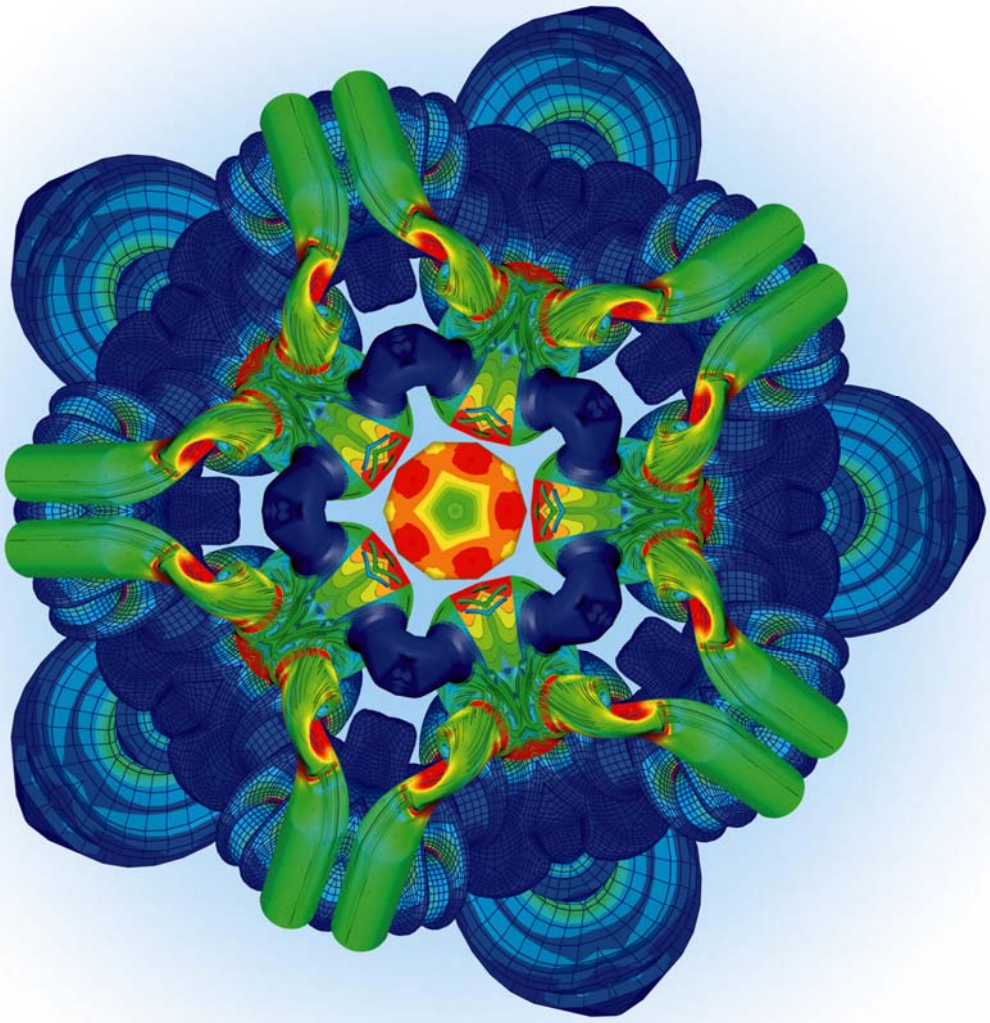
Edition 04/2009

# HYDSIM - FIRE Coupling Example

## AVL HYDSIM
## VERSION 2009



**AVL LIST GmbH**
Hans-List-Platz 1, A-8020 Graz, Austria
http://www.avl.com

AST Local Support Contact: www.avl.com/ast_support

| Revision | Date | Description | Document No. |
|---|---|---|---|
| A | 30-Apr-2009 | HYDSIM v2009 – FIRE Coupling Example | 11.0103.2009 |

# Table of Contents

**1. SAC Nozzles Example ..... 1-1**

1.2. FIRE-HYDSIM Online-Coupling ..... 1-3
1.3. ACCI Coupling and ACCI Input File ..... 1-4
1.4. Mesh and Selections ..... 1-5
1.5. Solver GUI Set Up ..... 1-8
1.5.1. Global formula ..... 1-8
1.5.2. Initialization ..... 1-10
1.5.3. Boundary conditions ..... 1-10
1.5.4. 2D-Result-Formula ..... 1-11
1.5.5. Mesh movement formula ..... 1-11
1.5.6. User function activation ..... 1-12
1.6. Simulation start procedure ..... 1-12

**2. Appendix: common.frml ..... 2-1**

**3. Appendix: mesh_deformation.frml ..... 3-1**

# 1. SAC NOZZLES EXAMPLE

## 1.1. Computational Model

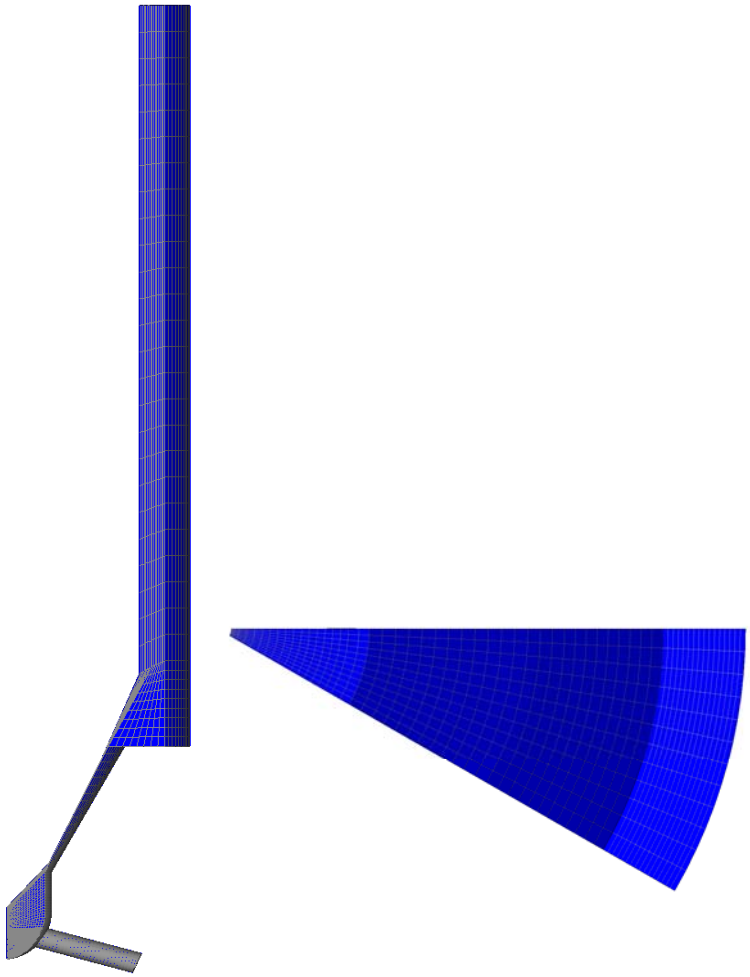


**Figure 1: SAC-nozzle cake-mesh, front view (left) and top view (right)**

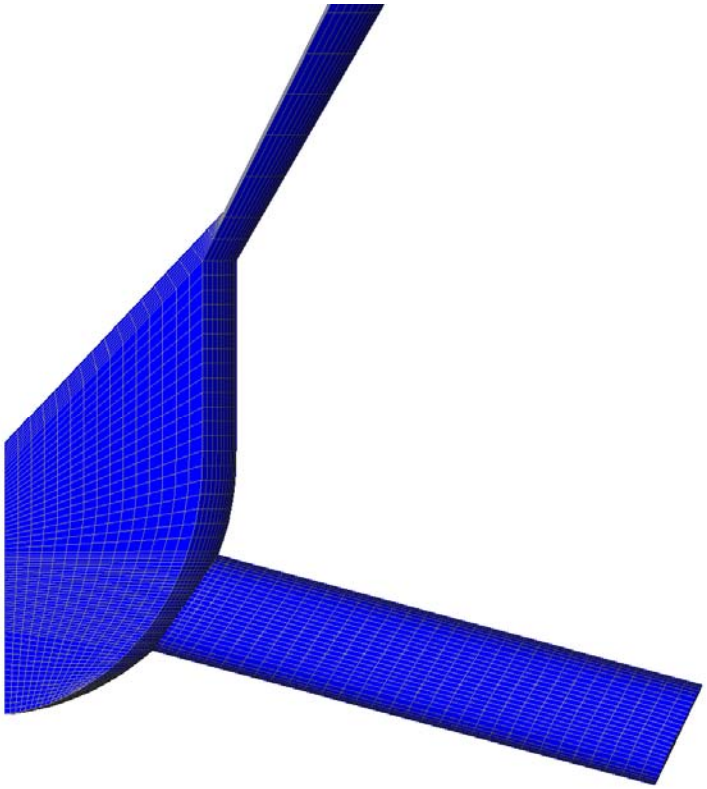


**Figure 2: SAC-nozzle cake-mesh, nozzle sac region with nozzle hole**

**Table 1: Specification of 3D volume mesh for FIRE-HYDSIM-Coupling test**

| | |
|---|---|
| mesh type | 3D; block structured |
| total number of elements | 33620 |
| tetrahedral elements | 0 |
| hexahedral elements | 33620 |
| pyramidal elements | 0 |
| octahedral elements | 0 |
| prismatic elements | 0 |
| mesh quality statistics | |
| negative volumes | 0 |
| negative normals | 0 |
| twisted faces | 0 |
| skewness > 0.85 | 0 |

**Table 2: Computing Parameters**

| | |
|---|---|
| Platform | Linux |
| Operating System | Red Hat Enterprise Linux WS release 3 |
| time steps | controlled by HYDSIM |
| start- / end-time [s] | 0 – HYDSIM-controlled-end-time |
| Run mode | Serial |
| No. of used CPU's | 1 |
| CPU time [s] | depends on HYDSIM setup |
| Elapsed time [s] | depends on HYDSIM setup |

# 1.2. FIRE-HYDSIM Online-Coupling

FIRE-HYDSIM Online-Coupling is supported by FIRE v2009. It is based on the ACCI-coupling-module and the mesh-deformation-by-formula feature, first introduced with FIRE v2008. Basically, HYDSIM provides FIRE with the current needle-lift, and FIRE provides HYDSIM with the resulting needle-force, seat-mass-flow and pressure in the nozzle-sac. The FIRE time-steps and end-time are entirely controlled by HYDSIM. Both angle- and time-mode are supported. The coupling-time-steps are equal to the FIRE time steps. The HYDSIM time steps may be smaller.

In this first version of the coupling implementation, only a longitudinal needle-motion is supported. In later versions a lateral displacement and eventually a tilting of the needle will be supported. For longitudinal motion only, a cake-segment of the entire nozzle is sufficient. A cake-mesh is also used in this verification-case, see Figure 1 and Figure 2. The mesh contains an arbitrary interface between the sac volume and the hole-channel.

The implementation of the HYDSIM coupling on the FIRE side is contained nearly entirely within user-formulas. As these formulas can be changed easily without need to recompile FIRE, it is possible to adapt the implementation to more specific needs.

The specific ingredients needed for FIRE simulation coupled with HYDSIM are:

- specific selections to indicate the different cone-sections of the needle etc. (see Table 3),
- an ACCI configuration file for the coupling (see chapter Coupling),

- a formula-file with some utility functions for the HYDSIM coupling (file common.frml, see chapter Global formula),
- a mesh deformation formula (see chapter Mesh movement),
- initialization-formulas for pressure and temperature (see chapter Solver-GUI-Set Up),
- a specific 2D-Result-formula to compute the data to be sent to HYDSIM (see chapter 2D-Result formula),
- activation of the USEBOD-user-function (implemented as formula in common.frml; see chapter User function)
- and, of course, a corresponding HYDSIM case.

The HYDSIM case is not described in this document. The starting procedure is described in the chapter Simulation start procedure.

# 1.3. ACCI Coupling and ACCI Input File

The following data is sent by HYDSIM to FIRE at each coupling time step:

- needle lift, applied in FIRE to move the mesh,
- pressure and temperature in the nozzle-volume, used in FIRE as inlet-boundary-condition,
- timestep or angle-step, if in angle mode,
- simulation end-time or end-angle, if in angle mode, so that FIRE terminates properly.

The following data is sent by FIRE to HYDSIM at each coupling step:

- force on the needle tip in longitudinal direction,
- mass flow over the seat area,
- mass flow through the nozzle holes (through outlet),
- pressure and temperature in the nozzle sac (averaged over a cell-selection),
- gas-pressure and -temperature (i.e. values at the FIRE outlet in this case),
- pressure and temperature at the FIRE inlet.

The force and mass flows are multiplied by FIRE automatically by the factor 360/cake-angle, so that HYDSIM always gets values for the entire nozzle model.

The applied ACCI-input file is reproduced below. The attribute FIRE_TO_HYDSIM contains all data sent by FIRE to HYDSIM and HYDSIM_TO_FIRE all data sent by HYDSIM to FIRE. The attributes deltaTC and endITC identify the time- or angle-step and end-time or –angle sent by HYDSIM, respectively.

**Note: if the FIRE case is not called Case, the client id 'Case' in the ACCI-Input-File must be replaced by the actual name of the case!**

```
HYDSIM at time_step_start  gets FIRE_TO_HYDSIM at DATA_USER_DEF
HYDSIM at time_step_start1 sets deltaTC        at DATA_USER_DEF
HYDSIM at time_step_start1 sets endITC         at DATA_USER_DEF
HYDSIM at time_step_end    sets HYDSIM_TO_FIRE at DATA_USER_DEF
```

```
Case at time_step_start  sets FIRE_TO_HYDSIM at DATA_USER_DEF
Case at time_step_start  gets deltaTC        at DATA_USER_DEF
Case at time_step_start  gets endITC         at DATA_USER_DEF
Case at mesh_deformation gets HYDSIM_TO_FIRE at DATA_USER_DEF
Case at pre_initialization gets HYDSIM_TO_FIRE at DATA_USER_DEF
```

This ACCI input file should be placed in the FIRE case directory, e.g. under the name *hydsim_fire.acci*. This file has to be specified when FIRE is started.

# 1.4. Mesh and Selections

The following selections, with these exact names, must be defined in the FIRE mesh. These selections, as present in the verification test mesh, are shown in Figure 3 to Figure 6.

**Note that the origin of the mesh must be the intersection point of the main axis with the plane containing the needle-seat-edge at closed position**

**Table 3: Specific mesh-selections needed for FIRE-HYDSIM-Coupling**

| *Selection name* | *Selection type* | *Interpretation* |
|---|---|---|
| needle_tip1 | boundary-faces | lowest conical section of the needle tip |
| needle_tip2 | boundary-faces | intermediate conical section of the needle tip |
| needle_tip3 | boundary-faces | upper conical section of the needle tip;<br>note that the needle-seat-cross-section is assumed through the edge between needle_tip2 and needle_tip3 |
| needle_tip | boundary-faces | must comprise of needle_tip1 to 3 |
| needle_seat | boundary-faces | conical needle-seat section (non-moving) |
| nozzle_sac | cells | the nozzle sac volume, used for computing the average sac-pressure and temperature for HYDSIM |
| PorosityPhase | cells | cells in the nozzle-gap; user for checking for blocked cells |
| inlet | boundary-faces | FIRE inlet (top surface), must be located in the nozzle volume |
| outlet | boundary-faces | FIRE outlet to the engine chamber |
| inter | nodes or cells | must contain all nodes for mesh-smoothing;<br>must NOT contain any nodes of the needle-surface or nozzle-surface!<br>Must contain the interior nodes on the lateral symmetry-surface in case of cake-meshes. |

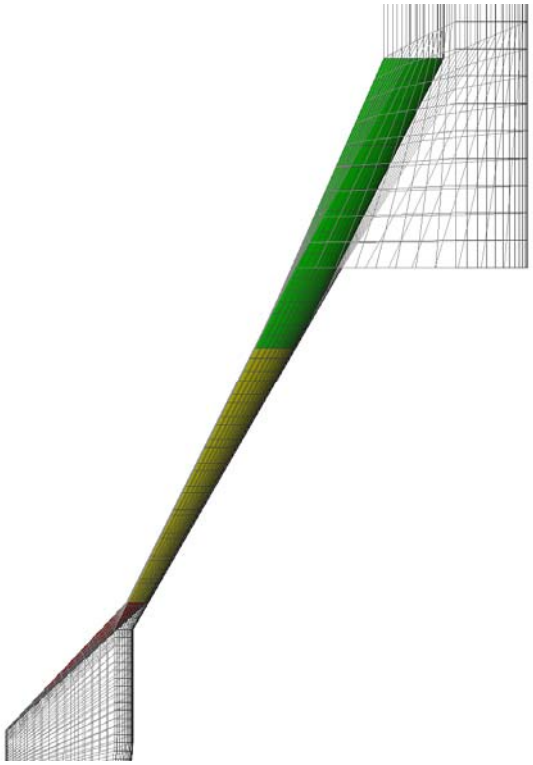


**Figure 3: Needle-tip-selections needle_tip1 (red, lowest), needle_tip2 (yellow, above) and needle_tip3 (green, top); the needle-seat-edge is assumed to be located between needle_tip2 and needle_tip3**

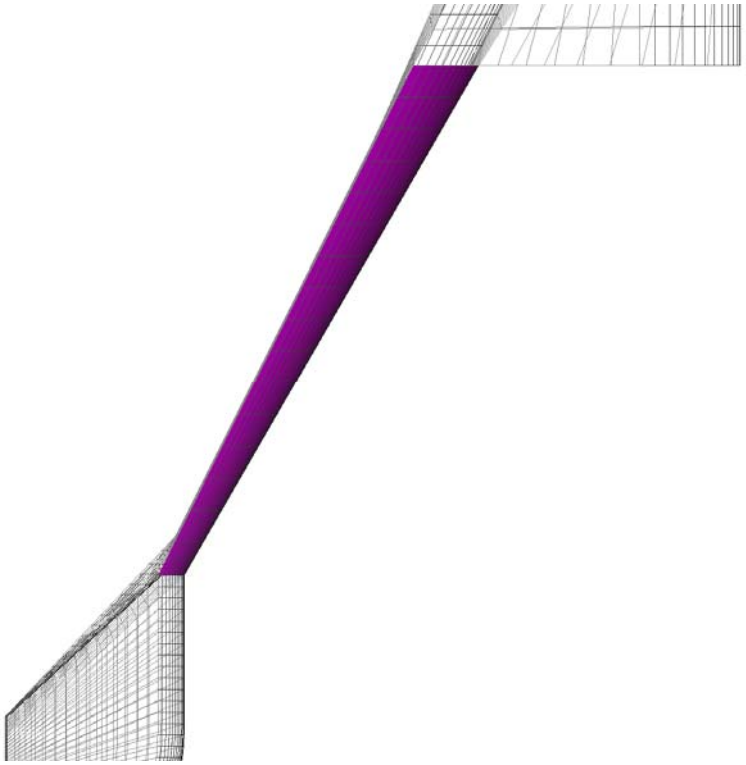


**Figure 4: Needle-seat-selection**

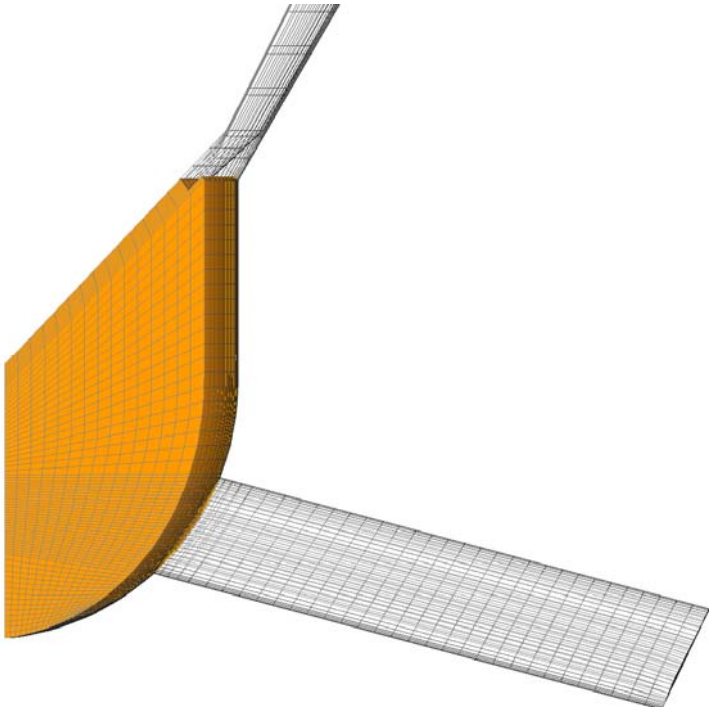


**Figure 5: Nozzle-sac cell-selection**

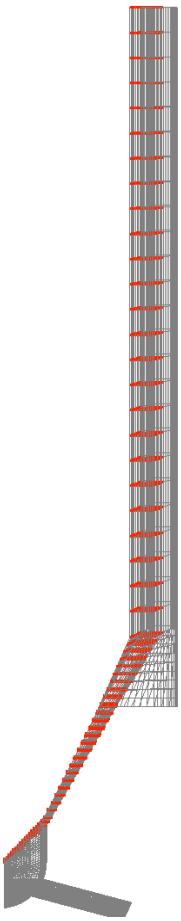


**Figure 6: Mesh-smoothing selection; must not contain nodes of the needle- or nozzle-surface**

# 1.5. Solver GUI Set Up

The verification case is set up as a multiphase-case, with phase 1 the liquid Diesel fuel and phase 2 the evaporated fuel. The cavitation model is activated. A USEBOD-user-function must be activated (activation value: 99). The user-function is implemented as a formula (see below), so no usebod.f must be compiled or linked with FIRE.

Note that the time steps and end-time are dictated by HYDSIM. The corresponding values defined in the Solver GUI are not effective.

The FIRE simulation may also run in crank angle mode. In this case, the start angle must be set to zero and the rotational speed must exactly match the speed defined for HYDSIM, which must also run in angle mode.

Below, the Global formula (at the bottom of the Solver GUI) is described first, because it defines formula functions that are used in all the other formulas. It is necessary to define the Global formula first before the other formulas are defined, unless the verification case is used or copied.

## *1.5.1. Global formula*

As mentioned, the coupling-implementation itself is basically contained in formulas, which again are a part of the FIRE case setup. The main part of the formula code is contained in the Global formula. Unfortunately, the formula is too long to store it with the ssf-file (there is currently a limitation to 4096 characters in ssf-string-values), so the actual formula source is contained in a file *common.frml* and this file is *included* in the Global formula definition with the statement

```
#include "common.frml"
```

in the topmost field in the formula editor. The include-statement refers to the formula file rather than to include it.

The file common.frml is reproduced in the appendix. It contains functions to identify the needle-tip-cone-parameters, the cake-angle, to retrieve the HYDSIM-value, to compute porosity-factors and to block cells actually covered by the solid needle.

The file common.frml must be located in the Case-, the Calculation- or the FIRE project directory. FIRE and the CFDWM search for the file in these directories and exactly in this order. It is recommended to put the common.frml into the Case-directory. The content of the common.frml is reproduced in the appendix.

There are some user-parameters defined in the Global formula directly. The figure below shows the content of the global formula together with opened Edit formula parameters to define the user-parameters.

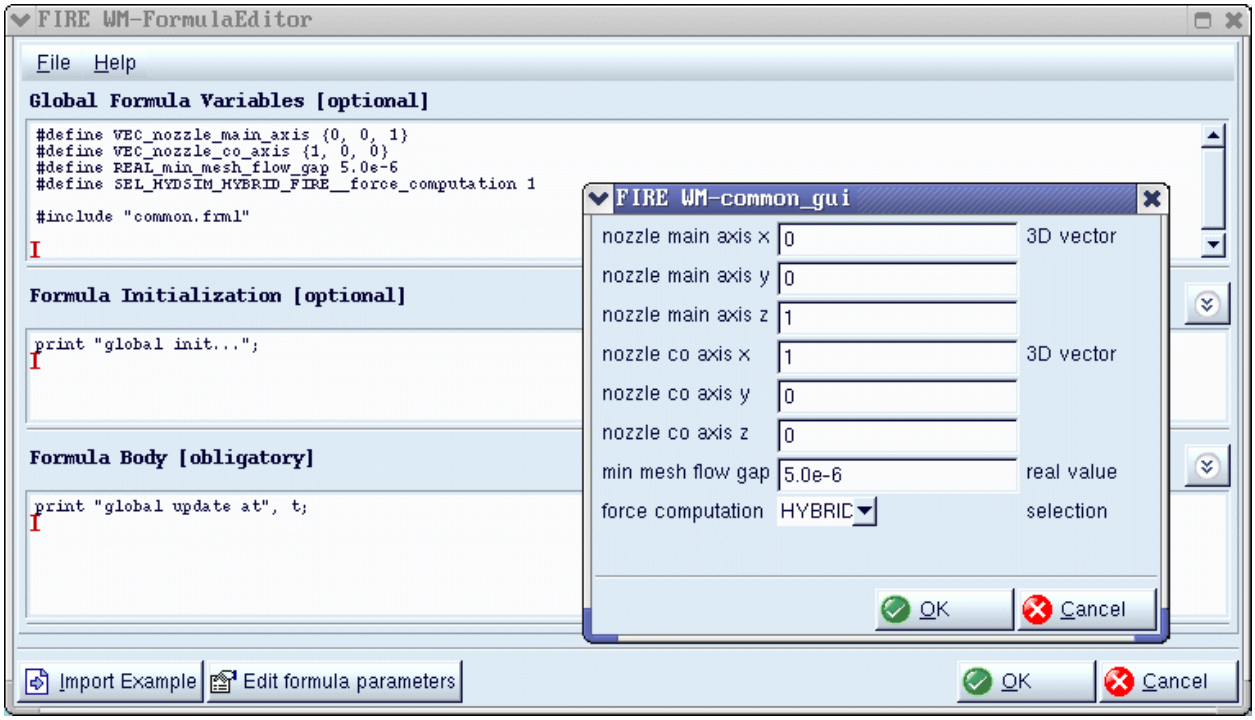



**Figure 7: Global Formula with open Formula Parameters Window**

The user parameters specify the *nozzle main axis* and the *nozzle co axis* (vectors need not be scaled to length one). Note again that he origin of the mesh must be the intersection point of the nozzle main axis with the plane containing the needle-seat-edge at closed position! The co-axis is arbitrary, but should be normal to the main axis. It must not be parallel to the main axis. however.

The *min mesh flow gap* specifies the smalles flow gap at the needle seat that should be realized with real mesh-deformation. The flow gap in the mesh will never be smaller than this value. This limit has been introduced to keep a good mesh quality. The value should be set to a value so that the resulting mesh-movement keeps a sufficient quality. The needle-movement exceeding this limit, specifically the position of the closed needle, is simulation by blocking the cells that are actually within the solid material of the needle, according to the true needle position.

There are three alternatives for the computation of the needle force: *HYDSIM*, *FIRE* amd *HYBRID*. With the *FIRE*-method, the needle force is computed by integrating the pressure force over the needle surface.

In case of very small needle lift, while cell-blocking is applied, the high-pressure-field upstream of the nozzle seat edge may extend over some area below the nozzle seat, so that the needle force computed by FIRE may be too large. For this case the *HYBRID* option has been implemented, which informs HYDSIM to compute the needle force itself, based on the sac-pressure returned by FIRE, while cells are blocked. For larger needle lift that can be represented entirely by mesh-deformation, the needle force is computed in FIRE as for the *FIRE*-option mentioned above.

With the *HYDSIM* option HYDSIM will always compute the needle force itself.

The common.frml contains also the user-function USEBOD, which is used to simulate the needle movement exceeding the minimal flow gap in the mesh.

## 1.5.2. Initialization

For initialization, the pressure and temperatures provided by HYDSIM are used for the part above the needle-seat. For the lower part, sac and holes, the chamber-pressure is initialized. This is done by the following initial pressure formula.

```
double pInl;

$$init
pInl = hydsimValue("p_nozV=", 0);


double z = x . baseZ;

if(z < zMinTip3) return 5.0e5;
return pInl;
```

The function *hydsimValue* is implemented in the Global formula (more precisely: in common.frml).

**Note: an initial pressure of 500000 Pa (value 5.0e5 in the formula) is used for the lower part. This value should be adjusted to the pressure prescribed at the nozzle outlet (see below).**

The formula for the initial temperature is:

```
double Tinl;
$$init
Tinl = hydsimValue("T_nozV=", 0);
$$formula
return Tinl;
```

## 1.5.3. Boundary conditions

The inlet-condition is static-pressure. The pressure provided by HYDSIM is applied by using the boundary-condition-formula:

```
double pInl;
$$init
pInl = hydsimValue("p_nozV=", 0);

$$formula
return pInl;
```

The HYDSIM-temperature is prescribed by using the same formula for the boundary-temperatures of all phases as for the initial temperature (see above).

At the nozzle-outlet to the chamber, a static pressure boundary condition with constantly 500000 Pa is used in this case.

### 1.5.4. 2D-Result-Formula

The computation of the data for HYDSIM (force acting on the needle, mass fluxes, sac pressure etc.) is triggered by a 2D-Result-Formula, applied to an arbitrary selection (the selection does not matter – NoName may be kept). The formula is given below. It just calls a formula function defined in common.frml.

```
$$init
if(init) computeHydsimData();


$$formula
return |n|;
```

### 1.5.5. Mesh movement formula

The mesh movement is done by a mesh-deformation formula. Only one (static) mesh in a reference position needed for the coupled simulation, a FAME moving mesh is not needed. The reference position shall be a position at medium needle lift (the needle lift at this reference position is determined by the formula automatically – it need not be specified by the user).

The mesh movement formula is defined in the Solver GUI, under Mesh deformation. The mesh deformation has to be activated first, then a Formula-Editor can be opened to enter the formula (see appendix for the formula text).

**Note that the Global formula must be defined first in the Solver GUI, because the mesh-deformation-formula calls some functions defined in the Global formula.**

The same mesh deformation formula can be used for all HYDSIM-coupled simulations with SAC-nozzles, as long as the specific selections named according to Table 3 are present.

The mesh-deformation-formula shifts the needle surface according to the needle lift received from HYDSIM. To prevent collapsing cells at needle lift zero, or close to zero, a minimal gap size (which is measured normal to the needle-seat-cone and can be defined by the user in the file common.frml, see appendix) is ensured by shifting back surface-nodes closer to the needle-seat-cone as this minimum. Its value is defined by by the user in the Global formula (see *min mesh flow gap*). After that, the mesh interior is smoothed. To account for the effects of the solid needle material in the area where nodes have been shifted back due to the minimal gap size restriction, *blocking* is applied. *Blocking* means that a velocity of zero (or very close to zero) is imposed on these cells. The flow is “blocked” by appropriate source-terms in the momentum equations (of all phases, if multiphase). All cells, whose center is within the volume of the solid needle (according to the true needle position as sent by HYDSIM), are blocked (see Figure 1). For this method, the USEBOD-user-function, defined in the Global formula (more precisely: in the common.frml) as described in the chapter User-function is needed and must be activated by setting the activation-flag to **99**.

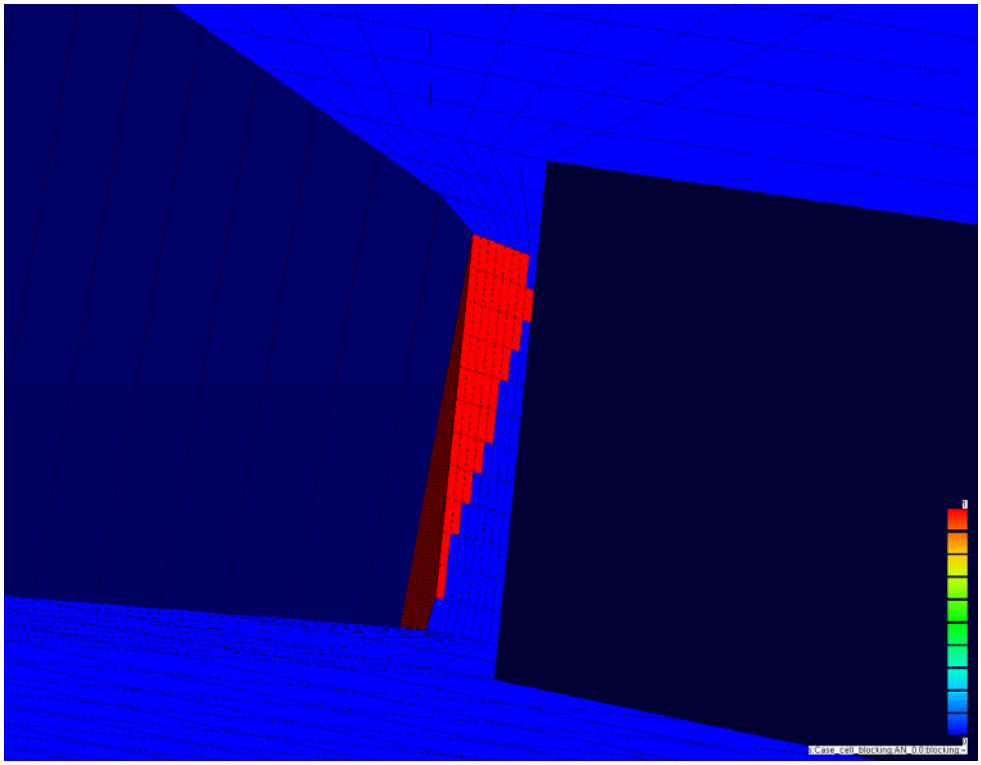


**Figure 8: Handling of solid needle intrusion – method A: cell-blocking; at this position (which is at needle lift zero), the red cells in the nozzle-gap are blocked; (the view is along the needle axis direction, so that the distances in this direction look much smaller than they actually are)**

Mesh deformation by formula must be activated in the Solver GUI. The content of the mesh-deformation-formula for this case is listed in the appendix.

### *1.5.6. User function activation*

As mentioned in the previous chapters, a USEBOD user-function is needed for HYDSIM-FIRE-coupling. The activation-flag must be set to a value of 99. A value of 99 informs FIRE that the user-function is not a compiled FORTRAN user-function, but is implemented as formula-function in the Global formula (in this case in common.frml). This means that no FORTRAN *usebod.f* has actually to be compiled or linked.

## 1.6. Simulation start procedure

To start the coupled HYDSIM-BOOST-simulation, it is recommended to start first FIRE and then HYDSIM. However, both programs will wait for each other for at least 20 minutes.

FIRE can be started from the Calculation Wizard in the CFDWM. “Coupling” has to be activated, followed by the specification of the computer that should run the ACCI-Server and a port number under which it should run. The “CfdCI”-coupling-option has to be selected together with the the ACCI-input-file. Note that the ACCI-Server is started by the Wizard automatically with the FIRE solver.

**Note that HYDSIM must be started specifying the same host and port number for the ACCI-Server as for FIRE!**

A Restart is possible, if the restart-data are defined for HYDSIM according to the desired FIRE backup- or restart-file.

# 2. APPENDIX: COMMON.FRML

```
// nozzle axis base vectors (baseZ is axis direction)
double baseX[3] = VEC_nozzle_main_axis;
double baseY[3] = VEC_nozzle_co_axis;
double baseZ[3] = baseX ^ baseY;
// minimal flow gap [m]
double minFlowGap = REAL_min_mesh_flow_gap;



double zMinSeat; double zMaxSeat; double rMinSeat; double rMaxSeat;
double zMinTip1; double zMaxTip1; double rMinTip1; double rMaxTip1;
double zMinTip2; double zMaxTip2; double rMinTip2; double rMaxTip2;
double zMinTip3; double zMaxTip3; double rMinTip3; double rMaxTip3;
double nozzle_cake_angle = 2.0*Pi;
double porFrac = 0;
double deltaZ;
double deltaZRef;


char nodeFlags[1];



int nBlockedCells = 0;
int mBlockedCells = 1;
int blockedCell[1];
double blockedFrac[1];




void printMeshQuality()
{
        int nNegVols = 0;
        int nNegPartTets = 0;
        int nNegPartTetsAlt = 0;
        int nNegNormals = 0;
        double maxAspectRatio = 1.0;
        double maxCellSkewness = 0.0;
        double maxCellWarpage = 0.0;
        int i;
        double d;
        for(i = 0; i < NCELL-NUMBUF; i++){
                if(CellVolume(i) < 0.0) nNegVols++;
```

```
                nNegPartTets += CellNumNegPartTets(i);
                nNegPartTetsAlt += CellNumNegPartTetsAlt(i);
                nNegNormals += CellNumNegNormals(i);
                d = CellAspectRatio(i);
                if(d > maxAspectRatio) maxAspectRatio = d;
                d = CellSkewness(i);
                if(d > maxCellSkewness) maxCellSkewness = d;
                d = CellWarpage(i);
                if(d > maxCellWarpage) maxCellWarpage = d;
        }
        MPI_SUM_I(nNegVols);
        MPI_SUM_I(nNegPartTets);
        MPI_SUM_I(nNegPartTetsAlt);
        MPI_SUM_I(nNegNormals);
        MPI_MAX_D(maxAspectRatio);
        MPI_MAX_D(maxCellSkewness);
        MPI_MAX_D(maxCellWarpage);
        if(IAMPRO < 2){
                print "nNegVols:", nNegVols;
                print "nNegPartTets:", nNegPartTets;
                print "nNegPartTetsAlt:", nNegPartTetsAlt;
                print "nNegNormals:", nNegNormals;
                print "maxAspectRatio:", maxAspectRatio;
                print "maxCellSkewness:", maxCellSkewness;
                print "maxCellWarpage:", maxCellWarpage;
        }
}



int getConeParameters(
        char nodeFlags[],
        double zMin,
        double zMax,
        double rMin,
        double rMax,
        double phiMin,
        double phiMax)
{
        double x[3];
        double xt;
        double yt;
        double zt;
        double phi;
        double zMinLoc;
        double zMaxLoc;
        int i;
        // find highest and lowest point, min and max azimuth-angle
        zMin =  1.0e30;
```

```
        zMax = -1.0e30;
        phiMin = Pi;
        phiMax = -Pi;
        for(i = 0; i < NVERT; i++){
                if(!nodeFlags[i]) continue;
                GetVertexCoords(i, x);
                xt = baseX . x;
                yt = baseY . x;
                zt = baseZ . x;
                if(zt > zMax){
                        zMax = zt;
                        rMax = xt*xt + yt*yt;
                }
                if(zt < zMin){
                        zMin = zt;
                        rMin = xt*xt + yt*yt;
                }
                phi = atan2(yt, xt);
                if(phi > phiMax) phiMax = phi;
                if(phi < phiMin) phiMin = phi;
        }
        zMinLoc = zMin;
        zMaxLoc = zMax;
        MPI_MIN_D(zMin); MPI_MAX_D(zMax);
        MPI_MIN_D(phiMin); MPI_MAX_D(phiMax);
        if(zMinLoc != zMin) rMin = 0.0;
        if(zMaxLoc != zMax) rMax = 0.0;
        MPI_SUM_D(rMin); MPI_SUM_D(rMax);
        rMax = sqrt(rMax);
        rMin = sqrt(rMin);
}



double coneAngleDeg(double zMin, double zMax, double rMin, double rMax)
{
        double tg = (rMax-rMin) / (zMax-zMin);
        return 180.0 * atan(tg) / Pi;
}



// get delta Z of a point (z, r) from cone given by (zMin, rMin) and (zMax, rMax)
double DeltaZToCone(
        double z,
        double r,
        double zMin,
        double zMax,
        double rMin,
```

```
        double rMax)
{
        double dr = (z - zMin) / (zMax - zMin) * (rMax - rMin) + rMin - r;
        double dz = dr * (zMax - zMin) / (rMax - rMin);
        return dz;
}



// get functional and gradient of cone-surface
// at point x
void getConeFunc(
        double zMin,
        double zMax,
        double rMin,
        double rMax,
        double x[3],
        double F,
        double gradF[3])
{
        double xt; double yt; double zt;
        double f; double rc; double r2;
        double grad[3];
        xt = x . baseX;
        yt = x . baseY;
        zt = x . baseZ;
        f = (zt - zMin) / (zMax - zMin);
        rc = rMin + f * (rMax - rMin);
        r2 = xt * xt + yt * yt;
        F = r2 - rc * rc;
        grad[0] = 2.0 * xt;
        grad[1] = 2.0 * yt;
        grad[2] = - 2.0 * rc * (rMax - rMin) / (zMax - zMin);
        gradF  = grad[0] * baseX;
        gradF += grad[1] * baseY;
        gradF += grad[2] * baseZ;
}



// compute nozzle geometry parameters: needle cones, seat cone, cake-angle
void ComputeNozzleGeometry()
{
        double phiMin;
        double phiMax;
        // adjust and normalize nozzle axis
        baseX /= |baseX|;
        baseZ = baseX ^ baseY;
        baseZ /= |baseZ|;
```

```
        baseY = baseZ ^ baseX;
        baseY /= |baseY|;
        // compute seat cone parameters
        resize(nodeFlags, NVERT);
        nodeFlags = 0;
        FlagSelectedNodes("needle_seat", 1, nodeFlags);
        getConeParameters(nodeFlags, zMinSeat, zMaxSeat, rMinSeat, rMaxSeat, phiMin, phiMax);
        nozzle_cake_angle = 2.0*Pi;
        if(fmod(phiMax - phiMin, 2.0*Pi) > 2.0*Pi/360.0) nozzle_cake_angle = phiMax - phiMin;
        if(IAMPRO<2){
                print "needle seat cone: zMin, zMax, rMin, rMax, alfa:",
                        zMinSeat, zMaxSeat, rMinSeat, rMaxSeat, coneAngleDeg(zMinSeat,
zMaxSeat, rMinSeat, rMaxSeat);
                print "phi min-max:", 180.0/Pi*phiMin, 180.0/Pi*phiMax,
                        ", cake angle:", 180.0/Pi*nozzle_cake_angle, ", multiplicator:",
2.0*Pi/nozzle_cake_angle;
        }

        // compute needle-tip parameters
        nodeFlags = 0;
        FlagSelectedNodes("needle_tip1", 1, nodeFlags);
        getConeParameters(nodeFlags, zMinTip1, zMaxTip1, rMinTip1, rMaxTip1, phiMin, phiMax);
        nodeFlags = 0;
        FlagSelectedNodes("needle_tip2", 1, nodeFlags);
        getConeParameters(nodeFlags, zMinTip2, zMaxTip2, rMinTip2, rMaxTip2, phiMin, phiMax);
        nodeFlags = 0;
        FlagSelectedNodes("needle_tip3", 1, nodeFlags);
        getConeParameters(nodeFlags, zMinTip3, zMaxTip3, rMinTip3, rMaxTip3, phiMin, phiMax);

        if(IAMPRO<2){
                print "needle tip cone 1: zMin, zMax, rMin, rMax, alfa:",
                        zMinTip1, zMaxTip1, rMinTip1, rMaxTip1, coneAngleDeg(zMinTip1,
zMaxTip1, rMinTip1, rMaxTip1);
                print "needle tip cone 2: zMin, zMax, rMin, rMax, alfa:",
                        zMinTip2, zMaxTip2, rMinTip2, rMaxTip2, coneAngleDeg(zMinTip2,
zMaxTip2, rMinTip2, rMaxTip2);
                print "needle tip cone 3: zMin, zMax, rMin, rMax, alfa:",
                        zMinTip3, zMaxTip3, rMinTip3, rMaxTip3, coneAngleDeg(zMinTip3,
zMaxTip3, rMinTip3, rMaxTip3);
                print "phi min-max:", 180.0/Pi*phiMin, 180.0/Pi*phiMax;
        }

        if(IAMPRO<2) print "z needle seat:", zMinTip3;

        // get deltaZRef: deltaZ of closed nozzle
        deltaZRef = 1.0e30;
        deltaZ = DeltaZToCone(zMinTip1, rMinTip1, zMinSeat, zMaxSeat, rMinSeat, rMaxSeat);
if(deltaZ < deltaZRef) deltaZRef = deltaZ;
```

```
	deltaZ = DeltaZToCone(zMaxTip1, rMaxTip1, zMinSeat, zMaxSeat, rMinSeat, rMaxSeat);
if(deltaZ < deltaZRef) deltaZRef = deltaZ;
	deltaZ = DeltaZToCone(zMinTip2, rMinTip2, zMinSeat, zMaxSeat, rMinSeat, rMaxSeat);
if(deltaZ < deltaZRef) deltaZRef = deltaZ;
	deltaZ = DeltaZToCone(zMaxTip2, rMaxTip2, zMinSeat, zMaxSeat, rMinSeat, rMaxSeat);
if(deltaZ < deltaZRef) deltaZRef = deltaZ;
	deltaZ = DeltaZToCone(zMinTip3, rMinTip3, zMinSeat, zMaxSeat, rMinSeat, rMaxSeat);
if(deltaZ < deltaZRef) deltaZRef = deltaZ;
	deltaZ = DeltaZToCone(zMaxTip3, rMaxTip3, zMinSeat, zMaxSeat, rMinSeat, rMaxSeat);
if(deltaZ < deltaZRef) deltaZRef = deltaZ;
	deltaZRef = - deltaZRef;

	if(IAMPRO<2) print "deltaZRef:", deltaZRef;
}



// simple iterators over bnd-faces and cells in selections

char selName[256] = "";
int iSel;
int nSelElements;
int iSelElement;

int getNextBndFace(
	char name[],
	int ib)
{
	// must use selection-functions of SWIFTIO!
	int iCell; int iFace;
	if(strcmp(name, selName)){
		strcpy(selName, name);
		iSel = INDEX_OF_SELECTION(selName);
		iSelElement = 0;
		if(iSel == 0){
			nSelElements = 0;
			print "ERROR: selection", selName, "not present!";
			return 0;
		}
		nSelElements = NELEMENTS_OF_SELECTION(iSel);
	}
	if(iSelElement >= nSelElements){
		iSelElement = 0;
		return 0;
	}
	iCell = ELEMENT_OF_SELECTION(iSel, iSelElement++);
	iFace = ELEMENT_OF_SELECTION(iSel, iSelElement++);
	ib = INDEX_OF_SEL_BND_FACE(iCell+1, iFace+1) - 1;
	return 1;
```

```
}

int getNextCell(
        char name[],
        int ic)
{
        // must use selection-functions of PredefinedMeshFormulaFunctions,
        // because initially called before SwiftIO-mesh-pointer is defined!
        if(strcmp(name, selName)){
                strcpy(selName, name);
                iSel = IndexOfSelection(selName);
                iSelElement = 0;
                if(iSel == 0){
                        nSelElements = 0;
                        print "ERROR: selection", selName, "not present!";
                        return 0;
                }
                nSelElements = NumberOfSelectionElements(iSel);
        }
        if(iSelElement >= nSelElements){
                iSelElement = 0;
                return 0;
        }
        ic = ElementOfSelection(iSel, iSelElement++);
        return 1;
}



// extract values from data sent by HYDSIM

double hydsimValue(
        char id[],
        int icomp)
{
        double value = 0.0; double dummy;
        char s[1];
        int i;
        resize(s, 1024);
        if(GetUserDef("HYDSIM_TO_FIRE", s)){
                i = istrstr(s, id);
                if(i >= 0){
                        i += strlen(id);
                        if(icomp == 0){
                                if(sscanf(s[i], " %lf", value) != 1) value = 0.0;
                        }else if(icomp == 1){
                                if(sscanf(s[i], " %lf %lf", dummy, value) != 2) value = 0.0;
                        }else if(icomp == 2){
```

```
                        if(sscanf(s[i], " %lf %lf %lf", dummy, dummy, value) != 3) value
= 0.0;
                }else{
                        print "ERROR in hydsimValue: bad icomp:", icomp;
                        exit(1);
                }
        }else{
                print "ERROR in hydsimValue: cannot find id:", id;
                exit(1);
        }
}else{
        print "ERROR in hydsimValue: HYDSIM_TO_FIRE undefined!";
        exit(1);
}
resize(s, 1);
return value;
}



// compute needle forces etc. needed by HYDSIM
double force0[3] = 0.0;
double force1[3] = 0.0;

void computeHydsimData()
{
        double cakeMultiplicator;
        double force[3];
        double momentum[2];

        double Q_seat;
        double Q_holes;
        double p_sac;
        double T_sac;
        double p_gas;
        double T_gas;
        double p_topV;
        double T_topV;

        char s[1];
        double cakeAngle;
        double zSeat;

        int i;
        int i1; double z1;
        int i2; double z2;
        double size;
        double fi[3];
        double mi[3];
```

```
char isotherm = 0;
double Tiso;

int backup;

resize(s, 1024);

Tiso = hydsimValue("T_nozV=", 0);
SSFGETSTRING("SC/EC/AE/Energy", s);
if(!strcmp(s, "No")) isotherm = 1;
if(isotherm && IAMPRO < 2) print "computing HYDSIM data: isothermal simulation";

cakeMultiplicator = 2.0*Pi/nozzle_cake_angle;
if(IAMPRO < 2) print "cake-multiplicator:", cakeMultiplicator;
zSeat = zMinTip3;

// needle force and momentum
fi = 0.0;
mi = 0.0;
while(getNextBndFace("needle_tip", i)){
        fi += (PB[i]+PREF[0]) * SB[i];
        mi += (PB[i]+PREF[0]) * SB[i] ^ XB[i] ;
}
MPI_SUM_VEC_D(fi, 3); MPI_SUM_VEC_D(mi, 3);
force[0] = fi . baseZ;
force[1] = fi . baseX;
force[2] = fi . baseY;
momentum[0] = mi . baseX;
momentum[1] = mi . baseY;
force *= cakeMultiplicator;
// in case of cake, keep only longitudinal force and set momentum to 0
if(fabs(cakeAngle-2.0*Pi) > 0.001*Pi){
        force[1] = 0.0;
        force[2] = 0.0;
        momentum = 0.0;
}

// Q_seat is mass flow through z == zSeat
// mass flow of first phase!
Q_seat = 0.0;
for(i = 0; i < NFACE; i++){
        i1 = LF[i][0]-1;
        i2 = LF[i][1]-1;
        z1 = XP[i1] . baseZ - zSeat;
        z2 = XP[i2] . baseZ - zSeat;
        if(z1 < 0.0 && z2 >= 0.0)
                Q_seat -= FM[i];
        else if(z2 < 0.0 && z1 >= 0.0)
```

```
                Q_seat += FM[i];
}
MPI_SUM_D(Q_seat);
Q_seat *= cakeMultiplicator;

// Q_holes is massflow over outlet
// mass flow of first phase!
Q_holes = 0.0;
while(getNextBndFace("outlet", i)) Q_holes += FMB[i];
MPI_SUM_D(Q_holes);
Q_holes *= cakeMultiplicator;

// p_sac is avg pressure in sac, T_sac temperature
p_sac = 0.0;
T_sac = 0.0;
size = 0.0;
while(getNextCell("nozzle_sac", i)){
        p_sac += P[i] * VOL[i];
        T_sac += TM[i] * VOL[i];
        size += VOL[i];
}
MPI_SUM_D(p_sac); MPI_SUM_D(T_sac); MPI_SUM_D(size);
p_sac /= size;
p_sac += PREF[0];
T_sac /= size;
if(isotherm) T_sac = Tiso;

// p_gas is avg pressure at outlet
p_gas = 0.0;
T_gas = 0.0;
size = 0.0;
while(getNextBndFace("outlet", i)){
        p_gas += PB[i] * |SB[i]|;
        T_gas += TM[i] * |SB[i]|;
        size += |SB[i]|;
}
MPI_SUM_D(p_gas); MPI_SUM_D(T_gas); MPI_SUM_D(size);
p_gas /= size;
p_gas += PREF[0];
T_gas /= size;
if(isotherm) T_gas = Tiso;

// p_topV is avg pressure at inlet, T_topV avg temperature
p_topV = 0.0;
T_topV = 0.0;
size = 0.0;
while(getNextBndFace("inlet", i)){
        p_topV += PB[i] * |SB[i]|;
        T_topV += TMB[i] * |SB[i]|;
```

```
                size += |SB[i]|;
        }
        MPI_SUM_D(p_topV); MPI_SUM_D(T_topV); MPI_SUM_D(size);
        p_topV /= size;
        T_topV /= size;
        p_topV += PREF[0];
        if(isotherm) T_topV = Tiso;

        // restart/backup flags
        backup = 0;
        if(OUTPUT_DUE(2))
                backup = 1;
        else if(OUTPUT_DUE(3))
                backup = 2;

        // NEW: corrections for closed nozzle
        if(hydsimValue("d=", 0) <= 0.0){
                if(IAMPRO < 2){
                        print "force from fluid:", force;
                        print "corrections for closed nozzle with radii", rMinTip3, rMaxTip3;
                }
                Q_seat = 0.0;
                Q_holes = 0.0;
                force[0] = rMaxTip3 * rMaxTip3 * Pi * p_topV -
                        rMinTip3 * rMinTip3 * Pi * (p_topV - p_gas);
        }
        // smooth force vector
        if(|force1| <= 1.0e-30) force1 = force;
        force0 = force1;
        force1 = force;
        force = 0.5 * (force0 + force1);
        //if(IAMPRO < 2) print "force vector is average of", force1, "and", force0;

        if(SEL_HYDSIM_HYBRID_FIRE__force_computation == 0){
                // set zero force always, so that computed in HYDSIM
                force = 0.0;
                momentum = 0.0;
        }else if(SEL_HYDSIM_HYBRID_FIRE__force_computation == 1){
                // set zero force while any cells blocked!
                if(nBlockedCells > 0){
                        force = 0.0;
                        momentum = 0.0;
                }
        }

        // pack data into user def FIRE_TO_HYDSIM
        sprintf(s,
"F=%20.10e%20.10e%20.10e;M=%20.10e%20.10e;Q_seat=%20.10e;Q_holes=%20.10e;p_sac=%20.10e;T_sac=%
20.10e;p_gas=%20.10e;T_gas=%20.10e;p_topV=%20.10e;T_topV=%20.10e;backup=%1d;",
```

```
                force[0], force[1], force[2], momentum[0], momentum[1], Q_seat, Q_holes,
                p_sac, T_sac, p_gas, T_gas, p_topV, T_topV, backup);
        if(IAMPRO < 2) print "FIRE_TO_HYDSIM:", s;
        SetUserDef("FIRE_TO_HYDSIM", s);

        resize(s, 1);
}



double blockedVolume(
        int index)
{
        double x[3];
        int i; int nBelow; double zt;
        double zMin; double zMax; double rMin; double rMax;
        double xv[3]; double nodeVals[256];
        double F; double gradF[3];
        double Fmin, Fmax;

        GetCellCenter(index, x);
        zt = x . baseZ;
        if(zt < zMinTip1) return 0.0;
        if(zt > zMaxTip3) return 0.0;
        if(zt < zMaxTip1){
                zMin = zMinTip1;
                zMax = zMaxTip1;
                rMin = rMinTip1;
                rMax = rMaxTip1;
        }else if(zt < zMaxTip2){
                zMin = zMinTip2;
                zMax = zMaxTip2;
                rMin = rMinTip2;
                rMax = rMaxTip2;
        }else{
                zMin = zMinTip3;
                zMax = zMaxTip3;
                rMin = rMinTip3;
                rMax = rMaxTip3;
        }
        nBelow = 0;
        Fmin =  1.0e30;
        Fmax = -1.0e30;
        for(i = NumCellVerts(index)-1; i >= 0; i--){
                GetVertexCoords(CellVertexIndex(index, i), xv);
                getConeFunc(zMin, zMax, rMin, rMax, xv, F, gradF);
                nodeVals[i] = F;
                if(F < 0.0) nBelow++;
                if(F < Fmin) Fmin = F;
```

```
                if(F > Fmax) Fmax = F;
        }
        if(nBelow == 0){
                return 0.0;
        }else if(nBelow == NumCellVerts(index)){
                return CellVolume(index);
        }else{
                Fmin = -sqrt(-Fmin);
                Fmax = sqrt(Fmax);
                return CellVolume(index) * (0.0 - Fmin) / (Fmax - Fmin);
                //return IntersectionVolume(index, nodeVals);
        }
}

int isInSolid(
        int index)
{
        double x[3];
        double zt;
        double zMin; double zMax; double rMin; double rMax;
        double F; double gradF[3];
        int i; int nv;
        double dr = 0.0;
        double f = 0.2; double f1 = 1.0 - f;
        double g = 0.6; double g1 = 1.0 - g;
        double r2Min;

        GetCellCenter(index, x);
        zt = x . baseZ;
        if(zt < zMinTip1) return 0;
        if(zt > zMaxTip3) return 0;
        if(zt < zMaxTip1){
                zMin = zMinTip1;
                zMax = zMaxTip1;
                rMin = rMinTip1;
                rMax = rMaxTip1;
        }else if(zt < zMaxTip2){
                zMin = zMinTip2;
                zMax = zMaxTip2;
                rMin = rMinTip2;
                rMax = rMaxTip2;
        }else{
                zMin = zMinTip3;
                zMax = zMaxTip3;
                rMin = rMinTip3;
                rMax = rMaxTip3;
        }
        getConeFunc(zMin, zMax, rMin, rMax, x, F, gradF);
        if(F < 0.0) return 1;
```

```
        if(porFrac <= 0.0) return 0;
        r2Min = 1.0e-2 * 0.5 * (rMin + rMax); r2Min *= r2Min; r2Min = - r2Min;
        nv = NumCellVerts(index);
        for(i = 0; i < nv; i++){
                GetVertexCoords(CellVertexIndex(index, i), x);
                getConeFunc(zMin, zMax, rMin, rMax, x, F, gradF);
                if(F < r2Min) return 1;
        }
        return 0;
}



double porosityFractionIn(char selName[])
{
        char s[256];
        double vol = 0.0;
        double volBlocked = 0.0;
        int ic;

        print "computing blocked volume in", selName;
        // compute volume and blocked volume in selection selName
        while(getNextCell(selName, ic)){
                //if(ic >= NCELL-NUMBUF) continue;
                vol += CellVolume(ic);
                volBlocked += blockedVolume(ic);
        }
        //MPI_SUM_D(vol);
        //MPI_SUM_D(volBlocked);
        if(volBlocked < 1.0e-8 * vol) volBlocked = 1.0e-8 * vol;
        print "blocked volume in", selName, "=", volBlocked;
        return volBlocked / vol;
}



void ComputeBlockedCells(char selName[], double lift)
{
        char s[256];
        double vol; double volBlocked;
        int ic;

        // retrieve seat cone parameters

        strcpy(selName, "PorosityPhase");
        print "proc.", IAMPRO, ": computing blocking-sources in", selName, "lift:", lift;
        nBlockedCells = 0;
        while(getNextCell(selName, ic)){
                vol = CellVolume(ic);
```

```
                if(lift <= 1.0e-10 && isInSolid(ic)){
                        volBlocked = vol;
                }else{
                        volBlocked = blockedVolume(ic);
                }
                if(volBlocked > 0.0){
                        if(mBlockedCells <= nBlockedCells){
                                mBlockedCells *= 2;
                                resize(blockedCell, mBlockedCells);
                                resize(blockedFrac, mBlockedCells);
                        }
                        blockedCell[nBlockedCells] = ic;
                        if(lift <= 1.0e-7)
                                blockedFrac[nBlockedCells] = 1.0;
                        else
                                blockedFrac[nBlockedCells] = volBlocked / vol;
                        nBlockedCells++;
                }
        }
        print "proc.", IAMPRO, ":", nBlockedCells, "cells blocked.";
}

void ComputeBlockedCellsSimple(char selName[])
{
        char s[256];
        int ic;
        double d = hydsimValue("d=", 0);

        print "proc.", IAMPRO, ": computing blocking-sources (simple) in", selName;
        nBlockedCells = 0;
        while(getNextCell(selName, ic)){
                if(isInSolid(ic) ){
                        if(mBlockedCells <= nBlockedCells){
                                mBlockedCells *= 2;
                                resize(blockedCell, mBlockedCells);
                                resize(blockedFrac, mBlockedCells);
                        }
                        blockedFrac[nBlockedCells] = 1.0;
                        blockedCell[nBlockedCells++] = ic;
                }
        }
        print "proc.", IAMPRO, ":", nBlockedCells, "cells blocked, porFrac:", porFrac;
}

void AddBlockedCells(char selName[], double rMin, double rMax, double zMin, double zMax)
{
        int ic; double x[3]; double F; double gradF[3];
        while(getNextCell(selName, ic)){
                GetCellCenter(ic, x);
```

```
                getConeFunc(zMin, zMax, rMin, rMax, x, F, gradF);
                if(F < 0.0){
                        if(mBlockedCells <= nBlockedCells){
                                mBlockedCells *= 2;
                                resize(blockedCell, mBlockedCells);
                        }
                        blockedCell[nBlockedCells++] = ic;
                }
        }
        print "proc.", IAMPRO, ":", nBlockedCells, "cells blocked after update in", selName;
}

void BlockSolidCells(
        double vol[],
        double den[],
        double sp1[],
        double sp2[],
        double sp3[])
{
        int i; int ic;
        double fac;
        double dt_virt = DT * 1.0e-5;
        for(i = 0; i < nBlockedCells; i++){
                ic = blockedCell[i];
                if(ic >= NCELL-NUMBUF) continue;
                fac = den[ic] * vol[ic] / dt_virt;
                sp1[ic] += fac;
                sp2[ic] += fac;
                sp3[ic] += fac;
        }
        if(IAMPRO < 2 && ITER == 1) print "set momentum sources in", nBlockedCells, "cells";
}

void BlockSolidCellsNew(
        double vol[],
        double den[],
        double sp1[],
        double sp2[],
        double sp3[],
        double su1[],
        double su2[],
        double su3[])
{
        int i; int ic;
        double fac;
        double dt_virt = DT * 1.0e-5;
        double alf;
        for(i = 0; i < nBlockedCells; i++){
                ic = blockedCell[i];
```

```
                if(ic >= NCELL-NUMBUF) continue;
                alf = 1.0 - blockedFrac[i];
                if(alf < 0.1){
                        fac = den[ic] * vol[ic] / dt_virt;
                        sp1[ic] += fac;
                        sp2[ic] += fac;
                        sp3[ic] += fac;
                        su1[ic] = 0.0;
                        su2[ic] = 0.0;
                        su3[ic] = 0.0;
                }else{
                        fac = 1.0 / alf; fac *= fac;
                        sp1[ic] *= fac;
                        sp2[ic] *= fac;
                        sp3[ic] *= fac;
                        su1[ic] *= alf;
                        su2[ic] *= alf;
                        su3[ic] *= alf;
                }
        }
        if(IAMPRO < 2 && ITER == 1) print "set momentum sources in", nBlockedCells, "cells";
}



void USEBOD(
        int iconv,
        int mph)
{
        BlockSolidCells(VOL, DEN, SP1, SP2, SP3);
        //BlockSolidCellsNew(VOL, DEN, SP1, SP2, SP3, SU1, SU2, SU3);
}
```

# 3. APPENDIX: MESH_DEFORMATION.FRML

```
$$init
double displacements[1][3];
int i; int j;
double d0; double d1; double d2;
double deltaZAct;
double x[3];
double F;
double gradF[3];
double s;
int nIters;
char saux[1];

double deltaZGap;
double tgAlfSeat;

double tol = 0.1;

resize(saux, 1024);

if(!GetUserDef("FIRE_TO_HYDSIM", saux)){
        sprintf(saux,
"F=%20.10g%20.10g%20.10g;M=%20.10g%20.10g;Q_seat=%20.10g;Q_holes=%20.10g;p_sac=%20.10g;p_gas=%
20.10g;p_topV=%20.10g;T_topV=%20.10g;",
                0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0);
        SetUserDef("FIRE_TO_HYDSIM", saux);
}

resize(nodeFlags, NVERT);
resize(displacements, NVERT);
if(rezone) displacements = 0.0;

// compute all nozzle parameters (needle cones, seat cone, cake angle)
ComputeNozzleGeometry();

// set needle displacements for nodes in "needle_tip",
// retract nodes to guarantee minimal flow gap
nodeFlags = 0;
FlagSelectedNodes("needle_tip", 1, nodeFlags);
tgAlfSeat = (rMaxSeat - rMinSeat) / (zMaxSeat - zMinSeat);
deltaZGap = minFlowGap * sqrt(1.0 + tgAlfSeat * tgAlfSeat) / tgAlfSeat;

// deltaZAct is set to actual needle lift (closed = 0.0)
if(IAMPRO<2){
```

```
        GetUserDef("HYDSIM_TO_FIRE", saux);
        print "hydsim values at", t, ":", saux;
        GetUserDef("endITC", saux);
        print "endITC:", saux;
}
deltaZAct = hydsimValue("d=", 0);
if(IAMPRO<2) print "deltaZAct:", deltaZAct;
// deltaZ is set to z-shift for the FIRE mesh (closed = deltaZRef)
deltaZ = deltaZAct + deltaZRef;
if(IAMPRO<2) print "deltaZ:", deltaZ;

// shift needle-cone z-values for later blocking-factor-computation
zMinTip1 += deltaZ; zMaxTip1 += deltaZ;
zMinTip2 += deltaZ; zMaxTip2 += deltaZ;
zMinTip3 += deltaZ; zMaxTip3 += deltaZ;

for(i = 0; i < NVERT; i++){
        if(nodeFlags[i]){
                GetVertexCoords(i, x);
                displacements[i] = x; // displacments[i] temporarily saves old node pos
                x += deltaZ * baseZ; // displaced position
                getConeFunc(zMinSeat+deltaZGap, zMaxSeat+deltaZGap, rMinSeat, rMaxSeat, x, F,
gradF);
                nIters = 3;
                while(nIters-- > 0 && F > 0.0){
                        s = F / (gradF . gradF);
                        x -= s * gradF;
                        getConeFunc(zMinSeat+deltaZGap, zMaxSeat+deltaZGap, rMinSeat, rMaxSeat,
x, F, gradF);
                }
                displacements[i] = x - displacements[i]; // final node displacement
        }
}

// flag interpolated nodes "0", all other "1"
nodeFlags = 1;
FlagSelectedNodes("inter", 0, nodeFlags);

if(IAMPRO<2) print "moving mesh...";
if(rezone) tol = 0.01;
if(deltaZ > 0.0)
        ComputeDisplacements(nodeFlags, displacements, "LaplaceArea", "LaplaceVolume", tol,
20);
else
        ComputeDisplacements(nodeFlags, displacements, "LaplaceEdgeLength",
"LaplaceEdgeLength", tol, 20);
for(i = 0; i < NVERT; i++){
        GetVertexCoords(i, x);
        x += displacements[i];
```

```
        SetVertexCoords(i, x);
}

if(IAMPRO<2) print "moving mesh done.";
printMeshQuality();

// computed blocked cells; to be applied in USEBOD
porFrac = 1.0 - deltaZAct / deltaZGap;
ComputeBlockedCellsSimple("PorosityPhase");

resize(saux, 1);



$$formula
// empty
```