Cite as: Aschmoneit, F.: A membraneFoam tutorial. In Proceedings of CFD with OpenSource Software, 2016, Edited by Nilsson. H., http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2016

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

Project work:

A membraneFoam tutorial

Developed for OpenFOAM-4.x

Author:
Fynn Aschmoneit
Danmarks Tekniske Universitet
fyna@env.dtu.dk

Peer reviewed by:
Phanindra Thummala
Håkan Nilsson

Disclaimer: This is a student project work, done as part of a course where OpenFOAM and some other OpenSource software are introduced to the students. Any reader should be aware that it might not be free of errors. Still, it might be useful for someone who would like learn some details similar to the ones presented in the report and in the accompanying files. The material has gone through a review process. The role of the reviewer is to go through the tutorial and make sure that it works, that it is possible to follow, and to some extent correct the writing. The reviewer has no responsibility for the contents.

February 14, 2017

Contents

Chapter 1

Learning outcomes

- The application of *membraneFoam* for the simulation of osmotic pumping through a semi-permeable membrane.
- Understanding of the essential features of a baffle boundary, such as face mapping and the implementation of non-analytic functions on it.
- Fixing specific problems when updating an OpenFOAM implementation to version 4.0.

Chapter 2

Introduction

The code package *membraneFoam* [1] is consisting of solvers, boundary conditions and utilities related to different membrane processes and their analysis. This project concentrates on the modification of the *FO_BC* boundary condition and the two solvers *potentialSalt* and *simpleSaltTransport* . The modified computation scheme is used for the optimization of spiral-wound membrane modules. There is yet no complete computational model resolving the whole spiral-wound module with a highly resolved flow- and pressure distribution, [2].

The spiral-wound module is basically a number of membrane envelopes enclosing the permeate channel rolled around a central permeate tube and separated by feed spacers, creating the spacer channel. This geometry provides a large membrane surface while keeping the module size compact. These modules are optimized with respect to several parameters:

- Number of sheets: For a given module diameter, one could apply few long envelope sheets or more short sheets. This will not change the membrane surface area but the longer an envelope sheet is, the higher is the pressure drop along that sheet and as the pressure drop in these sheets is not proportional with their lengths but higher, the ideal number of sheets for a given application method (applied pressure, inlet water flow, draw solution concentration,...) is a complex optimization task.
- Feed- and draw spacer design: The spacers keep the membranes from sticking to each other and also promote some turbulence inside the channels which will mix the solution better improving the membrane performance. See section on the theoretical background for performance impeding concentration polarization.

The *membraneFoam* package is described and validated in [1]. The sources are explained in chapter 4 and its underlying theory is explained in chapter 3. There are three major modification to *membraneFoam* described in this project:

- The solvers don't run in *OpenFOAM 4* .
- No hydraulic pressure difference across the membrane is supported but for the number of envelopes optimization, a constant pressure on both membrane sides can't be assumed.
- The simulation of bent membranes doesn't work: The package was developed for a flat sheet module.

In chapters 5, 6 and 7 the modifications to the *membraneFoam* package solving above problems are described. Finally, in chapter 8 a tutorial to the usage of *membraneFoam* is given.

The original source code is found in this Git Repository:

`https://github.com/MathiasGruber/MembraneFoam`

The modifications to it are found in the same repository in the *OF4* branch and its installation is outlined in chapter 8. Alternatively, the sources can be found in the accompanying *membraneFoam.tar* folder. It's file structure is presented through the following file tree:

```
MembraneFoam
├── membraneFoamExampleCase
│   ├── 0
│   ├── constant
│   ├── initialFields
│   └── system
└── src
    ├── boundaryConditions
    │   ├── FO_BC
    │   │   ├── explicitFOmembraneSolute
    │   │   │   ├── explicitFOmembraneSoluteFvPatchScalarField.C
    │   │   │   └── explicitFOmembraneSoluteFvPatchScalarField.H
    │   │   └── explicitFOmembraneVelocity
    │   │       ├── explicitFOmembraneVelocityFvPatchVectorField.C
    │   │       └── explicitFOmembraneVelocityFvPatchVectorField.H
    │   └── ...
    ├── solvers
    │   ├── potentialSalt
    │   │   ├── Make
    │   │   ├── createFields.H
    │   │   ├── potentialSalt.C
    │   │   └── readControls.H
    │   ├── simpleSaltTransport
    │   │   ├── Make
    │   │   ├── createFields.H
    │   │   ├── m_AEqn.H
    │   │   ├── pEqn.H
    │   │   ├── simpleSaltTransport.C
    │   │   └── UEqn.H
    │   └── ...
    └── ...
```

Chapter 3

Theoretical background

The forward osmosis process

When bringing two fluids of different concentration of some solute in contact the solute will be transported by diffusion and the concentration averages throughout the fluid. From a thermodynamic point-of-view the diffusion happens because an averaged solute concentration is energetically more favorable. But that implies that before the two fluids are brought in contact a potential energy depending on their concentration can be assigned to them. This potential energy is known as the osmotic potential.

If the two fluids are not directly in contact but separated by a membrane that is only permeable to water ("semi-permeable") the solute can't permeate through the membrane so the concentrations can only adjust if the water permeated through the membrane towards the higher concentrated fluid. This process is known as osmosis or forward osmosis (FO), see figure 3.1 (left).

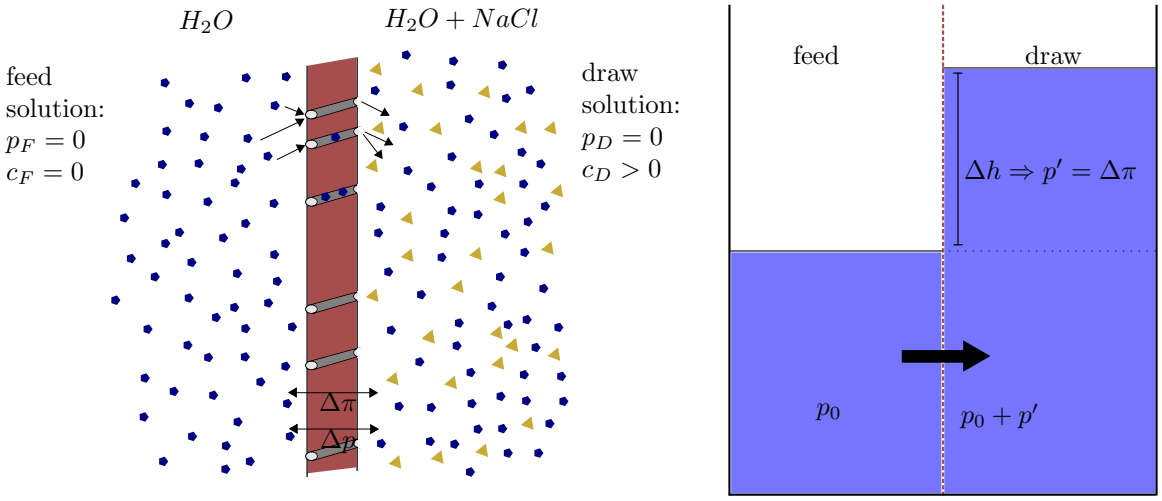

Figure 3.1: left: FO process through a semi-permeable membrane. Water molecules are drawn through the membrane channels towards the draw solution side, causing dilution of the salt concentration is the membrane vicinity.
right: in steady-state, the increased hydraulic pressure is equal to the osmotic pressure difference.

The water flux through the membrane is thus depending on the difference of the osmotic potential. It is common to relate the osmotic potential to an osmotic pressure such that the driving force per membrane area is given by the osmotic pressure difference. The osmotic pressure can be calculated from the solute concentration but here it is illustrated through the following set-up: Two open water

reservoirs, one called the "feed solution" and one called the "draw solution", are separated through a semi-permeable membrane. The feed solution is pure water and the draw solution is salt water with a $NaCl$ concentration of c_D. It is further assumed that the hydraulic pressure is initially same in both reservoirs. The difference in osmotic pressure forces the water through the membrane. This has two effects on the draw solution: First, the draw solution is diluted reducing its osmotic pressure. Secondly, due to the permeated water the water level in the draw reservoir rises, increasing the hydraulic pressure in that reservoir. Both effects effectively reduce the water flux and the system will eventually have reached a steady-state, see figure 3.1 (right). The additional hydraulic pressure due to the increased water level p' is then equal to the difference of the osmotic pressure in both reservoirs.

$$\pi_D - \pi_F = \Delta\pi = p' \tag{3.1}$$

The water flux across the membrane can now be defines as:

$$\mathbf{J}_W = A[\pi_D - \pi_F - (p_D - p_F)]\,\mathbf{n}_D = A[\Delta\pi - \Delta p]\,\mathbf{n}_D, \tag{3.2}$$

where A is the pure water permeability and $\mathbf{n}_D$ is the surface normal direction in the draw solution. Due to imperfections of the membranes, salt permeates through the membrane in the opposite direction. This unwanted effect is called reverse salt flux and it can be quantified through:

$$\mathbf{J}_S = B[c_D - c_F]\,(-\mathbf{n}_D) = -B\Delta c\,\mathbf{n}_D, \tag{3.3}$$

where B is the slat permeation coefficient.

Incomplete mixing and concentration polarization

The pure water flux dilutes the draw solution primarily in the vicinity of the membrane. The solute concentration is thus a local quantity at some $\mathbf{x}$:

$$c_{F,D} \longrightarrow c_{F,D}(\mathbf{x}) \tag{3.4}$$

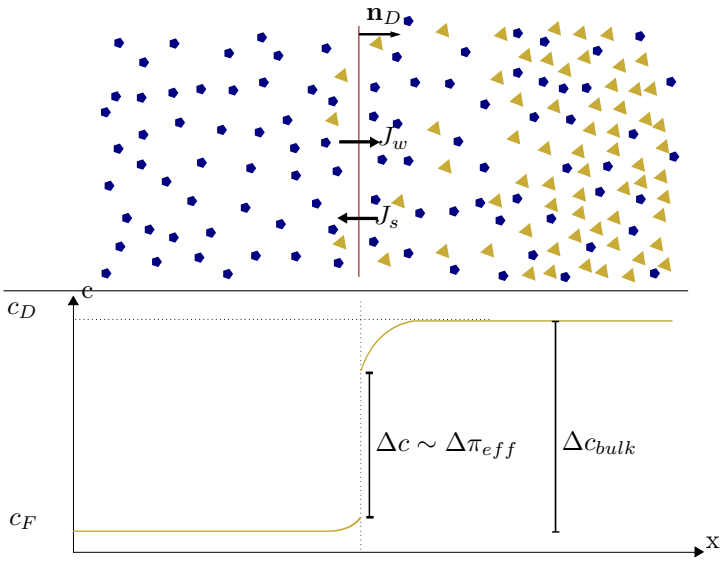

Figure 3.2: External concentration polarization (ECP) reduces the osmotic pressure difference and therefore the water flux, due to incomplete mixing. The density of yellow particles indicate the $NaCl$ concentration and the blue particles are H_2O. The thick arrow pointing to the right is the water flux and the arrow pointing to the left is the reverse salt flux.

As indicated in figure 3.2, the incomplete mixing on the draw side causes an inhomogeneous concentration distribution: The concentrations at the membrane surfaces differ from the bulk concentrations c_F, c_D. On top of that, due to imperfection of the membranes, salt leaks from the draw solution to the feed solution, which decreases the concentration difference across the membrane. These effects are called external concentration polarization (ECP). As the osmotic pressure is linearly dependent on the solute concentration the effective osmotic pressure difference will be smaller than the bulk osmotic pressure difference. This reduces the water flux and thus the performance of the FO process.

Typical FO membranes are a little more complicated than figure 3.1 suggests. They consist of an active layer ($d \approx 150nm$) and a support layer ($d \approx 150\mu m$), see figure 3.3. The solute rejection and the pure water transport take place at the thin active layer, whereas the thick support layer is holding the active layer in place and takes on the mechanical forces on it. It is made of a very porous material such that water and solute can be transported through it.

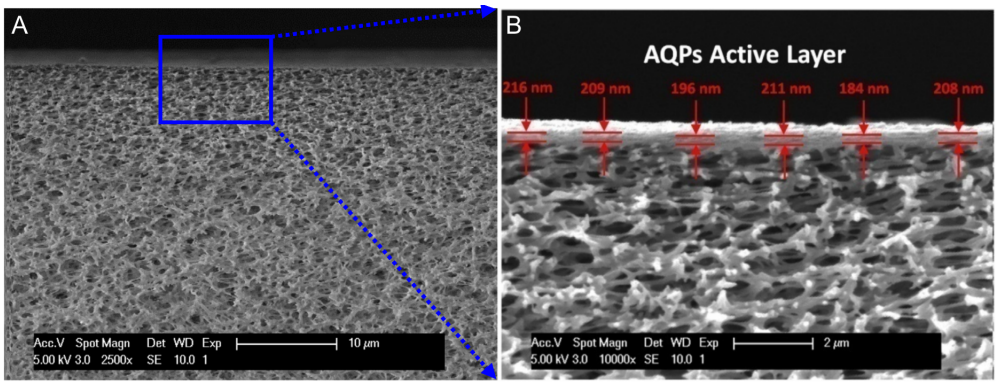

Figure 3.3: Electron microscope picture of membrane with porous support and active layer. With permission from [3].

The porous support layer is usually facing the draw solution in FO applications. The incomplete solute mixing in the support layer causes internal concentration polarization (ICP) in the support reducing the effective osmotic pressure difference further, see figure 3.4.

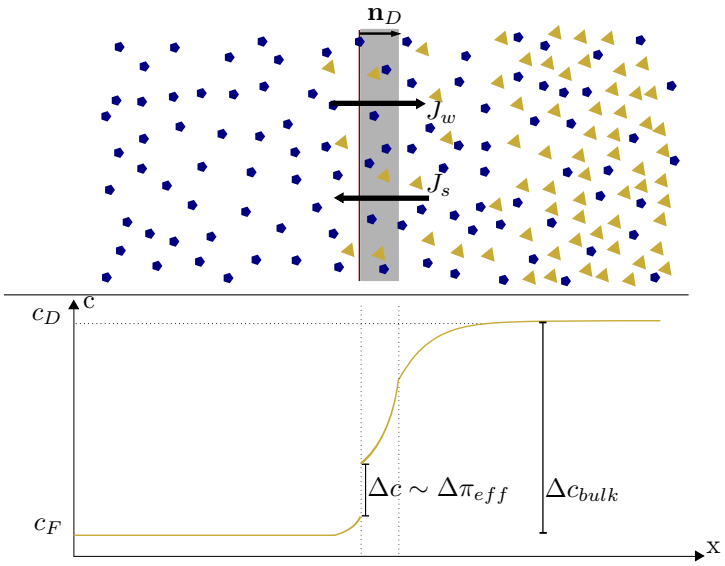

Figure 3.4: ICP and ECP through incomplete mixing: The red line resembles the active layer, the gray block is the porous support layer. The density of yellow particles indicate the $NaCl$ concentration and the blue particles are H_2O. The thick arrow pointing to the right is the water flux and the arrow pointing to the left is the reverse salt flux. The difference of $NaCL$ concentration across the membrane deviated from the bulk concentration difference, due to the combined effects of ICP and ECP.

Both ECP and ICP are severely impeding the water transport performance of the membrane. ICP can only be reduced through thinner support layers and current research targets at finding new materials that provide the same mechanical strength and water permeability while reducing the supports' thickness. The earlier mentioned practice of solute mixing through generating turbulence at the support layer surface is reducing the ECP. This practice is commonly applied by inserting a spacer grid but through the friction with the grid the pressure drop along the module becomes larger. The project of feed- and draw spacer design optimization is concerned with the optimal trade-off of solute mixing and pressure drop.

Chapter 4

Original membraneFoam sources

The *membraneFoam* package consists of a set of solvers and boundary conditions for different membrane flow applications. The present tutorial focuses on the modification of the following boundary condition and solvers:

- solver *simpleSaltTransport*
- solver *potentialSalt*
- boundary conditions *FO_BC*

The *simpleSaltTransport* solver couples the steady-state flow solver *simpleFoam* with the concentration transport solver *scalarTranportFoam* . The solver is therefore capable of simulating flow and the transport of salt concentration along the velocity field. The turbulence modelling was disabled for *simpleSaltTransport* , as in membrane flow the Reynolds Number is typically smaller than 100. Turbulence related property dictionaries don't need to be included.

The *potentialSalt* solver generates a velocity and pressure field, just like *potentialFoam* but it includes the concentration field *m_A* in its *createFields.H* source, which allows the membrane boundary condition to calculate the boundary velocity based on the concentration distribution.

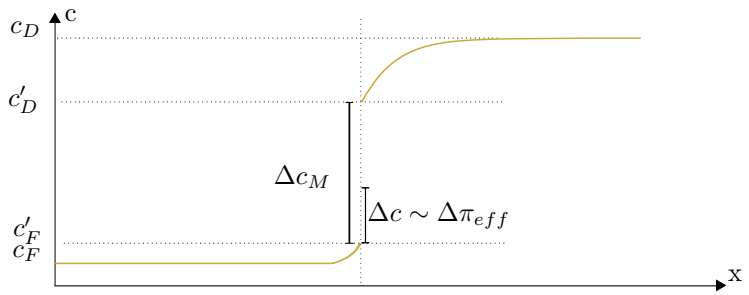

Figure 4.1: the exponential concentration increase can be modeled. thus the boundary models the ICP.

The *FO_BC* boundary conditions are the key sources of the *membraneFoam* package. It consists of two boundary conditions, one for the velocity field and one for the salt concentration field, namely *explicitFOmembraneVelocity* and *explicitFOmembraneSolute* . The boundary conditions basically solve for the correct water- and solute flux at membrane baffle. But since it is numerically not feasible to resolve the thin support layer a theoretical model is used that models the ICP in the support layer. So, in practice the whole membrane, consisting of active layer and support layer, is simulated as an

infinitely thin sheet. Figure 4.1 illustrates the situation: Due to ECP the concentration difference across the membrane is Δc_M, but the boundary conditions models the ICP resulting in the true concentration difference Δc_{eff}: The physical boundary condition at the membrane can be found in equation 4.1, which states that the difference of incoming solute ($\mathbf{J}_w c$) and back transported solute ($\mathbf{J}_s$) is given by the diffusive transport within the porous support.

$$\mathbf{J}_w c - D_{eff} \frac{d}{dx} c \, \mathbf{n}_D = \mathbf{J}_s \tag{4.1}$$

Equation 4.2 is the solution for the water transport $\mathbf{J}_w$ which only depends on membrane parameters and on the concentration c'_D and c'_F.

$$\mathbf{J}_w = \frac{1}{K} ln \frac{B + A\pi'_D}{B + |\mathbf{J}_w| + A\pi'_F} \mathbf{n}_D \tag{4.2}$$

As equation 4.2 has no analytic solution, it needs to iterated. This is basically what the boundary condition *explicitFOmembraneVelocity* is doing, for every face on that membrane boundary. With an approximated value for the water flux $\mathbf{J}_w$, the differential equation 4.1 can be solved for the solute concentration at the membrane. This is done through the boundary condition *explicitFOmembraneSolute* .

Chapter 5

OpenFOAM 4.x adjustments

In the following to the code in the *OF4* branch of the git repository or as found in the accompanying files is referred to. For comparing with the pre- OpenFoam 4 version, the respective files should be compared with the *master* branch of the same repository.

potentialSalt

In the wmake options file of potentialSalt the inclusion of the *meshTools* library was added. This modification alone let's potentialSalt compile fine.

simpleSaltTransport

The implementation *simpleSaltTransport* is based on the *simpleFoam* algorithm but since the solver is not set up to do turbulent modeling the inclusion of the respective header files

```
-I$(LIB_SRC)/turbulenceModels
-I$(LIB_SRC)/turbulenceModels/compressible/RAS/RASModel
```

and the respective library files

```
-lcompressibleTurbulenceModel
-lcompressibleRASModels
```

is superfluous and the files are excluded from the compilation options in the Make/options file.

As described in the previous chapter *simpleSaltTransport* includes *UEqn.H* , *m_AEqn.H* and *pEqn.H* in the time loop. The *m_AEqn.H* source works fine in OpenFoam 4. The other two files need to be modified for the use in OpenFoam 4.

In the velocity file *UEqn.H* the linear matrix equation to be solved is initialized as

```
tmp<fvVectorMatrix> tUEqn,
```

where the matrix algebra class *fvVectorMatrix* is used for the initialization of an instance of the *tmp* class. The *tmp* class is a template class which is described as a wrapper for large objects allowing its object to be returned from a function without copying it or to quickly clear the object from the memory. With *OpenFOAM 4* referencing *tmp* class members depends on whether the member is defined as constant or non-constant. In the example above the *tmp* class instance is called *tUEqn* , which is initialized with the type *fvVectorMatrix* . To access the member functions or variables of *fvVectorMatrix* (to unwrap the wrapper) the *()* operator was used in the *tmp* object: *tUEeqn().relax()* unwraps *tUEqn* and applies *fvMatrix'* member function *relax()* to it. Now with *OpenFOAM 4* the reference operator *()* can only be applied to constant objects, which the *fvVectorMatric* object in *UEqn.H* is not. Instead, a new function *ref()* is used for the reference of non-constant object.

For the modification in *UEqn.H* this means that the wrapped object in *tmp* needs to be referenced by:

```
fvVectorMatrix& UEqn = tUEqn.ref();
```

where *tUEqn* is the *tmp* object that wraps the matrix equation. The *fvVectorMatrix* member function *relax()* can now directly be applied to *UEqn* and *UEqn* can be used for later matrix operations in the same time step.

```
UEqn.relax();
```

Following the velocity fields calculation in *UEqn.H* comes the *m_AEqn.H* source, which is independent of the velocity field and thus doesn't need modifications. Finally, the iteration loop calculates the pressure field in *pEqn.H* . Here, the unwrapped *fvVectorMatrix* object *UEqn* is still defined and thus it can be used directly without prior unwrapping as in earlier versions. So it is sufficient to change *UEqn()* to *UEqn* in *pEqn.H* .

```
UEqn()    ------>    UEqn
```

The source modification for OpenFoam 4 is herewith finished, as the boundary conditions *FO_BC* works fine in OpenFoam 4.

Chapter 6

Non-zero hydraulic pressure difference

In the derivation of equation 4.2 it was assumed that the hydraulic pressure difference across the membrane was zero. That is a commonly applied assumption as the hydraulic pressure difference is usually a lot smaller than the osmotic pressure difference, making its contribution to the water flux insignificant. However, when simulating a membrane module in which the draw channel is approximately three times longer than the feed channel and in which spacer grids of different mesh sizes are inserted, the hydraulic pressure difference across the membrane might be significant.

The governing equation 4.1 has a different solution from 4.2, if the hydraulic pressure difference in 3.2 is different from zero:

$$\mathbf{J}_w = A \left[\frac{\pi_F |\mathbf{J}_W| \left[\frac{c_D}{c_F} - exp(|\mathbf{J}_w| K) \right]}{(|\mathbf{J}_w| + B)\, exp(|\mathbf{J}_w| K) - B} - \Delta p \right] \mathbf{n}_D \quad , \quad K = \frac{l_{eff}}{D_{eff}} \tag{6.1}$$

Just like in the case of equation 4.2, this equation is solved numerically at every membrane boundary face. In the boundary condition *explicitFOmembraneVelocity* , the solution to equation 6.1 is found through the Ridder root finding algorithm, in the function *ridderSolve(...)* . This function calls the function *fluxEquation(...)* , in which the the function to be solved is defined. In *fluxEquation(...)* the entry for the key word *eq* is looked up. As defined, *eq* is set to either *advanced* , in which case equation 4.2 is defined in *fluxEquation(...)* , or *simple* , in which case a simplified equation not allowing for reverse salt flux ($B = 0$) is defined. In the same manner as the other flux equation options are implemented, a new option *advancedPresDiff* is added, which defines equation 6.1 as the function to be solved in *ridderSolve(...)* :

```
1  Foam::scalar Foam::explicitFOmembraneVelocityFvPatchVectorField::fluxEquation(
       const scalar& Jvalue, const scalar& feedm_A, const scalar& drawm_A, const
       scalar& feedP, const scalar& drawP )
2  {
3
4      if( fluxEqName_ == "simple" || B() < SMALL )
5      {
6          /*- Implicit flux equation,
7          Valid when B is low compared to other terms, i.e. high rejection.
8          See "Modelling Water Flux in Forward Osmosis: Implications for Improved
                Membrane Design
9          American institude of Chemical Engineers, vol 53, No. 7, p. 1736-1744
10         */
11         return Jvalue - A()*( pi_mACoeff().value() * ( drawm_A*exp(-Jvalue* K()
               ) - feedm_A ) );
12     }
13     else if( fluxEqName_ == "advanced" )
14     {
```

```
15          /*- Implicit flux equation
16          Valid at any B-value.
17          See "Coupled effects of internal concentration polarization and fouling
                 on flux
18              behaviors of forward osmosis membranes during humic acid
                    filtration"
19          Journal of Membrane Science 354 (2010) 123-133
20          */
21          scalar numerator    = A()*pi_mACoeff().value()*drawm_A + B();
22          scalar denominator  = A()*pi_mACoeff().value()*feedm_A + Jvalue + B();
23
24          // To avoid floating point exceptions
25          if( denominator > SMALL ){
26              return Jvalue - ( 1/K() ) * log( numerator / denominator );
27          }
28          else{
29              return 0;
30          }
31      }
32      else if( fluxEqName_ == "advancedPresDiff" )
33      {
34          /*- Implicit flux equation
35          Like "advanced" flux equation but assuming a non-zero hydraulic
                pressure difference across the membrane.
36          */
37
38          scalar expJK = exp( Jvalue*K() );
39          scalar dP = drawP - feedP;
40          scalar numerator = pi_mACoeff().value()*feedm_A * Jvalue * ( drawm_A/
                feedm_A - expJK );
41          scalar denominator = ( Jvalue + B() ) * expJK - B();
42
43          // To avoid floating point exceptions
44          if( denominator > SMALL ){
45              return Jvalue -  A()  * ( numerator/denominator - dP );
46          }
47          else{
48              return 0;
49          }
50      }
51      else
52      {
53          FatalErrorIn
54          (
55              "In the file: explicitFOmembraneVelocity.C"
56          ) << "No flux model was selected. Select either of the following models
                 in 0/U, eq = {simple, advanced, advancedPresDiff}. " << abort(
                FatalError);
57      }
58      return 0;
59  }
```

Running a case, the algorithm crashes when *fluxEquation(...)* is called. The reason for that was found in the function *riddeSolve(...)* : The root finding algorithm first sets lower- and upper limits for the flux equation by solving it with a minimum- and maximum value (*minBound* and *maxBound* , defined in function *updateCoeffs()*) for the water flux $\mathbf{J}_w$.

```
1  Foam::scalar Foam::explicitFOmembraneVelocityFvPatchVectorField::ridderSolve(
       const scalar& feedMemSol,
2  const scalar& drawMemSol, const scalar& feedMemPres, const scalar& drawMemPres,
        const scalar& minBound, const scalar& maxBound, const scalar& xacc, scalar
       & i )
3  {
4      // Function of boundaries
5      scalar fl = fluxEquation( minBound , feedMemSol , drawMemSol , feedMemPres
           , drawMemPres );
```

```
6       scalar fh = fluxEquation( maxBound , feedMemSol , drawMemSol , feedMemPres
            , drawMemPres );
```

Both *minBound* and *maxBound* need to be changed for solving equation 6.1 Originally, *minBond* was set to zero. But if $|\mathbf{J}_w|$ was zero in equation 6.1 the denominator is zero causing an floating point exception error. *minBound* was thus set to 10^{-10}.

In the original implementation *maxBound* was set to 10^{-1}, but since $K = \mathcal{O}(10^5)$, the exponential functions in 6.1 can't be calculated because their values can't be expressed as double precision floating point numbers, causing floating point exception errors. The maximum flux therefore needed to be reduced to $maxBound = 10^{-3}$, which is fine for the flux investigations at hand. The *ridderSolve(...)* algorithm can now successfully calculate the water flux through the membrane with a non-zero hydraulic pressure difference.

Since the solute flux $\mathbf{J}_s$ can be expressed only through membrane parameters and the water flux $\mathbf{J}_w$, no changed are needed for the *explicitFOmembraneSolute* boundary.

Chapter 7

Automatic recognition of membrane feed side

Because the solver is to be applied to bent membrane surfaces the surface normal directions cannot be assumed to be parallel. The *0/U* file sets the initial values for the velocity field. In the section that defines the membrane boundary condition, a forward direction must be specified. This is the flux forward direction pointing towards the draw solution, as indicated ad n_D in equation 3.2. But since the membrane is bent a uniform n_D doesn't make sense.

Looking in the source code of that velocity boundary, *explicitFOmembraneVelocityFvPatchVectorField.C* , an algorithm for the automatic detection of the forward direction is found in the function *initialise()* , which is explained in the following. The following snippet is extracted from the *checkMesh* output of the test case. It is important to note that the mesh consists of two regions, see line 7, that were stitched together along the patch *Membrane* , see line 24.

```
1  Checking topology...
2      Boundary definition OK.
3      Cell to face addressing OK.
4      Point usage OK.
5      Upper triangular ordering OK.
6      Face vertices OK.
7     *Number of regions: 2
8      The mesh has multiple regions which are not connected by any face.
9    <<Writing region information to "0/cellToRegion"
10   <<Writing region 0 with 6932 cells to cellSet region0
11   <<Writing region 1 with 7683 cells to cellSet region1
12
13 Checking patch topology for multiply connected surfaces...
14                      Patch    Faces   Points                  Surface topology
15                       Wall      370      744  ok (non-closed singly connected)
16                  OutletFeed      111      224  ok (non-closed singly connected)
17                   InletFeed      125      252  ok (non-closed singly connected)
18                         PB1     1247     2366  ok (non-closed singly connected)
19                         PB2     1247     2366  ok (non-closed singly connected)
20                  InletDraw      196      394  ok (non-closed singly connected)
21                     Inside        0        0                        ok (empty)
22                  OutletDraw      197      396  ok (non-closed singly connected)
23                    Inside_2        0        0                        ok (empty)
24                    Membrane     5482     5484  multiply connected (shared edge)
25   <<Writing 5480 conflicting points to set nonManifoldPoints
```

The *Membrane* patch has $N = 5482$ faces but since it is an internal boundary (a baffle) it has half that number of faces on either side of the patch (conformal patches are imposed). An algorithm compares the solute concentration of opposing sides of the membrane faces and thus detects the feed side. For that, for some face with index i the index of the opposing face j must be known, see figure 7.1 and source snippet from *explicitFOmembraneVelocityFvPatchVectorField.C* .

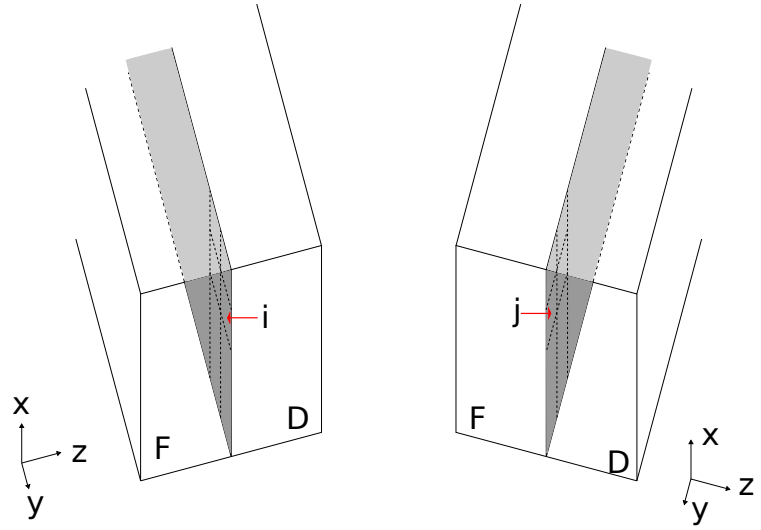

Figure 7.1: Membrane face mapping: internal boundaries have faces on both sides. Face i must be mapped to face j for later comparison of solute concentration on these faces. D and F stand for draw- and feed channel.

```
1  void Foam::explicitFOmembraneVelocityFvPatchVectorField::initialise()
2  {
3      calcFaceMapping();
4
5      // fill out the fs_ list so that it contains the indices of all the "feed-
           side" faces
6      if (mag(forwardDirection_) < VSMALL)
7      {
8          Info << patch().name() << ": forward direction specified by mass
               fraction\n" << endl;
9
10         // the forward direction of the membrane has not been defined by the
               user
11         // so the mass fraction will be used to determine the feed side
12         const fvPatchScalarField& m_Apsf = patch().lookupPatchField<
               volScalarField, scalar>(m_AName_);
13         tmp<scalarField> tm_Asf = m_Apsf.patchInternalField();
14         const scalarField& m_Asf = tm_Asf();
15         forAll(fs_, facei)
16         {
17             fs_[facei] = (m_Asf[facei] > m_Asf[fm_[facei]]) ? fm_[facei] :
                   facei;
18         }
19     }
20     else
21     {
22         Info << patch().name() << ": forward direction specified by user\n" <<
               endl;
23
24         tmp<vectorField> tvfnf = patch().nf();
25         const vectorField& vfnf = tvfnf();
26         forAll(fs_, facei)
27         {
28             fs_[facei] = ((vfnf[facei] & forwardDirection_) < 0.0) ? fm_[facei]
                    : facei;
29         }
30     }
31 }
```

This face mapping is done through the function *calcFaceMappping()* which is called first in *initialise()* , see line 3. Given a list wit all face indices *calcFaceMapping()* creates a list *fm_* with the respective opposing indices:

$$fm\_[i] = j$$

$$fm\_[j] = i$$

With this face indexing list the algorithm for the membrane feed side detection can be explained: It first tests if the modules of the supplied forward direction vector is smaller that $VSMALL = 10^{-37}$, see line 6, and if that is the case it defines a *scalarField m_Asf* with the solute concentration values of all N faces (both sides). The feed side index list *fs_* is a *scalarField* of length $\frac{N}{2}$ which only contains the indices of the face feed sides through the code in line 17: The solute concentration on the feed side face is smaller than on its opposing draw side, so the feed side index list *fs_* at index *facei* is assigned to *fm_[facei]* if the concentration is higher at *facei* that at *fm_[facei]* .

If the concentration at *facei* is smaller than that at *fm_[facei]* , *fs_[facei]* is set to *facei* , see line 17. The list *fs_* is thus containing the face indices of the $\frac{N}{2}$ feed side faces.

If a forward direction vector of some modulus greater than *VSMALL* is specified (usually you would specify unit vectors), the feed side index list is simply defined through the scalar product of the forward direction vector and the face normal direction (which is defined to be directed outwards at boundaries), see line 28.

With the membrane feed side recognition algorithm in the function *initialise()* the automatic definition of the flux forward direction for bent membranes should work, when the forward direction vector is given as *(0 0 0)* . It is assumes that setFields was used to define the bulk solute concentration distribution in the two regions, such that *initialise()* can actually detect a difference in solute concentration on opposing faces, see the initial state of the test case in figure 8.2.

But unfortunately the program has a bug: running *potentialSalt* crashes with the error message:

```
1  --> FOAM FATAL ERROR:
2
3      request for volScalarField m_A from objectRegistry region0 failed
4      available objects of type volScalarField are
5  1(p)
```

The error occurs in the include of *createFields.H* in *potentialSalt.C* . More precisely it occurs when the velocity field is defined from the *0/U* file. The definition of the velocity field also includes the definition of its boundaries, using the *initialise()* function described above. As discussed, it needs the solute concentration distribution *m_A* at the membrane surface, should the flux forward direction not explicitly be defined.

The error message basically says that a *volScalarField m_A* is requested but not defined. The only *volScalarField* known is that of the pressure *p* . With that the bug is easily found: in the original source the *volScalarField* object *m_A* is defined after the *volVectorField U* (the velocity field) in *createFields.H* . Only moving the definition of *m_A* above that of *U* in *createFields.H* fixes the problem and *potentialSalt* can now be applied to bent membrane boundaries.

Presently, *paraFoam* still crashes when it is called on the simulated fields but it works fine when it is called with the *-builtin* options.

Chapter 8

Application tutorial

For the download of the sources it is necessary to have *git* installed. The installation of the sources and the tutorial case can then be done through copying the following lines. The example case is a low resolution test application of a bent membrane geometry. A high resolution example of a plane membrane can be found in the master branch of the git repository. The high resolution example takes considerably more time to converge and does not test bent membranes. The following chapters explain the set up of the test application.

- Source installation:

```
mkdir $WM_PROJECT_USER_DIR/src
cd $WM_PROJECT_USER_DIR/src
git clone git://github.com/MathiasGruber/MembraneFoam.git
cd MembraneFoam
git checkout OF4
./Allwmake
```

- Copying and running example case:

```
cp -r $WM_PROJECT_USER_DIR/src/MembraneFoam/membraneFoamExampleCase $FOAM_RUN
cd $FOAM_RUN/membraneFoamExampleCase
mkdir 0
cp initialFields/* 0/
setFields
potentialSalt
simpleSaltTransport
```

Geometry and Mesh design

The meshing is a crucial part of the *membraneFoam* application: As described above, two regions connected through the membrane surface need to be defined. The two respective meshes need to be confined at this surface, so that the face mapping algorithm can connect opposing faces. The provided mesh was generated in *STAR-CCM+* as an unstructured, confined mesh. It was then conversed to an *OpenFOAM* mesh though the *ccm26toFoam* utility, which generated the polymesh folder. The membrane boundary condition is then defined through the *createBaffle* utility, which modifies the polymesh files in such a way as you find them in the example case.

Figure 8.1 shows a cut of the example mesh: The resolution along the membrane surface is increased. But this mesh serves only testing- or illustration purposes, as the resolution along the no-slip boundaries at the top and bottom is very coarse and the sharp edges on the bottom lead to numerical defects. The resolution along the membrane is also too coarse. This is indicated by the

fluctuating values for the water- and salt flux in the simulation output. Nevertheless, the mesh suits the purpose of testing source modification of this tutorial.

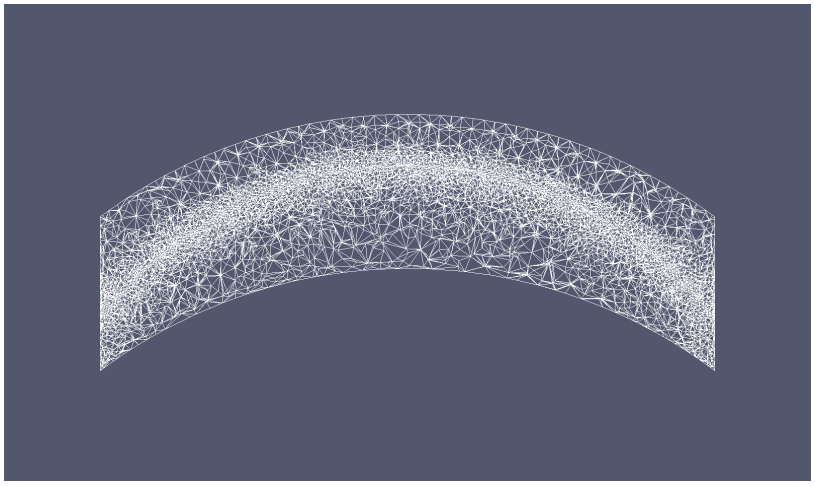

Figure 8.1: Cut through the mesh for the test case: increased resolution along the membrane surface

Case set up and simulation

When the simulation is run it first defines the flux forward direction. For that, it is crucial that the bulk values for the concentration are defined beforehand. This is done through the *setFields* utility. It sets the concentration value in the draw channel to $6.5 \cdot 10^{-2}$, see figure 8.2.

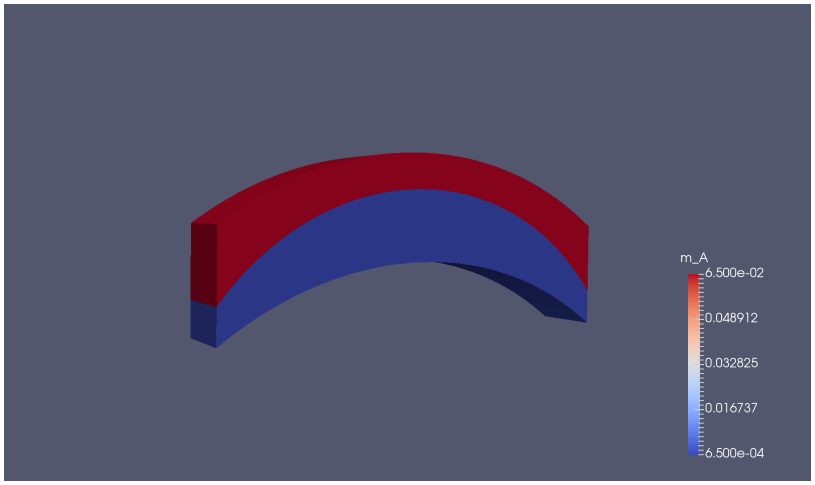

Figure 8.2: state at t=0: setFields was used to define initial bulk values for m_A in the two regions. The draw solution is red and the feed solution is blue. The two domains are flow channels which supply the respective domain with the respective bulk concentration value. The draw inlet is on the left-hand side, the feed inlet is on the right-hand side. The configuration is called a cross flow set up.

The *potentialSalt* solver is then used to develop an initial velocity- and pressure field. For solving the flux equations at the membrane boundary, the dictionary *transportProperties* in the *constant* folder is looked up. It is here, where all relevant membrane- and flow parameters are stored. The files in the 0 folder are overwritten, which is the reason why the initial fields are kept in the *initialFields* folder and always copied from there.

Finally the case is set up and the water- salt flux through the membrane can be simulated with *simpleSaltTranport* .

Qualitative analysis

As previously described, *paraFoam* needs to be called with the *builtin* option. From the concentration distribution in figure 8.3, the dilution of the draw solution clearly be seen. The concentration of salt in the feed channel grew slightly, due to the reverse salt flux.

Comparing the inlet and outlet velocity of the draw channel in figure 8.5, a higher velocity at the outlet (right-hand side of channel) indicates that more water flows out of the channel than it flows in. The surplus water is the permeated water through the membrane. It is balanced by a the decreasing water flow in the feed channel, where more water flows in than out.

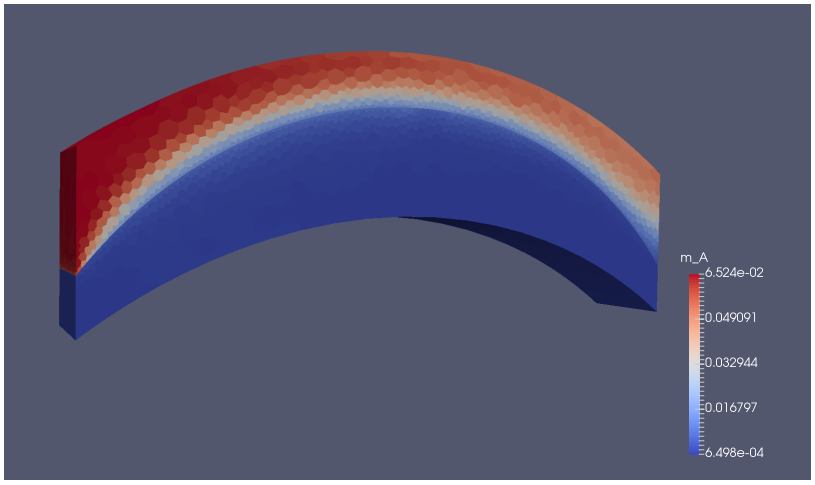

Figure 8.3: concentration distribution after 300 iteration steps: the permeation of water through the membrane diluted the solute concentration on the draw side. Some solute was transported through the reverse salt flux to the feed side.

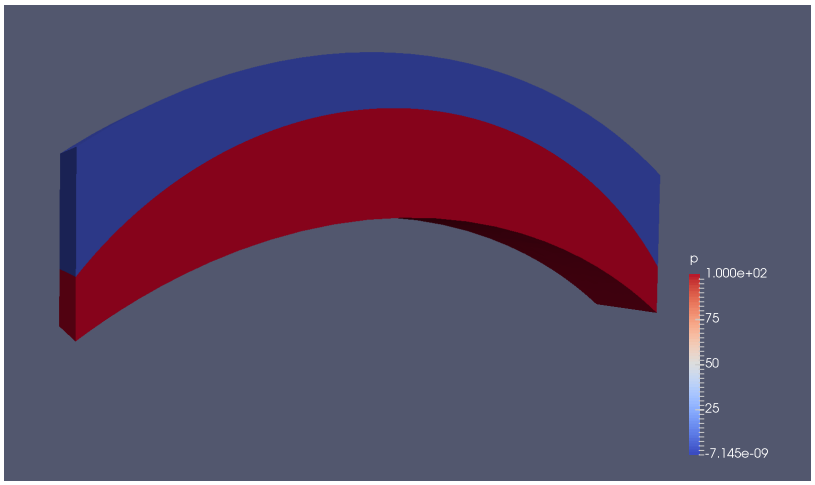

Figure 8.4: hydraulic pressure difference across the membrane: A difference of only 100 Pa has only a minor effect of the flux.

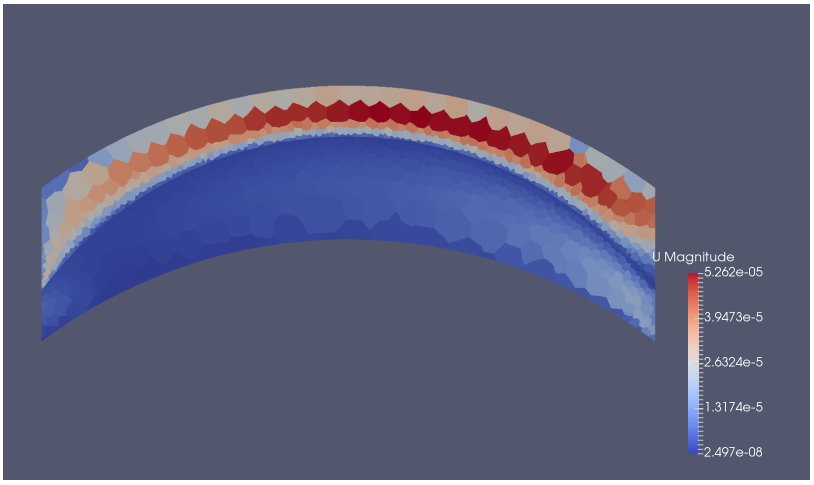

Figure 8.5: Cut plane showing the velocity magnitude distribution. More flux at draw channel outlet compared to its inlet. The resolution is poor but it suits the illustrative purpose.

Bibliography

[1] M. Gruber et al. "Computational fluid dynamics simulations of flow and concentration polarization in forward osmosis membrane systems". In: *Journal of Membrane Science* 379 (2011).

[2] G. Blandin et al. "Effeciently Combining Water Reuse ans Desalination through Forward Osmosis-Reverse Osmosis (FO-RO) Hybrids: A critical review". In: *Membranes* 37 (2016).

[3] W. Ye, H.T. Madsen, and E.G. Soegaard. "Enhanced performance of a biomimetic membrane for Na 2 Co 3 crystallization in the scenario of CO 2 capture". In: *Journal of Membrane Science* 498 (2016).

Study questions *membraneFoam*

1. Let *tElement* be defined as a wrapped *fvVectorMatrix* element:

```
tmp<fvVectorMatrix> tElement
```

Since OpenFAOM 4 there is a difference in the unwrapping operations *tElement()* and *tElement.ref()* . Explain that difference.
2. Why is it imperative that the meshes separated by the membrane are conformal?
3. In the test case, what explains the flow velocity difference of the inlet and outlet of the draw channel?