Cite as: Grimler, H.: An openFuelCell tutorial. In Proceedings of CFD with OpenSource Software, 2017,
Edited by Nilsson, H., http://dx.doi.org/10.17196/OS_CFD#YEAR_2017

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

An openFuelCell tutorial

Developed for OpenFOAM-v1706

Author:
Henrik Grimler
University of KTH Royal
Institute of Technology
hgrimler@kth.se

Peer reviewed by:
David Segersson

Mohammed Arabnejad

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.

2017-12-22

Contents

Learning outcomes

The reader will learn:

How to use it

- How to run any of the example cases, in single or parallel run

The theory of it

- The basics behind a fuel cell system

How it is implemented

- How the openFuelCell code is constructed

How to modify it

- How to make the code work with OpenFOAM-5.x and OpenFOAM-v1706

Chapter 1

Prerequisites

1.1 Obtaining openFuelCell

The openFuelCell code is available through a git repo hosted at sourceforge.

To obtain the code which targets OpenFOAM-3.0.x, use

```
git clone -b V3 git://git.code.sf.net/p/openfuelcell/git openfuelcell
```

and to checkout the specific commit built on in this tutorial, use

```
cd openfuelcell
git checkout 9b94b74
```

1.2 Patching openFuelCell to work with OpenFOAM-v1706

The necessary changes are supplied in the file `OF1706+_Grimler.patch`which is attached to this pdf.

Checkout a new branch since the modified version will not work for OpenFOAM 3.0.x anymore. `V5` can be a suitable name since the model will work both with OpenFOAM-v1706 and OpenFOAM-5.0.x

```
git checkout -b V5
```

To apply the patch when standing in the repo, use

```
git apply OF1706+_Grimler.patch
```

The contents of the patch are summarized in the following section.

1.2.1 Necessary changes for openFuelCell to work with OpenFOAM-v1706

In OpenFOAM-4.x, the way to access the internal fields of the mesh was changed. The changes means that *variable*.internalField() needs to be replaced with *variable*.primitiveFieldRef(), and *variable*.boundaryField() with *variable*.boundaryFieldRef(). More details about the changes and what lead to them can be found in commit a4e2afa4b in the OpenFOAM-dev branch.

On 8 places, there are changes analogous to this patch as well:

```
-    OPstream toNeighbour(Pstream::blocking, neighbour);
+    OPstream toNeighbour(Pstream::commsTypes::blocking, neighbour);
```

These changes originate from commit 1e6c9a0a5 in the OpenFOAM-dev branch. From the limited information leading up the commit, it seems this is done to make the code more robust. This stackexchange question explains it better than I can: `https://stackoverflow.com/questions/18335861/why-is-enum-class-preferred-over-plain-enum`.

Smaller changes to improve compatibility with OpenFOAM-v1706

Between OpenFOAM-2.x.x and OpenFOAM-3.0.0, the blockMeshDict file was moved from `constant/polyMesh/blockMeshDict` to `system/blockMeshDict`. The code works without the change but gives a warning.

Chapter 2

Introduction to fuel cells

A fuel cell is a device capable of transforming chemical energy into electrical energy. The process is highly efficient with an thermodynamic efficiency surpassing 80 %. If the waste heat is utilised, the overall efficiency can become even higher.

There are many types of fuel cells, utilising different types of fuels. Some examples are proton-exchange membrane fuel cell (PEMFC), direct methanol/ethanol fuel cell, solid oxide fuel cell (SOFC) and molten carbonate fuel cell. This work focuses on PEMFC.

There are several types of PEMFCs as well, namely acidic (proton-exchange membrane fuel cell) and alkaline (anion-exchange membrane fuel cell). Of these, the acidic one is furthest developed and this work will therefore focus on this type. Both types can be further divided into low-temperature (LT) (below 100 °C) and high-temperature (HT) (above 100 °C). The available openFuelCell code is currently setup for the high temperature type.

A PEMFC utilises hydrogen as a fuel and oxygen as an oxidant. The reactant gases are fed to different compartments separated by an electrolyte, in the form of a polymer membrane. The compartments contain catalyst material, on electrodes, which are connected to an outer circuit containing for example an electric engine.

The polymer membrane conducts hydrogen ions (protons) but is impermeable to the reactant gases. This separation prevents the reactants from reacting as in a normal combustion, instead these two half-cell reactions take place:

$$2\,H_2 \longrightarrow 4\,H^+ + 4\,e^- \quad \{1\}$$

$$O_2 + 4\,e^- + 4\,H^+ \longrightarrow 2\,H_2O \quad \{2\}$$

The electrons are transfered through the outer circuit, thereby creating a current that can be used. To close the circuit, the protons ions are transfered through the membrane. The entire process is depicted in figure Figure 2.1.

2.1 Resistances and other losses

In this system, there are resistances in the different subparts. The reactant gases have to diffuse from the bulk of the gas phase, to the electrode surface. To complicate further, the electrodes are highly porous, which means that the reactants have to diffuse into the narrow pores of the electrodes. This diffusion can give rise to concentration gradients meaning that the active sites experience a lower reactant concentration than the concentration in the bulk of the gas phase. The reaction rate drops due to this.

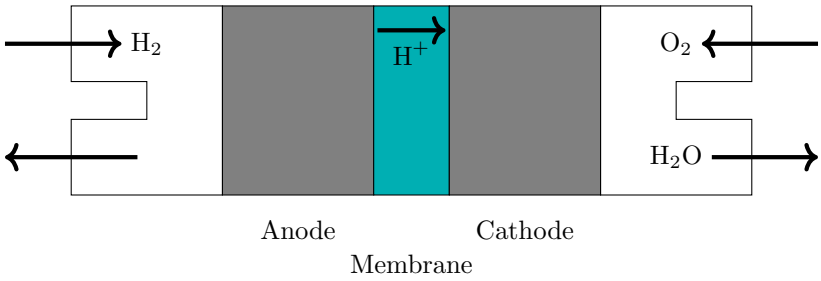

Figure 2.1: Schematic overview of the reactions and ion movement in an proton-exchange PEMFC operating with co-current reactant gas flows.

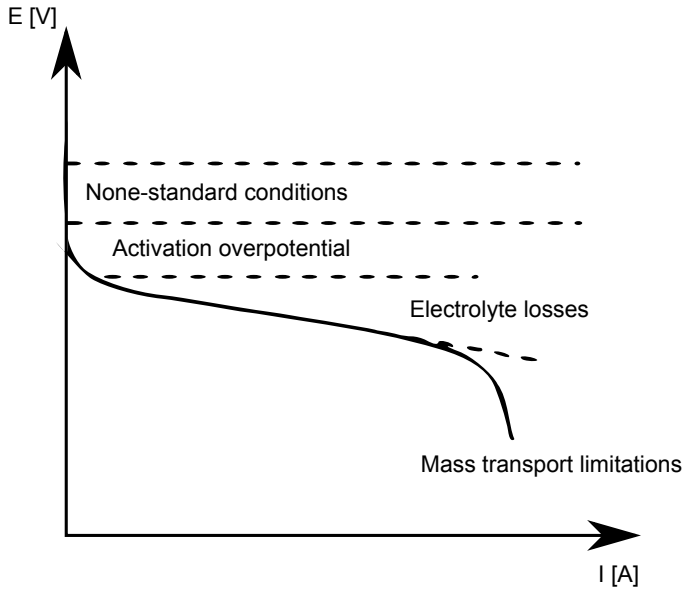

Figure 2.2: Losses from different processes in a fuel cell system, marked in a polarization curve.

The membrane has a finite and none-constant proton conductivity as well. During operation, the transport of protons in the membrane can limit the reaction rate at the active sites. Even as the reactants reach the active sites, there is still a thermodynamical barrier to cross. A so called charge-transfer resistance, or activation overpotential, give additional resistance as the electrons are forced to move between the electrodes and the electrolyte species. The definition of an overpotential is the difference between the actual potential and the potential at standard conditions and can be written as

$$\eta = E - E^{\circ} \tag{2.1}$$

Where E° is the potential at standard conditions.

In a polarization curves, the different losses are usually marked as in Figure 2.2.

If the reactant gas pressures vary, this affects the potential the electrodes experience. The correlation is called Nernst's equation.

$$E = E^{\circ} + \frac{RT}{nF} \ln \left(\frac{a_{\mathrm{H_2}}^2 a_{\mathrm{O_2}}}{a_{\mathrm{H_2O}}} \right) \tag{2.2}$$

where E° is the potential at standard conditions and a_x is the activity of specie x.

In the activation region, the relation between the current and overpotential is described by the

Butler-Volmer equation

$$i = i_0 \left(\exp\left(\frac{2\beta F}{RT} \eta \right) - \exp\left(-\frac{2(1-\beta)F}{RT} \eta \right) \right) \tag{2.3}$$

Where i_0 is the exchange current density, which describes the activity of the surface, β is a symmetry parameter describing if the forward or backwards reaction is more favourable than the other one. F is Faradays constant ($96\,485\,\mathrm{A\,s\,mol^{-1}}$), R is the gas constant and T the temperature.

i_0 can be calculated from a Arrhenius expression

$$i_0 = p_{\mathrm{O_2/H_2}}^{\alpha} \gamma T \exp\left(-\frac{E_A}{RT} \right) \tag{2.4}$$

where p_x is the partial pressure of specie x, γ is a pre-exponent factor and E_A is the activation energy for the reaction. The values for γ and E_A are specified in constant/electrolyte/activationParameters. The default values are from Leonide et al. [1].

One more correlation between current density i and overpotential η is needed. openFuelCell uses a lumped resistance model

$$i = \frac{E - \eta_{an} + \eta_{cath}}{R} \tag{2.5}$$

where E is the potential calculated by Equation (2.2) and the resistance R is calculated from an empirical correlation dependent on temperature. The correlation is described further in Section 3.1.1.

2.2 Fuel cell modeling with OpenFOAM

The openFuelCell code was originally created for modeling SOFCs, but has since then been modified to also target HT-PEMFC. The project was originally founded by Forschungszentrum Jülich, National Research Council Canada, Queen's University/Royal Military College Fuel Cell Research Centre, and Wikki Ltd.

There are other open-source codes available for modeling fuel cells. One notable example is FAST-FC which is also based on OpenFOAM (Extended). It has been developed for performance and degradation modeling [1].

[1] https://www.fastsimulations.com/

Chapter 3

Tutorial openFuelCell

3.1 The OpenFuelCell project structure

The openFuelCell code is divided into a library and an executable. When standing in openfuelcell/src, there are therefore two subfolders, `libSrc` and `appSrc`.

A brief introduction to files whose purpose is not obvious will now follow.

3.1.1 Walkthrough of the main code sofcFoam.C

The main code, which includes all other files, is `sofcFoam.C`. The name comes from the fact that openFuelCell earlier was called sofcFoam. When the code was generalized to also deal with HT-PEMFC, it was renamed (at some places).

The file starts off with the normal header, crediting the main authors.

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           |
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software; you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by the
    Free Software Foundation; either version 2 of the License, or (at your
    option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM; if not, write to the Free Software Foundation,
    Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA
```

```
Application
    sofcFoam

Description
    Steady solver for the idealised fuel cell model under project work for
    NRC Canada, Feb/2007-

Developed by
    Hrvoje Jasak (h.jasak@wikki.co.uk)
    Steven Beale (s.beale@fz-juelich.de)
    Dong Hyup Jeon (DongHyup.Jeon@nrc-cnrc.gc.ca)
    Helmut Roth (helmut.roth@nrc-cnrc.gc.ca)
    Hae-won Choi (haewon1972@gmail.com)

\*---------------------------------------------------------------------------*/
```

First, necessary headers and functions are included.

```
#include <iostream>
#include <stdio.h>
#include <iomanip>

#include "fvCFD.H"
#include "atomicWeights.H"
#include "physicalConstants.H"
#include "specie.H"
#include "speciesTable.H"
#include "sofcSpecie.H"

#include "patchToPatchInterpolation.H"
#include "continuityErrs.H"
#include "initContinuityErrs.H"
#include "fixedGradientFvPatchFields.H"
#include "smearPatchToMesh.H"

#include "diffusivityModels.H"
/* #include "porousZones.H" Deleted SBB */
#include "porousZoneList.H" //Added SBB
#include "polyToddYoung.H"
#include "RiddersRoot.H" //added SBB (previously)
#include "testFunction.H" //added SBB (previously)

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

The

```
#include "fvCFD.H"
```

to

```
#include "sofcSpecie.H"
```

block defines the finite volume method, atomic weights, physical constants (Advogadros constant, the gas constant, Faraday's constant and ambient pressure) and defines the specie class. The code uses a slightly modifed species class, which is described in **??**.

The second code block,

```
#include "patchToPatchInterpolation.H"
```

to

```
#include "smearPatchToMesh.H"
```

maps the velocity boundary condition from the anode to the cathode (patchToPatchInterpolation.H), calculates the continuity errors and then calculates the patch field values using the errors and cell values.

```
#include "diffusivityModels.H"
```

defines four different diffusion coefficient models, these are defined in `libSrc/diffusivityModels/` and the available models are FixedDiffusivity, Knudsen, binaryFSG and porousFSG.

Diffusivity models

In FixedDiffusivity, the diffusion coefficient for a species is constant (and the value given in the dictionary used as is). In Knudsen diffusion, the diffusion coefficient is calculated as a function of pore diameter, temperature and the molecular weight of the specie considered:

$$D_{knudsen} = \frac{D_{pore}}{2} 97 \sqrt{\frac{T}{M_w}} \tag{3.1}$$

Where D_{pore} is the pore diameter (specified in the constant/(air or fuel)/porousZones dictionary).

For the binaryFSG case the diffusion coefficient is calculated as a function of molecular weights, temperature and diffusion volumes. Diffusion volumes are tabulated values specified in libSrc/diffusivityModels/fsgDiffusionVolumes/fsgDiffusionVolumes.C.

$$D_{binaryFSG} = \frac{10^{-7} T^{1.75} \sqrt{M_{w,A}^{-1} + M_{w,B}^{-1}}}{p \left(V_A^{1/3} + V_B^{1/3} \right)^2} \tag{3.2}$$

porousFSG calculates the diffusion coefficient as a combination of the knudsen diffusion and the binaryFSG diffusion:

$$D_{porousFSG} = \frac{\epsilon}{\tau} \frac{1}{D_{binaryFSG}^{-1} + D_{knudsen}^{-1}} \tag{3.3}$$

where ϵ is the porosity and tau is the tortuosity of the porous material.

```
#include "porousZoneList.H"
```

takes porosity into account by manipulating the Navier-Stokes equations (by attenuation of the time derivative and adding a sink term).

There are three models available,

- Fixed Coeffient (fixedCoeff)
- Power Law (powerLaw)
- Darcy Forchheimer (DarcyForchheimer)

These models have originated from `$FOAM_SRC/finiteVolume/cfdTools/general/porosityModel/` but (except for renaming off porosityModel.* to porousZone.*) only porousZone.C, porousZone.H and porousZoneI.H differs. More member functions have been added and a sanity check on ϵ added ($0 < \epsilon < 1$).

`#include "polyToddYoung.H"`

calculates heat capacities (and from the heat capacity, the specific enthalpy and entropy can be calculated), viscosities and thermal conductivity from a polynomial of degree 6. The coefficients are specified in constant/(air or fuel)/sofcSpeciesProperties.

$$\sum_{k=0}^{k=6} a_k \cdot T^k \tag{3.4}$$

The correlation and coefficients that are present in the tutorial code are obtained from Todd and Young [2].

```
int main(int argc, char *argv[])
{
#   include "setRootCase.H"
#   include "createTime.H"

    // Complete cell components
#   include "createMesh.H"
#   include "readCellProperties.H"
#   include "createCellFields.H"

    // Interconnect0 components
#   include "createInterconnectMesh.H"

     // Air-related components
#   include "createAirMesh.H"
#   include "readAirProperties.H"
#   include "createAirFields.H"
#   include "createAirSpecies.H"

    // Electrolyte components
#   include "createElectrolyteMesh.H"
#   include "readElectrolyteProperties.H"
#   include "readActivationParameters.H" //Added SBB
#   include "createElectrolyteFields.H"

    // Fuel-related components
#   include "createFuelMesh.H"
#   include "readFuelProperties.H"
#   include "createFuelFields.H"
#   include "createFuelSpecies.H"

#   include "readInterconnectProperties.H"

#   include "readRxnProperties.H"

#   include "setGlobalPatchIds.H"

    // calculate electrolye thickness, hE
#   include "electrolyteThickness.H"

    // Cathode & Anode interpolation
#   include "createPatchToPatchInterpolation.H"
```

```
    // Gas diffusivity models
#   include "createDiffusivityModels.H"

#   include "varInit.H" //Added Qing, 25.06.2014

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

Then the `main()` function is entered and the various meshes, variables and fields are setup.

```
    Info<< "\nStarting time loop\n" << endl;

    bool firstTime = true;

    for (runTime++; !runTime.end(); runTime++)
    {
        Info<< "Time = " << runTime.timeName() << nl << endl;

    #   include "mapFromCell.H"     // map global T to fluid regions

    #   include "rhoAir.H"
    #   include "rhoFuel.H"

    #   include "muAir.H"
    #   include "muFuel.H"
// Following lines added SBB
    #   include "nuAir.H"
    #   include "nuFuel.H"
// End lines added SBB
    #   include "kAir.H"
    #   include "kFuel.H"

    #   include "solveFuel.H"
    #   include "solveAir.H"
    #   include "ReynoldsNumber.H"

    #   include "diffusivityAir.H"
    #   include "diffusivityFuel.H"

    #   include "YfuelEqn.H"
    #   include "YairEqn.H"


    #   include "solveElectrochemistry.H"

    #   include "mapToCell.H"
    #   include "solveEnergy.H"

        runTime.write();

        if(firstTime)
        {
            firstTime = false;
        }

        Info<< "ExecutionTime = "
```

```
            << runTime.elapsedCpuTime()
            << " s\n\n" << endl;
    }

    Info<< "End\n" << endl;
    return(0);
}
```

After that the actual loop begins.

```
#include "mapFromCell.H"
```

maps the temperature on the full cell mesh to the anode and cathode submeshes.

On these submeshes, the gas properties are updated and then the Navier-Stokes equations solved. Next step is to update the diffusion coefficients. And after this the molecular fractions can be updated using the Kirchoff-Ohm law.

```
#include "solveElectrochemistry.H"
```

then deals with the electrochemistry by calculating the local current density and potential. The boundary conditions for the Navier-Stokes equations are then updated before the resulting heat capacity and temperature fields are copied back into the full cell model. Lastly, the energy balance is solved for and the loop restarts if the simulation is not finished.

Electrochemistry files

These files deal with the electrochemistry:

`activationOverpotential.H`
`idensity.H`
`NernstEqn.H`
`ASRfunction.H`
`solveCurrent.H`

They are included in `solveElectrochemistry.H`. `solveElectrochemistry.H` also contains code to correct the molar fractions due to the consumption and production of the different species. This is calculated using Faradays law

$$\Delta\dot{m} = \frac{\nu M i}{F n} \tag{3.5}$$

where ν is the half cell reaction coefficient of the species and n is the number of electrons involved in the considered half cell reaction. ν is negative for a reactant and positive for a produced species. F is Faradays constant, $\Delta\dot{m}$ the change in mass flux of the considered species and M the molecular weight of the considered species.

`activationOverpotential.H` calculates the activation overpotential by solving the Butler-Volmer equation (Equation (2.3)) using Ridders' Method.

`idensity.H` smears the current distribution and potential.

`NernstEqn.H` (Equation (2.2)) corrects for the fact that the conditions in the system are not at standard conditions, using Nernst equation. Activities and temperatures are corrected for.

`ASRfunction.H` calculates the electrolyte resistance as a function of temperature. For the HT-PEMFC case, the default empirical correlation looks like:

$$R = 1.0 \cdot 10^{-4}(0.4025 - 0.0007 T_{cath}) \tag{3.6}$$

With a unit of $\Omega\,\mathrm{m}^2$.

`solveCurrent.H` solves for the stack current and potential.

Temperature files

`electrochemicalHeating.H` calculates the heat produced by the chemical reactions using thermodynamical data.

The overall heat balance in the electrode subdomains looks like

$$-\left(H(\mathrm{H_2O}) - H(\mathrm{H_2}) - 0.5H(\mathrm{O_2})\right) \cdot \frac{i}{(2F)} - E \cdot i = Q \cdot L_{\mathrm{electrolyte}} \tag{3.7}$$

Where the H values depend on the local reaction. Q is the change in heat with unit $\mathrm{kJ\,m^{-3}\,s^{-1}}$, E the local potential, i the local current density and $L_{electrolyte}$ is the thickness of the electrolyte layer.

The overall energy balance is dealt with in `energyBalance.H` where all the heat fluxes from all the reactants and products are summed.

Molecular fluxes and physical properties

The files

`kAir.H`
`kFuel.H`
`muAir.H`
`muFuel.H`
`nuAir.H`
`nuFuel.H`
`rhoAir.H`
`rhoFuel.H`

calculates weighted mean values for the physical properties k (thermal conductivity), μ (viscosity), ν (dynamic viscosity) and ρ (density). The values are simply weighted using the molar fractions.

Diffusion coefficients are calculated from binary diffusion coefficients using the relation

$$D_a = \frac{1 - x_a}{\sum_{b \neq a}(x_b / D_{a,b})} \tag{3.8}$$

These calculations happen in `diffusivityAir.H` and `diffusivityFuel.H`.

It is also wort noting that the file `physicalConstants.H` contain physical constants such as the gas constant, Faradays constant, avogadros number, and more.

Molar fractions

The molar fractions are calculated by the files

`appSrc/getXair.H`
`appSrc/getXfuel.H`
`appSrc/YairEqn.H`
`appSrc/YfuelEqn.H`

3.1.2 Files in libSrc

Many of the files in libSrc are copies, with minor changes, of files in the OpenFOAM src directory. Some files are unique to openFuelCell though.

`libSrc/sofcSpecie/sofcSpecie.C`
`libSrc/sofcSpecie/sofcSpecie.H`
`libSrc/sofcSpecie/sofcSpecieI.H`

These files contain the definitions properties needed to define the chemical species in the system. The full definitions is:

```
inline sofcSpecie
(
    const word& name,
    const scalar molWeight,
    const scalar nElectrons,
    const label rSign,
    const scalar hForm,
    const scalar sForm
);
```

- *name* is the name of the species, for example H2
- *molWeight* is the molecular weight of the species
- *nElectrons* is the number of electrons released per mole species in the reaction
- *rSign* is the reaction sign, -1 if reactant, 0 if inert and +1 if product
- *hForm* is the specific enthalpy of formation for the species
- *sForm* is the specific entropy of formation for the species

The sofcSpecie files are based on the files in `$FOAM_SRC/thermophysicalModels/specie/specie/`, with some added properties.

`libSrc/polyToddYoung/polyToddYoung.C`
`libSrc/polyToddYoung/polyToddYoung.H`

These files contain code to calculate physical parameters from 6 coefficients as a function of temperature.

3.2 The different meshes

The openFuelCell code uses in total 5 meshes.

- cell, the full cuboid/shape including all regions
- air, the channels where the cathodic species flow
- electrolyte, the thin region in between the cathode and anode sides
- fuel, the channels where the anodic species flow
- interconnects, the current-collectors or bi-polar plates

The files responsible for the mesh generation are in `appSrc/create<region>Mesh.H`.

Chapter 4

Running openFuelCell

After the repository has been checked out with the instructions in Section 1.1, the openFuelCell solver can be built and then tested.

To build the solver, setup OpenFOAM by sourcing an etc/bashrc file or run an appropriate alias. After that, navigate to ./src/ (from the base of the repository) and run `./Allwmake`. The produced solver is named fuelCellFoam and ends up in $FOAM_USER_APPBIN.

OpenFuelCell comes with 5 example cases, coFlow, counterFlow, crossFlow, quickTest and quickTestStack that are found in ./run/ from the base of the git repository. The difference between the cases is the geometry of the cell, as indicated in the names. QuickTestStack does calculations on a stack consisting of three cells in series, the geometry can be seen in Figure 4.1

The quickTest is based on coFlow but simplified.

All the example cases include Makefiles which can help running the cases. It is possible to run the cases both in single mode and in parallel mode. The command for "make all" looks like:

```
All: mesh parprep run reconstruct view
```

Which corresponds to the following steps

- The mesh is created from blockMeshDict by running blockMesh
- Then the mesh is decomposed so that the model can be run in parallel
- Then the actual solving is done by running fuelCellFoam
- After this the mesh can be reconstructed
- And finally the VTK files generated

Before the model can be run in parallel mode, the environmental variable NPROCS has to be set to the number of cores to use in the run. This can be done with `export PROCS=#`, where # is the number of processors to use.

To run the model in single mode do "make mesh", "make srun" followed by "make view" to generate the VTK files.

To instead run the model in parallel mode, it is enough to set the number of processors and then run "make all". Step by step this corresponds to running, "make mesh", "make parprep", "make run", "make reconstruct" and "make view".

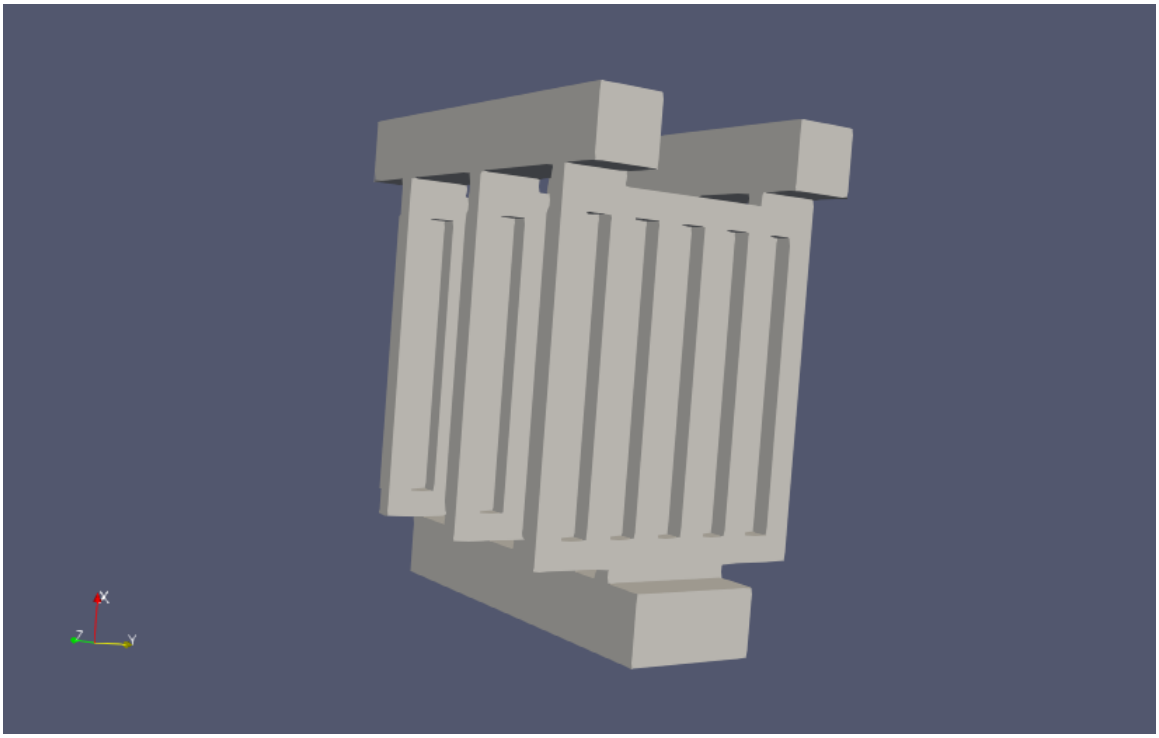

Figure 4.1: QuickTestStack geometry, air side. The air enters throw the two pipes on the top and can then go down through the channels in the middle. From these channels, the gas can diffuse into the cell and react. Surplus gas then exit in the gas channel in the bottom. The fuel side looks the same but upsidedown (not shown in the image).

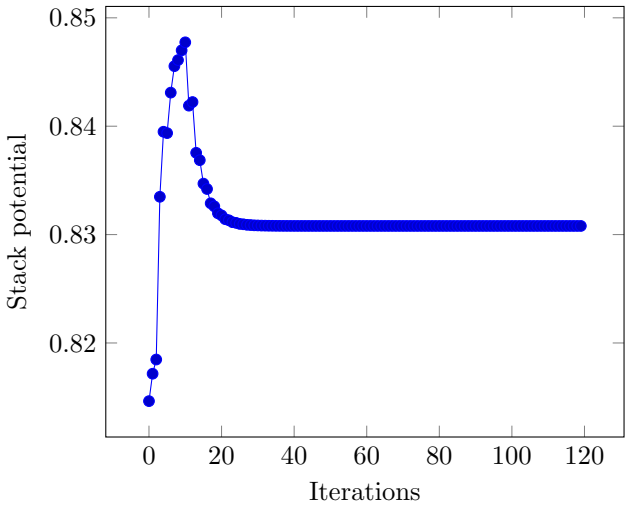

Figure 4.2: Convergence of stack potential for default, galvanostatic, model step.

4.1 Results for quickTest

In the quickTest case, the default setup case is a galvanostatic run where the cell current is set to $5000\,\mathrm{A\,m^{-2}}$ and the initial guess for the cell potential is 0.8 V.

Running the simulation outputs a cell potential that approaches a stable value as in Figure 4.2

After about 40 iterations, the system has stabilized.

Section 4.1 shows the cuboid geometry for the quickTest case. The reactants travel in co-current flow and the molar concentrations decrease along the channels as the reaction occur.

4.1.1 Obtaining polarization curve

By varying the cell current (or cell potential in potentiostatic mode), and changing the initial guess for the cell potential, a polarization curve can be obtained. This can be done in a shell script using sed.

```
#!/usr/bin/bash
# Run from quickTest folder (or another case)
# Start by cleaning folder just in case
./Allclean
make mesh
make parprep

# guess cell potentials corresponding to the cell currents below
guess_vol=(1.06 1.05 1.04 1.2 1.0  0.95 0.9  0.85 0.8  0.75 0.7  0.65 0.6)
cur=(200  400  600  800 1000 2000 3000 4000 5000 6000 7000 8000 9000)
# Remove previous results
rm polarization_curve.txt

for i in $(seq 0 15); do
    # sed and replace previous voltage and current with new guess
    sed -i "/ V [1 2 -3 0 0 -1 0]/c\voltage         V [1 2 -3 0 0 -1 0] \
        ${guess_vol[$i]};" constant/cellProperties
```

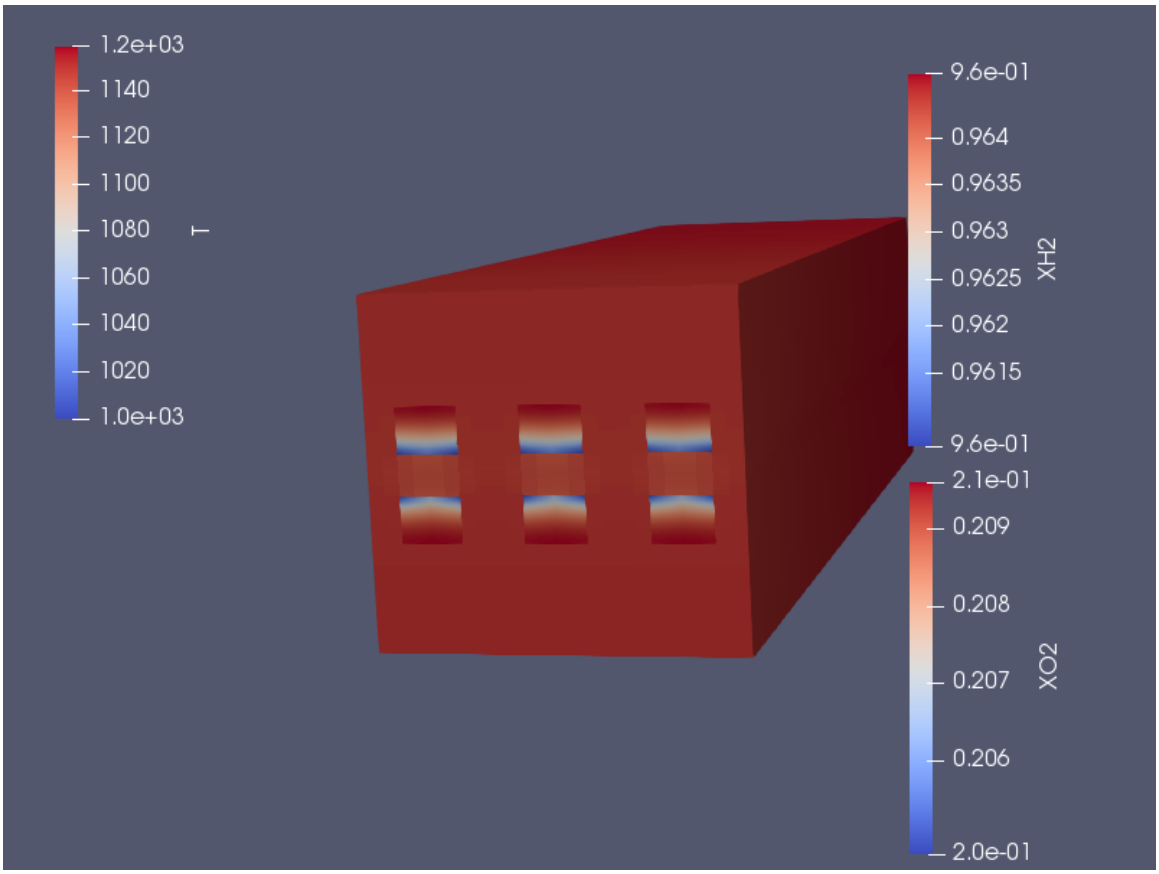

Figure 4.3: The geometry of the quickTest system. The top flow channels are the inlets of the fuel channels while the bottom ones are the inlet of the air channels. The simulation shows the converged solution at $5000\,\mathrm{A\,m^{-2}}$, corresponding to a potential of $0.831\,\mathrm{V}$. As can be seen, the default molar fraction of oxygen is $21\,\%$ and for hydrogen $96\,\%$.

```
    sed -i "/ ibar0 [0 -2 0 0 0 1 0]/c\ibar0           ibar0 [0 -2 0 0 0 10] \
        ${cur[$i]};" constant/cellProperties

    # run!
    make run
    # cp results to save it
    cp constant/cellProperties cellProperties.${cur[$i]}
    cp log.prun log.prun.${cur[$i]}
    # print current and grep voltage and output to text file
    echo ${cur[$i]} $(grep "stack Voltage" ./log.prun.${cur[$i]} | \
        tail -n 1 | awk -F= '{print $2}') >> polarization_curve.txt
done
```

The script prints cell current and cell potential to a text file (polarization_curve.txt).

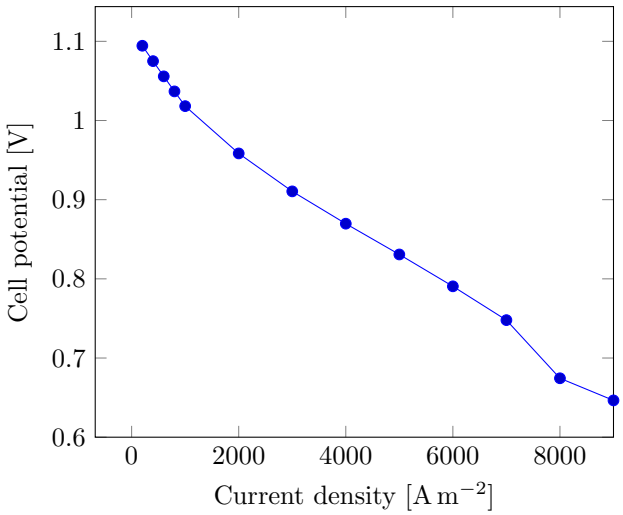

Figure 4.4: Polarization curve obtained by running openFuelCell at different cell currents and potentials.

Bibliography

[1] Andre Leonide, Yannick Apel, and Ellen Ivers-Tiffee. Sofc modeling and parameter identification by means of impedance spectroscopy. *ECS Transactions*, 19(20):81–109, 2009. doi: 10.1149/1.3247567. URL http://ecst.ecsdl.org/content/19/20/81.abstract.

[2] B Todd and J B Young. Thermodynamic and transport properties of gases for use in solid oxide fuel cell modelling. *Journal of Power Sources*, 110(1):186–200, 2002. ISSN 0378-7753. doi: https://doi.org/10.1016/S0378-7753(02)00277-X. URL http://www.sciencedirect.com/science/article/pii/S037877530200277X.

Study questions

4.1.2 How to use it

- How is the example cases run in parallel mode and how are they run in single mode?

4.1.3 The theory of it

- Which processes gives rise to a temperature change in the cell?

4.1.4 How it is implemented

- Which form are the specific heat capacities for the molecular species supplied in?

4.1.5 How to modify it

- Briefly describe the necessary changes to have the openFuelCell code working with OpenFOAM-v1706