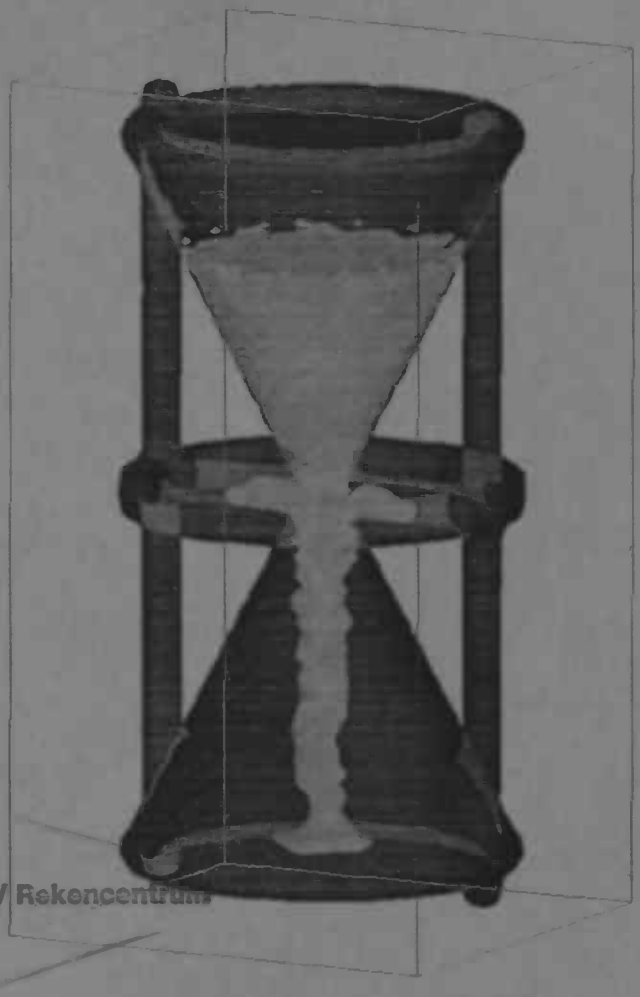
Free Surface Flow in Three-Dimensional Complex Geometries

Erwin Loots



Department of
Mathematics

RuG

Master's thesis

Free Surface Flow in Three-Dimensional Complex Geometries

Erwin Loots

University of Groningen
Department of Mathematics
P.O. Box 800
9700 AV Groningen

August 1997

Contents

1 Introduction 3

2 Mathematical model 5
- 2.1 The Navier-Stokes equations . . . 5
- 2.2 Boundary conditions . . . 6
  - 2.2.1 Solid boundary . . . 6
  - 2.2.2 Free surface . . . 7
  - 2.2.3 In- and outflow . . . 7

3 Numerical model 8
- 3.1 Apertures . . . 8
- 3.2 Labeling . . . 8
- 3.3 The Navier-Stokes equations discretized . . . 10
  - 3.3.1 The pressure Poisson equation . . . 10
  - 3.3.2 Free surface . . . 11
  - 3.3.3 SOR-iteration . . . 12
  - 3.3.4 In- and outflow . . . 13
  - 3.3.5 The Donor-Acceptor algorithm . . . 14
- 3.4 Determining the time step δt . . . 14

4 Pre- and postprocessing 16
- 4.1 CSG-trees . . . 16
  - 4.1.1 Attributes to primitives . . . 16
  - 4.1.2 Performance of the algorithm . . . 18
- 4.2 2D-rotation . . . 21
- 4.3 Combination . . . 21
- 4.4 Postprocessing . . . 21
  - 4.4.1 Movies . . . 21
  - 4.4.2 Streamlines . . . 22
  - 4.4.3 Fluxes . . . 23
  - 4.4.4 Filling ratios . . . 23
  - 4.4.5 Monitor points . . . 24
  - 4.4.6 Forces . . . 24
  - 4.4.7 Other information . . . 25

5 Results 26
5.1 Dambreak in a ball . . . 26
5.2 Falling drop . . . 27
5.3 Toricelli . . . 29
5.4 In- and outflow in a 2D-cylinder . . . 30
5.5 Demonstrations . . . 32
5.5.1 Dambreak in ball . . . 32
5.5.2 Y-junction . . . 32
5.5.3 Moving spheres-cylinders combination . . . 33

6 Conclusions 35

A Program description 36
A.1 Calling sequence . . . 36
A.2 Common block variables . . . 37
A.3 Subroutines . . . 40

B Pre- and postprocessing 47
B.1 Main calling sequence . . . 47
B.2 Subroutines . . . 47
B.3 Files . . . 49
B.3.1 Input files . . . 49
B.3.2 Output files . . . 53
B.4 Postprocessing tools . . . 54
B.4.1 AVS . . . 54
B.4.2 MATLAB . . . 54

Chapter 1

Introduction

Fluid dynamics plays an important role in our daily existence: blood flows in our bodies, air streams around wings of aircraft, ships wrestling through high waves and shaking cups of coffee are just a few examples.
In the science field of Computational Fluid Dynamics (CFD) these phenomena are examined in several ways. First, the physics behind fluid flows are translated into a mathematical model. Since 1845 the Navier-Stokes equations provide this model. After that, these equations are discretized (both in space and in time) to form a numerical model (solving analytically is only possible in very simplified cases).
The (approximate) solving of these discretized equations is generally done by a computer program which uses extensive iterative methods. The fourth step is the validation of the obtained data, usually done by comparison with theoretical and experimental results.

The bottleneck is usually the third step, since both memory and speed of even the most sophisticated computers were (and still are, in many aspects) not able to meet the demands. However, the last decade(s) significant improvements have been made. What generally is less known, though, is the even greater progress in the algorithms that currently happens.

Now, living in the end of the nineties, we have reached a stage where computer science meets a significant part of the wishes: to perform more or less simple 3D-calculations with free surfaces.
Thereto, in 1995 at the R*u*G , the development of a computer program called *ComFlo* has been started that solves fluid flow (without and with free surfaces) in 3D-complex geometries. Here the Navier-Stokes equations are solved on a Cartesian (rectangular) grid (see [3], [4], [2]).
This report describes the extension of this previous work in fields considering preprocessing, increasing the possibilities to define complex flow domains; postprocessing, to handle and evaluate the large amount of data that is created; other features that were implemented in the still developing program like in- and outflow; and several other actions to bring *ComFlo* to a stage where possibilities for the non-academic world appear.

In the following chapters, the theoretical model is explained (chapter 2). Then the numerical model, as implemented in *ComFlo*, is partly explained with aspects including apertures, labeling and the pressure Poisson equation in chapter 3 (for more detailed information about

this model, see [3] and [4]).
Afterwards, issues like pre- and postprocessing are handled, including the introduction of CSG-trees (chapter 4), followed by some results consisting of test cases and demonstrations of the present capabilities of *ComFlo* (chapter 5).
Also, in Appendix A, the structure of the main program (i.e. *ComFlo*) is explained. Finally, in Appendix B, detailed information about the pre- and postprocessing is given, including the structure of input files, a short description of the preprocessing program, and the wide variety of output files. These appendices can help new users of *ComFlo* to get started.

Chapter 2

Mathematical model

2.1 The Navier-Stokes equations

The motion of a viscous, incompressible fluid is governed by the unsteady, incompressible Navier-Stokes equations. For this report we assume that the pressure is scaled by the density ρ. Then the equations are:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0 \tag{2.1}$$

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z} = -\frac{\partial p}{\partial x} + \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right) + F_x + f_x \tag{2.2}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial z} = -\frac{\partial p}{\partial y} + \nu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2}\right) + F_y + f_y \tag{2.3}$$

$$\frac{\partial w}{\partial t} + u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial z} = -\frac{\partial p}{\partial z} + \nu\left(\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} + \frac{\partial^2 w}{\partial z^2}\right) + F_z + f_z \tag{2.4}$$

Here u, v, w are the velocity components in each direction (x-, y-, and z-) and p is the pressure. The symbol ν is the kinematic viscosity.

Further, $\boldsymbol{F} = \begin{pmatrix} F_x \\ F_y \\ F_z \end{pmatrix}$ is an external bodyforce (such as gravity), while $\boldsymbol{f} = \begin{pmatrix} f_x \\ f_y \\ f_z \end{pmatrix}$ is a virtual body force upon the fluid caused by geometry motion (by Newton's third law, there is no problem introducing this force). Note that the coordinate system containing the geometry is therefore relative to an inertial coordinate system. For instance, having $\boldsymbol{q}$ the absolute velocity of the relative system, $\boldsymbol{\omega}$ its angular velocity about the absolute origin and on the other hand $\boldsymbol{x}$ and $\boldsymbol{u}$ being the position and velocity of a fluid particle with respect to the relative origin, the virtual body force yields

$$\boldsymbol{f} = -\frac{\mathrm{d}\boldsymbol{q}}{\mathrm{d}t} - \boldsymbol{\omega} \times (\boldsymbol{\omega} \times \boldsymbol{x}) - \frac{\mathrm{d}\boldsymbol{\omega}}{\mathrm{d}t} \times \boldsymbol{x} - 2\boldsymbol{\omega} \times \boldsymbol{u} \tag{2.5}$$

Equation (2.1) says that mass is conserved in every volume, and equations (2.2) to (2.4) express conservation of momentum in each of the three directions.

Using the vector notation (with $\boldsymbol{u} = (u, v, w)^T$) and the divergence and gradient operators, the equations can be written simply as

$$\nabla \cdot \boldsymbol{u} = 0, \tag{2.6}$$

$$\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla)\boldsymbol{u} = -\nabla p + \nu(\nabla \cdot \nabla)\boldsymbol{u} + \boldsymbol{F} + \boldsymbol{f} \tag{2.7}$$

Here the term $(\boldsymbol{u} \cdot \nabla)\boldsymbol{u}$ can be replaced by $\nabla \cdot (\boldsymbol{u}\boldsymbol{u}^T)$ due to the fact that the velocity is divergence-free. Notice further that diffusive terms are represented by second order derivatives, while convection is expressed in first order derivatives.

2.2 Boundary conditions

The whole region where fluid is allowed to flow will be called Ω; the subset containing fluid is denoted by Ω_f. Ω is constant in time, Ω_f is not. Boundary conditions have to be described at the boundary of Ω_f. The solid boundary, $\partial\Omega \bigcap \partial\Omega_f$, is handled in a different way than the free surface, $\partial\Omega_f \backslash (\partial\Omega_f \bigcap \partial\Omega)$. Moreover, a third kind of boundary condition occurs at inflow and outflow areas, which deals with interaction with respect to fluid outside the computational domain.

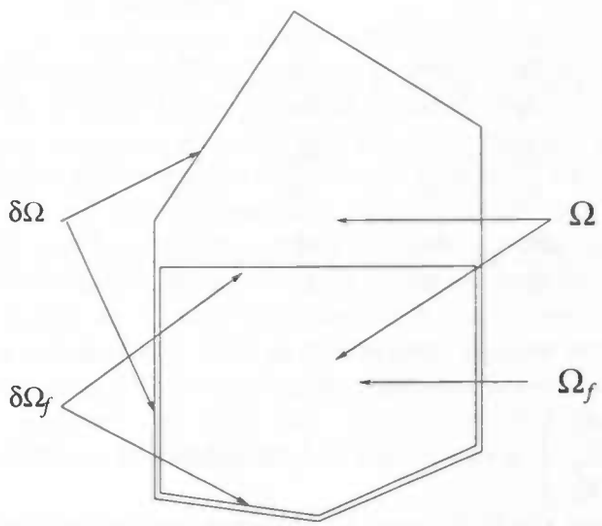


Figure 2.1: *The flow domain and edges*

2.2.1 Solid boundary

There are two forms of solid boundary: $\partial\Omega_{ns}$ and $\partial\Omega_{fs}$, meaning respectively no-slip walls and free-slip walls.

The conditions are:

$$\begin{aligned} u &= 0 \quad \text{on} \quad \partial\Omega_{ns}, \\ u_n &= 0 \quad \text{and} \quad \tau = 0 \quad \text{on} \quad \partial\Omega_{fs}. \end{aligned}$$

where $\tau = \frac{\partial u_t}{\partial n}$ represents the tangential stress. $u_n = \boldsymbol{u} \cdot \boldsymbol{n}$ is the velocity component perpendicular to the surface, just as $u_t = \boldsymbol{u} \cdot \boldsymbol{t}$ is the tangential velocity component.

This means that in both cases the normal component of the velocity equals zero (simply forbidding fluid to move through solid walls), whereas in the former case also the tangential velocity component is equal to zero, expressing the fact that fluid sticks to the wall due to the viscosity. The second condition at free-slip walls, $\tau = 0$, conversely, expresses that u_t does not decay when approaching the wall.

2.2.2 Free surface

The boundary with respect to the free surface needs additional boundary conditions. These are:

$$-p + 2\mu \frac{\partial u_n}{\partial n} = -p_0 + 2\gamma H \tag{2.8}$$

$$\mu \left(\frac{\partial u_n}{\partial t} + \frac{\partial u_t}{\partial n} \right) = 0 \tag{2.9}$$

where p_0 is the reference pressure outside the fluid (the atmospheric pressure), γ the surface tension and $2H = \frac{1}{R_1} + \frac{1}{R_2}$ is the total curvature of the surface. $\mu = \rho\nu$ is the dynamic viscosity. R_1 and R_2 denote the curvatures of the intersection of the free surface with two planes, orthogonal with respect to each other, through the normal; the total curvature is independent of the choice of the two planes.

2.2.3 In- and outflow

At an inflow area the velocity $\boldsymbol{u}$ is given: $\boldsymbol{u} = \boldsymbol{u_{in}}$. This velocity can, of course, be time-dependent.

In outflow areas usually a homogeneous Neumann condition for all velocity components is prescribed: $\frac{\partial \boldsymbol{u}}{\partial n} = 0$ instead of a similar Dirichlet condition in order to prevent the creation of a boundary layer. It is also convenient to set the pressure at its atmospherical value: $p = p_0$.

In the following chapter the discretization of the above model is discussed.

Chapter 3

Numerical model

After having handled the mathematical model, now the numerical model is studied. For readability purposes, only graphics of two-dimensional geometries are shown; extension to 3D is straightforward.

3.1 Apertures

We start with laying a rectilinear (Cartesian) grid over the three-dimensional flow domain Ω. Since the form of Ω is generally not rectilinear, and neither is the (time-varying) configuration of the fluid (Ω_f), the grid cells (boxes in 3D) cut the boundaries in several ways. To order the possibilities, *apertures* are introduced.
Apertures are divided into two classes: First the *volume apertures*. In each cell, the geometry aperture F_b defines the fraction of the cell volume in which fluid should be able to flow, i.e. the part of the cell contained in Ω. On the other hand, the fluid aperture F_s indicates the fraction of the cell which is indeed occupied by fluid. Note that the latter aperture is time-dependent. Obviously, $0 \leq F_s \leq F_b \leq 1$.
The other class of apertures is the *edge-aperture*. At every cell surface, the part of the surface contained in Ω is called A_x, A_y or A_z (dependent of the orientation of the surface). Thus an edge-aperture between two cells indicates the part of the dividing surface open to flow.
Edge-apertures with respect to the fluid itself are not defined. Figure 3.1 shows an example of apertures defined in a part of Ω.
Note that, in the computation, Ω is only represented by F_b and A_x, A_y and A_z while the original notice of the geometry has been lost. This approach allows to handle arbitrary complex forms of Ω, as shown in chapter 4.

3.2 Labeling

Now the equations (2.1), (2.2), (2.3) and (2.4) must be discretized. This is done on a so-called *totally staggered grid*, which means that the velocity components are located in the middle of cell faces, between two cell centres (although not necessarily equally distanced from them), and the pressure is situated in cell centres.
The different treatment of pressure inside cells and velocities between cells is expressed in the labeling of these cells and velocities. With respect to the time-dependency there are

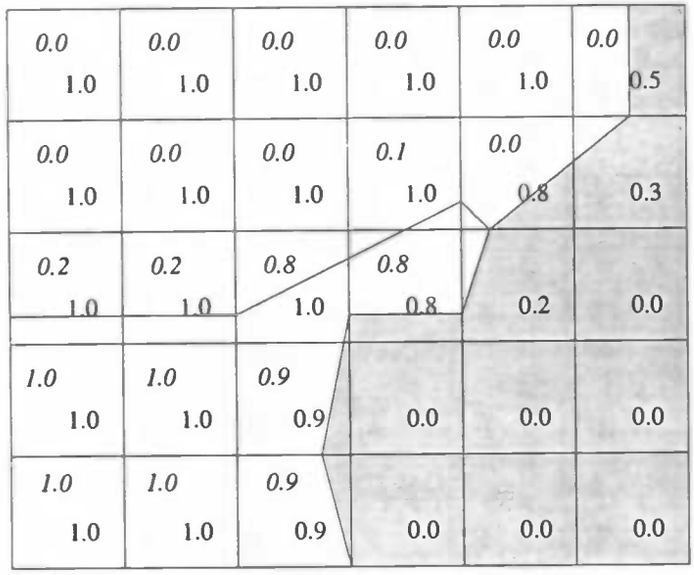


Figure 3.1: *Geometry apertures F_b and (in italics) fluid apertures F_s* (rounded to one decimal place)

two classes of labeling, each consisting of cell labels and velocity labels; the latter ones are acquired from the former.

Labels based on the geometry are time-independent. First **F**-(flow) cells are all cells with $F_b \geq \frac{1}{2}$. After that, all cells with city-block distance 1, meaning cells sharing a cell face with an **F**-cell (and, consequently, $F_b < \frac{1}{2}$) are labeled as **B**-cells. The remaining (exterior) cells are called **X**-cells; they are not interesting.

Velocity labels are a combination of the labels of the cells where these velocities lie between. There are five combinations: **FF, FB, BB, BX,** and **XX**. The combination **FX** does not occur because of the sequence of labeling.

The second labeling class, being time-dependent, is that of free-surface labels. The cell labels here are a subdivision of **F**-cells (the other two geometry labels represent cells which are less than half-filled and therefore neglected for this purpose). First, cells with $F_s = 0$ are obviously empty: they become **E**-cells. After that, cells having city-block distance 1 from **E**-cells are called **S**-(surface) cells. The remaining **F**-cells keep their **F**-label, but now in the sense of 'fluid' instead of 'flow'. After that eight free-surface velocity labels can be recognized: **FF, FS, SS, SE, EE, FB, SB** and **EB**.

Note that the geometry labeling uses decreasing F_b, while free surface labeling uses increasing F_s.

A final labeling step is the mutation of specified **B**- cells into **I**-(inflow) cells and **O**-(outflow) cells.

Figure 3.2 shows the obtained labels of the geometry drawn in figure 3.1.

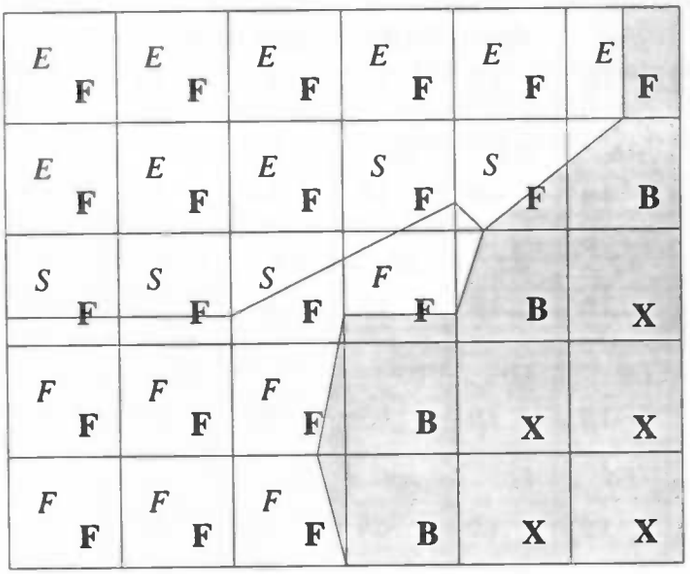


Figure 3.2: *Geometry labels and (in italics) free-surface labels*

3.3 The Navier-Stokes equations discretized

As the described labeling method does not cause any stability problems (see [3]), the Navier-Stokes equations can be discretized explicitly in time. The well-known first-order Forward-Euler method yields

$$\nabla \cdot u^{n+1} = 0 \tag{3.1}$$

$$u^{n+1} - u^n + \delta t \nabla p^{n+1} = \delta t \boldsymbol{R}^n \tag{3.2}$$

where $\boldsymbol{R}^n$ contains all internal, external and body forces:

$$\boldsymbol{R}^n = \left(\nu \nabla \cdot \nabla u^n - \nabla (u^n u^{nT}) + \boldsymbol{F}^n + \boldsymbol{f}^n\right)$$

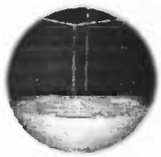
Here n and $n+1$ denote the old and new time level, respectively. δt is the time step; it can vary in time.

3.3.1 The pressure Poisson equation

By substituting (3.1) into (3.2) we get

$$\nabla \cdot \nabla p^{n+1} = \nabla \cdot (\frac{u^n}{\delta t} + \boldsymbol{R}^n) \tag{3.3}$$

which is called the *Poisson equation* for the pressure. However, boundary conditions for (3.3) are not yet available, since they only involve u. A better strategy is to discretize and substitute boundary conditions first, and manipulate equations afterwards.



Therefore we use the discrete operators D_h and G_h, the analogons of $\nabla\cdot$ and ∇, respectively. Further we split the divergence operator in D_h^F, operating on the inner domain, and D_h^B, operating on the boundary. Thus (3.1) becomes $D_h^F u^{n+1} = -D_h^B u^{n+1}$. However, we do not know the boundary velocities at the new time step yet; in fact, they should be computed from the internal velocities at the new time step. That is why we simply set $u^{n+1} = u^n$ on the boundary; the error introduced here is of the same order (δt) as already caused by the Euler discretization.
So the full discretization of (3.1) and (3.2) is

$$D_h^F u^{n+1} = -D_h^B u^n \qquad \text{in } \mathbf{F}\text{-cells} \tag{3.4}$$

$$u^{n+1} = u^n + \delta t \boldsymbol{R}_h^n - \delta t G_h p^{n+1} \qquad \text{on } \Omega_f \tag{3.5}$$

By substituting the latter into the former at this point we get

$$D_h^F G_h p^{n+1} = D_h^F(\frac{u^n}{\delta t} + \boldsymbol{R}_h^n) + D_h^B(\frac{u^n}{\delta t}) \quad \text{in } \mathbf{F}\text{-cells.}$$

Now this Poisson equation can be solved. The operator $D_h^F G_h$ is a seven-point molecule in the grid, consisting of a central (C_p) and six other coefficients (C_w, C_e, C_n, C_s, C_u and C_d) at city-block distance one. But first we take a look at the treatment of the free surface.

3.3.2 Free surface

Near the free surface, the equations (2.8) and (2.9) must somehow be discretized.
First we take a look at the velocities. The **FF**-, **FS**- and **SS**-velocities near the free surface are obtained by solving the momentum equations. In the discretization molecules for the derivatives **SE**- and **EE**-velocities appear. They are obtained in the following way:

- **EE**-velocities. Here discrete versions of (2.8), considered on surfaces parallel to the $x = 0$-, $y = 0$- and $z = 0$- planes, are used, taking some equations out of

  $$\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x} = 0,\ \frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} = 0,\ \frac{\partial w}{\partial y} + \frac{\partial v}{\partial z} = 0$$

  dependent on the exact configuration of adjacent **SS** velocities. If an **EE**-velocities has no **SS**-neighbour, there is likely to be not much fluid there, and the **EE**-velocity is taken zero. The method is more explicitly treated in [4].
  However, a neighbouring velocity can be **SE**, which has not been accounted for yet.

- **SE**-velocities. These velocities can appear in the momentum equations or in the equations needed for **EE**-velocities. Thus, there are a lot of possible configurations. In the corresponding **S**-cell $\nabla \cdot u = 0$ is demanded. The discretization uses (in 3D) six velocities:

  $$\frac{u_e - u_w}{h_x} + \frac{v_n - v_s}{h_y} + \frac{w_u - w_d}{h_z} = 0$$

  with one of them being the desired **SE**-velocity. If the other five are known, which is the case if they are computed from the momentum equations (**FS**- and **SS**-velocities) or from the boundary conditions (**BS**), the **SE**-velocity is easily solved. However, if one or more of the other five is **SE**, other measures have to be taken, see [4].

Now we consider the conditions for the pressure. In E-cells, the pressure is set to its atmospheric value. The pressure in S-cells is less easily obtained.
Considering equation (2.8), the pressure p_f on the free surface is taken to be $p_f = p_0 - 2\gamma H$, neglecting the term $\mu \frac{\partial u_n}{\partial n}$. Because pressures are described in cell centra, p_f is considered to be linearly interpolated between p_F and p_S, the pressures in the **F**- and **S**-cell, respectively (see figure 3.3):

$$p_S + (\eta - 1)p_F = \eta p_f, \quad \text{where} \quad \eta = \frac{h}{d} \tag{3.6}$$

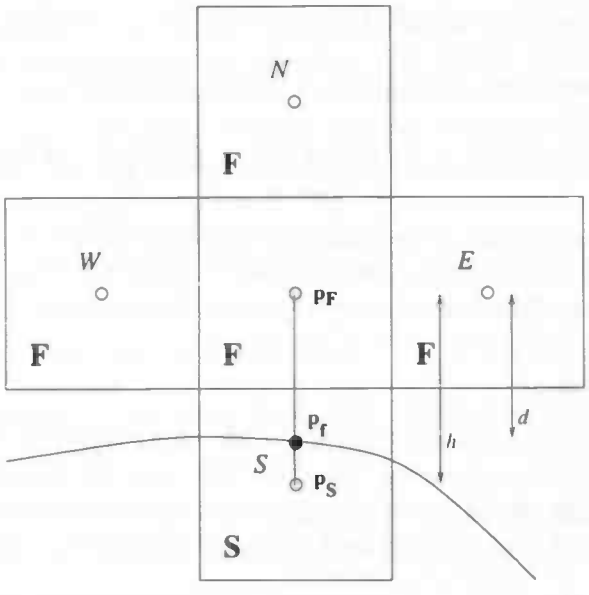


Figure 3.3: *Pressure locations at free surface and in the neighbouring cells*

This equation can easily be combined with the Poisson equation in the **F**-cell as follows: Using the introduced coefficients and not using C_u and C_d in this 2D-example,

$$\begin{aligned} C_p p_p + C_e p_e + C_n p_n + C_w p_w + C_s p_s &= f_p \qquad \text{becomes} \\ (C_p + (1-\eta)\, C_s)\, p_p + C_e p_e + C_n p_n + C_w p_w &= f_p - \eta\, C_s\, p_f \end{aligned}$$

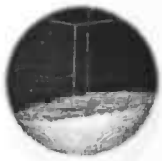
Now the Poisson equation can be solved; this is done by SOR-iteration.

3.3.3 SOR-iteration

Using a relaxation parameter ω, SOR for the pressure yields

$$\begin{aligned} p_p^{n+1^{k+1}} &= (1-\omega) p_p^{n+1^k} + \\ \frac{\omega}{C_p} & \left(-C_n p_n^{n+1^k} - C_e p_e^{n+1^k} - C_u p_u^{n+1^k} - C_s p_s^{n+1^{k+1}} - C_w p_w^{n+1^{k+1}} - C_d p_d^{n+1^{k+1}} + f_p \right) \end{aligned}$$

where $f_p = \nabla \cdot (\frac{\boldsymbol{u}}{\delta t} + \boldsymbol{R}^n)$. Here ω can be varied automatically to obtain rapid convergence (see [1]).



The advantage of SOR, besides its simple implementation, is the ease with which it vectorizes and parallellizes. The independence of the vector elements is attained by using *Red-Black ordening*, dividing the cells into two groups, like a (3D equivalent of) checkerboard.
However, to avoid *if*-statements in this part of the code, the Poisson equation should not only hold for **F**-cells, but for all other cells as well. This demand can be fulfilled relatively easily:

- **E**,**X** and **B**- cells: Here the central coefficient (C_p) is set to unity while all other are zero. f_p is set equal to p_0, causing the Poisson equations in these cells to be redundant.
- **S** - cells: Here the equation (3.6) is used: According to figure 3.3, for the molecule in the **S**-cell $p_p = p_S$ and $p_n = p_F$ holds. Setting $C_p = 1$, $C_n = \eta - 1$, $f_p = \eta p_f$ and other coefficients at zero, the molecule becomes

$$1 \cdot p_S + (\eta - 1) \cdot p_F = \eta \cdot p_f$$

what was needed.

Having performed the SOR-iterations, the pressure at the new time step is entirely known. The new momentum velocities are simply found from

$$u^{n+1} = u^n + \delta t(-\nabla p^{n+1} + \boldsymbol{R}_h^n) \tag{3.7}$$

Hereafter the other velocities near the solid wall (see [3]) and the free surface can also be computed.

3.3.4 In- and outflow

In- and outflow cells are mutated **B**-cells restricted to different conditions. They are treated separately after the pressure Poisson equation has been solved.
The velocities between an inflow cell and a regular **F**-cell get a distinct (**FI**-)label. Independent of the fluid configuration, these velocities always get a prescribed value. These values are set together with the other boundary velocities.

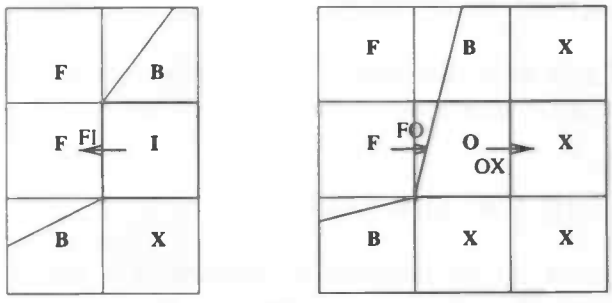


Figure 3.4: *inflow cell (left) and outflow cell (right)*

Outflow cells are controlled less easy. Consider the triplet of **F**-,**O**-, and an **X**- cell next to each other (Configurations where the other side of the **O**-cell is not an **X**- cell (rather

rare) are not yet supported by the program code). Between the first two cells, a momentum equation is solved; hence that **FO**-velocity is treated like **FF** and **FS**.

To ensure that fluid flows through the cell smoothly, the outflow condition $\frac{\partial u}{\partial n} = 0$ is discretized by setting the **OX**-velocity equal to the **FO**-velocity. Finally, the pressure in the outflow cell is set to p_0.
Although there are other possible implementations (prescribing second derivatives of the velocity, for example), this way turned out to be practical.

3.3.5 The Donor-Acceptor algorithm

After the new velocity field has been computed entirely, the free surface is likely to be moved. Therefore the free-surface labeling must be adjusted; and, before this, the fluid aperture F_s. A natural way to achieve this is by computing fluxes between cells. They are computed out of the velocity, the area of the cell surface between the two cells and the edge apertures (since a solid wall is inpenetrable the flux is lineary dependent of the non-boundary area of a cell face). A nonzero flux indicates that fluid is to be transported from a donor cell to an acceptor cell. However, there are a few restrictions to be considered:

- The donor cell may not loose more fluid than it already contains, so the maximum amount is $F_s \cdot V_{\text{cell}}$. Note that an **E**-cell cannot be a donor cell.
- The acceptor cell cannot receive more fluid than the amount of available void space, so the maximum amount is $(F_b - F_s) \cdot V_{\text{cell}}$.
- In a surface cell, the position of the fluid is important; to anticipate the creation of 'holes', the fluid must be 'tamped' towards **F**-cells.
- Rounding effects can cause that $0 \leq F_s \leq F_b$ not longer holds; in that case F_s has to be adjusted.

Fluxes that are computed out of a momentum velocity do generally not suffer of violations of the first two points mentioned. However, particularly at the solid boundary, several adjustments are needed to prevent the change of the total amount of fluid in Ω due to cumulative effects.
Having computed all new fluid apertures, the free surface labels for the next time step can be adjusted. This operation marks the end of a single time cycle.

3.4 Determining the time step δt

A significant speed boost can be achieved by permitting a varying time step, just as the grid supports stretching. In calm intervals, when the fluid is hardly moving, this time step should easily be increased while during more violent fluid motion a reduction is necessary. The above considerations are quantified in the CFL-number: the Courant-Friedrichs-Levy number, defined as

$$\text{CFL} = \text{MAX}\left(\frac{|u| \cdot \delta t}{h_x}, \frac{|v| \cdot \delta t}{h_y}, \frac{|w| \cdot \delta t}{h_z}\right)$$

where h_x, h_y and h_z are the distances between two cell centra in each direction. Fourier analysis shows that this number must not exceed 1.
This criterium can also be explained by looking at the Donor-Acceptor algorithm: if the CFL-number is less than 1, then the movement of the fluid towards a cell does not exceed the width of that cell; in other words, each time step the fluid is transported over no more than one cell. Therefore changes in the free surface labeling are limited; moreover, the precise transportation of fluid in the donor-acceptor algorithm does not allow a trans-cell movement; a too large CFL-number will be the cause of fluxes which are too large and would 'overfill' destination cells. On the other hand, one must not be too severe; the overall CFL-number is only computed out of velocities where fluxes are defined.

In the program, the overall CFL-number is taken to be the maximum over all cells of the number defined above.
The time step is doubled if the CFL-number is small enough for a few consecutive cycles. On the other hand, if it turns out to be larger than a certain constant C in the present cycle, the time step is immediately halved. The present cycle is not repeated since it is rather awkward to reset all variables. For safety reasons, therefore, the threshold constant C should be taken less than 1.

Chapter 4

Pre- and postprocessing

A main characteristic of more industrial applications is the presence of geometries that cannot mathematically be described in a simple way: the problem doesn't fit in a cube, is not exactly axisymmetrical or has other special features.
Therefore, a method is needed that handles geometries which are complex in the full sense of the word. Also, this method must allow easy insertions in and adaptations of existing geometries. Moreover, a nontrivial initial configuration of the liquid could be needed in the same manner as the geometry.
In the following, two different -and complementing- methods are discussed: *CSG-trees* and *2D-rotation.*

4.1 CSG-trees

CSG stands for *constructive solid geometry.* The idea of this method is relatively simple: a new object is created by applying the union, intersection, or difference operation to two previously specified objects. This operation can be repeated, thus creating a complex object out of simple primitives. More specified, the process is:

1. Define two primitives and drag them, eventually rotated, to specified places.
2. Choose an operator to create a new object.
3. Continue with this object and other primitives until the final shape is reached.

An object designed with the procedure above is called a CSG-tree, and since the operators are binary, the tree is a binary one.
The leaves of the tree contain the primitives, while in each internal node one of the three set operations is placed. This means that an object built up from N primitives is represented by a tree of exactly $2N - 1$ nodes (which is easy to see with induction), having a depth of at least $^2\log N$.

4.1.1 Attributes to primitives

A primitive object is described by a number of parameters. The challenge is to use as few as possible primitives with as few as possible parameters and yet allowing a wide range of

possibilities. The primitives used by the program are only the box, cylinder, cone and ellipsoid (planes must be represented as thin boxes).

All the primitives have at least three parameters giving a reference position (e.g. for an ellipsoid: the centre), a few parameters for other aspects like radius or length, and two or three describing a rotation. That is, a primitive object is assumed to be laid along the x-axis. Rotation parameters describe a sequence of an x-axis, y-axis and z-axis rotation; objects with x-axis symmetry, like a cylinder, therefore only need two rotations. The rotation is described using a 4×4 rotation matrix.[1]

For example, a rotation around the x-axis is described as

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1' \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

The matrix will be called $R_x(\theta)$. The other two rotations are obtained by shifting the x-, y- and z-coordinates around(see [5]). Subsequent rotations are obtained by multiplying the matrices, where the last rotation is set at the front of the concatenation. So the result, namely $R_z(\omega)R_y(\phi)R_x(\theta)$, is:

$$\begin{pmatrix} \cos\omega\cos\phi & \cos\omega\sin\phi\sin\theta \\ & -\cos\theta\sin\omega \\ \cos\phi\sin\omega & \cos\omega\cos\theta \\ & +\sin\omega\sin\phi\sin\theta \\ -\sin\phi & \cos\phi\sin\theta \\ 0 & 0 \end{pmatrix} \begin{matrix} \cos\omega\cos\phi\cos\theta & 0 \\ +\sin\omega\sin\theta & \\ -\cos\omega\sin\theta & 0 \\ +\cos\omega\cos\phi\cos\theta & \\ \cos\phi\cos\theta & 0 \\ 0 & 1 \end{matrix}$$

The question arises if this rotation is powerful enough; indeed, the standard form is described as a rotation around a single arbitrary axis. However, it is rather difficult to specify an axis given the position the object should be set to, as we will show below.

Let us assume we have an object, laid along the x-axis. The farthest point on the x-axis, P, is $(d, 0, 0)^T$ and O also belongs to the object. The question is by which rotation axis l the object will be laid with P placed to $Q = (a, b, c)^T$.

Since O stays the same, the axis is in a plane V through O (l goes through O as well), equally distanced from P and Q. It is easy to see that the two generating vectors are

$$v_1 = (a + d, b, c)^T \quad \text{and} \quad v_2 = (0, -c, b)^T$$

[1] In computer graphics, one of the aims is to describe general transformations in a standard way. To include transformations such as scaling and translation in a single operator (matrix), a fourth vector element is introduced, called the homogeneous coordinate; this is in practice set to 1. E.g., a translation in x-direction over a distance t is written as $x' = 1 \cdot x + t \cdot 1$, giving a matrix $\begin{pmatrix} 1 & 0 & 0 & t \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$.

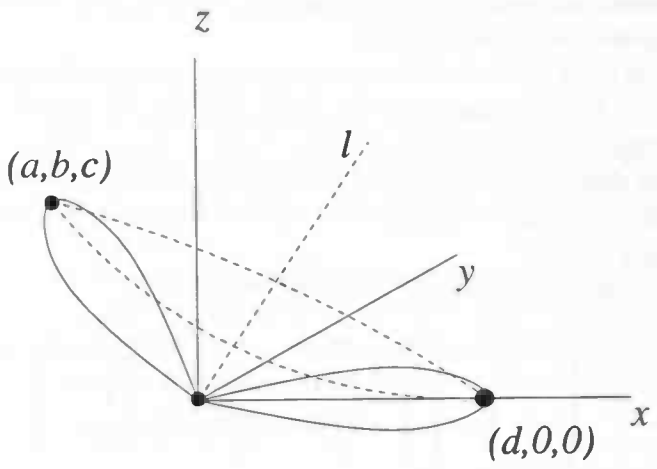


Figure 4.1: *rotation of object around rotation axis* l

expect for $d = -a$, when, using $d^2 = a^2 + b^2 + c^2$,

$$v_1 = (0,1,0)^T \quad \text{and} \quad v_2 = (0,0,1)^T$$

suffices.
The point K on l which rotates P onto Q is of the form $\lambda v_1 + \mu v_2$. This rotation has the property that $OK \perp KP$ and $OK \perp KQ$. Writing out the inner product, the former restriction leads to

$$\lambda^2 \left((a+d)^2 + b^2 + c^2\right) + \lambda\left(-d(a+d)\right) + \mu^2\left(b^2 + c^2\right) = 0 \tag{4.1}$$

while the latter turns out to deliver, after some algebra, the same equation.
For example, let $P = (1,0,0)^T$ and $Q = (0,1,0)^T$, solving for λ gives $\lambda = \frac{1 \pm \sqrt{1-8\mu^2}}{4}$. Having $v_1 = (1,1,0)^T$ and $v_2 = (0,0,1)^T$, solutions in the ground plane ($\mu = 0$) are $(0,0,0)^T$ from an axis prependicular to V and $(\frac{1}{2},\frac{1}{2},0)^T$ on an axis in V while μ is generally restricted to $|\,\mu\,| \leq \frac{\sqrt{2}}{4}$. The elevation of l w.r.t. the ground plane determines the body rotation of objects not symmetrical in x-direction. This makes the choice of a single rotation axis, not even mentioning (4.1), quite difficult.

Therefore, we have decided to perform the rotation in three steps, namely by using the three main rotation axes. This method is easier (see figure 4.2):
Rotation around x-axis (θ): gives the desired angle with ground plane of an object not axisymmetrical around its body axis (the x-axis, initially), like a box .
Rotation around y-axis (ϕ): gives the height above the ground plane: $\sin\phi = \frac{c}{d}$.
Rotation around z-axis (ω): gives the direction when projected on the ground plane: $\tan\omega = \frac{b}{a}$.

4.1.2 Performance of the algorithm

The tree scan algorithm checks for every point if it is in the desired geometry, that is, if it is in the root of the tree (see figure 4.3 for an example). Unfortunately, we cannot use the

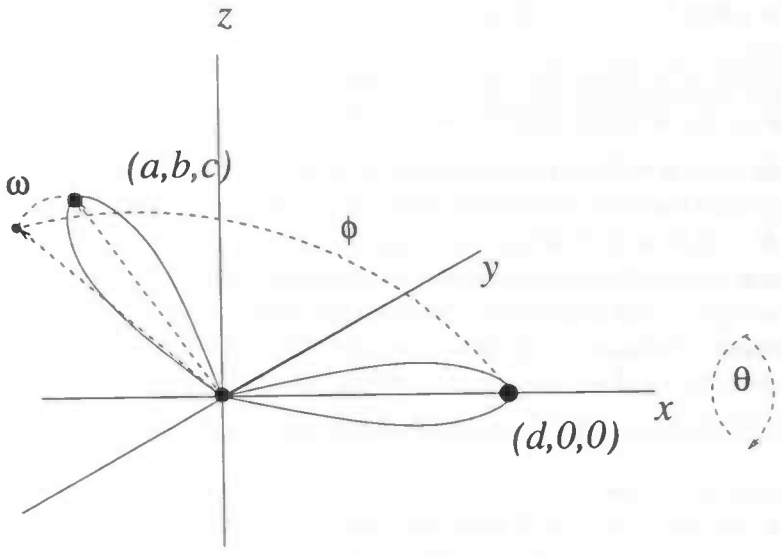


Figure 4.2: *rotation of object using three rotation axes*

sophisticated procedures used by red-black trees or even binary search trees.

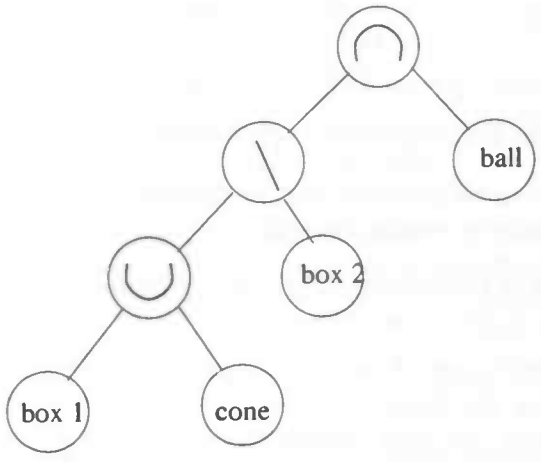


Figure 4.3: *An example of a CSG-tree*

Suppose a tree, build up out of M nodes, contains N primitives in the leaves. All M nodes have to be examined. For each point the following steps must be taken:
1. call left branch
2. call right branch
3. in-tree := left *operator* right, i.e. if the point P satisfies $P \in$ left branch *operator* right branch, it is in the tree.

Hence in each internal node, after two recursive calls only one statement is needed. For the leaves, where the primitive is, it is less simple:

1. Translate the point back according to the given translation parameters of the tree.
2. Rotate the point back using the rotation matrix.
3. Check if it is in the original object, laid along the x-axis.

Having a the time needed for these three leave actions, and b the time to perform the set operation, the total time is $T = aN + b(M - N)$, $b \ll a$. Since $M = 2N - 1$, the total time satisfies $T = aN + b(N - 1) \approx aN$.

Hence attention should be paid to methods which are smart enough to not further descend when not necessary, i.e. are able to skip some of the leave actions (which take the most time). It follows also that a balanced tree is not necessarily profitable; it depends on the specified geometry which choice is the smartest, i.e. skips the greatest parts of a subtree most often.

The algorithm can be slightly modified to get the result faster:

- The geometries consist in the program in a mere vage form: not the object itself is described, but the fraction of each cell and cell face: the apertures. If per cell only the eight vertices are examined, and they all return 'out' or 'in', assuming this result for the whole cell and walls is not too risky.

- Often only one of the two branches of a subtree needs to be examined; this follows from the following logical expressions, where the left branch has already returned a boolean value, and the right branch (denoted by $\uparrow$ right) is still unknown :

$$\begin{array}{rcll} \text{IN} & \cup & \uparrow \text{right} & \Rightarrow \text{IN}; \\ \text{OUT} & \cap & \uparrow \text{right} & \Rightarrow \text{OUT}; \\ \text{OUT} & \setminus & \uparrow \text{right} & \Rightarrow \text{OUT} \end{array}$$

  Similar expressions hold when the right branch is examined first.

- A bounding box can be attained at each primitive; combined bounding boxes can then be set automatically in each node while building up by:

  1. The vertices of the bounding box of a primitive are rotated; the rotated primitive is contained in the rotated box;
  2. The bounding boxes of two subtrees A and B are combined to a bounding box C by examining the three directions, for example in the x-direction, where A_l and A_r are the left and right coordinates of bounding box A, respectively:

$$\begin{array}{lllll} \text{union:} & C_l = & \min(A_l, B_l); & C_r = & \max(A_r, B_r); \\ \text{intersection:} & C_l = & \max(A_l, B_l); & C_r = & \min(A_r, B_r); \\ \text{difference:} & \textit{if} & A_l \geq B_l & & \\ & \textit{then} & & C_l = & \max(B_r, A_l) \\ & \textit{else} & & C_l = & A_l; \\ & \textit{if} & A_r \leq B_r & & \\ & \textit{then} & & C_r = & \min(B_l, A_r) \\ & \textit{else} & & C_r = & A_r; \end{array}$$

  3. A simple check while scanning the tree is in this way: If a point is outside the bounding box then further descending is not necessary.

4.2 2D-rotation

This method simply means that plain objects, defined in a 2D-plane, are rotated by a specified angle around the z-axis. In an input file simple 2D-objects are specified while the program (*GeoMake*) cares for the exact locations in 3D.
At an earlier stage it has been found that a description of rather diverse figures in 2D can be reached using only lines (which allow triangles and rectangles) and circle segments (which allow disk segments) (see [7]). This idea has been developed further.
The user defines the amount of arcs and lines and specifies thereafter all parameters (begin- and end coordinates, begin- and end angles and filling parameters (to obtain 2D-areas out of these 1D-varieties)) that are needed. The obtained areas are cut out of the x-z-plane, thus indicating the area which is open for flow. Now the entire plane is rotated by a quarter, a half, or for the whole. An example of an input file is described in Appendix B.

4.3 Combination

The two methods just described can easily be combined. In practice, this is mostly done by creating a super-tree with the CSG-tree as the left branch and the rotated combination as the right, having a union in the root. It can also be handy to incorporate the rotated 2D-combination, interpreted as a single object, in the CSG-tree. This combination is adequate to describe a lot of industrial-applicated configurations. As suggested before, not only a geometry can be created, but also the *fluid* configuration. However, the fluid is bounded by the part of the geometry that is open to flow. So the geometry data (i.e. the apertures) is used to satisfy the invariant $F_b \geq F_s$.
Of course, degenerated cases are also possible: if the problem does not call for the use of a tree or a 2D-rotation, these parts are used as neutral elements in the union of the super-tree. The specification of dummy input files suffices in that case.
In figure 4.4, an example of the possibilities of a combination is found. The 'rings' parallel to the ground plane are created from 2D-structures; the rest consists of a nine-node tree. The input files used to create this figure are found in Appendix B, together with a short explanation.

4.4 Postprocessing

The final step of the computation procedure is to process and to interpret the enormous amount of data. Below a list of the methods, as implemented in *ComFlo*, is discussed.

4.4.1 Movies

At equidistant time intervals, the whole state of the system (i.e. liquid configuration, velocities and pressure) is written to file. Using the visualization system AVS, resulting images from each state can be combined, thus creating movies. It is for example possible to read F_b once at the beginning, while for every frame only F_s and the pressure and velocities are read. All large F_s-values can then be displayed, coloured according to pressure or velocity values.
In the case of motion with respect to an inertial reference system, the frames are combined with the right translations or rotations relatively to each other.

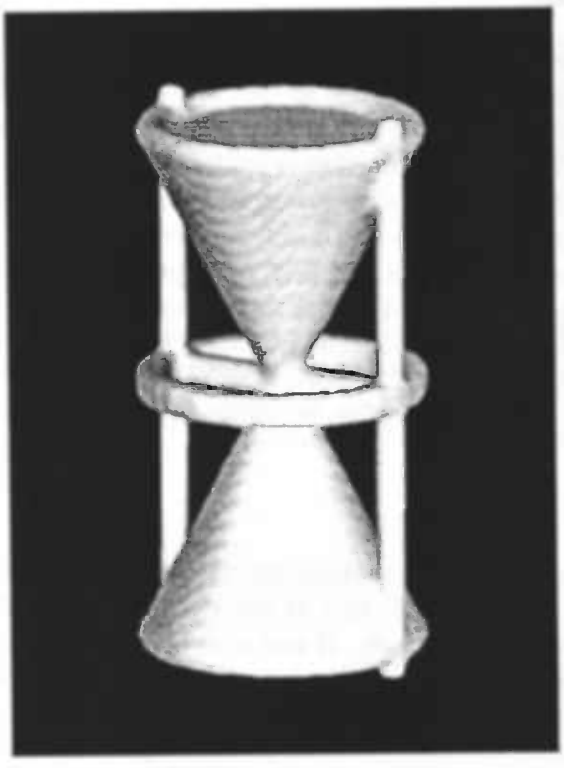


Figure 4.4: *Example of a combined geometry*

4.4.2 Streamlines

Methods to describe the motion of a fluid include following particle tracks: streamlines. Note that for unsteady flows, these streamlines are *not* closed varieties in 3D, thereby forcing the program to take care of this by itself instead of having the work done by an external package. To compute the movement of a particle, ordinary Euler time integration is used:

$$x(t_2) = \int_{t_1}^{t_2} u(\tau)d\tau + x(t_1) \quad \text{or, discrete:} \quad x(T_2) = \sum_{T_1}^{T_2-\delta t} u(t) \cdot \delta t + x(T_1)$$

This involves an accurate computation of the velocity $(u, v, t)^t$ in each time cycle. This is done by interpolation. However, the set of velocities which are to be interpolated is different for each octant the concerned particle is in at that time step, as shown in figure 4.5.

Here the point P is somewhere in the upper(z) left(y) back(x) octant of cell (i, j, k). Then the interpolated velocity in x-direction of a particle in point P is:

$$\begin{aligned} u(t) &= \frac{U_f \cdot h_{x,b} + U_b \cdot h_{x,f}}{\Delta x_i} \\ \text{where} \quad U_f &= \frac{U_{f,d} \cdot h_{z,u} + U_{f,u} \cdot h_{z,d}}{\frac{1}{2}(\Delta z_k + \Delta z_{k+1})} \\ U_b &= \frac{U_{b,d} \cdot h_{z,u} + U_{b,u} \cdot h_{z,d}}{\frac{1}{2}(\Delta z_k + \Delta z_{k+1})} \end{aligned}$$

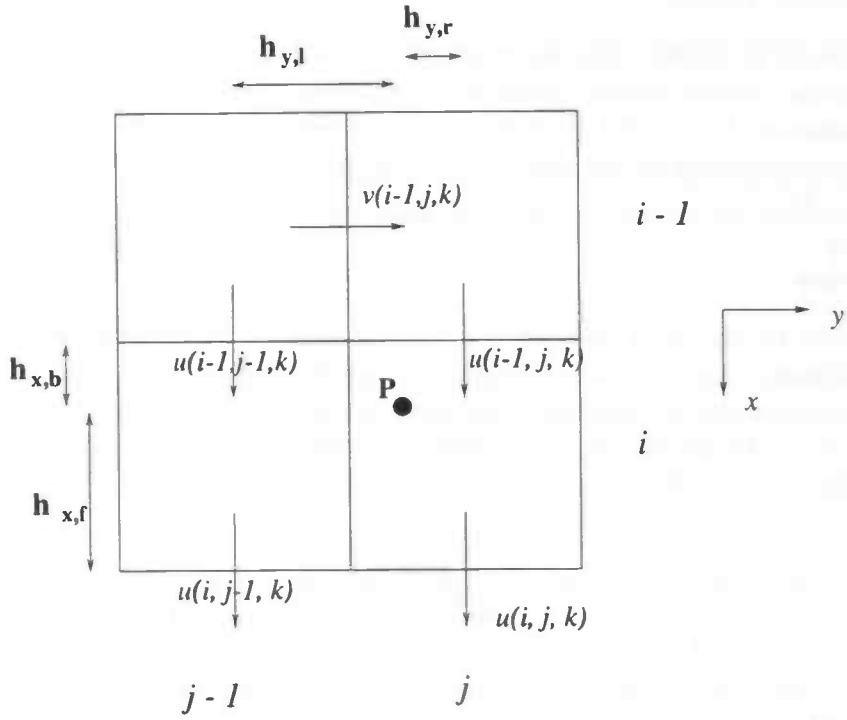


Figure 4.5: *Interpolating u in the x,y plane*

where again

$$
\begin{aligned}
U_{f,d} &= \frac{u(i,j-1,k)\cdot h_{y,r} + u(i,j,k)\cdot h_{y,l}}{\frac{1}{2}(\Delta y_{j-1} + \Delta y_j)} \\
U_{f,u} &= \frac{u(i,j-1,k+1)\cdot h_{y,r} + u(i,j,k+1)\cdot h_{y,l}}{\frac{1}{2}(\Delta y_{j-1} + \Delta y_j)} \\
U_{b,d} &= \frac{u(i-1,j-1,k)\cdot h_{y,r} + u(i-1,j,k)\cdot h_{y,l}}{\frac{1}{2}(\Delta y_{j-1} + \Delta y_j)} \\
U_{b,u} &= \frac{u(i-1,j-1,k+1)\cdot h_{y,r} + u(i-1,j,k+1)\cdot h_{y,l}}{\frac{1}{2}(\Delta y_{j-1} + \Delta y_j)}
\end{aligned}
$$

4.4.3 Fluxes

Fluxes are defined through planes in the geometry. Since the fluxes have already been computed to be used by the donor-acceptor algorithm, implementation of this aspect is quite straightforward.

4.4.4 Filling ratios

In order to get knowledge about the fluid configuration in time, fill boxes are a quantitative alternative to movies. In interesting areas (more exactly, boxes surrounding them), the amount of fluid is recorded. This is also handy to obtain fluid heights at certain points, by taking fill boxes with width and length equal to zero.

4.4.5 Monitor points

Often information in points which are a priory known is needed. Therefore it is possible for the user to define certain points, called monitor points, where at equidistant time intervals physical information is recorded (velocities and pressure; this data is also obtained by interpolation). To make things easier, these points may also be described as grouped on lines and circles.

4.4.6 Forces

Forces resulting by the fluid pushing the walls (called $\boldsymbol{F}^l$ to distinguish them from body and external forces) are often interesting features. Because they are described by pressure times area, it is natural to compute them using the pressure and the apertures. Namely, by resolving the force $\boldsymbol{F}^l$ into $(F_x^l, F_y^l, F_z^l)^T$, they are easily combined with wall apertures A_x, A_y and A_z, as shown in the figure.

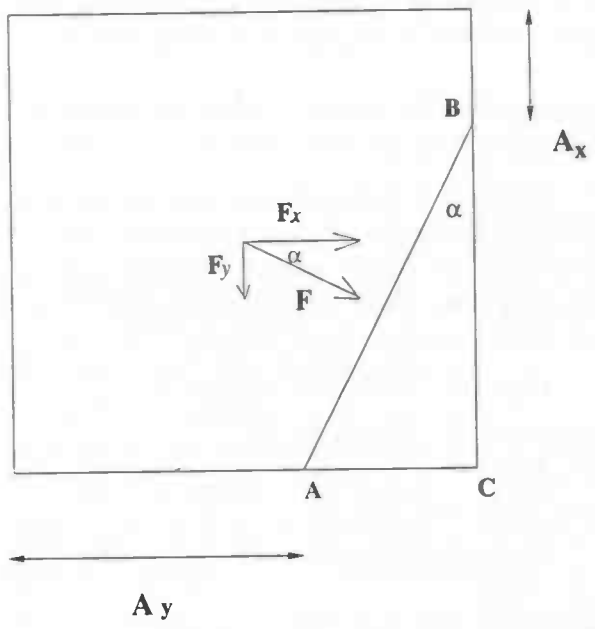


Figure 4.6: *This figure shows in what way the forces are computed*

In this 2D-example, $\boldsymbol{F}^l = (F_x^l, F_y^l)^T$, and the force in the positive x-direction is computed by

$$F_x^l = \boldsymbol{F}^l \cos\alpha = (p \cdot (AB)) \cos\alpha = p \cdot ((AB)\cos\alpha) = p \cdot (BC) = p \cdot (1 - A_x)$$

In this case, the force in negative x-direction is zero, since there is not a wall on the left. The total force in x-direction is obtained by taking the sum over all cells of the force in positive x-direction minus the force in negative x-direction in each cell (using the two x-edge apertures at the left and right of the cell). Note that in- and outflow areas do not endure fluid forces on the geometry.

4.4.7 Other information

Other interesting features include data about numerical aspects and safety measures.

Defragmentation

Often it is interesting to know if an initially connected fluid configuration remains connected or if it breaks down into several smaller parts and drops. The labeling method gives a simple aid to determine this: namely, to look at the number of S-cells at a certain time.

Time step evaluation

Another numerical quantity is the varying value of the time step. Changes in δt due to controlling the CFL-condition are registered. It is found that such data indicates quite directly in which state (wild, quiet) the fluid is moving through the geometry.

Autosaving an restarting

It is unevitable that sometimes something may go wrong during a multiple hour calculation. Therefore an autosave routine has been implemented: at equidistant simulation time intervals, the whole state of the system (that is, the subset of variables at the end of a time cycle that is needed to begin the next cycle) is written to file. After a crash the computation can be continued, starting from the last autosave action. It is also necessary that all secondary files, where postprocessing results are stored, do not remain open during the computation. As this autosaving occurs also automatically after the computation has ended in a regular way (at `T = TMAX`), an interesting consequence is that prolongation after `TMAX` is possible.

Chapter 5

Results

5.1 Dambreak in a ball

To test the postprocessing options together with some numerical aspects, a dambreak in a ball (radius $\frac{1}{2}\,m$) has been simulated. The centre of the ball is at the Cartesian origin. At $t = 0$, the ball is filled with fluid for $x \leq 0$ and $y \leq 0$; the only force is a gravitation in $z-$ direction of $5\,m\,s^{-2}$.
The simulation has been performed for a mesh of N^3 cells, where N was 20, 40 and 80; the latter grid required some 20 hours CPU time on a Cray J90 (effective performance: 40 Mflops).

When finally a stationary condition has been achieved, the pressure at the 'south pole' should satisfy the hydrostatic pressure $p_h = p_0 + \rho g h$, where p_0 is the atmosperhic pressure ($= 1$), g the gravitational force and the density ρ set, as always, on unity. The height h is analytically computed, given that a quarter of the sphere is filled, from below, with liquid:

$$\frac{1}{2} \cdot \frac{4}{3}\pi R^3 = I_{-h,h} = \int_0^{2\pi} \int_0^R \int_{-h}^h \left(1 - \tfrac{z^2}{R^2}\right) \cdot r \, dz \, dr \, d\theta \tag{5.1}$$

$$= 2\pi \cdot \tfrac{1}{2} R^2 \left[z - \tfrac{z^3}{3R^2} \right]_{-h}^{h} \tag{5.2}$$

$$= 2\pi R^2 \left(h - \tfrac{h^3}{3R^2} \right) \tag{5.3}$$

Solving for h with $R = 0.5$ we get $h \approx 0.167$ and thus the height of the fluid is $R - h \approx 0.333$. As shown in figure 5.1, the liquid height above the south pole indeed converges to this value. The pressure at the south pole also satisfies the hydrostatic pressure, but tends to stay under the predicted value for a long time due to the non-steadiness of the flow.

Another postprocessing option tested was the flux computation. To that purpose the filling degree of the ball under the equator, i.e. the bottom half, was measured, together with the flux through the plane $z = 0$. Because the wall is inpenetrable, the following holds:

$$V_e - V_0 = \int_{z=0} Q(t)\, dt$$

where V_0 and V_e are the amount of liquid under the equator at the $t = 0$ and $t = 5$, respectively, and Q the flux.

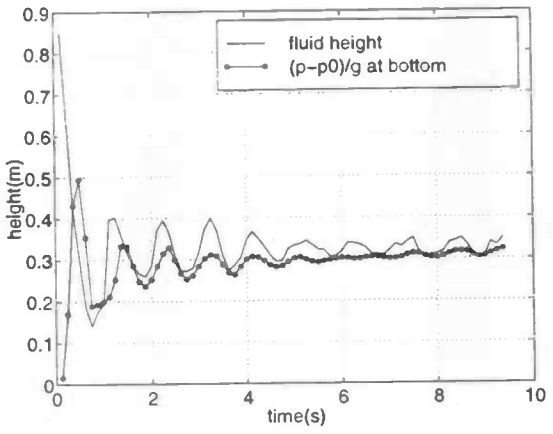


Figure 5.1: *Liquid height and maximum pressure on the vertical NS-axis*

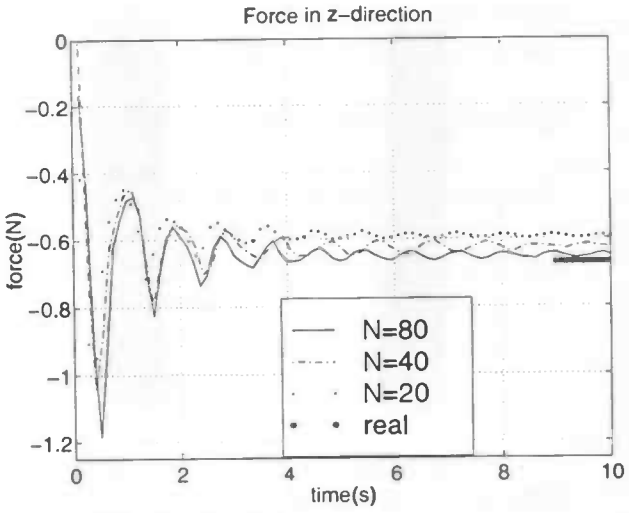


Figure 5.2: *Force in the ball in z-direction*

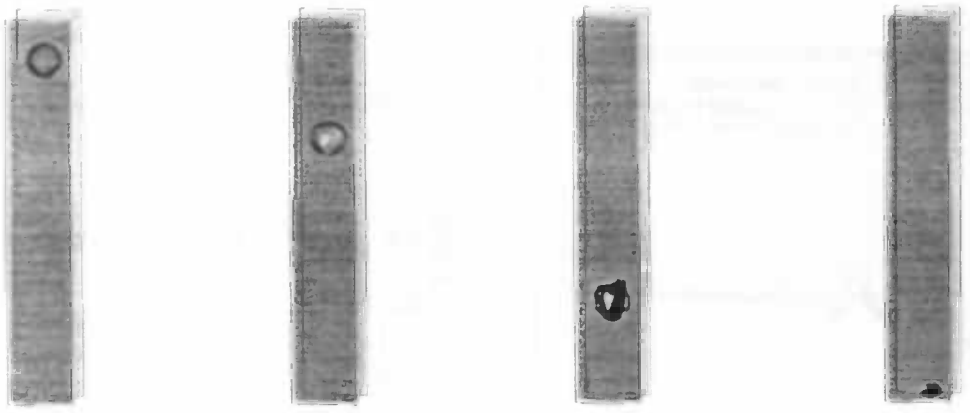
After having integrated the obtained flux numerically between $t = 0\,s$ and $t = 5\,s$, which turned out to be $-0.0622\,m^3$, the fluid difference of the bottom half of the ball was computed, giving $-0.0631\,m^3$. This small difference can be explained by the various rounding actions along the walls and the free surface.
Finally, the force as computed in section 4.4.6 has been measured. In the steady-state situation ($t \to \infty$), the force in z-direction should be: $F_z^l = \int\limits_{x^2+y^2 \leq R^2} g\,h(x,y)\,dx\,dy$ where $h(x,y) = \sqrt{R^2 - x^2 - y^2} + h_0 - R$, and h_0 being the steady-state height as computed above. Taking the right integration limits and in polar coordinates the integral is simply computed as

$$F_z^l = 2\pi g \int\limits_0^{\sqrt{h_0 - h_0^2}} (\sqrt{\frac{1}{4} - r^2} + h_0 - \frac{1}{2}) \cdot r\,dr$$

which is approximately $0.67N$. However, this value is rather dependent of the precise value of h_0 as obtained in the simulation. The graph of figure 5.2 shows the forces for $N = 20, 40$ and 80. Although the oscillations begin to differ for larger values of t, the convergence with respect to the real value is apparent. The difference at $t = 10\,s$ between $N = 20$ and the real value is less than 13% while the difference with respect to $N = 80$ is only 2% .

5.2 Falling drop

In this section we simulate the fall of a small amount of fluid (a drop) in order to test either the outflow characteristics and the force implementation.
At the top of a long, small rectilinear cylinder a drop (in this case a ball) of fluid is positioned at $t = 0$. The bottom of the cylinder consists of an outflow opening. With a normal gravitational force of $g = 9.8\,m\,s^{-2}$, it is to fall through the cylinder, while when reaching the outflow opening, it should still be accelerating until it has totally disappeared through the



$t = 0.03s$ $t = 0.43s$ $t = 0.77s$ $t = 0.93s$

Figure 5.3: *snapshots of the fall of a drop a liquid*

opening.
The formula for the height of the drop is

$$y(t) = y(0) + v(0) \cdot t + \frac{1}{2} \cdot g \cdot t^2 \tag{5.4}$$

where $y(0)$ is the initial height (here $4.0\,m$), $v(0)$ the initial velocity (zero in this case) and t the time.

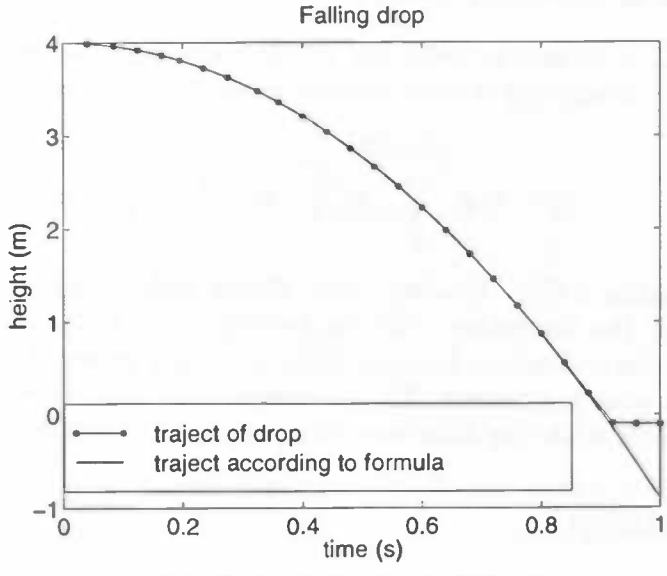
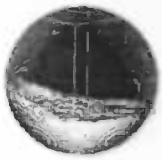


Figure 5.4: *Movement of a particle in the drop.*

The physical sizes are $1\,m \times 1\,m \times 4.5\,m$, while the covering grid consisted of $16 \times 16 \times 90$ cells. The drop itself has a radius of $0.2\,m$. It is of course not necessary to simulate for more than one second.



Particles have been released at several places in the ball; the following figure (5.4) shows the movement of a particle, set in the center of the ball at $t = 0$, together with the position according to formula (5.4).
Because the fluid moves through a lot of computational cells, a slight deformation of the drop occurs.
Due to the sparse fluid distribution, this computation could be done in less than five minutes on a moderate workstation, meaning ten minutes on a Pentium 200 personal computer.

Note that the height of the centre of the drop is not becoming negative due to the fact that the interpolated velocity in the B-cell at the bottom is zero.

5.3 Toricelli

ComFlo is also capable of handling two-dimensional problems; this is done by setting one (inner) dimension to 1, for example the y-dimension (this means that through the whole grid only one F-cell in y-direction is used; the B-cells before and behind have $F_b = 0$) and setting the x-z-walls to free-slip. Indeed, the v velocity is zero, due to the condition $u_n = v = 0$. Furthermore two outer coefficients (out of six) in the left hand side of the pressure Poisson equation are now zero.
Toricelli's theorem, assuming potential flow, states that having a box filled with fluid and an opening at height h under the free surface, the velocity with which the fluid goes out of the box satisfies

$$u = \sqrt{2gh}$$

where g is the gravitational acceleration. Indeed, if no energy is lost, a particle having a potentional energy of $m\,g\,h$ at the top of the fluid column, has this totally converted into kinetic energy $\frac{1}{2}\,m\,v^2$. Therefore all the walls in the box must be set to free-slip to support this total conversion.

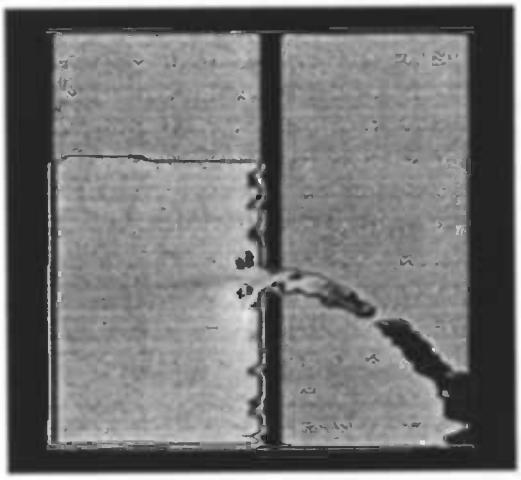


Figure 5.5: *snap shot of Toricelli*

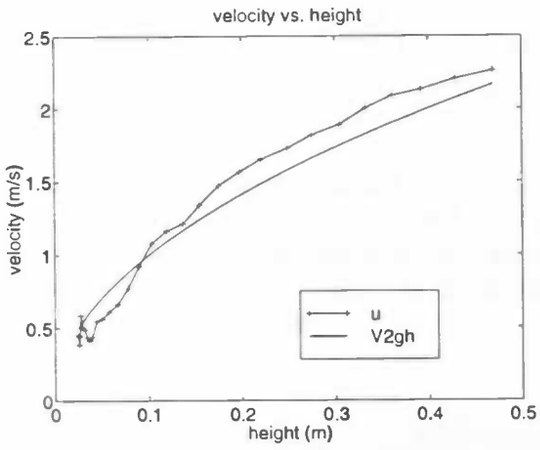


Figure 5.6: *Here the velocity and the height are plotted against each other*

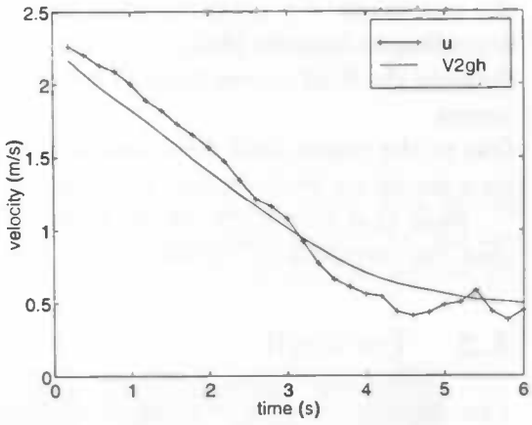


Figure 5.7: *...and here against time.*

The simulation was performed with a grid of 60 x 1 x 60 cells and lasted only a quarter of an hour on a Pentium 200. The bottom of the box on the right part consists of an outflow area (see figure 5.5). The following two figures show the velocity at the opening (measured by a monitor point) versus the height of the fluid (acquired from a fill box around the left part). The differences can be explained by the fact that the numerical model does not use potential theory but the unsteady Navier-Stokes equations instead, which include viscous dissipation and transient effects.

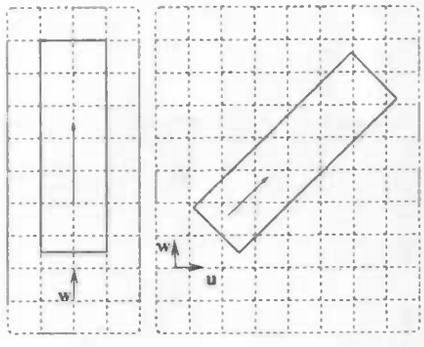


Figure 5.8: *2D-cylinder, normal (left) and rotated (right)*

5.4 In- and outflow in a 2D-cylinder

Another two-dimensional example is the motion of fluid through a cylinder. The profile of the velocity field in the direction of the length of the cylinder will be a parabolical one, according to the well-known Poseuille-Hagen formula:

$$p_1 - p_2 = \frac{12\mu LQ}{\rho h^3} \tag{5.5}$$

where p_1 and p_2 are pressures at the begin and the end of the considered cylinder, L the length, Q the flux through the cylinder (evidently constant), ρ the density of the fluid and h two times the radius of the cylinder.
This formula is tested with the lower seventy percent of the cylinder initially filled with fluid, using an outflow opening at the top and an inflow area (with a constant velocity $w = 1.0$ prescribed) at the bottom. After some time the filling degree of the cylinder reaches 1.0. The grid consisted of $20 \times 1 \times 80$ cells, while the physical sizes where $1\,m \times 9\,m$ (the h above equals 1).

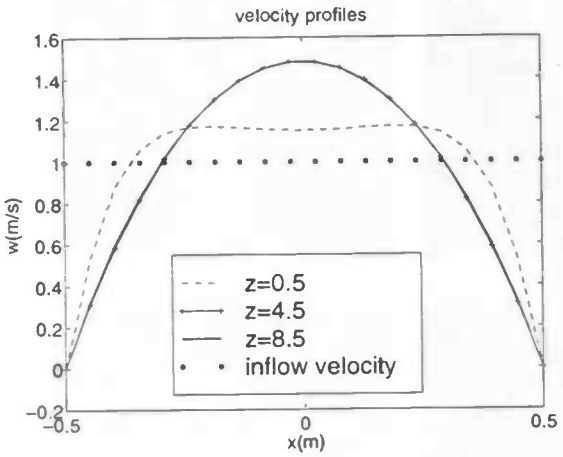


Figure 5.9: *Velocity profiles at different distances*

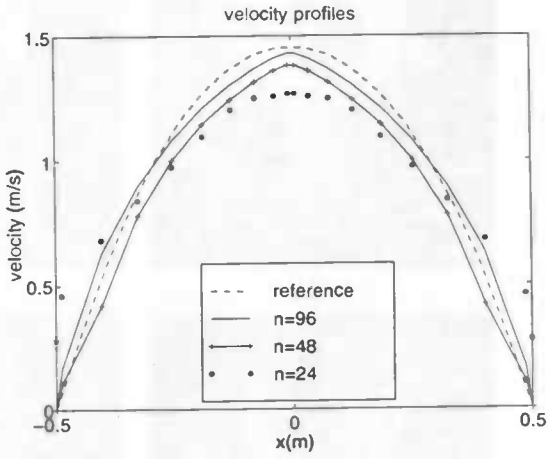


Figure 5.10: *Velocity profiles with rotated grid*

It is seen that the velocity profile soon turns into the desired parabolic one (in figure 5.9 one sees that this may be assumed from $x = 4$). Note that the area under the graph stays the same, indicating a constant flux. Measuring the pressure at two such parabolic profiles at a distance of $L = 4$ (where the profile doesn't change anymore, of course) gives $\Delta p = 0.468$. With Q, h and ρ equal to unity and $\mu = 0.01$, these values clearly satisfy equation (5.5) up to a reasonable degree.

Now the same computation is done on a grid, rotated by $\frac{1}{4}\pi$ (in practice, of course, only the geometry is rotated since the grid is rectilinear). It is clear that the number of cells is increased since the bounding box of Ω is larger. The simulation has been repeated on a grid of $k \times 1 \times k$ cells, $k = 24, 48, 96$. The results of the velocity at the end of the cylinder are shown in figure 5.10, together with the velocity of the first simulation. For the coarser grids, several manipulations had to be carried out to get the inflow velocity right (since the number of inflow cells is difficult to define here and the velocity consists of two nonzero components).

5.5 Demonstrations

Now a few demonstrations follow, giving an indication of *ComFlo's* current capabilities.

5.5.1 Dambreak in ball

An AVS-movie has been made from the dambreak-in-ball computation (with $N = 80$) in section 5.1. The simulation time was $4.5\,s$, while 120 frames have been made. At the top of most of the right pages in this report, a subset of these frames is shown; the frames are equidistant in time.

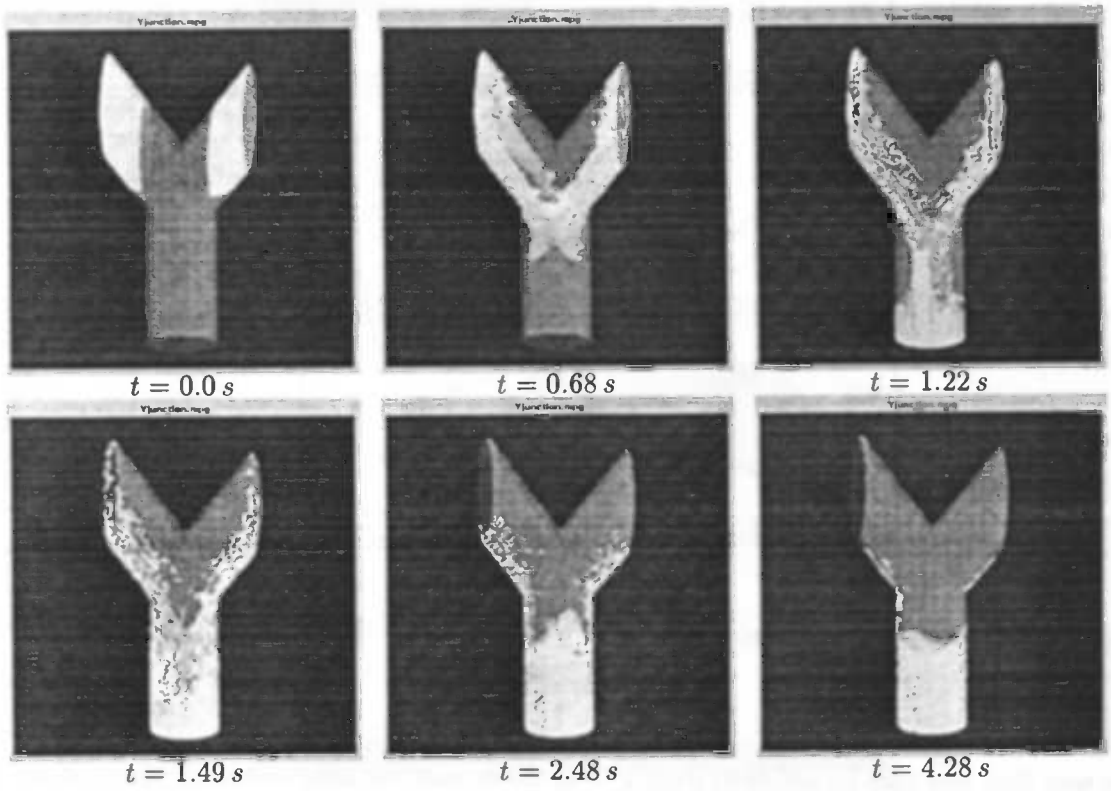
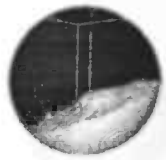


Figure 5.11: *Snapshots of a dambreak in a Y-junction*

5.5.2 Y-junction

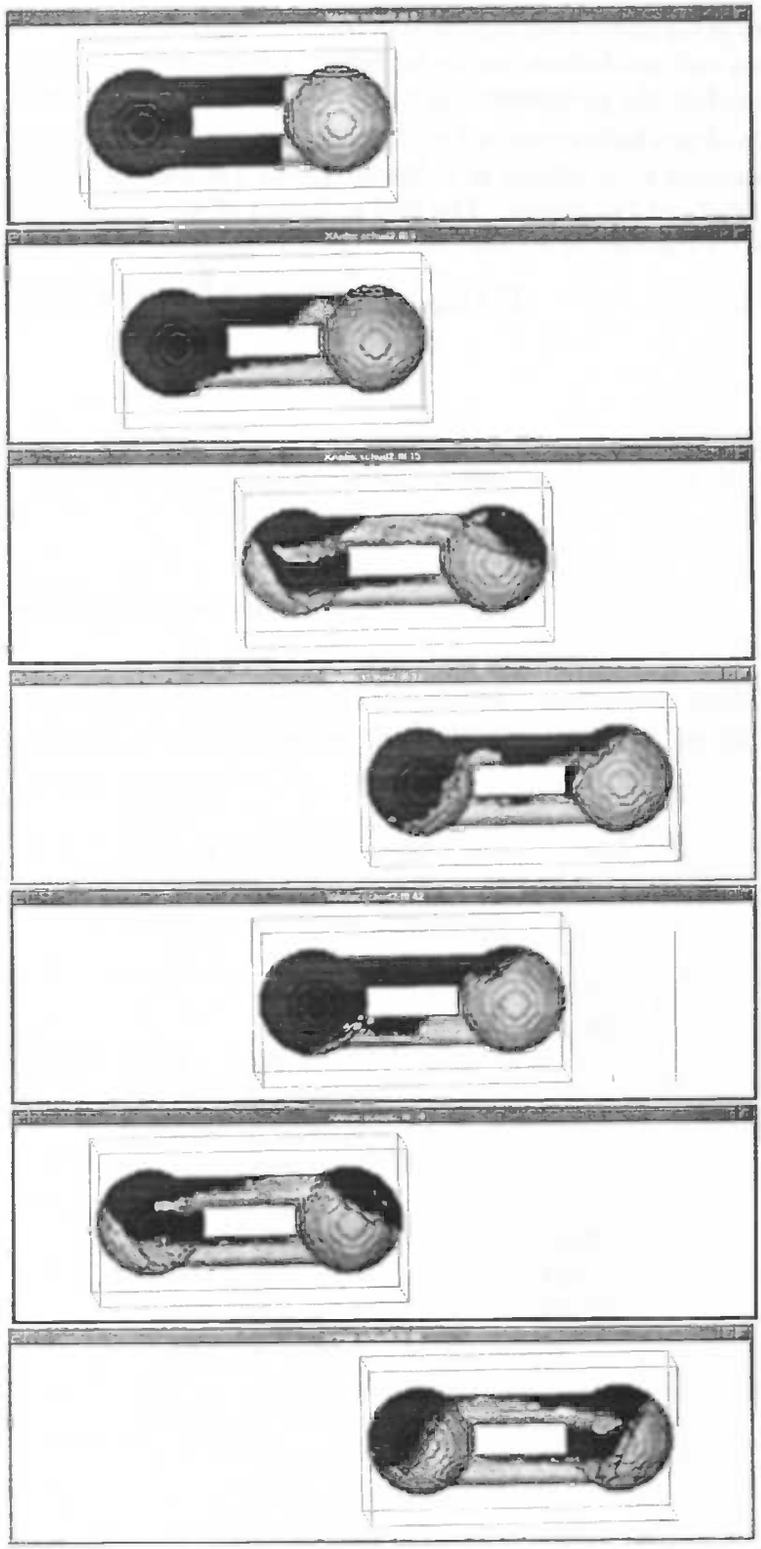
A nontrivial complex geometry is the union of three cylinders forming the character Y (That is the way it was created: a five-node tree with unions in the internal nodes and three cylinders, rotated by $\frac{1}{2}\pi, -\frac{1}{3}\pi$ and $-\frac{2}{3}\pi$ around the y-axis). It can also be regarded as two blood veins combined to form a third.
In the top of the two upper branches an amount of liquid is positioned, kept in place by a dam. At $t = 0$, the dam is removed and the liquid streams downward by a gravitation of $\frac{1}{2}\,g$. In figure 5.11, snapshots of the movie made by AVS give an indication of this simulation.



5.5.3 Moving spheres-cylinders combination

Up to now all examples of liquid sloshing appeared in non-moving geometries. However, using formula (2.5), rotations and oscillations can be included. The movement parameters, as set in the input file, are used by the postprocessing tools in order to visualize those movements.

Here is an example of two balls connected with two cylinders. The gravitation is $5\,m\,s^{-2}$ in z-direction. The geometry is oscillating in x-direction with a frequency of $0.25\,s^{-1}$ and an amplitude twice the length of the object. The grid consisted of $40 \times 40 \times 80$ cells and the computation took five CPU-hours on a Cray J90.


$t = 0.0\,s$
$t = 0.40\,s$
$t = 1.00\,s$
$t = 2.06\,s$
$t = 2.80\,s$
$t = 3.73\,s$
$t = 5.87\,s$

Chapter 6

Conclusions

In the previous chapters we have seen that, using a simple Cartesian grid and a sophisticated labeling system, free-surface flow in 3D-complex geometries can easily be simulated; results show a good accuracy, even with fairly coarse grids.
The adding of in- and outflow, the flow domain describing method using trees and 2D-rotation and several posprocessing possibilities makes the current program suitable for industrial problems, especially when the high performance on supercomputers such as the Cray J90 and the production facilities of movies is considered.

At this stage (summer 1997) still a lot can be done. Extensions for the near future could be:

- Dynamical interaction: the moving fluid in the geometry exerts forces on the geometry wall. This directly influences the motion of the geometry itself, thus causing a feedback process. Examples are spacecraft and tanks in vehicles.
- Diversifying the B-cells: it has been noticed that boundary cells near a free surface need another treatment than 'normal' boundary cells.
- Local grid refinement, making it possible to pay attention to interesting parts of the flow domain, in first instance near the boundary of Ω.
- Further enhancement of pre- and postprocessing features. Although at this stage almost all forms of geometries can be drawn, a WYSIWYG-system by constructing geometries is desireable (an import function for data out of drawing packages would also be handy). Another never ending goal is to further enhance the 'backside' of the program and supply highly sophisticated graphical output.
- Heat transfer, further increasing the industrial applications.
- More robust implementation of, and diversifying the possibilities with virtual body forces.

Appendix A

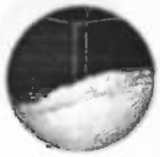
Program description

The numerical model has been implemented in a program called *ComFlo*. This appendix gives more detailed information about the calling sequence and the several variables and subroutines.

A.1 Calling sequence

The following scheme indicates the order in which the various subroutines are called. Routines in italics are not completely necessary or represent a compulsary choice out of more possibilities; they can be controlled by setting the right parameters in the input file **Comflo.in**.

Initialization	SETPAR		
	GRID		
	BNDLAB	*BNDDEF / GETGEO*	
		IOLAB	
	SETFLD	*AUTOSV / SURDEF*	
		SURLAB	
		BC	IOBC
			VELBC
begin LOOP	INIT	IOBC	
	TILDE	BDYFRC	
	SOLVEP	COEFL	
		COEFR	
		PRESBC	
		PRESIT	SLAG
		BC	IOBC
			VELBC
	VFCONV	SURLAB	



CFLCHK
PRNT
MATLAB
FILLBX
FLXOUT
FRCBOX
MPTOUT INTERP
STREAM INTERP
AUTOSV

end LOOP

In the LOOP sequence, the time integration is executed; the loop continues while the maximum simulation time has not been reached.
The named subroutines are described in section A.3.

A.2 Common block variables

The globally used variables are defined within the *common blocks* structures in Fortran. All but a few uninteresting are listed here, with short descriptions.

- **/ADAMS/**
  Extra variables used for Adams-Bashforth time integration.
  `UNN(I,J,K)`, `VNN(I,J,K)`, `WNN(I,J,K)` These are the old velocities in cell (i,j,k).
  `COEF1,COEF2` These coefficients describe in what way the velocities at the 'intermediate' time level are defined: `WN = COEF1 W + COEF2 WNN`.
  The combination (`COEF1,COEF2`) $= (1,0)$ defines forward Euler, the combination $(1.5,-0.5)$ Adams-Bashforth.

- **/APERT/**
  The volume and edge apertures.
  `AX(I,J,K)` Edge aperture between cell (i,j,k) and cell $(i+1,j,k)$
  `AY(I,J,K)` Edge aperture between cell (i,j,k) and cell $(i,j+1,k)$
  `AZ(I,J,K)` Edge aperture between cell (i,j,k) and cell $(i,j,k+1)$
  `FB(I,J,K)` Volume aperture w.r.t. the geometry of cell (i,j,k).
  `FS(I,J,K)`, `FSN(I,J,K)` Volume apertures w.r.t. the free surface of cell (i,j,k), at new and old time level, respectively.

- **/COEFP/**
  The coefficients in the pressure Poisson equation.
  `DIV(I,J,K)` The right hand side of the equation in cell (i,j,k)
  `CC(I,J,K)` Coefficient of $p_{i,j,k}$ in left hand side.
  `CXL(I,J,K)`,`CXR(I,J,K)` Coefficients of $p_{i\pm1,j,k}$ in left hand side.
  `CYL(I,J,K)`,`CYR(I,J,K)` Coefficients of $p_{i,j\pm1,k}$ in left hand side.
  `CZL(I,J,K)`,`CZR(I,J,K)` Coefficients of $p_{i,j,k\pm1}$ in left hand side.

- **/FLUID/**
  Characteristics of the fluid.
  NU Kinematic viscosity number (ν)
  SIGMA Surface tension parameter
  THETA Contact angle RHO Density

- **/FORCE/**
  Characteristics of the external and body forces.
  AMPL, FREQ Amplitude and frequency of the oscillation used in the subroutine **BDYFRC**
  GRAV(i) Gravitational acceleration in direction i, $i = 1, 3$.

- **/GRIDAR/**
  Positioning and distances of the cartesian grid.
  IMAX, JMAX, KMAX Number of cells in the three directions
  X(I), Y(J), Z(K) Coordinates of grid lines; cell (i, j, k) lies between
  $x(i-1), x(i), y(j-1), y(j), z(k-1)$ and $z(k)$.
  DXP(I),DYP(J),DZP(K) Sizes of each cell: DXP(I)=X(I)-X(I-1)
  DXU(I),DYV(J),DZW(K) Distances between cell centres:
  DXU(I) = (DXP(I) + DXP(I+1))*0.5

- **/LABELS/**
  Cell and velocity labeling.
  PLABEL(I,J,K) Cell label based on the geometry only.
  PLABFS(I,J,K), PLABFSN(I,J,K) Cell label based on the free surface, at new and old time levels, respectively.
  ULABEL(I,J,K), ULABFS(I,J,K) Velocity labels for u based on the geometry and free surface, respectively, between cells (i, j, k) and $(i+1, j, k)$.
  VLABEL(I,J,K), VLABFS(I,J,K) Velocity labels for v based on the geometry and free surface, respectively, between cells (i, j, k) and $(i, j+1, k)$.
  WLABEL(I,J,K), WLABFS(I,J,K) Velocity labels for w based on the geometry and free surface, respectively, between cells (i, j, k) and $(i, j, k+1)$.

- **/NUMER/**
  Numerical parameters.
  EPS Allowed error in pressure Poisson SOR-iteration process.
  OMSTART Relaxation factor in SOR-iteration process at $t = 0$
  OMEGA Relaxation factor (adjusted to improve convergence).
  ITER, ITMAX Amount of SOR-iterations in a time cycle; divergence occurs when ITER exceeds ITMAX.
  NOM Counter for amount of time steps with constant OMEGA.
  ITTOT Cumulative number of iterations.
  ALPHA Upwind parameter: ALPHA= 1: full upwind .

- **/PHYS/**
  These are the variables characterizing the physics of the system.
  `U(I,J,K)`,`UN(I,J,K)` Velocity u between cells (i,j,k) and $(i+1,j,k)$ at new and old time level, respectively.
  `V(I,J,K)`,`VN(I,J,K)` Velocity v between cells (i,j,k) and $(i,j+1,k)$ at new and old time level, respectively.
  `W(I,J,K)`,`WN(I,J,K)` Velocity w between cells (i,j,k) and $(i,j,k+1)$ at new and old time level, respectively.
  `P(I,J,K)` Pressure p in cell (i,j,k) at new time level.

- **/PLOTS/**
  Specification parameters for the data being written to file to produce primary flow information: plots and movies.
  `MATLDT`, `MATLNR` Every `MATLDT` simulation seconds data for *Matlab* plots is written to file, where `MATLNR` is the amount of times this has been done sofar.
  `AVSDT`, `AVSNR` Every `AVSDT` simulation seconds data for *Matlab* plots is written to file, where `AVSNR` is the amount of times this has been done sofar.

- **/PRT/**
  Specification parameters for numerical data written to screen and files.
  `PRTDT`, `PRTNR` Every `PRTDT` simulation seconds numerical data is written to screen and files, where `PRTNR` is the amount of times this has been done sofar.

- **/SAFETY/**
  Autosave parameters.
  `AUTONR`, `AUTODT` Every `AUTODT` simulation seconds all simulation information is written to file, where `AUTONR` is the amount of times this has been done sofar.
  `RESTART` Check box indicating if the autosave file has to be used instead of the regular initialization actions.

- **/SPACE/**
  Information about the geometry size and kind of flow domain.
  `DOMAIN` Type of domain the fluid is set in (cube, cylinder, etc. or a user-specified domain)
  `XMIN`, `XMAX` Mesh size in x-direction: `XMIN=X(0)`, `XMAX=X(IMAX)`
  `YMIN`, `YMAX` Mesh size in y-direction: `YMIN=Y(0)`, `YMAX=Y(IMAX)`
  `ZMIN`, `ZMAX` Mesh size in z-direction: `ZMIN=Z(0)`, `ZMAX=Z(IMAX)`

- **/TIME/**
  Variables about the discrete time.
  `T`, `TMAX` Current and maximum simulation time, respectively.
  `DT` Current time step δt.
  `CYCLE` Current time cycle.

- **/TSTEP/**
  Parameters specifying the variable time steps.
  CFLON Parameter indicating whether variable time steps are used
  CFL Current CFL-number
  CFLMIN Low reference CFL-number: if CFL stays under CFLMIN, DT can be doubled.
  CFLMAX High reference CFL-number: if CFL jumps above CFLMAX, DT must be halved.
  SMALLCFL Counter for number of cycles with low CFL-numbers
  NRDTCHANGE Counter for amount of times the timestep has been halved or doubled so far.

A.3 Subroutines

- **AUTOSV**

  In: T, DT,CYCLE,AUTODT, AUTONR,U,V,W,P,FS and all other necessary variables.
  Out: File **autosav.dat**
  Description: At equal time intervals (every AUTODT simulation seconds), the whole state of the computation is saved, which can be restarted after a crash. Moreover, since this routine is called after the regular computation, it can be used to continue after TMAX.

- **AVS**

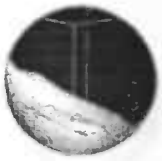
  In: FB, FS, IMAX, JMAX, KMAX,U, V, W, P, AVSDT, AVSNR, T, DT
  Out: Files **comflo###.fld** and **comflo###.dat**
  Description: At equal time intervals (after every AVSDT seconds), files for visualizing (3D) data in AVS are created.

- **BC**

  In: AX, AY, AZ, FB, IMAX, JMAX, KMAX,
  DXP, DYP, DZP, U, ULABEL,V, VLABEL, W, WLABEL
  Out: U, V, W
  Description: The four solid boundary velocities, namely **BB,FB,SB** and **EB** are computed using the apertures and the boundary conditions (no-slip or free-slip). The velocity labels define the precise computation of these velocities. Finally, the routines **IOBC** and **VELBC** are called to determine the free surface and in- and outflow velocities as well (see sections 3.3.2 and 3.3.4).
  Note that boundary velocities use internal velocities from the previous time step.

- **BDYFRC**

  In: AMPL, FREQ, DOMAIN, GRAV, T
  Out: DQDT, OMET, DOMEDT, GRAV



Description: This routine is called from **TILDE** in order to determine the virtual body force, described by $\frac{\partial q}{\partial t}$ (DQDT), ω (OMET) and $\frac{\partial \omega}{\partial t}$ (DOMEDT). It is also possible to prescribe or adjust the external body force $\boldsymbol{F}$ (GRAV).

- **BNDDEF**

In: DOMAIN, ILOC, JLOC, KLOC, NRINTP
Out: AX, AY, AZ, FB
Description: For each cell this subroutine is called from **BNDLAB** to compute the apertures in the case that DOMAIN $\neq 0$, i.e. if only a standard geometry is needed. The cell is divided into (NRINTP)^3 points (where each cell wall gets (NRINPT)^2 points) and the fraction of the points in the cell or cell wall that belong to the interior of the geometry is exactly the value of the volume and edge apertures, respectively.

- **BNDLAB**

In: DOMAIN, IMAX, JMAX, KMAX
Out: AX, AY, AZ, PLABEL, ULABEL, VLABEL, WLABEL
Description: First, either **GETGEO** or **BNDDEF** is called to obtain the apertures. Second, using these apertures, the cells are labeled **F**,**B** and **X**. Third, the geometry-based velocity labels are set. Fourth, these labels are adjusted to involve free slip and in- and outflow velocity labels.

- **CFLCHK**

In: CFLMIN, CFLMAX, DT, U, V, W, DXU, DYV, DZW
Out: DT
Description: Each time step the CFL-value (see section 3.4 is computed. If this value is small enough ($\leq$ CFLMIN) a few consecutive cycles, the time step is doubled. On the other hand, if it is larger than CFLMAX, the time step is immediately halved.

- **COEFL**

In: IMAX, JMAX, KMAX, DXP, DYP, DZP, DXU, DYV, DZW,
PLABFS, ULABFS, VLABFS, WLABFS
Out: CC, CXL, CXR, CYL, CYR, CZL, CZR
Description: The coefficients (namely one central and six neighbour coefficients (city block distanced)) of the matrix on the lefthand side of the Poisson equation are computed in **F**-cells every time cycle. In other cells the coefficients are set in a different way: see section 3.3.3.

- **COEFR**

  In: IMAX, JMAX, KMAX, DXP, DYP, DZP, PLABFS, U, V, W, DT
  Out: DIV
  Description: The righthand side of the Poisson equation in **F**-cells is computed every time cycle. In other cells the righthand side is computed in a different way: see section 3.3.3.

- **FILLBX**

  In: DXP, DYP, DZP, FB, FS, NRBOXES, FILLDT,FILLNR, FCOORD1, FCOORD2
  Out: files **fill###.dat**
  Description: This subroutine produces during the computation every FILLDT seconds information about the filling degree of boxes specified with FCOORD1 and FCOORD2.

- **FLXOUT**

  In: DXP,DYP,DZP, XFLUX,YFLUX,ZFLUX,NRFLUXES,FLUXDT,FLUXNR,FLCO1,FLCO2
  Out: files **flux###.dat**
  Description: Here the value of the fluxes through planes specified with FLCO1 and FLCO2 is written to file every FLUXDT seconds.

- **FRCBOX**

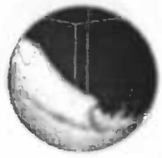
  In: DXP, DYP, DZP,FB,FS,P, NRFBXS,AX,AY,AZ, FCO1, FCO2, FRCNR, FRCDT
  Out: files **force###.dat**
  Description: Here the forces on the wall in boxes specified with FCO1 and FCO2 are written to file every FRCDT simulation seconds.

- **GETGEO**

  In: IMAX, JMAX, KMAX, files **geodata.dat** and **liqdata.dat**
  Out: AX, AY, AZ, FB, FS
  Description: If DOMAIN = 0, this routine is called by **BNDLAB** to obtain the geometry and initial liquid configuration from files made by the preprocessing programs.

- **GRID**

  In: IMAX, JMAX, KMAX
  Out: X, Y, Z, DXP, DYP, DZP, DXU, DYV, DZW
  Description: A uniform grid is created.



- **INIT**

  In: COEF1, COEF2, CYCLE, FS, IMAX, JMAX, KMAX, U, V, W
  Out: CYCLE, FSN, U,UN,UNN, V,VN,VNN, W,WN,WNN
  Description: This routine begins a new time cycle. Here the cycle is increased by one, U,V,W are saved in UNN,VNN,WNN and reset; and UN,VN and WN are computed according to the time integration coefficients COEF1 and COEF2. Finally FS is copied to FSN.

  INTERP

  In: I,J,K, XPT, YPT, ZPT, DXP,DYP,DZP,DXU,DYV,DZW
  Out: PI, UI, VI, WI
  Description: The pressure and velocities are computed at coordinates (XPT,YPT,ZPT) in cell (i, j, k). This is done by interpolating neighbouring values using the location of the point in the cell.

- **IOBC**

  In: IMAX,JMAX,KMAX, ULABEL,VLABEL,WLABEL
  Out: P, U, V, W
  Description: Boundary conditions with respect to in- and outflow cells are set.

- **IOLAB**

  In: IMAX,JMAX,KMAX,PLABEL
  Out: PLABEL
  Description: In- and outflow cells are labeled by changing boundary cells, i.e. cells with PLABEL=2.

- **LIQPCT**

  In: DXP, DYP, DZP, IMAX,JMAX,KMAX, FB, FS
  Out: LIQUID, VOLUME
  Description: Computes the amount of liquid and writes the liquid percentage to screen.

- **MATLAB**

  In: FB, FS, IMAX,JMAX,KMAX, X,Y,Z, U,V,W,P, MATLNR,MATLDT, T, DT
  Out: files **comflo###.m** and file **coord.m**
  Description: Every MATLDT seconds files are created to visualize the geometry, the free surface, the velocity field and the pressure in certain planes in *Matlab*.

- **MPTOUT**

  In: U,V,W,P,NRMPTS,NRLINES,MPDT,MPNR,FCOOR1,FCOOR2, files **mpoints.dat**
  Out: files **speed.###.dat** and **lines###.dat**
  Description: in **mpoints.dat**, monitor points and monitor lines are defined. This routine writes the pressure and velocities on those positions, as computed by **INTERP** to the specified files every MPDT simulation seconds.

- **PRESBC**

  In: IMAX,JMAX,KMAX, PLABEL, PLABFS, SIGMA, THETA
  Out: P,CXL,CXR,CYL,CYR,CZL,CZR,CC
  Description: Here the free surface condition is applied to **S**-cells; See section 3.3.2 for further information.
  At the end the coefficients of the pressure Poisson equation are scaled by the central coefficient.

- **PRESIT**

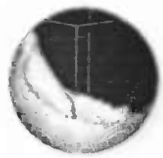
  In: IMAX,JMAX,KMAX, PLABFS, PLABFSN, EPS, OMSTRT, OMEGA,
  CYCLE, ITER, ITMAX, NOM
  Out: P
  Description: This routine solves the pressure Poisson equation using SOR. Starting with OMSTRT, the relaxation parameter ω (OMEGA) is, when possible, changed to obtain the highest convergence ratio. The routine **SLAG** is called for every individual SOR-sweep while a certain error (DELTA) exceeds EPS and the number of iterations ITER stays less than ITMAX. If the second guard is not longer the case, the program terminates because of an apparent non-convergence.

- **PRNT**

  In: IMAX,JMAX,KMAX, U,UN,V,VN,W,WN,PLABEL, PLABFS,PRTDT,PRTNR, T,DT
  Out: screen information, file **comflo.out**, file **nriter.dat**, file **defrag.dat**
  Description: This subroutine handles the more general output to screen and files, which is done every PRTDT seconds. First, information like time, iterations and changes in velocities and pressure is printed. The file **nriter.dat** shows the relation between timesteps and total amount of SOR-iterations versus the simulation time. The file **defrag.dat** gives an indication of the defragmentation (breaking up) of the fluid.

- **SETFLD**

  In: IMAX,JMAX,KMAX, RESTART
  Out: U,V,W, P
  Description: Here the state of the fluid is initialized. If RESTART equals one (in case



of a restart or a continuation) the routine **AUTOSV** is called to obtain the necessary information and a call to **SURLAB** sets the labeling. At this point all data to continue has been acquired.
Otherwise, in an ordinary startup, **SURDEF** and **SURLAB** are called to set the initial liquid configuration and labels. Also the velocities are initialized (usually to zero) and the atmospheric pressure is defined in the whole grid. At last **BC** is called to obtain the initial boundary conditions.

- **SETPAR**

  In: file **comflo.in**, file **mpoints.dat**
  Out: XMIN,XMAX,YMIN,YMAX,ZMIN,ZMAX,IMAX,JMAX,KMAX,NRINTP,TMAX,DT,
  CYCLE, SIGMA,THETA, NU,DOMAIN,AMPL,FREQ,GRAV,ALPHA,EPS,OMSTRT,
  OMEGA,ITMAX,NOM,CFLON,CFLMIN,CFLMAX,NRPRNT,PLOT23,NPMATL,
  MATLDT,NPMATH,MATHDT,PRTNR,PRTDT
  and all other postprocessing information (more than 50 variables)
  Description: All variables given above are read or initialized. Moreover, all information about the postprocessing (frequence of writing, location of boxes, planes, lines and points) is read. For more information see the input files.

- **SLAG**

  In:IMAX,JMAX,KMAX,DIV,CXL,CXR,CYL,CYR,CZL,CZR,ITER,OMEGA,PLABFS,P
  DELTA,ITER,P
  Out: DELTA, ITER, P
  Description: Here one SOR-sweep is executed using a red-black ordening. The relaxation factor is OMEGA. An adjusted pressure field P is returned, together with the norm DELTA of the difference with the previous P.

- **SOLVEP**

  In: IMAX,JMAX,KMAX,ULABFS,VLABFS,WLABFS,U,V,W,P,DT,DXU,DYV,DZW
  Out: U,V,W,P
  Description: This is the main routine to solve the pressure and obtain new velocities as described in chapter 3. First, **COEFL** and **COEFR** are called; then the pressure Poisson equation is solved by calling **PRESBC** and **PRESIT**. The new momentum velocities are computed according to formula (3.7); **BC** is called to adjust the boundary velocities.

- **STREAM**

  In:IMAX,JMAX,KMAX,NRPART,STRMDT,STRMNR,XPART,YPART,ZPART,DT
  Out: files **partic###.dat**

Description: This routine produces files containing streamlines. Every cycle, the velocity of each particle is computed by **INTERP** and the position of the particle is computed using forward Euler. The files are adjusted every STRMDT simulation seconds.

- **SURDEF**

In: DOMAIN,FB, IMAX,JMAX,KMAX,X,Y,Z
Out: FS
Description: The initial fluid configuration, described by FS, is defined.

- **SURLAB**

In: IMAX,JMAX,KMAX, PLABEL, PLABFS, FS, ULABEL,VLABEL,WLABEL
Out: ULABFS, VLABFS, WLABFS, PLABFS, PLABFSN
Description: Here the free-surface labels (**F**, **S**, and **E** and the various velocity labels) are set at every time step (since the fluid configuration is time-dependent).

- **TILDE**

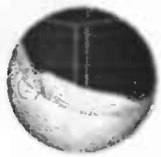
In: IMAX,JMAX,KMAX, X,DXP,DXU,Y,DYP,DYV,Z,DZP,DZW,
GRAV,DQDT,DOMETDT,OMET,DT,UN,VN,WN,ULABFS,VLABFS,WLABFS
Out: U,V,W
Description: Here the term $u^n + \delta t \boldsymbol{R}^n$ as described in section 3.3.1 is computed. Since $\boldsymbol{R}^n$ contains all internal, external and body forces, the routine **BDYFRC** is called for each grid point.

- **VELBC**

In: IMAX,JMAX,KMAX,FS, DXP,DXU,DYP,DYV,DZP,DZW,U,V,W,
ULABFS,VLABFS,WLABFS
Out: U,V,W
Description: In this routine the free-surface velocities **EE** and **SE** are computed using the free-surface labels and the discretized free-surface boundary conditions.

- **VFCONV**

In: IMAX,JMAX,KMAX, FB,FS,FSN,AX,AY,AZ,DXP,DYP,DZP,
DT,PLABEL,PLABFS,ULABFS,VLABFS,WLABFS,U,V,W
Out: FS, XFLUX,YFLUX,ZFLUX
Description: Here the donor-acceptor algorithm is performed. First the fluxes between **F**-, **S**- and **E**-cells are computed. In these cells the values of F_s are recomputed using those fluxes. Since after that the labeling is obsolete, SURLAB is called to set the new labels.



Appendix B

Pre- and postprocessing

To initialize the geometry and liquid configuration, a program called *GeoMake*, respectively *LiqMake* has been written. In the following sections we will discuss the the main calling sequence, the routines of this program, the necessary input files and postprocessing possibilities.

B.1 Main calling sequence

To run a complete computation, the sequence is:

```
geomake in2D_1.txt intree_1.txt

liqmake in2D_2.txt intree_2.txt

comflo
```

It is noted that the input file **comflo.in**, which is not stated here, is used by all three programs.

B.2 Subroutines

Since *GeoMake* and *LiqMake* are mostly the same, we will only discuss the former one. Because FORTRAN does not support recursive functions (which are used to descend and ascend the binary tree) the program was written in ANSI-Pascal.

- **MAIN**
  readin; makegrid; init;
  arcsscan ; linesscan;
  scangeo ;
  writegeo

- **procedure arcsscan**
  In: all parameters of all arcs
  Out: A subset of the x-z domain indicating which regions w.r.t. the arcs are inside the geometry.

- procedure **init**
  In: ANGLE,IMAX,JMAX,KMAX
  Out: FB,AX,AY,AZ,MAT,XCEN,YCEN
  Initializes the apertures, the 2D-description matrix and the rotation center (the latter is determined by ANGLE, which indicates an entire, a half, or a quart rotation).

- function **inner**
  In: tree
  Out: TRUE/FALSE
  The actual tree-descending function. Called by **placecell** with the root of the tree, it recursively calls itself at every branch, while at the leaves it calls the several functions that handle the primitives.

- function **inrot**
  In:XPT,YPT,ZPT
  Out: TRUE/FALSE
  This function, called from **placecell**, determines if
  the point(XPT,YPT,ZPT) belongs to the rotated 2D-geometry.

- procedure **linesscan**
  In: all parameters of all lines
  Out: A subset of the x-z domain indicating which regions w.r.t. the lines are inside the geometry.

- procedure **makegrid**
  In: IMAX,JMAX,KMAX
  Out: X,Y,Z, DXP,DYP,DZP,DXU,DYV,DZW
  This is the analogon of the subroutine **GRID** in *ComFlo*.

- procedure **placecell**
  In: I,J,K,NRINTP
  Out: FB,AX,AY,AZ
  Here the apertures for each cell are returned. First a number of points in the cell (influenced by NRINTP) are defined, whereafter for each point **inner** and **inrot** are called. According to the number of times when TRUE is returned, the apertures are computed.

- procedure **readin**
  In: files **comflo.in**, intree_1.txt and in2D_1.txt
  Out: IMAX,JMAX,KMAX,NRINTP,LEFT,ANGLE, tree, lines, arcs
  Handles reading in the user- defined input files containing domain information, the tree and the 2D-description.

- procedure **scangeo**
  In: tree, `IMAX,JMAX,KMAX,X,Y,Z`
  Out: TRUE
  Here the routine **placecell** is called for every cell, taking the domain of the tree into account.

- procedure **writegeo**
  In: `FB,AX,AY,AZ`
  Out: files **geodata.dat**, **fbdata.fld** and **fbdata.dat**
  Writes apertures to file to be used by *ComFlo*, while **fbdata.fld** and **fbdata.dat** are created to view the geometry in AVS.

B.3 Files

ComFlo needs in essence only one file, **comflo.in**, but can produce hundreds of output files, though many of them have the same characteristics. Furthermore, *GeoMake* also needs two input files. Here is a description of them:

B.3.1 Input files

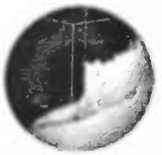
comflo.in

The file comflo.in is the main input file. An example:

```
-----------------------------------------------------------------------
dim     cray
3       0
-----------------------------------------------------------------------
domain  par1    par2    par3    par4    par5    par6    slip
4       -0.3648 0.3648  -0.2508 0.2508  -0.2508 0.2508  0
-----------------------------------------------------------------------
object  par1    par2    par3    par4    par5    par6    slip
0       0.0     0.0     0.0     0.0     0.0     0.0     0
-----------------------------------------------------------------------
liquid  par1    par2    par3    par4    par5    par6
0       0.0     0.0     0.0     0.0     0.0     0.0
-----------------------------------------------------------------------
rho     nu      sigma   theta
1.0     1.0E-6  7.3E-5  0.0
-----------------------------------------------------------------------
imax    jmax    kmax    sx      sy      sz      xc      yc      zc
64      44      44      1.0     1.0     1.0     0.0     0.0     0.0
-----------------------------------------------------------------------
eps     omega   itmax   alpha   orde4   feab1   feab2   nrintp  exact
1.0E-5  1.3     10000   1.0     0       0.0     0.0     1       0
```

```
-------------------------------------------------------------------------
dt      tmax    cfl     cflmin  cflmax
0.001   0.001   1       0.1     0.5
-------------------------------------------------------------------------
gravx   gravy   gravz
0.0     0.0     0.0
-------------------------------------------------------------------------
amplx   freqx   amply   freqy   amplz   freqz   u0      v0      w0
0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
-------------------------------------------------------------------------
omex    omey    omez    tup     tdown   x0      y0      z0
0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
-------------------------------------------------------------------------
load    nsave
0       0
-------------------------------------------------------------------------
npavs   tbavs   comavs  npmatl  tbmatl  nprnt
0       0.0     0       1       0.0     100
-------------------------------------------------------------------------
mpt     ntmpt
0       0
-------------------------------------------------------------------------
nfill   ntfill
0       0
x1      x2      y1      y2      z1      z2
-------------------------------------------------------------------------
nfrc    ntfrc
0       0
x1      x2      y1      y2      z1      z2
-------------------------------------------------------------------------
npart   ntpart
0       0
xstrt   ystrt   zstrt   tstrt
-------------------------------------------------------------------------
nflux   ntflux
0       0
x1      x2      y1      y2      z1      z2
-------------------------------------------------------------------------
```

A more detailed description is found at the end of the file itself.



in2D.txt

An example of the file in which the initially 2D-object (before rotating) is described, follows.

```
angle   left
1       0
narcs nlines
2 1
arcs:
xm        zm        r         t1        t2        side
0.45     0.9       0.05       0.0      6.29       1
0.45     -0.9      0.05       0.0      6.29       1
lines:
x1        z1        x2        z2        side
0.4      -0.05      0.5       0.05       3
```

`Angle` gives the rotation angle: 1 means 2π, 2 means π, while 3 indicates a rotation of a quarter. Note that the centres of rotation depend on this angle. A degenerated state is represented by an angle of 0, which means that the x, z pattern obtained by the arcs and lines holds for each value of y.
`Left` is a check box indicating if the arcs and lines are situated on the left of the rotation center.
After the number of arcs and lines is specified, they are listed by their parameters. For the arcs, `xm` and `zm` denote the center, `r` the radius, and `t1` and `t2` the begin- and end angle of the arc, in radians. If `side` equals one, the part at the side of the center is taken, if it is two, the other side is added to Ω.
Line segments are described by their begin point (`x1,z1`) and their end point (`x2,z2`). `side` $= 1$ denotes the rectangular triangle under the line segment, `side` $= 2$ the upper triangle. If `side` $= 3$, the end points are the opposites vertices of a rectangle. Note that horizontal and verticle line segments do not contribute to Ω.

intree.txt

An example of the CSG-tree description is the following input file. Note that before the tree is defined, all primitive objects are listed.

```
nprims
5
nr; ptype ;  parameters
1
4
0.0 0.0 -0.9 0.5 1.0 -1.57 0.0
2
4
0.0 0.0 0.9 0.5 1.0 1.57 0.0
```

```
3
1
-0.5 -0.05 -0.05    1.0 0.1 0.1 0.0 0.0 0.0
4
2
-0.45 0.0 -1.0     2.0 0.05 -1.57 0.0
5
2
0.45 0.0 -1.0     2.0 0.05 -1.57 0.0
ntrees
1
length   x1        y1       z1       x2       y2      z2
9 -0.5 -0.5 -1.0 0.5 0.5 1.0
nr ; op/prim      left     right
1
-1 2 3
2
-1 4 5
3
-1 6 7
4
1 0 0
5
2 0 0
6
3 0 0
7
-1 8 9
8
4 0 0
9
5 0 0
```

After the amount of primitive objects has been specified (using `nprims`), they are listed by their number, object type (where 1 is a box, 2 a cilinder, 3 an ellipsoid and 4 a cone). Each type has a (different) amount of parameters. The first three always indicate the reference point from which the object is laid in $x-$direction. Then several object properties are specified (length,radius etc.) while the last parameters are the rotating angles (θ, ϕ, ω) around the axes (see chapter 4). However, cone, ellipsoid and cilinder do not need a rotation around the $x-$axis.
After the tree check box is set to one, the number of nodes (`length`) is specified, together with the bounding box of the tree. Then the list of nodes follows, beginning with the number and after that the type of the node:
A value smaller than zero represents a set operation: -1 for union, -2 for intersection and -3 for difference. The other two values are the node numbers for which the operator holds, i.e. they are the roots of the subtrees.

When the type value exceeds zero, the node is a leaf and the type value is just the object number listed above the tree. Since a leaf has no subtrees, the other two values may be set as dummy numbers.

It is noted that the files **intree.txt** and **in2D.txt** may be dummy files, consisting only of zeros at places where the number of arcs, lines, objects and trees is to be found. For example, it is possible to run **liqmake** with two dummy files and specify the fluid configuration in the subroutine **SURDEF** in *ComFlo*.

B.3.2 Output files

- **AVS files**
  comflo.###.fld describes the structure of the corresponding **comflo.###.dat** file stating the dimensions, coordinates, vector length, and so on. An example:

```
# AVS field
ndim = 3
dim1 = 86
dim2 = 86
dim3 = 32
nspace = 3
veclen = 5
data = float
field = rectilinear
 label = fluid
label = x-velocity
label = y-velocity
label = z-velocity
  label = pressure
#
variable 1 file=comflo101.dat filetype=unformatted  skip=          0  stride=1
variable 2 file=comflo101.dat filetype=unformatted  skip=     946688  stride=1
variable 3 file=comflo101.dat filetype=unformatted  skip=    1893376  stride=1
variable 4 file=comflo101.dat filetype=unformatted  skip=    2840064  stride=1
variable 5 file=comflo101.dat filetype=unformatted  skip=    3786752  stride=1
coord 1 file=coordx.dat filetype=ascii skip=1 offset=0 stride=1
coord 2 file=coordy.dat filetype=ascii skip=1 offset=0 stride=1
coord 3 file=coordz.dat filetype=ascii skip=1 offset=0 stride=1
```

  The data in **comflo.###.dat** is binary formatted; however, on different machines binary data is not compatible, forcing to make a different **comflo.###.fld** according to the current machine.
  The files **coordx.dat, coordy.dat, coordz.dat** contain all coordinates.

- **MATLAB files**
  coord.m

comflo###.m
See below.

- **speed###.dat**
  Contains columns with time,velocity (u, v, w) and pressure in specified point.
- **fill###.dat**
  Contains columns with time, amount of liquid and liquid percentage in specified box.
- **force###.dat**
  Contains columns with time and forces in x, y- and z direction in specified box.
- **flux###.dat**
  Contains columns with time and flux through specified plane.
- **partic###.dat**
  Contains columns with time and position (in x, y, z-coordinates) of a specified particle.
- **nriter.dat**
  Contains columns with total number of cycles and SOR-iterations at certain time levels.
- **dt_hist.dat**
  Contains the cycles where time step is changed, denoted by $+1$ if doubled or -1 if halved in the second column.
- **defrag.dat**
  Contains columns with time and absolute and relative number of S-cells.
- **autosav.dat**
  The restart file containing all information to continue a stopped calculation. Is automatically compressed because of the great size.

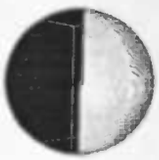
B.4 Postprocessing tools

B.4.1 AVS

The files **comflo###.fld** and **comflo###.dat** are used together in AVS-networks. A network consists of several modules attached to each other. After data has been read in by the Read Field module, several operations on the multi-vector filed are performed when finally part of the data (e.g. the free surface coloured with a velocity) is visualized in the Geometry Viewer module. AVS has been a powerful tool to visualize the 3D-aspects of the generated data.

B.4.2 MATLAB

The files **coord.m** and **comflo###.m**, as created by *ComFlo*, can straightforward be read into Matlab. **coord.m** contains only data of the grid (vectors x, y, z) while a file of the other kind defines matrices which values are physical (including the free surface) data in a plane of the geometry at one time step. Using commands like **contour** (for the free surface), **quiver** (for velocity fields) and **surf** (for the pressure), this data is visualized.
For the other aspect, information about evolution in time instead of place, a menu system has



been written with which graphs of velocities and pressure in points or on lines, filling degrees in certain boxes, forces in certain boxes, fluxes through certain planes and streamlines can be made. The concerned data files cannot be read in directly, but with commands like **fscanf**. It is also quite easy to process the data files containing numerical data. For more information we refer to the concerning routines in *ComFlo* and the Matlab help functions.

Bibliography

[1] E.F.F Botta (1992) Eindige differentiemethoden, *Lecture notes* RuG.

[2] J. Dijkstra (1997) Simulation of Flow past Complex Geometries using Cartesian Grids, *Master's thesis* RuG.

[3] J. Gerrits (1996) Fluid Flow in 3-D Complex Geometries – A Cartesian Grid Approach, *Master's thesis* RuG.

[4] J. Gerrits (1996) Three-Dimensional Liquid Sloshing in Complex Geometries, *Master's thesis* RuG.

[5] D. Hearn and P. Baker (1994) Computer Graphics, Prentice Hall.

[6] H.W. Hoogstraten (1992) Stromingsleer, *Lecture notes* RuG.

[7] B. de Groot (1996) SAVOF'96 – Simulation of free-surface liquid dynamics in moving complex geometries, *Master's thesis* RuG.

[8] A.E.P. Veldman (1994) Numerieke stromingsleer, *Lecture notes* RuG.