Abstract

A three dimensional parametric model is constructed to represent the simplified geometry of a waterjet duct. The number of variables in the model is not fixed. The motivation behind the model is to get a solution to the CFD (computational fluid dynamic) problem related to the flow of water in the duct. The parametric model is implemented in a computer program, which enables the geometry to be generated in the preprocessor of a CFD solver. The preprocessor converts the parametric geometry to a mesh, that becomes the implicit geometry existing in the solver. In the computer program algorithms are implemented to read, write, work with and manipulate this mesh.

The solver produces data files, where the solution values are stored after solving the CFD problem. The computer program can read these files. By using the program the information in a data file, for a mesh of a parametric duct can be modified to be used as initial data for another parametric waterjet duct. The quality of the initial data will be of good if the geometries of the ducts are similar.

Sammanfattning

En tredimensionell, parametrisk model av en förenklad geometri av ett vattenjet rör konstrueras. Antalet variabler i modellen är inte fixerat. Modellen används till att lösa fluidflödesproblemet, som är kopplat till vattenflödet i röret. Modellen implementeras i ett dataprogram, vilket möjliggör att geometrin kan genereras i preprocessorn till en fluidflödeslösare. Preprocessorn omvandlar geometrin till ett beräkningsnät, som blir den implicita geometri som existerar i lösaren. Algoritmerna i dataprogrammet konstrueras så att programmet kan läsa in, skriva ut, arbeta med och modifiera beräkningsnätet.

Lösaren genererar data filer, innehållande lösningsvärdena, när fluidflödesproblemet är löst. Även dessa filer kan programmet läsa. Genom att använda programmet kan informationen i en data fil från ett parametriskt rör modifieras så att den kan användas som initiallösning till ett annat vattenjet rör. Kvaliteten på den genererade initiallösningen blir hög, om geometrierna av vattenjet rören inte är alltför olika.

Preface

The work presented here completes my studies in Problem Solving in Science at Göteborg University, leading to the degree Master of Science. The idea behind the problem came from the cooperation between the two companies Kamewa and Caran Automotive. The great support and interest from the people at Kamewa, who has commissioned this work has been a great inspiration to me and I am grateful that they gave me this opportunity.

It has been a' privilege to be able to get knowledge from both the industry and the academic world and I would like to thank both my supervisors Johan Lennblad at Caran Automotive and Klas Samuelsson at Chalmers University of Technology and Göteborg University for making this possible, and for all the input and support through out the whole project.

Contents

1 Background

While technology is getting more advanced, the related problems are getting more complicated. In the search for new and improved technology many sorts of mathematical tools are used When the solution of a problem is impossible to find using practical testing, theoretical methods became a necessary compliment. For example, in most construction problems, the question is to optimise a free form geometry, which is described by a large number of variables. In these cases, mathematical optimisation is a useful tool, though the large number of parameters involved makes it difficult to find a workable optimisation method. Therefore there is a growing need for parametric geometry models, from which geometries easily can be generated, with a few parameters. These geometry models can then be combined with optimisation tools.

The work presented here is a first attempt to approach such a problem. The application is to enable optimisation of the geometry of a waterjet duct in order to minimise the pressure loss and the fluctuations in the outflow. An industry interested in this problem is Kamewa, situated in Kristinehamn, who has commissioned the study. The basic idea is to find a parametric model of the jet-tube, which can be generated easily to fit the solver for the fluid flow problem in the duct. It is an advantage if the number of parameters in the parametric model is small, because these are the variables that will be varied to find the optimal solution.

Because of the difficulties of finding a numerical solution for a viscous turbulent fluid flow problem, a lot of iterations has to be made before the solution converges. If the problem has a good initial solution the numerical solution will converge more quickly. Without an initial solution, an optimisation loop would be very costly, because of the high number of iterations that is needed for each geometry that enters the optimisation loop. The idea is to make small changes in the geometry during the optimisation process so that the initial solution can be generated from the calculations of the previous geometry.

The fluid flow problem to be evaluated in the optimisation is to be solved by existing software tools. The specifications from Kamewa is that the commercial program Fluent5 shall be used, and the geometry shall be generated in Gambit, the preprocessor to Fluent5.

2 Problem description

The work done here is a prerequisite of the fluid flow problem to make optimisation possible. The optimisation step itself is not at all addressed here. In general terms the work consists of making a parametric model of the waterjet duct, and mapping of a solution between two different meshes.

As stated before there is a need to make a mathematical model of the waterjet duct, which describes the geometry in a good way. The model has to have a small number of variable parameters and it needs to be very flexible so a large number of different geometries can be generated.

It is very time consuming to perform large CFD (computational fluid dynamics) computations. A large number of iterations has to be made to get a numerical convergent solution. In the process of optimising the geometry a large number of different geometries are to be evaluated, and with that a very large number of fluid flow problems will be solved numerically in an iterative process. Due to the limited computational power available, the iterations that solve the CFD problem have to be minimised, and this is the main issue addressed here. The general idea is that a solution from one geometry can be used as an initial solution to a similar geometry. If the solution fits the new geometry well, less iterations are necessary to get a convergent numerical solution and the computational power can be used to produce many CFD results, that can be compared in the optimisation process. When mapping the meshes the parametric model of the geometry will be used. By solving the fluid flow problem for one parametric waterjet duct, initial solutions for all other ducts, generated by the same parametric model can be generated. The quality of the initial data will of cause depend on how similar the geometries are. However, there will always be possible to map a solution, because even if the meshes do not have a Cartesian coordinate frame in common, the parametric coordinates will always be comparable. The general idea behind the mapping is that cells will inherit the solution values of the cells with the corresponding parametric values in the mesh, for which the fluid flow has been solved. Figure 1 shows the connection between two cell centre points in two different meshes, through the parametric space.

The main result of the work is a computer program written in the programming language C++. The functionality of the program is:

• to generate files which contain information about the geometry of the jet duct, which can be read by the solver.

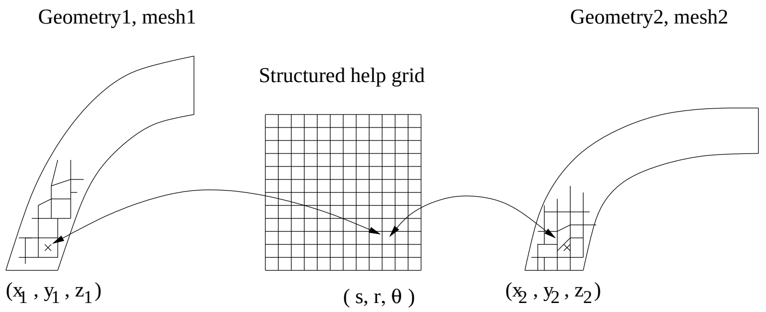

Figure 1: *The figure shows how the parametric space connects the two different geometries and meshes.*

• to read the mesh from mesh files, generated in the preprocessor of the solver.
• to compare cells in two different geometries, with different meshes, to enable the computed solution to be moved from one geometry to another.
• to map a mesh to a new geometry, while keeping the logical connection and deliver the information about the mesh to the solver.
• to read the solution values from the data files, which are created by the solver, after it has found a solution to the fluid flow problem
• to "move" the solution of one mesh and one geometry to another mesh of another (or the same) geometry
• to write a file containing the solution data to give a new mesh an initial solution.

In the following seven sections of the report, the above problem is further discussed and a solution is presented. Section 3, contains the theory behind the fluid flow problem. This section intends to give a reader, unfamiliar to fluid flow problems, an introduction to the subject and the background information needed to understand the problem addressed.

Before implementing the algorithms of the above items a lot of preparatory work, had to be done. This involved studying the solver, which was going to be used, to get an understanding about how to communicate with the program. Also an understanding about the formats of the files had to be achieved. The concerned files were those containing the input data, that the program reads, and the files, containing the output data which the program generates. Information gained during this study of importance to understanding this report, can be found in Section 4.

The main problems related to the algorithms to be implemented are presented in Section 5, together with a description of how these problems are solved.

The description of the computer implementation, which is the main result, is found in Section 6. General remarks and descriptions are made about what can be done using the program files, as well as detailed descriptions of the functions.

Section 7 contains an example of calculations made on a waterjet duct generated by the program files. Here it can be seen how the different programs interact, with the solver and the complete chain that the programs are part of becomes clear. Comments are also made about the result of the example, which gives an idea about how well the programs work in practice.

Conclusions are given in Section 8, and the report ends with Section 9, where recommendations are given about what can be improved by further work on the software and the algorithms.

3 Physical background to the fluid flow problem

In this section the aim is to point out what makes it difficult to solve a fluid flow problem. This difficulty is part of the motivation behind the present work. The fluid flow problem addressed here is assumed to be a viscous, incompressible fluid flow, with constant temperature. In the following section the general way to model viscous flow is presented, including the Navier-Stokes equations. In Section 3.2 the behaviour of turbulence is described and the special conditions for the application of the waterjet duct can be found in Section 3.3.

3.1 Modelling of incompressible viscous fluid flow

The motion of the fluid flow can be described by four partial differential equations, which enforces conservation of mass and momentum [6]. The conservation of mass is described by

$$\frac{\partial \rho}{\partial t} + div(\rho \mathbf{u}) = 0, \tag{1}$$

where ρ is the density and $\mathbf{u} = \mathbf{u}(x, y, z)$ is the velocity vector. It can easily be seen that the equation sets the rate of increase of mass in a fluid element equal to the net rate of flow of mass into the fluid element. In the case of incompressible fluids, where ρ is constant the equation for mass conservation is reduced to

$$div\ \mathbf{u} = 0. \tag{2}$$

The momentum of the fluid is conserved, when the rate of increase of momentum of a fluid particle equals the sum of forces working on the fluid particle. There are two different kinds of forces working on the fluid particle, that is the body forces ant the forces resulting from surface stress. The body forces (gravity force, centrifugal force, Coriolis force and electro-magnetic force) are usually summed and put in a source term, while the different forces (pressure forces and viscous forces), caused by surface stress, are described separately.

The magnitude of the forces resulting from stress is the product of the stress intensity and area. The state of stress of the fluid element is defined in terms of pressure and nine viscous stress components. The viscous stress is modelled to be a function of the local deformation rate. There are both linear and volumetric deformation rates. The linear deformation is $d\mathbf{u}/d\mathbf{x}$, where $\mathbf{x} = (x, y, z)$ and the volumetric deformation is $div\ \mathbf{u}$. For Newtonian fluid viscous stress is proportional to the rate of deformation, and there are two proportionality constants. One is called the dynamic viscosity, μ, and is linked to the linear deformation. The other constant is λ, which is linked to the volumetric deformation. For an incompressible fluid there is only linear deformation because $div\ \mathbf{u} = 0$. The viscous stress terms τ are therefore for incompressible flow twice the local rate of linear deformation, as Equations 3-8 shows.

$$\tau_{xx} = 2\mu \frac{\partial u_x}{\partial x} \tag{3}$$

$$\tau_{yy} = 2\mu \frac{\partial u_y}{\partial y} \tag{4}$$

$$\tau_{zz} = 2\mu \frac{\partial u_z}{\partial z} \tag{5}$$

$$\tau_{xy} = \tau_{yx} = \mu \left(\frac{\partial u_x}{\partial y} + \frac{\partial u_y}{\partial x} \right) \tag{6}$$

$$\tau_{xz} = \tau_{zx} = \mu \left(\frac{\partial u_x}{\partial z} + \frac{\partial u_z}{\partial x} \right) \tag{7}$$

$$\tau_{yz} = \tau_{zy} = \mu \left(\frac{\partial u_y}{\partial z} + \frac{\partial u_z}{\partial y} \right) \tag{8}$$

The continuity equations for momentum are found by setting the rate of change of the momentum in the x-direction of a fluid particle equal to the total force in the x-direction, due to surface stress, working on the particle plus the rate of change of the momentum in the x-direction, due to sources (body forces). Equation 9-11, show this equality for the momentum in the x, y and z direction respectively, where p is the pressure. S_{Mx}, S_{My} and S_{Mz} are the changes of momentum due to sources.

$$\rho \frac{du_x}{dt} = \frac{\partial(-p + \tau_{xx})}{\partial x} + \frac{\partial \tau_{yx}}{\partial y} + \frac{\partial \tau_{zx}}{\partial z} + S_{Mx} \tag{9}$$

$$\rho \frac{du_y}{dt} = \frac{\partial(\tau_{xy})}{\partial x} + \frac{\partial(-p + \tau_{yy})}{\partial y} + \frac{\partial \tau_{zy}}{\partial z} + S_{My} \tag{10}$$

$$\rho \frac{du_z}{dt} = \frac{\partial \tau_{xz}}{\partial x} + \frac{\partial \tau_{yz}}{\partial y} + \frac{\partial(-p + \tau_{zz})}{\partial z} + S_{Mz} \tag{11}$$

Equations 2, 9, 10 and 11 constitute the Navier-Stokes equations for incompressible viscous fluid flow at constant temperature.

3.2 Turbulent behaviour in viscous fluid flow

The random velocity fluctuations characteristic for the turbulent flow, give rise to additional stresses on the fluid. There is a close relation between the turbulence of the flow and its Reynolds number, which measures the relative importance of inertia forces and viscous forces of a fluid flow [6]. The turbulent behaviour increases with higher values of the Reynolds number, and for each type of flow there is a critical Reynolds number at which turbulence first occurs. The Reynolds number Re is described by the characteristic velocity U, the length scale of the main flow L and the kinematic viscosity ν according to $Re = UL/\nu$. The turbulence can be seen as eddies of different length scales. The wide range of length scales of the eddies that occur in the same fluid flow is a difficulty for the numerical methods, because it is too costly to adapt the mesh according to the smallest length scales. The eddies contribute to mix the flow effectively, which results in high diffusion coefficients for mass, momentum and heat. Turbulence is also inherently instationary, while the interest most often is in stationary flows.

To overcome the problem with different length scales various models for turbulence is used. One of these models is the two equation method called the $k\epsilon$-method. This method uses the time averaged Navier-Stokes equations, and accounts for the small scales that can not be fitted into the mesh by also modelling the turbulent kinetic energy k and the rate of dissipation of turbulent kinetic energy ϵ.

3.3 The flow in the waterjet duct

The flow in a waterjet is characterised by high Reynolds numbers. The Reynolds number depends on the flow rate through the waterjet and the dimensions of the inlet. Typical Reynolds numbers of waterjets are of the order of 10^7, based on the diameter of the inlet [8]. The turbulent behaviour is most apparent in a layer close to the walls of the duct, while there is a more uniform flow in the centre of the duct. In the centre flow the inertia effects dominate, while in the layer closest to the walls the viscous forces are equal in magnitude, or larger than, the inertia forces. This viscous layer becomes larger the more the pipe bends. For example, the boundary layer may develop to cover one-half of the cross section area, in a pipe that bends about 90 degrees [8]. Such a large boundary layer would have great impact on the the mean flow. It is therefore essential to have a good mesh, along the walls and as far into the duct as the boundary layer reaches, to accurately calculate the boundary layer development.

When solving these kinds of problems numerically, the cells closest to the walls are treated somewhat differently than the other cells, in the iterative process. This because the behaviour in the boundary layers successfully can be described with simple laws, included in the so called wall functions, which has been developed through both dimension analysis and studies of experimental data. The mean flow velocity close to the wall, only depends on the distance y, from the wall, the density ρ and the viscosity μ of the fluid and the wall shear stress τ_{wall}. By finding a relation between the wall shear stress, and the mean velocity, the mean velocity profile for the cells on the boundary, can be calculated.

The boundary layer of the flow in a pipe, with solid walls, can be divided into three different sub layers. At the solid surface the fluid is stationary, that is the velocity at the boundary is zero. In a very thin layer close to the wall there can also be no motion due to turbulence. This leaves only the impact of shear stress. Because the thickness of the layer, the shear stress can be approximated to be constant and equal to the wall shear stress. In the next layer, in which y still is very small, the turbulent and viscous effects are both important. In the last layer, the turbulent effects are the most dominating [6].

The large Reynolds number of these types of applications causes problems when calculating the numerical solution. A large Reynolds number indicates that transportation is the dominating force, compared to diffusion, in the flow. When calculating a numerical solution this means that small local computational errors upstream can amplify and spread, so that the solution downstream in the next iteration is greatly affected. This is why so many iterations are needed to find a convergent solution. The accuracy of the numerical solutions are also influenced by the stability of the physical problem.

4 The computational fluid dynamics program Fluent5

The computational fluid dynamics program Fluent5 is a widely used commercial program. It uses the finite volume method to solve both laminar and turbulent fluid flow. The range of different schemes include among others: first and second upwind schemes and the QUICK scheme. There are several turbulence models available in the program. The most popular model, is the $k\epsilon$-model with wall functions[7]. This partly because the constants involved are quite well known, and the model is considered to be adequate for many problems.

4.1 Communicating with Fluent5

To be able to communicate with Fluent5 it is important to know how the information is handled internally. Below it is described where and how information is used in Fluent5.

The geometry is created in Gambit, the preprocessor of Fluent5, either by "drawing" the geometry in Gambit and letting Gambit record this geometry in a journal file, or running a journal file generated outside Fluent5. The latter is what is done here. The mesh is also created in Gambit, using the information in the journal file, which apart from the geometry also includes mesh size and the types of the boundary conditions.

In Fluent5 no explicit geometry exists. The geometry is implicitly described by the nodes in the mesh. To enable the mesh to be read by the solver Fluent5, the mesh is exported by Gambit to a msh-file. The mesh file contains the coordinates of the nodes, together with information about how the different cells are constructed from the nodes.

When solving the flow problem, using Fluent5, first the msh-file has to be read. Additional to the information in the msh-file, the solver needs information about the boundary conditions, and physical properties, such as density and viscosity. The initialisation is done by setting a velocity magnitude and

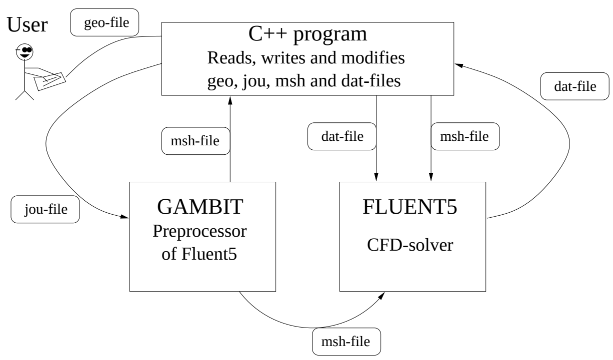

Figure 2: *The information flow between the different programs*

direction at the inlet. All of these specifications are given manually by the user in the user interface of the program. When all information is gathered a case-file is made, that holds all this information, so the process of setting the boundary conditions and other initialisation has to be done only once for each mesh. If there is a data file, which can be used as initial data, also such a file should be read before the iterative process to find a solution starts. If there is no data file, the solution will be set to zero in all cells as initial solution. Consequently only the case file is necessary for the solver.

When the numerical solution is found the user commands the program to write a new data file, where the solution is stored. The stored solution values in the data file are of two types. Some of the values are calculated through the iterative process, and are therefore situated in the centre of the cell. The other solution values are found through calculations based on the the centroid values. These post-processed solution values are situated on the cell faces.

5 Solving the problem

The general purpose of this work is to generate a computer program, that can serve as a compliment to Fluent5 and Gambit. Figure 2 shows a visual presentation of how the computer program will work and interact with Fluent5 and Gambit. The computer program should fulfil the items below.

- Generate a parametric geometry of a waterjet duct, and store the information about the geometry and the mesh size in a journal file, which can be read by Gambit, the preprocessor of Fluent5.

- Read the information about the mesh in the msh-file that Gambit creates, when it has executed all the commands in the journal file. The information should also be sorted and stored adequately.
- Read and sort the computed solution from the data files created by Fluent5, and relate the solution to a grid.
- Compare cells in two different geometries, with different meshes, to enable the computed solution to be moved from one geometry to another.
- Meshing a new geometry, by using the parametric values of the nodes in an already existing mesh, and writing a new msh-file, so the new mesh can be communicated to Fluent5.
- Mapping the solution of one mesh and one geometry to another mesh of another (or the same) geometry
- Write the dat-files containing the computed or moved solution to give a new mesh an initial solution.

Before writing the program some principal problems have to be solved. This section deals with the principal solutions to these problems, while the computer implementation is described in Section 6.

The first problem is to give a parametric description of the geometry of the waterjet duct. The general idea is then to use the parametric coordinates to map different meshes. There is therefore a need to find a way to go from the parametric coordinates to the Cartesian coordinates and the other way around. The last part is to create a way to move solution values of one geometry to another. All these problems are discussed separately in the following sections.

5.1 Modelling the waterjet duct

5.1.1 The parametrisation

As a first approximation, the waterjet duct is assumed to be shaped as a cylindrical bended pipe. The bending of the pipe is described through a nurb curve (non uniform rational B-spline curve), which is situated in the centre of the pipe [1]. All points on the curve corresponds to a parametric value s, which goes from zero at the inlet to one at the outlet, i.e. $0 \leq s \leq 1$. In this way all points on the curve can be described through just a one dimensional parameter, instead of the three coordinates describing the location in the Cartesian coordinate system. By knowing the parametric value s a plane perpendicular to the curve is found. All points on this plane can then

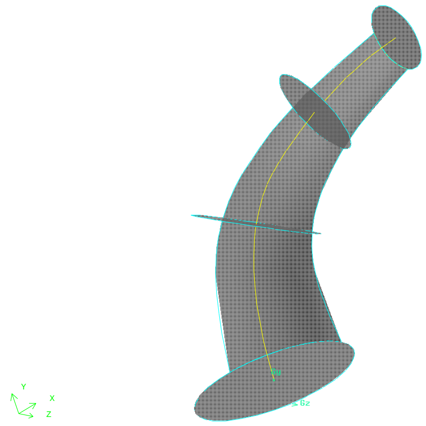

Figure 3: *The figure shows the parametric model of the water jet duct. The nurb curve describing the bending of the pipe can be seen in the centre of the duct. From inlet to outlet the parameter s takes values between zero and one. Four elliptic cross sections of the duct are also shown. In each plane the parameter radius take values between zero and one and the angular parameter θ values between zero and 2π.*

be described by setting the distance from the centre curve (the radius r) and the angle θ, between the normal of the centre curve and the vector, going from the centre curve to the point being described. Figure 3 shows the parametric geometry of the waterjet duct.

Below the general b-spline representation is shown. The weighting functions that are connected to rational b-splines are not accounted for, because in this application the weights are set to one, and the rational b-splines thereby becomes "ordinary" b-splines.

The polynomial that defines the centre curve is created from i, piecewise defined, polynomials and a set of control points $\mathbf{V_i}$. The knots s_i set the interval sizes for the so called blending functions, $N_{i,k}(s)$, which are of degree (k-1) and are C^{k-2} continuous, see Figure 4. The only requirement on the knots are that they are of nondecreasing order, that is $s_i \leq s_{i+1}$, and the value may not be the same for more than k knots. The relation between the order of the blending functions and the number of control vertices and knots is that the order should amount to the difference between the number of knots and the number of control vertices.

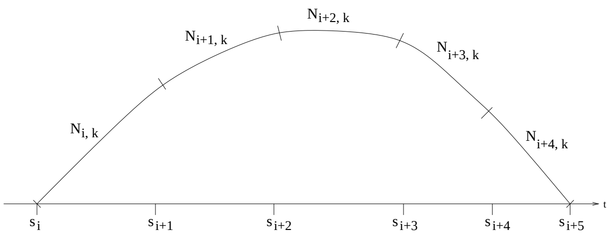

Figure 4: *Non uniform b-spline basis functions $N_i(s)$*

Below, in Equation 12 it can be seen how the one dimensional value of s together with the scalar base functions and the matrix, with the control points give the three dimensional function value $\mathbf{P}(s)$, $\mathbf{R} \rightarrow \mathbf{R^3}$. The vector $\mathbf{P}(s)$ corresponds to the x, y and z coordinates of the point on the nurb curve

$$\mathbf{P}(s) = \sum_{i=0}^{n} N_{i,k}(s)\mathbf{V_i}, \qquad 0 \leq s \leq 1. \tag{12}$$

The definition of the blending function on each interval i is:

$$N_{i,1}(s) = \begin{cases} 1 & \text{for } s_i \leq s \leq s_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

and the recursive definition:

$$N_{i,k}(s) = \frac{(s - s_i)}{(s_{i+k-1} - s_i)} N_{i,k-1}(s) + \frac{(s_{i+k} - s)}{(s_{i+k} - s_{i+1})} N_{i+1,k-1}(s).$$

The variable parameters that control and change the centre curve of the pipe are the control points. Only a few points are needed to get a smooth curve, and because the model shall be used for optimisation, as few parameters as possible is preferred. When implementing this parametric model four control points are used. This seems to be a reasonable number of points, regarding both to maximise the flexibility of the geometry, while the number of parameters to be optimised is minimised. In the case where the position of the centre point of the inlet and outlet are fixed, there would only be two free control points.

To get a pipe with smooth walls, but at the same time not need too many different variable parameters, also the radii from the centre curve to the walls are described by nurb curves. A special two dimensional nurb curve is created to include information about the two radii a and b, that are needed to get an elliptic cross section. In this way all cross sections of the pipe are set.

This curve is called ab-curve, and also for this nurb curve four control points are used. These are the rest of the variable parameters.

If one assumes that the centre point of both the inlet and outlet are fixed, the total number of variable parameters in the parametric model becomes two three dimensional points on the centre curve and four two dimensional points on the ab-curve defining the cross sections.

A summary of the variables describing a point in the parametric space is given below.

s , is the scalar value, of the nurb curve in the centre of the duct. The parametric value defines the different cross sections of the pipe, when $0 \leq s \leq 1$.

r , is the radius, or more specifically the distance from the centre curve to the point, in the plane set by s. The radius r is scaled by the distance from the centre curve to the wall, so $0 \leq r \leq 1$.

θ , is the angle between the radius r and the normal vector to the centre curve in the plane defined by s. The angle lies in the interval $0 \leq \theta < 2\pi$.

5.1.2 Implementation of the parametric model to generate geometry in Gambit

The parametric model is used to generate a geometry in Gambit (the preprocessor to Fluent5). Gambit uses vertices, edges and faces to create volumes, so the first step is to calculate the coordinates of the vertices that shall define the pipe. To enable a good structured mesh the geometry consists of five different volume segments, as shown in Figure 5. Therefore there is a need to calculate both the vertices defining the walls of the pipe and those defining the inner structure of the different volume segments.

By evaluating the normal and binormal along the centre curve, at certain parametric values, the cross section planes are found, see Figure 3. Eight vertices are calculated for each cross section, by using the normal and binormal vectors and evaluating the radii at the given parametric value, see Figure 5. The vertices are then used to form eight edges. A command in Gambit is used, which interpolates the vertices to get a smooth nurb curve. Because the information, about how Gambit creates these nurb curves, is not available, the calculations are made at several cross sections to ensure that Gambit creates the intended geometry. The inlet and outlet of the pipe are also made of nurb curves. The vertices, which shape the curve, describe

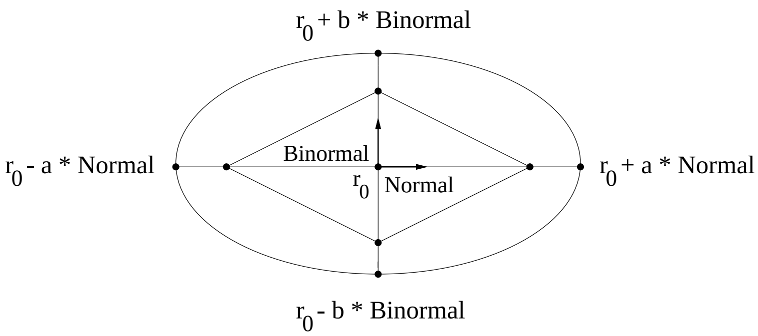

Figure 5: *Elliptic cross section, of a pipe, perpendicular to the tangent of the centre curve in the pipe. The cross section is divided into a diamond shaped section in the middle, which corresponds to the cross section of the inner volume of the pipe, and four sections around the diamond, corresponding to the cross sections of the four outer volumes.*

ellipses in this model, but this could easily be changed if desired.

To form faces in Gambit all that is needed is a closed set of edges. Four faces are formed using the four edges describing the walls and the inlet and outlet curves, each split into four separate edges. In a similar way, faces are constructed in the inner structure of the pipe. The faces are then stitched together to form five volumes. The reason for making these five different volumes, is the importance of meshing of the boundary layer correctly. One volume is situated in the middle of the duct, where the flow is easier to numerically resolve. This volume need not be meshed as fine as the other four, which are situated along the walls, see Figure 5. The dimensions of the volumes can be varied in the program, by setting the inner vertices closer or further away from the centre. If the geometry represents a very bended pipe, the outer volumes should be enlarged, so that the whole boundary layer is fitted into these.

5.1.3 Advantages and limitations of the parametric model

Often in computational problems a great difficulty may be just to generate or find a way to represent the geometry. A parametric representation makes it easy to generate many different, but yet similar geometries.

One additional purpose of the parametric model is to limit the variables describing the geometry to enable optimisation. When using the nurb curve to describe the shape, the information that lies in the three coordinates can be described by only one parametric value. Another great advantage is the flexible form of the nurb curve. This is an important aspect when the parametric model is used for optimisation, because the wish might be to find a global optimum, and not just a local optimum for the geometry. This means that it is necessary to be able to not only make small changes in the geometry during the optimisation loop. A lot of different geometries can be made with the nurb curve describing the bending of the pipe and the shape of the inlet and outlet, though there are some limitations.

One limitation is that the centre nurb curve lies in one plane. This limitation is caused by the way the vertices, that Gambit uses to generate the geometry, are calculated. When the z-coordinate is not constant the normal starts to spin around the centre curve, and the way that the vertices, along the normal and the binormal, are calculated will no longer work. Also for s-shaped curves the normal will at some point change direction, which means that these kinds of curves are also not allowed in the model. One way of eliminating this problem would be letting also the radii change directions along the centre curve. This is however not in the scope of the present work.

When the meshes of two geometries are compared it is a great advantage to be able to use the parametric variables, instead of the normal Cartesian coordinate system. The parametric coordinates make it easy to copy and stretch one mesh from one geometry to fit another geometry.

5.2 Changing coordinates between the Cartesian and the parametric coordinate systems

One method to compare two meshes, based on different geometries, is to compare the parametric node values. To enable this kind of comparison there is a need to be able to make transformations back and forth the parametric coordinates.

When going from Cartesian coordinates to parametric coordinates, the parametric value of the nurb curve is calculated. However, before doing this the closest point on the curve has to be found. This is done by finding the parameter s, which corresponds to the minimum distance between the point, which coordinates are being converted, and the nurb curve in the centre of the duct. This can also be described as minimising the function $f(s) = |\mathbf{x} - \mathbf{x_0}(s)|$, where $\mathbf{x}$ is the vector with the x, y and z coordinates of the point that is being transformed and $\mathbf{x_0}(s)$ is a point on the centre curve, depending on s. The optimum is found through an iterative procedure.

The next parameter to be calculated is the radius. This is done by subtracting the coordinates of the point found on the centre curve from the coordinates of the point that is being evaluated. This vector is named $\mathbf{v_1}$. The radius is then scaled, by being divided by the centre to wall radius R_{wall}, corresponding to the distance between the centre of the duct and the wall in the direction given by $\mathbf{v_1}$. This radius is given by $R_{wall} = (a^2 \cos^2\theta + b^2 \sin^2\theta)^{1/2}$, where a and b are the radii of the ellipse, see also Figure 6

The angular parameter is now easy to compute. This parameter θ cor-

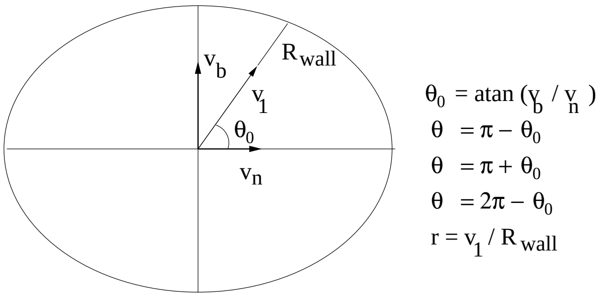

Figure 6: *Cross section of the waterjet duct, showing the parametric variables the angle θ and the radius r. See also the different values of the angle depending on the quadrant.*

responds to the angle between the normal vector and $\mathbf{v_1}$. The angle is calculated using arctan of the length of the projection of $\mathbf{v_1}$ on to the binormal, and the length of the projection of $\mathbf{v_1}$ on to the normal. The angle is later adjusted according to which quadrant $\mathbf{v_1}$ is in. The following adjustments are made in the different quadrants:

The first quadrant: $\theta = \theta_0$
The second quadrant: $\theta = \pi - \theta_0$
The third quadrant: $\theta = \pi + \theta_0$
The fourth quadrant: $\theta = 2\pi - \theta_0$

Going from parametric coordinates to Cartesian coordinates is more straight forward. First the parametric s value is used, to determine which cross section of the duct the point is located on. By using equation 12 the Cartesian coordinates are found, for the point on the centre curve located on the cross section in question. Both the normal (x_n, y_n, z_n) and the binormal

(x_{bn}, y_{bn}, z_{bn}) of the centre curve are evaluated at the parametric value. The Cartesian coordinates for the point, that is being converted is then obtained, by

$$x = r_{0,x} + r * (a * \cos\theta * x_n + b * \sin\theta * x_{bn}), \tag{13}$$

$$y = r_{0,y} + r * (a * \cos\theta * y_n + b * \sin\theta * y_{bn}) \; and \tag{14}$$

$$z = r_{0,z} + r * (a * \cos\theta * z_n + b * \sin\theta * z_{bn}). \tag{15}$$

In the equations $r_{0,x}$ is the x-coordinate for the point on the centre curve, $r_{0,y}$ the y-coordinate and $r_{0,z}$ the z-coordinate. The radii of the elliptic cross section are called a and b, see also Figure 5.

5.3 The comparison of two grids and mapping solutions between them

The whole idea of moving a solution from one mesh to another, is based on the possibility to compare cells in the two meshes. First the concept of how cells in one grid correspond to cells in another grid will be discussed, and then some ideas of how the solution can be moved and modified to make a good initial solution for the second mesh is presented.

5.3.1 Finding the corresponding cell in another mesh

Due to the parametric coordinates two meshes, that are based on different geometries, can be compared. To exemplify the discussion two meshes are defined, mesh1 and mesh2. These meshes may be of different size and be based on different geometries. In this whole discussion mesh2 is the mesh that has a solution, and mesh1 is the mesh, to which the solution is moved. The corresponding cell, cell2, in mesh2 to a cell, cell1, in mesh1, is here defined as the cell that contains the transformed centroid of cell1. The transformation of the centroid, consists of first converting the Cartesian coordinates of the centroid in cell1 in mesh1 to parametric coordinates using the geometry of mesh1. Then the parametric coordinates are converted to Cartesian coordinates using the geometry of mesh2. These coordinates are those that are compared with the nodes of cell2 in mesh2, see also Figure 7

The simplest way to find the corresponding cell, is to check all cells in mesh2, for each cell in mesh1, until the right cell is found, but when the meshes are large this is too costly. It is clear that the number of operations will be proportional to $N * N$, where N is the number of cells. This problem can be solved by using the parametric coordinates to relate all cells in the two

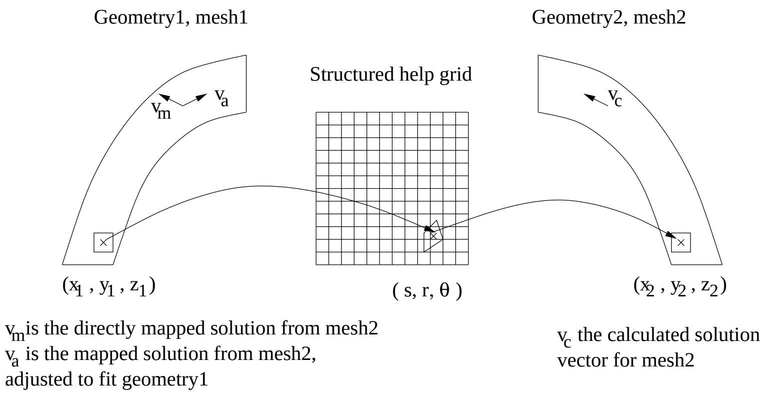

Figure 7: *The figure shows how a cell centroid corresponds to a point in another geometry, and points out the difference between a directly mapped solution vector and an adjusted one. When adjusting the vector the local coordinate system, with the base consisting of the normal, the binormal and the tangent is used.*

meshes to a help mesh, which is a totally structured mesh. The three dimensional cell numbers, of the structured mesh, corresponds to the parametric values. By using this help mesh as a reference the search for the cell can start in an area close to the correct cell. The process of relating the meshes to the help mesh is not very costly. However, if the difference between the size of the two meshes, that are compared, is large, some problems may arise. The size of the help grid is based on the smallest distance between two neighbour cells found in the two meshes. By looking at the parametric coordinates of the centroid of the cell in the "real" mesh the related cell in the help grid is found. In the same way the help cell is related back to the cell in the "real" grid. This means that in places where the mesh is coarse a lot of help cells will not be given a reference to a cell in the "real" grid. The whole idea of going from one mesh to the other via the help grid is then spoiled, because the chain of information between the two meshes that are being compared is broken. To overcome this problem the "empty" help cells are given references to those cells that the neighbour help cells are related to. This problem is illustrated in Figure 8.

Another consideration regarding the help grid, is that if the help grid is made too coarse, more than one cell in the real grid will be related to the same cell in the help grid. As this is implemented here only one cell can be referred to per help cell. This means that only the first cell related to the help cell, will in fact be referred to. Note that the cell stored in the help grid

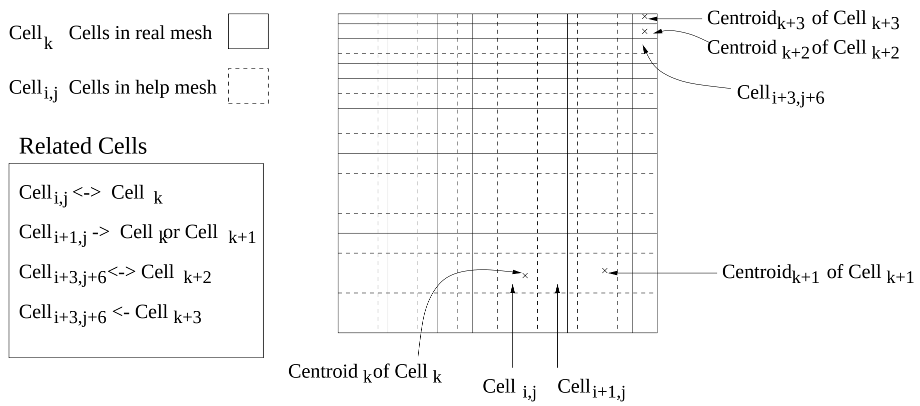

Figure 8: *The figure shows how references are made to the parametric structured help grid from the "real grid", and the other way around. When a help cell does not include a centroid of a cell in the "real grid", a reference is given to the same cell as one of the neighbouring help cells refers to. If a help cell has several centroids located inside it, the cell with the lowest cellnumber will be referred to, because only one reference can be made per cell.*

is close to the cell not used. Also this problem is illustrated in Figure 8. It is difficult to make the help grid in the right size. If the mesh is made very fine then the problem with overwriting information is solved, but then the algorithm that fills the empty help cells will be very costly.

When using this algorithm, with its intrinsic weaknesses, there is a need to control that the transformed centroid of cell1 in mesh1 really is situated in the cell, referred to by the help grid. By expressing the centroid in local cell coordinates it is possible to decide if the point lies inside or outside the cell. If the centroid is not situated in the cell, the neighbour cells are investigated, and so on. If the related cell for cell1 in mesh1 is not found in the first three layers of neighbours, a loop is made of all the cells in mesh2 to find the related cell.

Yet another difficulty appeared, when working with the grid. Because of the way the preprocessor sets the nodes in the mesh, some nodes are set outside the parametric space, and the linear interpolation between the nodes contribute to that parts of the geometry is put outside the mesh. This immediately produces a problem when two meshes are compared, because even if a centroid is part of the parametric space and the mesh in one geometry, it is not certain that it is located inside the other mesh. This means that even if all cells are checked in the other grid no related cell will be found.

This problem gives rise to a need for another way of defining a corresponding cell in the other grid. The new way of finding a corresponding cell is to choose the cell, that has the centroid situated closest to the transformed coordinates of the centroid in question. The problem with this way of finding the corresponding cell is that all cells in the other grid must be investigated, because there is no way of knowing how close the closest centroid is located, before all cells are checked. This is a time consuming task that should be avoided if possible.

One way of avoiding looping through all cells is to just check those cells that are suspected to be situated close to the centroid. The cell that is referred to in the help grid should of cause be checked, and also its neighbours.

If the parametric coordinates of the centroid are larger than the parametric space allows, the parametric help grid is enlarged, to handle also the cells, who has nodes outside the parametric space.

In the implementation there are several functions that find the related cell in the other grid. These functions all use slightly different combinations of the methods described above. The functions are `getRelatedCellsInOtherGrid`, `getRelatedCellsInOtherGrid2`, `getRelatedCellsInOtherGrid3` and the algorithms of these functions are found in Section 6.3.2, where the Grid class is described. The function mentioned first is the most "accurate" one, but it is also the one that takes the longest to run. The last one is quickest, and probably "accurate enough".

5.3.2 Moving the solution

When the corresponding cell in the other mesh is found, as a first step, the initial solution for the cell is set to the solution of the corresponding cell centre. The solution values that are situated on the cell faces that also are given in the data file from Fluent5, are not used in the present work.

The scalar solution values are mapped over to the new geometry without any adjustment, but the solutions that are vectors are treated in a special way. As can be seen in Figure 7 vectors mapped directly to the new geometry, without any adjustments, can be directed in an inaccurate way. In Figure $\mathbf{v}$ the original vector solution, $\mathbf{v_m}$ is the directly mapped vector solution, and $\mathbf{v_a}$ is the adjusted mapped vector solution. A local coordinate system, which is unique for every cross section of the pipe, is used to map vectors. If the base is composed of the normal, the binormal and the tangent of the centre curve, the length of the vector can be preserved, while the direction is adjusted. The vector is written as a linear combination of the base vectors, for the cross section that it is "moved" from. These base vectors are called $\mathbf{T_{m2}}$, $\mathbf{N_{m2}}$ and $\mathbf{B_{m2}}$. The expression for the vector solution becomes

$$\mathbf{v} = a_t * \mathbf{T_{m2}} + a_n * \mathbf{N_{m2}} + a_b * \mathbf{B_{m2}}. \tag{16}$$

Because of the ortonormality of the base the coefficients are $a_t = \mathbf{v} \cdot \mathbf{T_{m2}}$, $a_n = \mathbf{v} \cdot \mathbf{N_{m2}}$ and $a_b = \mathbf{v} \cdot \mathbf{B_{m2}}$. The adjusted moved solution is written as a linear combination of these coefficients and the base vectors given by the tangent, the normal and the binormal in the cross section, where the solution vector is put in the other mesh. The base vectors in the cross section, to which the solution is moved are called $\mathbf{T_{m1}}$, $\mathbf{N_{m1}}$ and $\mathbf{B_{m1}}$. The adjusted vector then becomes

$$\mathbf{v_a} = a_t \mathbf{T_{m1}} + a_n \mathbf{N_{m1}} + a_b \mathbf{B_{m1}} \tag{17}$$

Because of the ortonormality of both bases the length of the original solution vector is preserved, but the direction is somewhat adjusted to better fit the geometry where it has its new location. Figure 7 points out the importance of adjusting the direction of the mapped solution vector by showing two geometries that are exactly the the same, but the pipe bends in opposite directions. The situation when the geometry of the water jet duct is optimised, will not be this drastic, and the changes in the direction of the adjusted solution vector will not be this great, but hopefully this adjustment will make the mapped solution a better initial solution.

By mapping the solution in the ways described above, all cells get a solution value, and these are the values that later on is given to Fluent5 as the initial solution to the new fluid flow problem. The quality of the total initial solution may though be quite bad, especially if the mesh sizes are very different. Even if the meshes were of exactly the same size, the initial solution may be bad, because the solution of the Navier-Stokes equation is dependent on the shape of the geometry. If a method of mapping could be found analytically that takes both the geometry, equations and the solution into account, then the converged solution could be found directly. To find the exact way to map is however impossible, dealing with the Navier-Stokes equations and these relatively complicated geometries.

To improve the initial solution, more than one cell in the other grid may be accounted for. If the mesh, that the initial solution shall be moved to, is finer than the original mesh, where the solution is taken from, linear interpolation, is a better idea. For the interpolation also the solution in the neighbouring cells, to the corresponding cell, can be used. In this way the moved initial solution becomes smoother and more of the information hidden in mesh2 is used.

In the case where the solution is moved to a coarser grid, some kind of average value of the corresponding cell and those other cells whose centroids lie in the cell1 in mesh1, should be used. It seams reasonable to weight the solution values according to the distance to the centroid of cell1 that is given the initial solution.

Even further improvements of the initial solution, would be to adjust the solution values so that continuity would be fulfilled for all the cells. That is the velocities should be adjusted so the mass flow into each cell would be equal to the mass flow out of the cell.

6 The computer implementation

The result of the present work is not one single program. What is accomplished is a number of functions that can be used when making different smaller programs. To enable separate treatment of the geometry, mesh and solution values the computer implementation consists of four different parts, which are connected to each other.

The geometry of the waterjet duct is represented by a Pipe object. The Pipe object is described by a centre curve and another nurb curve, describing the geometry of the elliptic cross sections of the waterjet duct.

To each Pipe object there is also a Grid object. The Grid object contains the information about the mesh. In Gambit the mesh is represented by different faces that constitute cells. Each cell then consists of six (quadratic elements) or four (tetrahedral elements) faces. The faces consists of four or three nodes. The parameters describing the mesh, are therefore nodes, faces and cells. For each face the number of cells sharing the face is given. In this way one knows how the cells are connected. This information is also computed and stored the Grid object.

The Grid object also has a Pipe object, in the same way as the Pipe object has a Grid object . The geometry parameters are essential when the mesh is compared to another mesh, and this is why the two are linked. To the Grid object there is also a DataToGrid object, which holds the information about all the solution values. To each cell there is a solution value, and the solution value is situated in the centre of the cell because Fluent 5, uses a collocated finite volume method.

To each Grid object there are several types of solution. Velocity and pressure are two of those. To ease the handling of all these DataToGrid objects, all the DataToGrid objects that are connected to one Grid object is gathered in a AllDataToGrid object.

Figure 9 shows an overview of the class objects that constitute the spine of

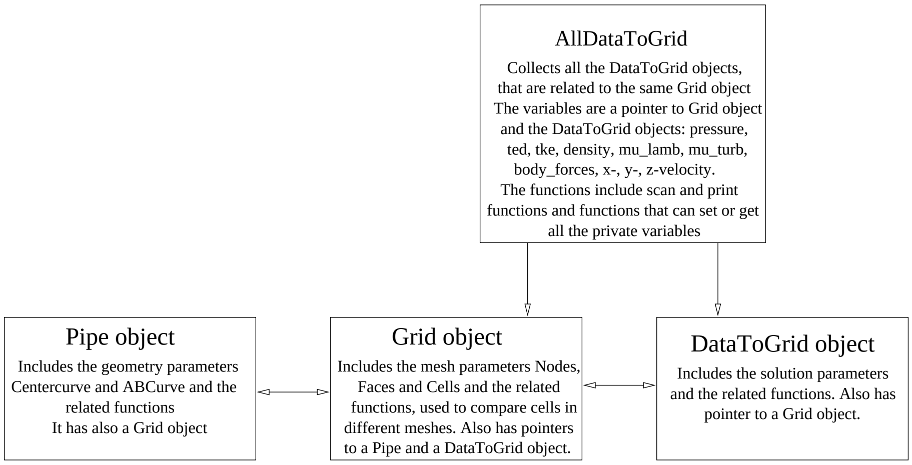

Figure 9: *Overview of the classes in the program, consisting of the geometry class Pipe, the mesh class Grid, the solution class DataToGrid and the class AllDataToGrid which contains the solutions values of all solution types.*

the program. Before further descriptions are made about the program files Section 6.1 gives a short introduction to the programming language C++, so that a reader unfamiliar to C++ can follow the discussions related to the computer implementation. In Section 6.2 some examples are given of what can be done using the functions. This in done in the form of main-functions, together with descriptions of what they are used for and also what the functions in the main-functions do in very general terms. In the Section 6.3 more detailed information is given about the classes and the related functions.

6.1 A short introduction to C++

Typical for the object oriented, programming language C++, are "classes" that are the building stones of the programs written in C++. These classes are made to hold and handle information that is related and should be treated as an item. Each class has variables, in which the data is stored, and member functions that handle the variables, and thus the data. The variables are usually "private", which means that the variables can not be reached outside the class. These variables are set or accessed through special member functions, which are "public", and therefore can be used outside the class.

By using these larger building blocks the programs get easier to grasp, and easier to handle by the user. Also these classes can be used independently in other programs.

The different classes are defined in header files called "classname.h". In these files the variables and member functions are declared. In the file "classname.C" the implementation of the functions is found. The program itself is defined in a main.C file, where the header files of the classes that are included files.

A C++ function is declared as follows. First the type, returned by the function, is stated. The type could be either an "ordinary" type as integer or float, but could also be of abstract type, i.e. a class, defined by the user. If nothing is returned the type is void. Then the class that the function belongs to is stated followed by "::" and then the name of the function. The arguments that the function takes are inside parentheses after the function name and each argument type is declared before the argument name. For example a function, named testFunction, that is of the class Pipe, which returns nothing and takes the integer variable "testnumber" as an argument, is declared `void Pipe::testFunction(int testnumber)`.

When using the class functions, the object that the function shall be used on is stated, followed by a dot, and then the function name. In the parenthesis behind the function the information that the function needs is stated. This can be done, either by making a copy of a variable that exists outside the function, and sending this copy into the function, or by giving a reference to a variable, that exists outside the function, to the function. In the first case, only the copy can be changed by the function, and the original variable is kept intact. As for the other case the variable itself is treated by the function, and changes can be made, on the variable, that is remembered outside the function. When variables are sent as references into functions, the type declaration before the argument is followed by a &, when the function is declared. Assuming that the variable "testnumber", is given as a reference to the function declared above, the declaration would instead be `void Pipe::testFunction(int& testnumber)`. When using the function no type is specified in the parenthesis, and therefore no & symbol is used.

In addition to the classes made here also classes that defines vectors and matrices are used. These classes are called `MV_Vector` and `MV_ColMat`. The last class handles two dimensional matrices.

6.2 Main-functions using the computer program files

The following examples of main-functions are those that has been made to fulfil the demands that has been put on the programming work. The requirements are itemised below.

- To enable generation of the simplified geometry, of a waterjet duct in Gambit, that easily can be changed and modelled.
- To easily generate meshes for different geometries.
- To "move" the solution from one mesh to another, where the meshes may be based on different geometries.
- To communicate information about the mesh and solution to Fluent5.

6.2.1 Creating a Pipe object and printing a journal file

One of the goals with the program was to generate a volume using the parametric model and make the information about the geometry readable for Fluent5 or rather its preprocessor Gambit. This is achieved by the main function below, which uses functions from the Pipe class.

```
int main(int argc,char **args)
{
   Pipe pipe1("pipe1_in");

   int no_mesh_intervals_length_in=60;
   int no_mesh_intervals_each_part_ellipse_in=10;
   int no_boundary_pts_in=30;
   int no_pts_on_ellipse_in=60;
   int no_mesh_intervals_short_edge_in=10;
   double a_dist_inner_volume;
```

```
    double b_dist_inner_volume;
    int jderiv=0;

    pipe1.printStructMeshedVolume(no_boundary_pts_in,
       "journal_pipe1.jou",
       jderiv, no_mesh_intervals_each_part_ellipse_in,
       no_mesh_intervals_length_in,
       no_pts_on_ellipse_in, no_mesh_intervals_short_edge_in,
       a_dist_inner_volume, b_dist_inner_volume);
}
```

First a Pipe object is created through the constructor that takes a file name as argument. In this file the control points, for the centre curve and the ab-curve, i.e. the minor and major axis of the cross section ellipse, are set in advance by the user. If the file does not exist, the function will let the user know this by printing a message in the command window. The constructor reads the file and creates the object, which is called pipe1. The next function `printStructMeshedVolume` that is used demands that a few parameters regarding the mesh size, and the number of cross sections that should be calculated, are set by the user before the function is used. These parameters are arguments to the function, together with the name of the journal file that is written, when the program is run.

6.2.2 Reading the mesh from a msh file and copying it to another Pipe object

When there is a grid available for one geometry, it is of great use if this grid can be adapted to fit another geometry. This could be an alternative to using Gambit, the preprocessor to Fluent5. The advantage of this method is that a new msh-file for the new geometry can be printed, based on the information from the old grid. The data file to the old grid can therefore be used as input to an initial solution, without any adjustments. The following main-function creates a new Grid object, with a new geometry, by copying and stretching a mesh from another Grid object, with another geometry, to fit the new geometry.

```
int main(int argc,char **args)
{
    Pipe pipe1("pipe1_in");

    int no_faces_per_cell_in=6;
```

```
    Grid grid1(pipe1, no_faces_per_cell_in);
    grid1.scanGridMSH("pipe1_a.msh", no_faces_per_cell_in);

    Pipe pipe2("pipe2_in");
    Grid grid2(grid1, pipe2);
    grid2.printMSHfile("grid2_pipe2.msh");

    return 0;
}
```

The main-function consists of two different parts. In the first part the Grid object is created. It is then used to create the new mesh to the other geometry. By using the Pipe constructor that reads a file, the geometry is set and a Pipe object is created. After this the number of faces that constitute a cell is set, because this is one of the input parameters to the constructor, which creates the Grid object. The other parameter used by the constructor is the pipe object that has been created. To set the coordinates of the nodes the function "scanGridMSH" is used, where the file that should be read is set in the argument. At this point all information about the grid is set, and the second part, where the grid is "copied" to another Grid object, starts.

First a new Pipe object is created, as before, but the information about the geometry is now collected from another file. The new Grid object is then created by the special constructor, that calculates new coordinates for the grid based on the "old grid", the "old geometry", and the new geometry of the new Grid object. The constructor also copies the faces and how they are situated. Finally a new msh-file is printed for the new Grid object by the function "printMSHfile". The name of this file is given in the argument. This file can then be read directly by Fluent5.

6.2.3 Creating two Grid objects from two msh-files, reading the solution for one of the grids from a dat-file, and from this generating an initial solution to the other grid

When there are two msh-files available, that have been generated by Gambit based on two different Pipe objects, and an iterative solution is found by Fluent5, for one of the meshes, then the wish might be to use the existing solution to generate an initial solution to the other mesh. If the initial solution can be put in a dat-file, Fluent5 can use it to get a convergent solution for the mesh, with less iterations.

To enable the generation of the initial solution the following main file can be used.

```
int main(int argc,char **args)
{
   int no_faces_per_cell_in=6;

   Pipe pipe1("pipe1_in");
   Grid grid1(pipe1, no_faces_per_cell_in);

   grid1.scanGridMSH("pipe1.msh", no_faces_per_cell_in);
   AllDataToGrid alldata_grid1(grid1);
   alldata_grid1.readAllSolutionsFromDatFile("pipe1.dat");

   Pipe pipe2("pipe2_in");
   Grid grid2(pipe2, no_faces_per_cell_in);
   grid2.scanGridMSH("pipe2.msh", no_faces_per_cell_in);
   AllDataToGrid alldata_grid2(grid2);
   alldata_grid2.moveAllSolutionsToOtherGrid(alldata_grid1);
   alldata_grid2.printAllSolutionsToDatFile("pipe2.dat");
   return 0;
}
```

First the Pipe object (`pipe1`), corresponding to the mesh, with the calculated solution, is created. Then the Grid object (`grid1`) that will carrying the information about this mesh is created. The mesh information is given to the grid object by the function `scanGridMSH`, which reads the msh-file `pipe1.msh` that Gambit has created. After this the AllDataToGrid object that will contain all solution values for the mesh is created, and the solution values are set by the function `readAllSolutionsFromDatFile`.

Next the other Pipe and Grid objects are created, and set in the same way. Also a new AllDataToGrid object is created. The solution values of this object is set by the function `moveAllSolutionsToOtherGrid`, which uses some of the solution values of the other AllDataToGrid object. Finally the data for grid2, is printed to a dat-file, by using the function `printAllSolutionsToDatFile`.

6.3 Description of the four classes and their functions

In this section a detailed description follows, of the functions, the program consists of, to enable the making of further main-files.

6.3.1 The Pipe class and its functions

To create a Pipe object, the parameters, centre curve and ab-curve, the nurb curve describing the radius of the duct at the four cross sections of the duct, which are the private member variables, have to be initialised. This is done by setting the control vertices and the knot vector for each nurb curve, see Section 5.1. This can be done through one of the constructors described below, which also calculates the base functions and initialises the nurb curves.

`Pipe::Pipe()` Constructor that initialises the private variables by setting the control vertices to standard values.

`Pipe::Pipe(char *file)` Constructor that reads the coordinates of the control vertices from a file, written by the user, and initialises the private variables centre curve and ab-curve. The file that provides the information about the coordinates, should first set the four three dimensional control vertices on the centre curve, and then the four control vertices for the ab-curve, the nurb curve describing the radius of the duct at the four cross sections of the duct.

`Pipe::Pipe (const double x_coord_in[], const double y_coord_in[], const double z_coord_in[], const double a_coord_in[], const double b_coord_in[])` Constructor that takes the control vertices as arguments and initialises the private variables centre curve and ab-curve. This constructor takes vectors with four elements, that is four control vertices are used to set the nurb curves.

`Pipe::Pipe(const Pipe& pipevolume)` Constructor that creates a Pipe object by copying the Pipe object in the argument.

`void Pipe::operator = ( const Pipe& pipe )` Copies a Pipe object to a new Pipe object, by copying all the private variables of the object.

`Pipe::~Pipe()` Default destructor.

To get the information, stored in the private variables, there is a need for functions that return these variables. For the Pipe class these functions are:

`NurbCurve& Pipe::getCentercurve()` Returns a reference to the centre curve which is of the type NurbCurve, which is a parametric description of the bending of the waterjet duct.

`NurbCurve& Pipe::getABCurve()` Returns the ab-curve that holds the information about the radius of each elliptic cross section.

`void Pipe::evalCentercurve(double par_value, double coord[], double normal[], double binormal[], double tangent[], int jderiv)` Evaluates the Cartesian coordinates, the normal, the binormal and the tangent of the centre curve at the parametric value `par_value`.

When a Pipe object is made the information about the geometry can be converted to the format that Gambit requires by printing a journal file. There are two different print functions which are specified below:

`void Pipe::printVolume(int no_boundary_pts_in, char *file, int jderiv, int no_pts_on_ellipse_in, double a_dist_inner_volume, double b_dist_inner_volume)` Generates a journal file with only information about the geometry of the Pipe object that the function is used for. The number of boundary points, along the walls, and the number of points on the inlet and outlet ellipses are given as arguments. Also the name of the file created by this function is set by the user and the proportions of the inner and outer volumes are set by the last parameters in the argument list.

`void Pipe::printStructMeshedVolume(int no_boundary_pts_in, char *file, int jderiv, int no_mesh_intervals_each_part_ellipse_in, int no_mesh_intervals_length_in, int no_pts_on_ellipse_in, int no_mesh_intervals_short_edge, double a_dist_inner_volume, double b_dist_inner_volume)` Generates the same kind of journal file as printVolume, but this journal file also includes the mesh. Apart from the parameters mentioned earlier for printVolume this function also takes the number of mesh intervals on the inlet and outlet ellipse, on the outer walls and on the edges in the inner structure of the duct.

In addition to the functions described earlier and functions that return the class members, the class also has member functions that converts coordinates from Cartesian coordinates to the parametric coordinates described in Section 5.1 and from parametric coordinates to Cartesian coordinates. These functions are:

`void Pipe::changeXYZCoordToSRTheta(double x_value, double y_value, double z_value, double& s, double& r, double& theta)` Converts the Cartesian coordinates, given as argument, to the parametric coordinates also given in the argument list. How the conversion is done more specifically can be read in Section 5.2.

`void Pipe::changeSRThetaCoordToXYZ(double& x, double& y, double& z, double s_value, double r_value, double theta_value)` Converts the parametric coordinates to Cartesian coordinates. All these parameters are given as arguments to the function. How the conversion is done more specifically can be read in Section 5.2.

These functions are primarily used when working with different meshes. The use of parametric coordinates enables a comparison of cells in different geometries.

6.3.2 The Grid class and its functions

The class Grid has private member variables that holds information about a mesh. Each mesh generated by the program corresponds to a pipe geometry, and therefore the Grid class also has the corresponding Pipe geometry, as as a member variable. This because the handling of the grid demands knowledge about the Pipe geometry. The member functions consist of functions, which creates the Grid object, reads the mesh and functions that handle and compare different grids.

The functions creating the Grid objects (the constructors) are all rather different. Some are quite straight forward as those in the Pipe class but there are also more complicated. Below all constructors are described.

`Grid::Grid()` Creates a default grid by setting the private variables to default values.

`Grid (const Grid& grid_1)` Creates a Grid object by copying an already existing Grid object.

`Grid::Grid(int no_nodes_per_cell_in, int no_faces_per_cell_in, int no_nodes_per_face_in)` Creates a Grid object for which not all variables are set to default values. The parameters in the argument are set by the user.

`Grid::Grid (Pipe& pipe_1, int no_faces_per_cell_in)` Constructor, where the private variables, pipe geometry and `no_faces_per_cell`, are set.

`Grid::Grid(Grid& grid_old_geometry, Pipe& new_geometry)` Constructor that creates a complete, new Grid object, with the geometry parameter set to `new_geometry` and the mesh is based on the mesh in the Grid object `grid_old_geometry`. All private variables except the pipe geometry and the coordinates of the nodes are copied from the old Grid object. The coordinates of the nodes in the new grid are given the same parametric coordinates as the nodes in the old grid. This constructor uses the Pipe functions "changeXYZCoordToSRTheta" and "changeSRThetaCoordToXYZ", to map the values of the coordinates of the nodes.

`void Grid::operator = ( const Grid& grid )` Copies a Grid object to a new Grid object, by copying all the private variables of the object.

`Grid::~Grid()` Default constructor.

Most of the constructors do not create complete Grid objects, because most variables are just set to default values.To get the information, about how the mesh is built, there is a need for a tool that reads the mesh files that Gambit, the preprocessor to Fluent5, creates, when the mesh is generated in Gambit. The following function does just this.

`void Grid::scanGridMSH ( const char *file, int no_faces_per_cell_in)` Reads the mesh from the msh-file that Gambit generates from the journal file. This function sets the private variables, such as the grid size where the nodes are situated and how the different cells are built from the nodes. In the msh-file first the coordinates of the nodes are set. Then faces are defined, through a number of nodes (four nodes for hexahedral elements and three for tetrahedral elements). The two cells (only one cell for boundary faces) sharing the face are also given here. The function sorts this information into different matrices. One private variable is the "cellmatrix", where the faces that constitute the cells are stored for each cell. Another is the matrix "cellfaces", containing the node numbers constituting each face. Also there are three vectors where the information about the coordinates (x,y,z) of the nodes are stored, respectively.

Next comes the group of functions that returns information about the cell structure, which more or less directly returns the private variables.

int Grid::getNoCells() const Returns the number of cells in the grid.

int Grid::getNoFacesPerCell () const Returns the number of faces a cell has in the grid.

Pipe& Grid::getPipeGeometry() Returns a reference to the Pipe object that is a member variable of the Grid object.

int Grid::getNoNodes()const Returns the number of nodes in the grid.

int Grid::getNoFaces()const Returns the number of cell faces in the grid.

MVColMat<int>& Grid::getNodesOnFace() Returns the matrix that holds information about which node numbers the faces that constitute the cells in the grid, have.

MVColMat<int>& Grid::getCellMatrix() Returns the cellmatrix where the face numbers for each cell are listed.

double Grid::getNodeXCoord(int i)const Returns the x-coordinate for node i.

double Grid::getNodeYCoord(int i)const Returns the y-coordinate for node i.

double Grid::getNodeZCoord(int i)const Returns the z-coordinate for node i.

MV_Vector<double>& Grid::getNodeXCoord() Returns the vector which contains the x-coordinates for all nodes.

MV_Vector<double>& Grid::getNodeYCoord() Returns the vector which contains the y-coordinates for all nodes.

MV_Vector<double>& Grid::getNodeZCoord() Returns the vector which contains the z-coordinates for all nodes.

`MV_ColMat<int>& Grid::getCellFaces()` Returns the matrix which holds the information about which cells that share the the same face. For each face, but the boundary faces, there are two cells sharing each face.

`MV_ColMat<int>& Grid::getIdZonesFaces()` Returns information about how the faces are sorted into identification zones. These zones are created by Fluent5, and are needed to correctly communicating the grid information back to Fluent5.

`int Grid::getNoIdZonesFaces()` Returns the number of identification zones that the faces in the grid consists of.

The next group of functions are those that are used to get more complicated information about the grids. For example these functions are used to identify related cells in two different grids, by looking at the parametric values of the centre points of the cells. For those functions where it is not obvious how they work there is a short description of the algorithm.

`void Grid::getSortedNodesForCell(int cellnumber, MV_Vector <int>& nodes_reordered)` Returns the node numbers for a cell in the same order as a reference cell, see Figure 10. First the nodes for two opposite faces

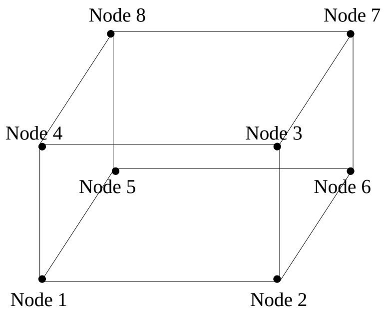

Figure 10: *A reference cell, showing the order of the nodes, that is required by the function `Grid::checkIfNodeIsInCell`*

in the cell are chosen. Then by looking at the sign of the volume of the tetrahedron that is created by three nodes from one of the opposite faces, and the centre point of the cell, the order is decided. If the sign is negative the original order is changed. By looking at one of the connecting faces between the opposite faces the nodes on the other opposite face, that is situated at the same place as the first node on the first face can be identified, and the correct start node is found. By looking at the corresponding tetrahedron for this face the order is set. This function is used in the function `checkIfNodeIsInCell`, where a local coordinate system is used, and the sorted nodes are therefore needed.

`void initSortedNodesForCell()` Function which initialises the private variable `cellnodes`, which is a matrix containing the sorted nodes for all cells.

`int Grid::getCellNodesCoord(int cellnumber, MV_Vector <double>& x_coord, MV_Vector<double>& y_coord, MV_Vector <double>& z_coord, MV_Vector<int>& nodes)` Returns the node numbers for one cell, and the coordinates of these nodes.

`int Grid::getNeighbourCells(int cellnumber, MV_Vector<int>& neighbourcells)` Returns the neighbour cells in a vector "neighbourcells" for the cell with the number "cellnumber". Apart from the arguments that are handled, the function returns the number of neighbour cells of the cell. The neighbour cells are found by looking at which other cells that share the faces that constitute the cell.

`void removeCopies( MV_Vector<int>& vec, int& size)` Function which removes any existing copies in the vector `vec` that is given in the argument. The size of the vector `size` must be given when using the function. Both the changed vector and size are returned by the function

`void Grid::getParCoordforCell(int cellnumber, MV_Vector <double>& s_array, MV_Vector<double>& radius_array, MV_Vector <double>& theta_array)` Returns the parametric coordinates of the cell nodes. Each of the arrays returned has the same number of elements as the number of nodes constituting the cell. This function uses the Pipe function changeXYZCoordToSRTheta.

`void Grid::getAllCellCentreParCoord(MV_Vector<double>& r_centre, MV_Vector<double>& s_centre, MV_Vector<double>& theta_centre)` Returns the parametric coordinates of the centre point for each cell. This function first calculates the geometric average of all the nodes, to get the centre point for each cell. It then uses the Pipe function changeXYZCoordToSRTheta to get the corresponding parametric coordinates for the centroid of the cells.

`void Grid::getCellCentreCoord(int cellnumber, double& x_centre, double& y_centre, double& z_centre)` Returns the x, y and z coordinates for the cell centre point for the cell with cell number ”cellnumber”, by calculating the geometric average of the nodes.

`bool Grid::outsideBoundingBox(int e, double x, double y, double z)` Function that creates a bounding box from the maximum x, y and z values of the nodes of the cell, with number e. The function evaluated if the point given by the x, y and z values that are given as argument is situated in the bounding box. If it is not true is returned otherwise false. The function is used to in an early stage sort out points that definetly is not located inside the cell.

`int Grid::checkIfNodeIsInCell(double x, double y, double z, int cellnumber, Grid& other_grid)` Returns true (=1), if the node, with the coordinates x, y and z, lies in the cell with cell number ”cellnumber” in the grid ”other_grid”. If the node does not lie in the cell the function returns false (=0). This function can be used when cells in two different grids are compared. By converting the coordinates of the node to parametric coordinates and then to Cartesian coordinates using the geometry of the other grid. If the parametric coordinates that are given by the x, y, z coordinates lie outside the parametric space, the function returns -1. By using the function `outsideBoundingBox` in the beginning of the function, nodes that definetly are not inside the cell can quickly be checked, without using a lot of computer power.

`int Grid ::checkIfNodeIsInCell(double x, double y, double z, int cellnumber)` Returns true (=1) if the node, given by the coordinates x, y and z, is situated inside cell number `cellnumber`. This function can only check nodes located in the same geometry as the cell, opposite to the function above.

`double Grid::calculateMinDistInGrid(Grid& other_grid, double& r_max)` Returns the minimum distance between two centre points in two neighbour cells. Both the grid in the argument and the grid calling the function are checked, so the smallest distance in both the grids are returned. At the same time all r-coordinate for the centre points are checked and if there are points, with the radius greater than one, the largest one of these are stored. The r_max and the minimum distance, returned by the function, are used to design the size of the help grid.

`void Grid::relateCellsToStructuredHelpGrid (MVColMat<int>& help_cell_for_cell_in_grid, Array3d<int>&`

`cell_in_grid_for_helpcell, double min_dist, double r_max)` Relates the cells in one grid to a structured help grid, where the grid size is decided by the minimum distance from the function calculateMinDistInGrid, and the maximum value of the radius is set by the value of r_max, which comes from the same function . By the parametric value of the centre point in each cell, the cell is given a help grid cell. This information is returned in the matrix `help_cell_for_cell_in_grid`. When one cell has matched one cell in the help grid this information is also stored for these cells in the help grid in the matrix `cell_in_grid_for_help_grid`. Because the help grid normally has a larger number of cells, some of the cells in the help grid does not match any cells in the real grid. To enable the use of the help grid, those cells that do not relate to a cell are set to relate to the same cell as one of its neighbour help cells.

`int Grid::getRelatedCellsInOtherGrid(Grid& other_grid, MV_Vector<int>& cells_other_grid)` For each cell the function returns the corresponding cell in the other grid, by using the function relateCellsToStructuredHelpGrid on both the grids, and use these help grids as a reference between them. Note that the help grids are of the same size. The function also checks that the centre point in the cell really lies in the cell found in the other grid, by also using the function checkIfNodeIsInCell. If the node is not in the cell referred to by the function relateCellsToStructuredHelpGrid, the function also checks if the centre point lies in the neighbour cells of the cell referred to in the other grid. If the right cell is not found among the six first layers of neighbours, the function `loopOtherGridToFindRelatedCell` is used. If the centre point does not lie in any cell in the other grid, the cell given by the help grid is taken, alternatively one of its neighbours, depending on which cell has the smallest distance to the center point. For further information about how the related cell is found see Section 5.3.

`int Grid::getRelatedCellsInOtherGrid2(Grid& other_grid, MV_Vector<int>& cells_other_grid)` For each cell the function returns the corresponding cell in the other grid, by using the function relateCellsToStructuredHelpGrid on both the grids, and use these help grids as a reference between them. The function also checks that the centre point in the cell really lies in the cell found in the other grid, by also using the function checkIfNodeIsInCell. If the node is not in the cell referred to by the function relateCellsToStructuredHelpGrid, the function also checks if the centre point lies in the neighbour cells of the cell referred to in the other grid. If the right cell is not found among the six first layers of neighbours, the function `loopAllCellsToFindClosestCentroidInOtherGrid` is used. For further information about how the related cell is found see Section 5.3.

int Grid::getRelatedCellsInOtherGrid3(Grid& other_grid, MV_Vector<int>& cells_other_grid) For each cell the function returns the corresponding cell in the other grid, by using the function relateCellsToStructuredHelpGrid on both the grids, and use these help grids as a reference between them. The function also checks that the centre point in the cell really lies in the cell found in the other grid, by also using the function checkIfNodeIsInCell. If the node is not in the cell referred to by the function relateCellsToStructuredHelpGrid, the function also checks if the centre point lies in the neighbour cells of the cell referred to in the other grid. If the right cell is not found among the six first layers of neighbours, the cell given by the help grid is taken, alternatively one of its neighbours, depending on which cell has the smallest distance to the center point. For further information about how the related cell is found see Section 5.3.

int Grid::loopOtherGridToFindRelatedCell(Grid& other_grid, int cellnumber, int& cellnumber_other_grid) Loops all cells in the grid other_grid, to find the related cell (cellnumber_other_grid) to the cell with number cellnumber.

void Grid::loopOtherGridToFindRelatedCells(Grid& other_grid, MV_Vector<int>& cells_other_grid) Loops all cells in the grid other_grid, to find the related cell to each cell in the grid that the function is called for. The related cells are returned in the vector cells_other_grid

int Grid::loopAllCellsToFindClosestCenteroidInOtherGrid(double x_centre, double y_centre, double z_centre, Grid& other_grid) Checks all cells to find the cell, in the other grid, which has the centroid located closest to the transformed centroid.

void Grid::changeDirectionOfSolutionVectorToFitOtherGrid (int cellnumber, double solution_x, double solution_y, double solution_z, Grid& other_grid, int cellnumber_other_grid, double& new_solution_x, double& new_solution_y, double& new_solution_z) Adjustes the direction of a vector moved from one mesh to another, to better fit the new location. The cellnumber is the cell where the vector is taken from, and cellnumber_other_grid is the cell where it is put in the grid other_grid.

void Grid::copyGridToNewGeometry(Grid& grid_old_geometry, `Pipe& new_geometry)` Makes a new grid to a Pipe object (`new_geometry`), by copying a grid (`grid_old_geometry`) corresponding to another Pipe object and by stretching the grid to fit the new geometry. What is done is that the coordinates of the cell nodes are changed by setting the nodes in the new grid to have the same parametric coordinates, and then converting the coordinates to Cartesian coordinates to fit the format in Fluent5.

When the information about the mesh shall be communicated to Fluent5, there is a need to convert the information into the format that Fluent5 requires, this is done by the following function.

`void Grid::printMSHfile(const char *mshfile)` Prints the information of a Grid object to a mesh (msh) file, so the mesh can be read by Fluent5.

6.3.3 The DataToGrid class and its functions

The DataToGrid class holds the solution values for one of the solution types, that is given by Fluent5 in the dat-file. In addition to the solution values also the type and the size of the solution are variables. The size is a scalar (size=1) or a vector (size=3). Furthermore there is a variable for the Grid for which the solution was found. The functions that are members to the class are constructors or functions that reads or writes files, with the solution values, coming from or going to Fluent5. There are also functions that moves a solution to another grid, and of course also the usual functions that returns or sets the private variables of the class. First all the constructors are presented, and then the functions returning and setting the private variables are described.

`DataToGrid::DataToGrid ()` Constructor that generates a DataToGrid object, and sets the solution values to zero.

`DataToGrid::DataToGrid(Grid& grid_in, int id_type_in, char file)` Constructor that generates a DataToGrid object, and sets the grid for the object.

`DataToGrid::~DataToGrid()` Default destructor.

`MV_ColMat<double>& DataToGrid::getSolutionValueCells()` Returns all the solution values in a vector.

`double DataToGrid::getSolutionValue(int cellnumber, int vectornumber)` Returns the solution value for the cell with cellnumber `cellnumber`. The integer `vectornumber` tells the function, which element in the solution vector it should return. The integers possible to give to the function are 0, 1 and 2.

`int DataToGrid::getIdType()` Returns the identification number of the solution field, that Fluent5 uses to recognize the solution type.

`int DataToGrid::getNoCells()` Returns the number of cells in the grid that the solution belongs to.

`int DataToGrid::getSize()` Returns the size of the solution. Size=1 if the solution is a scalar and size=3 if it is a vector.

`void DataToGrid::setSolutionValueCells(int cellnumber, double value)` Sets a scalar solution value for cell number `cellnumber` to `value`.

`void DataToGrid::setSolutionValueCells(int cellnumber, double value_1, double value_2, double value_3)` Sets the three different elements in the solution vector for cell number `cellnumber` to `value_1`, `value_2` and `value_3`.

`void DataToGrid::initDataToGridObject(Grid& grid_in, int id_type_in)` The function initialises the private variables in the DataToGrid object, according to the identification number `id_type_in`, and the grid variable is set to `grid_in`.

Next the member functions that reads data files, writes data files and moves solution values from one DataToGrid object to another are described.

`void DataToGrid::scanCellDataToGrid(Grid& grid_in, int id_type_in, char *file)` Reads the data file that Fluent5 has created after solving the fluid flow problem, and stores the data for the solution type, with the identification number `id_type_in`. The function also checks that the grid size corresponds to that for the grid given in the argument `grid_in`, to make sure that the correct data file has been read.

`void DataToGrid::moveSolutionToNewGrid(DataToGrid data_old_grid, MV_Vector<int> cells_other_grid)` The function sets the solution values of a new DataToGrid object, by taking the solution values, from `data_old_grid`, that correspond to the cells given in the vector `cell_sother_grid`. This vector is created by using the function `getRelatedCellsInOtherGrid`, which is a member of the Grid class. When solutions with size=3 are moved the direction of the solution vector is adjusted in the way explaned in Section 5.3.2.

`void DataToGrid::printCellDataToGridToDatFile(char *file)` The function writes the solution values of the DataToGrid object to a dat-file with the name *file.

6.3.4 The AllDataToGrid class and its functions

In the class AllDataToGrid all the types of solutions that belongs to one grid is gathered to be treated together. All the types of solutions that are available in the data file from Fluent5 that are solutions set at the centre of the cells are pressure, turbulent kinetic energy (tke), rate of dissipation of turbulent kinetic energy (ted), body forces, density, laminar viscosity (mu-lamb), turbulent viscosity (mu-turb), x-velocity, y-velocity and z-velocity. These are the private variables of the AllDataToGrid class, and all these variables are DataToGrid objects. The last private variable of the AllDataToGrid class is the grid that the solutions belong to. Below the member functions are described.

`AllDataToGrid::AllDataToGrid()` Constructor that creates a AllDataToGrid object without setting any private variables.

`AllDataToGrid::AllDataToGrid(Grid& grid_in)` Constructor that creats an AllDataToGrid object and uses the DataToGrid function `initDataToGridObject` to set the private variables.

`AllDataToGrid::~AllDataToGrid()` Default destructor

`void AllDataToGrid::readAllSolutionsFromDatFile(char *file)` Reads the data file, that Fluent5 has created after solving the fluid flow problem, and stores all the solution values for the types that correspond to the private variables of the AllDataToGrid class.

`void AllDataToGrid::moveAllSolutionsToOtherGrid(AllDataToGrid solutions_old_grid)` The function uses the DataToGrid function `moveSolutionToNewGrid` to move all the solution values, except those corresponding to the velocities. This, because the values of the x, y and z velocities should be treated together as a vector, and not separately as scalar values. If all the velocity components are treated as a unity, the direction of the vector can be adjusted to fit the geometry of the AllDataToGrid object that the solution is moved to. This is done in the same way as it is done for the DataToGrid objects that are of size=3 in the function `moveSolutionToNewGrid`.

`void AllDataToGrid::printAllSolutionsToDatFile(char *file)`
The function prints all the solution values of the private variables to a data file, that Fluent5 can read, so that a grid can be given an initial solution.

7 Computational example of a waterjet duct

This example will show how a solution can be mapped from one mesh to another, in order to generate an initial solution for the mesh. The tools that are used are the existing main-files, Gambit and Fluent5.

First the geometry of the first pipe is set. This is done by making a geo-file, containing the control vertices for the two nurb curves, defining the geometry of the pipe. When this is done, the main file presented in Section 6.2.1, is used to create the pipe object, and generate the journal file, that can be read by Gambit. The number of mesh intervals on the edges that describe the length of the pipe are set to be 60, 15 on all four parts of the inlet and outlet ellipse, and 10 on the small edges defining the inner volume.

The journal file is then run in Gambit, where the geometry and the mesh is created. The mesh is exported in Gambit to a msh-file. Figure 11 shows the generated mesh.

When Fluent5 has read the msh-file, the boundary conditions and the the characteristics of the fluid are set. The fluid flow problem is then given an initial solution at the inlet. This is done by setting the size and the direction of the velocity at the inlet. After this the iterative process of finding a numerical solution starts. Approximately 50 iterations are made, using the stable first order upwind scheme. After these iterations the scheme is changed to second order upwind scheme, which is not as stable as the first order scheme, but will yield a more accurate. The stability is needed in the first iterations, but as the solution becomes better the solution also becomes more robust. Further iterations are made until the solution converges. Ap-

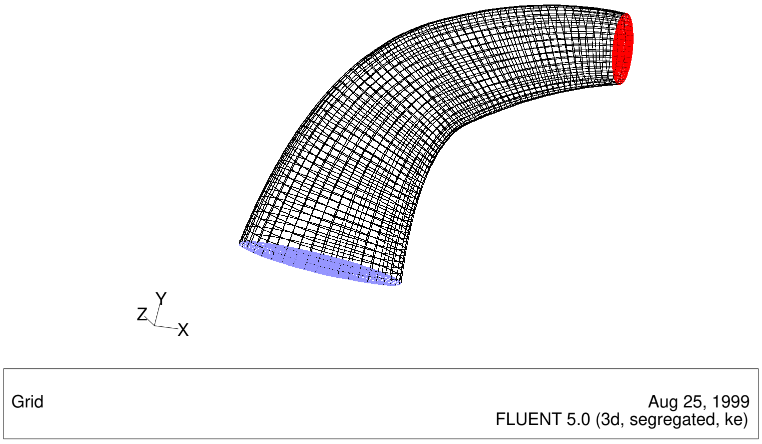

Figure 11: *The meshed geometry generated in Gambit, shown in Fluent5.*

proximately 200 iterations are required in this case. The solution is stored in a dat-file, which Fluent5 writes. The solution of the velocity is visualised, for the plane corresponding to a cross section ($z = 0$) in the length direction of the duct, in Figure 12, using velocity vectors. The pressure at the wall, where the velocity is zero is visualised in Figure 13

A second geometry is now initialised by making another geo-file, with other control vertices. The same main-function is used as before, to read the geo-file, make the new Pipe object and generate a new journal file to Gambit. The mesh parameters are this time set to 80, 16 and 12. That is, the new geometry will be meshed almost twise as fine.

The journal file is as before run in Gambit, and the mesh is exported to a msh-file. Figure 14 shows the mesh of the new geometry

Now the process of comparing the two meshes begin. The main-function described in Section 6.2.3 is used for the whole process. First the information about the geometry is refreshed, by again creating the Pipe objects, and the information about the two meshes is also stored by the program by reading the msh-files and putting the information in Grid objects. The existing data file is read and the solution values are put in a AllDataToGrid object. Another AllDataToGrid object is also created to hold the initial solution for

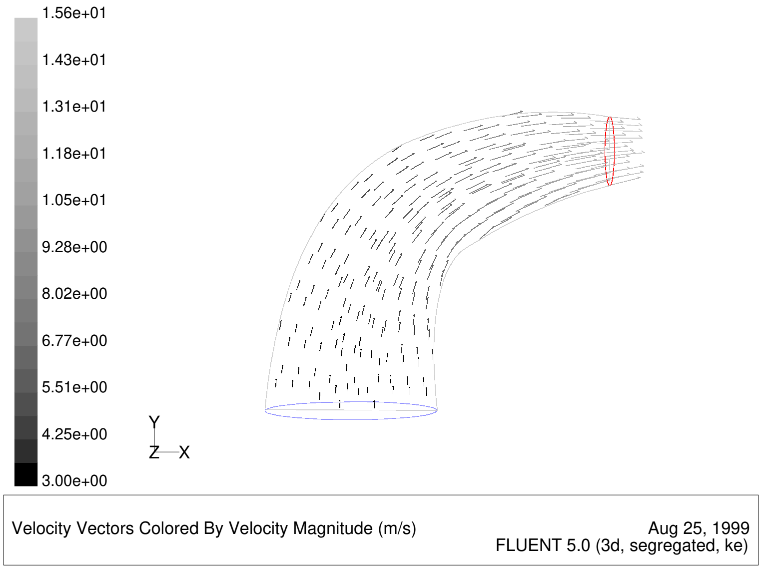

Figure 12: *The velocity vectors in the waterjet duct, in the a cross section plane* $z = 0$ *in the length direction, after (208 iterations) solving the fluid flow problem for the first mesh.*

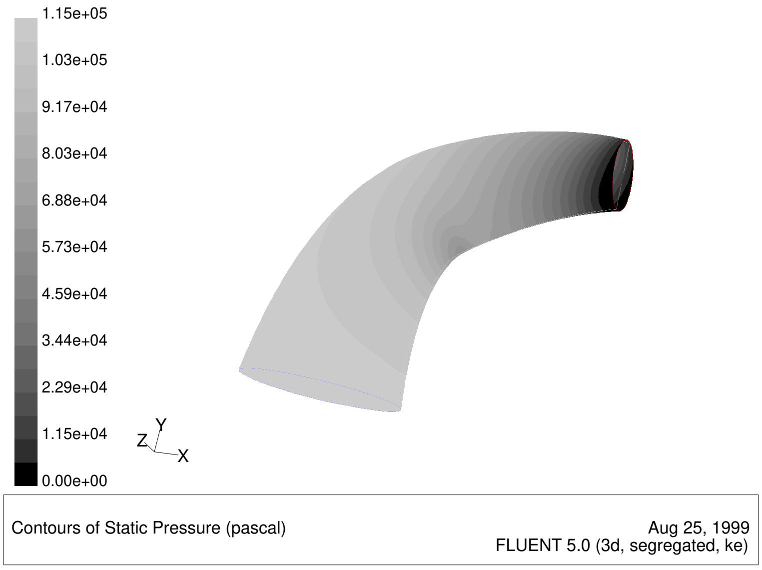

Figure 13: *The static pressure in the waterjet duct, at the boundary where the velocity is zero, after (208 iterations) solving the fluid flow problem for the first mesh.*

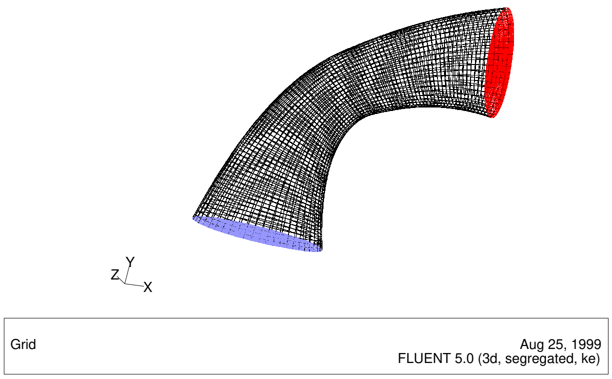

Figure 14: *The figure shows the meshed geometry of a second Pipe object.*

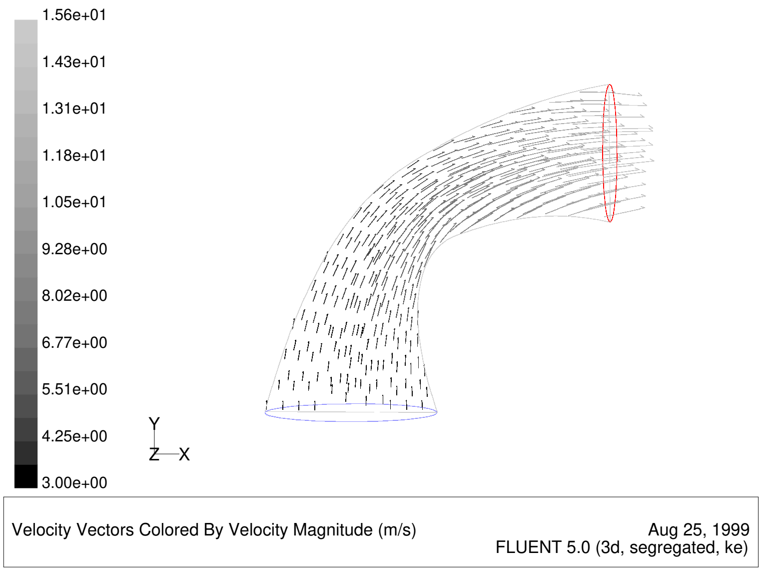

Figure 15: *The figure shows the moved and patched initial solution for velocity, in a cross section plane* $z = 0$ *in the length direction of the duct, for the new mesh.*

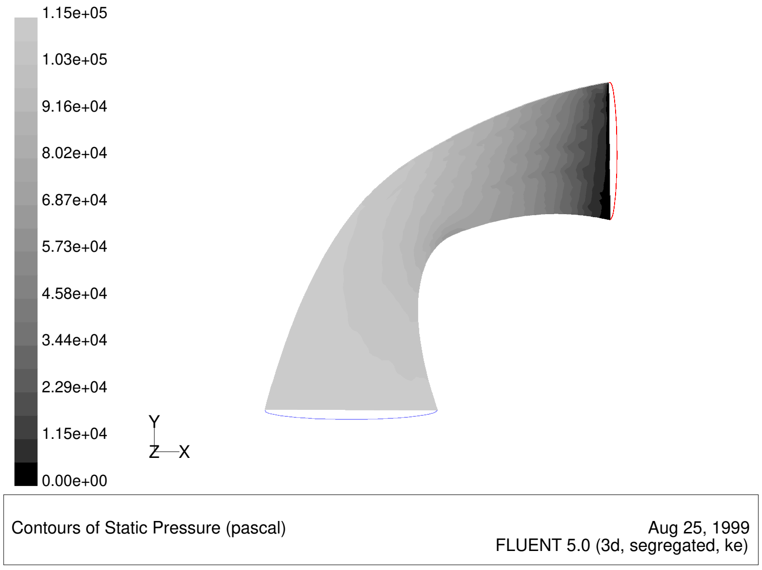

Figure 16: *The figure shows the moved and patched initial solution for the static pressure, for the new mesh.*

the new mesh, and the solution is finally moved to this object. The program stores the initial solution in a dat-file, which can be read by Fluent5. By using a function called "Patch" in Fluent, the face values of the cells in the mesh are created. These values are calculated from the centroid values by predefined functions. The new mesh, with the new patched initial solution for velocity can be seen in Figure 15, and the static pressure can be seen in Figure 16. A converged numerical solution for the new mesh is obtained in the same way as for the other mesh, but now the dat-file with the patched initial data is read before the iterations are made. Unfortunately enough the solution does not converge as fast as assumed. The possible reasons for this is discussed below.

For a comparison, the solver is also run for the new mesh, with a zero initial solution for a great number of iterations. The solution converges short after 300 iterations. Totally 516 iterations are made, and Figure 17 and 18 shows the plots of the velocity field, and the static pressure field. When visually comparing the mapped and patched initial solution to the convergent solution, the similarity of the solution fields is easy to see. Looking at the velocity field, it is clear that the vectors in most places are directed correctly. The pressure is however not correctly suited for the new geometry, where the most narrow passage is situated half way through the pipe. In the previous

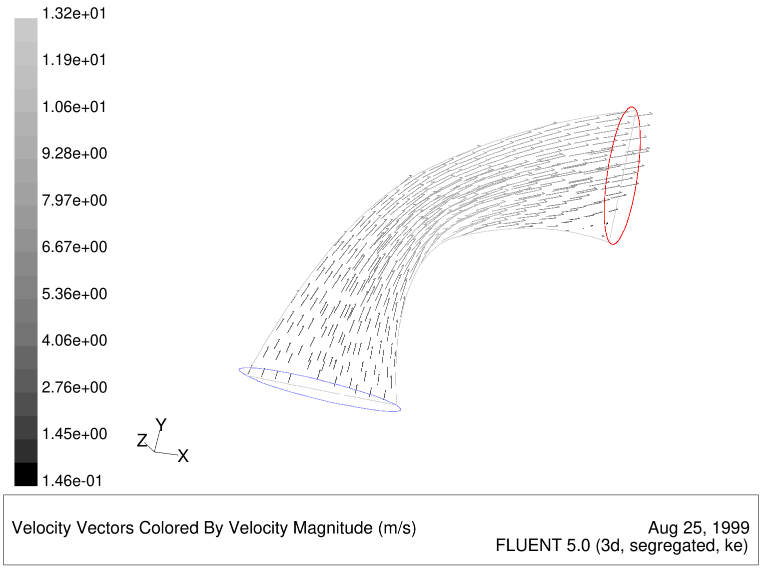

Figure 17: *The figure shows the converged velocity solution for the new mesh in the z=0 plane. The solution was calculated with zero initial data, and 516 iteration was made.*

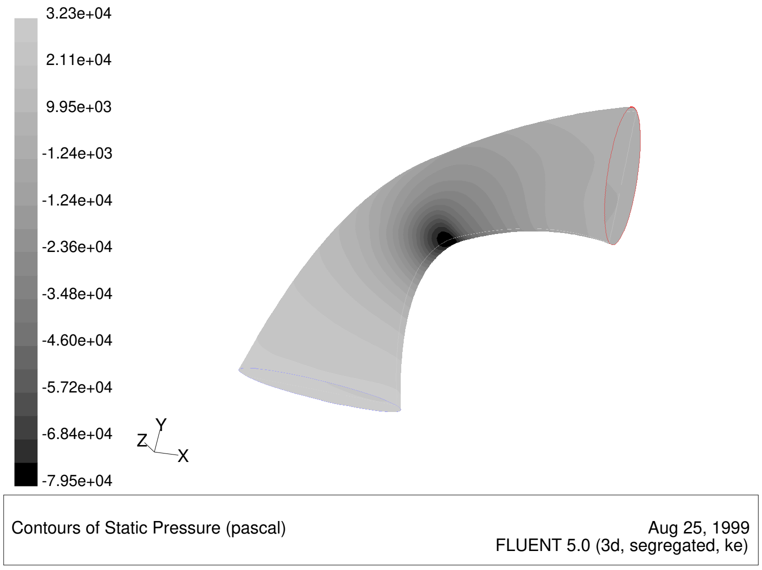

Figure 18: *The figure shows the converged velocity solution for the new mesh on the surface defined by velocity=0. The solution was calculated without initial data, and 516 iteration was made.*

geometry the most narrow passage of the duct was the outlet. The velocity in this region is therefore highest. The initial solution for the new geometry uses this pressure field as the starting field. This distribution will however change quite a lot.

Tests have also shown that Fluent5 converges faster if also the faces of the cells are initialised. The patch function helps the solution to converge, but the result would have been better if the face values could be mapped in the same way as the centroid values. This seem a bit strange, because the fact that Fluent5 calculates the face values from the centroid values, but the answer to why this is so can probably be found if the algorithm that Fluent5 uses is investigated.

8 Conclusions

The main result of the work is the program files containing all the functions. The main-functions created fulfils the specifications that was put on the work, so in this sense the programs are successful. A parametric geometry of the waterjet duct is generated and the information about the geometry is communicated to Gambit, which creates the mesh. Further, the mesh can be handled and mappings are made between different geometries and meshes. Finally, initial data is given to a mesh, by using existing data from another mesh.

However, the general purpose of the program is to make the fluid flow problem in the waterjet duct to be solved with less iterations, by giving the mesh a good initial solution. Unfortunately no thorough investigation of how fast the solution generally converges, if the initial solution is given this way has been made, due to lack of time. A limited number of tests was performed and the calculations show that the moved solution, that is used as initial data, looks pretty good, when it is visualised in Fluent5. However so far the solution does not converge as fast as expected.

It has been established that the parametric model for the geometry works well, and is very useful, when comparing grids. The structured parametric help grid is also a successful tool in this process, and most often the reference given by the help grid is the corresponding cell, or in some cases one of its neighbours.

9 Discussion on possible further work

The purpose of this work was not to solve all problems, concerning the mapping of a solution optimally, and to get a program that works for all possible scenarios, of modelling the waterjet duct. The aim was to find one way of mapping a solution. All different parts of the program works, and the chain of processes that is needed to in the end generate the initial solution holds. The program is however far from a finished product.

Disregarding all the improvements that can be made on the actual code in the program files, a first step in proceeding this work is to further investigate how Fluent5 uses the information in the data file, and what happens between the iterations. By understanding this more accurate information can be given to Fluent5, and a lot would be gained. What has been seen already is that the solution values of the faces of the cell should be calculated and printed to the data file, where the initial solution is saved. There may also be further variables that Fluent5 is dependent of. One simple way of finding out, what Fluent5 really needs from the initial solution, is to take a data file of a convergent solution, and remove one variable from the data file at a time, and then test how fast the solution converges, when this data file is used as initial data.

The algorithm that finds the corresponding cell in another grid seems to be working well. As it is now the solution values that are given to the cells are just values taken from one corresponding cell in the other grid. This could be further developed to make averages of several cell values if the solution is mapped from a fine grid to a coarse grid, alternatively interpolation can be used, when the solution is moved from a coarse grid to a finer one.

References

[1] V.B Anand, Computer Graphics and Geometric Modelling for Engineers, Cambridge John Wiley & Sons, Inc., 1993.

[2] J.J Barton and L.R Nackman, Scientific and Engineering C++, An Introduction with Advanced Techniques and Examples, Addison Wesley Longman, Inc., 1994

[3] G.K. Batchelor, An introduction to Fluid Dynamics, Cambridge University Press, 1967.

[4] I.G. Currie, Fundamental Mechanics of Fluids, McGraw-Hill Book Company, 1974.

[5] J Liberty, Lär dig C++ på 3 veckor (Teach Yourself C++ In 21 Days), Pagina Förlags AB, 1998.

[6] H.K Versteeg and W. Malalasekera, An introduction to Computational Fluid Dynamics, The Finite Volume Method, Addison Wesley Longman Limited, Inc., 1995.

Personal Communication

[7] J. Lennblad, *Caran Automotive (Dynamics Department),* Gothenburg, Sweden, 1999.

[8] G. Seil, *Kamewa,* Kristinehamn, Sweden, 1999.