Automatic Boundary Recognition for Thermal Fluid-Structure Interaction

by Nicklas Linder
March 2009
Supervision: Plamen Pironkov M. Sc.
Abstract

The present Bachelor Thesis deals with the simulation of thermal fluid structure interaction, especially with the recognition of boundary conditions between the solid and the fluid. The first part introduces the theoretical background of the finite volume and finite element methods, which are needed to solve the fluid flow and the structural analysis numerically. In addition, the k-ε method for simulating turbulent flows and the transfinite interpolation for grid generation are presented. Afterwards, this work illustrates how these theories work together in the fluid structure interaction (FSI). When simulating an FSI, one has to consider the coupling technique. In this work, the weak coupling with a partition approach is applied to use the software FEAP\(^1\) for the structural part and FASTEST\(^2\) for the fluid flow. The coupling of these programs is provided by the software MpCCI\(^3\). Then the newly developed functions for recognising the boundary conditions automatically are introduced and explained. In the last part some complex simulations are presented validating the functionality.

---

1 Finite Element Analysis Program - http://www.ce.berkeley.edu/~rlt/feap/
3 Mesh-based parallel Code Coupling Interface - http://www.mpcci.de
Statutory declaration

I hereby declare that I wrote this thesis independently and without use of other sources than acknowledged. Passages taken literally or analogously from published or non published sources are marked as such. Drawings or figures in this thesis have either been created by myself or their source is given. This work has not been presented to an examination board in this or a related form previously.

Darmstadt, February 28, 2009
Contents

1 Introduction .................................................. 1
  1.1 Motivation ................................................ 1
  1.2 Objective and outline .................................... 1

2 Fundamentals ................................................ 3
  2.1 Governing equations for fluid mechanical problems .......... 3
  2.2 Governing equations for continuum mechanics problems ....... 4
  2.3 Governing equations for heat transfer problems .............. 5
    2.3.1 Linear Thermo Elasticity ............................ 6
  2.4 Finite Volume Method ...................................... 6
    2.4.1 Discretisation ....................................... 6
    2.4.2 Approximation of Surface and Volume Integrals ....... 7
    2.4.3 Discrete Transport Equation ......................... 8
    2.4.4 Treatment of Boundary Conditions ................... 9
    2.4.5 Algebraic System of Equation ....................... 10
  2.5 Finite Element Method ................................... 10
    2.5.1 Discretisation ....................................... 10
    2.5.2 Method of the Weighted Residuals .................. 11
    2.5.3 The Galerkin Method and the Resulting System of Equation .................. 11
  2.6 Solution of Linear Systems of Equations .................... 11
    2.6.1 Jacobi and Gauss-Seidel ............................ 11
  2.7 Computation of Turbulent Flows .......................... 12
    2.7.1 Characterization of Computational Methods ........... 12
    2.7.2 Statistical Turbulence Modelling ................... 12
    2.7.3 The k-ε Turbulence Model ......................... 13
  2.8 Transfinite Interpolation ................................ 13

3 Coupled thermal FSI with MpCCI, FEAP and FASTEST ............ 15
  3.1 Coupled Fluid-Solid Problems ............................ 15
    3.1.1 Modelling of Coupled Fluid-Solid Problems .......... 15
  3.2 Structure ............................................... 15
  3.3 MpCCI as Coupling Interface ................................ 16
    3.3.1 FEAP Structure .................................... 16
    3.3.2 Setup of the input files ............................ 16

4 Implemented functions ...................................... 19
  4.1 Installing the functions .................................. 19
  4.2 General Workflow ........................................ 19
  4.3 Implemented Functions ................................... 20
    4.3.1 Planar Coupling Surfaces ........................... 20
    4.3.2 Cylindrical Coupling Surfaces ...................... 20
    4.3.3 Coned Coupling Surfaces ........................... 21
    4.3.4 Freeform Coupling Surfaces ....................... 21
  4.4 Examples ................................................. 23
    4.4.1 Planar example ...................................... 23
    4.4.2 Cylindrical example ................................. 25
    4.4.3 Coned example ...................................... 28
    4.4.4 Freeform example .................................... 29
    4.4.5 Comparative Simulation ................................ 31
    4.4.6 Incident Flow on both Sides of a Plate ............... 33
# List of Figures

2.1 Notation if the CV .......................................................... 6  
2.2 Relation between coordinates and grid points in physical and logical domains. .......................... 13  
   (a) Quadrilateral CV with notations ............................................. 13  
   (b) Notations for neighboring CV .................................................. 13  
3.1 Simplified illustration of the work flow of MpCCI .......................................................... 16  
4.1 Computation of the expected distance for coned surfaces ................................................. 21  
4.2 Numbering of the box - Needed to define the TFI Surface ................................................. 22  
4.3 Example of a planar coupling surface .................................................................................. 23  
4.4 Results of the planar coupling: Temperature plot of the geometry in Kelvin - Slice in the x-z-plane through the middle of the geometry .................................................. 24  
4.5 Example of a polar geometry ......................................................................................... 25  
4.6 Results of the tubeflow - Temperature plot in Kelvin ......................................................... 27  
4.7 Geometry and results of the coned example ....................................................................... 28  
4.8 Example of a freeform geometry ....................................................................................... 29  
4.9 Results of the tube .............................................................................................. 30  
4.10 Geometry - Solid diffuser with fluid inside ....................................................................... 31  
4.11 Results of the tubeflow with diffuser - Temperature plot in Kelvin ........................................ 32  
4.12 Geometry - Incident Flow an a Plate ............................................................................. 33
List of Tables

4.1  List of all needed files ................................................................. 19
# List of Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Standard coupling material</td>
<td>16</td>
</tr>
<tr>
<td>3.2</td>
<td>MpCCI definitions</td>
<td>17</td>
</tr>
<tr>
<td>3.3</td>
<td>a Macro for FSI</td>
<td>17</td>
</tr>
<tr>
<td>3.4</td>
<td>Pathdefinitions for FEAP</td>
<td>18</td>
</tr>
<tr>
<td>4.1</td>
<td>Input file for planar coupling faces</td>
<td>23</td>
</tr>
<tr>
<td>4.2</td>
<td>Input file for cylindrical coupling faces</td>
<td>25</td>
</tr>
<tr>
<td>4.3</td>
<td>Input file for freeform coupling faces</td>
<td>28</td>
</tr>
<tr>
<td>4.4</td>
<td>Input file for freeform coupling faces</td>
<td>29</td>
</tr>
<tr>
<td>4.5</td>
<td>MACRo for the tubeflow with diffuser</td>
<td>31</td>
</tr>
<tr>
<td>A.1</td>
<td>FASTEST id-file for the tubeflow with diffuser</td>
<td>39</td>
</tr>
<tr>
<td>B.1</td>
<td>FEAP Input File for the Incident Flow on a Plate</td>
<td>43</td>
</tr>
<tr>
<td>C.1</td>
<td>umacr6.F</td>
<td>47</td>
</tr>
<tr>
<td>D.1</td>
<td>umacr7.F</td>
<td>49</td>
</tr>
<tr>
<td>E.1</td>
<td>boundary8brickelem.F</td>
<td>53</td>
</tr>
<tr>
<td>F.1</td>
<td>definesurfcbc.F</td>
<td>57</td>
</tr>
<tr>
<td>G.1</td>
<td>definepolcbc.F</td>
<td>61</td>
</tr>
<tr>
<td>H.1</td>
<td>definediffcbc.F</td>
<td>65</td>
</tr>
<tr>
<td>I.1</td>
<td>definetfisurface.F</td>
<td>69</td>
</tr>
<tr>
<td>J.1</td>
<td>TFI.F</td>
<td>73</td>
</tr>
<tr>
<td>K.1</td>
<td>functions.F</td>
<td>77</td>
</tr>
<tr>
<td>L.1</td>
<td>alloctfiarray.F</td>
<td>93</td>
</tr>
<tr>
<td>M.1</td>
<td>definespline.F</td>
<td>97</td>
</tr>
</tbody>
</table>
1 Introduction

Nowadays thermal fluid structure analysis play a decisive role in industries and research. It is often not sufficient to consider the structural mechanics, fluid mechanics or the heat transfer individually, as there are significant coupling effects like deformation of a structure caused by thermal expansion and fluid forces or heat transfer due to friction of a fluid. If one wants to simulate these phenomena, one has to decide between different coupling approaches. On the one hand, one can use one solver for both, the solid and fluid (strong coupling) computing the problems simultaneously or the domains can be split up into two solvers with an coupling interface, that provides the exchange of the forces, deformations and temperatures after each timestep (weak coupling). Applying the latter, one can use the software MpCCI\(^1\) in order to couple the software FEAP\(^2\) for the structure mechanics analysis and FASTEST\(^3\) for the fluid mechanics.

1.1 Motivation

When using MpCCI as coupling interface for FEAP and FASTEST, the user has to provide a definition of the coupling surface. The coupling succeeds, when MpCCI gets from both softwares a certain amount of nodes defining the same coupling surface. This definition is already implemented for FASTEST by the use of a grid generator like icemCFD. In FEAP the definitions have to be done manually in the input file. Each coupling node and element have to be defined explicitly. For more complex geometries this leads to extremely large input files, which are hard to be parametrized what implicates, that the definition has to be updated when the mesh changes due to refining or editing. It is obvious that one can not handle the definitions manually for complex geometries and the need for an automatisation gets clear. This is what was done in this work. The developed functions provide a recognition of coupling nodes and elements by the use of parameters that define this surface uniquely, e.g. three points for a planar coupling face.

With the use of the new functions, simulation of fluid-structure interactions using FEAP and FASTEST gets more interesting as the user can also simulate more complicate geometries or edit the mesh whenever needed.

1.2 Objective and outline

The objection of this theses is to identify the coupling nodes for different geometries (planar, polar, coned and freeform) only with a bunch of parameters which define the surface exactly. This includes the implementation of usermacros that provide the user interaction in the FEAP input file to pass the parameters. These parameters have to be conditioned for the use of computations within the recognition-routines. Founded nodes then have to be prepared to be sent to MpCCI in respect to the requirements the software has.

To complete this project, the implemented functions have to be evaluated, documented and also tested in more complex geometries.

\(^{1}\) Mesh-based parallel Code Coupling Interface - http://www.mpcci.de
\(^{2}\) Finite Element Analysis Program - http://www.ce.berkeley.edu/~rlt/feap/
2 Fundamentals

For the numerical solution of problems in fluid and continuum mechanics, the finite volume method for the fluid and the finite element method for the structural problems are mainly employed. They are based on mathematical concepts, definitions and methods which will be discussed in the following chapters. Afterwards the \( \kappa - \epsilon \) method is introduced for the computation of turbulent flows. Especially in context of coupling free form surfaces, the grid generation via transfinite interpolation is an important tool and will be described at the end.

2.1 Governing equations for fluid mechanical problems

In order to characterize the flow behaviour of liquids or gasses (possibly with additional consideration of heat and species transport processes) one usually employs the Eulerian formulation (a spaciously fixed reference system). This formulation allows to describe the properties of a flow at a certain location in the flow domain.

The mass \( m \) of an arbitrary volume \( V \) is defined by

\[
m(t) = \int_V \rho(x, t) \, dV
\]

with the density \( \rho \). If there are no mass sources or sinks, the total mass of a body remains constant for all time:

\[
\frac{D}{Dt} \int_V \rho \, dV = 0.
\]

During a deformation, the volume and the density may change, but the mass will be the same. Written in differential formulation yields to

\[
\frac{\partial \rho}{\partial t} + \frac{\partial (\rho v_i)}{\partial x_i} = 0.
\]

We restrict ourselves to the Newtonian fluids which are characterized by the following material law for the Cauchy stress Tensor \( T \):

\[
T_{ij} = \mu \left( \frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i} - \frac{2}{3} \frac{\partial v_k}{\partial x_k} \delta_{ij} \right) - p \delta_{ij}
\]

with the pressure \( p \) and the dynamic viscosity \( \mu \). Using this tensor together with the conservation laws for mass, momentum and energy and again applying Fourier’s law yields to

\[
\frac{\partial \rho}{\partial t} + \frac{\partial (\rho v_i)}{\partial x_i} = 0
\]

\[
\frac{\partial (\rho v_i)}{\partial t} + \frac{\partial (\rho v_i v_j)}{\partial x_j} = \frac{\partial}{\partial x_j} \left[ \mu \left( \frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i} - \frac{2}{3} \frac{\partial v_k}{\partial x_k} \delta_{ij} \right) \right] - \frac{\partial p}{\partial x_i} + \rho f_i
\]

\[
\frac{\partial (\rho e)}{\partial t} + \frac{\partial (\rho v_i e)}{\partial x_i} = \mu \left[ \frac{\partial v_i}{\partial x_j} \left( \frac{\partial v_j}{\partial x_i} + \frac{\partial v_i}{\partial x_j} \right) - \frac{2}{3} \left( \frac{\partial v_k}{\partial x_k} \right)^2 \right] - p \frac{\partial v_i}{\partial x_i} + \frac{\partial}{\partial x_i} \left( \kappa \frac{\partial T}{\partial x_i} \right) + \rho q.
\]

(2.6) is known as the (compressible) Navier-Stokes equation. Two more equations are needed, as we have more unknowns than equations. They can be derived from the thermodynamic properties of the fluid, namely the thermal and the caloric equations of states. If we consider caloric ideal gas, they are

\[
p = \rho RT \quad \text{and} \quad e = c_v T
\]

with the specific gas constant \( R \) of the fluid and the specific heat capacity \( c_v \). These equations can be simplified by assuming incompressible an inviscid flows e.g.. Together with appropriate boundary conditions we now have a closed formulation of a fluid mechanical problem.
(2.6) can be simplified for incompressible flows where \( \partial v_i / \partial x_i = 0 \) which leads to

\[
\frac{\partial v_i}{\partial x_i} = 0, \quad \text{(2.9)}
\]

\[
\frac{\partial (\rho v_i)}{\partial t} + \frac{\partial (\rho v_i v_i)}{\partial x_i} = \frac{\partial}{\partial x_i} \left[ \mu \left( \frac{\partial v_i}{\partial x_i} + \frac{\partial v_i}{\partial x_i} \right) \right] - \frac{\partial p}{\partial x_i} + \rho f_i, \quad \text{(2.10)}
\]

\[
\frac{\partial (\rho e)}{\partial t} + \frac{\partial (\rho v_i e)}{\partial x_i} = \mu \frac{\partial v_i}{\partial x_i} \frac{\partial v_i}{\partial x_i} + \frac{\partial}{\partial x_i} \left( \nu \frac{\partial T}{\partial x_i} \right) + \rho q. \quad \text{(2.11)}
\]

For a criterion for the validity of this assumption the Mach number

\[
Ma = \frac{\bar{v}}{a} \quad \text{(2.12)}
\]

is taken into account, where \( \bar{v} \) is a characteristic flow velocity of the problem and \( a \) is the speed of sound in the corresponding fluid. Incompressibility usually is assumed if \( Ma < 0.3 \).

How these equations can be solved will be presented in (2.4)

### 2.2 Governing equations for continuum mechanics problems

When dealing with structural mechanics, the Lagrangian formulation is preferred, where the reference system is linked to a material point of the solid described by the transformation

\[
x_i = \chi(X_j, t). \quad \text{(2.13)}
\]

The momentum conservation law states, that the temporal change of the momentum of a body equals the sum of all body and surface forces acting on the body. The momentum vector \( \mathbf{p} = \rho \mathbf{v} \) of a body is defined by:

\[
p_i(t) = \int \rho(x(t), t) v_i(x(t)) dV \quad \text{(2.14)}
\]

(with the density \( \rho(x, t) \) and the velocity \( v(x, t) \)). The differential formulation of this law yields

\[
\frac{\partial (\rho v_i)}{\partial t} + \frac{\partial (\rho v_i v_j)}{\partial x_j} = \frac{\partial T_{ij}}{\partial x_j} + \rho f_i. \quad \text{(2.15)}
\]

In this context, \( f_i \) denotes the acting volume forces and \( T_{ij} \) are the components of the symmetric Cauchy stress tensor which describes the body’s state of stress.

For linear problems, where only small deformations are allowed, the strain of each point is characterized by the strain displacement relation

\[
e_{ij} = \frac{1}{2} \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right). \quad \text{(2.16)}
\]

(2.15) formulated for displacements leads to

\[
\rho \frac{D^2 u_i}{Dt^2} = \frac{\partial T_{ij}}{\partial x_j} + \rho f_i. \quad \text{(2.17)}
\]

For the unknown tensor \( T_{ij} \) we have to assume a material law like Hooke’s law for linear elastic material behaviour:

\[
T_{ij} = \lambda e_{kk} \delta_{ij} + 2\mu e_{ij}. \quad \text{(2.18)}
\]

\( \lambda \) and \( \mu \) are the Lamé constants, which depend on the corresponding material and can be derived from the elasticity modulus \( E \) and the Poisson ratio \( \nu \):

\[
\lambda = \frac{E}{(1 + \nu)(1 - 2\nu)} \quad \text{and} \quad \mu = \frac{E}{2(1 + \nu)}. \quad \text{(2.19)}
\]

Now (2.17) combined with (2.18) yields to the Navier-Cauchy equations of linear elastic theory:

\[
\rho \frac{D^2 u_i}{Dt^2} = (\lambda + \mu) \frac{\partial^2 u_i}{\partial x_j \partial x_j} + \mu \frac{\partial^2 u_i}{\partial x_i \partial x_j} + \rho f_i. \quad \text{(2.20)}
\]

In association with appropriate boundary and initial conditions (e.g. prescribed displacements and/or stresses), (2.20) denotes a closed system of partial differential equations which can be used for the determination of the unknown displacements \( u_i \). For non-linear material properties or great body deformations these equations are not suitable but this is not within the scope of this work. Additional information can be found in [11] and [7].
For simple cases like diffusion in solids or diffusion and convection in fluids the heat transfer can be described by the **scalar transport equations**. As the heat conduction in solids is a special case of heat transfer in fluids, the latter will be discussed here.

The **first law of thermodynamics** states that the total energy $W$ of a body equals the total external energy supply which is composed by the **power of external forces** $P_e$ and the **heat supply** $Q$:

$$\frac{dW}{dt} = P_e + Q.$$  \hspace{1cm} (2.21)

The total energy $W$ of a body is defined by

$$W(t) = \int_\mathcal{V} \rho e dV + \frac{1}{2} \int_\mathcal{V} \rho v_i v_i dV$$ \hspace{1cm} (2.22)

with $e$ the **specific internal energy**. $P_e$ and $Q$ are defined by

$$P_e(t) = \int_\mathcal{S} T_i v_j n_i dS + \int_\mathcal{V} \rho f_i v_i dV \quad \text{and} \quad Q(t) = \int_\mathcal{V} \rho q dV - \int_\mathcal{S} h_i n_i dS$$ \hspace{1cm} (2.23)

where $q$ denotes (scalar) heat sources and $h = h_i e_i$. With these definitions, (2.21) can be written as:

$$\frac{D}{Dt} \int_\mathcal{V} \rho \left( e + \frac{1}{2} v_i v_i \right) dV = \int_\mathcal{S} T_i v_j - h_i \right) n_i dS + \int_\mathcal{V} \rho (f_i v_i + q) dV$$ \hspace{1cm} (2.24)

with the (known) velocity $v = v_i e_i$. Using Gauss integral theorem, the momentum conservation law 2.15 and the Reynolds transport theorem, one obtains the energy balance in the differential form:

$$\partial \left( \rho e - \frac{1}{2} v_i v_i \right) + \rho \frac{\partial v_i}{\partial x_i} = T_j v_j \frac{\partial h_i}{\partial x_i} + \rho g$$ \hspace{1cm} (2.25)

Now we assume **Fourier’s Law** (for isotropic materials)

$$h_i = -\kappa \frac{\partial T}{\partial x_i}$$ \hspace{1cm} (2.26)

(where $\kappa$ is the **heat conductivity**) as a constitutive relation for the heat flux and a flow. In addition we assume, that $\kappa$ of the fluid is constant and the work done by pressure and friction forces can be neglected. Then the energy balance 2.25 yields to the convection-diffusion equation for the temperature $T$

$$\partial \left( \rho c_p T \right) + \rho \frac{\partial v_i}{\partial x_i} \left( \rho c_p v_i T - \kappa \frac{\partial T}{\partial x_i} \right) = \rho q.$$ \hspace{1cm} (2.27)

where $q$ donates possibly present heat sources or sinks and $c_p$ the **specific heat capacity**.

In order to solve this equation, one has to adopt certain boundary conditions. These can be

- prescribed temperatures: $T = T_b$
- prescribed heat flux: $\kappa \frac{\partial T}{\partial x_i} n_i = h_b$
- heat flux proportional to heat transport: $\kappa \frac{\partial T}{\partial x_i} n_i = \tilde{\alpha} \left( T_b - T \right)$

where $T_b$ and $h_b$ are prescribed values at the problem domain boundary $\Gamma$ for the temperature and the heat flux in normal direction, respectively, and $\tilde{\alpha}$ is the **heat transfer coefficient**.

If one now set $v_i = 0$, one considers only the diffusion and obtains the heat conduction equation in a medium at rest (fluid or solid). If one now also drops the terms with the time derivation, the corresponding equations for steady heat transfer are obtained:

$$\kappa \frac{\partial^2 T}{\partial x_i^2} = \rho q$$ \hspace{1cm} (2.28)
2.3.1 Linear Thermo Elasticity

Quite often, thermal effects play an essential role for structural deformation. Therefore it is necessary to find equations describing a coupled thermo-mechanical problem. They can be derived from the momentum law (2.15) and the energy conservation (2.24) using the strain displacement relation (2.16). We assume a simple linear thermo-elastic material.

The specific dissipation function \( \psi \) is given as:

\[
\psi = T_{ij} D_{ij} - \rho \frac{D}{Dt} (e - Ts) + \rho_s \frac{DT}{Dt} \tag{2.29}
\]

where \( s \) is the specific internal entropy. This function is a measure for the energy dissipation in the continuum. As we assume a simple thermo elastic material, there is no energy dissipation. Therefore \( \psi = 0 \). Using this equation together with the energy conservation (2.24) and again assuming the validity of Fourier's Law (2.26) yields to

\[
T \rho \frac{Ds}{Dt} = -\frac{\partial}{\partial x_i} \left( \kappa \frac{\partial T}{\partial x_i} \right) + \rho q \tag{2.30}
\]

2.4 Finite Volume Method

As there does not exist an universal algebraic solution for the partial differential equation introduced before, the solutions have to be approximated. Different methods have been developed and the most frequently used for fluid flow problems is the finite volume method which will be introduced here for the two dimensional case. The equations for the three dimensional case follow straight forward. For mechanical problems, the finite element method will be outlined in the next chapter.

2.4.1 Discretisation

As a first step, the problem domain \( \Omega \) will be discritized into a finite number of non overlapping subdomains \( V_i \) \((i = 1, \ldots, N)\), the control volumes (CV) and related nodes, where the variable values are computed. They can be defined on a numerical grid, which can be generated with different techniques (details in [12], [6] and [7]). If one now integrates the differential balance equations over an arbitrary control volume \( V \) and applies the Gauss integral theorem, one obtains for the general stationary transport equation (2.28)

\[
\int_S \left( \rho v_i \phi - a \frac{\partial \phi}{\partial x_i} \right) n_i dS = \int_V f dV, \tag{2.31}
\]

where \( S \) is the surface of the CV and \( n_i \) are the components of the unit normal vector of the surface.

The same notation as in [6] is used here: in compass notation the four sides and the four vertexes are named. The neighbours of the element are also named in compass notation but with capitol letters (see image 2.1)
Then the surface integral (2.31) can be split into the sum of the four surface integrals over the faces $S_c$ ($c = e, w, n, s$) of the CV:
\[
\sum_c \int_{S_c} \left( \rho v_i \phi - \alpha \frac{\partial \phi}{\partial x_i} n_{ci} \right) dS_c = \int_V f \, dV. \tag{2.32}
\]

Physically this can be interpreted as the sum of the convective and diffusive fluxes $F^C_c$ and $F^D_c$ through the CV faces, respectively.

Looking at the face $S_e$, for instance, the unit normal vector $n_e = (n_{e1}, n_{e2})$ is defined by
\[
n_e = \frac{(y_{ne} - y_{se})}{\delta S_e} e_1 - \frac{(x_{ne} - x_{se})}{\delta S_e} e_2 \tag{2.33}
\]
where
\[
\delta S_e = |x_{ne} - x_{se}| = \sqrt{(x_{ne} - x_{se})^2 + (y_{ne} - y_{se})^2} \tag{2.34}
\]
denotes the length of the face $S_e$. Until here, no approximation has been made, and the equations are still exact.

### 2.4.2 Approximation of Surface and Volume Integrals

Now the integrals in (2.32) have to be approximated as there is no direct solution. We only consider the two dimensional case here, but the three dimensional can be adopted straightforward. Several methods for the approximation of the surface integral can be used like the midpoint rule, trapezoidal rule and the Simpson rule.

Let us have a look at the approximation of a general surface integral
\[
\int_{S_2} w_i n_{ci} dS_c \tag{2.35}
\]
over the face $S_c$ of a CV. Other faces can be treated in an analogous way. $w_i$ is a general integrand function $w = (w_1(x), w_2(x))$. Applying the midpoint rule for this integral means to use the value of the midpoint of the CV as an approximation of the integral. Other approximation rules can be found in [6] and will not be discussed here.

Using the midpoint rule for the convective and diffusive fluxes through the CV faces in 2.32 we obtain
\[
F^C_c \approx \rho v_i n_{ci} \delta S_c \phi_c \quad \text{and} \quad F^D_c \approx -\alpha n_{ci} \delta S_c \frac{\partial \phi}{\partial x_i} \tag{2.36}
\]
if $v_i, \rho$ and $\alpha$ are constant across the CV, what we assume here. With the use of the definition of the normal vector, the convective flux through the face $S_c$ can be approximated.

Considering the volume integral in (2.32) the approximation can be done with the assumption that the value of the centre $f_p$ of the CV represents an average value over the CV. This leads to the two dimensional midpoint rule:
\[
\int_V f \, dV \approx f_p \delta V \tag{2.37}
\]
where $\delta V$ denotes the volume of the CV. There are several integration formulas for Cartesian grids that differs in the corresponding error order (with respect to $\delta V$). They can be found in [6] and will not be described here. The integration formulas for three dimensional volume integrals are available analogously.

The midpoint rule applied to all integrals leads to the following approximated balance equation of (2.32):
\[
\sum_c \bar{n}_c \phi_c - \alpha \sum_c n_{ci} \delta S_c \frac{\partial \phi}{\partial x_i} = f_p \delta V. \tag{2.38}
\]
Now the function values and derivatives of $\phi$ at the CV which occur in the convective and diffusive fluxes have to be approximated by the variable values in the CV centre. This will be outlined for two dimensional Cartesian grid and can be transferred as well to non Cartesian and three dimensional grid.
Discretisation of Convective Fluxes

What should be done next, is to approximate the values of $\phi_e$ in the neigbouring CV. Therefore exist several different techniques. The central differencing scheme (CDS) and the upwind differencing scheme (UDS) are two of them and they will be explained. Other techniques like the flux blending technique are described in [6].

The CDS approximates $\phi_e$ by the use of the centre point $P$ of the CV and the centre point of the eastern neighbour $E$:

$$\phi_e \approx \gamma_e \phi_E + (1 - \gamma_e) \phi_P. \tag{2.39}$$

The interpolation factor $\gamma_e$ is defined by

$$\gamma_e = \frac{x_e - x_P}{x_E - x_P}. \tag{2.40}$$

For Cartesian grids, this approximation has for equidistant and non equidistant grids an interpolation error of second order and this can be proved by the use of a Taylor series expansion of $\phi$ around the point $x_P$. Approximation of a higher order can be found by involving additional grid points. Here one should also use an integration formula of corresponding order.

The UDS determines $\phi_e$ depending on the direction of the mass flux as follows

$$\phi_e = \phi_P, \quad \text{if} \quad m_e > 0, \tag{2.41}$$

$$\phi_e = \phi_E, \quad \text{if} \quad m_e < 0 \tag{2.42}$$

and has an interpolation error of first order (this can be shown by a Taylor series expansion of $\phi$ around $x_P$, evaluated at the point $x_e$). This approximation is quite good, if the transport direction is nearly perpendicular to the CV face. Otherwise the approximations can be quite inaccurate and for large mass fluxes it can be necessary to employ very fine grids for the computation.

Discretisation of Diffusive Fluxes

To approximate the diffusive fluxes, it is necessary to approximate the values of the normal derivative of $\phi$ at the CV faces by nodal values in the CV centres.

Assuming, that $\phi$ is a linear function between the points $x_P$ and $x_E$, one obtains a central differential formula:

$$\left( \frac{\partial \phi}{\partial x} \right)_e \approx \frac{\phi_E - \phi_P}{x_E - x_P}. \tag{2.43}$$

The error of the approximation is of second order for equidistant grid and again can be proved by the use of Taylor expansion around $x_e$ at the locations $x_P$ and $x_E$. For non-equidistant grid, the approximation error is proportional to the grid spacing.

Other techniques of higher approximation order can be found in [6] or [7] as well as the necessary modifications for non Cartesian grids which will not be discussed here.

2.4.3 Discrete Transport Equation

Employing the midpoint rule for the integral approximations, the UDS method for the convective flux, the CDS method for the diffusive flux and assume that the velocity components $v_1, v_2 > 0$ on a Cartesian grid yields to a relation of the form

$$a_P \phi_P = a_E \phi_E + a_W \phi_W + a_N \phi_N + a_S \phi_S + b_P \tag{2.44}$$

with the coefficients
\[ a_E = \frac{\alpha}{(x_e - x_p) (x_e - x_w)} \],  
\[ a_W = \frac{\rho v_1}{x_e} + \frac{\alpha}{(x_p - x_w) (x_e - x_w)} \],  
\[ a_N = \frac{\alpha}{(y_N - y_p) (y_N - y_s)} \],  
\[ a_S = \frac{\rho v_2}{y_n y_s} + \frac{\alpha}{(x_p - x_w) (x_e - x_w)} + \frac{\alpha}{y_n y_s} \]  
\[ b_p = f_p. \]  

The relation of the coefficients

\[ a_p = a_e + a_w + a_N + a_S \]  

is characteristic for the finite volume discretisation and expresses the conservativity of the method.

### 2.4.4 Treatment of Boundary Conditions

When dealing with boundary value problems, boundary conditions are needed to solve them numerically. There are three boundary condition types that most frequently occur for the considered type of problems:

- a prescribed variable value
- a prescribed flux
- and a symmetry boundary

If we have a prescribed boundary value \( \phi_w = \phi^0 \), the convective flux at the boundary yields to:

\[ F^C_W \approx \dot{m}_w \phi_w = \dot{m}_w \phi^0 \]  

Now \( F^C_W \) as well as \( \dot{m}_w \), are known (see above) and can be used in the balance equation (2.38). With the same approach as in the interior of the domain, the diffusive flux through the boundary is determined:

\[ \left( \frac{\partial \phi}{\partial x} \right)_W \approx \frac{\phi_p - \phi_w}{x_p - x_w} = \frac{\phi_p - \phi^0}{x_p - x_w} \]  

If we now consider a prescribed flux at a boundary (e.g. \( F_w = F^0 \)) the flux through the CV face is obtained by dividing \( F^0 \) through the length of the face (e.g. \( x_e - x_w \)). The resulting value is introduced in (2.38) as total flux.

If a problem can be split into two or more symmetric domains, one can downsize the problem domain to save computing time or to get a higher accuracy (with a finer grid) with the same computational effort. Boundary conditions of this type can be applied using:

\[ \frac{\partial \phi}{\partial x_i} n_i = 0 \]  

which means that the diffusive flux through the symmetry boundary is zero. At a symmetry boundary, the normal component of the velocity vector has to be zero. Therefore the mass flux is zero which causes that the convective flux through the boundary is also zero. Thus the total flux through the corresponding CV face can be set to zero in the balance equation.

Together with the boundary conditions at all boundaries of the problem domain the algebraic system of equations resulting from a finite volume discretisation has an unique solution which is described in the next chapter.
2.4.5 Algebraic System of Equation

We now have an algebraic equation for each CV. Summed up over all \( N \) CVs

\[ a_i^p \phi_i^p - \sum_{c} a_i^c \phi_i^c = b_i^p \]  \hspace{1cm} (2.55)

with \((c = n, e, s, w)\). In the one dimensional case, the centre point of a CV has only one eastern and one western neighbour. Therefore the equation for the \( i \)-th CV with a second order central differencing scheme is

\[ a_i^p \phi_i^p - a_i^e \phi_i^e - a_i^w \phi_i^w = b_i^p \]  \hspace{1cm} (2.56)

whereas

\[ \phi_i^e = \phi_{i+1}^e \text{ for all } i = 2, \ldots, N \quad \text{and} \quad \phi_i^w = \phi_{i-1}^w \text{ for all } i = 1, \ldots, N-1. \]  \hspace{1cm} (2.57, 2.58)

For 2D it is analogously. Written in matrix form this yields to

\[ A^{N\times M} \Phi^{1\times N} = b^{1\times N}. \]  \hspace{1cm} (2.59)

whereas \( A \) has a diagonal form.

The solution of the equation will be explained in Chapter 2.6 as it is also used for the finite element method which is introduced in the next chapter.

2.5 Finite Element Method

The following chapter introduces the Finite Element Method (FEM) which is most frequently used for an approximation of the equations governed in 2.2.

2.5.1 Discretisation

Analogously to the FVM, the problem domain has to be discretized before the mathematical foundations of the FEM can be considered. This is done by the use of a finite number of non overlapping elements that might be triangles or quadrilateral structures in a two dimensional case.

The state of an element is usually described by the piecewise polynomial ansatz functions. These are formulated by the use of designated local attributes \( \Phi_i^1, \ldots, \Phi_i^P \) such as node values and/or derivatives (of the \( i \)-th element) in contrast to the FVM where these are values of the element centre. The approximation can be written as

\[ \Phi(x) = \sum_{j=1}^{P} \Phi_j^i N^i_j(x) \]  \hspace{1cm} (2.60)

with the local shape functions \( N_1, \ldots, N_p \) and the unknown \( \Phi_j^i \). If only function values are used as nodal variables, i.e. at suitable locations \( x_1, \ldots, x_p \) in the Element \( E_i \), the local shape functions fulfill the relations

\[ N_j^i(x_n) = \begin{cases} 1, & \text{for } j = n \\ 0, & \text{for } j \neq n \end{cases} \]  \hspace{1cm} (2.61)

since \( \Phi^j \) at the nodes \( x_n \) must take the nodal value \( \Phi_{i,n}^j \).

A global representation can be derived when numbering the problem domain consecutively, where common local nodal variables of adjacent elements are counted only once:

\[ \Phi(x) \approx \phi_0(x) + \sum_{k=1}^{N^i} \Phi_k N_k(x) \]  \hspace{1cm} (2.62)

\( N_k \) is that local shape function, where the local nodal variable \( \Phi_k^i \) coincides with the global nodal variable \( \Phi_k \). In the function \( \phi_0 \) the Dirichlet boundary conditions are subsummed.
Neumann boundary conditions

\[
\frac{\partial \phi}{\partial x_i} n_i = t_b
\]

(2.63)

can be adopted in the load vector \( b \) (see 2.5.3).

### 2.5.2 Method of the Weighted Residuals

As we have introduced an approximation for the solution of the equations, the residual \( R \) is introduced, which represents the current deviation from the exact solution. We now claim that the integral mean value of \( R \) over the problem domain vanishes. This leads to conditions for the unknown values \( \Phi_i^1, \ldots, \Phi_i^p \). To determine the mean value one can use an arbitrary test function \( \phi \) which vanishes on all Dirichlet boundaries (prescribed value at a certain point) of the problem domain \( \Omega \). This approach is called the method of weighted residuals and can be written as

\[
\int_{\Omega} R \phi d\Omega = 0
\]

(2.64)

### 2.5.3 The Galerkin Method and the Resulting System of Equation

Using the ansatz functions from (2.5.1) as test functions as well, (2.64) leads to the formulation of the Galerkin method. In combination with an adequate numerical integration scheme, all local formulations might be incorporated into a (preliminary) global linear system of equations for the unknown values \( \Phi_i^1, \ldots, \Phi_i^p \) for each timestep.

To solve the system of equations, boundary conditions are necessary. They can be applied straightforward as prescribed displacements directly determine the values of the corresponding unknowns. Prescribed stresses can be integrated into the right side of the resulting system

\[
A\Phi = b
\]

(2.65)

with the sparsely filled stiffness matrix \( A \), the load vector \( b \) and the unknowns \( \Phi \).

### 2.6 Solution of Linear Systems of Equations

Both, the FVM and the FEM have a linear system of equations to solve. Both are of the form \( A\Phi = b \), whereas \( A \) is a sparsely filled matrix. In the recent years, many powerful methods for efficient solving of the equations have been developed. Several direct solvers like the Gaussian elimination compute the exact solution with at most \( n^3 \) operations, iterative solvers compute an approximation of the solution and usually have a linear complexity. These are solvers like the Jacobi and Gauss-Seidel that will be shortly introduced here. More solvers and theory can be found in [6].

#### 2.6.1 Jacobi and Gauss-Seidel

Both method requires, as a first step, a decomposition of the system matrix \( A \) similar to

\[
A = L + D + U
\]

(2.66)

with the lower triangle \( L \), the upper triangle \( U \) and the diagonal part \( D \) of \( A \). The algorithms differ in the formulation of the iteration step. For the Jacobi method it may be written as

\[
D\Phi^{k+1} = -(L + U)\Phi^k + b.
\]

(2.67)

whereas the Gauss-Seidel uses

\[
(D + L)\Phi^{k+1} = -U\Phi^k + b.
\]

(2.68)

Although the system has to be solved for each iteration step, it is mostly more efficient than inverting the sparsely filled matrix \( A \). Further iterative methods can be found in [1].
2.7 Computation of Turbulent Flows

In practical applications, most flow processes are turbulent. The introduced Navier-Stokes equations in Sec. (2.1) could theoretically be taken into account for simulating the flow, but practically it is not possible due to the enormous computational effort. Therefore special numerical methods were introduced to handle the complexity. They are mostly based on statistical turbulence models and we will restrict ourselves to the incompressible case.

2.7.1 Characterization of Computational Methods

A decisive factor for the simulation of turbulent flows are the great scales of length and time dimensions and consequently the high resolution. They have been characterised by Kolmogorov (1942) and are directly coupled to the Reynolds number (a dimensionless number that gives a measure of the ratio of inertial forces to viscous forces). The spatial scale must be \( l_k \sim Re^{3/4} \) and the time scale \( t_k \sim Re^{1/2} \) in order to fully compute the flow. Because of these requirements the direct numerical simulation (DNS) is not practically usable for great Reynolds numbers due to the enormous computational effort of the great amount of timesteps and gridpoints.

Therefore alternative approaches have been employed for computing turbulent flows and will be introduced in the next chapter. We restrict ourselves to the statistical turbulence models. Other approaches like the large eddy simulation (LES) can be found in [5] and [6].

2.7.2 Statistical Turbulence Modelling

When using a statistical turbulence model, each flow variable \( \Phi \) is expressed by a mean value \( \bar{\Phi} \) and a fluctuation \( \phi' \):

\[
\phi(x, t) = \bar{\phi}(x) + \phi'(x, t),
\]

with the mean value

\[
\bar{\phi}(x) = \lim_{T \to \infty} \frac{1}{T} \int_{t_0}^{t_0 + T} \phi(x, t) dt
\]

in the statistically steady case. If the averaging time \( T \) is large enough, \( \bar{\phi} \) does not depend on the point of time \( t_0 \) when the averaging started. If \( \phi \) is statistically unsteady, the mean value is time dependent too. It is then defined by ensemble averaging:

\[
\bar{\phi}(x, t) = \lim_{N \to \infty} \frac{1}{N} \sum_{n=1}^{N} \phi(x, t).
\]

Inserting this ansatz in the conservation equations for mass, momentum and energy (2.9) and subsequently averaging yields to the Reynolds averaged Navier-Stokes (RANS) equations (or Reynolds equations):

\[
\frac{\partial \bar{\rho} \bar{v}_i}{\partial t} + \frac{\partial}{\partial x_j} \left( \rho \bar{v}_i \bar{v}_j + \bar{v}_i' \bar{v}_j' \right) = - \frac{\partial \bar{p}}{\partial x_i} + \rho f_i,
\]

We now have simplified equations as the mean values are time independent or at least the time dependence can be resolved with a certain amount of time steps. But we have as well the Reynolds stresses \( \rho \bar{v}_i' \bar{v}_j' \) as new unknowns. To solve the equations system, suitable approximations for the correlations have to be employed. There are different models for this turbulence modelling whereas the most important are:

- algebraic models (zero-equation models),
- one- and two-equation models,
- Reynolds stress models.

In this work the \( k-e \) model was used.
2.7.3 The $k$-$\varepsilon$ Turbulence Model

The $k$-$\varepsilon$ model, developed in the 1960s by Spalding and Launder, assumes that

$$\rho \overline{v'_i v'_j} = -\mu_t \left( \frac{\partial \overline{v}_i}{\partial x_j} + \frac{\partial \overline{v}_j}{\partial x_i} \right) + \frac{2}{3} \rho \delta_{ij} k$$

(2.74)

for the Reynolds stresses (also known as Bussinesq approximation). $\mu_t$ denotes the turbulent viscosity and depends on the flow variables, $\delta_{ij}$ is the Kronecker function and $k$ is the turbulent kinetic energy, defined by

$$k = \frac{1}{2} \overline{v'_i v'_i}.$$  

(2.75)

Since $\mu_t$ and $k$ are unknown too, the system is not yet well defined. Two other assumptions of the $k$-$\varepsilon$ model relates $\mu_t$ to $k$ and to the dissipation rate of the turbulent kinetic energy $\varepsilon$:

$$\mu_t = C_\mu \rho \frac{k^2}{\varepsilon}, \quad \varepsilon = \frac{\mu}{\rho} \frac{\overline{v'_i v'_j}}{\overline{v'_i v'_j}}.$$  

(2.76)

Two transport equations for $k$ and $\varepsilon$ are needed. Inserting 2.74 into the momentum equation (2.5) and defining $\tilde{p} = \bar{p} + 2k/3$ yields to the equation system

$$\frac{\partial \bar{v}_i}{\partial x_i} = 0,$$

(2.77)

$$\frac{\partial (\rho \bar{v}_i)}{\partial t} + \frac{\partial}{\partial x_i} \left[ \rho \bar{v}_i \bar{v}_j - (\mu + \mu_t) \left( \frac{\partial \bar{v}_i}{\partial x_j} + \frac{\partial \bar{v}_j}{\partial x_i} \right) \right] = -\frac{\partial \tilde{p}}{\partial x_i} + \rho f_i,$$

(2.78)

$$\frac{\partial (\rho k)}{\partial t} + \frac{\partial}{\partial x_j} \left[ \rho \bar{u}_j \bar{u}_k - (\mu + \mu_t) \frac{\partial k}{\partial x_j} \right] = G - \rho \varepsilon,$$

(2.79)

$$\frac{\partial (\rho \varepsilon)}{\partial t} + \frac{\partial}{\partial x_j} \left[ \rho \bar{u}_j \varepsilon - (\mu + \mu_t) \frac{\partial \varepsilon}{\partial x_j} \right] = C_{\varepsilon 1} \frac{\varepsilon}{k} - C_{\varepsilon 2} \rho \frac{\varepsilon^2}{k},$$

(2.80)

which has to be solved for the unknowns $\tilde{p}, \bar{v}_i, k$ and $\varepsilon$, $C_{\varepsilon i}$ are model constants. With this equation system and reasonable boundary conditions one has reduced the computational effort enormously and one can now simulate turbulent flows with an adequate accuracy.

2.8 Transfinite Interpolation

The transfinite interpolation (TFI) is usually used for grid generation. The idea is to map a structured mesh from a logical domain to a physical domain. This is illustrated in fig. 2.2.

![Relation between coordinates and grid points in physical and logical domains.](image)

Figure 2.2: Relation between coordinates and grid points in physical and logical domains.

It is now necessary to have unique mapping
(x, y) = (x(ξ, η), y(ξ, η)) or (ξ, η) = (ξ(x, y), η(x, y))  \hspace{1cm} (2.82)

between given discreet values \(ξ = 0, 1, \ldots, N\) and \(η = 0, 1, \ldots, M\) and the physical problem values.

First, the boundary of the problem domain is described:

\[
x(ξ, 0) = x_i(ξ), \quad x(ξ, M) = x_n(ξ) \quad \text{for} \quad ξ = 0, \ldots, N, \hspace{1cm} (2.83)
\]

\[
x(0, η) = x_w(η), \quad x(N, η) = x_c(η) \quad \text{for} \quad η = 0, \ldots, M. \hspace{1cm} (2.84)
\]

The corner points have to fulfill the compatibility conditions. Now the inner points can be determined by the use of an interpolation rule. A simple linear interpolation yields to the following relation

\[
x(ξ, η) = (1 - \frac{η}{M}) x_i(ξ) + \frac{η}{M} x_n(ξ) + (1 - \frac{ξ}{N}) x_w(η) + \frac{ξ}{N} x_c(η)
- \frac{ξ}{N} \left[ \frac{η}{M} x_M(0) + (1 - \frac{η}{M}) x_M(0) \right]
- (1 - \frac{ξ}{N}) \left[ \frac{η}{M} x_L(0) + (1 - \frac{η}{M}) x_L(0) \right]
\hspace{1cm} (2.85)

which is known as the TFI. One can get the three dimensional case straightforward. We assume \(0 \leq ξ \leq 1, 0 \leq η \leq 1\) and \(0 \leq ζ \leq 1:\)

\[
t_{uw}(ξ, η, ζ) = (1 - ζ) x(0, η, ζ) + ζ x(1, η, ζ)
\]
\[
t_{uy}(ξ, η, ζ) = (1 - ζ) x(ξ, 0, ζ) + ζ x(ξ, 1, ζ)
\]
\[
t_{uw}(ξ, η, ζ) = (1 - ζ) x(ξ, η, 0) + ζ x(ξ, η, 1)
\]
\[
t_{uw}(ξ, η, ζ) = (1 - ζ)(1 - η) x(0, 0, ζ) + (1 - ζ) x(0, η, 1) + η x(0, 0, ζ) + ζ x(1, η, 1)
\]
\[
t_{uy}(ξ, η, ζ) = (1 - ζ)(1 - η) x(ξ, 0, ζ) + (1 - ζ) x(ξ, 0, 1) + η x(ξ, 0, ζ) + ζ x(ξ, 1, ζ)
\]
\[
t_{uw}(ξ, η, ζ) = (1 - ζ)(1 - η) x(ξ, 1, ζ) + η x(ξ, 1, ζ) + ζ x(ξ, 1, 1)
\]
\[
t_{uw}(ξ, η, ζ) = (1 - ζ)(1 - η) x(0, 0, ζ) + (1 - ζ) x(0, η, 0) + η x(0, 0, ζ) + ζ x(0, 1, ζ)
\]
\[
x(ξ, η, ζ) = t_{uw} + t_{uy} + t_{uw} - t_{uw} - t_{uw} + t_{uw}
\]

Other techniques of grid generation and further information can be found in [12].
## 3 Coupled thermal FSI with MpCCI, FEAP and FASTEST

### 3.1 Coupled Fluid-Solid Problems

For many engineering applications, mechanically and/or thermally coupled fluid-solid problems play a decisive role. For thermally coupled problems one has to consider convective heat transfer and fluid properties of the fluid and, on the other side, thermal stresses and mechanical dissipation of the solid. Forces and deformations can be transferred directly between the fluid and the solid.

There are two approaches for coupling a fluid and a structure simulation. As it is possible to solve the equation system of the solid also via the FVM, one could use one single solver for both geometries. Then the coupling would be provided like any other parameter in the equation system. On the other side, one can partition the problem domain in a fluid and a solid problem. Afterwards they have to be coupled by an interface that exchanges the designated parameters. Several commercial software packages provide an "All-in-one" solution where the coupling service and both solvers are included. The user does need to generate specific coupling faces or specific consistent boundary conditions.

But it is also possible to couple two different programs like FASTEST for the fluid-flow and FEAP for the structure analysis, where one program does not know about the other one. The interpolation between the different grids on the coupling interface is done via the quasi-standard software MpCCI. The idea is, that both programs solve their problems separately, but with alternating boundary conditions, derived from the results of the other solver, at previously defined coupling faces.

In the following chapters the setup for a coupled fluid-structure simulation with the mentioned software-packages are described. The mathematical background is again based on the fundamental conservation laws and will be described in the next chapter.

#### 3.1.1 Modelling of Coupled Fluid-Solid Problems

The basis is again the fundamental conservation laws for mass, momentum, moment of momentum and energy which are valid for any subvolume $V$ of the problem domain either in fluid or solid parts but with the different material laws and individual needs as described in Chap. 2.

When dealing with the movement of a solid within a fluid, it is convenient to consider the arbitrary Lagrangian-Eulerian (ALE) formulation. With this formulation one has the advantages of both formulations. Detailed information see [6].

For the boundary conditions one has to distinguish between different kinds: a solid boundary, a fluid boundary and a fluid-solid interface. The treatment of the solid and fluid boundary conditions are described in the chapters before. At the fluid-solid interface the velocities and the stresses have to fulfill the conditions:

\[ v_i = \frac{Du_b^i}{Dt} \quad \text{and} \quad \sigma_{ij}n_j = T_{ij}n_j, \quad (3.1) \]

where $u_b^i$ and $Du_b^i/Dt$ are the displacement and velocity of the interface, respectively. When also heat transfer is involved, the temperatures as well as the heat fluxes have to coincide on the coupling surface.

### 3.2 Structure

The coupling via MpCCI is based on the exchange of parameters at certain coupling nodes between the two programs. The iterative process can be described by the following steps:

- The fluid solver computes a first time step calculation and sends wall forces and thermal fluxes to MpCCI
- MpCCI interpolates the information to the coupling surface of the solid problem domain
- The structural solver computes temperatures and velocities and sends them back to MpCCI
- MpCCI interpolates the data to the coupling surface of the fluid problem domain
- The fluid solver computes the grid movement
• Check of TFSI convergency. If it is true, the next time step will be computed or another FSI iteration will be done.

This is also illustrated in (fig. 3.1) taken from [4].

![Flow diagram](image)

**Figure 3.1:** Simplified illustration of the work flow of MpCCI

This thesis deals with the definition of the coupling-face in FEAP. As described above, MpCCI needs each node of the coupling surface. For the structural analysis, they can be defined by the node number in the FEAP input file. For the fluid flow, icemCFD can be used as a grid generator to construct the geometry and define the coupling surfaces.

### 3.3 MpCCI as Coupling Interface

MpCCI provides the coupling of FEAP and FASTEST with just some extra definitions to the usual input file. Both geometries have to be in the same dimensions and the coupling faces have to be the same size. Details about the setup of the FASTEST input file can be found in [2].

#### 3.3.1 FEAP Structure

The FEAP input file consists of two blocks. The first block includes the geometry and material definitions as well as the boundary conditions and initial displacements. For a coupled computation another command for MpCCI has to be placed there. In the old implementation this command included the naming of each coupling node and element. The second block includes the commands for solving the problem and the communication with MpCCI. Further information about the setup of the input file can be found in [9].

#### 3.3.2 Setup of the input files

First we will have a look at the setup of the FEAP input file for a coupled computation. If the problem one wants to simulate has thermal boundary conditions, FEAP simulates them by the use of an extra material that has to be assigned to the boundary domain. This additional material simulates the thermal fluxes at the surface. Its dimension has to be \( n - 1 \) and shall be applied like any other material by the use of the `BLENd` command e.g. and have the following properties:

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MATErial 2 ! or any other free number</td>
</tr>
<tr>
<td>2</td>
<td>COUpling</td>
</tr>
<tr>
<td>3</td>
<td>PLANe</td>
</tr>
</tbody>
</table>

**Listing 3.1:** Standard coupling material

For an FSI simulation where one is interested in thermal coupling effects, this material has to be assigned on the coupling surface to simulate the thermal fluxes of the fluid.
Then the MpCCI parameters have to be set up and the coupling nodes defined within its own MPCCI-block. This can be done by naming each coupling node and element individually:

```
MPCCI
MESH 20 ! Mesh Parameter
PART25 ! Part ID
NODE
  1  x1 y1 z1
  2  x2 y2 z2
.. .
ELEM
  1  n1 n2 n3 n4 mate
  2  n1 n2 n3 n4 mate
.. .
MPENd
```

Listing 3.2: MpCCI definitions

or, as later described, automatically with the new functions. Hence the NODE command gets redundant and a new command has to be placed in the the MACRo at the end.

The last step is to modify the MACRo command. First of all the MpCCI has to be initialised (line 4) what makes FEAP to send the information defined in the MPCCI command (3.2) for the coupling nodes and coupling elements to MpCCI. This is followed by a LOOP command with a desired amount of timesteps. In each timestep another LOOP command is necessary for the fsi iterations. As an example the MACRo in (listing 3.3) computes one timestep (line 6) and 50 fsi iterations (line 8). This loop contains the sending and receiving of the desired parameters and the solving of the solid problem domain (line 11-13) by a linear newton solver.

```
MACR
NOPrint
DT,,1.0e10
MPCCI,INIT ! MPCCI initialised and parameters are sent
TRANSient,BACK
LOOP,ts,1 ! 1 timestep
TIME
  LOOP,fsiiter,50 ! 50 fsi iterations
  RECV,HEAT ! temperatures from FASTEST are received
  RECV,FORC ! forces from FASTEST are received
  LOOP,newton,300 ! solving
  TANG,1
  next,newton
  SEND,TEMP ! newly calculated temperatures are send to FASTEST
  SEND,DISP ! newly calculated displacements are send to FASTEST
  CONVergency ! Check for convergency
  next,fsiiter
  next,ts
MPCCI,end ! Close the MPCCI Server
END
```

Listing 3.3: a Macro for FSI

This data needs to be saved and named with a trailing i and no file extensions, e.g. iprojectname. Another file has to be written, where the paths to the log and output files are declared. It should be named just with the project name and could look like the following:
This is all that needs to be done for preparing a FEAP input file for a coupled FSI-computation for the FEAP-part. The changes needed in the input files of FASTEST are not described here but can be found in [2].
4 Implemented functions

As one can see in 3.2, MpCCI needs the number of each coupling node in order to define the coupling surface. As they can change by refining the mesh or editing the geometry, the node definitions have to be updated as well. Quite often a coupling surface has several thousand coupling nodes and the need for an automatisation of this procedure gets obvious. This is done in this work by the implementation of several functions for different geometries and will be explained in details in the following chapter. Afterwards the functions are validated by examples.

4.1 Installing the functions

The following instruction describes how all functions that have been implemented are installed. We assume, that one already has FEAP properly installed (information about the installation of FEAP can be found in [8]) and has a basic knowledge about user functions and their implementation. Further details about how to adopt usermacros can be found in [10]. The extensions have been tested in FEAP Ver. 7.5.

The extension comprises several newly developed files:

<table>
<thead>
<tr>
<th>File</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>umacr6.F</td>
<td>Usermacro that provides the FEAP-commands for calling the functions for planar, circular or conical surfaces. Of course it can be numbered differently if 6 is already in use.</td>
</tr>
<tr>
<td>umacr7.F</td>
<td>Usermacro that provides the FEAP-commands for calling the functions for freeform coupling surfaces. Can of course also named with any other free number.</td>
</tr>
<tr>
<td>boundary8brickelem.F</td>
<td>This file provides the finding of the orientation as well as organising the map for the coupling nodes which is described later.</td>
</tr>
<tr>
<td>definesurfcbc.F</td>
<td>Includes the routines for finding coupling nodes on planar surfaces.</td>
</tr>
<tr>
<td>definepolcbc.F</td>
<td>Includes the routines for finding coupling nodes on circular coupling surfaces, e.g. a tubeflow</td>
</tr>
<tr>
<td>definediffcbc.F</td>
<td>Includes the routines for finding coupling nodes on conical coupling surfaces like diffusors</td>
</tr>
<tr>
<td>definetfsurface.F</td>
<td>Includes the routines for finding coupling nodes on freeform coupling surfaces</td>
</tr>
<tr>
<td>TFI.F</td>
<td>Transformation from logical coordinates to real coordinates by the use of TFI</td>
</tr>
<tr>
<td>functions.F</td>
<td>Subroutines for the TFI coupling</td>
</tr>
<tr>
<td>defineSpline.F</td>
<td>Subroutine that conditions the user-definitions of a spline from the FEAP input file</td>
</tr>
<tr>
<td>alloctfiarray.F</td>
<td>Subroutine that allocates the needed arrays for the TFI</td>
</tr>
</tbody>
</table>

Table 4.1: List of all needed files

All these files have to be put in the
/FEAP75/src/user/MpCCI/
folder. Afterwards FEAP has to be compiled and the following functions can be used. The sources can be found in the Appendix C-M.

4.2 General Workflow

The implemented functions have a common general workflow, which will be described here. The user defines a coupling surface by naming parameters in the FEAP input file that describe the coupling surface uniquely. These can be points or radii, or support points for the TFI. This data will be edited as arrays to be used in the routines. Then each node of the geometry will be considered elementwise and checked whether it is a coupling node or not by computing its distance to the coupling face. If it is less than $\delta$ (parameter $\delta_{in}$ in the sources), it gets identified as a coupling node. As only eight nodes brick elements are considered here, a coupling element always has four coupling nodes. Since the nodes are considered element wise, the founded node number is the local one and needs to be transferred to global. This is done by the use of the orientation of the element. Then together with the element number one can get the global node numbers out of the nodes array. After each element has been checked, the coupling nodes and elements are stored in an array and sent to MpCCI.

That means that still each node and element are itemized, but the recognition is automated.
4.3 Implemented Functions

4.3.1 Planar Coupling Surfaces

A typical problem geometry in research and industry is a plane coupling surface, for example a fluid acting on a straight wall. The function PLAN provides the recognition of coupling nodes on a plane surface just by the use of three points \(P_1(x_1,y_1,z_1), P_2(x_2,y_2,z_2)\) and \(P_3(x_3,y_3,z_3)\) of the surface. FEAP then transforms the given points into the Hessian normal form and computes the distance between the defined plane and every node of the grid:

\[
dis = \frac{|(v_1 - a_1)\cdot n_1 + (v_2 - a_2)\cdot n_2 + (v_3 - a_3)\cdot n_3|}{\sqrt{n_1^2 + n_2^2 + n_3^2}}
\]  

(4.1)

where \(\vec{V}(v_1,v_2,v_3)\) is the vertex which distance shall be computed, \(\vec{A}(a_1,a_2,a_3)\) the position vector of one point of the surface and \(\vec{N}(n_1,n_2,n_3)\) the normal vector of the plane. If \(dis\) is less than \(\delta\), the node gets buffered as a coupling node.

The mentioned parameters \((P_1, P_2, P_3)\) have to be made available in the FEAP input file, after the last END command:

```
PLAN x1 y1 z1 x2 y2 z2 x3 y3 z3
```

This will be read after the command FCBC,init in the MACRo section is processed by FEAP.

4.3.2 Cylindrical Coupling Surfaces

In research topics, very often one has to consider tube flow analysis. They are also theoretically well known and can be compared to computations made manually. The functions can of course also be used for a plate e.g., where the fluid moves around as in 4.4.6. The function POLA has been developed for an easy recognition of coupling nodes on a cylindrical surface.

The workflow of finding coupling nodes in this case is based on computing each node’s distance of the geometry to the coupling surface.

Again a node is identified as coupling node, when the computed distance is less than \(\delta\). In order to make FEAP to search for the coupling nodes, the command FCBC,init has to be placed in the very beginning of the MACRo. Then FEAP reads the lines after the last END command where one has to provide the needed definitions for the coupling surface:

- Two points \(P_1(x_1,y_1,z_1)\) and \(P_2(x_2,y_2,z_2)\), defining the axis
- The direction \(d\) (1, 2 or 3) of the axis,
- The radius \(r\).

by the use of the command

```
POLA x1 y1 z1 x2 y2 z2 r d
```

The determination of the distance is done by the computation of the distance of a point \(\vec{T}(t_1,t_2,t_3)\) to a straight line (axis) defined by the two points \((P_1\) and \(P_2)\):

- Computation of the direction vector \(\vec{v}\) given by \(P_1\) and \(P_2\).
- Construction of a plane through the testing point \(\vec{T}\) whereas the normal vector is \(\vec{v}\).
- Computation of the intersection \(P'\) between the plane and the axis.
- Computation of the distance between the two nodes \(P'\) and \(T\).
4.3.3 Coned Coupling Surfaces

Very often when dealing with tube flow problems, the profile gets reduced or expanded. These geometries are known as diffusers and are often considered in research and industry too. The function DIFF provides the automatic recognition of coupling nodes on coned surfaces like diffusers.

The selection of the coupling nodes here is similar to the selection at circular faces with the difference, that the distance of the coupling nodes to the axis differs at each point of the axis. To recognise the correct nodes, the expected distance of a coupling node is compared to the actual. As we know the x₁ position of the testing node $T(x_1, y_1, z_1)$, and the points $R_1$ and $R_2$ we also know its $x_2$ position and therefore the expected distance. This shall be illustrated in fig. 4.1.

![Computation of the expected distance for coned surfaces](image)

The needed parameters here are

- Two points $P_1(x_1, y_1, z_1)$ and $P_2(x_2, y_2, z_2)$ defining the axis,
- The origin or (1, 2 or 3) of the axis,
- The two radii ($r_1$ and $r_2$),

and have to be placed after the MACRo by the use of the command

DIFF x₁ y₁ z₁ x₂ y₂ z₂ or r₁ r₂

4.3.4 Freeform Coupling Surfaces

Sometimes it is not possible to define a surface just by the use of the geometries described above. The TFI can be used in order to describe freeform surfaces. In this case, the user only has to provide functions for the curves surrounding the surface. The implemented functions provide a definition of the curves by the use of a linear interpolation or cubic splines, which could also be easily expanded.

Here the identification of a coupling node is a little bit different: first a logical mesh, with the definitions of the input file is generated. This mesh is transferred via TFI into a physical mesh, which should match the coupling surface of the geometry. The coordinates of these nodes are stored. Then FEAP computes the difference (distance) of a testing node to each node of the TFI surface. If it is less than $\delta$ it is identified as a coupling node.

In details: first of all the input data of the spline definitions get transferred in order to get an equation with which each point of the spline can be computed. As also non equidistant grid points are allowed, first the length of $i$-th interval is determined. Then a linear equation system with the equations of each interval is set up. As we have cubic splines here, we have four unknowns. For each spline we have two points (the starting and ending point) and the derivatives as well. Therefore each spline can be identified exactly.

The next step is the TFI where the logical mesh is transferred to the physical mesh. This is done by the use of the equation described in 2.8. As a result we have the physical coordinates that are buffered.

FEAP checks every node’s distance to the surface, buffers matching nodes and stores their global coordinates for sending to MpCCI.

The parameters the user has to provide are

- The coordinates of the vertices $\vec{v}_1 - \vec{v}_8$. 
The curves $C_1 - C_8$,

The type of each curve (linear or spline).

To make FEAP to read the TFI definitions, instead of FCBC,init one has to write TFIC,init in the very beginning of the MACRo. The parameters are provided in following configuration (again placed after the END):

```
NODE, 8
  1 x1 y1 z1 ! nodenumber and coordinates
  4 x4 y4 z4
...
```

Before defining the curves, the user has to specify the discretisation and total amount of curves as well as the surface that will be described via TFI.

```
CURV, ncurve, dis1, dis2, nTFI ! number of curves, discretisation in 1-direction, ! discretisation in 2, number of TFI-Surface
```

The numbering (nTFI) can be found in (fig. 4.2):

```

gpiknaf

Figure 4.2: Numbering of the box - Needed to define the TFI Surface
```

When defining a curve, the user has to provide the node numbers ($nn$) of the vertexes ($n_1$ and $n_2$) that are connected and whether it is linear (1) or a spline (3). If a spline should be defined, the user has to send more parameters. These are the number of support points ($n_{sp}$) of the spline, the initial and final bending ($in_b$ and $fin_b$) and the axis of the constant values ($ax_1$) and the axis with the variable values ($ax_2$):

```
nn n1 n2 1 ! Linear connection of node 1 with node 2
nn n1 n2 3 n_sp in_b fin_b ax1 ax2 ! Spline between node n1 and n2
```

The coordinates ($x_s, y_s, z_s$) of the support points have to be placed directly after a spline definition command. The numbering ($spn$) of the following support points has to start with 1 again:

```
spn xs ys zs
```

A whole numerical example will be presented in (4.4.4).
4.4 Examples

4.4.1 Planar example

As an example we will have a look at a bevel cube illustrated in figure 4.3. The solid is on the left side and bevelled, the fluid on the right hand side.

![Figure 4.3: Example of a planar coupling surface](image)

The input file for this geometry is:

```
FEAP
0 0 0 3 4 8

PARAmeter
nx=8
ny=8
nz=8
x1=0.1
y1=0.0
z1=0.0
x2=0.2
y2=0.1
z2=0.1
x3=0.05

BLOCK
CARTesian nx ny nz 0 0 1 10
1 x1 y1 z1
2 x2 y1 z1
3 x2 y2 z1
4 x1 y2 z1
5 x3 y1 z2
6 x2 y1 z2
7 x2 y2 z2
8 x3 y2 z2

BLOCK
CARTesian ny nz 0 0 2 1 0
1 x1 y1 z1
2 x1 y2 z1
```

4 Implemented functions
First the parameters (measures and discretisation) are set. Afterwards the BLOCK command with the eight nodes creates the box and another BLOCK command sets the material 2 to the coupling surface for using a thermal flux boundary condition. Other definitions like the materials, displacements, boundaries are taken out for clearance. In the last step the MACRo calls the FCBC command and the parameters after END are read, where three of the four vertices of the coupling plane are set.

The velocities $u$, $v$, and $w$ of the fluid were set to zero, so just a stationary heat transfer between the solid and the fluid is considered here. The solid is modelled as a thermo mechanical isotropic body with the thermal diffusivity $\lambda = 45\, \text{W/m}^2\text{K}$ (steel) and the fluid with a heat conductivity of $\lambda = 0.600176\, \text{W/m}^2\text{K}$ (water). The heat from the left side of the solid gets transferred to the coupling surface and there it heats up the fluid. So one can see the convective heat transfer which qualitatively looks normal due to the greater heat conductivity in the solid (fig. 4.4).

Figure 4.4: Results of the planar coupling: Temperature plot of the geometry in Kelvin - Slice in the x-z-plane through the middle of the geometry
4.4.2 Cylindrical example

As an example of a cylindrical coupling surface we will have a look at a tube with a fluid inside (see fig. 4.5). Again just a stationary heat transfer is considered here. The tube is 2 metres long, the inner radius is 1 metre, the outer 2 metres:

![Figure 4.5: Example of a polar geometry](image)

For the computation, the outer surface of solid has an initial temperature of 200°K, the Fluid 300°K. The input file of the FEAP geometry:

```
FEAP
0 0 0 3 4 8

SNODES
1 0 0 0
2 0 0 1
3 0 1 0
4 0 0 2
5 0 2 0
6 2 0 1
7 2 1 0
8 2 0 2
9 2 2 0
10 2 0 0
11 0 0 -1
12 0 0 -2
13 2 0 -1
14 2 0 -2
15 0 -1 0
16 0 -2 0
17 2 -1 0
18 2 -2 0

SIDE
POLAr 3 2 1
POLAr 5 4 1
POLAr 7 6 10
POLAr 9 8 10
POLAr 3 11 1
POLAr 5 12 1
```
First we have a definition of supernodes that are used for the polar curve definitions afterwards. A POLAr n1 n2 c command creates a circular side between n1 and n2 with the centre c. As this works only for a quarter of a circle 16 commands (eight for each side) are necessary. Then the BLENd - SOLId command creates solid body by the use of 8 nodes.
and BLEND - SURFace creates surfaces with Material 2 to simulate the heat conductivity there. In the MACRo again the 
FCBC command makes FEAP to read the last lines where the parameters for the coupling surface are found.

![Overview - Slice in the x-y-plane through the middle of the geometry](image1)

![Detail inlet and diffuser - Slice through the x-y-plane in the middle of the geometry](image2)

Figure 4.6: Results of the tubeflow - Temperature plot in Kelvin

The material and fluid properties were the same as in the planar example (steel and water). One can see, that the solid near the coupling face heats up and the fluid cools down respectively and the coupling succeeded. The temperature profile qualitatively looks normal as the heat conductivity in the solid is greater.
4.4.3 Coned example

As an example we will have a look at a simple coned geometry with a cold fluid inside and a hot solid around it illustrated in fig 4.7. The smaller radius is 1 metre, the greater 3 metres and its total length is 2 metres. The fluid’s initial temperature is 300 °K and the outer surface of the solid is 310°K and again the same material/fluid properties are used. We consider stationary heat transfer here. One can see in (b) that the fluid heats up towards the solid whereas the solid cools down and the coupling succeeded. The fluid near the coupling surfaces heats up on quite a small distance whereas the heat transfer through the solid is widespread over a longer distance due to the greater heat conductivity.

![Figure 4.7: Geometry and results of the coned example](image)

The FEAP input file is nearly the same as in 4.4.2 just with different coordinates of the supernodes and is therefore not presented completely. The greater radius of the Diffuser is 3 metres, the other 1 metre as before. Therefore the parameters for the coupling functions are:

```
1 FEAP
2 0 0 0 3 4 8
3
4 SNODes
5 6 2 0 3
6 7 2 3 0
7 8 2 0 4
8 9 2 4 0
9 13 2 0 -3
10 14 2 0 -4
11 17 2 -3 0
12 18 2 -4 0
13 ...
14
15 END
16
17 MACR
18 FCBC, INIT
19 ...
20 END
21
22 DIFF,0,0,0,2,1,0,0,1,1,3
```

Listing 4.3: Input file for freeform coupling faces
4.4.4 Freeform example

In order to use a freeform geometry the user has to provide more data to specify the the geometry. One has to define all nodes and curves together with needed support points for the splines. This shall be illustrated by the use of the figure fig. 4.8.

![Figure 4.8: Example of a freeform geometry](image_url)

The geometry is a box with a rolling top. The edges are of a length of four metres. The spline definitions were provided by three support points, equally spaced. The highest point is 4.25 metres high, the lowest 3.75 metres marked with a circle in fig. 4.8. The nodes are numbered as shown, the TFI surface number is 3.

In order to describe this geometry, the input file has to look as shown in 4.4.

```
1 FEAP
2 0 0 3 4 8
3
4 COORDinates
5  1 1 0.0 0.0 0.0
6  9 0 4.0 0.0 0.0
7 10 1 0.0 0.5 0.0
8 18 0 4.0 0.5 0.0
9 ...
10
11 ELEMENTs
12  1 0 1 1 2 11 10 82 83 92 91
13  2 0 1 2 3 12 11 83 84 93 92
14 ...
15  513 0 2 649 650 659 658
16  514 0 2 650 651 660 659
17 ...
18
19 END
20 TIE
21 MACR
22  TFIC, INIT
23 ...
24 END
25 NODE, 8
26
```
The geometry is defined nodewise and each element individually. Therefore the input file has been shortened. What we can see in the definition of the coupling parameters are the splines connecting node 6 with 7 and 5 with 8, respectively, together with the support points (marked with circles in fig. 4.8) afterwards.

If we now assume a cold fluid (500°K) (water) on the top of the surface (see Fig. 4.8), and a temperature of 700°K at the bottom of the solid (steel) we expect a similar behaviour like the other geometries. The geometry of both, the fluid and the solid are presented in fig. 4.8 and the results in fig. 4.9

As one can see, the heat is transported from the bottom of the solid to its top, and then transferred to the fluid above. The coupling succeeded which can be qualitatively seen in fig. 4.9

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Listing 4.4: Input file for freeform coupling faces**

```plaintext
CURV, 12, 9, 9, 3
```

Figure 4.9: Results of the tube
## 4.4.5 Comparative Simulation

In order to evaluate the functions, a former computation which was made with the node- and elementwise, definition should be compared to a computation using the developed functions. The geometry, boundary and material definitions were identical, just like the fluid properties. Only the MpCCI definitions in the FEAP-input file were edited. What we expect is the same amount of coupling nodes and elements and therefore identical solutions.

![Figure 4.10: Geometry - Solid diffuser with fluid inside](image)

The geometry in figure 4.10 is a tubeflow with a diffuser. The outer geometry is the solid, the inner the fluid. The origin is in the centre of the the small radius on the outer side. It has a length of 0.1 metre and its radius is 0.01 metre. The diffuser has a length of 0.02 metre and ends with a radius of 0.03 metre. The larger tube has a length of 0.3 metre.

The coupling surface is the whole inner surface of the solid and the outer surface of the fluid respectively. We have an inlet flow at the side with the smaller radius and a outlet at the other side. The velocity at the inlet in x-direction is 0.073 m/s, no velocity in y- and z-direction. The temperature is 1000°K at the inlet boundary. The solid tube has an initial temperature of 450 K. The flow has been modelled with the use of the k-ε-model. The Reynoldsnumber was 5000. All other fluid properties can be found in appendix [A].

Now we will have a look at the FEAP input files. The geometry definitions derive from an export of a CAD Tool. Each node and element is defined individually. The coupling material was defined element wise and the boundary conditions also node wise. As these are several thousand lines of code, they are not presented here.

The definitions for the coupling surface was in the former input file a list of all coupling nodes. Now the functions POLA and DIFF can be used to recognise the coupling nodes automatically:

```plaintext
MACR
NOPRint
DT, 1.0e10
FCBC, INIT
MPCCI, INIT
TRANSient, BACK
LOOP, ts, 1
TECP, init, 1
TIME
LOOP, fsiiter, 50
RECV, HEAT
RECV, FORC
LOOP, newton, 300
TANG, 1
next, newton
SEND, TEMP
SEND, DISP
CONVergency
next, fsiiter
TECP, write, 1, 5600, 4080
```

### 4 Implemented functions
As it is a known quirk of FEAP, one should leave some clear lines after the last definition to make it work. That is all that has to be changed in the input file.

As the computation will not start, as long as MpCCI has not a valid partner for each node, one can say, that as soon as MpCCI does not report any errors, the coupling succeeded. The amount of founded coupling nodes and coupling elements can be found in the feap.out-file and can be compared to the amount of nodes in the former declaration or at least check if it is in a realistic range. The FCBC-functions founded 1400 coupling nodes and 1360 coupling elements, which is exactly the same amount as in the former definition. Of course it may be that the computation diverges in case of inconsistent definitions, but this is not matter of this thesis.

To illustrate the coupling we will have a look at the results of the computation in fig. 4.11. They are identical to the solutions with the former coupling surface definition. Therefore one can say, that the functions provide the expected functionality.

As one can see in image (b), the solid heats up at the inlet where the inlet flow is hot (1000\textdegree K). Another typical behaviour can be found in the transition from the diffuser to the greater tube. The solid does not heat up as the hot flow does not reach this part. Additionally one can see, that the the fluid cools down at the coupling surface.
4.4.6 Incident Flow on both Sides of a Plate

Another example where the functions are used is a computation of my supervisor P. Pironkov. He analysis the incident flow on both side of a plate, whereas the fluid is hot on one side and cold on the other. Image 4.12 should illustrate the geometry. The plate in the centre is the solid geometry provided by FEAP The box simulates a infinite stretched room as all sides have adiabatic boundary conditions. The inlet flows were applied in the two tubes, positioned at the same level as the plate.

![Image 4.12: Geometry - Incident Flow an a Plate](image)

The velocity at the inflow is 26.1 m/s and -7.19 m/s, the temperature 640°K and 294°K respectively. As turbulence model the k-ε-model has been adopted as well with a Reynoldsnumber of the fluid of 5000. The used FEAP input file can be found in appendix [B]. In order to avoid bad elements a block structured mesh has been applied here. That was implemented by the use of five blocks. One cuboidal block in the centre and four blocks around it whereas the outer sides are circular and the inner straight. The POLA command provides the recognition of the coupling nodes on the cylindrical surface whereas the PLAN commands recognise thecoupling nodes on the plane surfaces where the flow impacts.

As the computation is still in progress no results are presented here but the coupling succeeded as the computation started successfully.
5 Conclusion

The task of the present Bachelor Theses was to implement various functions that provide an automatic recognition of boundary conditions. The developed functions should be applied for the use of FEAP and FASTEST in thermal FSI by the use of the coupling interface MPCCI.
This was done for simple geometries like plane, polar or coned as well as free-form surfaces by the use of some user provided parameters.

5.1 Evaluation

As shown in 4 the functions provide an automatic recognition of coupling nodes for the mentioned geometries. In Chapter 4.4, the multiple usage of the commands have been proved. Therefore one can say, that the implemented functions fulfil the requirements. A great advantage now is, that in case of a mesh refinement or a change in the geometry, the definitions of the coupling surfaces do not have to change. The user is more flexible in the use of the solid geometry. As MpCCI will not start a computation if not all coupling nodes are well defined, one can say that coupling succeeded as soon as the computation starts. The results then do not depend on the used functions, but on the boundary conditions of the solid and the properties of the fluid flow.

By the use of freeform surfaces one can also describe several more complicated surfaces that can be described by the TFI. If a geometry does not fit any provided functions, new routines can be implemented straight forward or existing routines can be extended.

5.2 Prospect

The implemented functions can easily be extended if for example a spline definition for the TFI does no match the given geometry or cannot be described by any function. Also other geometries like spherical elliptical ones could be implemented as standard geometries.

As the boundary and displacement definitions still need to be set nodewise, the implemented functions could be used for these definitions as well. The implementation would be straight forward and be very useful for the user. Then FEAP would again be a powerful tool for FSI.

Another feature of the implemented functions are, that the functions developed for the TFI can also be used for grid generation. The geometry in 4.4.4 has been created by the use of the implemented functions. Hence with a few modifications a TFI grid generator could be adopted as well.
Bibliography


A FASTEST id-file

The FASTEST id-file of the coupled FSI simulation tubeflow with diffuser. All parameters are well commented.

```plaintext
### parameters for large eddy simulation
2 ;[1|2] determine filterwidth by cell [vol|area]
7 ;[7|27] number of adjacent cells used for test filtering
0.5 ;underrelaxation for germano parameter

### subgrid scale variance model
3 equil [equilib|scalesim|res_var]

### gravity
1 1 ;no. of coarse / fine grid

### geometric scale
1.0 1.0 1.0
```

### turbulence model
keps74 ;according to the above choice of [turb|rsm|visles]

### scalar flux model
1s [hp|ls|jm]

### write output
0 0 1 ; restart ascii visual (0 = no output)

### file formats for visual output
tecplot6 ;[tecplot6|tecplot6temp]

### write additional output
0 0 0 0 0 ; louts lsurf ldebug lboundc llagr lvogd

### visual output variables
vel pres temp ;[vel][pres][tke][edis][temp][conc]

### ascii output variables
velx vely velz pres tke edis temp conc

### ascii output geometry
[x][y][z][xc][yc][zc][vol]

```
### small
1.0 \times 10^{-30}

### residuum limits
1.0 \times 10^{-4} \quad 1.0 \times 10^{10}

### interpolation method
cds ; [cds|tbi]

### flux blending
0.0 \quad 0.0 \quad 1.0 \quad 0.0 \quad 0.0 \quad 0.0 \quad 0.0 \quad 0.0 \quad 0.0 \quad 0.0 \quad 0.0 \quad 0.0 \quad 0.0 \quad 0.0 \quad 0.0

### underrelaxation
0.6 \quad 0.6 \quad 0.6 \quad 0.5 \quad 0.9 \quad 0.9 \quad 0.5 \quad 0.3 \quad 0.2 \quad 0.9 \quad 0.9 \quad 0.9 \quad 0.9 \quad 0.9 \quad 0.3 \quad 0.9 \quad 0.9

### sipsol
0.92 ; alfa
0.5 \quad 0.5 \quad 0.5 \quad 0.0 \quad 0.0 \quad 0.0 \quad 0.0 \quad 0.0 \quad 0.0 \quad 0.5 \quad 0.0 \quad 0.0 \quad 0.0 \quad 0.0 \quad 0.0 ;

### multigrid cycle definition
10000

### number of multigrid cycles
100 \quad 100 \quad 100

### time discretization
sofi ; discretization method [fofi|sofi|crni]
1 \quad 1.0 \times 10^{10} ; no of timesteps, size of timestep

### heat production by dissipation
n
### bouyancy by temperature gradient
boussinesq ; [none|boussinesq|dengrad]

### fluid regions
t ; for each flow region: t – fluid, f – solid

### moving grids
y ; lmvgr
n ; elliptic non-orthogonal, falls y 2 Zeilen einkommentieren
6 ; number of blocks
3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 ; block numbers in increasing order
y ; elliptic orthogonal, falls y 2 Zeilen einkommentieren
6 ; number of orthogonal–blocks
3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 ; block numbers in increasing order!!

### fluid structure interaction
y \quad y \quad y ; lfsi lfsiread lfsiwrite
n ; estimation of distortions
50 ; max. number of outer iterations
2 ; coupling interface (0=none, 1=GRISSLi, 2=MpCCI)
1 ; interpolation scheme (0=NNB, 1=CONS)
0.5 \quad 0.0 ; undrel. param for struct distortions, aitken damping fac
0.1 \quad 0.1 \quad 0.1 ; fsi–residuum limit (x,y,z–direction)
n ; fsi predictor
1.0 \times 10^{-5} ; convergence criterion for the TFSI
### mpcci input data

1. underrelaxation parameter for the TFSI
2. Number of blocks to distort

1 1 0 0 0 0 0 0

### time limit

1. e30 ;cpu-time in seconds

### pressure gradient

0. 0. 0. ;pressure gradient in x,y,z direction

### pressure correction steps

1 ;feature not yet implemented !!!

### tasks per process

1 ;for each process (processor)

### convective exit boundary condition

0 ;switch

### turbulence statistics

none ;average method [none|plane|line|time]

t ;higher order statistics (skewness,flatness)

100 10 ;starting time (timestep), step (timesteps)

(500 3) ;starting time, step

### residuum norming

0.02935d0 2570.d0 0.1d0 0.1d0 0.1d0 1710.d0 1.d0 ;area,rho,vel,tui,len,temp,conc

Listing A.1: FASTEST id-file for the tubeflow with diffuser
B  FEAP Input File for the Incident Flow Simulation

The FEAP input file of the incident flow simulation.

```plaintext
FEAP
0 0 0 3 4 8

SNODES
1 0 0 0
2 0.02625 0 0
3 0 0.02625 0
4 -0.02625 0 0
5 0 -0.02625 0
6 0.013125 0 0
7 0 0.013125 0
8 -0.013125 0 0
9 0 -0.013125 0
10 0 0 0.00525
11 0.02625 0 0.00525
12 0 0.02625 0.00525
13 -0.02625 0 0.00525
14 0 -0.02625 0.00525
15 0.013125 0 0.00525
16 0 0.013125 0.00525
17 -0.013125 0 0.00525
18 0 -0.013125 0.00525

SIDE
POLAR 2 5 1
POLAR 5 4 1
POLAR 4 3 1
POLAR 3 2 1
POLAR 6 9 1
POLAR 9 8 1
POLAR 8 7 1
POLAR 7 6 1
POLAR 11 14 10
POLAR 14 13 10
POLAR 13 12 10
POLAR 12 11 10
POLAR 15 18 10
POLAR 18 17 10
POLAR 17 16 10
POLAR 16 15 10

BLEND 1
SOLID 5 5 15 0 0 1
6 2 3 7 15 11 12 16

BLEND 2
SOLID 5 5 15 0 0 1
4 8 7 3 13 17 16 12

BLEND 3
SOLID 5 5 15 0 0 1
```

B  FEAP Input File for the Incident Flow Simulation
BLENd 18
  SURFace 15 5 0 0 2
    4 13 14 5

BLENd 19
  SURFace 15 5 0 0 2
    5 14 11 2

END

TIE

MACR
  FCBC, INIT

... 

END

POLA,0,0,0,0,0,0.00525,0.02625,3

PLAN,0,0,0,0.02625,0,0,0,0.02625,0

PLAN,0,0,0.00525,0.02625,0,0.00525,0,0.02625,0.00525

Listing B.1: FEAP Input File for the Incident Flow on a Plate
C Sourcecode of usermacro 6

Source code of the usermacro for plane, polar and coed coupling.

```fortran
subroutine umacr6 (lct,ctl,prt)

* * F E A P * * A Finite Element Analysis Program

Copyright (c) 1984–1998: Robert L. Taylor

Purpose: MpCCI commands for initialization, termination and
data exchange (send/recv).

Inputs:
- lct  — Command character parameters
- ctl(10) — Command numerical parameters
- prt  — Flag, output if true

Outputs:

implied none

include 'iofile.h'
include 'cdata.h'
include 'sdata.h'
include 'umac1.h'
include 'pointer.h'
include 'upointer.h'
include 'comblk.h'
include 'mpcci.h'

integer iError, i, dir
logical pcomp,prt,prth, pinput, errck, tinput
logical setvar,ualloc
integer nNodesPerElem
parameter(nNodesPerElem = 4)
character lct*15, yyy*15
real*8 ctl(10), axis, td(12)
save

prth = .true.

if(pcomp(uct,'mac6',4)) then
  uct = 'fcbc'
elseif(pcomp(lct,'init',4)) then
  setvar = ualloc(2, 'CCI_C', numnp, 1) ! scratch
```
setvar = ualloc(4, 'CCI_H', numnp=nNodesPerElem, 1) ! scratch
setvar = ualloc(5, 'CCI_M', numnp, 1) ! MAP FEAP -> CCI nodes

errck = tinput(yyy,1,td,12)
123 if (.not. errck) then
  errck = tinput(yyy,1,td,12)
  if(pcomp(yyy,'plan',4)) then
    write(*,*,'("Planar coupling")')
    call defineSurfaceCouplingBC(mr(np(33)), hr(np(43)),$
      td(1), td(2), td(3), td(4), td(5),
      td(6), td(7), td(8), td(9),
      mr(up(2)), mr(up(4)), mr(up(5)), nNodes, nElems)
    write(*,*,'("nNodes:", nNodes, " - nElems:", nElems')
    goto 123
  elseif(pcomp(yyy,'pola',4)) then
    write(*,*,'("Polar coupling")')
    call definePolarCouplingBC(mr(np(33)), hr(np(43)), td(1),$
      td(2), td(3), td(4), td(5),
      td(6), td(7), td(8),
      mr(up(2)), mr(up(4)), mr(up(5)), nNodes, nElems)
    write(*,*,'("nNodes:", nNodes, " - nElems:", nElems')
    goto 123
  elseif(pcomp(yyy,'diff',4)) then
    write(*,*,'("Coned coupling")')
    call defineDiffusorCouplingBC(mr(np(33)), hr(np(43)),$
      td(1), td(2), td(3), td(4), td(5),
      td(6), td(7), td(8),
      mr(up(2)), mr(up(4)), mr(up(5)), nNodes, nElems)
    write(*,*,'("nNodes:", nNodes, " - nElems:", nElems')
    goto 123
  elseif(pcomp(yyy,'0',4)) then
    goto 345
  endif
345 endif

c optimize the allocated space

setvar = ualloc(1, 'CCI_N', nNodes, 1) ! cci_node
call pmovei(mr(up(2)), mr(up(1)), nNodes) ! CCI_C --> CCI_N
setvar = ualloc(2, 'CCI_C', 0, 1) ! free scratch
setvar = ualloc(3, 'CCI_E', nNodesPerElem*nElems, 1)
  call pmovei(mr(up(4)), mr(up(3)), nNodesPerElem*nElems)
c ! CCI_H --> CCI_E
setvar = ualloc(4, 'CCI_H', 0, 1) ! free scratch
D Sourcecode of usermacro 7

Source code of the usermacro for TFI coupling.

```fortran
subroutine umacr7(lct,ctl,prt)

c * * F E A P * * A Finite Element Analysis Program

c .... Copyright (c) 1984–1998: Robert L. Taylor

meynen 24.02.03

c Purpose: MpCCI commands for initialization, termination and
data exchange (send/recv).

c Inputs:
lct (command character parameters)
ctl(10) (command numerical parameters)
prt (flag, output if true)

c Outputs:

implicit none

include 'iofile.h'
include 'cdata.h'
include 'sdata.h'
include 'umac1.h'
include 'pointer.h'
include 'upointer.h'
include 'comblk.h'
include 'mpcci.h'

integer iError, i, dir, scratch, temp,j,ii, offset
logical pcomp,prt,prth, pinput, errck, tinput
logical setvar, ualloc
integer nNodesPerElem
parameter(nNodesPerElem = 4, scratch = 10)
character lct=15, yyy=15
real*8 ctl(10), td(12), nodesco(scratch,3)
real*8 sppoints(scratch*10, 4), tempfirst, templast, tempnum
real*8 tempj, initialbend, finalbend, direqu, dirval
real*8 x, y, z
integer tempnn(12), dis1, dis2, surfacenumber
save

do i = 1, scratch*10
  do j = 1, 3
    sppoints(i,j) = 0
  enddo
```

D Sourcecode of usermacro 7
enddo

do i = 1, 12
    tempnn(i) = 2
enddo

prth = .true.

if(pcomp(uct,'mac7',4)) then
    uct = 'tfic'
elseif(pcomp(lct,'init',4)) then

    setvar = ualloc(2, 'CCI_C', numnp, 1) ! scratch
    setvar = ualloc(4, 'CCI_H', numnp*nNodesPerElem, 1) ! scratch
    setvar = ualloc(5, 'CCI_M', numnp, 1) ! MAP FEAP -> CCI nodes

    errck = tinput(yyy,1,td,12)
    if(.not. errck) then
        write(*,*') yyy
    if(pcomp(yyy,'node',4)) then
        temp = td(1)
        do i = 1, temp
            errck = pinput(td,4)
            nodesco(i,1) = td(2)
            nodesco(i,2) = td(3)
            nodesco(i,3) = td(4)
        enddo
        goto 123
    elseif(pcomp(yyy,'curv',4)) then
        temp = td(1)
        dis1 = td(2)
        dis2 = td(3)
        surfacenumber = td(4)
        tempj = 1
        do i = 1, temp
            errck = pinput(td,12)
            tempfirst = td(2) ! first node of the curve
            templast = td(3) ! last node if the curve
            tempnum = td(5) ! number of support points
            initialbend = td(6)
            finalbend = td(7)
            direqu = td(8)
            dirval = td(9)
            tempnn(td(1)) = td(4)+1 ! number of nodes for i-th curve
            if (td(4) .eq. 1) then ! linear
                write(*,*') "linear spline def"
                sppoints(tempj,1) = nodesco(tempfirst,1)
                sppoints(tempj,2) = nodesco(tempfirst,2)
                sppoints(tempj,3) = nodesco(tempfirst,3)
                sppoints(tempj+1,1) = nodesco(templast,1)
                sppoints(tempj+1,2) = nodesco(templast,2)
                sppoints(tempj+1,3) = nodesco(templast,3)
                tempj = tempj+2
            endif
        if (td(4) .eq. 3) then ! cubic Spline
write( *, * ) "cubic spline definition"
sppoints(tempj, 1) = nodesco(tempfirst, 1)
sppoints(tempj, 2) = nodesco(tempfirst, 2)
sppoints(tempj, 3) = nodesco(tempfirst, 3)
sppoints(tempj, 4) = initialbend
do j = tempj+1, tempj+td(4)
   errck = pinput(td, 3)
   sppoints(j, 1) = td(1)
   sppoints(j, 2) = td(2)
   sppoints(j, 3) = td(3)
endo
sppoints(tempj+td(4)+1, 1) = nodesco(templast, 1)
sppoints(tempj+td(4)+1, 2) = nodesco(templast, 2)
sppoints(tempj+td(4)+1, 3) = nodesco(templast, 3)
sppoints(tempj+td(4)+1, 4) = finalbend
tempj = tempj + td(4) + 2
endif
enddo
goto 123
elseif (pcomp(yyy, '0', 4)) then
goto 345
endif
345 endif
setvar = ualloc(103, 'nump', 12, 2)
do i = 1, 12
   hr(up(103)+i-1) = tempnn(i)
   if (tempnn(i) .gt. 2) tempnn(i) = tempnn(i)+1
endo
call alloctfiarray(tempnn)
offset = 0
do i = 1, 12
   if (tempnn(i) .gt. 3) then ! spline
call columnmove(sppoints, 1, hr(up(3*(i+1)+1)), tempnn(i), offset)
   $ call columnmove(sppoints, 2, hr(up(3*(i+1)+2)), tempnn(i), offset)
   $ call columnmove(sppoints, 3, hr(up(3*(i+1)+3)), tempnn(i), offset)
   $ call definespline(hr(up(3*(i+1)+1)), hr(up(3*(i+1)+2)),
         $ hr(up(3*(i+1)+3)), sppoints(i, 4),
         $ sppoints(tempnn(i), 4), tempnn(i), surfacenumber,
         $ hr(up(4*(i+9)+3)), hr(up(4*(i+9)+4),
         $ hr(up(4*(i+9)+5)), hr(up(4*(i+9)+6)))
   elseif (tempnn(i) .eq. 2) then ! linear
      call columnmove(sppoints, 1, hr(up(3*(i+1)+1)), tempnn(i), offset)
      $ call columnmove(sppoints, 2, hr(up(3*(i+1)+2)), tempnn(i), offset)
      $ call columnmove(sppoints, 3, hr(up(3*(i+1)+3)), tempnn(i), offset)
      endif
      offset = offset + tempnn(i)
endo
setvar = ualloc(104, 'tfisurf ', dis1*dis2, 2)
setvar = ualloc(105, 'tfisurf ', dis1*dis2, 2)
setvar = ualloc(106, 'tfisurf ', dis1*dis2, 2)

c  now TFI
  do i = 1, dis1
    do j = 1, dis2
      call TFI((i-1)*(1d0/(dis1-1)) , (j-1)*(1d0/(dis2-1)) ,
        1d0,x,y,z)
      hr(up(104)+(i-1)*dis1+j-1) = x
      hr(up(105)+(i-1)*dis1+j-1) = y
      hr(up(106)+(i-1)*dis1+j-1) = z
    enddo
  enddo

  c  Now search the coupling nodes!
  call define tfisurf ace(mp(np(33)), hr(np(43)), dis1, dis2,
    mr(up(2)), mr(up(4)), mr(up(5)), nNodes, nElems)

  c  optimize the allocated space
  setvar = ualloc(1, 'CCI_N', nNodes, 1) ! cci_node
  call pmovei(mr(up(2)), mr(up(1)), nNodes) ! CCI_C --> CCI_N
  setvar = ualloc(2, 'CCI_C', 0, 1) ! free scratch
  setvar = ualloc(3, 'CCI_E', nNodesPerElem*nElems, 1)
  call pmovei(mr(up(4)), mr(up(3)), nNodesPerElem*nElems)
  c  ! CCI_H --> CCI_E
  setvar = ualloc(4, 'CCI_H', 0, 1) ! free scratch
  call clean tfarray()
  write(*,*) "nNodes: ", nNodes, " -- nElems: ", nElems

  endif

  return
end

subroutine columnmove(Orig, column, Dest, length, offset)
  integer column, length, offset, i
  real*8  Orig(100,3), Dest(length)
  do i = 1, length
    Dest(i) = Orig(i+offset,column)
    if (length .gt. 3) then
      write(*,*) Dest(i), i+offset, column
    endif
doendo

  return
end

Listing D.1: umacr7.F
Sourcecode of functions for finding the orientation of an eight brick element.

```fortran
subroutine boundary8BrickElem(cciMap, cciElems, nNodes, nElems, orientation, source)

implicit none

include 'cdata.h'
include 'sdata.h'

integer cciMap(*), cciElems(*), source(*)
integer nNodes, nElems, orientation
integer addCouplingNode

local variables
integer index, temp
save

index = nElems*4

if (orientation.eq.1) then ! west coupling face
  temp = addCouplingNode(cciMap, source(1), nNodes)
  cciElems(index+1) = temp
  temp = addCouplingNode(cciMap, source(4), nNodes)
  cciElems(index+2) = temp
  temp = addCouplingNode(cciMap, source(8), nNodes)
  cciElems(index+3) = temp
  temp = addCouplingNode(cciMap, source(5), nNodes)
  cciElems(index+4) = temp
else if (orientation.eq.2) then ! south
  temp = addCouplingNode(cciMap, source(1), nNodes)
  cciElems(index+1) = temp
  temp = addCouplingNode(cciMap, source(2), nNodes)
  cciElems(index+2) = temp
  temp = addCouplingNode(cciMap, source(6), nNodes)
  cciElems(index+3) = temp
  temp = addCouplingNode(cciMap, source(5), nNodes)
  cciElems(index+4) = temp
else if (orientation.eq.3) then ! bottom
  temp = addCouplingNode(cciMap, source(1), nNodes)
  cciElems(index+1) = temp
  temp = addCouplingNode(cciMap, source(2), nNodes)
  cciElems(index+2) = temp
  temp = addCouplingNode(cciMap, source(3), nNodes)
  cciElems(index+3) = temp
  temp = addCouplingNode(cciMap, source(4), nNodes)
  cciElems(index+4) = temp
else if (orientation.eq.4) then ! top
  temp = addCouplingNode(cciMap, source(5), nNodes)
  cciElems(index+1) = temp
  temp = addCouplingNode(cciMap, source(6), nNodes)
  cciElems(index+2) = temp
  temp = addCouplingNode(cciMap, source(7), nNodes)
  cciElems(index+3) = temp
  temp = addCouplingNode(cciMap, source(8), nNodes)
end if
```

E Sourcecode of boundary8brickelem
cciElems(index+4) = temp
else if (orientation.eq.5) then ! north
  temp = addCouplingNode(cciMap, source(4), nNodes)
  cciElems(index+1) = temp
  temp = addCouplingNode(cciMap, source(3), nNodes)
  cciElems(index+2) = temp
  temp = addCouplingNode(cciMap, source(7), nNodes)
  cciElems(index+3) = temp
  temp = addCouplingNode(cciMap, source(8), nNodes)
  cciElems(index+4) = temp
endif
return
end

function addCouplingNode(cciMap, nodeindex, nNodes)
implicit none
include 'cdata.h'
include 'sdata.h'
integer addCouplingNode
integer cciMap(*), nodeindex, nNodes
integer i
save
if (cciMap(nodeindex).eq.0) then
  nNodes = nNodes + 1
  cciMap(nodeindex) = nNodes
endif
addCouplingNode = cciMap(nodeindex)
return
end

integer function findOrientation_8Brick(elementnodes, surfacenodes)
implicit none
integer elementnodes(*), surfacenodes(*), i
logical contains

contains
orientation: 1–W; 2–S; 3–B; 4–T; 5–N; 6–E
if (contains(surfacenodes, elementnodes(1), 4) .and.
  $ contains(surfacenodes, elementnodes(4), 4) .and.
  $ contains(surfacenodes, elementnodes(8), 4) .and.
  $ contains(surfacenodes, elementnodes(5), 4)) then
  findOrientation_8Brick = 1
else if (contains(surfacenodes, elementnodes(1), 4) .and.
  $ contains(surfacenodes, elementnodes(2), 4) .and.
contains(surfacenodes, elementnodes(6), 4) and 
contains(surfacenodes, elementnodes(5), 4) then

findOrientation_8Brick = 2
else if (contains(surfacenodes, elementnodes(1), 4) and 
contains(surfacenodes, elementnodes(2), 4) and 
contains(surfacenodes, elementnodes(3), 4) and 
contains(surfacenodes, elementnodes(4), 4)) then

findOrientation_8Brick = 3
else if (contains(surfacenodes, elementnodes(5), 4) and 
contains(surfacenodes, elementnodes(6), 4) and 
contains(surfacenodes, elementnodes(7), 4) and 
contains(surfacenodes, elementnodes(8), 4)) then

findOrientation_8Brick = 4
else if (contains(surfacenodes, elementnodes(4), 4) and 
contains(surfacenodes, elementnodes(3), 4) and 
contains(surfacenodes, elementnodes(7), 4) and 
contains(surfacenodes, elementnodes(8), 4)) then

findOrientation_8Brick = 5
else if (contains(surfacenodes, elementnodes(2), 4) and 
contains(surfacenodes, elementnodes(3), 4) and 
contains(surfacenodes, elementnodes(7), 4) and 
contains(surfacenodes, elementnodes(6), 4)) then

findOrientation_8Brick = 6
endif

return
end

logical function contains(source, element, n)
implicit none
integer source(*), element, n
integer i
contains = .false.
do i = 1, n
if(source(i) .eq. element) then
contains = .true.
goto 111
endif
111 continue
enddo
return
end

Listing E.1: boundary8brickelem.F
Source code of the functions for finding coupling nodes on plane surfaces.

```fortran
subroutine defineSurfaceCouplingBC(ix, x, x1, y1, z1, x2, y2, z2, x3, y3, z3, cciNodes, cciElems, cciMap, nNodes, nElems)

 Purpose: Define an coupling BC for rectangular surfaces.
 Inputs:
 ix          — the element connection array in feap
 x           — the coordinates array in feap
 xn, yn, zn  — The 3 nodes defining the Surface where the Boundary Condition should be applied

 Outputs:
 cciNodes   — Array with all nodes which lay on the boundary
 cciElems   — Array with all elements which lay on the boundary
 cciMap     — Array containing the connection FEAP index-> cci index (the opposite of cciNodes)
 nNodes     — number of interface nodes on the defined boundary
 nElems     — number of interface elements on the interface boundary

 implicit none
 include 'cdata.h'
 include 'sdata.h'
 include 'iofile.h'

 integer ix(nen1, *), cciNodes(*), cciElems(*), cciMap(*)
 integer nNodes, nElems
 real=8 x(ndm, *)
 real=8 x1,x2,x3,y1,y2,y3,z1,z2,z3

 local variables
 real=8 Vertex(3,3), n(3), b, TesVer(3)
 real=8 temp, dif, dis
 integer nNodesPerElem
 parameter (dif = 1.d−10, nnodesPerElem = 4)
 integer i, j, k, count, scount, orientation, axis
 c orientation: 1-W; 2-S; 3-B; 4-T; 5-N; 6-E
 integer couplelem(nen), surfaceelem(nen)
 integer findOrientation_8Brick
 logical flag
 save

 Vertex should be an array with the coordinates of the three nodes defining the rectangular contact surface
 HNF will compute the normal vector stored in n
 b is the scaling factor (computed in HNF)

 Vertex(1,1) = x1
```

F Sourcecode of definesurfcbc
Vertex(2,1) = y1
Vertex(3,1) = z1
Vertex(1,2) = x2
Vertex(2,2) = y2
Vertex(3,2) = z2
Vertex(1,3) = x3
Vertex(2,3) = y3
Vertex(3,3) = z3

call HNF(Vertex, n, b)
do i = 1, numel
  flag = .false.
  count = 0
  scount = 0
  do j = 1, nen
    if(ix(j, i) .eq. 0) then ! surface element
      flag = .false.
    else
      count = count + 1
      couplelem(count) = ix(j, i)
    endif
  enddo
axis = 1
if(flag) then ! coupling boundary element
  orientation = findOrientation_Brick(couplelem, surfaceelem)
  call boundaryBBrickElem(cciMap, cciElems, nNodes, nElems, orientation, couplelem)
  nElems = nElems + 1
endif
enddo

c now we have the filled cciMap. Construct the cciNodes array
do i = 1, numnp
  if(cciMap(i) .ne. 0) then ! coupling node
    cciNodes(cciMap(i)) = i
  endif
enddo

return
subroutine HNF(Vertex, n, b)
  implicit none
  real*8 Vertex(3,*), n(*), b
  local variables:
  real*8 u(3), v(3)
  u(1) = Vertex(1,1) − Vertex(1,2)
  u(2) = Vertex(2,1) − Vertex(2,2)
  u(3) = Vertex(3,1) − Vertex(3,2)
  v(1) = Vertex(1,1) − Vertex(1,3)
  v(2) = Vertex(2,1) − Vertex(2,3)
  v(3) = Vertex(3,1) − Vertex(3,3)
  n(1) = u(2) * v(3) − u(3) * v(2)
  n(2) = u(3) * v(1) − u(1) * v(3)
  n(3) = u(1) * v(2) − u(2) * v(1)
  b = sqrt(n(1) * n(1) + n(2) * n(2) + n(3) * n(3))
  return
end

subroutine dis_Ve_Pl(b, n, TesVer, Vertex, dis)
  implicit none
  real*8 TesVer(*), n(*), b, Vertex(3,*), dis
  dis = dabs((TesVer(1)−Vertex(1,3)) * n(1) +
             (TesVer(2)−Vertex(2,3)) * n(2) +
             (TesVer(3)−Vertex(3,3)) * n(3))
  dis = dis/b
  return
end

Listing F.1: definesurfcbbc.F

F Sourcecode of definesurfcbbc
**G Sourcecode of definepolcbc**

Source code of the functions for finding coupling nodes on polar surfaces.

```fortran
  subroutine definePolarCouplingBC(ix, x, x1, y1, z1, x2, y2, z2, r,
          or, cciNodes, cciElems, cciMap, nNodes, nElems)

  ** F E A P ** A Finite Element Analysis Program
  .... Copyright (c) 1984–2004: Regents of the University of California
  All rights reserved

  Purpose: Define an coupling BC for polar surfaces (tubes, ...).

  Inputs:
  ix     -- the element connection array in feap
  x      -- the coordinates array in feap
  x1, y1, z1
  x2, y2, z2 -- The 2 nodes defining the axis of the tube
  r      -- The radius
  or     -- The orientation of the axis (1, 2, or 3)

  Outputs:
  cciNodes -- Array with all nodes which lay on the boundary
  cciElems -- Array with all elements which lay on the boundary
  cciMap   -- Array containing the connection FEAP index-> cci index
  (the opposite of cciNodes)
  nNodes   -- number of interface nodes on the defined boundary
  nElems   -- number of interface elements on the interface boundary

  implicit none
  include 'cdata.h'
  include 'sdata.h'
  include 'iofile.h'

  real*8 x(ndm, *)
  integer ix(nen1, *), cciNodes(*), cciElems(*), cciMap(*)
  integer nNodes, nElems
  real*8 x1, y1, z1, x2, y2, z2, r, or

  local variables
  real*8 g(3), dis, P(3), Ax(3,2), dif, temp, disnodes(8)
  integer nNodesPerElem
  parameter (dif = 1.d-8, nNodesPerElem = 4)
  integer i, j, k, l, count, scount, orientation, axis, ornodes(4)
  c orientation: 1-W; 2-S; 3-B; 4-T; 5-N; 6-E
  integer couplelem(nen), surfaceelem(nen)
  integer findOrientation_8Brick, ppp
  logical flag, contains, onax
  save

  Organizing the input data
  Ax(1,1) = x1
  Ax(2,1) = y1
  Ax(3,1) = z1
```

G Sourcecode of definepolcbc
\( Ax(1,2) = x_2 \)
\( Ax(2,2) = y_2 \)
\( Ax(3,2) = z_2 \)

```plaintext
c  computing the direction vector (g)
call LinEq(Ax,g)

do i = 1, numel
  flag = .false.
onax = .true.
count = 0
count = 0
ornodes(1) = 0
ornodes(2) = 0
ornodes(3) = 0
ornodes(4) = 0

do ppp = 1, 8
  disnodes(ppp) = 0
  surfaceelem(ppp) = 0
enddo

do j = 1, nen
  if (ix(j, i) .eq. 0) then ! surface element
    flag = .false.
  else
    count = count + 1
couplelem(count) = ix(j, i)
P(1) = x(1,ix(j,i))
P(2) = x(2,ix(j,i))
P(3) = x(3,ix(j,i))
  c  Check if the current node is outside the Axis
  if (P(or) .gt. Ax(or,2)) then
    onax = .false.
  elseif (P(or) .lt. Ax(or,1)) then
    onax = .false.
  endif
  if (.not. onax) then
    flag = .false.
goto 767
  endif
  c  Computing the distance of the Point to the Axis (dis)
call dis_Ve_Ax(g,Ax,P,dis)
temp = dabs(dis - r)
disnodes(j) = temp
  if(temp .lt. dif) then
    flag = .true.
scount = scount + 1
    surfaceelem(scount) = ix(j, i)
    write(*,*) P(1), P(2), P(3), scount
  endif
```
endif
enddo

continue

if (flag) then ! coupling boundary element
  orientation = findOrientation_8Brick(couplelem, 
  $ surfaceelem)
  call boundary8BrickElem(cciMap, cciElems,
  $ nNodes, nElems, orientation, couplelem)
  nElems = nElems + 1
endif
enddo

now we have the filled cciMap. Construct the cciNodes array

do i = 1, numnp
  if (cciMap(i) .ne. 0) then ! coupling node
    cciNodes(cciMap(i)) = i
  endif
enddo

return
dend

subroutine LinEq(Ax, g)
  Subroutine to compute the direction vector

  implicit none
  real*8 Ax(3,*), g(*)
  real*8 u(3), v(3)

  g(1) = Ax(1,1)−Ax(1,2)
  g(2) = Ax(2,1)−Ax(2,2)
  g(3) = Ax(3,1)−Ax(3,2)

  return
dend

subroutine dis_Ve_Ax(g, Ax, P, dis)
  Subroutine to compute the distance of Point P the the Axis (Ax, g)

  implicit none
  real*8 g(*), Ax(3,*), P(*), dis, d(3)

  dis=(g(1)*(P(1)—Ax(1,1))+g(2)*(P(2)—Ax(2,1))+g(3)*(P(3)—Ax(3,1)))/ 
  $ (g(1)*g(1)+g(2)*g(2)+g(3)*g(3));

  d(1)=Ax(1,1)+dis*g(1)—P(1);
  d(2)=Ax(2,1)+dis*g(2)—P(2);
  d(3)=Ax(3,1)+dis*g(3)—P(3);

  dis=sqrt(d(1)*d(1)+d(2)*d(2)+d(3)*d(3));

  return
dend

Listing G.1: definepolcbc.F
H Sourcecode of definediffcbc

Source code of the functions for finding coupling nodes on coned surfaces.

```
subroutine defineDiffusorCouplingBC(ix, x, x1, y1, z1, x2, y2, z2, 
  $ dirr, r1, r2, cciNodes, cciElems, cciMap, nNodes, nElems)

* * F E A P * * A Finite Element Analysis Program

* . . . . Copyright (c) 1984−2004: Regents of the University of California
  All rights reserved

Purpose: Define an coupling BC for diffusors.

Inputs:
  ix − the element connection array in feap
  x − the coordinates array in feap
  x1, x2 − The 2 nodes defining the axis
  r1 − The smaller radius
  r2 − The greater radius
  dir − 1, 2 or 3 −> Direction of the diffusor

Outputs:
  cciNodes − Array with all nodes which lay on the boundary
  cciElems − Array with all elements which lay on the boundary
  cciMap − Array containing the connection FEAP index−> cci index
  nNodes − number of interface nodes on the defined boundary
  nElems − number of interface elements on the interface boundary

implicit none
include 'cdata.h'
include 'sdata.h'
include 'iofile.h'

real=8 x(ndm,*), cciNodes(*), cciElems(*), cciMap(*)
integer ix(nen1,*), nNodes, nElems
real=8 x1, y1, z1, x2, y2, z2, r1, r2

local variables
real=8 g(3), P(3), Ax(3,2), temp, DP(3,2), set, gv(3)
real=8 dis1, dis2, dif, L, dirr, disnodes(8)
integer nNodesPerElem, nn
parameter (dif = 1.d−8, nnodesPerElem = 4)
integer i, j, k, m, count, scount, orientation, axis, ornodes(4)
c orientation: 1−W; 2−S; 3−B; 4−T; 5−N; 6−E
c couplelem(nen), surfacelem(nen)
c findOrientation_8Brick, dir, ppp
logical flag, contains, onax
save

Organizing the input data
if (dirr .eq. 1) then
  dir = 1
```
else if (dirr .eq. 2) then
  dir = 2
else if (dirr .eq. 3) then
  dir = 3
endif
write(*,*) dir, dirr
do nn = 1, 3
  g(nn) = 0
  gv(nn) = 0
endo
Ax(1,1) = x1
Ax(2,1) = y1
Ax(3,1) = z1
Ax(1,2) = x2
Ax(2,2) = y2
Ax(3,2) = z2
DP(1,1) = Ax(1,1)
DP(2,1) = Ax(2,1)
DP(3,1) = Ax(3,1)
DP(1,2) = Ax(1,2)
DP(2,2) = Ax(2,2)
DP(3,2) = Ax(3,2)
DP(2,1) = DP(2,1)+r1
DP(2,2) = DP(2,2)+r2
L = dabs(Ax(dir,2) - Ax(dir,1))
c Computing the direction vectors of the Axis and the diffusor
call LinEq2(Ax,g,dir)
call LinEq2(DP,gv,dir)
do i = 1, numel
  flag = .false.
  count = 0
  scount = 0
  ornodes(1) = 0
  ornodes(2) = 0
  ornodes(3) = 0
  ornodes(4) = 0
  onax = .true.
do ppp = 1, 8
  disnodes(ppp) = 0
  surfaceelem(ppp) = 0
endo
do j = 1, nen
  if(ix(j, i) .eq. 0) then ! surface element
    flag = .false.
  else
    count = count + 1
    couplelem(count) = ix(j, i)
P(1) = x(1,ix(j,i))
P(2) = x(2,ix(j,i))
P(3) = x(3,ix(j,i))

if (P(dir) .gt. Ax(dir,2)) then
  onax = .false.
elseif (P(dir) .lt. Ax(dir,1)) then
  onax = .false.
endif

if (.not. onax) then
  flag = .false.
goto 767
endif

c    computing the distance of the current Vertex to the axis and
    the distance a point must have to be on the diffusor at this
    point
    call dis_Ve_Ax2(g,Ax,P,dis1)
    call dif_rad(P,set,Ax,gv,dir,r1,L)
    temp = dabs(dis1 - set)
    disnodes(j) = temp
    if(temp .lt. dif) then
      flag = .true.
      scount = scount + 1
      surfaceelem(scount) = ix(j,i)
    endif
endo
767 continue
if(flag) then                      ! coupling boundary element
  orientation = findOrientation_8Brick(couplelem,
    surfaceelem)
  call boundary8BrickElem(cciMap, cciElems,
    nNodes, nElems, orientation, couplelem)
  nElems = nElems + 1
endif
endo

c    now we have the filled cciMap. Construct the cciNodes array
    do i = 1, numnp
      if(cciMap(i) .ne. 0) then ! coupling node
        cciNodes(cciMap(i)) = i
      endif
    endo
  return
  end
  subroutine LinEq2(Te,t,dir)
  c    computing the direction vector
Listing H.1: definediffcbc.F

```fortran
implicit none
real*8 Te(3,*) , t(*)
integer dir

  t(dir) = Te(dir,2) - Te(dir,1)

return
eend

subroutine dis_Ve_Ax2(g,Ax,P,dis)
  
  computing the distance of point P to Axis (g, Ax)
  
  implicit none
  real*8 g(*), Ax(3,*), P(*), dis, d(3)

  dis=(g(1)*( P(1) - Ax(1,1)) + g(2)*(P(2) - Ax(2,1)) + g(3)*(P(3) - Ax(3,1)))/
  $(g(1)*g(1)+g(2)*g(2)+g(3)*g(3));

  d(1)=Ax(1,1) + dis * g(1) - P(1);
  d(2)=Ax(2,1) + dis * g(2) - P(2);
  d(3)=Ax(3,1) + dis * g(3) - P(3);

  dis=sqrt(d(1)*d(1)+d(2)*d(2)+d(3)*d(3));

return
eend

cccccccccccccccccccccccccc

subroutine dif_rad(P,set,Ax,gv,dir,r1,L)
  
  implicit none
  real*8 P(*), set, Ax(3,*), gv(*), r1, L, temp
  integer dir

  if (Ax(dir,1) .gt. Ax(dir,2)) then
    temp = Ax(dir,2)
  else
    temp = Ax(dir,1)
  endif

  set = P(dir) - temp
  set = set * gv(dir)/L+r1

return
eend
```

Listing H.1: definediffcbc.F
# Sourcecode of definetfisurface

Source code of the functions for finding coupling nodes on surfaces described by the use of TFI.

```fortran
subroutine definetfisurface(ix, x, dis1, dis2,
$ cciNodes, cciElems, cciMap, nNodes, nElems)

c * * F E A P * * A Finite Element Analysis Program
.... Copyright (c) 1984–2004: Regents of the University of California
All rights reserved

Purpose: Define an coupling BC for rectangular surfaces.
Inputs:
i x – the element connection array in feap
    x – the coordinates array in feap
    dis1, dis2 – the discretisation of the grid
    all other parameters are taken from the arrays

Outputs:
cciNodes – Array with all nodes which lay on the boundary
cciElems – Array with all elements which lay on the boundary
cciMap – Array containing the connection FEAP index-> cci index
(the opposite of cciNodes)
nNodes – number of interface nodes on the defined boundary
nElems – number of interface elements on the interface boundary

implicit none
include 'cdata.h'
include 'sdata.h'

include 'pointer.h'
include 'upointer.h'

include 'comblk.h'

real*8 x(ndm, *)
integer ix(nen1, *), cciNodes(*), cciElems(*), cciMap(*)
integer nNodes, nElems, dis1, dis2

local variables
real*8 P(3), temp, dif, distance
integer nNodesPerElem
parameter (dif = 1.d-3, nnodesPerElem = 4)
integer i, j, count, scount, orientation, axis
c orientation: 1–W; 2–S; 3–B; 4–T; 5–N; 6–E
couplelem(nen), surfaceelem(nen)
c findOriantation_8Brick, ppp, lll
logical flag
save

do i = 1, numel
    flag = .false.
    count = 0
```

I Sourcecode of definetfisurface
scount = 0

    do j = 1, nen
        if(ix(j, i) .eq. 0) then ! surface element
            flag = .false.
        else
            count = count + 1
            couplelem(count) = ix(j, i)

            P(1) = x(1, ix(j, i))
            P(2) = x(2, ix(j, i))
            P(3) = x(3, ix(j, i))

            call isonurf(P, distance, dis1, dis2)

            if(distance .lt. dif) then
                flag = .true.
                scount = scount + 1
                surfaceelem(scount) = ix(j, i)
            endif
        endif
    enddo

    if(flag) then ! coupling boundary element
        orientation = findOrientation_8Brick(couplelem, surfaceelem)
        call boundary8BrickElem(cciMap, cciElems, nNodes, nElems, orientation, couplelem)
        nElems = nElems + 1
    endif
enddo

    now we have the filled cciMap. Construct the cciNodes array
    do i = 1, numnp
        if(cciMap(i) .ne. 0) then ! coupling node
            cciNodes(cciMap(i)) = i
        endif
    enddo

    return end

subroutine isonurf(P, distance, dis1, dis2)
c computes the distance of P to the TFI-surface
implicit none
include 'pointer.h'
include 'upointer.h'
include 'comblk.h'
real*8 P(*), temp, temp1, temp2, temp3, distance
integer dis1, dis2, i, j

distance = 10d10
do i = 1, dis1
  do j = 1, dis2
    temp1 = P(1) - hr(up(104)+(+1)*dis1+j-1)
    temp2 = P(2) - hr(up(105)+(+1)*dis1+j-1)
    temp3 = P(3) - hr(up(106)+(+1)*dis1+j-1)
    temp = temp1*temp1 + temp2*temp2 + temp3*temp3
    temp = sqrt(temp)
    if (temp .lt. distance) distance = temp
  enddo
enddo
return
end
Source code of the functions transferring from logical coordinates to real coordinates by the use of TFI.

```
subroutine TFI(xi, eta, zeta, x, y, z)
  implicit none
  real*8 xi, eta, zeta, x, y, z
  real*8 ux, uy, uz, vx, vy, wz
  real*8 uwx, uwy, uwz, uvx, uvy, uvz, vwx, vwy, vwz
  real*8 f1, f2, f3, f4, f5, f6, f7, f8, f9, f10, f11, f12
  real*8 d0, d1
  
  parameter (d0 = 0.d0, d1 = 1.d0)
  
  c Goal: Computing all coordinates of the surface

  ux = (1-xi)*(1-zeta)*f5(d0, eta, d0, 1) + zeta*f8(d0, eta, d1, 1)
  & +xi*(1-zeta)*f6(d1, eta, d0, 1) + zeta*f7(d1, eta, d1, 1)
  & -eta*(zeta*f2(d1, d0, d1, 1) + (1-zeta)*f1(d0, d1, d0, 1))
  & -xi*(zeta*f7(d1, d1, d1, 1) + (1-zeta)*f6(d1, d1, d0, 1))

  uy = (1-xi)*(1-zeta)*f5(d0, eta, d0, 2) + zeta*f8(d0, eta, d1, 2)
  & +xi*(1-zeta)*f6(d1, eta, d0, 2) + zeta*f7(d1, eta, d1, 2)
  & -eta*(zeta*f2(d1, d0, d1, 2) + (1-zeta)*f1(d0, d1, d0, 2))
  & -xi*(zeta*f7(d1, d1, d1, 2) + (1-zeta)*f6(d1, d1, d0, 2))

  uz = (1-xi)*(1-zeta)*f5(d0, eta, d0, 3) + zeta*f8(d0, eta, d1, 3)
  & +xi*(1-zeta)*f6(d1, eta, d0, 3) + zeta*f7(d1, eta, d1, 3)
  & -eta*(zeta*f2(d1, d0, d1, 3) + (1-zeta)*f1(d0, d1, d0, 3))
  & -xi*(zeta*f7(d1, d1, d1, 3) + (1-zeta)*f6(d1, d1, d0, 3))

  vx = (1-eta)*(1-zeta)*f1(xi, d0, d0, 1) + zeta*f2(xi, d0, d1, 1)
  & +eta*(1-zeta)*f4(xi, d0, d1, 1) + zeta*f3(xi, d1, d1, 1)
  & -xi*(zeta*f7(d1, d1, d1, 1) + (1-zeta)*f6(d1, d1, d0, 1))
  & -eta*(zeta*f8(d0, d1, d1, 1) + (1-zeta)*f5(d0, d1, d0, 1))

  vy = (1-eta)*(1-zeta)*f1(xi, d0, d0, 2) + zeta*f2(xi, d0, d1, 2)
  & +eta*(1-zeta)*f4(xi, d0, d1, 2) + zeta*f3(xi, d1, d1, 2)
  & -xi*(zeta*f7(d1, d1, d1, 2) + (1-zeta)*f6(d1, d1, d0, 2))
  & -eta*(zeta*f8(d0, d1, d1, 2) + (1-zeta)*f5(d0, d1, d0, 2))
```

\[-x_i (zeta \times f2 (d1, d0, d1, 2) + (1 - zeta) \times f1 (d1, d0, d0, 2)) \]
\[-(1 - x_i) (zeta \times f9 (d0, d0, d1, 2) + (1 - zeta) \times f1 (d0, d0, d0, 2)) \]
\[+ eta \times ((1 - zeta) \times f4 (xi, d1, d0, 2) + zeta \times f3 (xi, d1, d1, 2) + eta \times f12 (d0, d1, zeta, 2) + xi \times f11 (d1, d1, zeta, 2) \]
\[-x_i (zeta \times f7 (d1, d1, d1, 2) + (1 - zeta) \times f6 (d1, d1, d0, 2)) \]
\[-(1 - x_i) (zeta \times f8 (d0, d1, d1, 2) + (1 - zeta) \times f5 (d0, d1, d0, 2)) \]

\[v_z = (1 - eta) \times ((1 - zeta) \times f1 (d1, d0, d0, 3) + zeta \times f2 (xi, d0, d0, 3) + xi \times f10 (d1, d0, zeta, 3) \]
\[-x_i (zeta \times f2 (d1, d0, d0, 3) + (1 - zeta) \times f1 (d1, d0, d0, 3)) \]
\[-(1 - x_i) (zeta \times f9 (d0, d0, d0, 3) + (1 - zeta) \times f1 (d0, d0, d0, 3)) + eta \times ((1 - zeta) \times f4 (xi, d1, d0, 3) + zeta \times f3 (xi, d1, d1, 3) + eta \times f12 (d0, d1, zeta, 3) + xi \times f11 (d1, d1, zeta, 3) \]
\[-(1 - x_i) (zeta \times f7 (d1, d1, d1, 3) + (1 - zeta) \times f6 (d1, d1, d0, 3)) + (1 - eta) \times f8 (d0, d0, d1, d1, 3) + (1 - zeta) \times f5 (d0, d1, d0, d0, 3) \]

\[w_x = (1 - zeta) \times ((1 - eta) \times f1 (xi, d0, d0, 1) + eta \times f4 (xi, d1, d0, 1) + (1 - x_i) \times f5 (d0, eta, d0, 1) + xi \times f6 (d1, eta, d0, 1) \]
\[-x_i (eta \times f6 (d1, d1, d0, 1) + (1 - eta) \times f1 (d1, d0, d0, 1)) \]
\[-(1 - x_i) (eta \times f5 (d0, d1, d0, 1) + (1 - eta) \times f1 (d0, d0, d0, 1)) + zeta \times ((1 - eta) \times f2 (xi, d0, d1, 1) + eta \times f3 (xi, d1, d1, 1) + (1 - x_i) \times f8 (d0, eta, d1, 1) + xi \times f7 (d1, eta, d1, 1) \]
\[-x_i (eta \times f7 (d1, d1, d1, 1) + (1 - eta) \times f2 (d1, d0, d1, 1)) \]
\[-(1 - x_i) (eta \times f12 (d0, d1, d1, 1) + (1 - eta) \times f9 (d0, d0, d1, 1)) \]

\[w_y = (1 - zeta) \times ((1 - eta) \times f1 (xi, d0, d0, 2) + eta \times f4 (xi, d1, d0, 2) + (1 - x_i) \times f5 (d0, eta, d0, 2) + xi \times f6 (d1, eta, d0, 2) \]
\[-x_i (eta \times f6 (d1, d1, d0, 2) + (1 - eta) \times f1 (d1, d0, d0, 2)) \]
\[-(1 - x_i) (eta \times f5 (d0, d1, d0, 2) + (1 - eta) \times f1 (d0, d0, d0, 2)) + zeta \times ((1 - eta) \times f2 (xi, d0, d1, 2) + eta \times f3 (xi, d1, d1, 2) + (1 - x_i) \times f8 (d0, eta, d1, 2) + xi \times f7 (d1, eta, d1, 2) \]
\[-x_i (eta \times f7 (d1, d1, d1, 2) + (1 - eta) \times f2 (d1, d0, d1, 2)) \]
\[-(1 - x_i) (eta \times f12 (d0, d1, d1, 2) + (1 - eta) \times f9 (d0, d0, d1, 2)) \]

\[w_z = (1 - zeta) \times ((1 - eta) \times f1 (xi, d0, d0, 3) + eta \times f4 (xi, d1, d0, 3) + (1 - x_i) \times f5 (d0, eta, d0, 3) + xi \times f6 (d1, eta, d0, 3) \]
\[-x_i (eta \times f6 (d1, d1, d0, 3) + (1 - eta) \times f1 (d1, d0, d0, 3)) \]
\[-(1 - x_i) (eta \times f5 (d0, d1, d0, 3) + (1 - eta) \times f1 (d0, d0, d0, 3)) + zeta \times ((1 - eta) \times f2 (xi, d0, d1, 3) + eta \times f3 (xi, d1, d1, 3) + (1 - x_i) \times f8 (d0, eta, d1, 3) + xi \times f7 (d1, eta, d1, 3) \]
\[-x_i (eta \times f7 (d1, d1, d1, 3) + (1 - eta) \times f2 (d1, d0, d1, 3)) \]
\[-(1 - x_i) (eta \times f12 (d0, d1, d1, 3) + (1 - eta) \times f9 (d0, d0, d1, 3)) \]

\[u_x = (1 - x_i) \times (zeta \times f5 (d0, eta, d0, 1) + (1 - x_i) \times zeta \times f8 (d0, eta, d0, 1) + \begin{array}{c} \times \end{array} \]
\[u_y = (1 - x_i) \times (zeta \times f5 (d0, eta, d0, 2) + (1 - x_i) \times zeta \times f8 (d0, eta, d1, 2) + \begin{array}{c} \times \end{array} \]
\[u_z = (1 - x_i) \times (zeta \times f5 (d0, eta, d0, 3) + (1 - x_i) \times zeta \times f8 (d0, eta, d1, 3) + \begin{array}{c} \times \end{array} \]
\[u_x = (1 - x_i) \times (1 - eta) \times f9 (d0, d0, zeta, 1) + (1 - x_i) \times eta \times f12 (d0, d1, zeta, 1) \]
\[\times \begin{array}{c} \times \end{array} \]
\[u_y = (1 - x_i) \times (1 - eta) \times f9 (d0, d0, zeta, 1) + (1 - x_i) \times eta \times f12 (d0, d1, zeta, 1) \]
\[\times \begin{array}{c} \times \end{array} \]
\[u_z = (1 - x_i) \times (1 - eta) \times f9 (d0, d0, zeta, 1) + (1 - x_i) \times eta \times f12 (d0, d1, zeta, 1) \]
uvy = (1 - xi) * (1 - eta - x + eta) * f9 (d0, d0, zeta, 2) + (1 - xi) * eta + f12 (d0, d1, zeta, 2) + xi * (1 - eta - x + eta) * f10 (d1, d0, zeta, 2) +
& xi * eta * f11 (d1, d1, zeta, 2)

uvz = (1 - xi) * (1 - eta - x + eta) * f9 (d0, d0, zeta, 3) + (1 - xi) * eta + f12 (d0, d1, zeta, 3) + xi * (1 - eta - x + eta) * f10 (d1, d0, zeta, 3) +
& xi * eta * f11 (d1, d1, zeta, 3)

vwx = (1 - eta - x + eta) * (1 - zeta - x + zeta) * f1 (xi, d0, d0, 1) + (1 - zeta - x + zeta) * eta * f4 (xi, d1, d0, 1) +
& zeta * (1 - eta - x + eta) * f2 (xi, d0, d1, 1) +
& eta * zeta * f3 (xi, d1, d1, 1)

vwy = (1 - eta - x + eta) * (1 - zeta - x + zeta) * f1 (xi, d0, d0, 2) + (1 - zeta - x + zeta) * eta * f4 (xi, d1, d0, 2) +
& zeta * (1 - eta - x + eta) * f2 (xi, d0, d1, 2) +
& eta * zeta * f3 (xi, d1, d1, 2)

vwz = (1 - eta - x + eta) * (1 - zeta - x + zeta) * f1 (xi, d0, d0, 3) + (1 - zeta - x + zeta) * eta * f4 (xi, d1, d0, 3) +
& zeta * (1 - eta - x + eta) * f2 (xi, d0, d1, 3) +
& eta * zeta * f3 (xi, d1, d1, 3)

uvwx = (1 - xi) * (1 - eta - x + eta) * (1 - zeta - x + zeta) * f1 (d0, d0, d0, 1) +
& (1 - xi) * (1 - eta - x + eta) * zeta * f9 (d0, d0, d0, 1) +
& (1 - xi) * eta * (1 - zeta - x + zeta) * f5 (d0, d0, d1, 0, 1) +
& (1 - xi) * eta * zeta * f8 (d0, d0, d1, 1) +
& xi * (1 - eta - x + eta) * f1 (d1, d0, d0, 1) +
& xi * (1 - eta - x + eta) * zeta * f2 (d1, d0, d0, 1) +
& xi * eta * (1 - eta - x + eta) * f6 (d1, d1, d0, 1) +
& xi * eta * zeta * f7 (d1, d1, d1, 1)

uvwy = (1 - xi) * (1 - eta - x + eta) * (1 - zeta - x + zeta) * f1 (d0, d0, d0, 2) +
& (1 - xi) * (1 - eta - x + eta) * zeta * f9 (d0, d0, d0, 2) +
& (1 - xi) * eta * (1 - zeta - x + zeta) * f5 (d0, d0, d1, 0, 2) +
& (1 - xi) * eta * zeta * f8 (d0, d0, d1, 1, 2) +
& xi * (1 - eta - x + eta) * f1 (d1, d0, d0, 2) +
& xi * (1 - eta - x + eta) * zeta * f2 (d1, d0, d0, 1, 2) +
& xi * eta * (1 - eta - x + eta) * f6 (d1, d1, d0, 2) +
& xi * eta * zeta * f7 (d1, d1, d1, 1, 2)

uvwz = (1 - xi) * (1 - eta - x + eta) * (1 - zeta - x + zeta) * f1 (d0, d0, d0, 3) +
& (1 - xi) * (1 - eta - x + eta) * zeta * f9 (d0, d0, d0, 3) +
& (1 - xi) * eta * (1 - zeta - x + zeta) * f5 (d0, d0, d1, 0, 3) +
& (1 - xi) * eta * zeta * f8 (d0, d0, d1, 1, 3) +
& xi * (1 - eta - x + eta) * f1 (d1, d0, d0, 3) +
& xi * (1 - eta - x + eta) * zeta * f2 (d1, d0, d0, 1, 3) +
& xi * eta * (1 - eta - x + eta) * f6 (d1, d1, d0, 3) +
& xi * eta * zeta * f7 (d1, d1, d1, 1, 3)

x = ux + vx + wx - uwx - uvx + uvwx
y = uy + vy + wy - uyv - uvy + uvwy
z = uz + vz + wz - uzv - uvz + uvwz

return

end

Listing J.1: TFI.F

J Sourcecode of TFI
K Sourcecode of functions

Source code of the functions used for the TFI.

```c
/* ------------------------------------------------------------- */
/* Functions: f1 - f12 */
/* Purpose: Defining the function for an edge of the tfisurface. */
/* Inputs: */
/* xi, eta, zeta - logical coordinates */
/* dir - Direction of the edge (1, 2 or 3) */
/* Outputs: */
/* stores the parameters directly in the arrays */
/* ------------------------------------------------------------- */

function f1(xi, eta, zeta, dir)
  include 'pointer.h'
  include 'upointer.h'

  include 'comblk.h'

  real*8 f1
  real*8 xi, eta, zeta, llog, lre, xtemp, yspline
  integer n, dir, j
  real*8 P1(3), P2(3), g(3)
  real*8 x, y, z, temp
  logical flag1, flag2

  if (hr(up(103)).eq.2) then ! linear
    P1(1) = hr(up(7))
    P2(1) = hr(up(7)+1)
    P1(2) = hr(up(8))
    P2(2) = hr(up(8)+1)
    P1(3) = hr(up(9))
    P2(3) = hr(up(9)+1)
  endif

  g(1) = P2(1) - P1(1)
  g(2) = P2(2) - P1(2)
  g(3) = P2(3) - P1(3)

  x = P1(1) + xi*g(1)
  y = P1(2) + eta*g(2)
  z = P1(3) + zeta*g(3)

  if (dir.eq.1) f1 = x
  if (dir.eq.2) f1 = y
  if (dir.eq.3) f1 = z
elseif (hr(up(103)).gt.2) then ! Spline
  j = hr(up(103))
  P1(1) = hr(up(7))
  P2(1) = hr(up(7)+j)
  P1(2) = hr(up(8))
  P2(2) = hr(up(8)+j)
```

K Sourcecode of functions
P1(3) = hr(up(9))
P2(3) = hr(up(9)+j)
g(1) = P2(1)−P1(1)
g(2) = P2(2)−P1(2)
g(3) = P2(3)−P1(3)
int = 0
xtemp = P1(1) + xi * g(1)
do i = 1, j ! in which splinecurve?
    if (xtemp .ge. hr(up(7)+i−1).and.xtemp .le. hr(up(7)+i)) then
        int = i−1
    endif
enddo
yspline = hr(up(43)+int) *(xtemp − hr(up(7)+int)) *
$ (xtemp − hr(up(7)+int)) *(xtemp − hr(up(7)+int)) *
$ + hr(up(44)+int) *(xtemp − hr(up(7)+int)) *
$ (xtemp − hr(up(7)+int)) *
$ + hr(up(45)+int) *(xtemp − hr(up(7)+int)) *
$ + hr(up(46)+int)
if (dir .eq. 1) f1 = xtemp
if (dir .eq. 3) f1 = P1(3) + zeta * g(3)
if (dir .eq. 2) f1 = yspline
endif
return
end

function f2(xi, eta, zeta, dir)
include 'pointer.h'
include 'upointer.h'
include 'comblk.h'
real*8 f2
real*8 xi, eta, zeta, llog, lre, xtemp, yspline
integer n, dir, j
real*8 P1(3), P2(3), g(3)
real*8 x, y, z, temp
logical flag1, flag2
if (hr(up(103)+1) .eq. 2) then ! linear
    P1(1) = hr(up(10))
P2(1) = hr(up(10)+1)
P1(2) = hr(up(11))
P2(2) = hr(up(11)+1)
P1(3) = hr(up(12))
P2(3) = hr(up(12)+1)
g(1) = P2(1)−P1(1)
g(2) = P2(2)−P1(2)
g(3) = P2(3) - P1(3)

x = P1(1) + xi * g(1)
y = P1(2) + eta * g(2)
z = P1(3) + zeta * g(3)

if (dir .eq. 1) f2 = x
if (dir .eq. 2) f2 = y
if (dir .eq. 3) f2 = z

elseif (hr(up(103)+1) .gt. 2) then ! Spline
  j = hr(up(103)+1)
P1(1) = hr(up(10))
P2(1) = hr(up(10)+j)
P1(2) = hr(up(11))
P2(2) = hr(up(11)+j)
P1(3) = hr(up(12))
P2(3) = hr(up(12)+j)

  g(1) = P2(1) - P1(1)
g(2) = P2(2) - P1(2)
g(3) = P2(3) - P1(3)

  int = 0

  xtemp = P1(1) + xi * g(1)

do i = 1, j ! in which splinecurve?
  if (xtemp .ge. hr(up(10)+i-1).and.xtemp .le. hr(up(10)+i)) then
    int = i-1
  endif
endo
dospline = hr(up(47)+int) *(xtemp - hr(up(10)+int)) *
  $ (xtemp - hr(up(10)+int)) *(xtemp - hr(up(10)+int)) *
  $ + hr(up(48)+int) * (xtemp - hr(up(10)+int)) *
  $ (xtemp - hr(up(10)+int)) *
  $ + hr(up(49)+int) *(xtemp - hr(up(10)+int)) *
  $ + hr(up(50)+int)

  if (dir .eq. 1) f2 = xtemp
  if (dir .eq. 3) f2 = P1(3) + zeta * g(3)
  if (dir .eq. 2) f2 = yspline
endif

return
end

function f3(xi, eta, zeta, dir)
include 'pointer.h'
include 'upointer.h'
include 'comblk.h'
real*8 f3
real*8 xi, eta, zeta, llog, lre, xtemp, yspline
integer n, dir, j
real*8 P1(3), P2(3), g(3)
real*8 x, y, z, temp
logical flag1, flag2

if (hr(up(103)+2) .eq. 2) then ! linear
  P1(1) = hr(up(13))
  P2(1) = hr(up(13)+1)
  P1(2) = hr(up(14))
  P2(2) = hr(up(14)+1)
  P1(3) = hr(up(15))
  P2(3) = hr(up(15)+1)
  g(1) = P2(1)−P1(1)
  g(2) = P2(2)−P1(2)
  g(3) = P2(3)−P1(3)
  x = P1(1) + xi*g(1)
  y = P1(2) + eta*g(2)
  z = P1(3) + zeta*g(3)
  if (dir .eq. 1) f3 = x
  if (dir .eq. 2) f3 = y
  if (dir .eq. 3) f3 = z
endif

elseif (hr(up(103)+2) .gt. 1) then ! Spline
  j = hr(up(103)+2)
  P1(1) = hr(up(13))
  P2(1) = hr(up(13)+j)
  P1(2) = hr(up(14))
  P2(2) = hr(up(14)+j)
  P1(3) = hr(up(15))
  P2(3) = hr(up(15)+j)
  g(1) = P2(1)−P1(1)
  g(2) = P2(2)−P1(2)
  g(3) = P2(3)−P1(3)
  int = 0
  xtemp = P1(1) + xi*g(1)
  do i = 1, j ! in which splinecurve?
    if (xtemp .ge. hr(up(13)+i−1).and. xtemp .le. hr(up(13)+i)) then
      int = i−1
    endif
  enddo
  yspline = hr(up(51)+int)*(xtemp − hr(up(13)+int))∗
             $ (xtemp − hr(up(13)+int))*(xtemp − hr(up(13)+int))$
             $ + hr(up(52)+int) *(xtemp − hr(up(13)+int))$
             $ + hr(up(53)+int) *(xtemp − hr(up(13)+int))$
             $ + hr(up(54)+int)$
if (dir .eq. 1) f3 = xtemp
if (dir .eq. 3) f3 = P1(3) + zeta*g(3)
if (dir .eq. 2) f3 = yspline
endif
return
dend

function f4(xi, eta, zeta, dir)
include 'pointer.h'
include 'upointer.h'
include 'comblk.h'
real*8 f4
real*8 xi, eta, zeta, lllog, lre, xtemp, yspline
integer n, dir, j
real*8 P1(3), P2(3), g(3)
real*8 x, y, z, temp
logical flag1, flag2

if (hr(up(103)+3) .eq. 2) then ! linear
P1(1) = hr(up(16))
P2(1) = hr(up(16)+1)
P1(2) = hr(up(17))
P2(2) = hr(up(17)+1)
P1(3) = hr(up(18))
P2(3) = hr(up(18)+1)
g(1) = P2(1)−P1(1)
g(2) = P2(2)−P1(2)
g(3) = P2(3)−P1(3)
x = P1(1) + xi*g(1)
y = P1(2) + eta*g(2)
z = P1(3) + zeta*g(3)
if (dir .eq. 1) f4 = x
if (dir .eq. 2) f4 = y
if (dir .eq. 3) f4 = z
elseif (hr(up(103)+3) .gt. 2) then ! Spline
j = hr(up(103)+3)
P1(1) = hr(up(16))
P2(1) = hr(up(16)+j)
P1(2) = hr(up(17))
P2(2) = hr(up(17)+j)
P1(3) = hr(up(18))
P2(3) = hr(up(18)+j)
g(1) = P2(1)−P1(1)
g(2) = P2(2)−P1(2)
g(3) = P2(3)−P1(3)
int = 0
xtemp = P1(1) + xi * g(1)

do i = 1, j ! in which splinecurve?
   if (xtemp.ge.hr(up(16)+i−1).and.xtemp .le. hr(up(16)+i))then
      int = i−1
   endif
endo

yspline = hr(up(55)+int)*(xtemp − hr(up(16)+int)) * $ (xtemp − hr(up(16)+int)) * $ + hr(up(56)+int) * (xtemp − hr(up(16)+int)) * $ + hr(up(57)+int) * (xtemp − hr(up(16)+int)) * $ + hr(up(58)+int)

   if (dir .eq. 1) f4 = xtemp
   if (dir .eq. 3) f4 = P1(3) + zeta * g(3)
   if (dir .eq. 2) f4 = yspline
endif

return
end

function f5(xi, eta, zeta, dir)

include ’pointer.h’
include ’upointer.h’
include ’comblk.h’

real*8 f5
real*8 xi, eta, zeta, llog, lre, xtemp, yspline
integer n, dir, j
real*8 P1(3), P2(3), g(3)
real*8 x, y, z, temp
logical flag1, flag2

if (hr(up(103)+4) .eq. 2) then ! linear
   P1(1) = hr(up(19))
   P2(1) = hr(up(19)+1)
   P1(2) = hr(up(20))
   P2(2) = hr(up(20)+1)
   P1(3) = hr(up(21))
   P2(3) = hr(up(21)+1)

   g(1) = P2(1)−P1(1)
   g(2) = P2(2)−P1(2)
   g(3) = P2(3)−P1(3)

   x = P1(1) + xi * g(1)
   y = P1(2) + eta * g(2)
   z = P1(3) + zeta * g(3)
endif

   if (dir .eq. 1) f5 = x
   if (dir .eq. 2) f5 = y
if (dir .eq. 3) f5 = z

else if (hr(up(103)+4) .gt. 2) then ! Spline
  j = hr(up(103)+4)
  P1(1) = hr(up(19))
  P2(1) = hr(up(19)+j)
  P1(2) = hr(up(20))
  P2(2) = hr(up(20)+j)
  P1(3) = hr(up(21))
  P2(3) = hr(up(21)+j)

  g(1) = P1(1)−P2(1)
  g(2) = P1(2)−P2(2)
  g(3) = P1(3)−P2(3)

  do i = 1, j
    if (hr(up(19)) .eq. hr(up(19)+i)) flag1 = .true.
    if (hr(up(21)) .eq. hr(up(21)+i)) flag2 = .true.
  enddo

  xtemp = P1(2) + eta * g(2)

  do i = 1, j ! in which splinecurve?
    if (xtemp .ge. hr(up(20)+i−1).and. xtemp .le. hr(up(20)+i)) then
      int = i−1
    endif
  enddo

  yspline = hr(up(59)+int)*(xtemp − hr(up(20)+int))*
           $ (xtemp − hr(up(20)+int))*(xtemp − hr(up(20)+int))*
           $ + hr(up(60)+int) * (xtemp − hr(up(20)+int))*
           $ (xtemp − hr(up(20)+int))
           $ + hr(up(61)+int) * (xtemp − hr(up(20)+int))
           $ + hr(up(62)+int)

  if (dir .eq. 2) f5 = xtemp
  if (dir .eq. 1) f5 = P1(1) + xi*g(1)
  if (dir .eq. 3) f5 = yspline
endif

return
end

function f6(xi, eta, zeta, dir)

  include 'pointer.h'
  include 'upointer.h'
  include 'comblk.h'

  real*8 f6
  real*8 xi, eta, zeta, llog, lre, xtemp, yspline
  integer n, dir, j
  real*8 P1(3), P2(3), g(3)
  real*8 x, y, z, temp
  logical flag1, flag2
if (hr(up(103)+5) . eq. 2) then ! linear
   P1(1) = hr(up(22))
   P2(1) = hr(up(22)+1)
   P1(2) = hr(up(23))
   P2(2) = hr(up(23)+1)
   P1(3) = hr(up(24))
   P2(3) = hr(up(24)+1)
   g(1) = P2(1)−P1(1)
   g(2) = P2(2)−P1(2)
   g(3) = P2(3)−P1(3)
   x = P1(1) + xi*g(1)
   y = P1(2) + eta*g(2)
   z = P1(3) + zeta*g(3)
   if (dir . eq. 1) f6 = x
   if (dir . eq. 2) f6 = y
   if (dir . eq. 3) f6 = z
elseif (hr(up(103)+5) . gt. 2) then ! Spline
   j = hr(up(103)+5)
   P1(1) = hr(up(22))
   P2(1) = hr(up(22)+j)
   P1(2) = hr(up(23))
   P2(2) = hr(up(23)+j)
   P1(3) = hr(up(24))
   P2(3) = hr(up(24)+j)
   g(1) = P2(1)−P1(1)
   g(2) = P2(2)−P1(2)
   g(3) = P2(3)−P1(3)
   int = i−1
   do i = 1, j ! in which splinecurve?
      if (xtemp . ge. hr(up(23)+i−1).and.xtemp . le. hr(up(23)+i)) then
         y spline = hr(up(63)+int)∗(xtemp − hr(up(23)+int))∗
                     $ (xtemp − hr(up(23)+int))∗(xtemp − hr(up(23)+int))∗$
         $ + hr(up(64)+int) ∗ (xtemp − hr(up(23)+int))∗$
         $ + hr(up(65)+int) ∗ (xtemp − hr(up(23)+int))$
         $ + hr(up(66)+int)$
      endif
   enddo
   xtemp = P1(2) + eta ∗ g(2)
   do i = 1, j ! in which splinecurve?
      if (xtemp . ge. hr(up(23)+i−1).and.xtemp . le. hr(up(23)+i)) then
         int = i−1
      endif
   enddo
   y spline = hr(up(63)+int)∗(xtemp − hr(up(23)+int))∗
   $ (xtemp − hr(up(23)+int))∗(xtemp − hr(up(23)+int))∗$
   $ + hr(up(64)+int) ∗ (xtemp − hr(up(23)+int))∗$
   $ + hr(up(65)+int) ∗ (xtemp − hr(up(23)+int))$
   $ + hr(up(66)+int)$
   if (dir . eq. 2) f6 = xtemp
   if (dir . eq. 1) f6 = P1(1) + xi*g(1)
   if (dir . eq. 3) f6 = y spline
```c
endif

return
dend

function f7(xi, eta, zeta, dir)
include 'pointer.h'
include 'upointer.h'
include 'comblk.h'

real*8 x, eta, zeta, lllog, llre, xtemp, yspline
integer n, dir, j
real*8 P1(3), P2(3), g(3)
real*8 x, y, z, temp
logical flag1, flag2

if (hr(up(103)+6) .eq. 2) then ! linear
  P1(1) = hr(up(25))
  P2(1) = hr(up(25)+1)
  P1(2) = hr(up(26))
  P2(2) = hr(up(26)+1)
  P1(3) = hr(up(27))
  P2(3) = hr(up(27)+1)
  g(1) = P2(1)-P1(1)
  g(2) = P2(2)-P1(2)
  g(3) = P2(3)-P1(3)
  x = P1(1) + xi*g(1)
  y = P1(2) + eta*g(2)
  z = P1(3) + zeta*g(3)
  if (dir .eq. 1) f7 = x
  if (dir .eq. 2) f7 = y
  if (dir .eq. 3) f7 = z
  int = 0
  xtemp = P1(2) + eta * g(2)
elseif (hr(up(103)+6) .gt. 2) then ! Spline
  j = hr(up(103)+6)
  P1(1) = hr(up(25))
  P2(1) = hr(up(25)+j)
  P1(2) = hr(up(26))
  P2(2) = hr(up(26)+j)
  P1(3) = hr(up(27))
  P2(3) = hr(up(27)+j)
  g(1) = P2(1)-P1(1)
  g(2) = P2(2)-P1(2)
  g(3) = P2(3)-P1(3)
  int = 0
  xtemp = P1(2) + eta * g(2)
endif
```

K Sourcecode of functions

85
do i = 1, j ! in which splinecurve?
  if (xtemp.ge.hr(up(26)+i-1).and.xtemp .le. hr(up(26)+i)) then
    int = i-1
  endif
enddo

yspline = hr(up(67)+int)*(xtemp - hr(up(26)+int))**
$ (xtemp -hr(up(26)+int))*(xtemp -hr(up(26)+int))**
$ + hr(up(68)+int) * (xtemp -hr(up(26)+int))**
$ + hr(up(69)+int) * (xtemp -hr(up(26)+int))**
$ + hr(up(70)+int)

if (dir .eq. 2) f7 = xtemp
if (dir .eq. 1) f7 = P1(1) + xi*g(1)
if (dir .eq. 3) f7 = yspline
endif
return
end

function f8(xi, eta, zeta, dir)

include 'pointer.h'
include 'upointer.h'
include 'comblk.h'
real*8 f8
real*8 xi, eta, zeta, llog, lre, xtemp, yspline
integer n, dir, j
real*8 P1(3), P2(3), g(3)
real*8 x, y, z, temp
logical flag1, flag2
flag1 = .false.
flag2 = .false.
if (hr(up(103)+7) .eq. 2) then ! linear
  P1(1) = hr(up(28))
  P2(1) = hr(up(28)+1)
  P1(2) = hr(up(29))
  P2(2) = hr(up(29)+1)
  P1(3) = hr(up(30))
  P2(3) = hr(up(30)+1)
  g(1) = P2(1)-P1(1)
  g(2) = P2(2)-P1(2)
  g(3) = P2(3)-P1(3)
  x = P1(1) + xi*g(1)
  y = P1(2) + eta*g(2)
  z = P1(3) + zeta*g(3)
endif
if (dir .eq. 2) f8 = y
if (dir .eq. 3) f8 = z

elseif (hr(up(103)+7) .gt. 2) then ! Spline
  j = hr(up(103)+7)
P1(1) = hr(up(28))
P2(1) = hr(up(28)+j)
P1(2) = hr(up(29))
P2(2) = hr(up(29)+j)
P1(3) = hr(up(30))
P2(3) = hr(up(30)+j)
g(1) = P2(1)−P1(1)
g(2) = P2(2)−P1(2)
g(3) = P2(3)−P1(3)
xtemp = P1(2) + eta * g(2)

  int = 0
  do i = 1, j ! in which splinecurve?
    if (xtemp.ge.hr(up(29)+i−1).and.xtemp.le.hr(up(29)+i)) then
      int = i−1
    endif
  enddo
  y spline = hr(up(71)+int) *(xtemp−hr(up(29)+int)) *
  $ (xtemp−hr(up(29)+int))*(xtemp−hr(up(29)+int)) *
  $ + hr(up(72)+int) *(xtemp−hr(up(29)+int)) *
  $ (xtemp−hr(up(29)+int)) *
  $ + hr(up(73)+int) *(xtemp−hr(up(29)+int)) *
  $ + hr(up(74)+int)

  if (dir .eq. 2) f8 = xtemp
  if (dir .eq. 1) f8 = P1(1) + xi* g(1)
  if (dir .eq. 3) f8 = yspline
endif

return
derend

function f9(xi, eta, zeta, dir)
include 'pointer.h'
include 'upointer.h'
include 'comblk.h'
real*8 f9
real*8 xi, eta, zeta, llog, lre, xtemp, yspline
integer n, dir, j
real*8 P1(3), P2(3), g(3)
real*8 x, y, z, temp
logical flag1, flag2
632  if (hr(up(103)+8) .eq. 2) then ! linear
633      P1(1) = hr(up(31))
634      P2(1) = hr(up(31)+1)
635      P1(2) = hr(up(32))
636      P2(2) = hr(up(32)+1)
637      P1(3) = hr(up(33))
638      P2(3) = hr(up(33)+1)
639
640      g(1) = P2(1)−P1(1)
641      g(2) = P2(2)−P1(2)
642      g(3) = P2(3)−P1(3)
643
644      x = P1(1) + x*i*g(1)
645      y = P1(2) + eta*g(2)
646      z = P1(3) + zeta*g(3)
647
648      if (dir .eq. 1) f9 = x
649      if (dir .eq. 2) f9 = y
650      if (dir .eq. 3) f9 = z
651  elseif (hr(up(103)+8) .gt. 2) then ! Spline
652      j = hr(up(103)+8)
653      P1(1) = hr(up(31))
654      P2(1) = hr(up(31)+j)
655      P1(2) = hr(up(32))
656      P2(2) = hr(up(32)+j)
657      P1(3) = hr(up(33))
658      P2(3) = hr(up(33)+j)
659
660      g(1) = P2(1)−P1(1)
661      g(2) = P2(2)−P1(2)
662      g(3) = P2(3)−P1(3)
663
664      xtemp = P1(3) + zeta * g(3)
665
666      int = 0
667
668      do i = 1 , j ! in which splinecurve?
669        if (xtemp.ge.hr(up(33)+i−1).and.xtemp .le. hr(up(33)+i)) then
670          int = i−1
671        endif
672      enddo
673
674      yspline = hr(up(75)+int)∗(xtemp − hr(up(33)+int))∗
675      $(xtemp − hr(up(33)+int))∗(xtemp − hr(up(33)+int))∗$
676      $+ hr(up(76)+int) ∗ (xtemp − hr(up(33)+int))∗$
677      $+ hr(up(77)+int) ∗ (xtemp − hr(up(33)+int))+$
678      $+ hr(up(78)+int)$
679
680      if (dir .eq. 3) f9 = xtemp
681      if (dir .eq. 2) f9 = P1(2) + eta*g(2)
682      if (dir .eq. 1) f9 = yspline
683  endif
684
685  return
686 end
function f10(xi, eta, zeta, dir)

include 'pointer.h'
include 'upointer.h'
include 'comblk.h'

real*8 f10
real*8 xi, eta, zeta, llog, lre, xtemp, yspline
integer n, dir, j
real*8 P1(3), P2(3), g(3)
real*8 x, y, z, temp
logical flag1, flag2

if (hr(up(103)+9) .eq. 2) then ! linear
  P1(1) = hr(up(34))
  P2(1) = hr(up(34)+1)
  P1(2) = hr(up(35))
  P2(2) = hr(up(35)+1)
  P1(3) = hr(up(36))
  P2(3) = hr(up(36)+1)
  g(1) = P2(1)−P1(1)
  g(2) = P2(2)−P1(2)
  g(3) = P2(3)−P1(3)
  x = P1(1) + xi * g(1)
  y = P1(2) + eta * g(2)
  z = P1(3) + zeta * g(3)
  if (dir .eq. 1) f10 = x
  if (dir .eq. 2) f10 = y
  if (dir .eq. 3) f10 = z
elseif (hr(up(103)+9) .gt. 2) then ! Spline
  j = hr(up(103)+9)
  P1(1) = hr(up(34))
  P2(1) = hr(up(34)+j)
  P1(2) = hr(up(35))
  P2(2) = hr(up(35)+j)
  P1(3) = hr(up(36))
  P2(3) = hr(up(36)+j)
  g(1) = P2(1)−P1(1)
  g(2) = P2(2)−P1(2)
  g(3) = P2(3)−P1(3)
  int = 0
  xtemp = P1(3) + zeta * g(3)
  do i = 1, j ! in which splinecurve?
    if (xtemp .ge. hr(up(36)+i−1).and.xtemp .le. hr(up(36)+i)) then
    int = i−1
  endif
endo
yspline = hr(up(79)+int)*(xtemp - hr(up(36)+int)) * 
$ & (xtemp - hr(up(36)+int))*(xtemp - hr(up(36)+int)) 
$ + hr(up(80)+int) * (xtemp - hr(up(36)+int)) * 
$ (xtemp - hr(up(36)+int)) 
$ + hr(up(81)+int) * (xtemp - hr(up(36)+int)) 
$ + hr(up(82)+int) 
if (dir .eq. 3) f10 = xtemp 
if (dir .eq. 2) f10 = P1(2) + eta*g(2) 
if (dir .eq. 1) f10 = yspline 
endif 
return 
end 

function f11(xi, eta, zeta, dir) 
include 'pointer.h' 
include 'upointer.h' 
include 'comblk.h' 
real*8 f11 
real*8 xi, eta, zeta, llog, lre, xtemp, yspline 
integer n, dir, j 
real*8 P1(3), P2(3), g(3) 
real*8 x, y, z, temp 
logical flag1, flag2 
if (hr(up(103)+10) .eq. 2) then ! linear 
P1(1) = hr(up(37)) 
P2(1) = hr(up(37)+1) 
P1(2) = hr(up(38)) 
P2(2) = hr(up(38)+1) 
P1(3) = hr(up(39)) 
P2(3) = hr(up(39)+1) 
g(1) = P2(1)-P1(1) 
g(2) = P2(2)-P1(2) 
g(3) = P2(3)-P1(3) 
x = P1(1) + xi*g(1) 
y = P1(2) + eta*g(2) 
z = P1(3) + zeta*g(3) 
endif 
elseif (hr(up(103)+10) .gt. 2) then ! Spline 
j = hr(up(103)+10) 
P1(1) = hr(up(37)) 
P2(1) = hr(up(37)+j) 
P1(2) = hr(up(38)) 
P2(2) = hr(up(38)+j)
P1(3) = hr(up(39))
P2(3) = hr(up(39)+j)
g(1) = P2(1)−P1(1)
g(2) = P2(2)−P1(2)
g(3) = P2(3)−P1(3)

int = 0
xtemp = P1(3) + zeta * g(3)
do i = 1, j ! in which splinecurve?
  if (xtemp .ge. hr(up(39)+i−1).and. xtemp .le. hr(up(39)+i)) then
    int = i−1
  endif
enddo

yspline = hr(up(83)+int) * (xtemp − hr(up(39)+int)) * $ (xtemp − hr(up(39)+int)) * (xtemp − hr(up(39)+int)) *
  + hr(up(84)+int) * (xtemp − hr(up(39)+int)) *
  + hr(up(85)+int) * (xtemp − hr(up(39)+int)) *
  + hr(up(86)+int)
if (dir .eq. 3) f11 = xtemp
if (dir .eq. 2) f11 = P1(2) + eta*g(2)
if (dir .eq. 1) f11 = yspline
endif
return
end

function f12(xi, eta, zeta, dir)
include 'pointer.h'
include 'upointer.h'
include 'comblk.h'
real*8 f12
real*8 xi, eta, zeta, llog, lre, xtemp, yspline
integer n, dir, j
real*8 P1(3), P2(3), g(3)
real*8 x, y, z, temp
logical flag1, flag2

if (hr(up(103)+11) .eq. 2) then ! linear
  P1(1) = hr(up(40))
P2(1) = hr(up(40)+1)
P1(2) = hr(up(41))
P2(2) = hr(up(41)+1)
P1(3) = hr(up(42))
P2(3) = hr(up(42)+1)
g(1) = P2(1)−P1(1)
g(2) = P2(2)−P1(2)
g(3) = P2(3)−P1(3)

x = P1(1) + xi*g(1)
y = P1(2) + eta*g(2)
z = P1(3) + zeta*g(3)

if (dir .eq. 1) f12 = x
if (dir .eq. 2) f12 = y
if (dir .eq. 3) f12 = z

else if (hr(up(103)+11) .gt . 2) then ! Spline
  j = hr(up(103)+11)
P1(1) = hr(up(40))
P2(1) = hr(up(40)+j)
P1(2) = hr(up(41))
P2(2) = hr(up(41)+j)
P1(3) = hr(up(42))
P2(3) = hr(up(42)+j)

  g(1) = P2(1)−P1(1)
g(2) = P2(2)−P1(2)
g(3) = P2(3)−P1(3)

  int = 0

  xtemp = P1(3) + zeta * g(3)

  do i = 1, j ! in which splinecurve?
    if (xtemp.ge.hr(up(42)+i−1).and.xtemp .le. hr(up(42)+i)) then
      int = i−1
    endif
  enddo
  y spline = hr(up(87)+int)*(xtemp−hr(up(42)+int))*
             $ (xtemp−hr(up(42)+int))*(xtemp−hr(up(42)+int))$
             $ + hr(up(88)+int) * (xtemp − hr(up(42)+int))$
             $ + hr(up(89)+int) * (xtemp − hr(up(42)+int))$
             $ + hr(up(90)+int)$

  if (dir .eq. 3) f12 = xtemp
  if (dir .eq. 2) f12 = P1(2) + eta*g(2)
  if (dir .eq. 1) f12 = y spline

  endif

return
end
### Sourcecode of alloctfiarray

Function allocating the arrays user for TFI.

```fortran
subroutine alloctfiarray(tempnn)
  implicit none
  integer tempnn(12)
  logical setvar, ualloc

  setvar = ualloc(43, 'A1', tempnn(1), 2)
  setvar = ualloc(44, 'B1', tempnn(1), 2)
  setvar = ualloc(45, 'C1', tempnn(1), 2)
  setvar = ualloc(46, 'D1', tempnn(1), 2)
  setvar = ualloc(47, 'A2', tempnn(2), 2)
  setvar = ualloc(48, 'B2', tempnn(2), 2)
  setvar = ualloc(49, 'C2', tempnn(2), 2)
  setvar = ualloc(50, 'D2', tempnn(2), 2)
  setvar = ualloc(51, 'A3', tempnn(3), 2)
  setvar = ualloc(52, 'B3', tempnn(3), 2)
  setvar = ualloc(53, 'C3', tempnn(3), 2)
  setvar = ualloc(54, 'D3', tempnn(3), 2)
  setvar = ualloc(55, 'A4', tempnn(4), 2)
  setvar = ualloc(56, 'B4', tempnn(4), 2)
  setvar = ualloc(57, 'C4', tempnn(4), 2)
  setvar = ualloc(58, 'D4', tempnn(4), 2)
  setvar = ualloc(59, 'A5', tempnn(5), 2)
  setvar = ualloc(60, 'B5', tempnn(5), 2)
  setvar = ualloc(61, 'C5', tempnn(5), 2)
  setvar = ualloc(62, 'D5', tempnn(5), 2)
  setvar = ualloc(63, 'A6', tempnn(6), 2)
  setvar = ualloc(64, 'B6', tempnn(6), 2)
  setvar = ualloc(65, 'C6', tempnn(6), 2)
  setvar = ualloc(66, 'D6', tempnn(6), 2)
  setvar = ualloc(67, 'A7', tempnn(7), 2)
  setvar = ualloc(68, 'B7', tempnn(7), 2)
  setvar = ualloc(69, 'C7', tempnn(7), 2)
  setvar = ualloc(70, 'D7', tempnn(7), 2)
  setvar = ualloc(71, 'A8', tempnn(8), 2)
  setvar = ualloc(72, 'B8', tempnn(8), 2)
  setvar = ualloc(73, 'C8', tempnn(8), 2)
  setvar = ualloc(74, 'D8', tempnn(8), 2)
  setvar = ualloc(75, 'A9', tempnn(9), 2)
  setvar = ualloc(76, 'B9', tempnn(9), 2)
  setvar = ualloc(77, 'C9', tempnn(9), 2)
  setvar = ualloc(78, 'D9', tempnn(9), 2)
  setvar = ualloc(79, 'A10', tempnn(10), 2)
  setvar = ualloc(80, 'B10', tempnn(10), 2)
  setvar = ualloc(81, 'C10', tempnn(10), 2)
  setvar = ualloc(82, 'D10', tempnn(10), 2)
  setvar = ualloc(83, 'A11', tempnn(11), 2)
  setvar = ualloc(84, 'B11', tempnn(11), 2)
  setvar = ualloc(85, 'C11', tempnn(11), 2)
end subroutine alloctfiarray
```
setvar = ualloc(86, 'D11', tempnn(11), 2)
setvar = ualloc(87, 'A12', tempnn(12), 2)
setvar = ualloc(88, 'B12', tempnn(12), 2)
setvar = ualloc(89, 'C12', tempnn(12), 2)
setvar = ualloc(90, 'D12', tempnn(12), 2)
setvar = ualloc(7, 'TFI1_N1', tempnn(1), 2)
setvar = ualloc(8, 'TFI1_N2', tempnn(1), 2)
setvar = ualloc(9, 'TFI1_N3', tempnn(1), 2)
setvar = ualloc(10, 'TFI2_N1', tempnn(2), 2)
setvar = ualloc(11, 'TFI2_N2', tempnn(2), 2)
setvar = ualloc(12, 'TFI2_N3', tempnn(2), 2)
setvar = ualloc(13, 'TFI3_N1', tempnn(3), 2)
setvar = ualloc(14, 'TFI3_N2', tempnn(3), 2)
setvar = ualloc(15, 'TFI3_N3', tempnn(3), 2)
setvar = ualloc(16, 'TFI4_N1', tempnn(4), 2)
setvar = ualloc(17, 'TFI4_N2', tempnn(4), 2)
setvar = ualloc(18, 'TFI4_N3', tempnn(4), 2)
setvar = ualloc(19, 'TFI5_N1', tempnn(5), 2)
setvar = ualloc(20, 'TFI5_N2', tempnn(5), 2)
setvar = ualloc(21, 'TFI5_N3', tempnn(5), 2)
setvar = ualloc(22, 'TFI6_N1', tempnn(6), 2)
setvar = ualloc(23, 'TFI6_N2', tempnn(6), 2)
setvar = ualloc(24, 'TFI6_N3', tempnn(6), 2)
setvar = ualloc(25, 'TFI7_N1', tempnn(7), 2)
setvar = ualloc(26, 'TFI7_N2', tempnn(7), 2)
setvar = ualloc(27, 'TFI7_N3', tempnn(7), 2)
setvar = ualloc(28, 'TFI8_N1', tempnn(8), 2)
setvar = ualloc(29, 'TFI8_N2', tempnn(8), 2)
setvar = ualloc(30, 'TFI8_N3', tempnn(8), 2)
setvar = ualloc(31, 'TFI9_N1', tempnn(9), 2)
setvar = ualloc(32, 'TFI9_N2', tempnn(9), 2)
setvar = ualloc(33, 'TFI9_N3', tempnn(9), 2)
setvar = ualloc(34, 'TFI10_N1', tempnn(10), 2)
setvar = ualloc(35, 'TFI10_N2', tempnn(10), 2)
setvar = ualloc(36, 'TFI10_N3', tempnn(10), 2)
setvar = ualloc(37, 'TFI11_N1', tempnn(11), 2)
setvar = ualloc(38, 'TFI11_N2', tempnn(11), 2)
setvar = ualloc(39, 'TFI11_N3', tempnn(11), 2)
setvar = ualloc(40, 'TFI12_N1', tempnn(12), 2)
setvar = ualloc(41, 'TFI12_N2', tempnn(12), 2)
setvar = ualloc(42, 'TFI12_N3', tempnn(12), 2)
return
d end subroutine clear_tfi_array()
integer tempnn(12)
logical setvar, ualloc
setvar = ualloc(43, 'A1', 0, 2)
setvar = ualloc(44, 'B1', 0, 2)
setvar = ualloc(45, 'C1', 0, 2)
setvar = ualloc(46, 'D1', 0, 2)
setvar = ualloc(47, 'A2', 0, 2)
```c
setvar = ualloc(48, 'B2', 0, 2)
setvar = ualloc(49, 'C2', 0, 2)
setvar = ualloc(50, 'D2', 0, 2)
setvar = ualloc(51, 'A3', 0, 2)
setvar = ualloc(52, 'B3', 0, 2)
setvar = ualloc(53, 'C3', 0, 2)
setvar = ualloc(54, 'D3', 0, 2)
setvar = ualloc(55, 'A4', 0, 2)
setvar = ualloc(56, 'B4', 0, 2)
setvar = ualloc(57, 'C4', 0, 2)
setvar = ualloc(58, 'D4', 0, 2)
setvar = ualloc(59, 'A5', 0, 2)
setvar = ualloc(60, 'B5', 0, 2)
setvar = ualloc(61, 'C5', 0, 2)
setvar = ualloc(62, 'A6', 0, 2)
setvar = ualloc(63, 'B6', 0, 2)
setvar = ualloc(64, 'C6', 0, 2)
setvar = ualloc(65, 'D6', 0, 2)
setvar = ualloc(66, 'A7', 0, 2)
setvar = ualloc(67, 'B7', 0, 2)
setvar = ualloc(68, 'C7', 0, 2)
setvar = ualloc(69, 'D7', 0, 2)
setvar = ualloc(70, 'A8', 0, 2)
setvar = ualloc(71, 'B8', 0, 2)
setvar = ualloc(72, 'C8', 0, 2)
setvar = ualloc(73, 'D8', 0, 2)
setvar = ualloc(74, 'A9', 0, 2)
setvar = ualloc(75, 'B9', 0, 2)
setvar = ualloc(76, 'C9', 0, 2)
setvar = ualloc(77, 'D9', 0, 2)
setvar = ualloc(78, 'A10', 0, 2)
setvar = ualloc(79, 'B10', 0, 2)
setvar = ualloc(80, 'C10', 0, 2)
setvar = ualloc(81, 'D10', 0, 2)
setvar = ualloc(82, 'A11', 0, 2)
setvar = ualloc(83, 'B11', 0, 2)
setvar = ualloc(84, 'C11', 0, 2)
setvar = ualloc(85, 'D11', 0, 2)
setvar = ualloc(86, 'A12', 0, 2)
setvar = ualloc(87, 'B12', 0, 2)
setvar = ualloc(88, 'C12', 0, 2)
setvar = ualloc(89, 'D12', 0, 2)
setvar = ualloc(90, 'A13', 0, 2)
setvar = ualloc(91, 'B13', 0, 2)
setvar = ualloc(92, 'C13', 0, 2)
setvar = ualloc(93, 'D13', 0, 2)
setvar = ualloc(94, 'A14', 0, 2)
setvar = ualloc(95, 'B14', 0, 2)
setvar = ualloc(96, 'C14', 0, 2)
setvar = ualloc(97, 'D14', 0, 2)
setvar = ualloc(98, 'A15', 0, 2)
setvar = ualloc(99, 'B15', 0, 2)
setvar = ualloc(100, 'C15', 0, 2)
setvar = ualloc(101, 'D15', 0, 2)
setvar = ualloc(102, 'A16', 0, 2)
setvar = ualloc(103, 'B16', 0, 2)
setvar = ualloc(104, 'C16', 0, 2)
setvar = ualloc(105, 'D16', 0, 2)
setvar = ualloc(106, 'A17', 0, 2)
setvar = ualloc(107, 'B17', 0, 2)
setvar = ualloc(108, 'C17', 0, 2)
setvar = ualloc(109, 'D17', 0, 2)
setvar = ualloc(110, 'A18', 0, 2)
setvar = ualloc(111, 'B18', 0, 2)
setvar = ualloc(112, 'C18', 0, 2)
setvar = ualloc(113, 'D18', 0, 2)
setvar = ualloc(114, 'A19', 0, 2)
setvar = ualloc(115, 'B19', 0, 2)
setvar = ualloc(116, 'C19', 0, 2)
setvar = ualloc(117, 'D19', 0, 2)
setvar = ualloc(118, 'A20', 0, 2)
setvar = ualloc(119, 'B20', 0, 2)
setvar = ualloc(120, 'C20', 0, 2)
setvar = ualloc(121, 'D20', 0, 2)
setvar = ualloc(122, 'A21', 0, 2)
setvar = ualloc(123, 'B21', 0, 2)
setvar = ualloc(124, 'C21', 0, 2)
setvar = ualloc(125, 'D21', 0, 2)
setvar = ualloc(126, 'A22', 0, 2)
setvar = ualloc(127, 'B22', 0, 2)
setvar = ualloc(128, 'C22', 0, 2)
setvar = ualloc(129, 'D22', 0, 2)
setvar = ualloc(130, 'A23', 0, 2)
setvar = ualloc(131, 'B23', 0, 2)
setvar = ualloc(132, 'C23', 0, 2)
setvar = ualloc(133, 'D23', 0, 2)
setvar = ualloc(134, 'A24', 0, 2)
setvar = ualloc(135, 'B24', 0, 2)
setvar = ualloc(136, 'C24', 0, 2)
setvar = ualloc(137, 'D24', 0, 2)
setvar = ualloc(138, 'A25', 0, 2)
setvar = ualloc(139, 'B25', 0, 2)
setvar = ualloc(140, 'C25', 0, 2)
setvar = ualloc(141, 'D25', 0, 2)
setvar = ualloc(142, 'A26', 0, 2)
setvar = ualloc(143, 'B26', 0, 2)
setvar = ualloc(144, 'C26', 0, 2)
setvar = ualloc(145, 'D26', 0, 2)
setvar = ualloc(146, 'A27', 0, 2)
setvar = ualloc(147, 'B27', 0, 2)
setvar = ualloc(148, 'C27', 0, 2)
setvar = ualloc(149, 'D27', 0, 2)
setvar = ualloc(150, 'A28', 0, 2)
setvar = ualloc(151, 'B28', 0, 2)
setvar = ualloc(152, 'C28', 0, 2)
setvar = ualloc(153, 'D28', 0, 2)
setvar = ualloc(154, 'A29', 0, 2)
setvar = ualloc(155, 'B29', 0, 2)
setvar = ualloc(156, 'C29', 0, 2)
setvar = ualloc(157, 'D29', 0, 2)
setvar = ualloc(158, 'A30', 0, 2)
setvar = ualloc(159, 'B30', 0, 2)
setvar = ualloc(160, 'C30', 0, 2)
setvar = ualloc(161, 'D30', 0, 2)
```

L Sourcecode of alloctfiarray

95
setvar = ualloc(21, 'TFI5_N3', 0, 2)
setvar = ualloc(22, 'TFI6_N1', 0, 2)
setvar = ualloc(23, 'TFI6_N2', 0, 2)
setvar = ualloc(24, 'TFI6_N3', 0, 2)
setvar = ualloc(25, 'TFI7_N1', 0, 2)
setvar = ualloc(26, 'TFI7_N2', 0, 2)
setvar = ualloc(27, 'TFI7_N3', 0, 2)
setvar = ualloc(28, 'TFI8_N1', 0, 2)
setvar = ualloc(29, 'TFI8_N2', 0, 2)
setvar = ualloc(30, 'TFI8_N3', 0, 2)
setvar = ualloc(31, 'TFI9_N1', 0, 2)
setvar = ualloc(32, 'TFI9_N2', 0, 2)
setvar = ualloc(33, 'TFI9_N3', 0, 2)
setvar = ualloc(34, 'TFI10_N1', 0, 2)
setvar = ualloc(35, 'TFI10_N2', 0, 2)
setvar = ualloc(36, 'TFI10_N3', 0, 2)
setvar = ualloc(37, 'TFI11_N1', 0, 2)
setvar = ualloc(38, 'TFI11_N2', 0, 2)
setvar = ualloc(39, 'TFI11_N3', 0, 2)
setvar = ualloc(40, 'TFI12_N1', 0, 2)
setvar = ualloc(41, 'TFI12_N2', 0, 2)
setvar = ualloc(42, 'TFI12_N3', 0, 2)
return
e end
M Sourcecode of definespline

Source code of the functions for conditioning the user defined parameters from the inputfile into a spline definition.

```fortran
subroutine defineSpline(xx, yy, zz, initbend, finalbend, n,
$ surfacenumber, A, B, C, D)

c * * F E A P * * A Finite Element Analysis Program

c .... Copyright (c) 1984–2004: Regents of the University of California
          All rights reserved

c--------------------------------------------------------------------------
c Purpose: Define an coupling BC for rectangular surfaces.
c Inputs:
  ix  -- the element connection array in feap
  x   -- the coordinates array in feap
  x,y,z -- The coordinates for the spline (support points)
  initbend -- initial bending for the spline
  finalbend -- final bending for the spline
  n   -- number of support points
  surfacenumber -- indicates which surface gets transformed

c Outputs:
  A, B, C, D -- Matrices for Splinedefintions

c--------------------------------------------------------------------------
implicit none
integer n, i, j, dir1, dir2, ntdma, nn, offset, surfacenumber
real*8 A(n-1), B(n-1), C(n-1), D(n-1), S(n), R
real*8 k(n-2,n-2), xx(n), yy(n), zz(n), h(n-1)
real*8 x(n), y(n), z(n)
real*8 initbend, finalbend

do i = 1, n-1
  do j = 1, n-1
    K(i,j) = 0
  enddo
enddo

c Organizing the data
if ( surfacenumber .eq. 1) then
  do i = 1, n
    x(i) = zz(i)
    y(i) = xx(i)
  enddo
elseif ( surfacenumber .eq. 2) then
  do i = 1, n
    x(i) = zz(i)
    y(i) = yy(i)
  enddo
elseif ( surfacenumber .eq. 3) then
  do i = 1, n
    x(i) = yy(i)
    y(i) = zz(i)
  enddo
endif
```
elseif ( surfacenumber .eq. 4) then
  do i = 1, n
    x(i) = zz(i)
y(i) = xx(i)
  enddo
elseif ( surfacenumber .eq. 5) then
  do i = 1, n
    x(i) = zz(i)
y(i) = yy(i)
  enddo
elseif ( surfacenumber .eq. 6) then
  do i = 1, n
    x(i) = yy(i)
y(i) = zz(i)
  enddo
endif

c Determine the width of the i-th interval
  do i = 1, n-1
    h(i) = x(i+1) - x(i)
  enddo

c Elements of the tri-diagonal matrix
  do i = 2, n-1
    j = i-1
    D(j) = 2*(h(i-1) + h(i))
    A(j) = h(i)
    B(j) = h(i-1)
  enddo

c Right Side
  do i = 2, n-1
    j = i-1
    C(j) = 6*((y(i+1) - y(i))/h(i) - (y(i)-y(i-1))/h(i-1))
  enddo

c Solving the tri-diagonal matrix
  ntdma = n-2
  do i = 2, ntdma
    R = B(i) / D(i-1)
    D(i) = D(i) - R*A(i-1)
    C(i) = C(i) - R*C(i-1)
  enddo
  C(ntdma) = C(ntdma)/D(ntdma)
i = ntdma-1
  continue
  if ( i .ge. 1) then
    C(i) = (C(i) - A(i)*C(i+1))/D(i)
i = i-1
  goto 111
  endif
doi = 2, n-1
  j = i-1
  S(i) = C(j)
enddo
S(1) = initbend
S(n) = finalbend
c  Computing the parameters
  do i = 1, n-1
    A(i) = (S(i+1) - S(i)) / (6 * h(i))
    B(i) = S(i) / 2
    C(i) = (y(i+1) - y(i)) / h(i) - (2 * h(i) * S(i) + h(i) * S(i+1)) / 6
    D(i) = y(i)
  enddo
return
end