI = 1.0	VI = 0.0	PI = 0.0	
wl = 1.0	wr = 3.0	wt =1.0	wb =1.0

b) Plane shear flow:



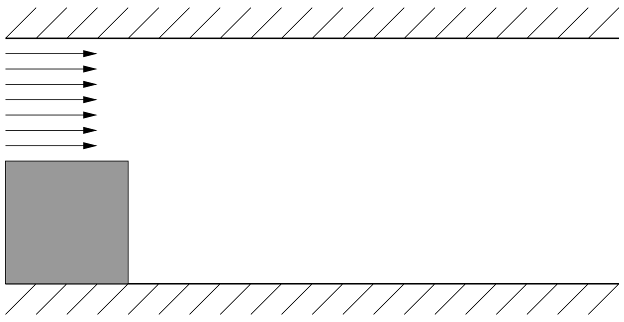
The plane shear flow is a favorite example problem to test the numerical methods and their functionality, since it is possible to give an analytic solution to this problem:

$$u(y) = -\frac{1}{2} Re \cdot \frac{\Delta p}{\Delta x} \cdot y \cdot (y - h) \qquad (8)$$

You can use, for example, the following parameters:

imax = 100	jmax = 20	xlength = 10	ylength = 2
dt = 0.05	t_end = 30	tau = 0.5	dt_value = 5.0
eps = 0.001	omg = 1.7	alpha = 0.9	itermax = 500
GX = 0.0	GY = 0.0	Re = 10	Δp = 4.0
UI = 1.0	VI = 0.0	PI = 0.0	
wl = 3	wr = 3	wt = 1	wb = 1

c) Flow over a step:



The fluid flows through a channel widening on one side. No-slip conditions ($u = v = 0$) are imposed at the upper and lower walls.

The obstacle domain is represented by a square filling up half of the channel height. Use the following values for the problem parameters:

imax = 100	jmax = 20	xlength = 10	ylength = 2
dt = 0.05	t_end = 500	tau = 0.5	dt_value = 10.0
eps = 0.001	omg = 1.7	alpha = 0.9	itermax = 500
GX = 0.0	GY = 0.0	Re = 100	
UI = 1.0	VI = 0.0	PI = 0.0	
wl = 3	wr = 3	wt = 1	wb = 1

While initializing the velocity values in `init_uvp`, in the lower half-part of the channel the value u might be set to 0 instead of 1.

5 Flow Visualisation by Particle Tracing and Streaklines

When studying flows such as the flow passing an obstacle as described above, one is often interested in analyzing the flow patterns by means of particle movement. We consider two methods for this purpose:

(A)

- **Pathlines (particle tracing):** At the starting point $t = 0$ particles are introduced at defined positions in the fluid. The successive positions of these particles in time represent the pathlines.
- **Streaklines:** Particles are "injected" into the flow at a fixed location in short regular time intervals. This creates the impression of lines if the time intervals are not too long and the flow is not too turbulent[1].

The basis for the computation of such particle movements is the position of a particle at time $t_{n+1} = t_n + \delta t$ (the position at time step t_n given). This is done in two steps:

(B)

- Find the velocities $u^{(n)}, v^{(n)}$ corresponding to positions $(x^{(n)}, y^{(n)})$ in the flow field.
- Determine the new position of a particle at time t_{n+1} using one step of the explicit Euler method:

$$x^{(n+1)} = x^{(n)} + \delta t \cdot u^{(n)}, \quad y^{(n+1)} = y^{(n)} + \delta t \cdot v^{(n)}\,.$$

The second of these steps is rather simple; the first, however, requires some explanations:

Interpolation of velocities in a "staggered grid":

Since the velocities u and v are available only at certain points of the staggered grid, one must somehow approximate the velocities between these grid points. Thus,
we have given: position (x, y), velocities u_{ij}, v_{ij} on the staggered grid:

$$u_{i,j} = u(x_i, y_j) \;\text{ with }\; x_i = i \cdot \delta x,\;\; y_j = (j - \frac{1}{2}) \cdot \delta y,\;\; i = 0, ..., imax,\; j = 0, ..., jmax + 1,$$

$$v_{i,j} = v(x_i, y_j) \;\text{ with }\; x_i = (i - \frac{1}{2}) \cdot \delta x,\;\; y_j = j \cdot \delta y,\;\; i = 0, ..., imax + 1,\; j = 0, ..., jmax,$$

we look for: the velocities $u(x, y)$, $v(x, y)$.

[1]The term "streakline" stems from the idea of *dye streaks* in paintings to visualise motion. One possibility to realize this in a physical experiment consists in introducing small air bubbles into the fluid along a thin wire (that does not disturb the flow). Another example are streaklines of soot particles in cigarette smoke.

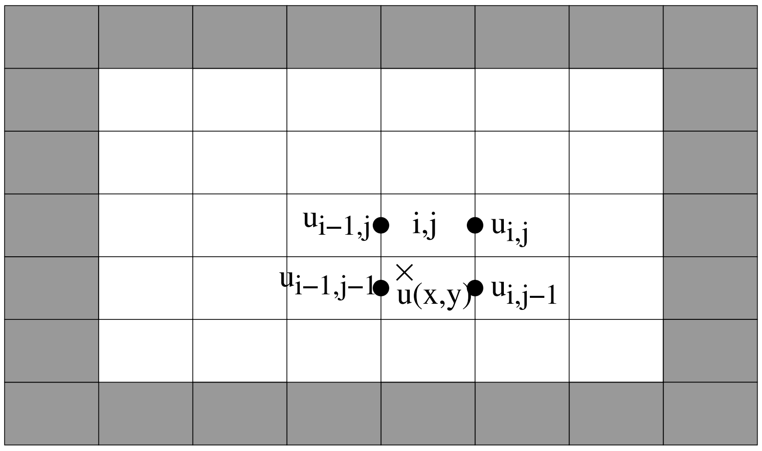


Fig. 4: u-cell (i,j) on a staggered grid

1. **Calculation of $u(x,y)$:**
One must determine the u–cell enclosing the position (x,y) (see Figures 4, 5). This is done with the help of the integer operations

$$i := (int)\left(\frac{x}{\delta x}\right) + 1, \quad j := (int)\left(\frac{y + \frac{\delta y}{2}}{\delta y}\right) + 1,$$

which define the indices of u-cells, and the operations

$$x_1 := (i-1)\cdot\delta x, \qquad y_1 := ((j-1) - \tfrac{1}{2})\cdot\delta y,$$
$$x_2 := i\cdot\delta x, \qquad y_2 := (j - \tfrac{1}{2})\cdot\delta y,$$

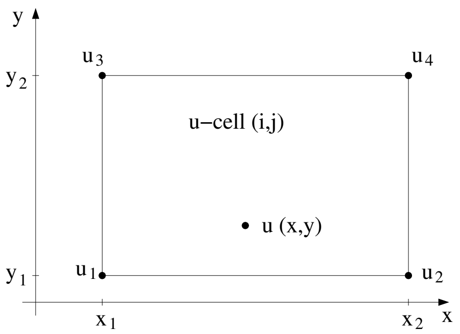


Fig. 5: u-cell (i,j)

defining the positions of the cell corners of these u-cells, where the four staggered-grid velocities

$$u_1 := u_{i-1,j-1}, \quad u_2 := u_{i,j-1}, \quad u_3 := u_{i-1,j}, \quad u_4 := u_{i,j}$$

are lying.

Now, we can approximate the required velocity $u(x,y)$ at the position (x,y) using bilinear interpolation:

$$u(x,y) := \tfrac{1}{\delta x\,\delta y}\Big[\,(x_2 - x)(y_2 - y)u_1 + (x - x_1)(y_2 - y)u_2 + (x_2 - x)(y - y_1)u_3 + (x - x_1)(y - y_1)u_4\,\Big]$$

2. **Calculation of $v(x, y)$:**
   The procedure here is analogous to the calculation of $u(x, y)$:
   The indices of the v-cell enclosing the position (x, y) are

$$i := (int)\left(\frac{x + \frac{\delta x}{2}}{\delta x}\right) + 1, \quad j := (int)\left(\frac{y}{\delta y}\right) + 1,$$

   the positions of the corners containing the v–velocities in the staggered grid are

$$\begin{aligned} x_1 &:= ((i-1) - \tfrac{1}{2}) \cdot \delta x, & y_1 &:= (j-1) \cdot \delta y \\ x_2 &:= (i - \tfrac{1}{2}) \cdot \delta x, & y_2 &:= j \cdot \delta y, \end{aligned}$$

   and the interpolation gives

$$\begin{aligned} v(x, y) := \tfrac{1}{\delta x\, \delta y}\Big[&(x_2 - x)(y_2 - y)v_1 + (x - x_1)(y_2 - y)v_2 \\ &+ (x_2 - x)(y - y_1)v_3 + (x - x_1)(y - y_1)v_4 \Big] \end{aligned}$$

Assignments:

Write two procedures `pathlines` and `streaklines` that are to be called within the time-stepping loop of the main program to compute the pathlines (particle lines) and streaklines for given initial positions and to write them to a file. The particles should be held in the following data structure:

```
struct particle{
       double x, y;
       struct particle *next;
}
struct particle_line{
       int length;
       struct particle *Particle;
}
struct particle_line *Particle_Lines;
```

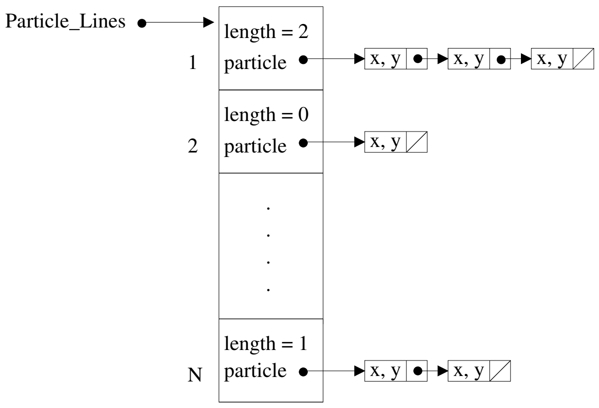


Fig. 6: Data structure `Particle_Lines` with the examples of different lengths

Thus, `Particle_Lines` is a list of all individual particle lines. Each `particle_line` represents the particle with a definite starting position at the current time step t_n. In the case of pathlines (particle lines), each `particle_line` has, consequently, the unitary length, while in the case of streaklines, this length is increased by 1 with each new particle appearing in the domain of interest. One has to take into account the following:

- `pathlines:` The first element of each `particle_line` from `Particle_Lines` is merely an auxiliary one, in order to ensure that the particles that are leaving the domain of interest can be easily deleted from the data structure.
- `streaklines:` The first element of each `particle_line` from `Particle_Lines` contains the particle position, where the particle must be injected at a given time.

Thus, in both cases the first "real" particle can be found in the second position of the `particle_line`, whereas the first position is filled by a "static" particle at the starting position. The required procedures can be written in a new module `visual.c`, to which also the already known function `output_uvp` belongs.
In total, one has to implement the following routines:

1.
   ```
   struct particle_line *set_particle(int N, double x1, double y1, double x2,
                                      double y2)
   ```
   The memory space required for the particle lines will be dynamically allocated and the particle positions are entered in each `particle_line`. Each `particle_line` has an initial length 1 (see Figure 6). The auxiliary element and the first "real" particle have the same coordinates. A pointer will be returned to the allocated region.

   The particles should be injected along the line segment with end points (`x1`, `y1`) and (`x2`,`y2`), the parameter `N` determining the number of particles equidistantly distributed along this line.

2.
   ```
   void move_particle(struct particle_line *Particle_Lines, int N, double dt,
                      double dx, double dy, int imax, int jmax, double** U,
                      double** V, int** Flag)
   ```

   The particle positions at time step t_n held in `Particle_Lines` (an array of `particle_line`) are updated, according to (B), to the particle positions of time step t_{n+1}. The particles that, as a result, leave the domain $\mathcal{R} = [0, i_{max}\,\delta x] \times [0, j_{max}\,\delta y]$ or run into the obstacle cells[2] are deleted from the list, and the memory occupied by these particles is returned using the `free` function. [3]

   `move_particle` may be called by `pathlines` and `streaklines`.

[2]You might have to extend the operation's signature to be able to identify obstacles.
[3] If the CFL condition is satisfied, particles at free-slip or no-slip boundaries can incorrectly disappear mostly in corner cells, but not in edge cells. If one wishes to prevent this, then the interpolation algorithm must be properly modified in the vicinity of edge cells by using the additional information that the normal velocity component should vanish not only at the boundary edge, but also along the entire edge.

3. `insert_particles (struct particle_line *Particle_Lines, int N)`
   The particles specified in `set_particle`-calls are injected along the given line segment by inserting a particle in the beginning of the list in each `particle_line` from `Particle_Lines`, with the position specified in the auxiliary element. Besides, the necessary storage must be allocated for the new particle.

   This function is called in `streaklines`.

4.
   ```
   void pathlines(struct particle_line *Particle_line, int N, double dt_path,
                  double t, double dx, double dy, double dt, int imax, int jmax,
                  double** U, double** V, int **Flag, const char *szProblem,
                  int timeStepNumber)
   ```
   This routine calculates the current positions of the particles to be traced at each time step using `move_particles` and overwrites their old positions in `Particle_Lines`. The current particle positions are appended, at time intervals `dt_path`, to the output file specified by the input parameter `outputfile` using `write_particles`. The header of this output file contains the quantities `xlength, ylength` and `N`. [4]

   Use the method write_vtkParticleFile() (described below) to write the pathline coordinates into a file. This function is called in the time-stepping loop of `main`.

5.
   ```
   void streaklines(struct particle_line *Particle_Lines, int N, double dt_inject,
                    double dt_streak, double t, double dx, double dy, double dt,
                    int imax, int jmax, double** U, double** V, int **Flag,
                    const char *szProblem, int timeStepNumber)
   ```
   At each time step, the current particle positions are computed with the help of `move_particle` and their old positions in `Particle_Lines` are overwritten. At time intervals `dt_insert`, particles are injected with the help of the function `insert_particles`, and at time intervals `dt_streak` the current particle positions are written by the function `write_particles` to the file specified in the input parameter `outputfile`. [4]. This file contains the quantities `xlength, ylength`.
   Data related to different times should also be visualised at different time points.

   Use the method write_vtkParticleFile() (described below) to write the pathline coordinates into a file. This function is called in the time-stepping loop of `main`.

6. The various time intervals `dt_path, dt_insert, dt_streak` should be read by `read_parameters`.

[4]Since the length of each time step `dt` is determined by a stepsize control scheme, the values for `dt_path, dt_inject, dt_streak` cannot be adjusted a priori in such a way that their respective "action time" in the time-stepping loop can be attained exactly.

Usually, however, `dt` is small compared to `dt_path, dt_inject, dt_streak` and permits each action to be performed the first time its scheduled action time is just surpassed.

- Apply the particle-tracing visualisation to the flow over a step and the streakline visualisation to the Karman vortex street!
- In order to write out the particle coordinate data for visualisation, the files visual.h and visual.c have been augmented by the method
```
write_vtkParticleFile(const char *prefix, const char *szProblem,
                      int timeStepNumber, struct particle_line *pl,
                      int numberOfParticleLines).
```
  This method has to be called in pathlines() and streaklines() with a corresponding prefix ("path" and "streak", e.g.).
- You may download the augmented files visual.h/visual.c from the web-site. In order to visualise the corresponding particles in Paraview, just open the desired data file and apply a Glyph filter (as for the usual flow field files). Instead of Arrows, you might choose Sphere0 under the Glyph drop-down menu.

6 Stream Function ψ and Vorticity ζ

The two-dimensional Navier-Stokes equations may be formulated not only in the so called "primitive" variables u, v, p (see Worksheet 1, (1) - (3)), but also in terms of the quantities ψ and ζ, called the stream function and vorticity. We will not go into the details of the numerical solution of this problem formulation. Instead, we are just going to define and interpret these two quantities in order to get additional tools to analyse the flow.

Definition:

Let u and v be the flow field velocities. Then, we define the *stream function* $\psi(x, y)$ by the relations

$$\frac{\partial \psi(x,y)}{\partial x} := -v, \qquad \frac{\partial \psi(x,y)}{\partial y} := u, \tag{9}$$

and the *vorticity* $\zeta(x, y)$ by the expression

$$\zeta(x,y) := \frac{\partial u}{\partial y} - \frac{\partial v}{\partial x}. \tag{10}$$

Interpretation:

- **Vorticity ζ:**
  ζ is a measure - as the name suggests - of the vortical motion strength in the velocity field. Figure 7 shows an example:

  In this thought experiment, an imaginary wheel is placed in the velocity field. In case (a) the u-velocity *increases* in y-direction, $\frac{\partial u}{\partial y} > 0$, forcing the wheel to rotate clockwise. In case

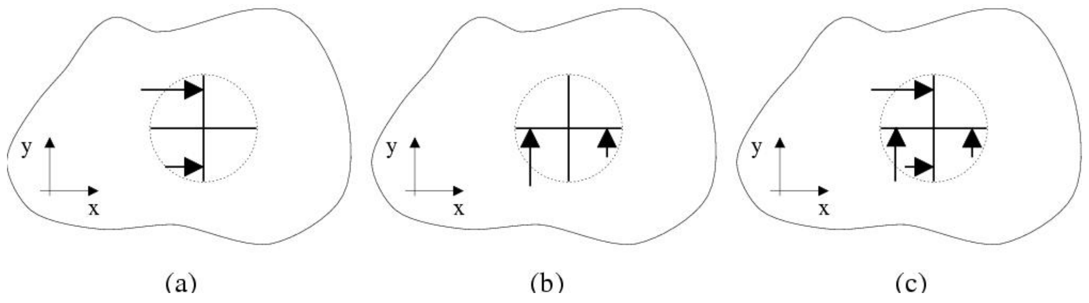


Fig. 7: Thought experiment illustrating the vorticity ζ.

(b), again a clockwise rotation is produced, but this time due to the v- velocity *decreasing* in x-direction: $\frac{\partial v}{\partial x} < 0$.

To get the total rotational effect, the cases (a) and (b) are put together in the case (c) resulting in the vorticity $\zeta = \frac{\partial u}{\partial y} - \frac{\partial v}{\partial x}$.

- **Stream function ψ:**
  First some words about the correct definition of the stream function:

  In order to get a well defined function of two variables x and y by its partial derivatives as in (9), a sufficient condition known as the *integrability condition* (interchangeability of the order of differentiation) must be fulfilled:

$$\frac{\partial^2 \psi}{\partial x \partial y} = \frac{\partial^2 \psi}{\partial y \partial x}.$$

  In the case of the stream function, the continuity equation (Worksheet 1, (3)) ensures the validity of this condition, since

$$\frac{\partial}{\partial x} u + \frac{\partial}{\partial y} v = 0 \iff \frac{\partial}{\partial x}\left(\frac{\partial \psi}{\partial y}\right) + \frac{\partial}{\partial y}\left(-\frac{\partial \psi}{\partial x}\right) = 0$$

$$\iff \frac{\partial^2 \psi}{\partial x \partial y} - \frac{\partial^2 \psi}{\partial y \partial x} = 0.$$

  For a physical interpretation of the stream function, we first introduce the notion of a *streamline.*

  Definition:

  > A streamline is a curve whose tangent is parallel to the velocity vector $\binom{u}{v}$ at a fixed time t at every space point (x, y).

  Between the stream function ψ and streamlines the following relationship holds: All contour (level) lines $\psi(x, y) = const$ are streamlines.

Furthermore, the volume flow rate $\dot{V}$ is constant ($\dot{V} = m$) between two streamlines $\psi(x,y) = \psi_a$, $\psi(x,y) = \psi_b$ with

$$m = \psi_b - \psi_a \ .$$

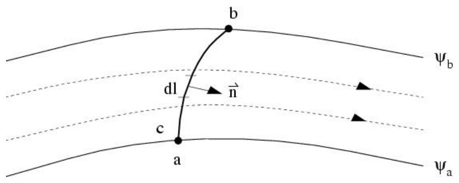


Fig. 8: Volume flow between streamlines.

Remark for those who are interested:

- The contour lines for a function of two variables $f(x,y)$ are defined as the set

$$N_c := \{(x,y): \ f(x,y) = c\}, \ c \in \mathbb{R}.$$

If f is sufficiently smooth, one can parameterize N_c as C^1–curve:
$\phi_c(s) = \binom{x_c(s)}{y_c(s)} \in N_c$, and we have:

$$f(\phi_c(s)) = f(x_c(s), y_c(s)) = c\,.$$

Therefore the gradient $\nabla f(\phi_c(s))$ is orthogonal to the tangent $\dot{\phi}_c(s)$ along the contour line $\phi_c(s)$ of f:

$$0 = \frac{df(\phi_c(s))}{ds} = \frac{\partial f}{\partial x}\frac{dx_c(s)}{ds} + \frac{\partial f}{\partial y}\frac{dy_c(s)}{ds} = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right) \cdot \begin{pmatrix} \dot{x}_c(s) \\ \dot{y}_c(s) \end{pmatrix} = \nabla f \cdot \dot{\phi}_c(s)\,.$$

Applying this expression to the stream function $\psi(x,y)$ and to the streamlines characterized by the tangent vector $\binom{u}{v}$, we conclude that the streamlines are the contour lines of the stream function, since

$$\nabla\psi \cdot \begin{pmatrix} u \\ v \end{pmatrix} = \left(\frac{\partial \psi}{\partial x}, \frac{\partial \psi}{\partial y}\right) \cdot \begin{pmatrix} u \\ v \end{pmatrix} = (-v, u) \cdot \begin{pmatrix} u \\ v \end{pmatrix} = -uv + uv = 0\,,$$

and the tangent directions $\dot{\phi}_c(s)$ along the contour lines run parallel to the velocity vector $\binom{u}{v}$.

- The volume flow rate $\dot{V}$ between two streamlines $\psi = \psi_a$, $\psi = \psi_b$ is the volume of fluid that passes through the curve c per unit time. For the volume V we have the relation $V = \int_\Omega dxdy$, and for the volume that passes through the curve c per unit time we get
$\dot{V} = \int_c \binom{u}{v} \cdot \vec{n}\, dl$ (see Figure 8).
One can see that the volume flow rate between two streamlines is constant:
$\dot{V} = \int_c \binom{u}{v} \cdot \vec{n}\, dl = \int_a^b \binom{u}{v} \cdot \binom{dy}{-dx} = \int_a^b u dy - v dx = \int_a^b \frac{\partial\psi}{\partial y} dy + \frac{\partial\psi}{\partial x} dx = \int_a^b d\psi = \psi_b - \psi_a\,.$

Assignments:

The quantities to be computed are the discrete values $\psi_{i,j}$ and $\zeta_{i,j}$. To avoid unnecessary tedious interpolations in the staggered grid, these quantities should be computed in the *right upper corner of each cell* (i, j), rather than in the cell center. The indices of the stream function are $\psi_{i,j}$ are $i = 0, .., i_{max}$, $j = 0, .., j_{max}$ and those of the vorticity $\zeta_{i,j}$, $i = 1, .., i_{max} - 1$, $j = 1, .., j_{max} - 1$.
One must implement the following:

- `void calculate_psi_zeta(dx,dy,imax,jmax,**U,**V,**Psi,**Zeta)` should first compute the discrete stream function $\psi_{i,j}$ using the defining equation

$$\frac{\partial \psi}{\partial y} = u \quad \text{in the discretized form} \quad \left[\frac{\partial \psi}{\partial y}\right]_{i,j} = \frac{\psi_{i,j} - \psi_{i,j-1}}{\delta y} \; .$$

  One may start with $\psi_{0,0} := 0$ calculating the lower boundary in a first recursion:

$$\psi_{i,0} = \psi_{i-1,0} - v_{i,0}\,\delta x \, , \quad i = 1, ..., i_{max} \; .$$

  Then, the rest follows:

$$\psi_{i,j} = \psi_{i,j-1} + u_{i,j}\,\delta y \, , \quad i = 0, ..., i_{max} \, , \;\; j = 1, ..., j_{max} \, ,$$

  Second, the discrete vorticity $\zeta_{i,j}$ is computed using the defining equation (10) according to the formula

$$\zeta_{i,j} := \frac{u_{i,j+1} - u_{i,j}}{\delta y} - \frac{v_{i+1,j} - v_{i,j}}{\delta x} \; .$$

  In both cases, the obstacle structure specified by the flag array must be considered. Thus, $\zeta_{i,j}$ can be set to zero inside obstacle cells and on their boundaries, and nothing is added for $\psi_{i,j}$ inside obstacle cells: $\psi_{i,j} = \psi_{i,j-1}$.

To visualize your computed data, use the function `void write_vtkFile` of `visual.c` (modified compared to worksheet 1). The new function `void calculate_psi_zeta` can be incorporated into the module `visual.c`.

7 Exercises

1. Implement the modifications to the program described in Section 1. Demonstrate, by providing your own examples, that the program works properly.

2. Supplement your program with the procedures described in Section 2 to ensure the possibility of performing simulations for complex geometries. To control the program functionality compute the von Karman vortex street (Example **4a**).

3. Implement the possibility to perform computations of the problems that are defined by setting the boundary pressure. Control your program with Example **4b** and compute Example **4c**.

4. Program the necessary functions for the visualisation with pathlines and streaklines (Section 5). It is useful to utilise the given functions contained in `visual.c` (one can download this file from the web-site of this practical course; attention: the version of worksheet 1 was augmented) Visualise the simulation results from Example **4a** with corresponding streaklines and pathlines in ParaView.

5. With the help of ParaView, visualise contour lines of the stream function (streamlines) and the vorticity for the following examples: driven cavity (Worksheet 1), flow past an obstacle (Example **4a**) and flow over a step (Example **4b**).