

Open-source Fuel Cell Simulation Toolbox (OpenFCST)

-

User's and Developer's Reference Guide

The OpenFCST development team

Energy Systems Design Laboratory
University of Alberta, Canada

Created on: June 11, 2012
Last updated: December 13, 2016

Contents

1	Introduction	5
1.1	About OpenFCST	5
1.2	About the Developers	6
1.3	Release notes	8
1.4	License	9
I	User's Guide	11
2	Getting started	13
2.1	Downloading OpenFCST	13
2.2	Documentation in OpenFCST	13
2.2.1	Tutorial examples	13
2.2.2	Reference guide	14
2.2.3	Class documentation	14
3	Installation	15
3.1	Installing OpenFCST	15
3.1.1	System requirements	15
3.1.2	Installation steps	16
3.1.3	Setting up a virtual Python environment: CONDA	16
4	OpenFCST structure	19
4.1	Install directory tree	19
5	Pre-processor	21
5.1	FuelCellShop::Geometry Namespace	21
5.2	Developing a mesh in SALOME	22
5.2.1	Tutorial	22
5.3	SALOME meshing using python scripts	31
5.3.1	Introduction	31
5.3.2	Scripting Examples	31
5.4	Generating a VTK mesh using PythonFCST	33
5.4.1	Using the Python executable writeVTK.py	33
5.4.2	Using the MESH module in PythonFCST	34
6	Running OpenFCST	37
6.1	Setting up a simulation in OpenFCST	37
6.2	OpenFCST examples	38
6.3	OpenFCST's graphical user interface	38
6.3.1	Overview	39

6.3.2	How To's	40
6.3.3	Configuration	41
6.3.4	Reporting Errors	41
6.4	The OpenFCST main file	42
6.4.1	Simulator section	42
6.4.2	The Logfile section	43
6.5	The OpenFCST data file	43
6.5.1	The Adaptive refinement section	44
6.5.2	The Newton section	45
6.5.3	The Grid generation section	46
6.5.4	The Discretization section	48
6.5.5	The System management section	48
6.5.6	The Equations section	49
6.5.7	The Reaction source terms section	49
6.5.8	The Initial Solution section	49
6.5.9	The Linear Solver section	50
6.5.10	The Fuel cell data section	50
6.5.11	The Output section	51
6.5.12	The Output Variables section	51
6.5.13	The Postprocessing section	51
7	Post-processor	53
II	Developer's Reference Guide (Under development)	55
8	Setting up the development environment for OpenFCST	57
8.1	Getting the development version of OpenFCST	57
8.2	Setting up OpenFCST under KDevelop	59
8.2.1	Formatting OpenFCST files	59
9	Coding Guidelines	63
9.1	Class and Member Naming Conventions	63
9.2	File headers	64
9.3	Developing documentation using Doxygen	64
9.3.1	Documenting classes	64
9.3.2	Documenting member functions	66
9.3.3	Documenting variables	66
9.3.4	Documenting namespaces	66
9.3.5	TODO list in HTML documentation	67
9.3.6	Linking to other functions	67
9.4	Assertions and exception handling	67
10	Development Process	69
10.1	Test Driven Development	69
10.1.1	Unit Tests	70
10.1.2	TDD Implementation in the OpenFCST	71
10.1.3	Implementing a new test suite	73
10.1.4	Refactoring	73
10.1.5	Unit Standards	74

Chapter 1

Introduction

1.1 About OpenFCST

The open-source Fuel Cell Simulation Toolbox (OpenFCST) is an open-source mathematical modelling package for polymer electrolyte fuel cells. OpenFCST has been developed as a modular toolbox from which you can develop your own applications. It contains a database of physical phenomena equations, fuel cell layers and materials, and mathematical models for reaction kinetics. In addition, it already contains several applications that allow you to simulate different fuel cell components. For example, you can simulate a cathode electrode (using either a macrohomogeneous or an ionomer-filled agglomerate model), an anode electrode or a complete membrane electrode assembly. The applications already provided in OpenFCST have been validated with respect to experimental data in the literature [5] as well as numerical results from other models implemented in a commercial package [2]. A thorough description of the model and validation is presented in [2].

OpenFCST is being developed at the [Energy Systems Design Laboratory](#) at the University of Alberta in collaboration with the [Automotive Fuel Cell Cooperation Corp.](#) that, together with the [Natural Science and Engineering Research Council of Canada](#), has provided the majority of the funding required to develop this code. The goal of OpenFCST is that research groups in academia and in industry use the current toolbox to better understand fuel cells and to develop new physics and material databases that can then be integrated in the current library.

OpenFCST is an integrated open-source tool for fuel cell analysis and design. It seamlessly integrates several open-source pre-processing, finite element, and post-processing tools in order to analyze fuel cell systems. OpenFCST contains a built-in mesh generator. If your problem requires you to simulate more complex geometries, it can also import quadrilateral meshes generated with the open-source pre-processor [Salome](#) and exported in UNV format. The physics and material database in OpenFCST allows you to setup the governing equations for the most important physical processes that take place in a fuel cell. OpenFCST already implements the weak form for many governing equations. They are solved using the finite element open-source library [deal.II](#). OpenFCST builds on top of the [deal.II](#) finite element libraries and many of its software requirements and coding philosophy is inherited from deal.II. In order to analyze your results, OpenFCST can output your results to .vtu files that can easily be read with the open-source post-processor [Paraview](#). OpenFCST is also integrated with the design and optimization package [Dakota](#). Therefore, it can be used for design and optimization as well as parameter estimation [2, 3, 4, 5].

OpenFCST is under development. If you like the library and would like to contribute towards the development, you can help the developers in the following ways:

- If you are an industrial researcher that is considering using OpenFCST for research and development in the company, please contact the developers in order to develop a research program with them.
- If you are either an industrial or academic researcher using the library, please make sure to cite the OpenFCST libraries in your publications. Please cite any relevant publication by the OpenFCST

developers as well as the current reference [1] and introduction paper [7].

- If you are either an industrial or academic researcher using the library and you have developed a new physics model or material database entry, please consider submitting it to the developers so that it can be integrated with the newest version of OpenFCST.
- If you are an industrial researcher considering using OpenFCST for research and development in the company, please consider hiring the graduate students that develop OpenFCST, i.e. the graduate students from the [Energy Systems Design Laboratory](#) at the University of Alberta.

Currently, the developers are working on:

- improving the code readability new classes are being developed for making the code easier to understand and more modular;
- developing a convective gas and liquid transport model for the electrodes;
- developing a Navier-Stokes solver for gas transport in the fuel cell channels.

1.2 About the Developers

OpenFCST was originally conceived by M. Secanell in 2006 while doing his Ph.D. at the University of Victoria [2]. In 2004, M. Secanell developed a small set of routines that were used to setup the governing equations for a fuel cell cathode in two dimensions. The governing equations were first linearized and then the weak form of the equations was implemented and solved using the [deal.II](#) finite element libraries [3]. In 2006, after attending a [deal.II](#) workshop in Heidelberg, Germany, and discussing the idea of creating an open-source code for fuel cells based on [deal.II](#) with Dr. Guido Kanschat and Dr. Wolfgang Bangerth, M. Secanell decided to integrate the routines he had developed into AppFrame, an application framework developed by Dr. Guido Kanschat, thereby initiating the development of a toolbox that could be used to create modules or applications for fuel cell analysis. From 2006 to 2008, OpenFCST development continued with the implementation of a complete membrane electrode assembly model; however, with M. Secanell as a sole developer, the code was too rough and disorganized to result in an open-source fuel cell package that the research community could use.

In 2009, once M. Secanell joined the University of Alberta, the idea of developing OpenFCST was solidified. Thanks to the funding provided by the [Automotive Fuel Cell Cooperation Corp.](#), [MITACS](#) and the [Natural Science and Engineering Research Council of Canada](#), a group of core developers was established at the [Energy Systems Design Laboratory](#) at the University of Alberta. Researchers at the Energy Systems Design Lab re-developed the majority of the classes in order to increase the modularity, usability and reliability of the code. Currently, OpenFCST is currently developed by 6-8 researchers at two different laboratories, and it is used by researchers in Canada, England and Germany. It contains unit and regression tests to guarantee accurate results, and contains a bug tracking site to report any issues with its performance. Current and past developers as well as other contributors to OpenFCST are listed below.

Current developers: The current group of OpenFCST developers is formed by:

- M. Secanell, Associate Professor, Energy Systems Design Laboratory, University of Alberta, Canada (2006-): Responsible for overall project management and coordination, framework design (base class concepts), cathode, pemfc and anodeKG applications, Fick's gas transport, Ohmic transport, protonic, membrane water transport, and kinetics classes.
- A. Jarauta, Post-doctoral Fellow, Energy Systems Design Laboratory, University of Alberta, Canada (2016-): Responsible for stabilizing the fluid flow and multi-component solvers
- A. Kosakian, Ph.D. student, Energy Systems Design Laboratory, University of Alberta, Canada (2014-): Responsible for the development of transient solvers and applications (for release 1.0)

- A. Putz, Senior Research Scientist, Automotive Fuel Cell Cooperation Corp. (2010-): Responsible for plug-points and AFCC contributions
- M. Sabharwal, Ph.D. student, Energy Systems Design Laboratory, University of Alberta, Canada (2014-): Responsible for the development of micro-scale simulation applications, e.g. appDiffusion and microscale example.
- J. Zhou, Ph.D. student, Energy Systems Design Laboratory, University of Alberta, Canada (2013-): Responsible for the development of a two-phase flow applications

Past developers: In addition to the current OpenFCST development team, scientists that have contributed substantial portions of code are:

- C. Balen, M.Sc. graduate from the Energy Systems Design Laboratory, University of Alberta, Canada (2014-6): Developed CMake and installation script for release 0.3, extended Navier-Stokes, Darcy and multi-component fluid flow applications, and developed cathodeKG application
- M. Bhaiya, M.Sc. student, Energy Systems Design Laboratory, University of Alberta, Canada (2012-14): Responsible for overall framework (base class concepts) and thermal physical models and applications
- P. Dobson, M.Sc. graduate from the Energy Systems Design Laboratory, University of Alberta, Canada (2010-12): Developed parts of overall framework (base class concepts), optimization interface and multi-scale framework (1D agglomerate models)
- M. Moore, M.Sc. graduate from the Energy Systems Design Laboratory, University of Alberta, Canada (2011-13): Responsible for installation script, double-trap kinetics model for ORR reaction and multi-scale framework (1D agglomerate models)
- V. Zingan, Post-doctoral Fellow, Energy Systems Design Laboratory, University of Alberta, Canada (2012-14): Developed a preliminary version of Navier-Stokes, Darcy and multi-component fluid flow physical models and applications
- P. Wardlaw, M.Sc. student, Energy Systems Design Laboratory, University of Alberta, Canada (2012-14): Responsible for installation script (v 0.1) and multi-scale framework (1D agglomerate models)

Contributors: Other scientists that have also contributed to OpenFCST are:

- K. Domican, M.Sc. student, Energy Systems Design Laboratory, University of Alberta, Canada
Responsible for optimization interface and documentation
- G. Kanschat, Universitt Heidelberg
Developer of AppFrame (now part of OpenFCST application.core routines)
- Simon Mattern, intern at the Energy Systems Design Laboratory, University of Alberta, Canada (2016)
Enhancements to the graphical user interface and PorousLayer class
- A. Malekpourkoupaei, former M.Sc. graduate student at the Energy Systems Design Laboratory, University of Alberta, Canada (2010)
Developed classes PureGas and classes to compute binary diffusivity (together with M. Secanell)

1.3 Release notes

New in Release 0.3 (December 2016):

- Improved graphical user interface (GUI): New toolbar menu for easy access. Ability to convert PRM files to project files (XML) directly through the GUI, as well as to export project files to PRM files. GUI now provides the option to switch between 2D and 3D simulations, as well as to select the number of processors to run OpenFCST with (for parallel computing)
- Micro-scale simulation capabilities allow users to generate a mesh from a .tiff stack and then perform gas transport (with and without Knudsen effects), electron transport, and reaction simulations (see M. Sabharwal, L. Pant, A. Putz, D. Susac, J. Jankovic, M. Secanell, "Analysis of Catalyst Layer Microstructures: From Imaging to Performance", Fuel Cells. doi: [10.1002/fuce.201600008](https://doi.org/10.1002/fuce.201600008))
- Non-isothermal, two-phase flow model based on saturation equation (Note: Release v1.0 will contain an updated model based on capillary equation)
- Parallel capabilities. Compile OpenFCST with flag `--with-petsc` in order to assemble the linear system in parallel and solve the problem using MUMPS, a parallel direct solver.
- Bug fixes in post-processing routines.

In release 0.2 (April 2015):

- New graphical user interface (GUI)
- Non-isothermal membrane electrode assembly model (see M. Bhaiya, A. Putz and M. Secanell, "Analysis of non-isothermal effects on polymer electrolyte fuel cell electrode assemblies", Electrochimica Acta, 147C:294-309, 2014. DOI: [10.1016/j.electacta.2014.09.051](https://doi.org/10.1016/j.electacta.2014.09.051))
- Double-trap kinetic model (see M. Moore, A. Putz and M. Secanell, "Investigation of the ORR Using the Double-Trap Intrinsic Kinetic Model", Journal of the Electrochemical Society 160(6): F670-F681. doi: [10.1149/2.123306jes](https://doi.org/10.1149/2.123306jes))
- Multi-scale framework for analysis of complex agglomerate structures (see M. Moore, P. Wardlaw, P. Dobson, J.J. Boisvert, A. Putz, R.J. Spiteri, M. Secanell, "Understanding the Effect of Kinetic and Mass Transport Processes in Cathode Agglomerates", Journal of The Electrochemical Society, 161(8):E3125-E3137 DOI: [10.1149/2.010408jes](https://doi.org/10.1149/2.010408jes))
- Improved compilation script and transition to CMake: OpenFCST will automatically look for all dependent libraries and download any missing libraries if necessary (installation tested nightly in OpenSUSE 13.1, 13.2, and Ubuntu 14.04)
- Improved documentation: Improved user guide and folder with input files for several of the articles above
- Improved post-processing capabilities: New classes developed to be able to output oxide coverage, agglomerate effectiveness, relative humidity, overpotentials and more
- Improved post-processing capabilities: New classes to compute functionals such as overall current density and all types of heat losses

1.4 License

The Fuel Cell Simulation Toolbox (OpenFCST) is distributed under the MIT License.

Copyright (C) 2013-16 Energy Systems Design Laboratory, University of Alberta

The MIT License (MIT)

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Part I

User's Guide

Chapter 2

Getting started

2.1 Downloading OpenFCST

There are three ways you can obtain OpenFCST,

- Source code from [GitHub](#) (using the `git clone` command)
- Zip file from the [OpenFCST project site](#)
- As a VirtualBox virtual machine from the [OpenFCST project site](#)

If you have a Linux operating system, you can use the first two methods. If you are using a Windows or a Mac, you need to use the third option to run OpenFCST.

If you have obtained OpenFCST from any of the first two options, you will first need to install OpenFCST in your computer. To install OpenFCST please follow the instructions in chapter 3. Once installed, you can run OpenFCST following the steps in chapter 6.

If you have obtained OpenFCST using the latter option, first install VirtualBox in your computer (you can download it [here](#)). Then, open the virtual box and import the OpenFCST virtual machine (VM). Once imported, you can start the machine (OpenSUSE environment). The VM desktop will already contain icons to the OpenFCST GUI. To run OpenFCST follow the steps in chapter 6.

2.2 Documentation in OpenFCST

OpenFCST comes with three different documentation forms to aid users and developers to run and customize OpenFCST as per their needs.

2.2.1 Tutorial examples

Tutorial examples provide a good starting point for the users and developers to execute the existing applications developed by the OpenFCST developers. The examples describe the problem statement and equations that are solved. Then, a step by step procedure to run the application is provided. Additionally, the examples also describe the parameters which are important for the problem being solved to facilitate users to setup their own simulations. The examples can be found at [Tutorial examples](#) or alternately they can be built locally in the OpenFCST installation folder under the examples subfolder. To build the examples, go to the examples folder in the OpenFCST Install directory and type the following,

```
1 make html
```

This will execute the sphinx compiler to create the examples. OpenFCST uses python 2.7 by default. See chapter 3 on how to setup a virtual python environment. To modify an existing example or add a new example see the “How to create your own tutorial” page.

2.2.2 Reference guide

The reference guide which is the current document provides an overview of OpenFCST to the users and developers. The reference guide describes the OpenFCST structure and philosophy. The document provides pre-processing examples for mesh generation using Salome, detailed description of the OpenFCST main and data files are provided which is essential for users and developers who would like to run their own simulations and procedure for setting up and executing a simulation with OpenFCST. The developer's reference guide also provides information for developers who would like to contribute to OpenFCST. Information on setting up an IDE (Kdevelop) and formatting options used for the OpenFCST source code. The document also highlights the coding guidelines and the testing procedure that is implemented to ensure consistency across developers and reliability of the code.

2.2.3 Class documentation

This is the C++ source code documentation generated using Doxygen. The class documentation can be found [here](#). This provides the documentation for the various classes, hierarchy and inheritance diagrams, and description of member functions and variables. Developers and advanced users looking to understand the working of the OpenFCST framework should use the class documentation.

Chapter 3

Installation

3.1 Installing OpenFCST

3.1.1 System requirements

OpenFCST is developed on Linux and compiled using the GCC compiler. OpenFCST developers have performed compilation tests on the following operating systems:

- OpenSUSE 13.1 (evergreen), Leap 42.1 and Tumbleweed;
- Ubuntu 16.04

These are the operating systems that the OpenFCST developers recommend.

If you would like to try to run the code under Windows environment, our recommendation is to install a VirtualBox with OpenSUSE and then, install and run OpenFCST on the virtual machine.

The following software needs to also be installed in your computer in order for OpenFCST to compile:

- GNU make and C++11 support, gcc version 4.7 or later (4.8.1 Recommended)
- GCC
- BLAS and LAPACK libraries (both the library and lapack-devel)
- OpenMPI compiler
- gfortran compiler
- Bison
- libqt4 and libqt4-devel
- For generating the documentation: DOxygen and Sphinx
- Boost; the specific packages are iostreams, serialization, system, thread, filesystem, regex, signals, & program_options)
- FLEX (For Dakota)
- Python Packages: SciPy, NumPy, ipython, Sphinx, evtk, vtk, mayavi, matplotlib (with backends)
- libconfig-devel and libconfig++-devel

Although it is recommended to use C++11, OpenFCST compiles with C++98 also.

In addition, the following packages might be useful if you are planning on developing new classes for OpenFCST:

- For debugging programs, we have found that the GNU debugger GDB is an invaluable tool. GDB is a text-based tool not always easy to use; kdbg is one of many graphical user interfaces for it.
- Most integrated development environments like KDevelop or Eclipse have built in debuggers as well.

3.1.2 Installation steps

Fuel Cell Simulation Toolbox is a fuel cell simulation package developed using several open-source libraries such as the deal.II libraries, PETSc and COLDAE. In order to run without any difficulties, OpenFCST needs to compile and link to all these applications which are provided with the code in the folder `src/contrib`. Please note that each package is distributed under a different license.

OpenFCST contains a script to compile all packages simultaneously. To compile OpenFCST and all other libraries use the following:

```
1 $ ./openFCST_install --cores=4
```

This is the basic OpenFCST install with deal.II and COLDAE. CMake will try to find the contributing libraries in the `src/contrib` folder. If they are not found then it will try to download them from the internet. Since MPI is mandatory and CMake finds OpenMPI for you we do not need to specify any flag to tell CMake where to find it. Finally, we select to compile on four cores to speed up the compilation process.

The install script assumes the default path for the OpenMPI compiler and that all the libraries are in `src/contrib`. If you already have a version of deal.II and you would like to use that version, use the flags `--deal-dir`. Please check the `src/README` for any necessary changes that must be made to deal.i for it to work with OpenFCST. For more information on the script options, type

```
1 $ ./openFCST_install --help
```

Release 0.3 of OpenFCST now supports parallel execution. To be able to use this feature OpenFCST and deal.II libraries need to be compiled using PETSc. To compile OpenFCST with parallel execution mode execute the following,

```
1 $ ./openFCST_install --with-petsc --cores=4
```

This will install PETSc, p4est and METIS, and then compile deal.II and OpenFCST. Once, installed the parallel option would be available on GUI also. In order to run the OpenFCST binaries in parallel, see chapter 6.

3.1.3 Setting up a virtual Python environment: CONDA

The PythonFCST package which can be used for generating VTK meshes for microstructure simulations and a bunch of plotting routines use additional python libraries. The python package uses python 2.7 and therefore is not compatible with python 3. The migration to python 3 will be done in a later release. Therefore, it is suggested that the users create a CONDA environment to use the PythonFCST package. This environment can be created before running OpenFCST install to ensure that the OpenFCST python dependencies are satisfied.

In order to set up a CONDA environment with the necessary OpenFCST dependencies, please follow these steps:

1. Download the Python 3.5 64 Bit CONDA installer from the website: <https://www.continuum.io/downloads>
2. To install CONDA open a Konsole/Terminal and go to the folder where the installer was downloaded and type,

```
1 $ bash Anaconda3-4.2.0-Linux-x86_64.sh
```

3. When it asks you for a path to install anaconda, preferably put some local folder. For this walkthrough, it is assumed that you install it in `/home/user/anaconda3`. Also it is assumed that you prepend your `.bashrc` with the anaconda bin folder that is:


```
1 $ export PATH="/home/secanell/anaconda3/bin:$PATH"
```

4. To create the python environment with the necessary packages required for OpenFCST type,

```
1 $ conda create -n fcst_python python=2.7 anaconda mayavi
```

This will install a general array of scientific python packages. To install the bare-minimum packages required for PythonFCST use,

```
1 $ conda create -n fcst_python python=2.7 scipy numpy mayavi matplotlib pandas pil
```

5. Next make sure that QT_API is set properly.

```
1 $ export QT_API=pyqt
```

6. To use the environment, open Konsole and type:

```
1 $ source activate fcst_python
```

This should enable the use of all python library functions within OpenFCST.

Chapter 4

OpenFCST structure

Prior to installation, OpenFCST contains the following main folders:

- **src** contains all source files for OpenFCST, documentation files, licenses and this guide;
- **pre_processing** contains a collection of additional programs to improve the usability of OpenFCST such as a collection of Python scripts for Salome;
- **post_processing** contains a collection of additional Python scripts to further post-process the results obtained using OpenFCST with ParaView;
- **python** contains a collection of Python scripts for plotting polarization curves and further post-processing of the solution.

After installation of OpenFCST, two additional folders will appear:

- **Install** contains the binaries of OpenFCST. **OpenFCST users should only work in this folder;**
- **Build** contains auxiliary files necessary for the compilation of OpenFCST. This folder is therefore of no interest to users of OpenFCST.

The **Install** folder is the only folder users should be concerned with. Users should think of **Install** as installation folder, i.e. the folder where all files necessary to execute OpenFCST are installed. Other folders contain either source code for OpenFCST, i.e. **src** and **python**, or auxiliary routines that are not critical to OpenFCST. In the remaining of the User Manual, we will assume that users have installed the program and they are working from the **Install** directory.

4.1 Install directory tree

The **Install** directory of OpenFCST contains two scripts and nine subfolders. The subfolders are namely:

- **bin** contains binary executable files for OpenFCST. The main three executables are: a) **fuel_cell-2d.bin**, b) **fuel_cell-3d.bin**, and c) **fcst_gui**. The first file is used to run OpenFCST through the terminal for solving 2D problems, the second for running 3D problems, and the last file is the file to execute the graphical user interface.
- **examples** contains a set of example problems to learn how to use OpenFCST. In particular, there are examples to simulate a cathode, a membrane electrode assembly (MEA) with macro-homogeneous and agglomerate models, and a non-isothermal MEA. The files in the example folder should not be modified. Instead, copy the appropriate files to **my_data** and modify as necessary.
- **doc** contains all documentation except for the examples. This includes:

- a main HTML page, i.e., `index.html`, that can be used to access all documentation in OpenFCST;
 - the Users Manual in PDF and \TeX format (in `RefGuide` folder);
 - the class documentation in HTML and \TeX , i.e. the documentation for each routine developed in OpenFCST (in `html` and `latex` folders).
- **contrib** contains the contributing libraries to OpenFCST. These are libraries that have been developed by other people and are used within OpenFCST. They include `deal.II` and `DAKOTA`. Note that some of these libraries have been slightly modified by OpenFCST developers (see `README` file in each subfolder).
 - **databases** contains databases used in the case of numerical agglomerates to speed-up OpenFCST simulations. If you are not using a numerical agglomerate model, you do not need to worry about this folder.
 - **fcst** contains the `.h` files needed in order to link other libraries to OpenFCST. This folder is not necessary for Users.
 - **test** contains the configuration files used to run the tests to make sure the OpenFCST has been installed correctly. These same files are used with `CDash` to make sure OpenFCST continues to provide the same results between releases.
 - **python** contains a collection of Python scripts to help with post-processing. This section is in its infancy.
 - **my_data** does not contain any information. It is created to allow users to store their simulation data.

The two scripts are `fcst.env.sh` and `run_tests`. The first file should be executed when you start the program. It contains environment variable definitions for OpenFCST. This file can be copied directly to your `.bash_profile` so that the variables are always defined. The definition of environment variables is needed for successful passing of all tests. To source the file, type the following:

```
1 . fcst.env.sh
```

The second file, `run_tests`, is a script used to execute all examples in the example folder. The results are compared to pre-computed results in order to make sure OpenFCST is running correctly on your system.

Chapter 5

Pre-processor

In order to generate a fuel cell domain using OpenFCST, two options are available:

- use the classes under `FuelCellShop::Geometry` namespace;
- read in a mesh generated with an open-source mesh generator such as SALOME.

OpenFCST also allows you to generate VTK meshes from stacks of images obtained with FIB-SEM, TEM, etc. This functionality is available through PythonFCST, a Python extension of OpenFCST.

5.1 FuelCellShop::Geometry Namespace

Namespace `FuelCellShop::Geometry` contains classes to generate a cathode and anode fuel cell electrodes and a membrane electrode assembly with five or seven layers (i.e. with and without microporous layer). To use these classes, you simply need to create an object of the class. Then, use the `declare_parameters` member function to define the variables required in the input file, initialize the object calling `initialize` and generate the grid using `generate_grid`. For example,

```
1 //Create object
2 FuelCellShop::Geometry::PemfcMPL<dim> grid;
3 //Declare the necessary variables in the ParameterHandler deal.ii object
4 grid.declare_parameters(param);
5 //Once the ParameterHandler object has been initialized by reading from file,
6 //initialize the geometry variables
7 grid.initialize(param);
8 //Generate the mesh and store it in the dealii::Triangulation variable tr
9 grid.generate_grid(*this->tr);
```

You can use some pre-defined geometry templates using OpenFCST GUI or the parameter files. The list of options available for the `FuelCellShop::Geometry::GridBase` class is as follows:

- **GridExternal** - import external mesh;
- **Cathode** - catalyst and gas diffusion layers on the cathode side;
- **Anode** - catalyst and gas diffusion layers on the anode side;
- **CathodeMPL** - same as **Cathode**, but with a microporous layer;
- **Pemfc** - combines **Cathode** and **Anode** and adds a polymer electrolyte membrane inbetween;
- **PemfcMPL** - same as **Pemfc**, but with microporous layers on both cathode and anode sides.

5.2 Developing a mesh in SALOME

SALOME is an open-source cross-platform software that provides a generic platform for pre-processing. It is distributed under the terms of the GNU LGPL license. You can download both source code and executable files from the [SALOME website](#).

5.2.1 Tutorial

This short tutorial demonstrates how to create a simple mesh in Salome 7.3.0, define material and boundary indicators, and adapt all of this to the needs of OpenFCST.

The object we would like to mesh is represented by a two dimensional H-shaped domain as shown in Figure 5.1.

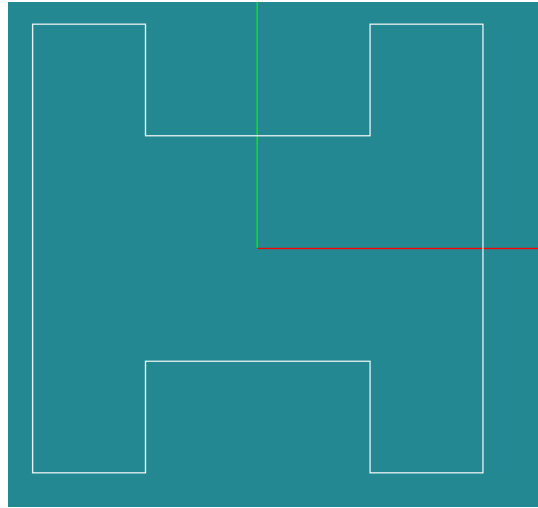


Figure 5.1: H-shaped domain.

OpenFCST only accepts meshes composed of either quadrilaterals in 2D or hexahedrons in 3D. **OpenFCST assumes that all the geometrical dimensions are in centimeters.** The current version of Salome is only able to produce these type of meshes with geometries that have an outer boundary composed exactly of 4 pieces in 2D, e.g. see Figure 5.2, and 6 pieces in 3D, e.g see Figure 5.3. In order to increase the quadrilateral and hexahedral properties of Salome however, a commercial package called Hexotic is distributed by Distine (for more information, please visit the [site](#)).

The two dimensional H-shaped domain shown in Figure 5.1 has 12 pieces for the outer boundary and hence can not be meshed in Salome directly by means of quadrilaterals. We can mesh the domain however by splitting it into 3 parts, such that each of the parts has 4 outer boundary segments. Then, we mesh each of these parts and combine them into the H-shaped domain.

Let us do this step by step.

Creating a geometric entity

Run Salome, select **New document**, and then select **Geometry** on the upper toolbox (Figure 5.4).

We are now in the Geometry module of Salome, and the first thing we need to do is to define 12 vertices along the object:

- 1(-1, -1)
- 2(-0.5, -1)

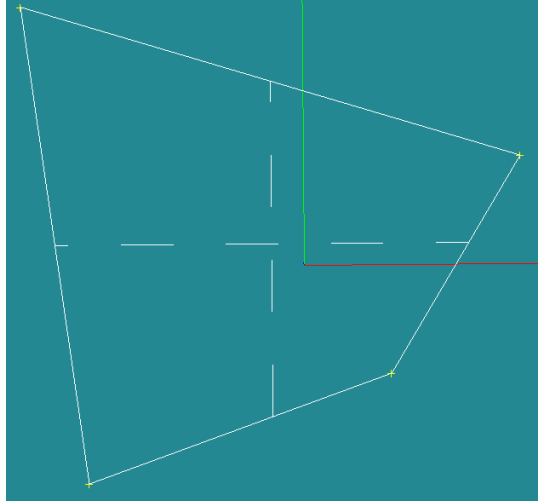


Figure 5.2: Linear quadrilateral.

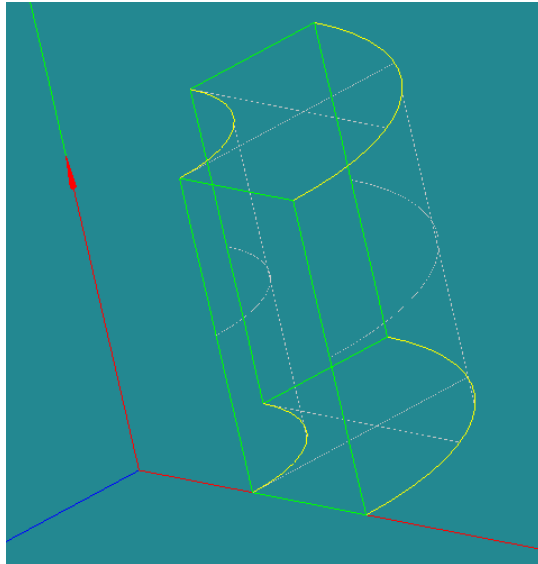


Figure 5.3: Quarter of cylindrical shell.

- 3(-0.5, 1)
- 4(-1, 1)
- 5(-0.5, -0.5)
- 6(0.5, -0.5)
- 7(0.5, 0.5)
- 8(-0.5, 0.5)
- 9(0.5, -1)

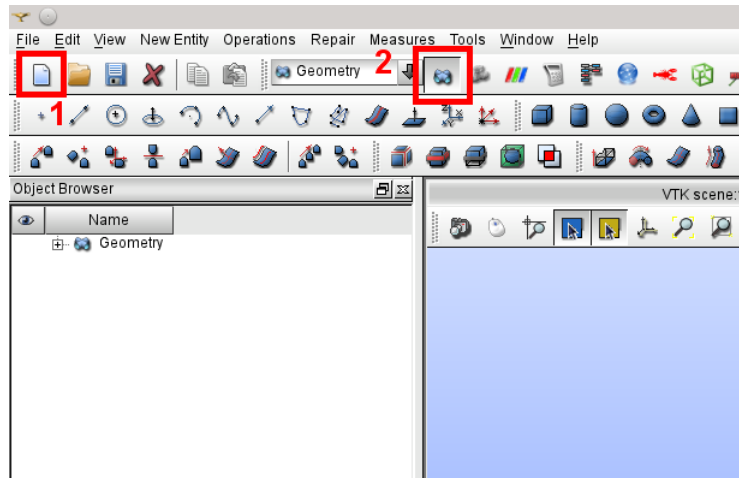


Figure 5.4: Starting a New Project.

- 10(1, -1)
- 11(1, 1)
- 12(0.5, 1)

To create any of these points, we go to **New Entity** → **Basic** → **Point**, specify a **Name** and the respective fields **X**:, **Y**:, and **Z**:, then select **Apply**. For the last vertex use the **Apply and Close** button instead of Apply. Note that instead of **New Entity** → **Basic** → **Point** we can simply choose **Create a point** on the upper toolbox. After initializing all the points select the -OZ view button to change the view and zoom into our geometry (Figure 5.5).

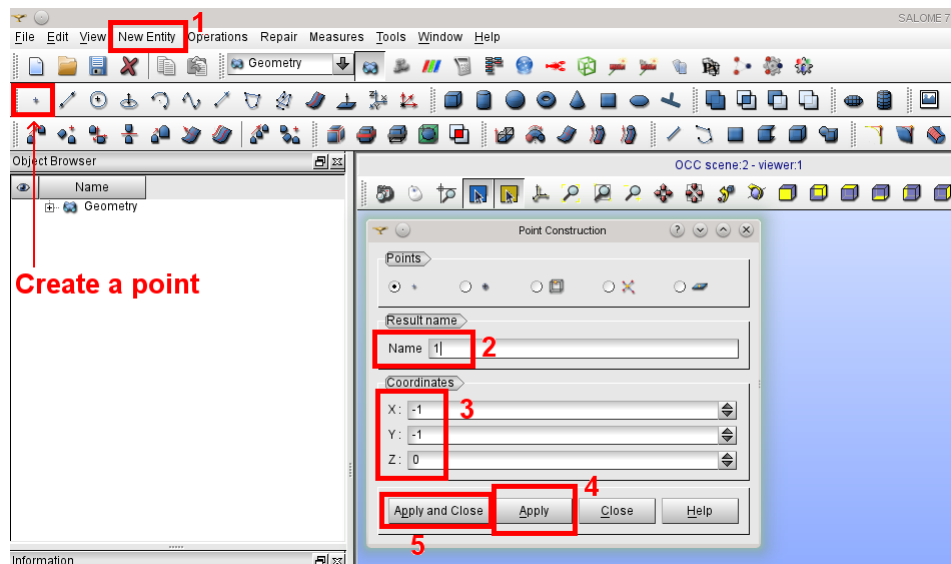


Figure 5.5: Creating a Point.

Next we create 3 quadrangle faces. Each of these faces consists of 4 points:

- 1(1, 2, 3, 4)
- 2(5, 6, 7, 8)
- 3(9, 10, 11, 12).

To create a quadrangle face, we go to **New Entity** → **Blocks** → **Quadrangle Face**, fill out the fields **Vertex 1**, **Vertex 2**, **Vertex 3**, and **Vertex 4** with the points from above, and select **Apply and Close** button (Figure 5.6). To prevent possible problems always specify vertices in a counter-clockwise direction.

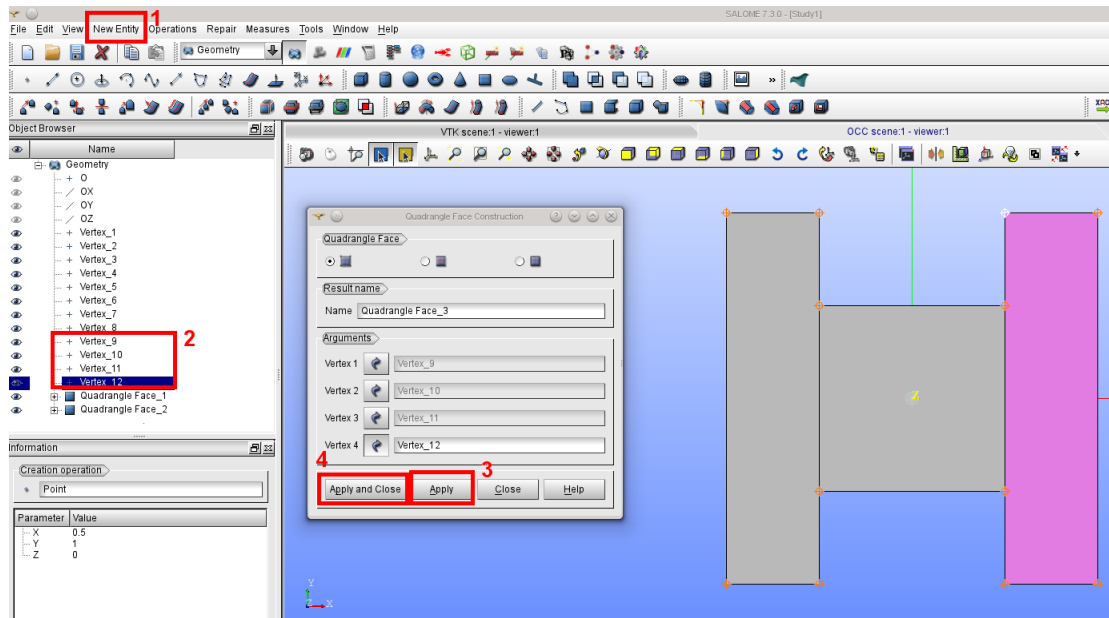


Figure 5.6: Creating a Quadrangle Face.

Generating a mesh for each domain

At this point we have created our geometry. We now would like to generate the mesh. For this, we switch our attention to the Mesh module. To enter the Mesh module, select the **Mesh** button on the upper toolbox. To create an appropriate mesh on each of the quadrangle faces, we go to **Mesh** → **Create Mesh**, where we pass the respective quadrangle face to the **Geometry** field. Once this is done, we select **Quadrangle (Mapping)** from the drop-down menu of the **Algorithm** field (Figure 5.7).

After that we select the **1D** tab and check that **Algorithm** is set up to **Wire discretization**. The number of 1D hypotheses is available here. We choose the one which is called **Local Length**. This method uses a uniform spacing between nodal mesh points to generate the mesh on the selected edge. Set the **Length** parameter to 0.5 as shown in Figure 5.8.

After clicking **OK**, the name of the **Hypothesis** field should change to **Local Length_1**. Then **Apply and Close**. Right click on the **Mesh_1** and **Compute**. After applying this strategy to all the quadrangle faces and selecting the -OZ view button again, we have something like Figure 5.9.

At this point we have generated one mesh for each geometrical entity. These meshes still need to be combined into an overall mesh. Furthermore, we might want to identify each geometrical entity and boundary with an indicator so that we can apply different properties in each domain.

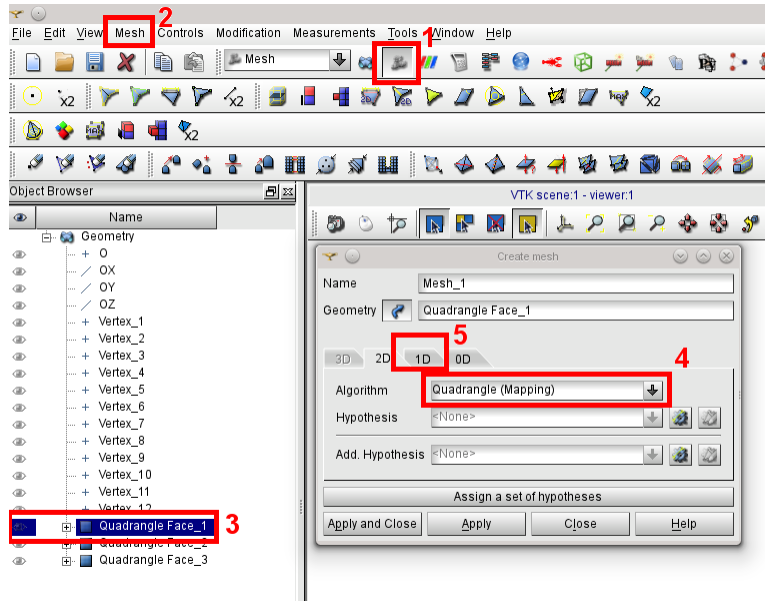


Figure 5.7: Creating a Mesh.

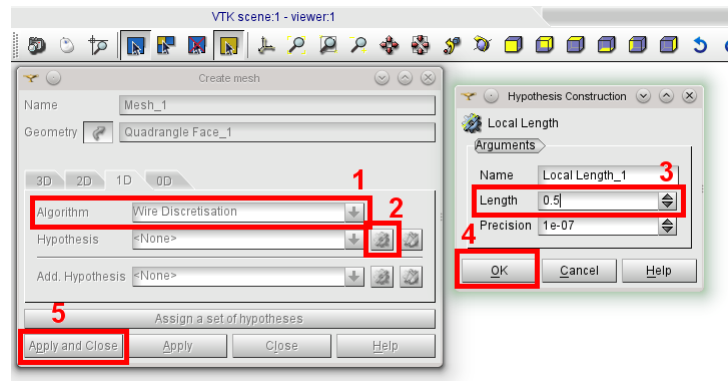


Figure 5.8: Setting Mesh Parameters.

Assigning material and boundary IDs to different parts of the mesh and creating an overall mesh

Let us now assign Material IDs to all the cells we have created. **Material IDs in OpenFCST need to be unsigned integers**, so please do not use names as material IDs. Let us assign all the cells of Mesh.1 a Material ID = 1, those belonging to Mesh.2 a Material ID = 2, and cells from Mesh.3 a Material ID = 3. To assign the material IDs, right click on **Mesh_1**, then select **Create Group**. On this dialog box specify an **Element Type** of **Face**, **Name** of 1, **Select All**, and finally **Apply and Close** (Figure 5.10). Repeat this process for the other Material IDs.

Tip: If you wish to use the **Manual Selection** option to add the cells to the **Id Elements** field, push and hold the Shift key on your keyboard and select the cells by left clicking. After all the desired cells have been highlighted, select **Add** in the Create Group dialog window.

Now create a Compound Mesh by simply merging all the previously created meshes. This is done by selecting all the meshes, then from the toolbox **Mesh** → **Build Compound**. Specify a **Name**, select

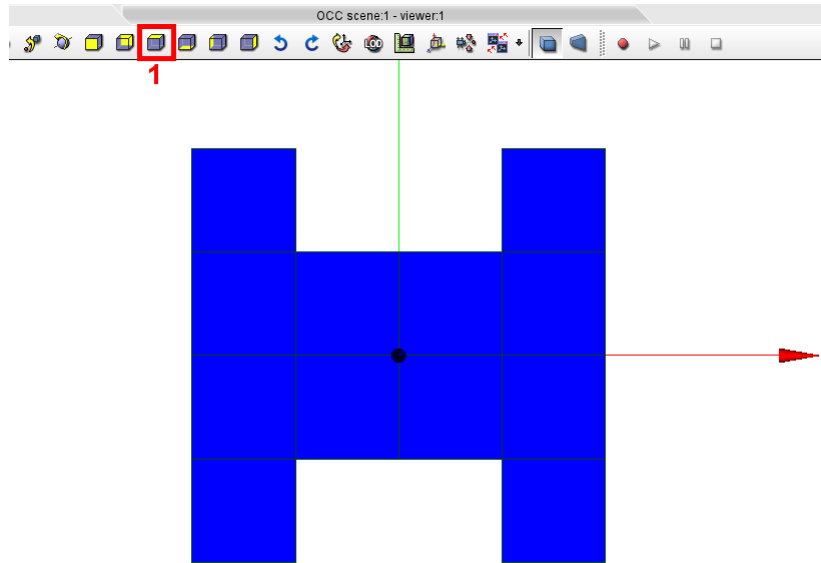


Figure 5.9: Overview of the Mesh.

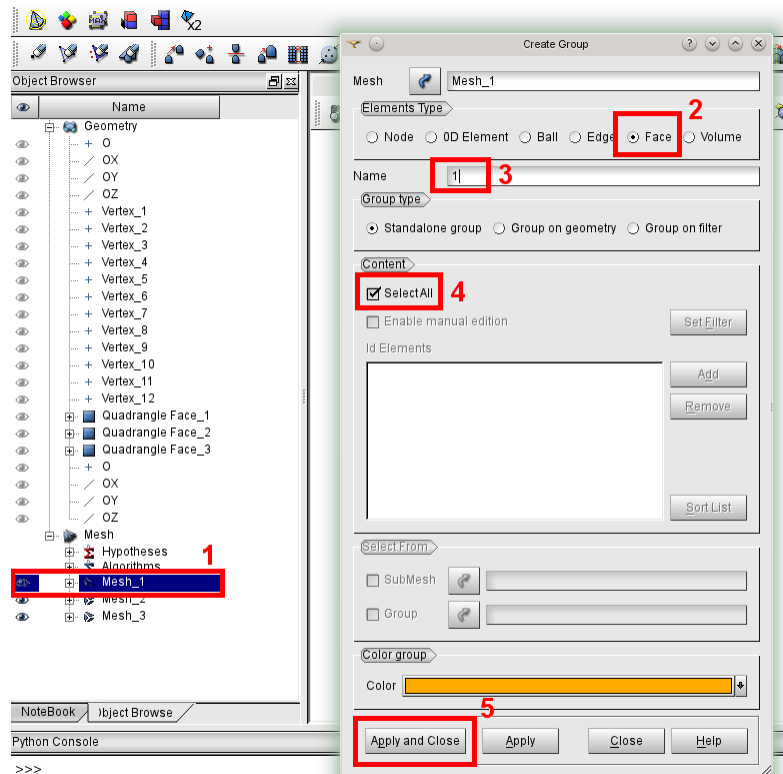


Figure 5.10: How to Set the Material ID.

Create common groups for initial meshes and Merge coincident nodes and elements, and finally select **Apply** and **Close** (Figure 5.11).

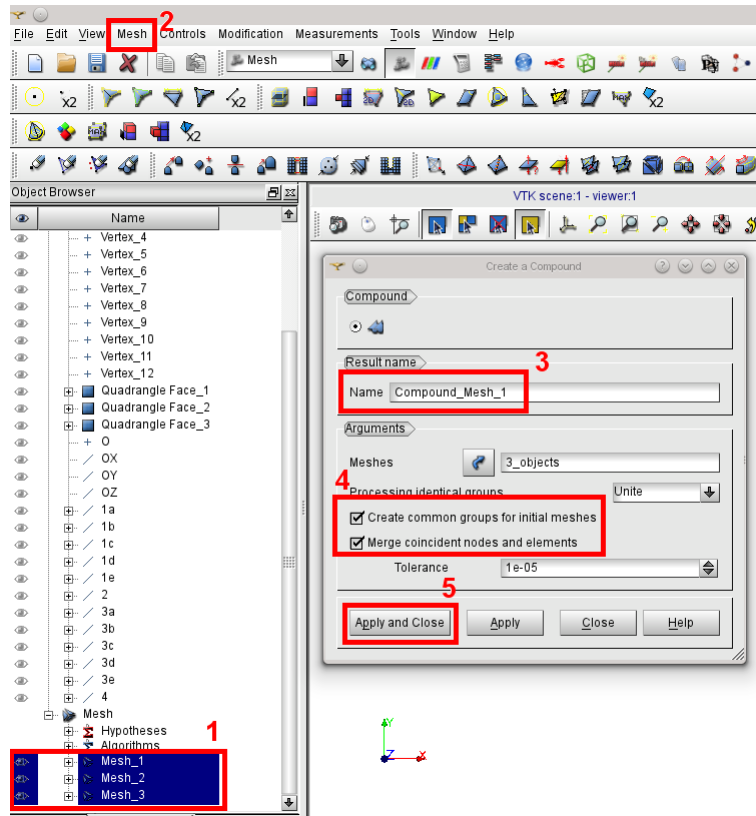


Figure 5.11: Building a Compound Mesh.

In the **Object Browser** expand the **Compound_Mesh.1** and delete all groups except for the Material IDs in the **Groups of Faces** (Figure 5.12).

The same technique as the above tip can be used to define the Boundary IDs by selecting them by hand. Another method that is easier for more complex shapes with finer meshes can be done as follows. Go back to **Geometry** and create lines around the Boundary. Like vertices this is done by going **New Entity** → **Basic** → **Line**. Change **Name** to whatever is easiest for you, I prefer to give them some name related to the Boundary ID they represent and use a letter afterwards if multiple lines represent the same Boundary ID. Select the vertices along the line, then select **Apply**. The vertex in Point 2 will then move to Point 1 and you can then continue this process in a clockwise or counter-clockwise direction. Once you reach the last line you can then select **Apply and Close** (Figure 5.13).

Back in **Mesh** right click the **Compound_Mesh.1** and select **Create Group**. Select an **Element Type** of **Edge** and change the **Name** to the designated Boundary ID. Select **Enable manual edition** and then select **Set Filter**. In the case of Boundary ID 1 we have 5 lines so press **Add** 5 times then change **Criterion** to **Belong to Geom**, **Binary** from **And** to **Or**, and for **Threshold value** select the empty square then select one of the lines from the **Object Browser**. Once all the lines for the filter are set, select **Apply and Close**. Back in the **Create Group** window select **Add** and then **Apply and Close** (Figure 5.14). Repeat this process for each Boundary ID.

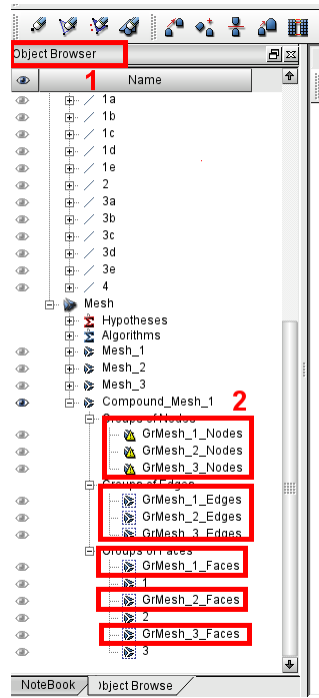


Figure 5.12: Deleting Unnecessary Groups.

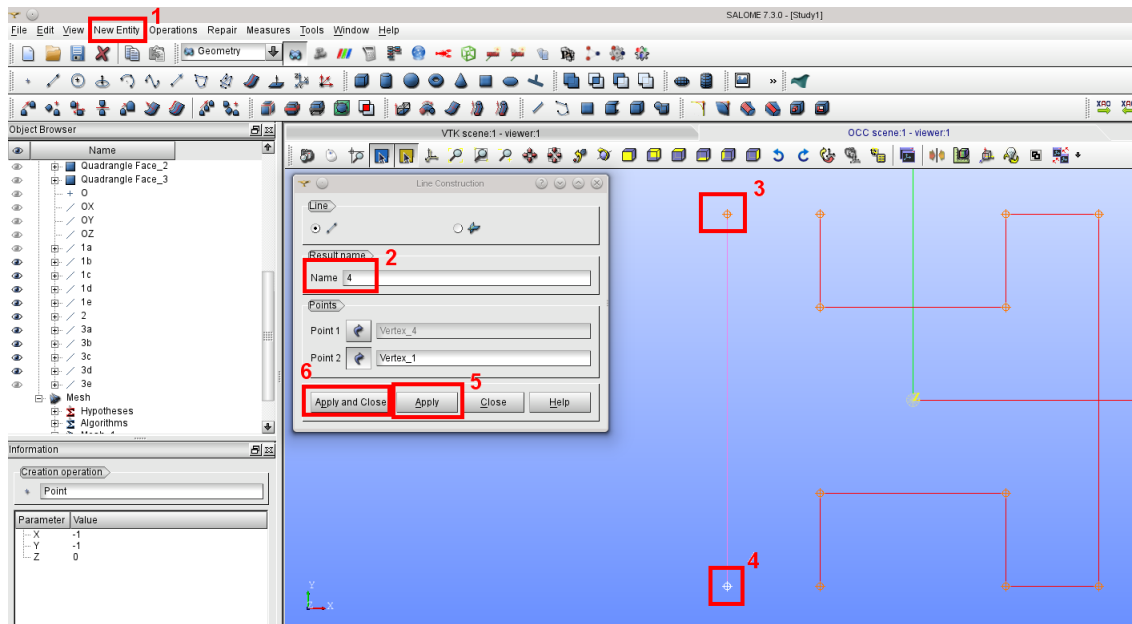


Figure 5.13: Creating Lines for Use in Boundary ID Filter.

Removing internal edges

Internal edges (and faces in 3D) are strictly prohibited by the OpenFCST architecture. To see these internal edges, left click the Compound Mesh then right click the picture of it in the view. Then in **Numbering**

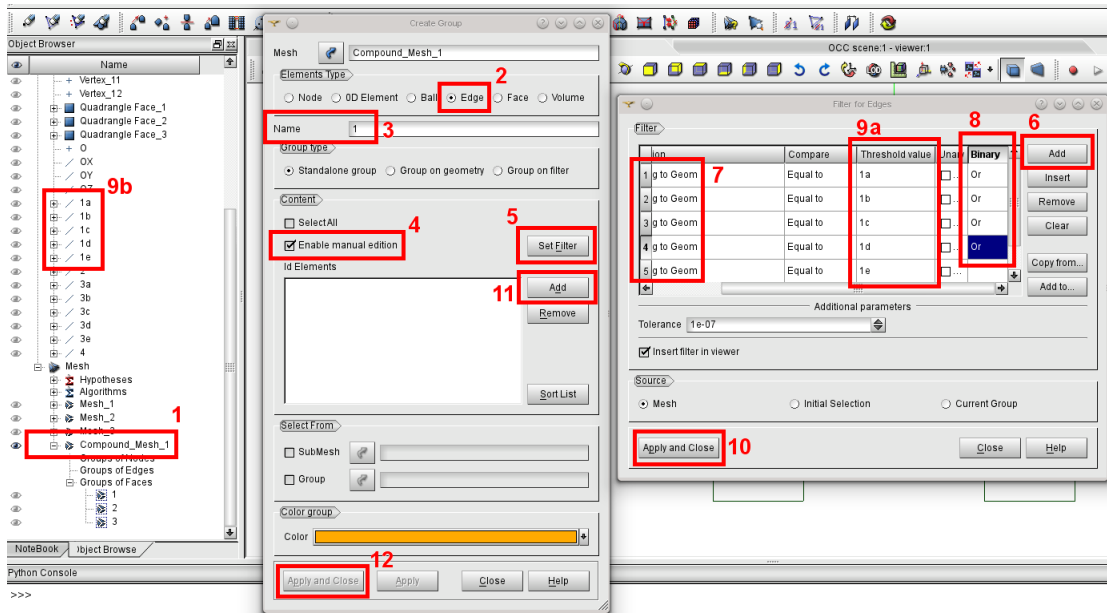


Figure 5.14: Creating a Boundary ID Filter.

select **Display Elements #**. This will display all the edge and face numbers. As shown in Figure 5.15, it can be seen that lines 3, 4, 17, and 18 must be deleted.

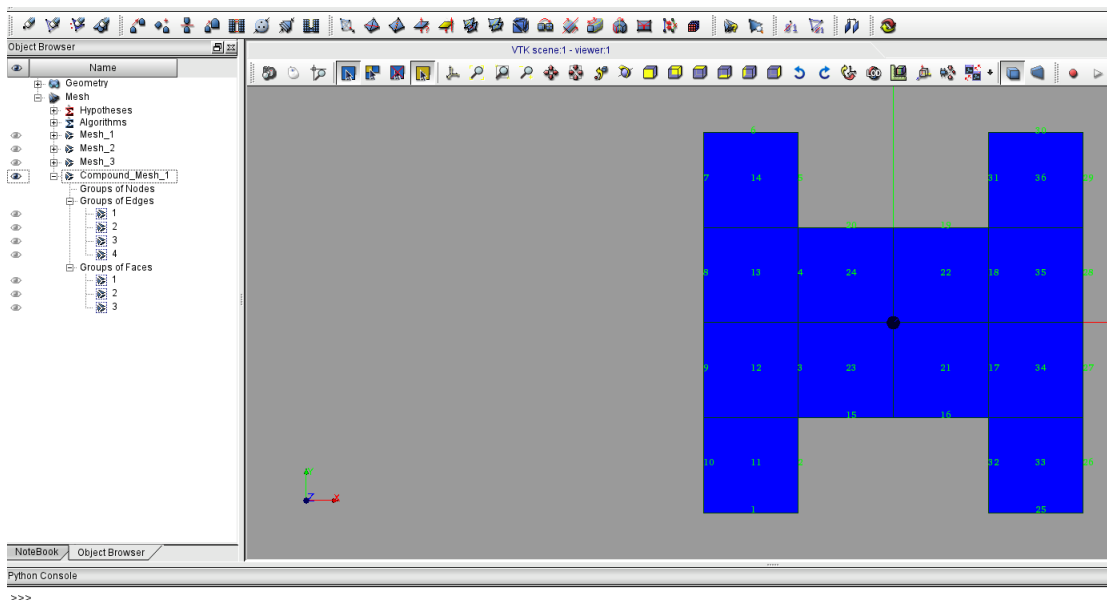


Figure 5.15: Overview of the Mesh with Internal Edge Numbering (NOT allowed).

To manually remove these internal edges, in the toolbox select **Modification** → **Remove** → **Elements**. You can then either select the edge or enter the numbers and then select **Apply and Close**. This is rather easy in this simple project, but it can become cumbersome in more complex meshes. In openfc-

st/pre-processing, there are two python scripts, RemoveInternalEdges.py and RemoveInternalFaces.py. In our case we need to open RemoveInternalEdges.py, change the mesh_name on line 12 to the name of our Compound Mesh and save. Back in SALOME select **File** → **Load Script** select the RemoveInternalEdges.py, and it will automatically remove all internal edges.

Exporting the mesh to UNV format

Now the mesh can be exported to a UNV file. Right click the Compound Mesh in the **Object Browser**, then select **Export** → **UNV file**. Finally, save your mesh. Once we export the whole mesh into an UNV file, we can use it for the computational purposes (see the respective OpenFCST tutorial).

Tip: Sometimes issues are caused by the first few lines of the UNV file when importing it to OpenFCST. To prevent this you can delete the first 17 lines of the UNV file so the file actually begins at line 18. The beginning of these UNV files all look similar to the following:

```

1      -1
2      164
3      1 SI: Meter (newton) 2
4      1.0000000000000000E+0 1.0000000000000000E+0 1.0000000000000000E+0
5      2.7314999999999998E+2
6      -1
7      -1
8      2420
9      1
10     SMESH_Mesh
11     1 0 0
12     Global Cartesian Coordinate System
13     1.0000000000000000E+0 0.0000000000000000E+0 0.0000000000000000E+0
14     0.0000000000000000E+0 1.0000000000000000E+0 0.0000000000000000E+0
15     0.0000000000000000E+0 0.0000000000000000E+0 1.0000000000000000E+0
16     0.0000000000000000E+0 0.0000000000000000E+0 0.0000000000000000E+0
17     -1
18     -1
19     2411
20     ...

```

5.3 SALOME meshing using python scripts

5.3.1 Introduction

The previous section discussed meshing in SALOME using the graphical user interface (GUI). This section will focus on creating and running scripts to create meshes and geometries. Reasons for using scripts instead of the GUI are as follows: improved repeatability of results, significant time saving due to automation, and removal of human error. Several Python scripts included in the pre_processing folder can be used to create various meshes of various geometries.

Meshing scripts are run through SALOME's text user interface (TUI). Loading scripts can be done simply via the File drop down menu, Load Script. Meshing scripts are written in Python programming language. Python is a very popular general purpose high level programming language. Unlike C++, Python code is not precompiled, but interpreted at run time by a Python interpreter. Some key features that make Python popular are its simple yet elegant syntax, dynamic typing, automatic memory management, and large selection of freely available libraries. If you are interested in learning the Python programming language, we recommend [Dive Into Python](#).

5.3.2 Scripting Examples

```

1 import smesh, geompy, SMESH
2 import SALOMEDS

```

The above lines import the necessary SALOME packages that will be required to create geometries and meshes. *smesh* is used to create python mesh objects, *geompy* is used for creating geometries. The other two packages contain constant flags. For more detail, please see the following resources:

1. [smesh functions](#);
2. [geompy documentation](#);
3. [Salome TUI documentation](#).

The following simple example shows how to use geompy to create a simple geometry and then mesh it using smesh.

```

1 def makeRectanglarMesh(self, width, height):
2
3     #Create vertices to describe rectangle
4     Vertex_1 = geompy.MakeVertex(dList[0], dList[1], dList[2])
5     Vertex_2 = geompy.MakeVertex(dList[0] + width, dList[1], dList[2])
6     Vertex_3 = geompy.MakeVertex(dList[0] + width, dList[1] + height, dList[2])
7     Vertex_4 = geompy.MakeVertex(dList[0], dList[1] + height, dList[2])
8
9     #Make rectangle geometries
10    rect = geompy.MakeQuad4Vertices(Vertex_1, Vertex_2, Vertex_3, Vertex_4)
11
12    #Create mesh object of rectangular geomtery
13    Mesh_1 = smesh.Mesh(rect)
14
15    #Set 1D meshing algorithm
16    Regular_1D = Mesh_1.Segment()
17    Local_Length_1 = Regular_1D.LocalLength(self.meshDensity)
18    Local_Length_1.SetPrecision( 1e-07 )
19
20    #Set 2D meshing algorithm
21    Mesh_1.Quadrangle()
22
23    #Compute and return
24    Mesh_1.Compute()
25    return Mesh_1

```

The following is an example of modifying meshes and using mesh filters.

```

1 def delInternalEdges(self):
2     'This function deletes internal edges from self.compoundMesh'
3
4     #Create a search filter to find free borders of the mesh
5     search_filter = smesh.GetFilter(smesh.EDGE, smesh.FT_FreeBorders)
6     external_edges = self.compoundMesh.GetIdsFromFilter(search_filter)
7
8     #Get a list of all edges
9     all_edges = self.compoundMesh.GetElementsByType(SMESH.EDGE)
10
11    edges_to_remove = []
12
13    #The difference between the external_edges list and all_edges list will be the internal edges.
14    #The following loop iterates through the all_edges list, comparing it wil the external_edges list.
15
16    for b in all_edges:
17        if b in external_edges:
18            pass
19        else:
20            #The edge is internal, add it to the list of items to be removed from the mesh
21            edge_to_remove.append(b)
22
23    print "Removing internal edges:"
24    print edges_to_remove
25
26    #Remove the edges from the mesh
27    self.compoundMesh.RemoveElements(edges_to_remove)

```

When developing a new Python function for generating a geometry or mesh in SALOME one may obtain a rough solution by following these steps:

1. open the SALOME GUI;
2. perform the necessary steps using the GUI to generate the desired geometries and/or surfaces;
3. use the “Dump Study” facility, accessed from the File menu, to produce a bulky but complete python program for the previously performed steps;
4. refine the script to the desired form.

This is a very good method for obtaining an initial coding solution or examples of correct code syntax and usage.

5.4 Generating a VTK mesh using PythonFCST

5.4.1 Using the Python executable writeVTK.py

PythonFCST comes equipped with a Python executable **writeVTK.py** which reads in a stack of tiff images and generates a Legacy Unstructured Grid file using the pixel values in the images as the material ids for the mesh. Before running **writeVTK.py**, it is important that OpenFCST is installed on the system and the OpenFCST environment variables are set using the **fcst_env.sh** file in the FCST Install Directory ($\${FCST_DIR}$). To set the environment variables, type the following command in the Terminal (or Konsole) from $\${FCST_DIR}$:

```
1 $ . fcst_env.sh
```

Once the environment variables have been set, the **writeVTK.py** should be available as a command line utility. To ensure that the environment variable has been set, the following command can be used:

```
1 $ writeVTK.py -h
```

The above command generates the help documentation for the writeVTK.py:

```
1 Usage: writeVTK.py [options]
2
3 Options:
4 -h, --help            show this help message and exit
5 -f FILE, --file=FILE  relative path of the images to be converted
6 -b BASENAME, --basename=BASENAME
7                       basename of the image files. Default: 'Slice_' would
8                       mean images with name Slice_001 and so on
9 -n NUM                Number of images to be read. Default: 200
10 -o OUTPUT             Output file name. Default: test.vtk
11 -m MATERIAL, --material=MATERIAL
12                       material id of the phase to be meshed. Default: 0
13 -v VOXEL, --voxel=VOXEL
14                       voxel size as x y z
15 -e EXTENSION, --extension=EXTENSION
16                       extension for the input files. Default is .tiff
```

In order to generate the mesh, the relative folder location from the current working directory for the images needs to be specified with the **-f** identifier. The basename of the images refers to the name of the image without the numerical part and extension. For example, for images named “**Slice_001.tiff**” to “**Slice_100.tiff**” the basename would be “**Slice.**”. The number of images to be read in is specified with the **-n** identifier. Identifier **-o** is used to specify the output file name. The voxel size, which is the dimension in cm per pixel in the X, Y, and Z directions, is specified with the **-v** identifier. By default, the extension for the images is “**.tiff**”. If the images have a different extension, then they must be specified with the **-e** identifier. Another important option is the material ID which can be specified with the **-m** option. The material ID represents the pixel value for which the mesh should be generated. The default material ID is 0 which corresponds to the black color in a grey-scale image.

In the following example, we will generate a mesh from 10 images in the folder $\${FCST_DIR}/examples/microscale/3D_Diffusion/images$ while the current working directory is $\${FCST_DIR}/examples/microscale/3D_Diffusion$:

```

1 $ writeVTK.py -f images/ -b Slice_ -n 10 -o my_test.vtk -m 0 -v 1e-6 1e-6 1e-6 -e .tiff
2
3 =====
4 =====
5 = Points and cells Created
6 = Writing a VTK file
7 =====
8 Calculating Fitted Sphere Size Distribution
9 *****
10 = Processing dataset
11 -----
12 =====
13 = Calculate PSD
14 = - Pore radius 1e-06
15 = - Pore radius 1.1643739803e-06
16 = - Pore radius 1.32874796059e-06
17 = - Pore radius 1.49312194089e-06
18 = - Pore radius 1.65749592118e-06
19 = - Pore radius 1.82186990148e-06
20 = - Pore radius 1.98624388177e-06
21 = - Pore radius 2.15061786207e-06
22 = - Pore radius 2.31499184237e-06
23 = - Pore radius 2.47936582266e-06
24 = - Pore radius 2.64373980296e-06
25 = - Pore radius 2.80811378325e-06
26 = - Pore radius 2.97248776355e-06
27 = - Pore radius 3.13686174384e-06
28 = - Pore radius 3.30123572414e-06
29 = - Pore radius 3.46560970443e-06
30 = - Pore radius 3.62998368473e-06
31 = - Pore radius 3.79435766503e-06
32 = - Pore radius 3.95873164532e-06
33 = - Pore radius 4.12310562562e-06
34 = The file has been successfully written!
35 -----
36 Phase with material id: 0 is represented by material id 0 in the mesh file: my_test.vtk
37 time elapsed is 17.7439901829 seconds

```

If the command runs successfully, the above output will be displayed and the mesh file named **my_test.vtk** will be written to the `${FCST_DIR}/examples/microscale/3D_Diffusion` (which was the current working directory). In addition to generating the mesh, the code also calculates the local radius based on a sphere fitting algorithm [9] of the phase and stores it in the generated mesh which can be later used in OpenFCST to account for the local Knudsen effects in the diffusion simulations.

5.4.2 Using the MESH module in PythonFCST

For people who are familiar with Python, using the **PythonFCST.mesh** module might be another option to generate the mesh. The mesh module contains the Grid Generator class which generates the VTK mesh. The arguments to be supplied are a 3D numpy array (argument name: **image**) and the output filename (argument name: **filename**). This provides a bit more flexibility to use any 3D numpy array which might have been generated from any source rather than being limited to a stack of tiff images.

For the purpose of this example, we would deal with the same set of images as above. Let's use the **PythonFCST.mesh** module to generate the mesh. The code snippet below describes how to use it.

```

1 #Initial imports for libraries and modules
2
3 import PythonFCST.mesh.PhaseGenerator as grid
4 from PIL import Image
5 import os
6 import numpy as np
7
8 #Defining variables to read in images; Omit this section if you already have a 3D array
9
10 foldername = "images"
11 basename = "Slice_"
12 num_images = 10
13 output = "my_test.vtk"
14 name=os.getcwd()+ '/' +foldername+ '/' +basename+'%.3d'
15 extension='.tiff'
16 tmp = []

```

```

17
18 #Looping over images and reading them
19 for i in range(num_images):
20     filename=name%(i+1)+extension
21     tmp.append(np.asarray(Image.open(filename),dtype=np.float))
22
23 #Generating the 3D numpy array
24 image=np.dstack(tmp)
25 image[image>0]=20 #changing all non-zero material ids to 20; assuming that pixel value of 0 was the one
26                   of interest
27
28 #Generating the VTK mesh using the Mesh module in PythonFCST
29 o=grid(image,output,material=0)
30 o.write()

```

The generated mesh file, **my_test.vtk**, has the material ID 0 from the grey scale images extracted. Figures 5.16a and 5.16b show the results of the example code discussed above.

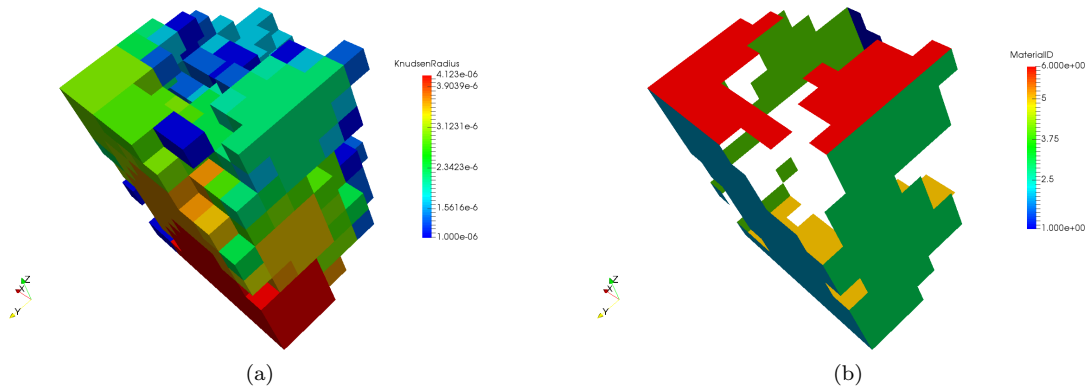


Figure 5.16: The resulting mesh of the pores (a) and representation of the boundary IDs 1-6 for the generated mesh (b).

Note. VTK files invert the X and Y axis from the images. Therefore, if the image had dimensions 20x30 in X and Y directions respectively, then the mesh will have dimensions 30x20 in the X and Y directions. By default, OpenFCST corrects for this so that the voxel sizes entered by user are in the true X and Y directions. Also, the boundary IDs are corrected so that 1-2 are boundary IDs in the true X direction and in the Y direction for the mesh as seen above.

The mesh can be further pre-processed using any of the filters in Paraview or Mayavi and can be written to a Legacy VTK file. This would still be compatible with the OpenFCST application.

Chapter 6

Running OpenFCST

6.1 Setting up a simulation in OpenFCST

In order to setup a simulation in OpenFCST, two parameter files are required. Note that the xml files are used with the graphical user interface (GUI) while prm files are used if you execute the program through terminal. The required files are:

1. **main.prm** (or **main.xml**): This is the file that is first specified in the OpenFCST GUI. When calling OpenFCST from the command line, this is the only file passed as an argument. This file is used to specify the type of simulation/application that the user would like to run. The other files are constructed based on the options in this file.
2. **data.prm** (or **data.xml**): This file contains all the information regarding the computational domain, boundary and operating conditions (*Relative Humidity, Temperature, ...*), equations, solution strategies and post-processing options to solve the simulation. It also specifies all the parameters that describe the fuel cell physical properties (*Porosity, Platinum & Nafion Loading, ...*).

Sometimes we refer to these files as an OpenFCST project.

These files are used to specify the type of problem, and the fuel cell parameters necessary to run a simulation in OpenFCST. To run OpenFCST, either the GUI is used with the xml files or the program can be called from the command line (with the prm files) using the following command:

```
1 $~/Openfcst/Install/bin/> ./fuel_cell-2d.bin main.prm
```

The main file already contains the path to the other files, so only this file is needed. Upon issuing this command OpenFCST will select the desired simulation to run, read the fuel cell parameters from the different files, and then solve the required finite element problem. The program flowchart is given in Figure 6.1. The output of the program is shown on screen and also recorded in a file, i.e., **logfile.log**.

In the next sections, first the OpenFCST graphical user interface is discussed. Then, the main and data files are discussed in detail. The files with extension .xml are read with the OpenFCST GUI. The files with extension .prm are ASCII files. The latter contain the same information, but they are easier to read in a text editor. OpenFCST can convert .prm files to .xml files easily. If you have a folder with a main.prm and a data.prm files (the data file name can be anything, but it must be specified in the main.prm file), then you can call OpenFCST as follows:

```
1 $~/Openfcst/Install/bin/> ./fuel_cell-2d.bin -c main.prm
```

Then main and data files in XML format will be generated based on the main.prm file provided. The XML files can then be visualized and modified with the OpenFCST GUI.

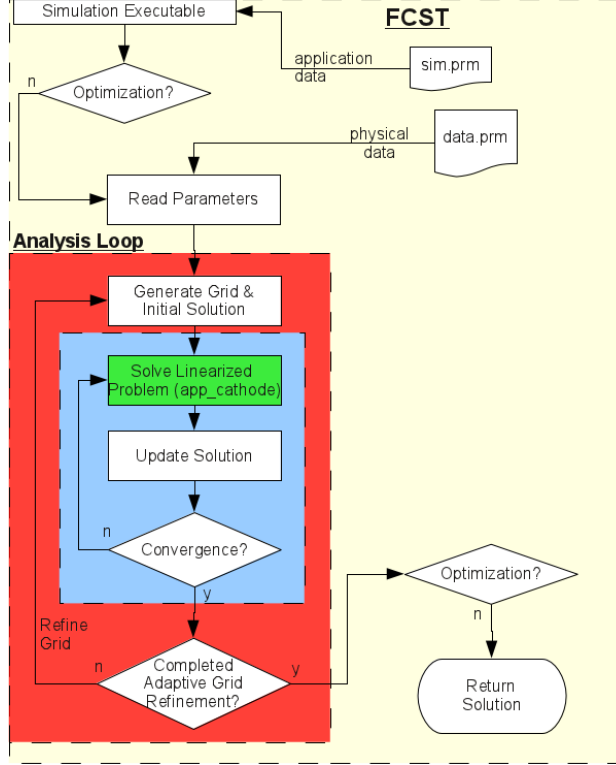


Figure 6.1: Schematic of Fuel Cell Analysis Code.

6.2 OpenFCST examples

The folder **examples** in the **Install** folder in OpenFCST contain an HTML manual with several examples. The examples are part of our testing suite to make sure that OpenFCST continues to provide the same results as new functionality is added. Each sub-directory in **examples** contains an explanation of the problem that is being solved as well as the data files to obtain the results. The easiest way to get started with OpenFCST is to run the example cases. We suggest that you do not modify the files in the examples folder, instead copy them to a different sub-folder such as **my_data** so that you can still test your code with **run_tests**.

The folder **examples/** contains .prm files. We recommend that you convert them to .xml files and then visualize them in the OpenFCST GUI.

6.3 OpenFCST's graphical user interface

The OpenFCST graphical user interface (GUI) allows one to create, configure, and run simulations within a single application. All the OpenFCST simulation parameters are viewable and editable through the GUI. Using the GUI, one can browse through the hierarchy of parameters to edit numerous aspects of a simulation. Parameter descriptions are displayed by mousing over a parameter name. Once suitable simulation parameters have been selected, a simulation can be run from within the GUI. Simulation output such as text logging and a list of generated files are displayed within the GUI in a convenient manner.

One may create a new project, whereby default parameter files will be created in a step-by-step process. Alternatively, one can load an existing project. A project is made of two to three files:

- **main.xml** contains the main selections for OpenFCST, such as the type of application, nonlinear solver, and type of study to be performed, i.e. one analysis run or a parametric analysis run.

- **data.xml** contains the parameters to setup the simulation for the selected application.

These files are discussed in more detail later in this guide.

Given the large number of parameters in a simulation, we recommend that new users start with a project from the **examples** directory since the step-by-step process requires an in-depth knowledge of the code. In order to generate a GUI project from the examples folder, go to an example folder such as **cathode/analysis**, and then generate the XML files from the parameter files using the OpenFCST:

```
1 $~/Openfcst/Install/examples/cathode/analysis> fcst2D -c main.prm
```

Using the **main.prm** file, the **.xml** files that constitute a project are generated. The following output will appear in the terminal:






```
1 Parameters: main.prm
2 Creating converted main file in XML
3 YOU ARE CURRENTLY SOLVING A CATHODE MODEL
4 Application->DoF->BlockMatrix->OptimizationBlockMatrix
5 ->FuelCellShop::Equation::NewFicksTransportEquation
6 ->FuelCellShop::Equation::ElectronTransportEquation
7 ->FuelCellShop::Equation::ProtonTransportEquation
8 ->FuelCellShop::Equation::ReactionSourceTerms
9 ->FuelCell::Application::AppCathode-2D
10 YOU ARE USING Newton3pp NEWTON SOLVER
11 Application->Copy->Newton3ppBase->->AdaptiveRefinement
12 Parameters: data.prm
13 Creating converted data file in XML
14 No opt file detected, skipping.
```

Then **main.xml** and **data.xml** files will be generated.

Once the files are generated, they can be loaded into the GUI.

6.3.1 Overview

The main components of the GUI shown in Figure 6.2 can be described as follows:

- **Menu:** This contains three sub-menus: File, Options and Help. From the File menu we can start a new project, open an existing project, save the current project, save a specific file from the current project, save the simulation log to a text file, and exit the GUI. The Option menu allows us to choose simulation dimensions, i.e., 2D or 3D, number of CPU's for running the simulation and file conversion, i.e., from xml to prn or vice versa.
- **Icon bar:** This contains several commonly used icons:  start a new project,  open an existing project,  save the current project,  close the current project,  settings for interface arguments, **2D** set the simulation to 2D, **3D** set the simulation to 3D.
- **Parameter editing panel:** This panel displays all the parameters of the loaded project. We can browse through the parameter hierarchy using the mouse to locate the parameter we desire. Descriptions/hints about a specific parameter are displayed by mousing over it.
- **Start/next/end button:** This button is used predominantly to progress our current project.
- **Application Output:** While running a simulation, text output describing the status of the simulation will be shown here.
- **Output Files:** While running a simulation, OpenFCST will produce several output files. Clicking on any item displayed in this list will open the file with the operating system default program associated to the file's type.

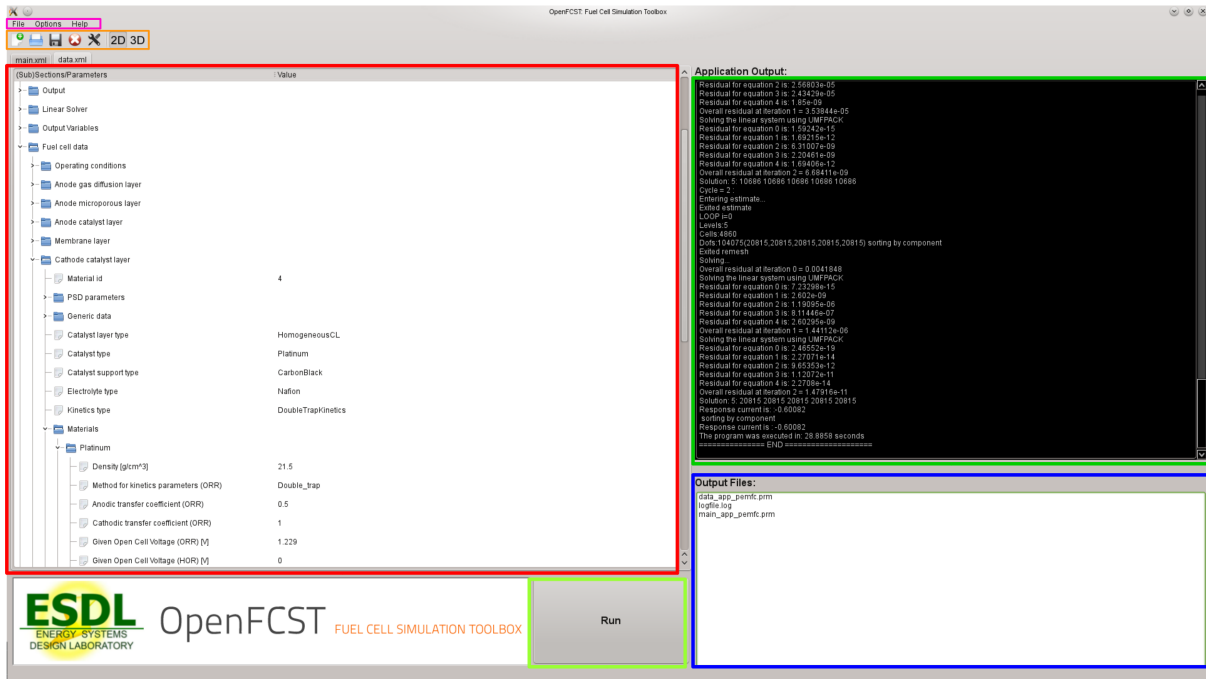



Figure 6.2: Main window of the OpenFCST graphical user interface. Components of interest are highlighted.

6.3.2 How To's

Start a New Project

Warning: Default parameter files created by OpenFCST version 0.3 require extensive modification in order to run a simulation. We suggest that users new to OpenFCST use the “Open Project” option explained later in this section.


Starting a new project from scratch using the GUI is simple:

1. First, select the “New Project” option from the **file menu** or click on the icon .
2. A dialogue will open asking you to select a folder, i.e. a “working directory”. This is the location where your project file and simulation output will be placed. **Warning:** placing a project in a folder with an existing project may cause some files to be overwritten.
3. Once a working directory has been selected, OpenFCST will generate a main parameter file. This file allows us to select important aspects of our simulation, such as the type of application we would like to run (Cathode simulation, non-isothermal application, etc) and the type of Newton solver we would like to use.
4. At this point, clicking the **Next** button will instruct OpenFCST to read our main file and generate a corresponding data file. A data file includes important parameters pertaining to simulation aspects such as equations, finite elements, mesh refinement, and fuel cell material parameters.
5. Now that we have edited our parameter files to our desire, clicking the **Run** button will start the simulation. Whilst the simulation is running, its status will be shown by the **Application Output** panel and a list of files generated by OpenFCST will be shown in the **Output File** panel.


6. If there are no errors, the simulation will run to completion.

Open an Existing Project

Opening an existing project using the GUI is a simple process, similar to starting a new one.

1. Select “Open Project” from the **file menu** or click on the icon .
2. You will be asked to specify the location of a **main** parameter file. Note: The directory of the main file will be used as the project’s working directory, i.e. the location where your project files and simulation output will be placed.
3. Once the location of the main file has been specified, you will be asked for the location of the **data** file.
4. Once the location of the data file has been set, you can load additional files. It is not necessary to load these additional files.
5. Now you can edit the loaded files and run the simulation by clicking the **run** button, in the same fashion as described earlier in this section.

6.3.3 Configuration

The version of the OpenFCST binary you are using may have unique interface arguments. To maintain functionality between the GUI and the OpenFCST binary the following parameters are editable by clicking on the GUI’s icon  or in its **settings.ini** file (which will be created in the same directory as the GUI once it is run). Typically, these parameters do not need to be edited.

- **OpenFCST binary path:** The path to the OpenFCST binary you wish to use to run your simulations.
- **OpenFCST used parameter during the simulation:** The command line argument which instructs OpenFCST to create a parameter file.
- **Number of CPUs used in parallel running:** The number of CPUs used when the simulation is executed.
- **mainFileName:** The name of the main file OpenFCST will create when generating a new Project.
- **dataFileName:** The name of the data file which OpenFCST will create when generating a new Project.

6.3.4 Reporting Errors

In the case of program errors or crashes occurring, please **contact us**.

Providing the following will help us to resolve your issue:

- a description of the circumstances in which the errors occurred and any error messages that you may have received;
- a copy of your parameter files (main.xml, data.xml, etc);
- the GUI log file (gui_log.txt);
- simulation log files (which can be produced using the graphical user interface, and found in the project’s working directory).

6.4 The OpenFCST main file

The **main** file is the initial file accessed by OpenFCST. The file contains two main subsections:

1. **Simulator**;
2. **Logfile**.

6.4.1 Simulator section

The **Simulator** section is used to setup the high-level parameters for the simulation such as the application to solve, the type of solver, and the type of results. This section is maybe the most important as it dictates the information that will appear in the data file for the GUI. Please note that some parameters cannot be modified once the data file has been generated. The parameters in this section are:

- **simulator name**: This entry specifies the type of simulation that you would like to perform, e.g., solve a cathode model (**cathode**), a full MEA model (**MEA**), an electrical conduction problem (**ohmic**). The data file is dependent on the selection of this parameter, therefore, if the default data file is generated with one value, it will not work with others.
- **simulator specification**: This entry is only used with fluid flow applications to specify different sub-problems based on boundary conditions.
- **solver name**: This entry specifies if the problem is linear (**Linear**) or non-linear. For the latter, the OpenFCST team has implemented several Newton solvers that can be used. Note that the data file should be re-generated if the type of solver is changed.
- **solver method**: This entry can be used to add an additional loop during the solution stage after the analysis loop (see Figure 6.1).
- **simulator parameter file name**: This entry should contain the name of the data file for the simulation. The data file should be in the same folder as the main file.
- **Analysis type**: This entry specifies if you would like to run a simulation for a specific cell operating condition, a polarization curve or a parametric study. The options for each one of these cases are specified in the later subsections. For **Analysis**, no further information is required. For the other cases, the necessary information is specified next.

Subsections **Parametric Study** and **Polarization Curve** are described next. For **Polarization Curve**, the following parameters can be specified:

- **Polarization curve file output**: This entry specifies the file where the polarization curve results should be stored;
- **Initial voltage [V]**: Voltage at which the first point in the polarization curve will be evaluated;
- **Final voltage [V]**: Voltage at which the polarization curve will be terminated. Note that if the value is set, for example, to 0.6 V, then the polarization curve will not include this voltage (it will run down to 0.6 V plus increment);
- **Increment [V]**: Spacing between points in the polarization curve;
- **Adaptive Increment**: Set to true if you would like to reduce the voltage increment adaptively if convergence could not be achieved with the larger value;
- **Min. Increment [V]**: If Adaptive Increment is set to true, this value controls the minimum change in cell voltage before the polarization fails to converge and the voltage is updated again. Note that this value has to be positive as a value of zero would lead to an infinite loop.

Subsection **Parametric Study** is used when a parametric study is to be performed for a different variable than cell voltage. In this case, the most important parameter to specify is **Parameter name** which identifies the parameter that is to be modified. Any parameter in the OpenFCST data file can be modified following the format specified below. The parameters and how to modified are explained below:

- **Parameter file output:** File where the parametric study results should be stored.
- **Parameter name:** Enter the name of the parameter you would like to study. Use one of the following formats:
 - For normal parameter: `Subsection_1>>Subsection_2>>Value;`
 - For boundary value or graded: `Subsection_1>>Subsection_2>>Material_id:Value,`

where `Subsection_1` and `Subsection_2` would be the sections where the parameter is found in the data file. For example, if we would like to change the temperature of the cell, we would write `Fuel cell data>>Operating conditions>>Temperature cell.`

- **Initial value:** Enter the value you would like to start the parametric study from.
- **Final value:** Enter the final value for the parametric study.
- **Increment:** Spacing between points in the parametric study.
- **Adaptive Increment:** Set to true if you would like to reduce the increment adaptively if convergence could not be achieved with the larger value
- **Min. Increment:** If Adaptive Increment is set to true, this value controls the minimum change in parameter before the polarization fails to converge and the voltage is updated. Note that this value has to be positive as a value of zero would lead to an infinite loop.
- **Parameter values:** If you would like to run the parametric study for only some points, then a list containing the discrete values of a parameter of study can be included here. If this value is empty, then the Initial and Final value entries are used, if this list is filled, then the list is used instead.

6.4.2 The Logfile section

The **Logfile** section is used to specify where the output from OpenFCST should be stored and how much output should be stored.

6.5 The OpenFCST data file

The OpenFCST data file will change depending on the parameters that are specified in the main file, but in general it contains the following sections:

- **Adaptive refinement:** This section is used to control adaptive refinement options. Only advanced users should modify this section.
- **Newton:** Section used to specify the parameters that control the Newton solver to solve the problem.
- **Grid generation:** Section used to specify the geometry of the domain. **This section is critical** as the ID specified in this section for `material_ID` and `boundary_ID` are used to impose the appropriate initial and boundary solution in section Equations.
- **Discretization:** This section is used to specify the type of finite element used to discretize the governing equations. The quadrature formula and degree are also set here.

- **System management:** This section is used to specify the equations that need to be solved. In most cases, this section is filled automatically by the application.
- **Equations:** **This section is critical.** In this section, the initial solution and boundary conditions for each equation need to be specified. The values are specified using the following format `material_ID:value` and `boundary_ID:value`. This section will be discussed below in more detail.
- **Reaction source terms:** This section is used to turn on/off source terms for the equations.
- **Initial Solution:** This section is used to control initialization and output of the initial solution. OpenFCST can generate a default initial solution or a previous solution can be used as an initial guess. This section allows users to create an initial solution for later use and to load previous solutions as an initial solution. Note that the initial and final solution have to be on the same mesh. OpenFCST also allows users to output the initial solution here.
- **Linear Solver:** This section is used to select the linear solver that users want to use. Several direct and iterative solvers are available. For non-linear problems, only direct solvers, e.g., `UMFPACK` and `MUMPS`, and iterative solver `ILU-GMRES` are satisfactory due to the nature of the Jacobian matrix to be inverted.
- **Fuel cell data:** This is the most important section of OpenFCST for a user as it encapsulates all the fuel cell information, e.g., operating conditions, type of kinetic model, catalyst layer model and GDL type.
- **Output:** This subsection is used to specify the output format for the mesh and the output solution data. In general, we recommend EPS and VTU formats for grid and data respectively. Only advanced users should modify this section.
- **Output Variables:** This section is used to request OpenFCST to calculate and output a variety of functionals, i.e., integral quantities, at postprocessing such as current density and water crossover.
- **Postprocessing:** This section is used in order to provide additional information to compute the functionals specified in **Output Variables**.

6.5.1 The Adaptive refinement section

This section is used in combination with the flags in **Grid generation** section to control refinement levels and output options for the mesh and solution.

This section has the following entries:

- **Number of Refinements:** This parameter is used to define the number of times the mesh will be refined. The minimum value is one, i.e., only the original mesh is solved. At each adaptive refinement level, either all the cells (global) or 30% of the cells with largest error (computed using an error estimator; adaptive) are split into four. The process is repeated at each refinement level. An example of mesh that is refined globally several times and an adaptively refined mesh is shown in Figure 6.6.
- **Refinement:** This flag specifies the type of refinement to be used if in **Adaptive refinement** section you decided to solve the problem in several meshes. Mainly two options are available:
 1. **global:** At each refinement level, each cell in the domain is divided into four new cells.
 2. **adaptive:** At each refinement level, a percentage, specified in **Refinement threshold**, of cells with the largest error are divided into four cells. Also, the percentage of cells in **Coarsening threshold** with the smallest error are re-merged into one cell if they had been previously refined.
- **Refinement threshold:** For adaptive refinement, the percentage of cells with largest error that should be refined.

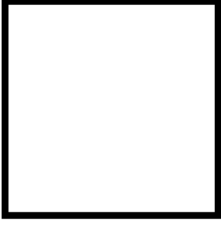


Figure 6.3: Initial Grid.

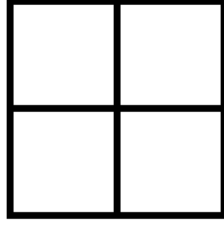


Figure 6.4: Global 1st Refinement.

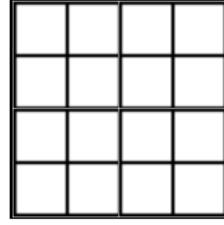


Figure 6.5: Global 2nd Refinement.

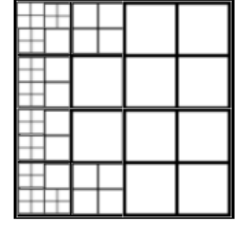


Figure 6.6: Adaptive Refinement.

- **Coarsening threshold:** For adaptive refinement, the percentage of cells with smallest error that should be coarsened.
- **Output initial mesh:** Set flag to true if you want to output an EPS figure of the initial mesh using the value in `Output initial mesh filename`.
- **Output initial mesh filename:** Filename of where the initial mesh will be output.
- **Output intermediate solutions:** Set flag to true if you would like the solution at each grid refinement to be output. Please note that outputting the solution is time consuming.
- **Output intermediate responses:** Compute the functionals in `Output variables` at each grid refinement. Use this option if you want to perform a grid refinement study. Please note however that computing the functionals is time consuming.
- **Output final solution:** Output the final solution to a file.
- **Compute errors and convergence rates:** Internal option for developers. Always set this value to false.
- **Use nonlinear solver for linear problem:** Internal option for developers. Always set this value to false.

In order to control the type of refinement, `Grid generation>>Refinement` is used. This value can be set to adaptive or global in order to specify the type of refinement. Furthermore, if adaptive is used, the percentage of cells that are refined and coarsened is given by `Grid generation>>Refinement threshold` and `Grid generation>>Coarsening threshold`.

6.5.2 The Newton section

This section specifies the parameters that are used to control the Newton iteration for the case of non-linear problems. There are many parameters most of which are self-explanatory. The most critical parameters are:

- **Max steps:** Used to limit the number of iterations carried out by the Newton solver.
- **Tolerance:** The value of the L^2 -norm of the residual. Ideally, this tolerance should be kept at $1.0e^{-9}$. However, in certain circumstances, convergence with that tolerance may not be possible or feasible given the computational time. In these scenarios it is possible to reduce the tolerance to $1.0e^{-4} - 1.0e^{-6}$ while still keeping reasonable accuracy.
- **Reduction:** Use if you want convergence to be accomplished after the initial residual (or whatever criterion was chosen by the solver class) is reduced by a given factor. This is useful in cases where you don't want to solve exactly, but rather want to reduce the residual by a small amount. We recommend setting this value always to $1.0e^{-20}$.

In order to control the solution output during the Newton iteration, the following four parameters can be used

- **Debug level:** Write debug output to the logfile. The higher the number, the more output. The range is between 0 and 3.
- **Debug residual:** Output the residual at every Newton iteration. This then can be used to locate errors/bugs in the code.
- **Debug solution:** Output the solution at every Newton iteration. This then can be used to locate errors/bugs in the code.
- **Debug update:** Output the solution update at every Newton iteration. This then can be used to locate errors/bugs in the code.

The parameters `special_block.i` are internal variables. They should not be used by the users.

6.5.3 The Grid generation section

The **Grid generation** section is one of the most important sections in OpenFCST, together with the **Fuel cell data** section. This section is used to define the geometry for the fuel cell. The fuel cell geometry is represented by the following data:

- Length of each fuel cell layer, channel, and land.
- A collection of **Material** IDs used to identify each layer in the domain to the layers in the **Fuel cell data** section.
- A collection of **Boundary** IDs used to identify the boundaries of the domain where boundary conditions are applied.

These sections are key and are used in the **Equations** section to specify initial solution and boundary conditions and in **Fuel cell data** to associate each layer in the geometrical domain with the corresponding fuel cell properties.

The **Grid generation** section contains many entries. The following entries are used to specify the type of mesh and if any refinement on the mesh should be performed prior to the simulation:

- **Type of mesh:** This entry specifies the type of mesh you would like to generate. You have two main options **ExternalMesh** will load an external mesh for your simulation. The other options use OpenFCST internal mesh generator to directly generate the geometry.
- **File name:** If type of mesh has been set to **ExternalMesh**, this section should contain the name of the mesh file you would like to load. The file should be in the same folder as main.
- **File type:** Specify the extension of the file.
- **Initial refinement:** Number of times we want to globally refine the original grid before starting to solve.

The following option is used to specify the numbering scheme for the degrees of freedom in the mesh. By default, degrees of freedom (DoFs) are sorted by component, but the following flag can be used to sort the DoFs using other schemes

- **Sort Cuthill-McKee:** Organize the degree of freedom numbering for the mesh using the Cuthill-McKee algorithm.

The subsection `Internal mesh generator parameters` is only needed if the OpenFCST internal mesh generator is used. If it is not used, the section can be almost entirely ignored since the `ExternalMesh` should already contain material and boundary IDs. Some post-processing routines however might use several entries such as `Cathode CL thickness [cm]` and `boundary IDs`, so it might be necessary to fill out these values even if using an `ExternalMesh` in some instances.

The Dimensions subsection in `Internal mesh generator parameters` is used to specify the dimensions of each parameter in the cell. It contains the following:

- `Cathode current collector width [cm]`: Thickness of the ribs of the bipolar plates (BPP) [cm].
- `Cathode channel width [cm]`: Thickness of the channels on the BPP [cm].
- `Cathode CL thickness [cm]`: Thickness of the cathode catalyst layer [cm].
- `Cathode GDL thickness [cm]`: Thickness of the cathode gas diffusion layer [cm].
- `Cathode MPL thickness [cm]`: Thickness of the cathode microporous layer [cm].
- For the remaining entries, please mouse over the entry in the GUI for meaning.

Subsection `Mesh refinement parameters`:

- `Initial vertical cell count`: Number of cells we want in the y-direction of the original grid before starting to solve
- `Horizontal division of cathode GDL`: Number of cells we want in x-direction in the cathode GDL layer
- `Horizontal division of cathode CL`: Number of cells we want horizontally in the cathode CL layer
- For the remaining entries, please hover the mouse over the entry in the GUI for its meaning.

Subsection `Material ID` is used to define the material ID for each component of the cell. The material ID is used in `Fuel cell data` to associate each one of the cells in the mesh with the desired fuel cell properties. The entries in this section look as follows:

- `Test`: Material ID for GridTest.
- `Cathode current collector`: Current collector material_id.
- `Cathode gas channel`: Cathode gas channel material_id.
- `Cathode GDL`: Cathode gas diffusion layer material_id.
- `Cathode MPL`: Cathode microporous layer material_id.
- For the remaining entries, please hover the mouse over the entry in the GUI for its meaning.

Subsection `Boundary ID` is used to define the boundary ID for each boundary in a fuel cell. These IDs are used in Equations section to specify Dirichlet and Neumann boundary conditions for each equation at each one of the defined boundaries. **The number 255 defines an interior boundary condition in deal.II. All internal boundaries MUST have a 255 boundary ID.** The entries in this section appear as follows:

- `c_Ch/GDL`: Cathode gas channel and gas diffusion layer boundary_id.
- `c_BPP/GDL`: Cathode bipolar plates and gas diffusion layer boundary_id.
- `c_GDL/CL`: Cathode gas diffusion layer and catalyst layer boundary_id. Since this boundary is an internal boundary in most cases, it must be set to 255.
- For the remaining entries, please mouse over the entry in the GUI for meaning.

6.5.4 The Discretization section

The **Discretization** section is used to select the finite element discretization and the quadrature formula used to evaluate the weak form integrals. The key parameters defined in the subsection are:

1. **Element**: Defines the finite element discretization for each equation. This parameter is discussed in detail below.
2. **Degree Mapping**: Defines the geometric mapping. In most cases a linear mapping is used, i.e. set the value to 1.
3. **Boundary fluxes**: Set to true if there are any either Neumann or Robin boundary conditions. If the parameter is set to false, then OpenFCST will skip looping over boundaries resulting in faster computational speeds.
4. **Interior fluxes**: Set to true if there are any flux jumps between elements. This will only occur if using a Discontinuous Galerkin (DG) formulation. So far we have not implemented any DG schemes in OpenFCST.
5. **Matrix**: Used to control the number of quadrature points required to evaluate the integrals on the left hand side of the local weak form defining the partial differential equation.
6. **Residual**: Used to control the number of quadrature points required to evaluate the integrals on the right hand side of the local weak form defining the partial differential equation.

Of the above parameters, **Element** is of critical importance as it specifies the type of finite element used for the spatial discretization. If only one equation is used, the element is specified as:

```
1 set Element = FE_Q(2)
```

where $FE_Q(2)$ refers to the type of element, i.e. Lagrange element (FE_Q), and the number in parenthesis, i.e. 2, is the order of the element, in this case quadratic.

For system of equations, the finite element discretization for each equation needs to be specified. For example, for a system of five equations we would write

```
1 set Element = FESystem[FE_Q(2)^5]
```

where $FE_Q(2)$ refers to the type of element and the number five in $FE_Q(2)^5$ refers to the number of variables included in the quadratic category. If we want to use different elements for different variables we would specify it by separating the elements with a dash. For example, if we wanted the first two elements solved with a cubic Lagrange element and the last three solved with a linear Lagrange element approximation, we would insert the following line. $FESystem[FE_Q(3)^2-FE_Q(1)^3]$

The final two sections in **Discretization**, are

1. **Matrix**;
2. **Residual**.

Matrix and Residual control the number of quadrature points required to evaluate the integrals in the local weak form of our partial differential equation. The default value of -1 will set the number of quadrature points to the order of the finite element used plus one in each direction, e.g., for second order elements, number of quadrature points in each direction is $2 + 1 = 3$ (in 2D, using quadratic elements, the number of quadrature points would be 9). Assigning a default value of -1, for most cases, should be sufficient to achieve an exact solution of the integrals.

6.5.5 The System management section

The **System management** subsection is responsible for defining the **Solution variables & Equations** being used. This section is populated by the application directly and should not be modified by the users.

6.5.6 The Equations section

This section is responsible for specifying the initial solution and boundary conditions for the application at hand. The section is subdivided in one subsection per equation that needs to be solved. Inside each subsection, the main section needs to be specified:

- **Initial data:** It is used to specify a piece-wise initial solution for the simulation. It contains two main entries:
 - **Variable initial data:** If set to true and the application has implemented a variable initial guess, the variable initial guess is used.
 - **variable_name:** The name of this section corresponds to the variable we are trying to initialize. In this section, for each material ID a value needs to be given in order to setup an initial solution. The initial solution might be overwritten using the section **Fuel cell data>>Operating conditions**, however the map of material ID must be included here. If you have a mesh with two material IDs, e.g. CL is 5 and GDL is 8, and you would like to setup the initial solution for your variable to 0.2 and 0.3 in CL and GDL respectively, then the entry will be: **5:0.2, 8:0.3**. For each solution variable we have a comma-separated list of material ID, colon, value.
- **Boundary conditions:** Provides the ID for the type of boundary condition that you would like to have.
- **Boundary data:** Provides the value for Dirichlet, Neumann and Robin boundary conditions. As for the case of **Initial data**, the same format is used. Again this section is mandatory and it is the user's responsibility to create the appropriate map.

6.5.7 The Reaction source terms section

This section is used to turn on/off source terms for the equations.

6.5.8 The Initial Solution section

This section is used to control the initial guess that the user would like to use. As specified in **Equations**, a piece-wise approximation can be used as an initial guess. Another possibility is to use a previous solution as an initial guess. In this case, simulation is run first with the '**Output solution for transfer**' option set to true. This will produce a hidden file containing the solution. This solution can then be read in if '**Read in initial solution from file**' is set to true and the new boundary conditions are applied. Reading in an old solution might be beneficial when convergence becomes an issue if the initial solution is similar to the new solution you are trying to obtain, i.e. all parameters are the same but one, e.g. when running a polarization curve.

Two more parameters appear in this section:

- **Output initial solution:** This option is usually used for debugging purposes for developers in order to make sure the initial solution is specified correctly.
- **Initial solution output file:** Specifies the name of the output file where the initial solution will be stored if the flag above is set to true.
- **Use pre-defined initial solution:** Some applications, like MEA, have a pre-defined initial solution. If set to true, this pre-defined solution is used.

6.5.9 The Linear Solver section

In this section, the linear solver to be used for solving the problem, either the full problem or the linearized equations in the case of a non-linear system, is specified. The most important parameter to select here is the `Type of linear solver` parameter. The parameter `Assemble numerically` is set to true if you would like to evaluate the Jacobian for the Newton loop numerically. This is extremely time consuming and therefore should only be used if an analytical Jacobian is not developed. All applications in OpenFCST use an analytical Jacobian. The other parameters are used to control the convergence of the program similar to the Newton section.

6.5.10 The Fuel cell data section

`Fuel cell data` subsection is the most important section in OpenFCST project as it specifies all relevant properties pertaining to operating conditions and each respective layer in a fuel cell. The section is divided into the following main subsections:

1. **Operating conditions:** Specify operating conditions for the fuel cell. It contains the following entries:
 - **Adjust initial solution and boundary conditions:** Use the parameters in Operating conditions to create an initial solution and overwrite the boundary conditions for the problem specified in `Equations>>Initial Data` using the parameters in this section. This is the recommended option as it will directly calculate the appropriate relative humidity for your cell.
 - **Temperature cell:** Fuel cell temperature in Kelvin.
 - **Cathode pressure:** Cathode pressure in Pascals.
 - **Cathode initial oxygen mole fraction (prior to humidification):** Oxygen molar fraction prior to humidification. For example, 0.21 for air.
 - **Cathode relative humidity:** Relative humidity as a fraction, i.e., between 0 and 1.
 - All other parameters are entered using the same units. Their meaning is self-explanatory from the GUI.
2. **Materials:** This section includes the properties of gases and other materials that are used in multiple layers.
3. Subsections defining every layer needed for the given application.

For each layer in a fuel cell, a subsection is defined here. All layers have the following common entries:

- **Material id:** This integer number should be set to the material ID in the computational mesh that corresponds to the layer you would like to use the properties in this section for. If this section is a catalyst layer, then the material ID number corresponds to the number used in `Grid generation>>Internal mesh generator parameters>>Material ID>>Cathode CL`. **This entry is extremely important.**
- **PSD parameters:** This subsection is used to specify a pore-sized distribution for the layer. This section is not used in release 0.2.
- **Generic data:** This subsection is used to specify porosity, permeability and other properties that relate to a porous layer.
- **Layer type:** This drop down menu is very important as it specifies the layers that are currently available in the OpenFCST library. The value used here corresponds to a sub-section below if any parameters are needed from file. Only the properties in that sub-section (if defined) are needed to specify your layer. Currently only a few layer types are available: a dummy layer where all parameters can be specified, a design layer where parameters are obtained using effective medium theory as in reference [3], and a limited number of commercial layers. **Input from users is needed to improve this database.**

Other entries are specific to each layer. They are self-explanatory by mousing over the parameters in the GUI. Here we will discuss the parameters in subsection **Cathode catalyst layer** in detail as it is the most complex entry. In this subsection we have the following additional entries

- **Catalyst type**: Drop-down menu in order to select the appropriate catalyst from the OpenFCST database.
- **Catalyst support type**: Drop-down menu in order to select the appropriate catalyst support from the OpenFCST database.
- **Electrolyte type**: Drop-down menu in order to select the appropriate electrolyte from the OpenFCST database.
- **Kinetics type**: Drop-down menu in order to select the appropriate kinetics from the OpenFCST database.

For each one of the types in the drop-down menu either all properties are specified in OpenFCST, or several parameters are required from the user. In the latter case, a sub-folder is available to modify the parameters. For catalyst, catalyst support, and electrolyte, the properties of the materials are in sub-folder **Materials**. For the case of the kinetics, the folder **Kinetics** contains the sub-folders for the different options. As in the case of the layer, only the relevant sub-section with the name of the type selected is applicable and can be modified.

The OpenFCST team has implemented several multi-step kinetic models discussed in references [4], [8] and [7]. In particular, the **DualPathKinetics** model is used for the hydrogen oxidation reaction and the **DoubleTrapKinetic** model is used for the oxygen reduction reaction. These models can be used with any of the applications here.

There are a large number of parameters in each subsection. Each parameter name is either self-explanatory or an explanation is shown by hovering the mouse over the parameter. If further information is needed, the users can go to the class documentation where additional information regarding each parameter is available.

6.5.11 The Output section

This subsection is used to specify the output format for the mesh and the output solution data. In general, we recommend EPS and VTU formats for grid and data respectively. Only advanced users should modify this section.

6.5.12 The Output Variables section

In this section, it is possible to specify integral equations that you would like to evaluate after the solution has been obtained such as current density, water crossover, and others.

6.5.13 The Postprocessing section

This section is used to input information to some **Output Variables** that might require it.

Chapter 7

Post-processor

OpenFCST can output results in many different formats using the deal.II output parser. OpenFCST developers however output the solution in .vtu format and use the open-source post-processing software [Paraview](#) to analyze their results. ParaView is an open-source data analysis and visualization program. It can run on multiple operating systems. ParaView users can quickly analyze their data visually using qualitative and quantitative methods already implemented in the software.

A ParaView tutorial can be found at the following [site](#). Wolfgang Bangerth and Timo Heister recently published a very good lecture on how to use Paraview at the following [site](#).

Part II

Developer's Reference Guide (Under development)

Chapter 8

Setting up the development environment for OpenFCST

In the following sections we describe how to create an OpenFCST branch where you can develop your own code and commit the changes to the OpenFCST team. Also, we show how to setup KDevelop, the program we recommend for compiling and modifying code in OpenFCST.

8.1 Getting the development version of OpenFCST

The development version of OpenFCST is hosted on a private repository on BitBucket. In order to access the `development` branch of OpenFCST, please contact the developers. You can also develop code from the GitHub version of the code and then submit your changes to the OpenFCST team. We will then merge the changes into the code for the next release.

If you want to modify OpenFCST, you will need to first clone the development version of OpenFCST, and then create your own branch of the code. Afterwards you can then modify, test, and commit this branch. Once your branch has been tested and validated, you can issue a Pull Request. Then, several senior OpenFCST developers will look at the code, suggest changes, and finally merge the code into the stable development version of OpenFCST. The stable development version of OpenFCST is then used to create new releases.

Creating a new branch

Committing changes to the development branch is not allowed. You will need to create a pull request of your branch. Every user can create their own branch of OpenFCST. The recommended convention for branch usage is that each user creates their own branch for each issue in OpenFCST they would like to address. The naming convention is: `username/issue_name`. For example, if Secanell wants to create an issue to fix a bug on postprocessing, the branch would be named `secanell/postprocessing`.

To create a branch, users can either create it on their own machine and then push it to BitBucket or create the branch directly on BitBucket. If the branch is created on BitBucket, then, in order to checkout the branch to the appropriate machine, the user needs to issue the following command:

```
1 git branch branch_name origin/branch_name
2 git checkout branch_name
```

Both steps can be performed simultaneously with

```
1 git checkout -b username/issue\_name origin/username/issue\_name}
```

If the branch is created on the local repository first using `git checkout -b branch_name`, then you can commit it to BitBucket, i.e. remote server, using `git push -u origin branch_name`. The `-u` flag means that from now on your branch `branch_name` will track the branch on BitBucket.

Adding, changing, staging, committing and pushing

Once the branch is created, users can work on that branch for as long as needed. Users can make changes and commit the changes to their local repository using:

```
1 git add file_name
2 git commit -m "message about commit"
```

Please DO NOT use `git add *` or `git add -u` as you then have little control over what you are staging to be committed. Using `git status` you can see which files have changed so that you can add them as appropriate.

To commit to BitBucket, you can use:

```
1 git push origin branch_name
```

Request for branch to be merged into development

Once you have finished fixing the issue you created the branch for, you need to follow these three steps:

1. Update your origin information using: `git remote update` (this will update all your local information regarding the branches on BitBucket).
2. Merge your branch with the latest version of development using: `git merge origin/development`. This is VERY important. The administration will not accept any pull requests that have not been fast-forwarded to the `origin/development` branch.
3. Issue a pull request in BitBucket

There are three main branches

- Master branch: Stable version of OpenFCST (no pull requests will be accepted to this branch).
- Development branch: The most up-to-date version of OpenFCST, personal branches should be started from this branch and all pull requests should be submitted to this branch.
- Release branch: Branch containing the latest release of OpenFCST.

Workflow for new development

If you want to develop new code, please follow these steps:

- Clone the repository using:
`git clone https://your_username@bitbucket.org/ESDLab/openfcst.git`
- Create a new branch related to the new component/issue you would like to work on using: `git checkout -b name_branch`. Note: The command above will create a branch named `name_branch` and will checkout that branch so you are ready to work.
- Once you are done with the development, ask for a pull request to merge your branch to the development branch.

Note: Merges to Master will be rejected without review.

A reminder: when developing code, please work on Debug mode (the current version gives an error once the program finishes in debug mode, please ignore for now as we will be working on fixing this) and test on Debug and Release mode before issuing a pull request. We are aware that running tests in Debug is more time consuming, but the issues that we have in debug mode have occurred precisely because we did not test on that mode.

8.2 Setting up OpenFCST under KDevelop

If you are going to be developing new routines for OpenFCST, we recommend that you use either KDevelop or Eclipse to modify, compile, and debug new code. In order to setup a KDevelop project with OpenFCST, follow the steps below:

- Compile OpenFCST using the provided script, i.e. `openFCST_install`. This will configure all the folders you will be using during configuration of KDevelop. This step is not required, however, it is recommended and the steps below assume OpenFCST has been installed and that the Build and Install directories already exist on your computer.
- Go to **Project > Open/Import Project...** Go to the OpenFCST folder, enter the `src` folder and then, select the `CMakeLists.txt` file in `openfcst/src` (see Figure 8.1). In the next window, enter the name of the project, e.g. OpenFCST, and select **CMake Project Manager**. At this point, the project should either appear or prompt another window asking for the Build folder. If the latter is the case, point KDevelop to the Build folder in the main folder of OpenFCST. Then, the project import is complete and the project menu will appear on the left hand side.
- If you want to modify the compilation parameters in the project, you can do that by right clicking on the project name and selecting **Open Configuration....** The menu in Figure 8.2 would appear. You should not need to modify many parameters, however several parameters are handy. The variable `OPEN_FCST_BUILD_TYPE` allows you to compile the code in Debug or Release mode. The former is used during code development as it provides a lot more information about errors, the latter is best for simulations as the code can be several times faster. Another useful parameter is `OPEN_FCST_DIMENSIONS`. If set to 2, it will only compile a 2D version of OpenFCST. If you compile with 3, it will compile a 3D version. If you compile with 1, it will compile both 2D and 3D versions. Finally, on the left menu if you click on Make, the parameter `Number of simultaneous jobs` sets up the compilation for using as many threads as specified.

Next, we will setup the environment to run and debug OpenFCST within KDevelop by following the steps below:

- Go to **Run > Configure Launches...** . The window in Figure 8.3 will open.
- Select either Global or your project option (we recommend your project).
- Press the '+' button on the top of the window. Once you press this button, a new option will open under either Global or OpenFCST.
- Select **New Native Application Configuration**, then on the right of the window, under Executables, enter the OpenFCST binary file, i.e. `OpenFCST_directory/Install/bin/fuel_cell-2d.bin`. Under Behaviour, in Working Directory enter the data folder from which you would like to run the code. In Arguments, enter the main parameter file, see Figure 8.3.
- Your code is set! Click OK on the window. Now, you can run the code with the **Execute** and **Debug...** buttons on the menu. If you have more than one **New Native Application Configuration** configured, rename them by clicking on the same. Then, you can switch between application configurations using **Run > Current Launch Configuration**

8.2.1 Formatting OpenFCST files

All files should start with the following information:

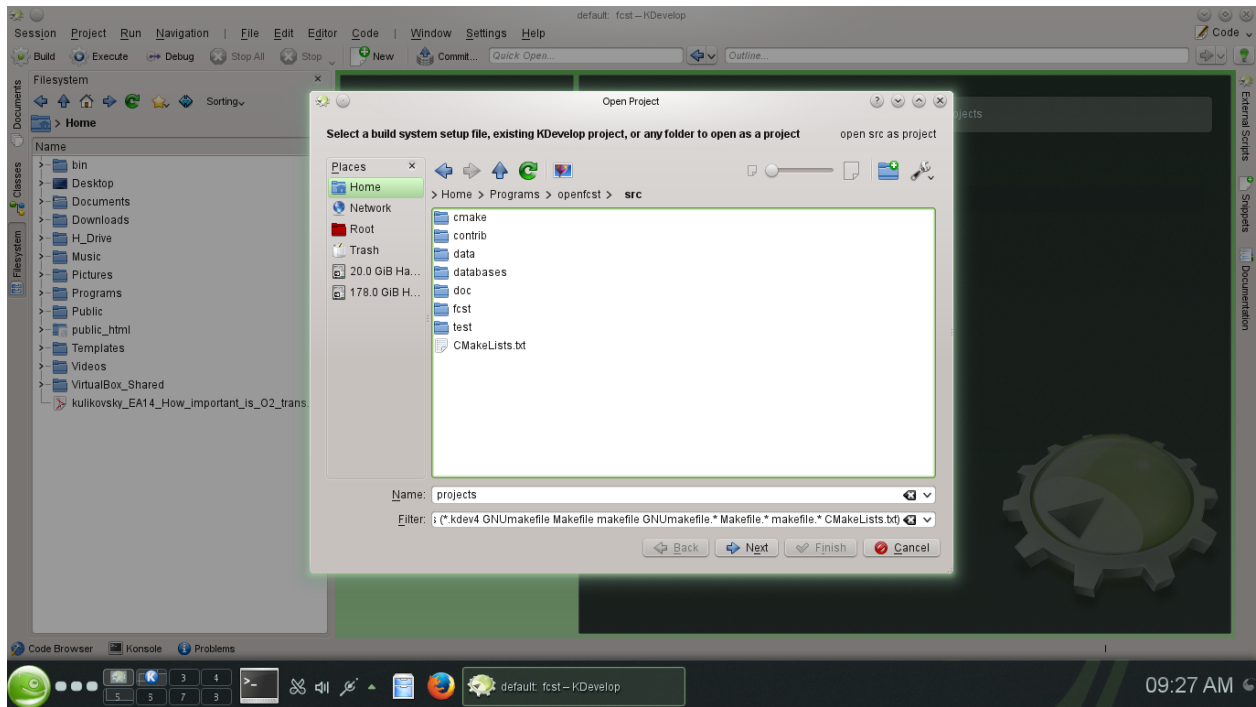


Figure 8.1: Initial window in KDevelop to import a CMake project.

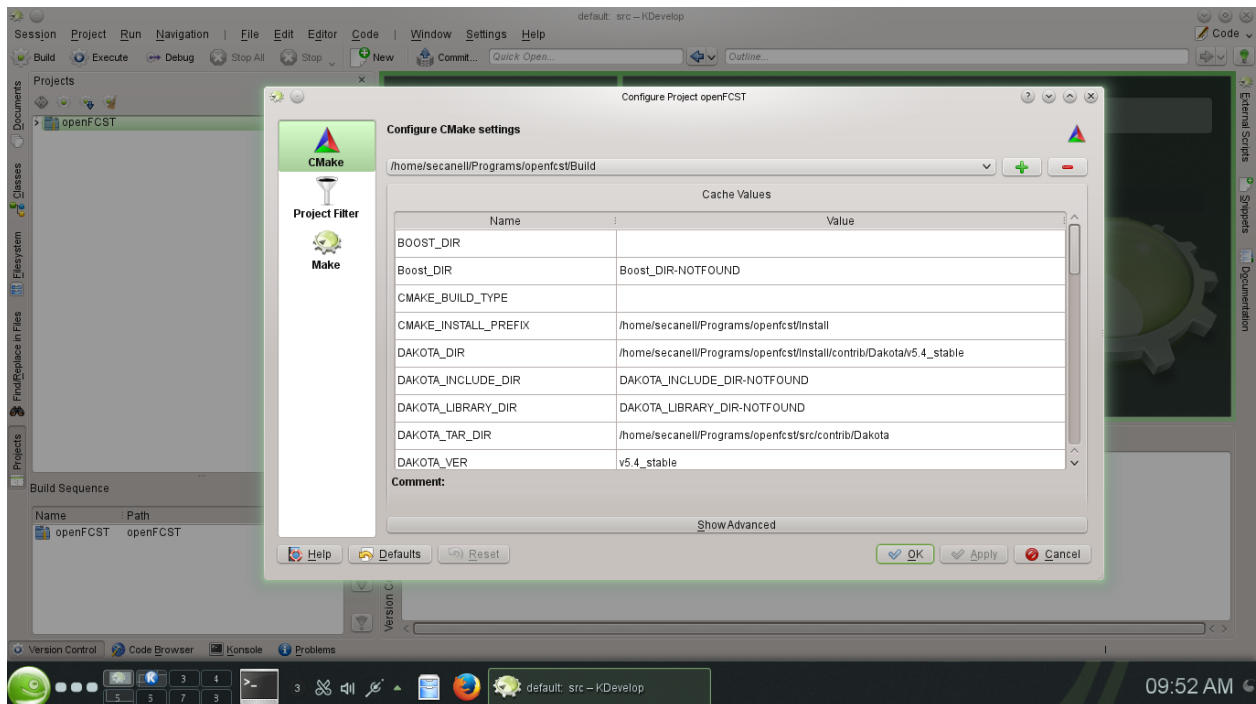


Figure 8.2: Configuring your CMake project in KDevelop.

Chapter 9

Coding Guidelines

The purpose of this chapter is to specify coding guidelines for developers of OpenFCST in order to improve code understanding, reliability and readability.

It is intended that this document will collaboratively cover topics of naming, syntax, documentation, and development.

9.1 Class and Member Naming Conventions

Naming conventions are defined in this section. Consistent naming is important as it improves code understanding and readability. Distinct naming styles help us understand whether a name pertains to a type, function, or variable. It is important that all names communicate without ambiguity of the meaning and/or purpose of the object they represent. The following convention is used in OpenFCST.

Class naming: Class names and Types should be written in camel-case with their first letter capitalized. Class names should consist of un-abbreviated nouns. For example:

```
1 class ClassName; //Good
2
3 class my_class; //Not good
```

Function naming definition: Function names should be written with words separated by an underscore. Function names should contain verbs that describe their actions without ambiguity. If a class contains two functions with similar names but different purposes then at least one of the functions should be renamed. Example:

```
1 compute_I(double a); // Good
2
3 generateInverse(double numToInvert): //Not Good
```

Variable naming definition: Use of simple variable names like `i` or `count` should be avoided for all cases except for loop counters. The variable name should reflect the content stored in the variable. Variable names should follow either camel-case style with the first letter being lower case or with words separated by an underscore, e.g.,

```
1 anodeKinetics //Good
2 int num //Not Good
```

Constant naming definition: Constants should be written as capital letters and the name should reflect the meaning of the constant. Also, avoid using a single letter, e.g. write `GAS_CONSTANT` instead of `R`.

```
1 SPEED_OF_LIGHT //Good
2 c //Not Good
```

OpenFCST contains a file with many constants already available named `fcst_constants.h`. If you need additional constants, please define them there so that we can all use them.

A Word on Commenting: Comments can be useful tips that will help us to understand code, but should not be used primarily to help us understand complicated code. Well written code with correct object and function naming should be self explanatory without the need for excess comments.

9.2 File headers

Each file in OpenFCST should start with the following header:

```
1 // -----
2 //
3 // FCST: Fuel Cell Simulation Toolbox
4 //
5 // Copyright (C) insert_date by author_name
6 //
7 // This software is distributed under the MIT License
8 // For more information, see the README file in /doc/LICENSE
9 //
10 // - Class: insert_class_name
11 // - Description: insert_one_sentence_description
12 // - Developers: insert_author_name
13 //
14 // -----
```

9.3 Developing documentation using Doxygen

OpenFCST uses Doxygen to automatically generate the documentation for namespaces, classes, and data members. Doxygen uses comments which accompany class, function and variable definitions in the header file to produce the class documentation for OpenFCST. Doxygen allows us to develop styled, easily readable documentation with minimal developer effort. The following are doc string templates that should be implemented by OpenFCST developers when creating new classes, functions, variables, and namespaces.

9.3.1 Documenting classes

The structure for the documentation for each class in a `.h` file is found below. A template file is located in `src/fcst/include/utils/documentation.template`. Documentation for a class should contain the following main sections:

- @brief:
- Introduction parameter
- Theory
- Input parameter
- Usage
- References

The documentation is placed prior to the class declaration, i.e., prior to `class TemplateClass` in the example above. The documentation must be placed in a section between a symbol `/**` and a symbol `*/` following the Doxygen input syntax, i.e.,

```
1 /**
2  *
3  */
```


For more information on Doxygen formatting tips visit the [Doxygen site](#).

An example template class documentation is shown below:

```
1 namespace FuelCell
2 {
3     /**
4     *
5     * @ brief SHORT DESCRIPTION OF THE CLASS
6     *
7     * MORE DETAILED DESCRIPTION OF THE CLASS
8     *
9     * Explain here the purpose of the class and its main use.
10    *
11    * If the class is a child of another base class, explain
12    * which member functions are redeclared
13    * and the extensions to the parent class
14    *
15    * <h3> Theory </h3>
16    * DETAILED EXPLANATION FOR THE THEORY BEHIND THE CLASS. FOR EQUATIONS DESCRIBE
17    * HERE THE PDE THAT YOU ARE SOLVING.
18    *
19    * <h3> Input parameters </h3>
20    * LIST OF INPUT PARAMETERS FOR THE CLASS.
21    * @code
22    * subsection FuelCell
23    *     subsection EXAMPLE
24    *         set PARAM1 = DEFAULT VALUE # EXPLANATION
25    *         set PARAM2 = DEFAULT VALUE # EXPLANATION (IF SEVERAL OPTIONS, ADD HERE)
26    *     end
27    * end
28    * @endcode
29    *
30    * <h3> Usage details</h3>
31    * Here please enter the usage details on how the class
32    * should be used. Including the following
33    * - Does it need to read data from file?
34    * - Are there any member functions that are required to initialize the class? In which order should
35    * they be called
36    * - Include a piece of code showing how the class would be used
37    * (see example below from FuelCellShop::Material::IdealGas
38    *
39    * @code
40    * //Create an object of TemplateClass
41    * FuelCellShop::TemplateClass example;
42    * // Set necessary variables
43    * marc = 358;
44    * example.set_variable(marc);
45    * // You can now request info from your class.
46    * double marc = example.get_variable();
47    * //Print to screen all properties
48    * example.print_data();
49    * @endcode
50    *
51    * <h3> References </h3>
52    *
53    * [1] articles
54    *
55    * @author YOUR NAME HERE
56    *
57    * @date 2013
58    *
59    */
60 //Name class as per coding conventions
61 class TemplateClass
62 {
63 public:
64     /** Constructor */
65     TemplateClass()
66     {}
67
68     /** Destructor */
69     ~TemplateClass()
70     {}
71
72     /** Explanation of what the function does. Use get_***()
73     * functions whenever you want to
74     * retrieve information from the class instead of accessing the data directly */
```

```

75     double get_variable()
76     {variable};
77
78     /** Develop a routine that prints the data stored in the class out. This class
79      * is extremely useful for debugging.
80      * Make sure you output to the variable deallog otherwise your output will not
81      * be stored in the .log file.
82      */
83     void print_data()
84     {
85         deallog<<"Output data: "<<variable<<std::endl;
86     }
87
88     private:
89     /** Explanation of what the variable means. Units of the variable if it is a physical quantity
90      * For example:
91      * This variable stores the inlet temperature of the gas. Units are in Kelvin.
92      */
93     double variable;
94 }; //class
95 } //namespace
96
97 #endif

```

9.3.2 Documenting member functions

Before each member function definition in every class, the following doc string should be implemented:

```

1 /**
2  *Description : A brief description of the purpose
3  *
4  *Use cases   : A list of intended uses
5  *
6  *Access rules: Public/Private/Protect
7  *
8  *Inputs      : Variable descriptions and Types
9  *
10 *Outputs     : Description of output
11 *
12 *Notes       : Other important information
13 *
14 */

```

9.3.3 Documenting variables

Before each data member definition, the following doc string should be implemented:

```

1 /**
2  *Description : A brief description of the purpose, units (if applicable)
3  *
4  *Use cases   : A list of intended uses
5  *
6  *Access rules: Public/Private/Protect
7  *
8  *Notes       : Other important information
9  *
10 */

```

9.3.4 Documenting namespaces

All namespace information is found on the file `namespaces.h` which is used only for documentation. In general, we have two namespaces:

- FuelCell: This namespace is used for Applications and supporting routines;
- FuelCellShop: This namespace is used for Equations, Layers, and Materials.

9.3.5 TODO list in HTML documentation

If you would like to include new tasks to the TODO list, you can include them in the *.h file where the task needs to be done. Doxygen will move all TODO tasks to a page in the HTML documentation. The Doxygen documentation has been setup to contain three TODO subcategories in order of priority. To include a TODO task, go to the *.h file and type the following:

```
1 \todo1 Task to do -- Top priority
2 \todo2 Task to do -- Medium priority
3 \todo3 Task to do -- Low priority
```

9.3.6 Linking to other functions

While referencing to a particular method used while explaining a function, it can be linked to the application by using # before the method name. If the method belongs to the same class, then this would suffice. Else, we can use the full namespace definition of the function in the documentation. Doxygen will automatically link the function to its documentation. Same thing can be done for the data members.

For example:

```
1 /** This structure has two constructors. Default constructor doesn't set any value. It also sets the
2 *   boolean member #initialized to \p \b false. This can be checked by using #is_initialized member
3 *   function and(...)
4 */
```

9.4 Assertions and exception handling

OpenFCST includes many assertions in order to check if member function are receiving the expected data. Please make sure that all your member functions check that the data you are expecting is received by the class. OpenFCST uses two types of assertions:

- Assert: Checks that the desired information is provided. This assertion will only work in debug mode. This means that when running in optimized mode this check will not take place. However, this also means that the code performance will not be impacted once you run in optimized mode, i.e. the default compilation method. If you are coding, always work on debug mode. If you are developing routines, always work on optimized mode.
- AssertThrow: Some assertions check that the parameters in the input file are correct. Such assertions should be active in either debug or optimized mode. For such cases use AssertThrow.

An example of an Assert call is as follows:

```
1 Assert( solution_vector.size() == residual_vector.size(),
2         ExcMessage("Solution and residual vectors are not the same size in Class XX, Function YY") );
```

In this case, if solution and residual are the same size, the code will continue without any problems. If solution and residual are of different size, i.e. if the assertion is FALSE, then it will output the ExcMessage.

Chapter 10

Development Process

This chapter outlines a development method known as Test Driven Development (TDD). TDD insures thorough testing of code throughout its development and implementation life cycle, resulting in improved reliability. Coupled with the process of Refactoring, TDD produces robust code that is easily read and understood. Concepts of TDD and Refactoring shall be briefly explained in the following sections. It is recommended that OpenFCST developers use this approach when developing new classes.

10.1 Test Driven Development

Test Driven Development (TDD) is a software development methodology which is rather different compared to the typical development process generally acquired when learning programming. Imagine a programmer is given a problem for which they must provide a software solution. Instead of diving in “head first” and writing code to provide the solution, a TDD programmer first writes a number of Unit tests. Unit testing is a method by which individual units of source code are tested to determine if they are fit for use. In object oriented programming units are individual member functions. The Unit tests define acceptable behavior of the code that the programmer intends to create. Once the Unit tests have been created, the programmer may then write the actual code that will provide their programming solution. Whilst writing this code the programmer uses their Unit tests to ensure the written code behaves correctly, i.e. passes the test.

A more detailed description of the TDD methodology as seen in Figure 10.1.

The main steps are as follows:

1. Creation of a set of Unit tests that define the correct behavior of production code. Note: We must ensure that these tests initially fail.
2. Creation of production code and subsequent checks to see if it passes unit tests. Work on production code continues until all tests are passed.
3. Code is cleaned. Refactoring to increase readability and understanding of code.
4. More test cases may be added in order to ensure sufficient testing. The number of unit tests required for satisfactory testing is subject to the programmers judgement.

Advantages of TDD are as follows:

1. Increased reliability of code.
2. Programmers who write more tests are more productive.
3. Not just a validation of correctness: TDD also drives development by forcing the programmer to think strongly about how their code will be used. This leads to smaller, more focused classes, and cleaner interfaces.

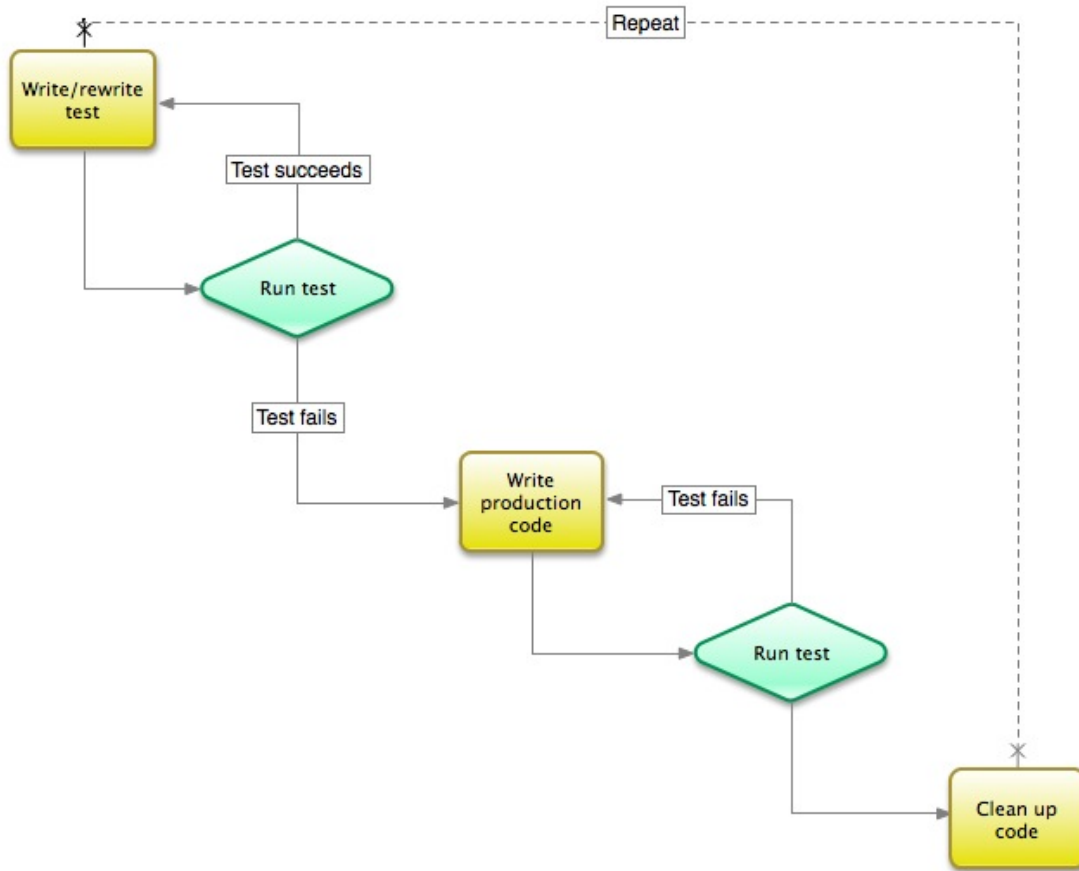


Figure 10.1: TDD Cycle.

4. Unit tests act as documentation: Testing functions are understandable examples of how the production code should be used.
5. TDD ensures consistent testing off all resources throughout the development of a piece of software.

10.1.1 Unit Tests

Unit tests, as already mentioned, are tests that determine if individual units of source code are fit for use. It is important that unit tests are written very simply in order to ensure correctness (since there is no tests to ensure that the unit tests are correct). The following is a simple example of a test function and the corresponding production code it is intended to test.

Unit Test:

```

1 void testAdd(){
2
3     int expectedAnswer = 5;
4     int answer = add(3 ,2);
5
6     TEST_ASSERT(expectedAnswer == answer); //Check to see if output is as expected, and make record if it
7       is not.
8 }
  
```

Production code (under test):

```

1 int add(int a, int b){
2   return a*b; //Obviously this will cause the test to fail
3 }

```

Obviously, the above test will fail because the function `add()` has been implemented incorrectly. Using a Unit testing library such as CppTest we will receive the following output:

```

1 FailTestSuite: 0/0, 0% correct in 0.000002 seconds
2   Test:      testAdd
3   Suite:    ExampleSuite
4   File:     mytest.cpp
5   Line:     9
6   Message:  "expectedAnswer == answer"

```

10.1.2 TDD Implementation in the OpenFCST

The unit testing structure that is implemented in OpenFCST is built using a library called CppTest. CppTest is a portable, powerful, and simple unit testing framework for handling automated tests in C++. The focus lies on usability and extendability.

Several output formats, including simple text output, compiler-like output, and HTML can be produced. The tests suit is launched from the system builder class's `run_tests()` function, see Figure 10.2. Firstly, unit tests are run (which will test individual components of various classes), then system level tests.

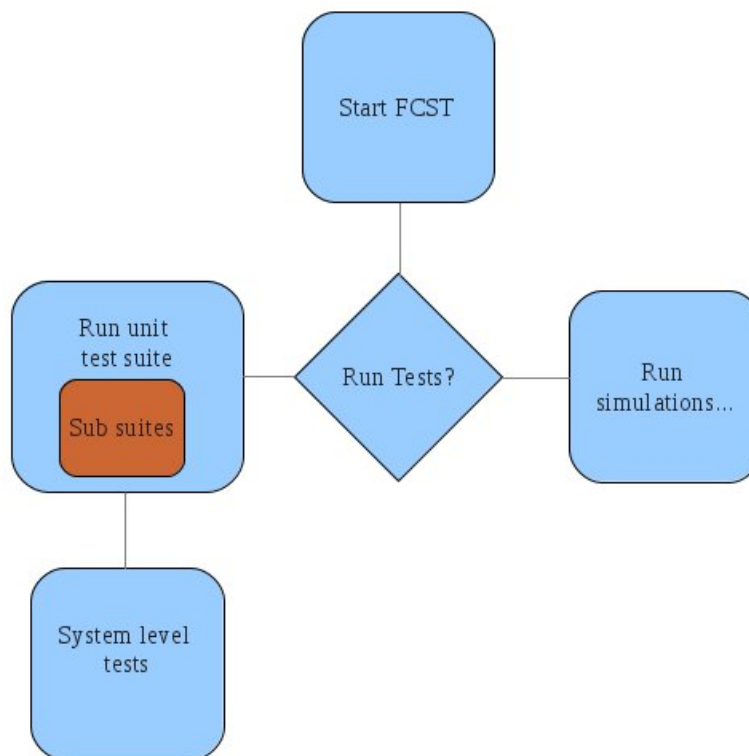


Figure 10.2: TDD illustration scheme.

The operation is as follows.

The “Run tests” parameter is set in the main parameter file.

```
1 set Run tests = true
```

The run test function in SimulatorBuilder is called, which in turn calls the OpenFCST testing suite.

```
1 void SimulatorBuilder<dim>::run_test()
2 {
3 deallog << "=====Running Unit Tests===== " << std::endl;
4
5 FcstTestSuite::run_tests();
6
7 deallog << "=====System Tests Complete===== " << std::endl;
```

The OpenFCST test suite runs all of the unit testing suites that it is composed of (currently only the OpenFCST units testing suite)

```
1 bool FcstTestSuite::run_tests()
2 {
3     Test::Suite ts;
4
5     //add sub tests suites
6     ts.add(std::auto_ptr<Test::Suite>(new FcstsUnitsTestSuite));
7     ts.add(std::auto_ptr<Test::Suite>(new IonomerAgglomerate3Test));
8
9     Test::TextOutput output(Test::TextOutput::Verbose);
10    return ts.run(output);
11
12 }
```

Below is an example of an individual "sub-testing" suite. Each individual unit test is added to the test suite in the constructor and will be called individually when the .run() function is called.

```
1 #ifndef _FCST_UNITS_TESTSUITE
2 #define _FCST_UNITS_TESTSUITE
3
4 #include <cpptest.h>
5 #include "fcst_units.h"
6
7
8
9 class FcstsUnitsTestSuite: public Test::Suite
10 {
11 public:
12     FcstsUnitsTestSuite()
13     {
14         //Add a number of tests that will be called during Test::Suite.run()
15         //Generic cases
16         TEST_ADD(FcstsUnitsTestSuite::perBigToSmallTest);
17         TEST_ADD(FcstsUnitsTestSuite::bigToSmallTest);
18         TEST_ADD(FcstsUnitsTestSuite::perSmallToBig);
19         TEST_ADD(FcstsUnitsTestSuite::smallToBig);
20
21         //specific Cases
22         TEST_ADD(FcstsUnitsTestSuite::btuToKwh);
23         TEST_ADD(FcstsUnitsTestSuite::kwhToBtu);
24     }
25 protected:
26     virtual void setup() {} // setup resources... called before Test::Suite.run()
27     virtual void tear_down() {} // remove resources...called after Test::Suite.run()
28
29 private:
30     //Generic cases
31     void perBigToSmallTest();
32     void bigToSmallTest();
33     void perSmallToBig();
34     void smallToBig();
35
36     //Specific cases
37     void btuToKwh();
```



```

38     void kwhToBtu();
39 };
40
41 #endif

```

Below is an example of an individual unit test taken from the “FcstsUnitsTestSuite” test suite. It checks that the function `convert` correctly converts units of BTU to units of KJ.

```

1 void FcstsUnitsTestSuite::btuToKwh()
2 {
3     TEST_ASSERT(Units::convert(1,Units::BTU_to_KJ) == 1.054);
4
5 }

```

10.1.3 Implementing a new test suite

If you would like to add a unit test suite for a class that you are creating, follow these steps:

1. Create the `class_test.h` and `class_test.cc` files in the `unit_test` folders (use the existing `.h` and `.cc` files as templates).
2. Add an include statement to your new `class_test.h` file in `FCST_TEST_SUITE.h` (under code comment “List of sub suites”).
3. In `FCST_TEST_SUITE.cc`, add your new test class in the `run_tests()` function.

If you would like your test to be able to see private variables inside the class that it is testing, you must add it as a friend to that class:

1. Go to the header of the class you are testing (`class.h`).
2. At the top of the file (outside namespace scope), make a reference to your test class (e.g. “class `nameOfTestClass`”).
3. In the class’s declaration, write the friend statement above the public section (e.g. “friend class `::nameOfTestClass`”).

10.1.4 Refactoring

Refactoring is a technique for restructuring existing code to improve it’s readability and user understanding, without changing the behaviour of the code in any way. When refactoring code, a programmer looks for “Bad Programming Smells” and uses various methods to remove them. Code smells are not bugs, but weakness in code design that makes code difficult to understand and can lead to bugs being introduced into the code.

Some examples of bad programming smells:

1. Duplicated code: identical or very similar code exists in more than one location.
2. Long method: a method, function, or procedure that has grown too large or complicated.
3. Inappropriate intimacy: a class that has dependencies on implementation details of another class.
4. Too many parameters: a long list of parameters in a procedure or function make readability and code quality worse.
5. Complex conditionals.
6. Temporary variables and fields.

7. Use of primitives rather than objects.
8. Classes and functions with multiple responsibilities.

The following is an example of code that exhibits bad smells (see if you can spot them):

```
1 void sendMessage(Message dataToSend, string phoneNumber, string networkOperator){
2
3     string areaCode = "213";
4
5     if (networkOperator == "Rogers"){
6         string _phoneNumber = areaCode + "4" + phoneNumber;
7         MessageBuffer b;
8
9         for(int i=0; i < dataToSend.length(), i++)
10             b.pack(dataToSend[i]);
11
12         send(b, _phoneNumber)
13     }
14     else if(networkOperator == "Telus"){
15
16         string _phoneNumber = areaCode + "9" + phoneNumber;
17         MessageBuffer b;
18
19         for(int i=0; i < dataToSend.length(), i++)
20             b.pack(dataToSend[i]);
21
22         send(b, _phoneNumber)
23     }
24 }
25
26 }
```

Bad smells include local data (the area codes should not be stored locally but should be the responsibility of another class such as PhoneBook), and duplicate code inside either if statement. The following code represents the above code refactored. The refactoring patterns extract method has been used to replace the code from within the for loop, the pattern extract data has been used to remove the local variables areaCode as well as the if statement comparisons. The result is code that is shorter, more easily understood, and more easily reused.

```
1 void sendMessage(Message dataToSend, string phoneNumber, string networkOperator){
2
3     string fullPhoneNumber = PhoneBook::getAreaCode() + PhoneBook::getOperatorCode(networkOperator) +
4         phoneNumber;
5     MessageBuffer buffer = package(dataToSend);
6     send(buffer, _phoneNumber)
7 }
8
9 MessageBuffer package(Message dataToSend){
10     MessageBuffer buffer
11
12     for(int i=0; i < dataToSend.length(), i++)
13         buffer.pack(dataToSend[i]);
14
15     return buffer;
16 }
```

Also note the change of variable names. The new names do a better job at describing their purpose.

10.1.5 Unit Standards

OpenFCST uses centimetres, grams and seconds (CGS) units for most of its variables.

Bibliography

- [1] M. Secanell, V. Zingan, M. Bhaiya, P. Wardlaw, M. Moore, K. Domican and P. Dobson, Fuel Cell Simulation Toolbox, User's Guide, 2015. URL: <http://www.openfcst.org>.
- [2] M. Secanell, Computational Modeling and Optimization of Proton Exchange Membrane Fuel Cells, Ph.D. thesis, University of Victoria, November 2007.
- [3] M. Secanell et al., Multi-variable optimization of PEMFC cathodes using an agglomerate model, *Electrochimical Acta*, 52(7):2668-2682, 2007.
- [4] M. Secanell, R. Songprakorp, A. Suleman, N. Djilali. Multi-objective optimization of a polymer electrolyte fuel cell membrane electrode assembly. *Energy and Environmental Science*. 1:378-388, 2008.
- [5] Dobson P., Lei C., Navessin T., Secanell M., Characterization of the PEM fuel cell catalyst layer microstructure by nonlinear least-squares parameter estimation, *Journal of the Electrochemical Society*, 159:B514-B523, 2012.
- [6] Setting the \$Id\$ Tag in Subversion. <http://www.startupcto.com/server-tech/subversion/setting-the-id-tag>. Accessed on April 6, 2015.
- [7] M. Secanell, A. Putz, P. Wardlaw, V. Zingan, M. Bhaiya, M. Moore, J. Zhou, C. Balen and K. Domican, OpenFCST: An Open-Source Mathematical Modelling Software for Polymer Electrolyte Fuel Cells, *ECS Transactions*, 64(3):655-680.
- [8] M. Moore, A. Putz and M. Secanell, Investigation of the ORR Using the Double-Trap Intrinsic Kinetic Model, *Journal of the Electrochemical Society* 160(6): F670-F681, 2013.
- [9] M. Sabharwal, L.M. Pant, A. Putz, D. Susac, J. Jankovic, M. Secanell, Analysis of Catalyst Layer Microstructures: From Imaging to Performance, Fuel Cells, 2016.