DRAFT: Open-source Fuel Cell Simulation Toolbox (OpenFCST)

-

User and Developer's Reference Guide

M. Secanell, A. Putz, V. Zingan, M. Bhaiya, M. Moore, P. Dobson, P. Wardlaw, K. Domican
Energy Systems Design Laboratory, University of Alberta, Canada

Created on: June 11, 2012
Last updated: September 1, 2013

Contents

Chapter 1

Introduction

1.1 Overview of the program

The Fuel Cell Simulation Toolbox (FCST) is an open-source mathematical modelling package for polymer electrolyte fuel cells. FCST builds on top of the open-source finite element libraries deal.II, therefore many of its requirements in terms of operating systems and such are the same as for deal.II. FCST is distributed under the MIT License. FCST 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 kinetics mathematical models. 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 FCST have been validated with respect to experimental data in the literature [5] as well as numerical results from another model implemented in a commercial package. A thorough description of the model and the validation is presented in [2].

FCST 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 developer this code. The goal of FCST 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.

FCST 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. FCST contains a build-in mesh generator as well as it can import quadrilateral meshes generated with the open-source pre-processor Salome Salome if your problem requires to simulate more complex geometries. The physics and material database in FCST allow you to setup the governing equations for the most important physical processes that take place in a fuel cell. FCST already implements the weak form for many governing equations that are finally solved using the finite element open-source libraries deal.II. In order to analyze your results, FCST can output your results to .vtk files that can easily be read with the open-source post-processor Paraview. FCST is not only an analysis tool. FCST also is also integrated with the design and optimization package Dakota. Therefore, it can be used for design and optimization as well as parameter estimation, e.g. reference [2, 3, 4, 5].

FCST is still under development. If you like the library and would like it to continue to be developed please help the developers in the following ways:

- If you are an industrial researcher that is considering using FCST for research and development in the company, please consider contacting 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 FCST libraries in your publications. Please cite any relevant publication by the FCST developers as

well as the current reference [1].

- 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 the class to the developers so that it can be integrated with the newest version of FCST.
- If you are an industrial researcher that is considering using FCST for research and development in the company, please consider hiring the graduate students that develop FCST, i.e. the graduate students from the Energy Systems Design Laboratory at the University of Alberta.

FCST is still under development. Currently, the developers are working on:

- Improving the code readability New classes are being develop to try to make the code more easy 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
- Developing a multi-scale framework for electrode analysis
- Developing a non-isothermal membrane electrode model

1.2 About FCST

FCST 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 originating the idea of a toolbox that could be used to create modules or applications for fuel cell analysis. From 2006 to 2008, FCST development continued with the implementation of a complete membrane electrode assembly model; however, with M. Secanell as a unique developed 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 FCST 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. The current core of developers re-developed the majority of the classes in order to increase the modularity, usability and reliability of the code. Currently, FCST is used by 6-8 researchers at two different laboratories, it is tested nightly for errors and it contains a bug tracking site to report any issues with its performance.

The current group of FCST developers is formed by:

- M. Secanell, Associate Professor, Energy Systems Design Laboratory, University of Alberta, Canada

  Responsible for overall framework (base class concepts), optimization interface, electronic, protonic, membrane water transport, Fick's gas transport and kinetics

- A. Putz, Senior Research Scientist, Automotive Fuel Cell Cooperation Corp.

  Responsible for plug-points and AFCC contributions

- V. Zingan, Post-doctoral Fellow, Energy Systems Design Laboratory, University of Alberta, Canada

  Responsible for overall framework (base class concepts), Navier-Stokes, Darcy and multi-component fluid flow physical models and applications

- M. Moore, M.Sc. graduate from the Energy Systems Design Laboratory, University of Alberta, Canada

  Responsible for fcst_install script, double-trap kinetics model for ORR reaction and multi-scale framework (1D agglomerate models)

- M. Bhaiya, M.Sc. Student, Energy Systems Design Laboratory, University of Alberta, Canada

  Responsible for overall framework (base class concepts) and thermal physical models and applications

- P. Wardlaw, M.Sc. Student, Energy Systems Design Laboratory, University of Alberta, Canada

  Responsible for fcst_install script and multi-scale framework (1D agglomerate models)

- K. Domican, M.Sc. Student, Energy Systems Design Laboratory, University of Alberta, Canada

  Responsible for optimization interface and documentation

Other scientists that have also contributed substantial portions of code to FCST are:

- G. Kanschat, Universitt Heidelberg

  Developer of AppFrame (now part of deal.ii as MeshWorker

- P. Dobson, M.Sc. graduate from the Energy Systems Design Laboratory

  Developed parts of overall framework (base class concepts), optimization interface and multi-scale framework (1D agglomerate models)

- Ali Malekpourkoupaei, former M.Sc. graduate student at the Energy Systems Design Laboratory

  Developed classes PureGas and classes to compute binary diffusivity (together with M. Secanell)

1.3 License

1.4 Release changes

- May 18, 2010: Added variable scaling to method section for OPT+ and NL2SOL.In order to do so I modified *optimization_blockmatrix_application.cc* > *declare_parameters* and *dakota_application.cc* and .h to include a new bool variable and to read it.

- May 19, 2010:

  - The reference exchange current density is now read in uA/cm2 in order to help optimization/least squares since Dakota only allows scaling up to 1e-3
  - Changes to volume fraction and *catalyst_layer*
  - Is *epsilon_agg* modified inside the electrochemical equations?

- July 10, 2010:I moved "Optimization options section" to dakotadirectinterface (where it belongs). Now, if you want to run a single run, you have to comment all the information about optimization

Part I

User's Guide

Chapter 2

Installation

2.1 Downloading FCST

2.1.1 Users

OpenFCST can be found at the OpenFCST website. In the download section you will be able to find a .tar file with the latest release. Download the .tar file and download it at an appropriate location.

2.1.2 Developers

The latest version of OpenFCST is hosted in a repository at the University of Alberta. To download the latest version of FCST type the following on your terminal (you will require a username and password to download the code):

```
1 $svn checkout http://129.128.14.197/esdl/fcst/
```

If you do not have a username and password contact M. Secanell (secanell@ualberta.ca).

If you already have already checkout a version of the code, you can update your local copy by typing the following on your terminal

```
1 $svn update
```

2.2 Installing OpenFCST

2.2.1 System requirements

FCST is developed on Linux and compiled using the GCC compiler. The configure script that FCST uses relies heavily on the configure script from are the same as those used by deal.II. As deal.II, even though the FCST developers work on a Linux operating system, the FCST should not be platform specific and we strive to keep the source code C++ Standard compliant.

FCST developers perform nightly compilation tests on the following OS

- OpenSUSE 12.X
- Fedora 18

These are the operating systems that the FCST developers recommend.

If you would like to try to run the code under a Windows environment, you can try using either the Cywin or MinGW environments.

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

- GNU make, version 3.78 or later (or any other generator supported by CMake)
- GCC
- BLAS and LAPACK libraries
- MPICH compiler
- For generating the documentation: DOxygen

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

- 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.

2.2.2 Installation steps

Fuel Cell Simulation Toolbox is a fuel cell simulation package developed using several OpenSource libraries such as the deal.II libraries, the deal.II Application Framework and Shop, DAKOTA and COLDAE. In order to run without any difficulties, Fuel Cell needs to compile and link to all these applications which are provided with the code under the folder /contrib (Please note that each package is distributed under a different license).

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

```
1 $./fcst_install --without-Dakota --with-MPI --cores=4
```

Since Dakota is not included in the openFCST release, we have selected to compile FCST without without Dakota. We use the default MPI, so we specify the MPI flag without a path. Finally, we select to compile on four cores to speed up the compilation process.

The install script assumes the default path for the MPICH compiler and that all the libraries are in `./contrib`. If you already have a version of deal.II and you would like to use that version, use the flags –with-deal. For more information on the script options type

```
1 $./fcst_install --help
```

Chapter 3

Pre-processor

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

- Use the classes under `FuelCell::Geometry` namespace
- Read in a mesh generated using an open source mesh generator such as Salome

3.1 `FuelCellShop::Geometry` Namespace

Namespace `FuelCellShop::Geometry` contains classes to generate a cathode and anode fuel cell electrode, an spherical agglomerate; and, a membrane electrode assembly with five or seven layers (i.e., with and without micro porous 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 varialbes
7 grid.initialize(param);
8 //Generate the mesh and store it in the dealii::Triangularization variable tr
9 grid.generate_grid(*this->tr);
```

3.2 Developing a mesh in Salome

SALOME is an open-source software that provides a generic platform for pre-Processing. SALOME is a cross-platform solution that is distributed as open-source software under the terms of the GNU LGPL license. You might download both the source code and executable files from the Salome site.

3.2.1 Tutorial

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

The object we would like to mesh is represented by a two dimensional H-shaped domain as shown on Figure 3.1.

deal.II only works with meshes composed of either quadrilaterals in 2D or hexahedral in 3D. The current version of Salome is only able to produce these type of meshes with geometries that have outer boundary composed exactly of 4 pieces in 2D, e.g. see Figure 3.2, and 6 pieces in 3D, e.g see Figure 3.3. In order to

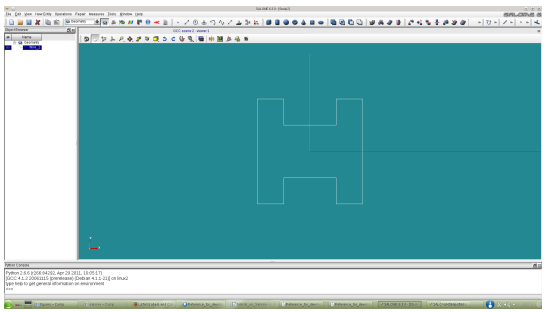

Figure 3.1: H-shaped domain.

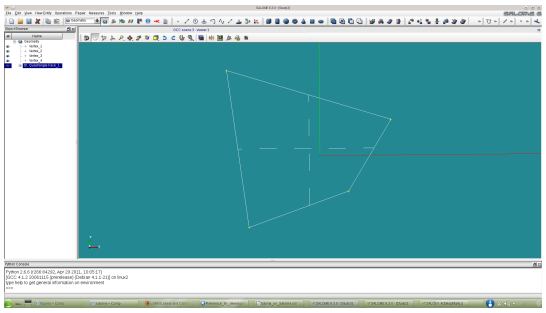

Figure 3.2: Linear quadrilateral.

increase the quadrilateral and hexahedral properties of Salome however, a commercial package is available called Hexotic distributed by Distine (For more information, please visit the following site)

The two dimensional H-shaped domain shown on Figure 3.1 has 12 pieces of the outer boundary and hence can not be meshed in Salome directly by means of quadrilaterals. If Hexotic is not available, then we can mesh the domain by splitting it into 3 parts such that each of these parts would have 4 fragments of the outer boundary. Then we mesh each of these parts and combine them into the H-shaped domain.

Let us do that step by step:

- Run Salome, **File** → **New**, and click on the **Geometry** button on the upper toolbox. We are now in the Geometry module of Salome, and the first thing we need to do is to define 12 main 2D points of the object: 1(-1, -1), 2(-0.5, -1), 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,

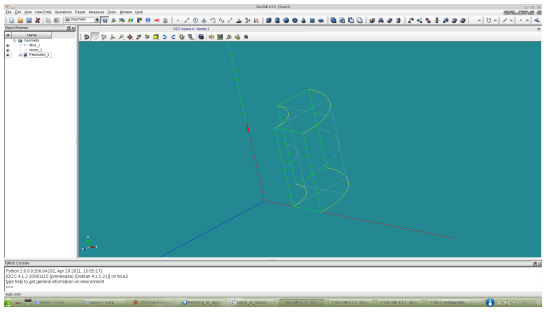

Figure 3.3: Quarter of cylindrical shell.

0.5), 9(0.5, -1), 10(1, -1), 11(1, 1), 12(0.5, 1). To create any of these points, we go to **New Entity** → **Basic** → **Point**, specify the respective fields **X:**, **Y:**, and **Z:**, and push **Apply and Close** button. Note that instead of **New Entity** → **Basic** → **Point** we can simpler choose **Create a point** on the upper toolbox.

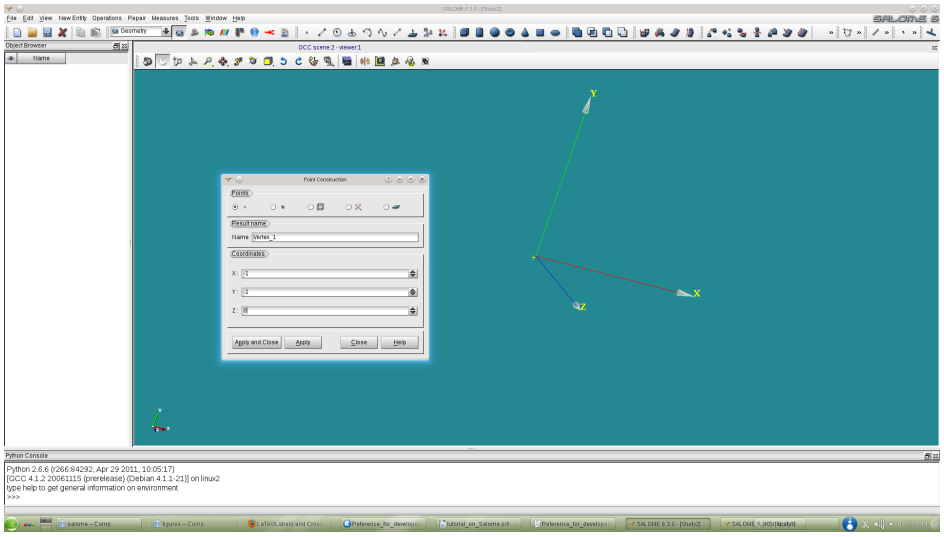

Figure 3.4: How to create a point.

After initializing all the points declared above, we have the following picture:

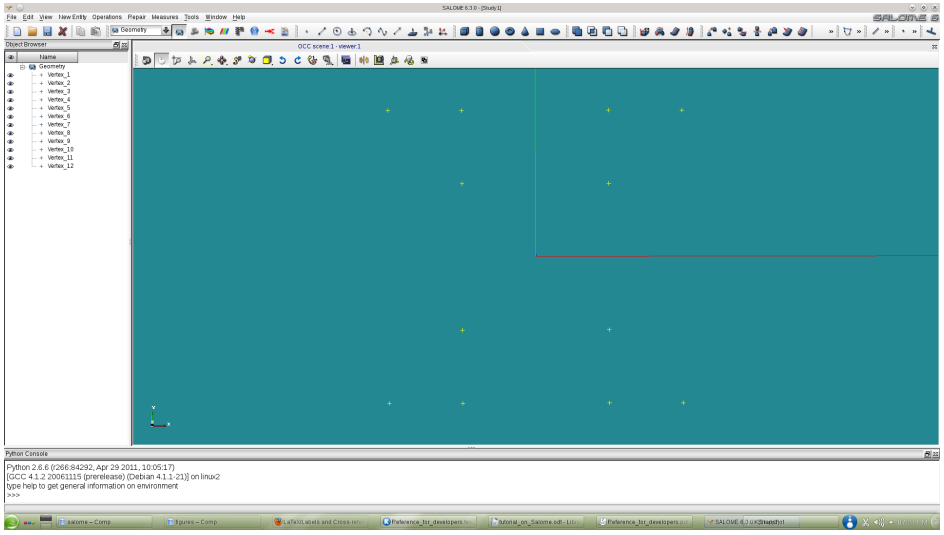

Figure 3.5: All points.

- At the next step we create 3 quadrangle faces. Each of these faces consists of 4 point: 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 push **Apply and Close** button:

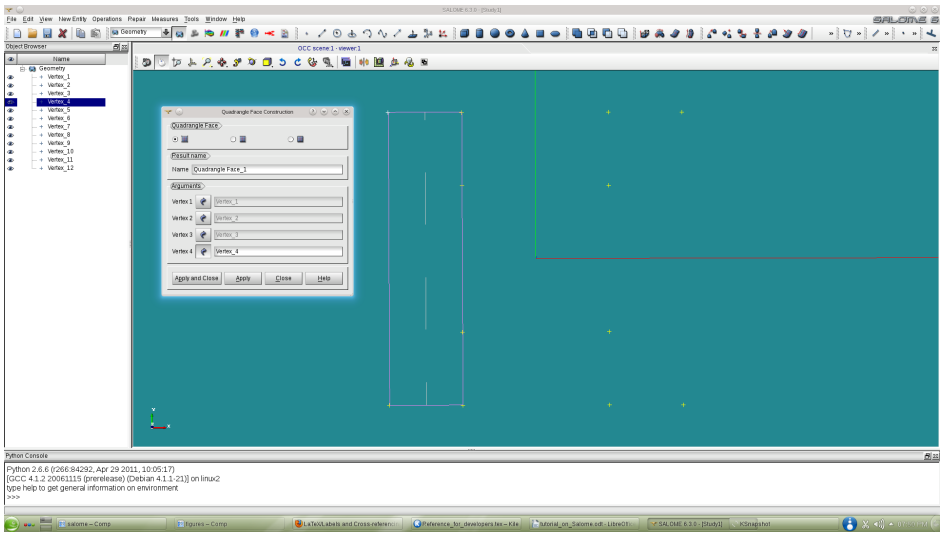

Figure 3.6: How to create a quadrangle face.

It is what we have when all the quadrangle faces have been created:

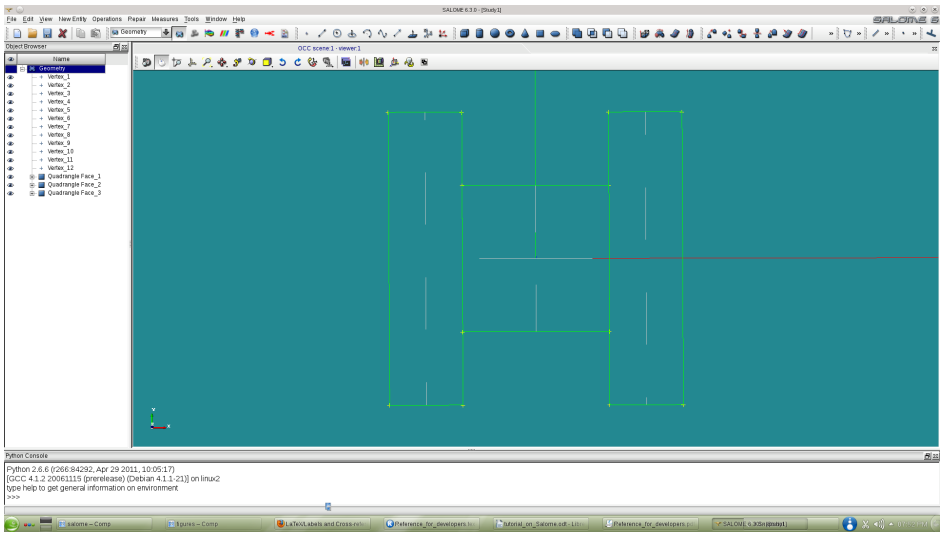

Figure 3.7: All quadrangle faces.

- At this point we have done with the Geometry module of Salome, and switch our attention to the Mesh module just by clicking 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 that is done, we choose **Quadrangle (Mapping)** from drop-down menu of the **Algorithm** field:

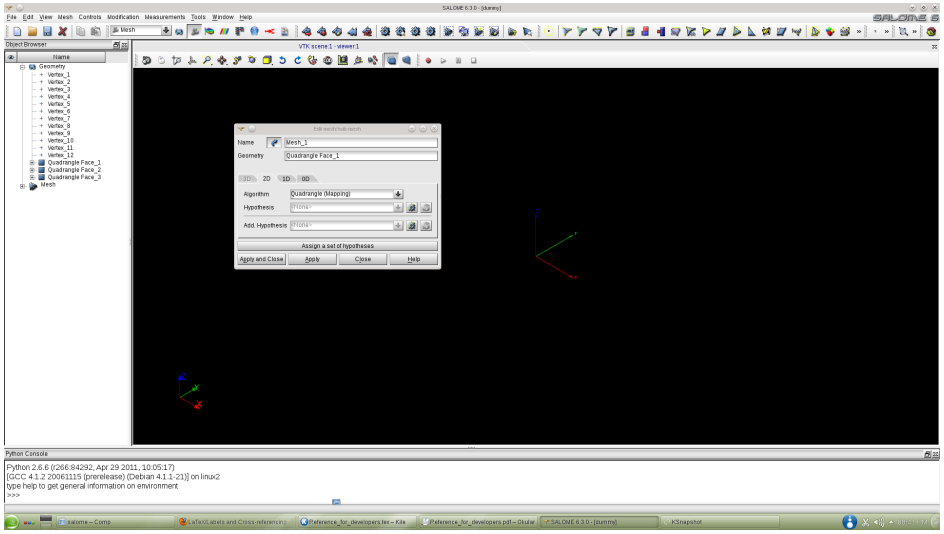

Figure 3.8

After that we click **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 called **Local Length** and set up the parameters of this hypothesis as follows:

After clicking **OK** the name of **Hypothesis** field should change to **Local Length_1**. Then **Apply and Close**, mouse right click on the **Mesh_1** and **Compute**:

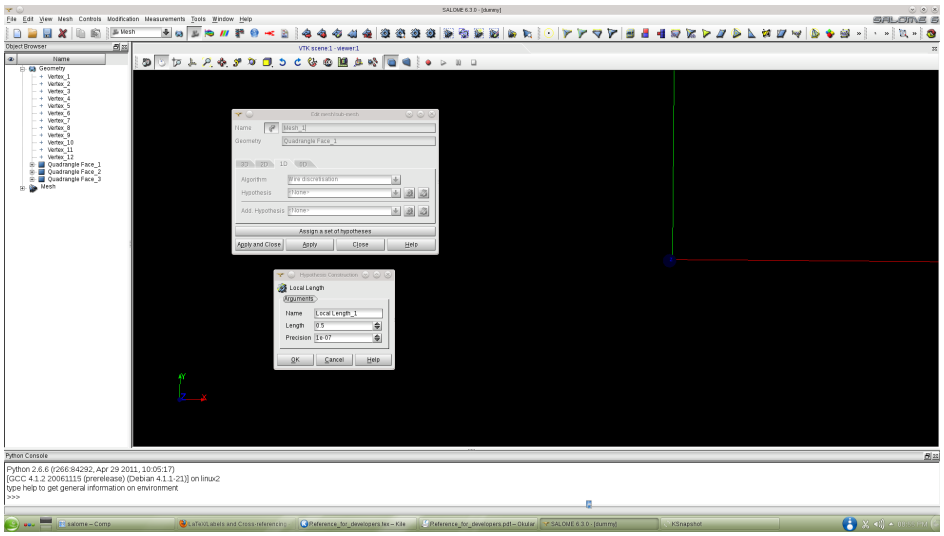

Figure 3.9

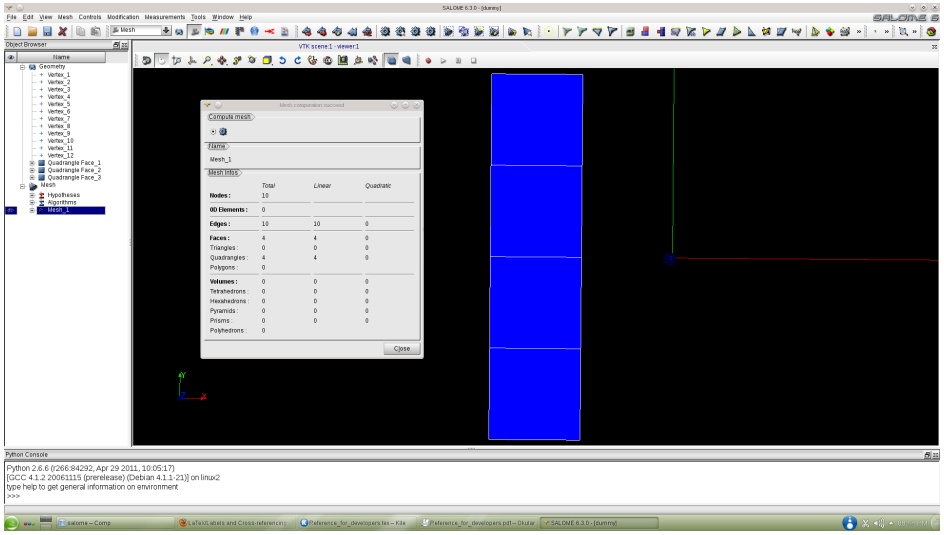

Figure 3.10

After applying this strategy to all the quadrangle faces we have something like this:

- Let us now assign meterial ids to all the cells we have created. For instance, all the cells of Mesh_1 have material id = 1, those belong to Mesh_2 - material id = 2, and the cells from Mesh_3 are supposed to have material id = 3. For instance, mouse right click on the **Mesh_1**, then choose **Create Group**. On this dialog box we set the following parameters:

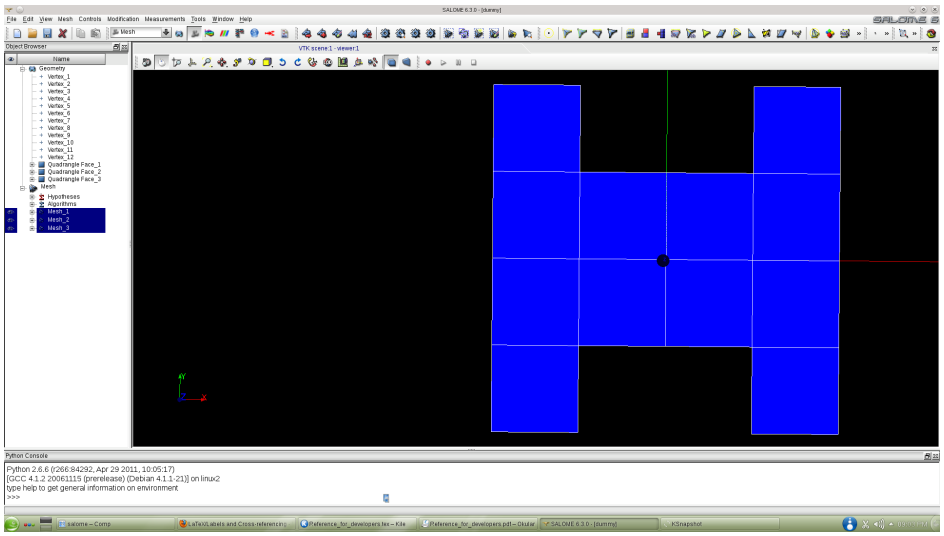

Figure 3.11

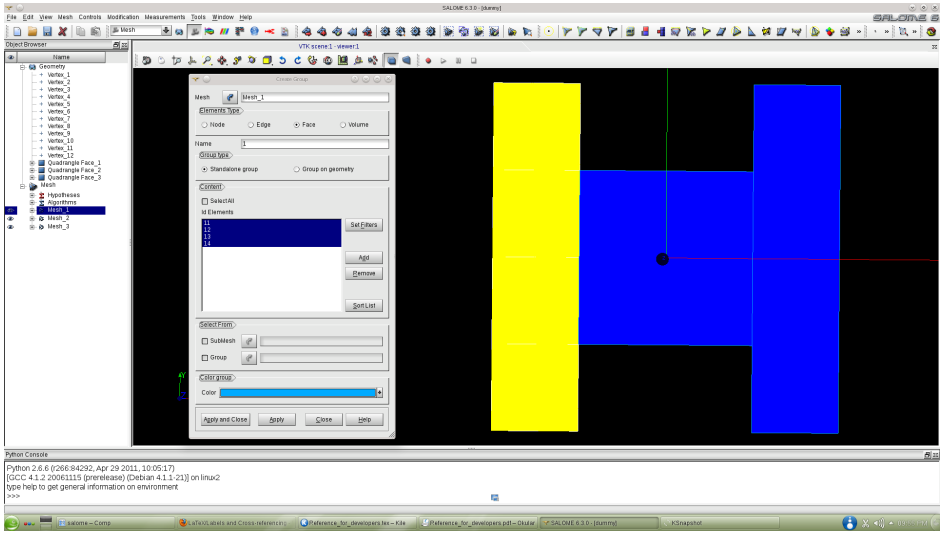

Figure 3.12

Hint: To add the cells to the **Id Elements** field, push and hold the Shift button on your keyboard and choose the cells by clicking the mouse left button. After all the desired cells have been highlighted, click **add** in the Create Group dialog window.

- Absolutely the same technique is used when we define the boundary ids. The only thing we should remember that the internal edges of the future compound mesh are not allowed to have the boundary ids. This restriction comes from deal.II limitations. As an example see the picture bellow:

- Finally, we create a compound mesh by simply merging all previously created meshes: **Mesh** → **Build Compound**:

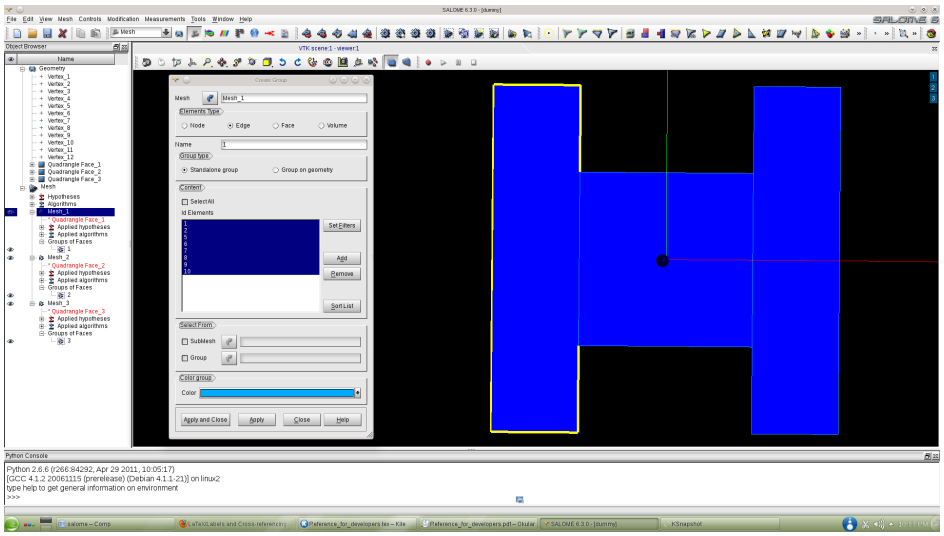

Figure 3.13

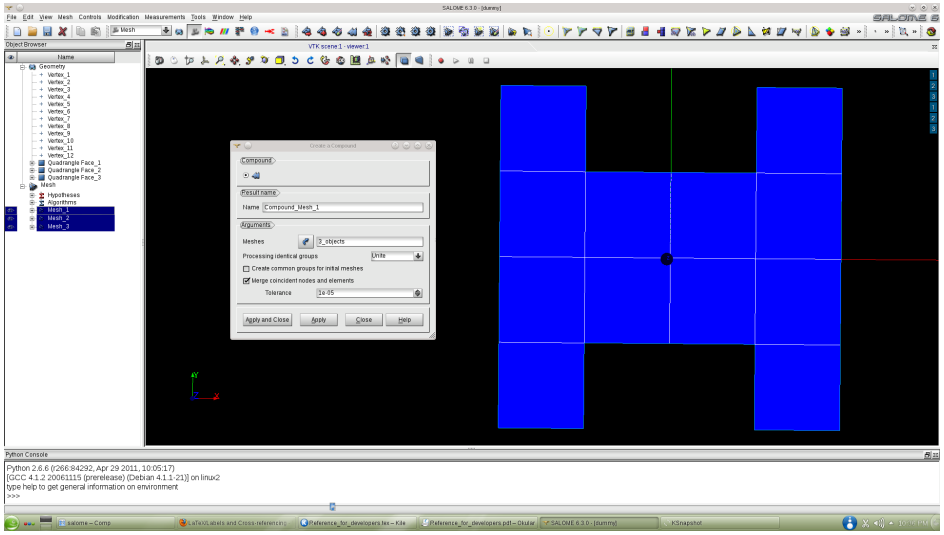

Figure 3.14

It is what we eventually have:

If you closely look at the picture above, you will see that some of the internal edges of the compound mesh still have the numbering. However, it is also strictly prohibited by the deal.II architecture !

- Therefore, the last step before exporting to the UNV format and deal.II library, is to manually remove all these internal numbers: **Modification** → **Remove** → **Elements**:

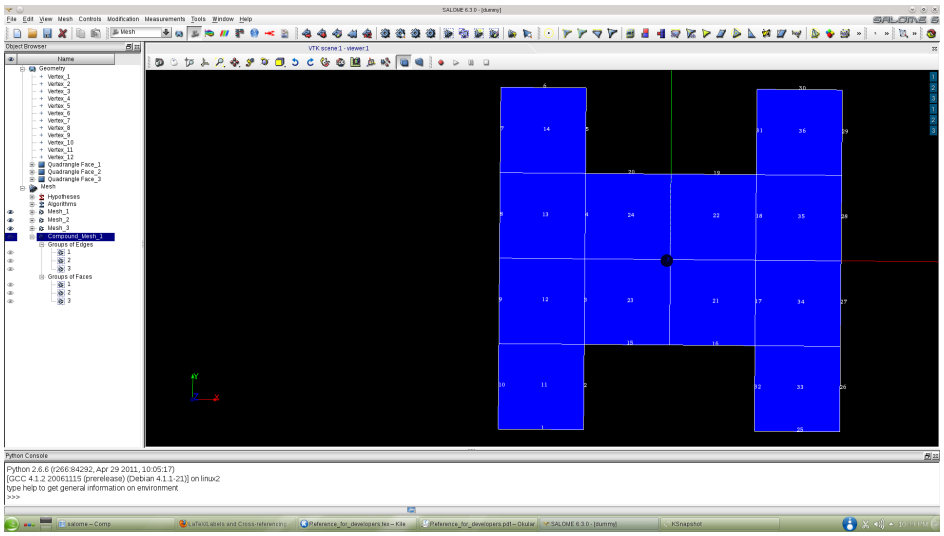

Figure 3.15

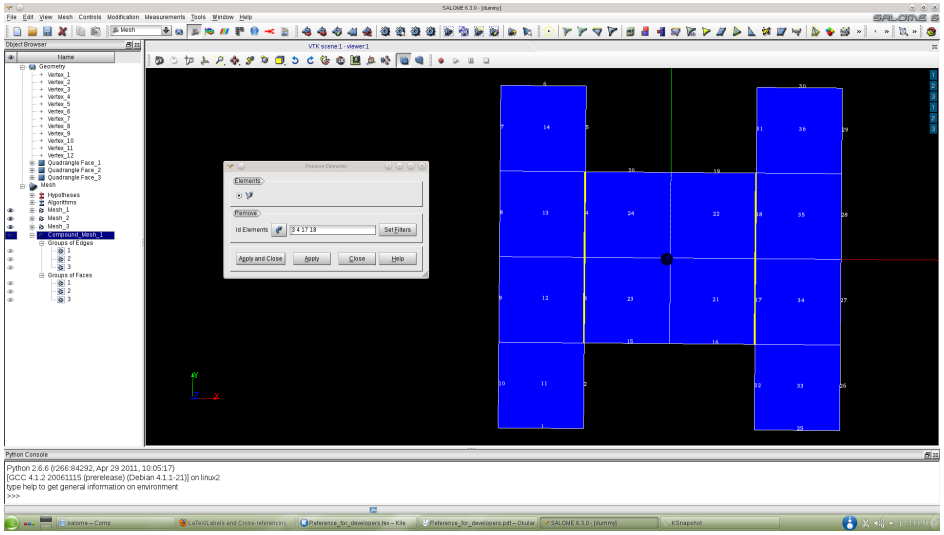

Figure 3.16

- The last step is Ctrl+U or **File** → **Export** → **UNV File**. Once we export the whole mesh into an UNV file, we can use it for the computational purposes (see the respective FCST tutorial).

3.2.2 Meshing with Hexotic

Hexotic is a fully automatic hexahedral mesh generator. It allows one to generate 3D meshes of complex geometries, without it we would be unable to meshes many things. Hexotic is a commercial piece of software, so users must activate it with a correct license key before they may use it - contact support@distine.com for more information. From experience it is often also necessary to use a proprietary 2D meshing algorithm from distine know as BLSURF.

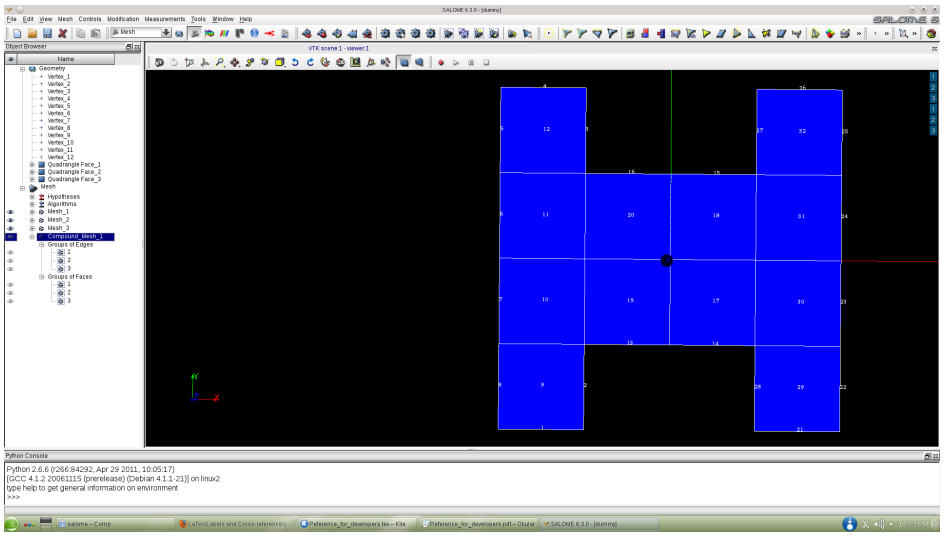

Figure 3.17

To mesh using Hexotic enter Salome's meshing view. Select the 3d geometry you wish to mesh. From the drop down menu "Mesh" select "Create Mesh". From the 3D menu select Hexotic. Hexotic parameters may be changed by editing the algorithm's hypothesis. From the 2D menu select BLSURF. Notable parameters for the BLSURF algorithm that one may wish to edit in order to achieve a better fitting mesh are the Min and Max "Physical Size" parameters.

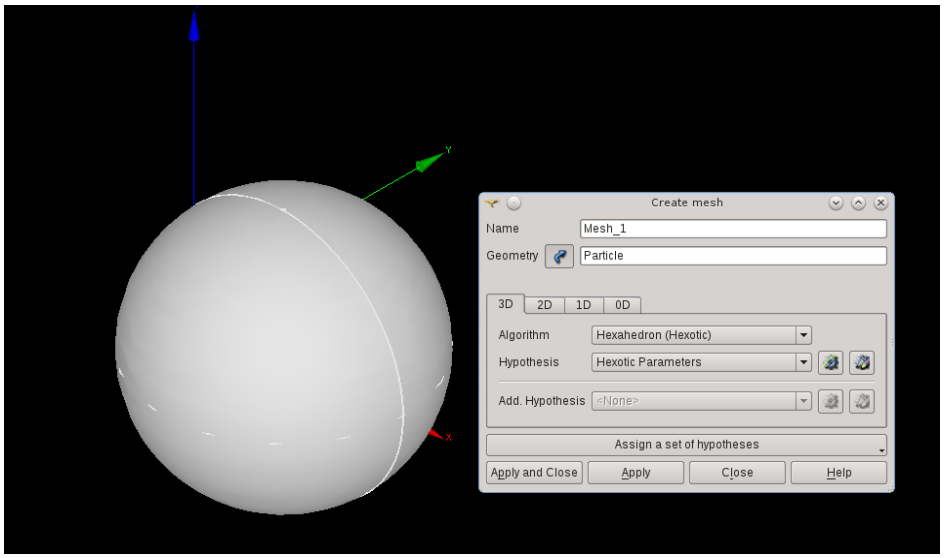

Figure 3.18: Meshing Dialogue Interface

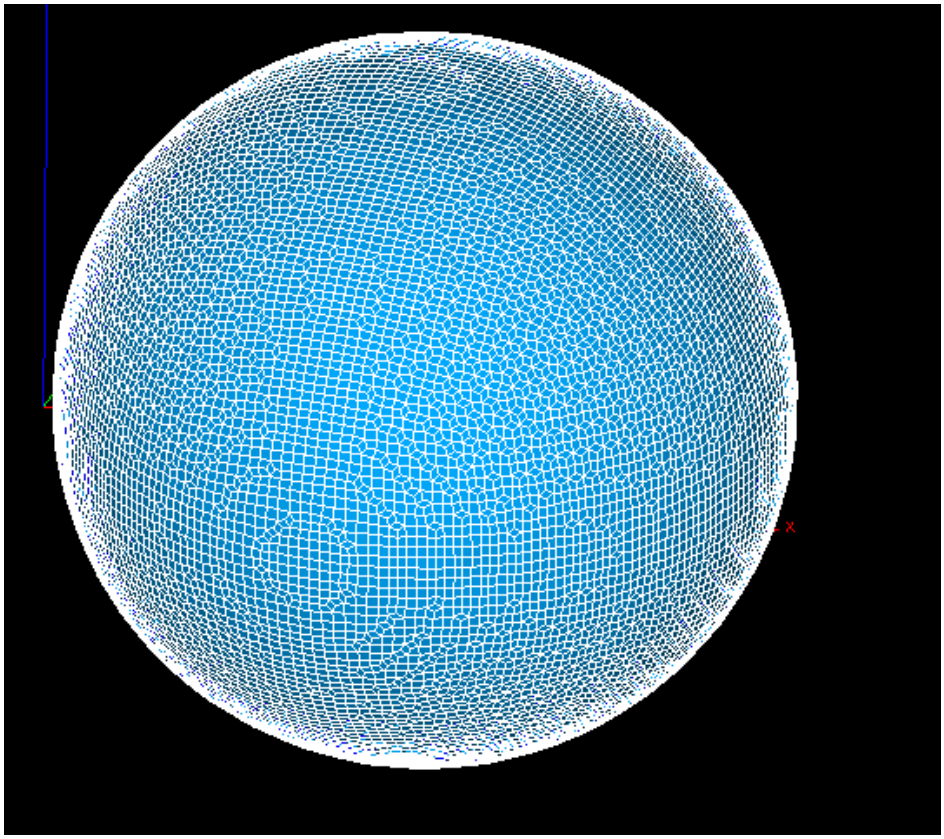

Figure 3.19: Mesh produced using Hexotic and BLSURF

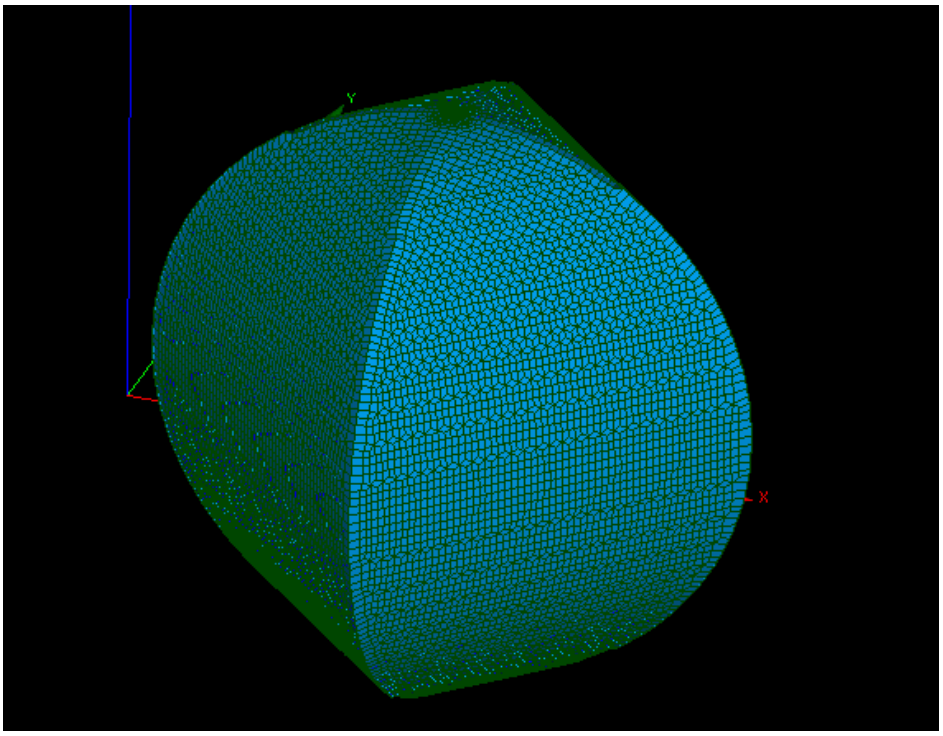

Figure 3.20: The same object as before, meshed using an inferior 2D algorithm.

3.3 Salome meshing using python scripts

3.3.1 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 are included in the pre processing folder, they can be used to create various meshes of various geometries.

Meshing scripts are run through Salomes text user interface (TUI). Loading scripts can be done simply via the file drop down menu, load script. Meshing scripts are written in the 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 it's 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, Phil recommends Dive Into Python.

3.3.2 Scripting Examples

```
import smesh, geompy, SMESH
import SALOMEDS
```

The above lines import the necessary Salome packages that will be required to create geometries and meshes. *smesh* is used for 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.

```
def makeRectanglarMesh(self, width, height):

    #Create vertices to describe rectangle
    Vertex_1 = geompy.MakeVertex(dList[0], dList[1], dList[2])
    Vertex_2 = geompy.MakeVertex(dList[0] + width, dList[1], dList[2])
    Vertex_3 = geompy.MakeVertex(dList[0] + width, dList[1] + height, dList[2])
    Vertex_4 = geompy.MakeVertex(dList[0], dList[1] + height, dList[2])

    #Make rectangle geometries
    rect = geompy.MakeQuad4Vertices(Vertex_1, Vertex_2, Vertex_3, Vertex_4)

    #Create mesh object of rectangular geomtery
    Mesh_1 = smesh.Mesh(rect)

    #Set 1D meshing algorithm
    Regular_1D = Mesh_1.Segment()
    Local_Length_1 = Regular_1D.LocalLength(self.meshDensity)
    Local_Length_1.SetPrecision( 1e-07 )

    #Set 2D meshing algorithm
    Mesh_1.Quadrangle()

    #Compute and return
    Mesh_1.Compute()
    return Mesh_1
```

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

```
def delInternalEdges(self):
    'This function deletes internal edges from self.compoundMesh'

    #Create a search filter to find free borders of the mesh
    search_filter = smesh.GetFilter(smesh.EDGE, smesh.FT_FreeBorders)
    external_edges = self.compoundMesh.GetIdsFromFilter(search_filter)

    #Get a list of all edges
    all_edges = self.compoundMesh.GetElementsByType(SMESH.EDGE)

    edges_to_remove = []

    #The difference between the external_edges list and all_edges list will be the internal
        edges.
    #The following loop iterates through the all_edges list, comparing it wil the
        external_edges list.

    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:

1. Open the Salome GUI
2. Perform the necessary steps using the GUI to generate desire geometries and/or surfaces
3. Use the "Dump Study" facility, accessed from the File menu, this will 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.

Chapter 4

Running FCST

4.1 Fuel Cell Analysis Using FCST

Analysis in FCST is the evaluation of a single data point on the polarization curve, it is usually carried out when quick evaluations are required. There are two files required when carrying out an analysis.

1. `main_app_param_file.prm`
2. `data_app_****.prm`

As this section is focused on how to run a simulation the details of these files will be further discussed in section 4.2. For now it is sufficient to know that `main_app_param` file is used to select the application you want to run (`cathode, MEA ...`) and the `data_app` file is used to hold all the parameters that describe the fuel cells physical properties (*Porosity, Platinum & Nafion Loading, ...*), dimensions, and operating conditions (*Relative Humidity, Temperature, ...*).

In order to run an analysis using FCST we need to open a terminal window on the desktop. Once this has been done we then need to navigate to the file that holds the `main_app_param` & `data_app` files of interest. In our example below we will run a simple MEA analysis. To do this we must go to the `mea` folder which is found in FCST's `data` folder. An illustration of this can be see below in image 4.1.

```
1 $~/fcst/data/mea/analysis/>
```

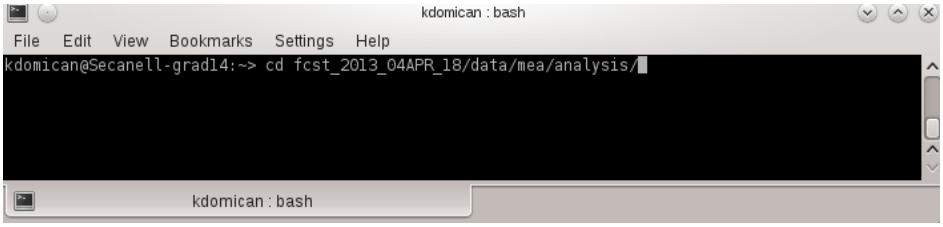

Figure 4.1: Terminal Window accessing MEA Analysis Model

Once the user has located the file of interest he/she is then required to "*drill back*" (*../../*) to the executable file (`fuel_cell-2d.bin`) located in the `lib` folder. An example of this can be seen in figure 4.2. Then finally hitting enter to run the code.

```
1 $~/fcst/data/mea/analysis> ../../../lib/fuel_cell-2d.bin main_app_pemfc.prm
```

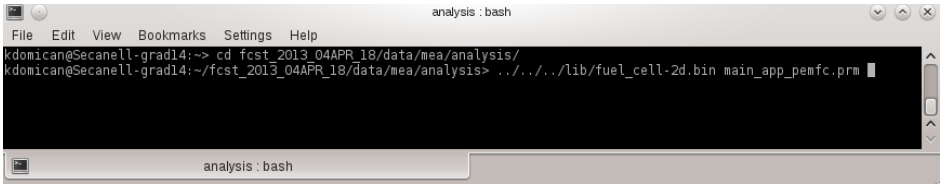

Figure 4.2: Terminal Window Drilling back to the executable file

In the above case as we are running the program in the `/fcst/data/mea/analysis`; therefore all the results will be outputted there.

FCST has several applications (See structure of the code below for details). By default, FCST solves the model of a fuel cell cathode (for the details on the model see M. Secanell et al., Electrochimical Acta, 52(7):2668-2682, February 2007). Sample input files to run this application are in `/data/cathode`.

Figure 4.3 illustrates the individual steps during a fuel cell analysis. This will be used as a building block later when looking at optimization in FCST.

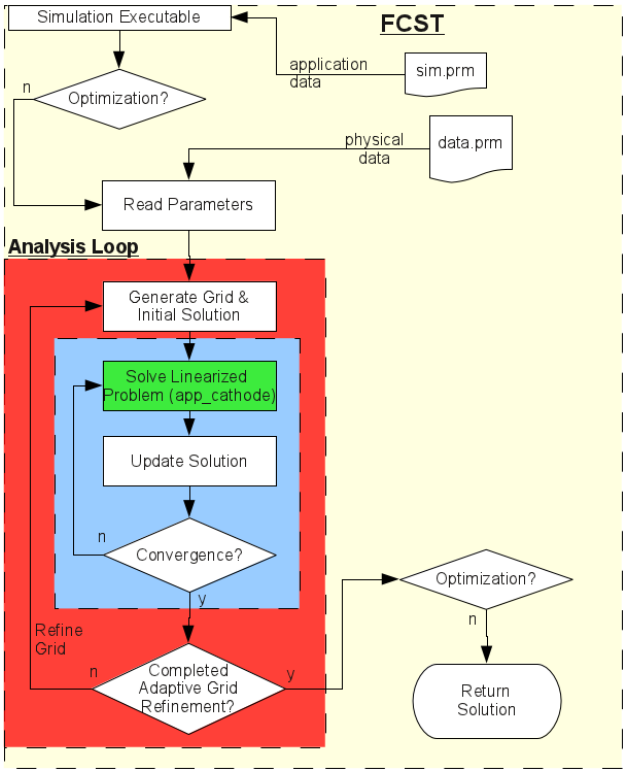

Figure 4.3: Schematic of Fuel Cell Analysis Code

4.2 Fuel Cell Parametric Study using FCST

When running a parametric study the user requires three files.

1. `main_app_param_file.prm`
2. `data_app_pemfc.prm`
3. `opt_app_parametric_default.prm`

4.2.1 Main Application File

1. `main_app_param_file.prm`

   The `main_app_param_file.prm` is the initial file accessed by FCST. Its purpose is to:

   (a) Assign the application (*line 2*) being run. In this case, MEA (Note: This application corresponds to the application implemented in `./fcst/source/app_pemfc.cc` and `./fcst/include/app_pemfc.h`).

   (b) Informs FCST on the files to access for data & optimization/parametric information (*line 3 & 4*).

   (c) Assign the type of solver to use (*line 5*). In this case, a 3 point parabolic Newton method.

   (d) And lastly a Boolean command which tells FCST whether we are utilizing DAKOTA (*line 6*).

```
1 subsection Simulator
2   set simulator name = MEA
3   set simulator parameter file name = data_app_pemfc.prm
4   set optimization parameter file name = opt_app_parametric_default.prm
5   set solver name = Newton3ppC
6   set Dakota direct = true
7 end
```

The Application / Simulator Name: The application is the simulation that the user would like to run.

For example, if the user wants to run the cathode side of the fuel cell to test or observe recent changes and their effects, the user might run a `cathode` model. However if the user is looking to run the entire fuel cell (MEA), they would select (`MEA`) as seen above in line 2. A list of other application options can be found in `simulation_selector.cc`.

```
1 cathode | MEA | test_mesh | laplace |. . .
```

Solver Name: The `solver name` tells FCST the type of solver that will be used to carry out the evaluation. These solvers are inherited from `APPFRAME` found in the `contrib` folder. In our case we have specified `Newton3ppC` which is also the default solver, it uses a three point parabolic function in order to determines step size for the next evaluation point. For a list of additional solvers utilized by FCST see `simulation_selector.cc`.

```
1 Linear | NewtonBasic | Newton3ppC | Newton3pp | NewtonLineSearch
```

Dakota direct: When running a simple *analysis* case we will set this to false and FCST will carry out one evaluation run. If on the other hand we are running an optimization or in our case a parametric study which has multiple point evaluations, therefore "Dakota direct" will be set to `true`.

4.2.2 Data Application File

2. `data_app_pemfc.prm`

   The `data_app_pemfc.prm` is a text file which is read by FCST in order to obtain the operating conditions and physical properties of the catalyst layer such as, Platinum & Nafion loading, porosity, and fuel cell dimensions. It also posses some instructions on the grid generation, adaptive refinement and how the output data should be formatted.

   Note: The example below has been included to give the reader an idea of the structure of a typical data file, and some of the information contained.

```
#####################################################################
#   $Id: $
#
#  This file is used to simulate a PEM electrode with
#  an agglomerate catalyst layer structure.
#
#   Copyright (C) 2008-13, Marc Secanell, Peter Dobson and Kailyn Domican
#
#####################################################################

subsection Grid generation

  set Type of mesh = PemfcMPL

  set Initial refinement = 2
  set Refinement         = adaptive  #global | adaptive
  set Sort Cuthill-McKee = false
  set Sort by component  = true

  ##############################################
  subsection Internal mesh generator parameters
    ####
    subsection Dimensions
      set Cathode current collector width [cm]          = 0.1         #[cm]
      set Cathode channel width [cm]                    = 0.1         #[cm]
      set Cathode CL thickness [cm]                     = 1.0e-3      #[cm]
      set Cathode MPL thickness [cm]                    = 5.0e-3      #[cm]
      set Cathode GDL thickness [cm]                    = 2.5e-2      #[cm]
      set Membrane thickness [cm]                       = 0.25e-2     #[cm] #NRE211
      set Anode CL thickness [cm]                       = 0.333e-3    #[cm]
      set Anode MPL thickness [cm]                      = 5.0e-3      #[cm]
      set Anode GDL thickness [cm]                      = 2.5e-2      #[cm]
      set Anode current collector width [cm]            = 0.1         #[cm]
      set Anode channel width [cm]                      = 0.1         #[cm]
    end
    ####
    subsection Material ID
      set Cathode GDL     = 2
      set Cathode MPL     = 3
      set Cathode CL      = 4
      set Membrane        = 5
      set Anode CL        = 6
      set Anode MPL       = 7
      set Anode GDL       = 8
    end
    ####
```

```
    subsection Boundary ID
      set c_Ch/GDL          = 2
      set c_BPP/GDL         = 1
      set c_GDL/CL          = 255
      set c_GDL/MPL         = 255
      set c_MPL/CL          = 255
      set c_CL/Membrane     = 255
      set Membrane/a_CL     = 255
      set a_CL/GDL          = 255
      set a_CL/MPL          = 255
      set a_MPL/GDL         = 255
      set a_GDL/BPP         = 3
      set a_GDL/Ch          = 4
    end
  end
end

################
subsection Adaptive refinement
  set Number of Refinements = 3                    # (default)
  set Output initial solution = true          # (default) false
  set Output intermediate solutions = true      # (default) true
  set Output intermediate responses = true      # (default) true
  set Output final solution = true             # (default) true
  set Output solution for transfer = true      # (default) false
  set Read in initial solution from file = true   # (default) true
end

#############################################################################
#############################################################################
subsection System management

 set Number of solution variables = 5

 subsection Solution variables
  set Solution variable 1 = oxygen_molar_fraction
  set Solution variable 2 = water_molar_fraction
  set Solution variable 3 = protonic_electrical_potential
  set Solution variable 4 = electronic_electrical_potential
  set Solution variable 5 = membrane_water_content
 end

 subsection Equations
  set Equation 1 = Ficks Transport Equation - oxygen
  set Equation 2 = Ficks Transport Equation - water
  set Equation 3 = Proton Transport Equation
  set Equation 4 = Electron Transport Equation
  set Equation 5 = Membrane Water Content Transport Equation
 end

end
#############################################################################
#############################################################################
subsection Discretization
  set Element = FESystem[FE_Q(2)^5] #System of 5 fe
  set Boundary fluxes = false
  set Interior fluxes = false

  subsection Matrix
   set Quadrature cell = -1
   set Quadrature face = -1
  end

  subsection Residual
   set Quadrature cell = -1
```

```
112     set Quadrature bdry = -1
113     set Quadrature face = -1
114   end
115 end
116 ##########################################################################
117 ##########################################################################
118
119 ###############
120 subsection Newton
121   set Debug residual     = false
122   set Debug solution     = false
123   set Debug update       = false
124   set Max steps          = 100
125   set Reduction          = 1.e-12
126   set Tolerance          = 1.e-9
127 end
128
129 ###############
130 subsection Fuel cell data
131
132   ######################################################################
133   subsection Operating conditions
134     set Temperature cell = 353                # [K]  #
135     set Cathode pressure = 101325             # [Pa] # 101325 (1.0 atm) |
136     set Cathode relative humidity = 0.7       # [%]  #
137     set Anode pressure = 101325                       # [Pa] # 101325 (1.0 atm) |
138     set Anode relative humidity = 0.7                 # [%]  #
139     set Voltage cell = 0.6                            # [V]  #
140   end
141   ######################################################################
142
143  ######################################################################
144   subsection Cathode gas diffusion layer
145
146     set Material id = 2
147     set Gas diffusion layer type = DesignFibrousGDL   #  DummyGDL | SGL24BA
148
149     subsection DesignFibrousGDL          #  DummyGDL | SGL24BA
150       ########## Composition: ##########
151       set Porosity = 0.6
152       ########## Gas transport ##########
153       ## Anisotropy
154       set Anisotropic transport = true          # (default) false
155       set Method effective transport properties in pores = Tomadakis  # (default)
              Bruggemann | Given | Percolation | Tomadakis | Mezedur
156       set Method effective transport properties in solid = Percolation  # (default)
              Bruggemann | Given | Percolation
157       ## XX
158       set Porosity threshold X = 0.11       # (default) 0.12 | 0.118 (Peters Thesis) |
              0.11
159       set Porosity network constant X = 0.785     # (default) 2.0  |0.785 (Peters
              Thesis)  [Page 69]
160       set Porosity gamma network constant X = 0.0     # (default) 0.0 |
161       #
162       set Electrical conductivity X = 16.03     # [S/cm]
163       set Solid network threshold X = 0.0       # (default) 0.12 |
164       set Solid network constant X = 1.5        # (default) 2.0 |
165       ## YY
166       set Porosity threshold Y = 0.11       # (default) 0.12 | 0.118 (Peters Thesis) |
              0.11
167       set Porosity network constant Y = 0.521     # (default) 2.0 |
168       set Porosity gamma network constant Y = 0.0     # (default) 0.0 |
169       #
170       set Electrical conductivity Y = 272.78      # [S/cm]
171       set Solid network threshold Y = 0.0       # (default) 0.12 |
```

```
172       set Solid network constant Y = 1.0        # (default) 2.0 |
173     end
174   end
175   ###################################################################
176   subsection Cathode microporous layer
177     set Material id = 3
178     set Micro porous layer type = DesignMPL       #  (default) SGL24BC | DesignMPL
179     subsection DesignMPL
180       set Porosity = 0.4              # From experimental data (manufacturers data) on
              Sigracel 24BC
181       set Anisotropic transport = false
182       ########## Pore network ##########
183       set Method effective transport properties in pores = Percolation
184       set Porosity threshold = 0.118
185       set Porosity network constant = 2.0
186       ########## Solid network ##########
187       set Method effective transport properties in solid phase = Percolation
188       set Electric conductivity = 88.84       # From experimental data (manufacturers
              data) on Sigracel 24BC
189       set Solid network threshold = 0.118
190       set Solid network constant = 2.0
191     end
192   end
193
194   ###################################################################
195   subsection Cathode catalyst layer
196
197     set Material id                                                          = 4
198     set Catalyst layer type                                                  =
          HomogeneousCL  #[ DummyCL | AgglomerateCL | HomogeneousCL ]
199
200     set Catalyst type                                                        = Platinum
201     set Catalyst support type                                                = Carbon Black
202     set Electrolyte type                                                     = Nafion
203     set Kinetics type                                                        =
          TafelKinetics
204
205     ###################
206     subsection Materials
207     ##
208     subsection Nafion
209       set Henrys Law Constant for Oxygen [Pa cm^3/mol]      = 3.1664e10
210       set Henrys Law Constant for Hydrogen [Pa cm^3/mol]    = 6.69e10
211       set Method to compute proton conductivity              = NRE211
212       set Method to compute water diffusion                  = Motupally
213       set Electro-osmotic drag method                        = Constant
214       set Electro-osmotic drag coefficient                   = 1.0
215       set Method for sorption isotherm                       = Hinatsu
216       set Method to compute enthalpy of sorption of water    = Constant
217       set Enthalpy of sorption of water [J/mol]              = 45000.0
218       set Oxygen diffusion coefficient [cm^2/s]              = 9.726e-6  # 9.726e-6 (
              Peters Thesis)
219     end
220     ##
221     subsection Platinum
222       set Cathodic transfer coefficient (ORR)               = 1.0   #  1.0 (Tafel
              Kinetics)
223       set Oxygen reaction order (ORR)                       = 1.0
224       set Reference exchange current density (ORR) [uA/cm2] = 2.47e-2
225       set Reference oxygen concentration (ORR)              = 0.725e-5  # 0.725e-5 (
              Tafel)
226      # set Method for kinetics parameters (ORR)              = Parthasarathy
227     end
228     ##
229     subsection Carbon Black
```

```
230         set Electrical conductivity [S/cm] = 88.84
231         set Density [g/cm^3]   = 1.25            # (default) 2.0 | 1.25 fit to General
                Motors data. With Modified Porosity threshold = 0.25884
232       end
233     end
234
235     ##
236     subsection ConventionalCL
237         set Platinum loading on support (%wt) = .46                  # 0.2 (ESDLab Ultra-
                thin CL) # 0.46 (Conventional CL) # 0.5 (GM)
238         set Platinum loading per unit volume (mg/cm3) = 400          # 400 (GM estimated
                width = .001[cm]) # 147.05 (ESDLab Ultra-thin width 0.00017 [cm]) # 125.0 (
                ESDLab Ultra-thin width 0.0002[cm] [ 125 = m_Pt/Cathode CL thickness])
239         set Electrolyte loading (%wt) = 0.30                         # 0.2 | 0.3 | 0.4 |
                0.5 (ESDLab testing)
240         set Active area [cm^2/cm^3] = 2.0e5                          # 2.0e5 (original) #
                 0.2, 67462.5 | 0.3, 42987.5 | 0.4, 62500 | 0.5, 60750  (ESDLab Ultra-thin CL
                )
241         ######### Pore network #########
242         set Method effective transport properties in pores = Percolation
243         set Porosity threshold = 0.3                                 # 0.3 (Peters Thesis)
                 # 0.118  | 0.25884 (Carbon Density = 1.25 [GM fitted])
244         set Porosity network constant = 4.0                          # 4.0 (Peters Thesis)
                 # 2.0
245         set Porosity gamma network constant = 0.0
246         ######### Solid network #########
247         set Method effective transport properties in solid phase = Percolation
248         set Solid network threshold = 0.118
249         set Solid network constant = 2.0
250         ######### Ionomer network #########
251         set Method effective transport properties in electrolyte phase = Percolation   #
                Bruggemann | Iden11 | Percolation | mesh_structure | Iden09
252         set Electrolyte network threshold = 0.0
253         set Electrolyte network constant = 2.0
254       end
255       ##
256       subsection AgglomerateCL
257         set Agglomerate composition = ionomer
258         set Agglomerate type = spherical
259         set Agglomerate solver = analytical
260         set Average current in cell = false
261         set Radius of the agglomerate [nm] = 100
262         set Agglomerate porosity = 0.3
263       end
264     end
265
266     ###################################################################
267     subsection Membrane layer
268       set Material id                                                = 5
269
270       set Membrane layer type                                        =
              NafionMembrane
271
272       set Electrolyte type                                           = Nafion
273       ####
274       subsection Materials
275         subsection Nafion
276           set Henrys Law Constant for Oxygen [Pa cm^3/mol]      = 3.1664e10
277           set Henrys Law Constant for Hydrogen [Pa cm^3/mol]    = 6.69e10
278           set Method to compute proton conductivity              = NRE211
279           set Method to compute water diffusion                  = Motupally
280           set Electro-osmotic drag method                        = Constant
281           set Electro-osmotic drag coefficient                   = 1.0
282           set Method for sorption isotherm                       = Hinatsu
283           set Method to compute enthalpy of sorption of water    = Constant
```

```
284         set Enthalpy of sorption of water [J/mol]                = 45000.0
285       end
286     end
287   end
288
289   ###################################################################
290   subsection Anode catalyst layer
291
292     set Material id = 6
293
294     set Catalyst layer type = HomogeneousCL    #[ DummyCL | AgglomerateCL |
          HomogeneousCL ]
295
296     set Catalyst type                                                  = Platinum
297     set Catalyst support type                                          = Carbon Black
298     set Electrolyte type                                               = Nafion
299     set Kinetics type                                                  =
          DualPathKinetics
300     ####
301     subsection Materials
302       subsection Platinum
303         #set Method for kinetics parameters (HOR)                = DualPathKinetics     #
              Required for Wangs Kinetics, rest parameters automatically calculated
304         set Reference hydrogen concentration (HOR)               = 0.59e-6              #
              Required for Dual Path Kinetics, rest parameters automatically calculated
305       end
306       subsection Carbon Black
307         set Density [g/cm^3]                                     = 1.25                 #
              GM data
308       end
309       subsection Nafion
310         set Henrys Law Constant for Oxygen [Pa cm^3/mol]         = 3.1664e10
311         set Henrys Law Constant for Hydrogen [Pa cm^3/mol]       = 6.69e10
312         set Method to compute proton conductivity                 = NRE211
313         set Method to compute water diffusion                     = Motupally
314         set Electro-osmotic drag method                           = Constant
315         set Electro-osmotic drag coefficient                      = 1.0
316         set Method for sorption isotherm                          = Hinatsu
317         set Method to compute enthalpy of sorption of water       = Constant
318         set Enthalpy of sorption of water [J/mol]                 = 45000.0
319       end
320     end
321     ####
322     subsection ConventionalCL
323       set Platinum loading on support (%wt) = .46          # 0.2 (ESDLab Ultra-thin CL)
            # 0.46 (Conventional CL)
324       set Platinum loading per unit volume (mg/cm3) = 400      # 400 (GM estimated width
            = .001[cm]) # 147.05 (ESDLab Ultra-thin width 0.00017 [cm]) # 125.0 (ESDLab
            Ultra-thin width 0.0002[cm] [ 125 = m_Pt/Cathode CL thickness])
325       set Electrolyte loading (%wt) = 0.30
326       set Active area [cm^2/cm^3] = 2.0e5                  # 1.2e5 (Experimental) | 0.3
            Nafion, 42987.5 (ESDLab Ultra-thin CL) | 2.0e5
327       ##
328       set Method effective transport properties in pores = Percolation
329       set Porosity threshold = 0.3
330       set Porosity network constant = 4.0              # 4.0 (Peters Thesis) | 2.0
331       set Porosity gamma network constant = 0.0
332       ##
333       set Method effective transport properties in solid phase = Percolation
334       set Solid network threshold = 0.118
335       set Solid network constant = 2.0
336       ##
337       set Method effective transport properties in electrolyte phase = Percolation
                # Bruggemann | Iden11 | Percolation *| mesh_structure | Iden09
338       set Electrolyte network threshold = 0.0
```

```
339       set Electrolyte network constant = 2.0
340     end
341     ##
342     subsection AgglomerateCL
343       set Agglomerate type = spherical
344       set Radius of the agglomerate [nm] = 100
345     end
346
347   end
348
349  ######################################################################
350   subsection Anode microporous layer
351     set Material id = 7
352     set Micro porous layer type = DesignMPL      #  (default) SGL24BC | DesignMPL
353     subsection DesignMPL
354       set Porosity = 0.4                # From experimental data (manufacturers data)
              on Sigracel 24BC
355       set Anisotropic transport = false
356       set Electric conductivity = 88.84       # From experimental data (manufacturers
              data) on Sigracel 24BC | 5.09
357       ########## Pore network ##########
358       set Method effective transport properties in pores = Percolation
359       set Porosity threshold = 0.118
360       set Porosity network constant = 2.0
361       ########## Solid network ##########
362       set Method effective transport properties in solid phase = Percolation
363       set Solid network threshold = 0.118
364       set Solid network constant = 2.0
365     end
366   end
367  ######################################################################
368   subsection Anode gas diffusion layer
369
370     set Material id = 8
371
372     set Gas diffusion layer type = DesignFibrousGDL   #  DummyGDL | SGL24BA
373
374     subsection DesignFibrousGDL          #  DummyGDL | SGL24BA
375       ########## Composition: ##########
376       set Porosity = 0.6
377       ########## Gas transport ##########
378       ## Anisotropy
379       set Anisotropic transport = true
380       set Method effective transport properties in pores = Tomadakis
381       set Method effective transport properties in solid = Percolation
382       ## XX
383       set Porosity threshold X = 0.11       # 0.118 (Peters Thesis) | 0.11
384       set Porosity network constant X = 0.785
385       set Porosity gamma network constant X = 0.0
386       #
387       set Electrical conductivity X = 16.03     # [S/cm]
388       set Solid network threshold X = 0.0
389       set Solid network constant X = 1.5
390
391     ## YY
392     set Porosity threshold Y = 0.11       # 0.118 (Peters Thesis) | 0.11
393     set Porosity network constant Y = 0.521
394     set Porosity gamma network constant Y = 0.0
395     #
396     set Electrical conductivity Y = 272.78      # [S/cm]
397     set Solid network threshold Y = 0.0
398     set Solid network constant Y = 1.0
399     end
400
401   end
```

```
402   ##################################################################
403   # END Fuel cell data
404 end
405
406 ###################################################################################
407 ###################################################################################
408 subsection Output Variables
409   set num_output_vars = 1
410   set Output_var_0 = cathode_current
411 end
412
413 ################
414 subsection Output
415   subsection Data
416     set Output format = vtk
417   end
418   subsection Grid
419     set Format = gnuplot     # eps # xfig
420   end
421 end
```

(a) `Grid generation` (*line 11-62*)

Type of mesh In the grid generation subsection the user is required to first specify the type of grid to be assigned to the simulation. Below is a list of possible options.

```
1 GridExternal | Cathode | Anode | CathodeMPL | Pemfc | PemfcMPL | Agglomerate_CL
```

Once the '`Type of mesh`' has been specified the user then indicates the number of initial refinements to be carried out (*line 15*). Figures 4.4, 4.5, & 4.6, illustrates a general refinement process. The refinement process can also be specified as `Global` or`Adaptive` (*line 16*). During adaptive refinement the sections that show the largest relative error will be further refined (As seen on the left hand side of Figure 4.7)

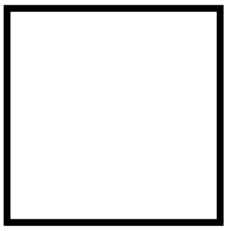

Figure 4.4: Initial Grid

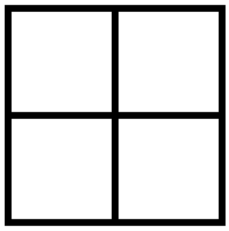

Figure 4.5: Grid 1^{st} Refinement

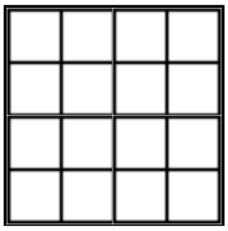

Figure 4.6: Grid 2^{nd} Refinement

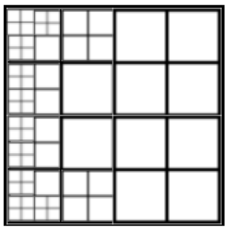

Figure 4.7: Grid Adaptive Refinement

GridExternal If the `GridExternal` option has been chosen the user is then also required to add the following lines to inform FCST as to where to obtain the unique grid.

```
1 set Type of mesh = GridExternal
2 set File name = test.unv    # Name of the mesh file
3 set File type = UNV         # Meshtype (see deal.ii supported mesh types)
```

i. `Internal mesh generator parameters`
This section is responsible for specifying:

A. `Dimensions` (GDL, Cathode,& Anode thicknesses ...)

B. `Material ID`

C. `Boundary ID`

(b) `Adaptive refinement` (*line 65-73*)

The user can also define the outputs (initial,intermediate solutions/responses, & final solutions) during the refinement steps.

The '`Output solution for transfer`' option, will produce a hidden file containing the current solution values. This solution can then be read in if '`Read in initial solution from file`' is set to true.

The '`Read in initial solution from file`' option gives the user the ability to read in previously obtained solutions. This is very beneficial when convergence becomes an issue.

Note: It is advised to have both '`Output solution for transfer`' & '`Read in initial solution from file`' set to `true` as it can improve convergence rates.

(c) `System management` (*line 77-97*)

The system management subsection is responsible for defining the `Solution variables` & `Equations` being used.

The current available variable options are:

i. X_{O_2} (Oxygen Molar Fraction)

ii. X_{H_20} (Water Molar Fraction)

iii. ϕ_m (Protonic Electrical Potential)

iv. ϕ_s (Electronic Electrical Potential)

v. λ (Membrane Water Content)

vi. T_{solid} (Temperature of Solid Phase) [Available in future releases]

(d) `Discretization` (*line 100-115*)

The `Discretization` section defines the order of the solver for the variables being solved for.

i. X_{O_2} (Oxygen Molar Fraction)

ii. X_{H_20} (Water Molar Fraction)

iii. ϕ_m (Protonic Electrical Potential)

iv. ϕ_s (Electronic Electrical Potential)

v. λ (Membrane Water Content)

Above in our example we have the following line (*line 101*)

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

Where;

- $FE\_Q(2)$ refers to the order of the solver in our case quadratic. This can also take any integer form 1-6.
- $FE\_Q(2)^5$. 5 refers to the number of variables included in the quadratic category.
  For example, if we wanted to have the first two elements(X_{O_2},X_{H_20}) solved cubically and the last three (ϕ_m,ϕ_s, & λ) solved linearly we would insert the following line.

$$FESystem[FE\_Q(3)^2 - FE\_Q(1)^3]$$

The final two sections in System management are:

i. `Matrix`

ii. `Residual`

These subsections 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, no. 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, would be sufficient to achieve an exact solution of the integrals.

(e) `Newton` (*line 120-127*)

Required to provide information to the Newton solver.

- Debug residual
- Debug solution
- Debug update

  The above debug options prints out the intermediate values for the residuals, solutions and the updates. This then can be used to locate errors/bugs in the code.

- Max steps
  Used to limit the number iterations carried out by the Newton solver.
- Reduction
- Tolerance
  Tolerance defines the decimal place accuracy of the Newton solver. Ideally this tolerance should be kept at $1.0e^{-9}$. However, in certain circumstances convergence at these 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.

(f) `Fuel cell data` (*line 130-403*)

In the Fuel cell data subsection all relevant properties pertaining to each respective layer and the operating conditions are specified.

i. `Operating conditions` (*line 133-140*)

ii. `Cathode gas diffusion layer`
A variety of gas diffusion layers may be chosen from. The current available options are:

A. `DesignFibrousGDL`

B. `DummyGDL`

C. `SGL24BA`

When changing between gas diffusion layers type (line 147 [cathode] & 372 [anode]) it is also required to change lines 149 [cathode] & 374 [anode], and the respective properties specific to the gas diffusion layer chosen.

iii. `Cathode microporous layer`
A variety of micro-porous layers may be chosen from. The current available options are:

A. `DesignMPL`

B. `SGL24BC`

When changing between micro-porous layers type (line 178 [cathode] & 352 [anode]) it is also required to change lines 179 [cathode] & 353 [anode], and the respective properties specific to the micro-porous layer chosen.

iv. `Cathode catalyst layer`
A variety of catalyst layers may be chosen from. The current available options are:

A. `DummyCL`

B. `HomogeneousCL`

C. `AgglomerateCL`

v. `Membrane layer`

A. `Materials` (Nafion)

vi. `Anode catalyst layer`
A variety of Catalyst layers may be chosen from. The current available options are:

A. `DummyCL`

B. `HomogeneousCL`

C. `AgglomerateCL`

vii. `Anode microporous layer`
A variety of micro-porous layers may be chosen from. The current available options are:

A. `DesignMPL`

B. `SGL24BC`

viii. `Anode gas diffusion layer`
A variety of gas diffusion layers may be chosen from. The current available options are:

A. `DesignFibrousGDL`

B. `DummyGDL`

C. `SGL24BA`

(g) `Output Variables` (*line 408*)
In this section it is possible to specify the number of output variables. In our above example we are only outputting one variable however if we are interested in printing additional variable onto the terminal screen we add the following format to the parameter file.

```
1 subsection Output Variables
2   set num_output_vars = 3
3   set Output_var_0 = cathode_current
4   set Output_var_0 = water_cathode
5   set Output_var_0 = anode_current
6 end
```

The remaining `Output Variables` subsections are used to specify the file format.

i. `Data`

ii. `Grid`

Additional sections of the data file have been continued here in order to give the reader an idea of good coding practices and the benefits of following these guidelines.

Cathode microporous layer: Below we can see that for the data file the values for porosity and electric conductivity are supported by a short piece of text specifying where these values have been obtained from.

```
subsection Cathode microporous layer
  set Porosity = 0.4                    # 0.4 (Manufacturers Data) on Sigracel 24BC
  ######### Solid network #########
  set Electric conductivity = 88.84     # 88.84 (Manufacturers Data) on Sigracel 24BC
end
```

Materials: Another example of this can be seen in the Materials section. We should note that the `Method to compute proton conductivity` has information on the default option and also the other options available to the user. We can also see that the `Oxygen diffusion coefficient` value has also been referenced to Peter Dobson's thesis (2011).

```
subsection Materials
  subsection Nafion 1100
    set Henrys Law Constant for Oxygen [Pa cm^3/mol] = 3.1664e10
    set Method to compute proton conductivity = NRE211  # (default) NRE211 | Constant
    set Oxygen diffusion coefficient [cm^2/s] = 9.726e-6  # 9.726e-6 (Peters Thesis,
        2011)
  end
```

The benefits of this type of text formatting is;

- The user is able to see what methods are available without having to search the code. Also if the user knows they are not going to change the method or values from the default (*which is easily on view*) it allows them to delete the text from their own personal file.

  The advantages of this is a reduced text file allowing the user to navigate more easily and a reduction in the probability of accidentally changing methods or variables located under different subsections.

- The user can now locate the referenced data to check whether that values is acceptable under the data files operating conditions. However this is not easily done if the value has not been referenced.

  This is not a major issue if the person who has implemented the value still remains contactable, however if they are not it is a timely procedure in gaining the relevant information (*Providing contact information is still correct*)

- By incorporating the reference text it ensures redundancy on the values. During the process of changing the data file sometimes keys on the keyboard can be accidentally hit causing a change to one of the values. The value will then go unnoticed if the effects are small. This can cause problems with the codes results later and will require the user to go through each value checking them individually. However if the reference text is available with the original value, it decreases the scrutiny time greatly and reduces the chance of errors entering the text files.

4.2.3 Parameter/Optimization Application File

3. `opt_app_parametric_default.prm`

The `opt_app_parametric_default.prm` is used when carrying out parametric studies.

```
#############################################################################
#
#  This file is used to run a multi-dimensional parametric study.
#  See end of file for list of possible design variables.
#
```

```
#############################################################################

subsection Optimization Parameters

#### NOTE THAT THIS SECTION ONLY EXISTS WHEN RUNNING IN OPTIMIZATION MODE ###
#####------------------------------------------------------------------------###
    subsection Optimization Program Options
      set Use dakota input file = false       # (default) false
      set Dakota_Input_File = dakota_input.in     # not needed if -Use dakota input
          file = false-

      set Optimization method = multidim_parameter_study  # multidim_parameter_study |
          optpp_q_newton | nl2sol | ncsu_direct
    end

    subsection Design Variables
      set num_design_variables = 1       # 2
      set DV_0_name = V_cell             # P_cell
      set DV_1_name = T_cell             # P_c    | RH_a
      set DV_2_name = prc_Pt_c           # RH_c   | prc_Pt_c

      #######   Lower Bound        #######
      ####### lb < -1e30 for -inf #######
      #-----------------------------------#
  set DV_0_lb = -1.1       # V # Changed to -1.1, force dekota to start at -1.1
  set DV_1_lb = 303        # K #
  set DV_2_lb = 0.2        # % #

      #######  Upper Bound      #######
      ####### ub > 1e30 for inf #######
      #-----------------------------------#
  set DV_0_ub = -0.1       # V #
  set DV_1_ub = 353        # K #
  set DV_2_ub = 0.5        # % #

      #######              Parameter Study Partitions              #######
      ### NOTE: Evaluated at n+1 points between lower and upper bound ###
      ###-------------------------------------------------------------------###
  set DV_0_partition = 50
  set DV_1_partition = 8
  set DV_2_partition = 10
      end

  subsection Responses
    set num_objectives = 1
    set num_nl_constraints = 0      # (default) 0
    set num_eq_constraints = 0      # (default) 0

    set RESP_0_name = current
    end
  end
```

Located at the bottom of all `opt_app` files in both parametric & optimization is a list of design variables available for the user to carry out a parametric studies or optimization. As of **1-SEP-2013** the following table lists the current parameters that can be passed to DAKOTA for parametric studies/ optimization.

If the user requires additional variables for parametric/optimization studies, modification of the `dakota_application.cc` file should be carried out.

```
      ########## List of Possible Design Variable Names ##########
      ##########-------------------------------------------#########
# //    Conventional_CL.cc
```

```
4  # V_Pt_c | V_Pt_a   // Platinum loading per unit volume [mg/cm3]  (Cathode | Anode)
5  # prc_Pt_c | prc_Pt_a     // Platinum loading on support [%wt]    (Cathode | Anode)
6  # prc_N_c | prc_N_a       // Electrolyte loading [%wt]      (Cathode | Anode)
7  # Av_c | Av_a             // Active area [cm^2/cm^3]      (Cathode | Anode)
8
9  # //    Agglomerate_CL.cc
10 # r_agg_c | r_agg_a     // Radius of the agglomerate [nm]     (Cathode | Anode)
11 # r_agg                // Radius of the agglomerate [nm] **possibly redundant**
12 # epsilon_agg_c | epsilon_agg_a   // Agglomerate porosity       (Cathode | Anode)
13 # epsilon_agg                 // Agglomerate porosity **possibly redundant**
14
15 # //    Operating_Conditions.cc
16 # V_cell          // Cell Voltage
17 # T_cell          // Cell Temperature
18 # dV_a            // Voltage drop in the Anode
19 # P_c | P_a       // Pressure           (Cathode | Anode)
20 #
21 # RH_c | RH_a       // Relative Humidity        (Cathode | Anode)
22 # OCV               // Open Circuit Voltage
23
24 # //    Geometries.cc
25 # L_CCL | L_ACL         // CL thickness         (Cathode | Anode)
26 # L_CGDL | L_AGDL       // GDL thickness        (Cathode | Anode)
27 # L_CMPL | L_AMPL       // MPL thickness        (Cathode | Anode)
28 # Ch_width              // Channel Width        (Cathode | Anode)
```

Optimization Program Options: The `Optimization Program Options` of the `opt_app` parametric file is responsible for telling FCST whether it is required to formulate its own `dakota_input.in` file or if you are supplying DAKOTA with a predefined input file (*line 13 & 14*).

"Use dakota input file" & "Dakota_Input_File": If `Use dakota input file` is set to `false` then FCST will pass on the information specified in the `opt_app_parametric_default.prm` file and DAKOTA will print out a new `dakota_input.in` at run time. If however it is set to `true` we are telling FCST that we have already specified an input file and that DAKOTA should use this directly rather than reading the information from the rest of the `opt_app` file.

Note: For completeness ``Use dakota input file'' & ``Dakota_Input_File'' have been included in the default parametric file, however, when the user is not using their own `dakota_input.in` file both line 13 & 14 can be deleted.

Given that in most cases the user specifies all the parametric & optimization information in the `opt_app` file. The following descriptions will be relevant for cases when `Use dakota input file = false`.

Optimization Method: The `Optimization method` command is used to specify the type of study that is being carried out (optimization, parametric study, least squares fit, ...) for additional information on `Optimization methods` see section 4.3. In our case we are looking to carry out a parametric study so the `multidim_parameter_study` should be specified.

Design Variables: In the design variables section (*line 19-45*) we specify the number of design variables that we want to change (*line 20*), the upper and lower bounds for that variable (*line 25-37*), and the number of points that we want to evaluate between the upper and lower bounds (*line 39-45*).

num_design_variables: In the example above we have specified one design variable `V_cell` for a single parametric study. The corresponding upper, lower bounds, and partitions can be found at line 28, 35, and 42.

If the user wants to conduct a multi-dimensional parametric study we would simply change `num_design_variables` value from one to whatever number of variables required. In the example about we have the capabilities of increasing the number of variables to three. If the user requires more variables than this the user can simply add additional `DV_#_name` and the corresponding upper, lower bound and partitions.

Note: The upper and lower bound of the voltage have been set to negative. This is because DAKOTA will vary its parameters from the lowest value to the highest value (In the non-negative case this is from 0.1 - 1.1 [V]).

During the solving process FCST uses the last mesh data and node values as the initial starting point for the next point evaluation. As the function evaluations become more difficult as we enter the mass transport region (`V_cell of 0.3 - 0.1`) the time taken to evaluate these points is much longer. If we change the voltage values to their negative the parametric study will go from 1.1 to 0.1 [V], this in turn decreases the solving time and allows the solver to use the previous values as appose to starting at the 0.1 [V] (the most difficult case).

Additional advantages as well as reduced time is that in some cases if the solver begins at lower voltages (e.g. 0.1 [V]) the solver is unable to to converge due to the low oxygen values however if the solver starts at the 'easier case' (high voltages 1.1 - 0.8 [V]) it will carry on the previous solutions and be able to converge at the lower voltages.

Responses: The response section of the `opt_app` parametric file, specifies the number of outputs desired in the `dakota_tabular.dat` data file, in our case there is ONLY ONE objective value (*Current Density [A/cm^2]*) line 52.

It also is responsibly for specifying the type and number of constraints. There are two types of constraints; Equality (*line 49*) and Inequality (*line 50*), in general we do not typically use constraints in parametric studies so this section will be covered in more detail in the **optimization section**.

4.3 Optimization using FCST

When running an Optimization study the user requires three files, as seen above with parametric studies. The only difference however is that we change out(*alter*) the third file to an optimization file/format.

`opt_app_optimization_default.prm`

```
1  #############################################################################
2  #
3  #  This file is used to run the optimization interface.
4  #  See end of file for a list of optimization variables.
5  #
6  #############################################################################
7
8  subsection Optimization Parameters
9
10 #### NOTE THAT THIS SECTION ONLY EXISTS WHEN RUNNING IN OPTIMIZATION MODE ###
11 ####-------------------------------------------------------------------------###
12     subsection Optimization Program Options
13       set Use dakota input file = false       # (default) false
14       set Dakota_Input_File = dakota_input.in
15
16       set Optimization strategy = single_method      # single_method | multi_start |
             pareto_set | hybrid
17       set Optimization method = optpp_q_newton       # (default) optpp_q_newton | nl2sol |
             ncsu_direct
```

```
18
19
20        ######### Method Independent Parameters #########
21        #########-------------------------------#########
22    set Maximum iterations = 200                    # (default) 100
23    set Maximum function evaluations = 2000         # (default) 1000
24    set Constraint tolerance = 1.0e-4               # (default) 1.0e-4
25    set Convergence tolerance = 1.0e-4              # (default) 1.0e-4
26
27        ######### Numerical Gradient Parameter #########
28        #########-------------------------------#########
29    set Numerical gradients = true                # (default) false | true
30    set Numerical gradient type = central         # (default) forward | central
31
32
33        ######### Method Specific Parameters #########
34        #########             OPT++             #########
35        #########-------------------------------#########
36    subsection OPT++
37        set Gradient tolerance = 1.0e-4     # (default) 1.0e-4
38        set Steplength to boundary = 0.2    # (default) 0.9
39        set Centering parameter = 0.8       # (default) 0.2
40        set Merit function = argaez_tapia   # (default) argaez_tapia
41    end
42     end
43
44      subsection Design Variables
45    set num_design_variables = 1
46    set DV_0_name = L_CCL
47    set DV_1_name = prc_N_c
48
49       ######## Initial Point ########
50       ########------------########
51    set DV_0_ip = 1.65e-4
52    set DV_1_ip = 0.30
53
54       ########     Lower Bound     ########
55       ######## lb < -1e30 for -inf  ########
56       #-----------------------------------#
57    set DV_0_lb = 0.8e-4
58    set DV_1_lb = 0.20
59
60        ########  Upper Bound      ########
61        ######## ub > 1e30 for inf ########
62        #-----------------------------------#
63    set DV_0_ub = 10e-4
64    set DV_1_ub = 0.50
65
66        ######## Scales ########
67        ########-------########
68    set DV_0_scale_method = value       # none | auto | value | log
69    set DV_1_scale_method = value       # none | auto | value | log
70
71    set DV_0_scale = 1e-4
72    set DV_1_scale = 0.1
73
74        ######## Step size ########
75        ########---------########
76    set DV_0_step = 1e-5
77    set DV_1_step = 1e-4
78
79        end
80
81    subsection Responses
82      set num_objectives = 1
```

```
83      set num_nl_constraints = 3
84      set num_eq_constraints = 0
85
86      set RESP_0_name = current
87      set RESP_1_name = m_Pt_c
88      set RESP_2_name = epsilon_V_cat_c
89      set RESP_3_name = epsilon_N_cat_c
90      set RESP_4_name = epsilon_S_cat_c
91      set RESP_5_name = L_CCL
92
93        ######## Response Numbers must match ########
94        ########    Constraint Lower Bound     ########
95        ########    lb < -1e30 for -inf        ########
96        ########-------------------------------########
97      set RESP_2_lb = 0.118
98      set RESP_3_lb = 0.118
99      set RESP_4_lb = 0.118
100     set RESP_5_lb = 0.8e-4          # (ESDLab, Ultra-thin CCM, = 2 microns) 2e-4
101
102       ######## Constraint Upper Bound ########
103       ########    ub > 1e30 for inf     ########
104       ########-------------------------########
105     set RESP_2_ub = 1.0
106     set RESP_3_ub = 1.0
107     set RESP_4_ub = 1.0
108     set RESP_5_ub = 2e-4            # (ESDLab, Ultra-thin CCM, = 2 microns) 2e-4
109
110       ######## Equality Constraint ########
111       ########--------------------########
112     set RESP_1_eq = 350
113       end
114   end
```

In the above example of a `opt_app_optimization` file we will note that many of the variables have been seen earlier in the `opt_app_parametric` file. These next sections will look at describing the additional changes and variables applicable to optimization in FCST.

Optimization Method: The `Optimization method` command is used to specify the type of study that is being carried out. There area

1. `single_method`

   The `single_method` is selected when the user is running parametric studies or optimization where they require only one optimization method.

2. `multi_start`

   The `multi_start` method will restart the optimization multiple times specified by the user.

3. `pareto_set`

   The `pareto_set` method is only utilize during multi-objective optimization (4.4).

4. `hybrid`

   The `hybrid` method uses additional optimization methods. An example of this would be to use a global method to locate an area in the entire feasible region. Then once a sufficient criteria has been met the optimization method will be changed to a local method in order to take advantages of the high convergence rate.

Optimization Program Options: The `Optimization Program Options` consist of the same variables as seen in `opt_app_parametric` file however we also notice three additional Classifications:

1. Method Independent Parameters
2. Numerical Gradient Parameters
3. Method Specific Parameters

Method Independent Parameters: Consists of parameters that have no dependencies on the type of optimization method being used. This section tells FCST the maximum number of iterations & function evaluates (*line 22 & 23*)that can be carried out during optimization.

It also sets how strictly the method sticks to the constraints and the tolerance needed for convergence (*line 24 & 25*).

Note: Depending on the optimization problem, sometimes convergence issues can arise. One way to alleviate this issue is to relax the `Convergence tolerance` from the default $1.0e^{-4}$ to maybe $1.0e^{-3}$.

The same idea can be applied to the `Constraint tolerance` depending on how heavily constrained the problem is.

Numerical Gradient Parameters: Here is where we specify the type of gradient method we want to employ.

1. Numerical Gradients, as seen in the example (*line 30*)
2. Analytical Gradients

When using numerical gradients we also have an additional specification on whether we want to use *Forward* or *Central* differentiation (*line 31*).

As we can see from figure 4.8 using central differentiation is a much more accurate form of predicting the slop of a function. Having said this we must also take note of equations 4.1 & 4.2. In equation 4.2 we can see that we have doubled the function evaluations which in turn doubles the amount of time required to carry out the analysis.

In some cases when carrying out function evaluations they will be highly expensive or in some cases convergence can be an issue. In these cases although not ideal it is preferable to use *Forward* differentiation

1. **Forward**

$$\frac{\triangle f}{\triangle x} = \frac{f(x + \triangle x) - f(x)}{\triangle x} \tag{4.1}$$

2. **Central**

$$\frac{\triangle f}{\triangle x} = \frac{f(x + \frac{\triangle x}{2}) - f(x - \frac{\triangle x}{2})}{\triangle x} \tag{4.2}$$

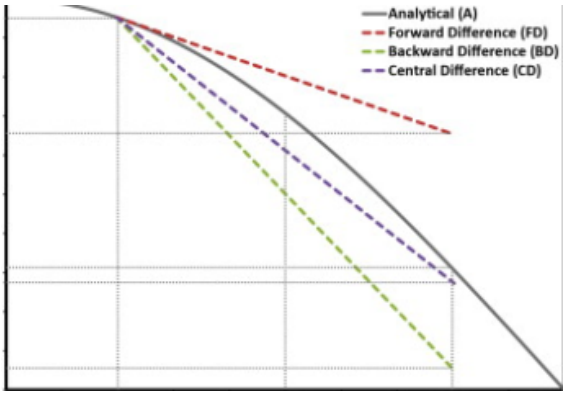

Figure 4.8: Comparison of Forward, Backward, & Central Differentiation

Method Specific Parameters: This section is specific to the method being used. In the above example it is specific to the $OPT++$ library. An additional example has been given below however if curious the reader is advised to see the default optimization methods located in:

```
$./data/cathode/optimization/optimization_methods_cathode/
```

or

```
$./data/mea/optimisation/optimization_methods_mea/
```

In the following short example we are using a method from the SCOLIB library, the `coliny_pattern_search` algorithm. In this case we would change the `Optimization method = coliny_pattern_search` as appose to `optpp_q_newton` (*line 17*).

We then would then replace (*line 33 - 41*) in the above `opt_app_optimization` file with the new method specific section.

```
    ######### Method Specific Parameters #########
    #########          SCOLIB (COLINY)          #########
    #########-----------------------------#########
subsection coliny_pattern_search
    set Initial Delta = 2              # (default) 1
    set Threshold Delta = 0.0001       # (default) 0.0001
end
```

Design Variables Section: The `Design Variables` Section is similar to the `opt_app_parametric` file except for two additional subsections.

1. Scales
2. Step size

Scales: The scales section has two specifications

1. `scale_method`

   The `scale_method` specifies whether you are going to specify no scale (`none`), `auto` scaling, `log`, or a `value`. In general it is good practice to specify a scale `value` as it allows the user to have a definite reference point, when using `auto` if there is a change in magnitude it will go unnoticed by the user in the final output solution.

2. `scale` value

   The scale value is the magnitude of the variable. For example if the variable is temperature we know that the scale is 100 as temperature is given in Kelvin (353 - 368 [K]). If its Nafion loading the scale is 0.1 as Nafion loading is a percentage (20 - 50 %).

Step Size: The step size refers to the $\triangle x$ in equations 4.1 & 4.2. Greater the step size the less computations that will be required, however this also means the greatest error as the error is proportional to $(\triangle x)^2$. Therefore there is a fine trade off between computational time and error.

Responses Section: The Responses section has changes slightly compared to the `opt_app_parametric` file as we are now considering constrained optimization. If the above example was unconstrained optimization there would be no difference between the `opt_app_parametric` and `opt_app_optimization` responses section.

There are two types of constraints:

1. Linear (Equality) Constraints (*line 83*)
2. Non-Linear Constraints (*line 84*)

In the above case we have three non-linear constrains and one linear constraint.

Non-Linear Constraints: Like the `Design Variable` section each nonlinear constraint requires a upper and lower bound (*line 97 - 108*). If no finite upper or lower bound is to be specified $1e^{30}$ or $1e^{-30}$ can be specified.

Linear (Equality) Constraints: Unlike Non-Linear Constraints, Equality constraints only require the response variable to equal a value (*line 112*).

4.4 Multi-Objective Optimization using FCST

To achieve multi-objective optimization we must first change three parameters.

1. `Optimization strategy` (*line 16*)
2. `num_design_variables` (*line 84*)
3. `num_objectives` (*line 82*)

Optimization strategy When carrying out multi-objective optimization we can no longer optimize for just one objective function this is especially the case when an improvement of one objective comes at the expense of another (*Performance & Cost*). In order account for the additional objective function we incorporate weighting factors which specifies the importance of one objective over the other. The weights are referred to a **Pareto Weights** or **Pareto Set**.

In FCST the default Pareto set is for two design variables. Figure 4.9 below shows the multi-objective weights for two design variables.

Pareto Set #	Obj_Function_1	Obj_Function_2
0	1.0	0.0
1	0.9	0.1
2	0.8	0.2
3	0.7	0.3
4	0.6	0.4
5	0.5	0.5
6	0.4	0.6
7	0.3	0.7
8	0.2	0.8
9	0.1	0.9
10	0.0	1.0

Figure 4.9: Pareto Set for 2 Design Variables

num_design_variables & num_objectives: Once the `Optimization strategy` has been set to `pareto_set` we then change both the num_design_variables & num_objectives to 2 or whatever number of design variables are specified. At present FCST like most multi-objective engineering problems, considers only two design variables however this can be easily modified by changing the default Pareto set found in `dakota_application.cc`.

4.5 DAKOTA Methods

The following list is all of the current DAKOTA Methods available as of **1-MAY-2013**. The methods are known to work with FCST and can be utilized. For detailed discriptions on the individual methods see DAKOTA manuals.

1. **asynch_pattern_search**
2. bayes_calibration
3. centered_parameter_study
4. **coliny_cobyla**
5. **coliny_direct**
6. **coliny_ea**
7. **coliny_pattern_search**
8. **coliny_solis_wets**
9. **conmin_frcg**
10. **conmin_mfd**
11. dace
12. dl_solver
13. dot
14. dot_bfgs
15. dot_frcg
16. dot_mmfd
17. dot_slp
18. dot_sqp
19. **efficient_global**
20. fsu_cvt
21. fsu_quasi_mc
22. global_evidence
23. global_interval_est
24. global_reliability
25. importance_sampling
26. list_parameter_study
27. local_evidence
28. local_interval_est
29. local_reliability
30. **moga**
31. **multidim_parameter_study**
32. **ncsu_direct**
33. **nl2sol**
34. nlpql_sqp
35. nlssol_sqp
36. nonlinear_cg
37. npsol_sqp
38. optpp_cg
39. **optpp_fd_newton**
40. **optpp_g_newton**
41. optpp_newton
42. **optpp_pds**
43. **optpp_q_newton**
44. polynomial_chaos
45. **psuade_moat**
46. richardson_extrap
47. sampling
48. **soga**
49. stanford
50. stoch_collocation
51. surrogate_based_global
52. surrogate_based_local
53. vector_parameter_study

4.6 Fuel Cell Design & Optimization Using FCST

As we've seen above FCST also has the capabilities to perform optimization studies. Any application that is inherited from `OptimizationBlockMatrixApplication` has the appropriate interface to be used for optimization studies. Information on how to run optimization can be found in sections 4.3 & 4.4.

To perform optimization studies, FCST interfaces with the open source libraries DAKOTA developed by Sandia National Laboratory. For more information about the DAKOTA library please click here. The FCST developers have developed an interface so that DAKOTA and FCST can interact seamlessly.

4.6.1 FCST classes that interact with DAKOTA (Developers Only)

Interaction between FCST and DAKOTA is achieved by using `simulation_builder` which will call the `run_optimization()` function.

When FCST is run as seen in figure 4.3 the FCST code is called on once in order to run a specific data point. However in parametric or optimization studies we require multiple points to be evaluated. This requires the use of the DAKOTA libraries in order to change the variables after each iteration. The two main files used to interface with DAKOTA are:

1. `dakota_direct_interface`
2. `dakota_application`

Once the initial stages of the code have been carried out by `simulator_builder`, `simulator_selector`, and `dakota_application`, declaring and initialing all the variables from the `main_app_`, `data_app_`, & `opt_app_` files. The `main.cc` file then proceeds to the `run()` function in `simulator_builder.cc` (see below) in order to run the simulation.

In the `run()` function we can see in line 10 where the code checks to see if its running an analysis or parametric/optimization study, as explained in 4.2.1. During parametric/optimization studies the code will enter line 12 and proceed to the `run_optimization()` function in `simulator_builder.cc`.

```
1  template<int dim>
2  void SimulatorBuilder<dim>::run()
3  {
4    timer.restart();
5
6    if (run_tests) run_test();
7    else
8    {
9      if (dakota_use || dakota_direct)
10     {
11       run_optimization();
12     }
13     else
14     {
15       //-- Select the application you want to run:
16       app_lin = sim_selector->select_application();
17       //-- Select the solver you want to run:
18       newton = sim_selector->select_solver(app_lin.get());
19       //-- Select the solving method you want to run, e.g. adaptive refinement:
20       solver = sim_selector->select_solver_method(app_lin.get(), newton.get());
21       // Here we have collected all information:
22       deallog << "Run program using input file: " << simulator_parameter_file_name << std::
             endl;
23       deallog.pop();
24       solver->solve(simulator_parameter_file_name, param);
25       timer.stop();
26     }
27   }
28
29   timer.stop();
```

```
30    deallog.push("MAIN");
31    deallog << "The program was executed in: " << timer.wall_time() << " seconds " << std::
          endl;
32    deallog << "=============== END ====================" << std::endl;
33    deallog.pop();
34
35  }
```

The main points to note once we enter the `run_optimization()` function are:

1. Is DAKOTA running in Parallel or Series? (*line 8*)

   As of **1-MAY-2013** Series is the only option available. This may change in the future.

2. Are we running a Non-Linear Least Squares (NLS) method or standard parametric/optimization routine? (*line 16-25*)

Note: These are questions that are answered in the `opt_app_` file explained earlier in section 4.3.

Once these have been specified the code will execute the `run()` function (*line 28*), which begins the iterative loop until the parametric study has been complete or the stopping criteria have been met in optimization.An illistration of this can be see in figure 4.10 taken from Peter Dobson's 2012 paper.

```
1  void SimulatorBuilder<dim>::run_optimization()
2  {
3    deallog.pop();
4    if (dakota_direct)
5    {
6      // NOTE: Must declare these in order for parameter handler to not complain when reading
           the parameter file specified.
7      //        Not exclusively required for dakota application to run.
8      Dakota::ParallelLibrary parallel_lib;
9      shared_ptr<Dakota::ProblemDescDB> problem_db(new Dakota::ProblemDescDB (parallel_lib));
10     SIM::DakotaApplication optimization(problem_db, optimization_parameter_file_name);
11     optimization.declare_parameters(param);
12     optimization.manage_inputs(param);
13
14     Dakota::DirectApplicInterface* optimization_interface;
15
16     if (optimization.use_NLS())
17     {
18       deallog<<"Entering DakotaLeastSquaresInterface"<<std::endl;
19       optimization_interface = new SIM::DakotaLeastSquaresInterface<dim> (optimization,
             problem_db, param, sim_selector, simulator_parameter_file_name);
20     }
21     else
22     {
23       deallog<<"Entering DakotaDirectInterface"<<std::endl;
24       optimization_interface = new SIM::DakotaDirectInterface<dim > (optimization,
             problem_db, param, sim_selector, simulator_parameter_file_name);
25     }
26
27     optimization.assign_interface(optimization_interface);
28     optimization.run();
29
30     deallog << "Optimization completed" << std::endl;
31 .
32 .
33 .
```

If running a standard parametric/optimization routine, the following `dakota_direct_interface` function will be used.

```
template <int dim>
int DakotaDirectInterface<dim>::derived_map_ac(const Dakota::String& ac_name)
```

If running a Non-Linear Least Squares (NLS) method, the following `dakota_direct_interface` function will be used.

```
template <int dim>
int DakotaLeastSquaresInterface<dim>::derived_map_ac(const Dakota::String& ac_name)
```

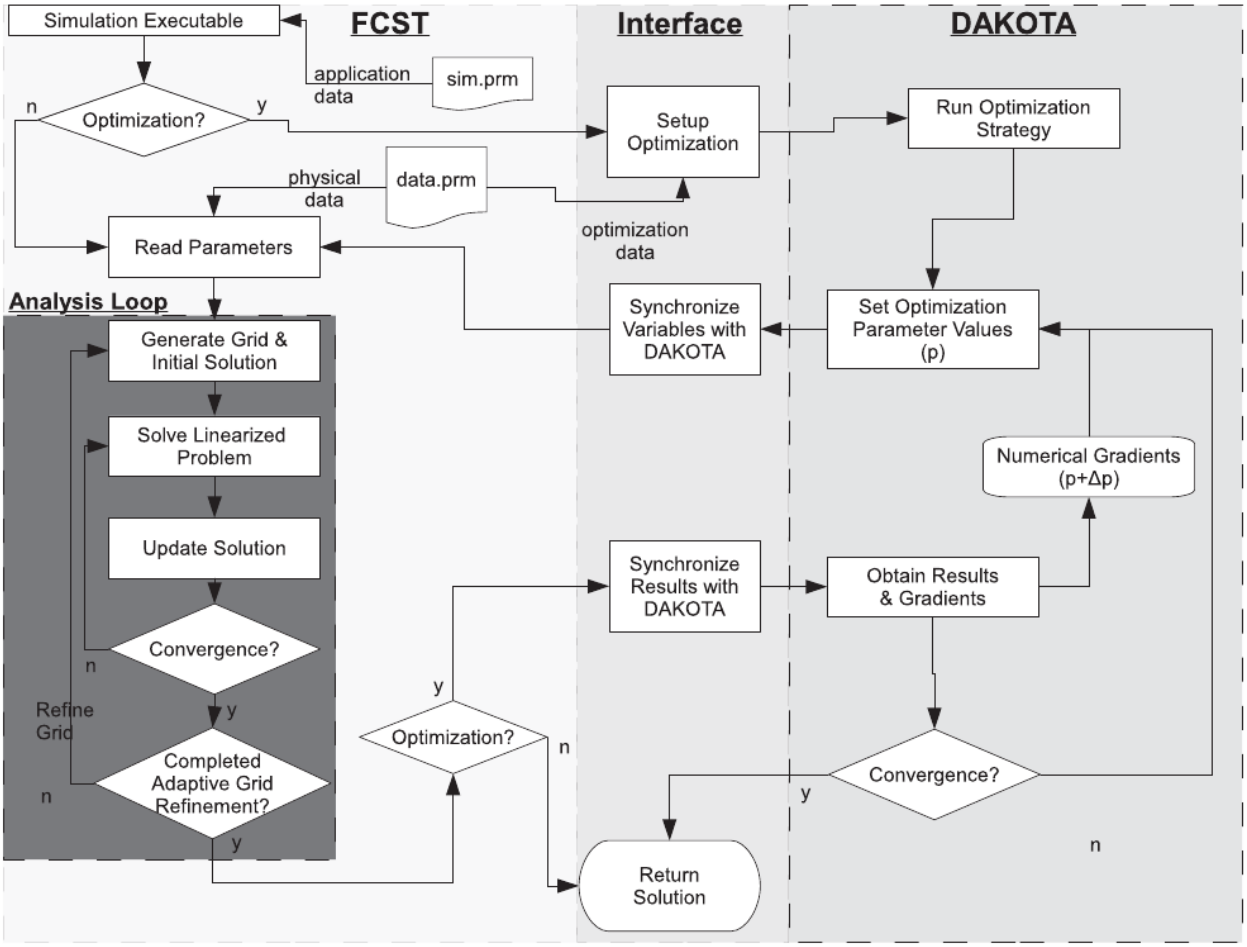

Figure 4.10: Schematic of Fuel Cell Analysis Code and DAKOTA Optimization Interface (Dobson, 2012)

Chapter 5

Post-processor

FCST can output result in many different formats using the deal.II output parser. FCST developers however output the solution in .vtk format and use the open-source post-processing software Paraview to analyze their results. ParaView is an open-source data analysis and visualization program. ParaView can run in multiple operating systems. Using 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 6

Preliminaries

6.1 Setting up FCST under KDevelop

If you are going to be developing new routines for FCST, we recommend that you use KDevelop. In order to setup a KDevelop project with FCST follow the steps below:

- In the main menu, go to Setting > Configure KDevelop... Select the Background Parser tab. Disable the Background Parser. Note this is necessary because FCST is a large code and, unfortunately, parsing takes a very long time.
- Go to Projects > Open/Import Project... Select the FCST folder. In the next window, select Generic Makefile. At this point, the project will appear on the left hand side and you can browse through all files.
- Next, we will setup the environment to run and debug the code within KDevelop.
- Go to Run > Configure Launches...
- Press the '+' buttom on the top of the window. Once you press this botton, a new option will open under either Global or FCST. Select New Native Application Configuration
- On the right of the window, under Executables, enter the FCST binary file. For example,
- 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 6.1.

6.1.1 Formatting OpenFCST files

All files should start with the following information:

```
1  // ------------------------------------------------------------------------
2  //
3  // FCST: Fuel Cell Simulation Toolbox
4  //
5  // Copyright (C) 20XX-20XX by Energy Systems Design Laboratory, University of Alberta
6  //
7  // This software is distributed under the MIT License
8  // For more information, see the README file in /doc/LICENSE
9  //
10 // - Class: class_name
11 // - Description: short description of class
12 // - Developers: name_developers, affiliation
13 // - Id: $Id$
```

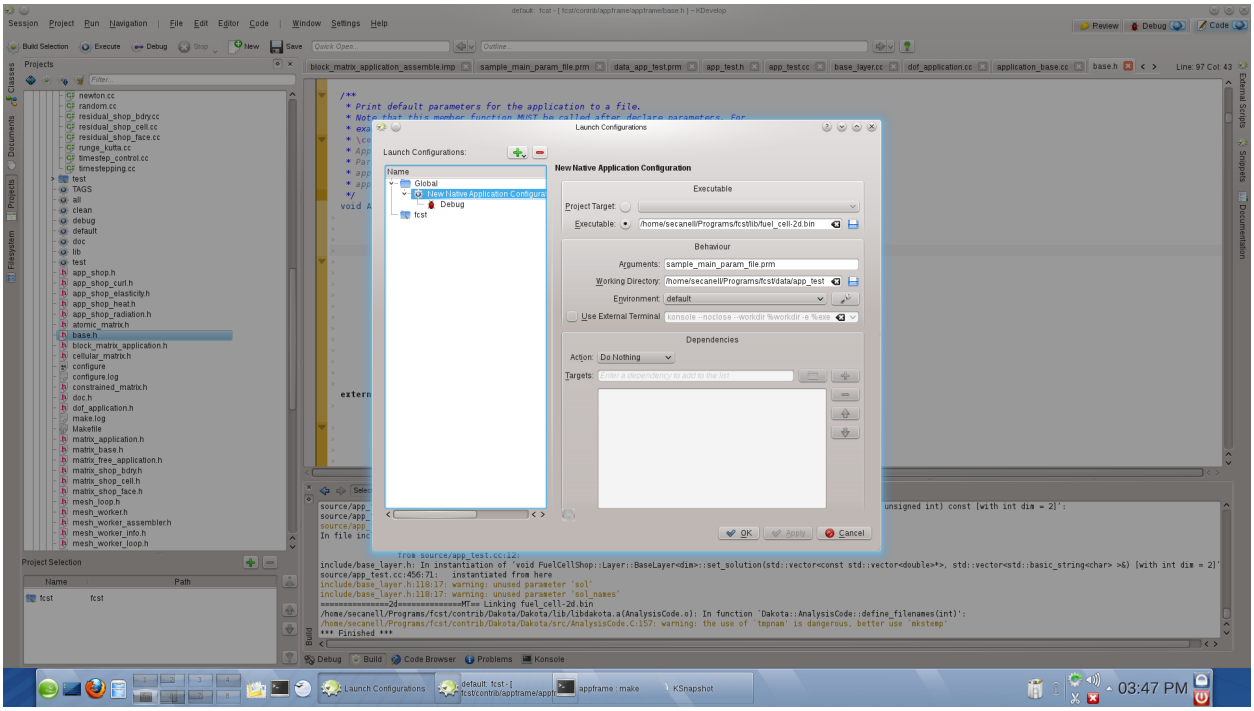

Figure 6.1: Configuring Configure Launches... in KDevelop

```
14 //
15 // ------------------------------------------------------------------------
```

In order to keep the formatting of all files consistent, it is recommended to use the space style for code readability. In KDevelop, set your formatting options:

- In the main menu, go to Setting > Configure Editor.
- In the 'Editing' section, select the Indentation tab.
- Set 'Indent Using' to *Spaces*, and set the spacing to **4 characters**.

Chapter 7

FCST structure

7.1 Directory tree

FCST contains six subfolders, namely:

- FuelCell/lib Contains the binary executable fuel cell simulator once FCST has been compiled. To compile FCST enter the FuelCell folder, change the Makefile to the correct path for Deal.II and Deal.II libraries and typing $make.
- FuelCell/include Contains all include files (*.h)
- FuelCell/source Contains all source files (*.cc)
- FuelCell/doc Contains all documentation. This includes the Users Manual in html (the main page is index.html). To generate the manual, enter the folder FuelCell and type

```
1  $make doc.
```

  To do so, Fuel Cell uses the free package Doxygen. In the makefile the path to Doxygen is specified by the variable $DOXYGEN\_PATH$. The default location is /opt/local/bin/doxygen.

- FuelCell/data Contains data used to run several of the applications in FCST.
- FuelCell/test - Used for CDash (see section 12)
- FuelCell/contrib - Contains OpenSource libraries and codes that have been developed by other people and are used within FCST. This include AppFrame (developed by G. Kanschat) and DAKOTA (developed by Sandia National Labs). Note that the codes have been modified.

The main program file is main.cc in FuelCell/source. This file creates:

- A NewtonExecution object. This object is used to implement the mesh adaptive loop
- NewtonExecution object. This object is used to implement the mesh adaptive loop
- A linear application that implements the fuel cell equations (linearized version of them). There are several linear applications. The linear solver implements:
  - *cell_residual*(): This member function is called by *dof_application.cc* in AppFrame in order to implement the residual. $cell_residual()$ is in charge to compute the residual for a given cell.
  - *cell_matrix*(): This member function is called by *block_matrix_application.cc* (via optimization_block_matrix_application.cc) and is used to assemble the cell matrix. *block_matrix_application.cc* uses this information to assemble the stiffness matrix for the complete problem.
  - solve(): This member function is used to solve the linear system.

7.2 Understanding FCST Architecture

EXPLAIN WHAT IS AN APPLICATION, PHYSICS, ETC.

7.3 Understanding FCST Applications: The FCST tutorials

FCST contains several tutorials to get you started developing your own applications. Currently two tutorial applications have been developed

- Cathode application
- Cathode and membrane application

You can find these two tutorials in the Modules section of the DOxygen documentation, i.e. in file `/doc/html/modules.html`. All FCST developers learnt to develop applications by first reading these tutorials. If you are developing new physics classes, then you will be relying heavily on classes from the deal.II finite element libraries. If this is the case, the FCST developers would recommend any new developers to look at the tutorials provided in the deal.II website.

!!!!!!!!!!!!!!! IS THERE ANYWAY WE CAN INSERT THE TUTORIALS FROM DOXYGEN HERE !!!!!!!!!!!!!!!

7.4 FCST Applications

7.4.1 Data files

It is recommended that every application also contains a data file in `/fcst/trunk/data` with a folder name corresponding to the name of the application. The data folder should contain four sub-folders as follows

- analysis: This folder contains default (and well documented) main, data and mesh input files to run a sample analysis problem.
- parametric: This folder contains default (and well documented) main, data, mesh and optimization files to run a parametric study using FCST and Dakota
- optimization: This folder contains default (and well documented) main, data, mesh and optimization files to run a parametric study using FCST and Dakota

7.5 Namespace structure

FCST contains four namespaces, namely

- *FuelCell*
- *FuelCellShop*
- *AppFrame*
- *AppFrameShop*

Namespaces *AppFrame* and *AppFrameShop* designate member function in the contributing library *AppFrame* which was originally developed by Dr. Guido Kanschat at the University of Heidelberg and is currently maintained by Dr. Guido Kanschat and the authors of Fuel Cell Simulation Toolbox (FCST).

Namespaces *FuelCell* and *FuelCellShop* form the core of FCST. Namespace *FuelCell* contains an Application and InitialSolution namespace as well as several classes such as OperatingConditions. Application namespace contains classes that can be used to solve a specific problem such as fluid flow in a channel, or

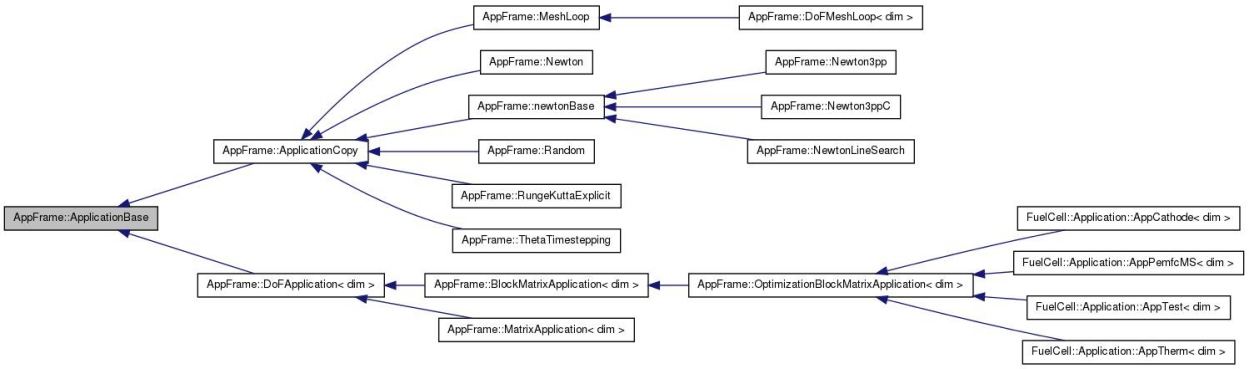

Figure 7.1: Inheritance tree for ApplicationBase

the phyiscal processes occuring in the cathode of a fuel cell. Based on the nature of the application, two types of classes are available:

- *AppFrame::DoFApplication*
- *AppFrame::ApplicationCopy*

Figure 7.1 shows an overview of the two types of applications.

Classes inherited from *AppFrame::DoFApplication* are used when we need to implement the governing equations of the physical problem, i.e. the weak from of the partial differential equations. Class *AppFrame::DoFApplication* implements all the methods used to assemble the right hand side, i.e. it contains a Triangularization (domain mesh), a deal.ii DoFHandler and several other objects to loop over cells. Class *AppFrame:: BlockMatrixApplication* is a child of *AppFrame::DoFApplication* and contains additional functionality in order to assemble and store the global system matrix into a BlockMatrix object. Finally, class *AppFrame:: OptimizationBlockMatrixApplication* implements optimization functionality such as functional evaluation. FCST applications that require assemble of a system matrix and right hand side are inherited from this application such as *FuelCell::AppCathode*.

Figure 7.2 provides an example of the inheritance tree for *FuelCell::AppCathode*. AppCathode inherits all the functionality of of *AppFrame::DoFApplication*, *AppFrame::BlockMatrixApplication* and *AppFrame:: OptimizationBlockMatrixApplication*. The responsibility of the FCST applications in namespace *FuelCell* is to initialize all the variables and to implement three main routines

- cell_matrix()
- cell_residual()
- solve()

The first two routines are used to implement the element-wise system matrix and the element-wise right hand side. The latter member function is use to solve the global finite element problem. A tutorial on how to develop an application can be found in the HTML documentation.

Classes inherited from *AppFrame::ApplicationCopy* are used to implement iterative loops. For example, when solving a nonlinear problem, a linear problem is solved iteratively. Therefore, classes that inherit from *AppFrame::ApplicationCopy* usually contain an application that inherits from *AppFrame::DoFApplication*. In terms of FCST, FCST developer will usually implement *AppFrame::DoFApplication* and use the already implemented classes of type *AppFrame::ApplicationCopy* in order to develop iterative loops for adaptive refinement, nonlinear problems and time-dependent problems.

As an example, in simulation_builder.cc the following process is employed to solve a nonlinear problem:

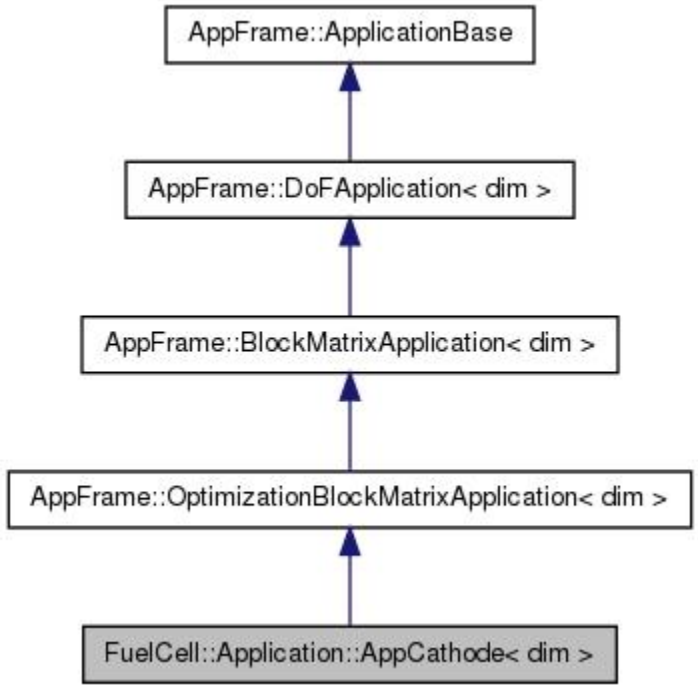

Figure 7.2: Inheritance tree for FuelCell::Application::AppCathode

```
1   //-- Select the application you want to run:
2   app_lin = sim_selector->select_application();
3   //-- Select the solver you want to run:
4   newton = sim_selector->select_solver(app_lin.get());
5   //-- Select the solving method you want to run, e.g. adaptive refinement:
6   solver = sim_selector->select_solver_method(app_lin.get(), newton.get());
7   // Here we have collected all information:
8   deallog << "Run program using input file: "
9     << simulator_parameter_file_name << std::endl;
10  deallog.pop();
11  solver->solve(simulator_parameter_file_name, param);
```

In the code above, first an FCST application that inherits from *AppFrame::DoFApplication* is created. Then, the application that is used to solve the linear system of governing equations at each iteration is handed in to a Newton solver that inherits from *AppFrame::ApplicationCopy*. This solver is in turn handed to another solver that inherits from *AppFrame::ApplicationCopy* and that implements an adaptive refinement loop.

Namespace *FuelCell* is therefore a place holder for applications of type *AppFrame::DoFApplication*. This applications are the core of FCST since the assemble the system of equations that need to be solved. Users can develop their own applications by inheriting *AppFrame::OptimizationBlockMatrixApplication* and re-implementing declare_parameters(), initialize(), cell_residual(), cell_matrix() and solve() as shown in Figure 7.2 and explained in detail in the tutorial program for AppCathode. If the problem requires solving the problem iteratively such as in nonlinear and transient problems, the application will be handed over to an application of type *AppFrame::ApplicationCopy* to be solved iteratively.

The namespace *FuelCellShop* is divided into several other namespaces as follows

1. Material namespace: Specify the properties of common materials used in fuel cells. Examples of material classes include *PureGas* baseclass.

2. Layer namespace: Specify the different MEA components in a fuel cell. A BaseLayer has been developed

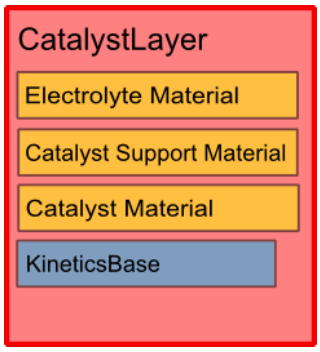

Figure 7.3: Layer namespace structure

in order to standarize this classes. They contain $material_id$ and $bondary_id$ elements to be able to relate the layer to the mesh as well as many other properties. Layer classes contain material classes in them that are used in order to obtain the appropriate physical parameters of the layer. Some layers are homogeneous and some are heterogeneous and anisotropic.

3. Matrix namespace (OLD): Used to assemble the governing equations of a system. For linear problems this class implements the stiffness matrix for a givem equation. For nonlinear problems it generally includes $\frac{\partial R}{\partial \vec{u}}$. This matrix is used to calculate the step size in a Newton solver.

4. Residual namespace (OLD): Used to assemble the governing equations of the system. For linear system these classes contain the right hand side of the problem. For nonlinear systems they contain the residual at the previous iteration, i.e. $R(\vec{u})$.

7.6 Layers Namespace

A Layer in OpenFCST is used to define the properties of a cell in a finite element mesh. A Layer can be formed with with a single material or, in the case of composite layers such as a gas diffusion layer or a catalyst layer, it contains several materials and reaction parameters which are then used to compute the effective properties. An example of a catalyst layer is shown in Figure ??.

The FuelCellShop::Layers namespace contains all the layers available with OpenFCST. All layers inherit from BaseLayer as shown in Figure ??. BaseLayer is a virutal class. An object of this class should never be created. BaseLayer simply provides common member functions and data members that apply to all layers.

The layer classes contain a member function named set_solution() which parses the uni

7.7 Materials Namespace

No materials contain a set solution. Materials will only set specific variables using a routine such as set_temperature(double).

7.8 Contributing libraries

FCST is distributed with copies of deal.II, Appframe, Dakota, COLDAE, ALGLIB and cpptest. These projects reside in the subdirectory `contrib/`. Please note that these projects are copyrighted by others than the FCST authors and are covered by different licenses. For details, consult their respective webpages. A copy of their respective license is provided with the code. Inside each contributing library folder you will also find a file called `README.txt`. This file contains a list of any modifications that the FCST developers have

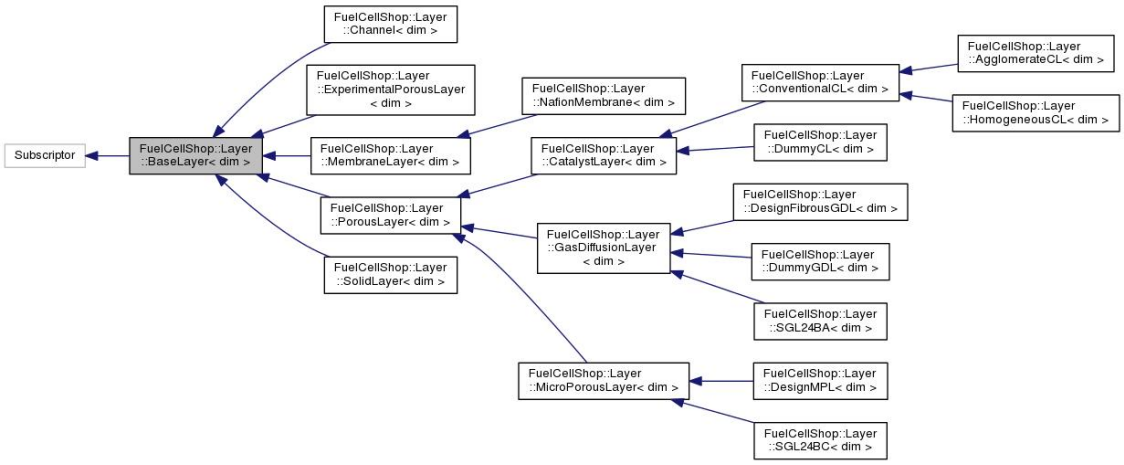

Figure 7.4: Layer namespace structure

made to the contributing libraries. Note that some FCST authors also contribute to the additional libraries. For example, Valentin Zingan is also contributor of the deal.II libraries.

7.8.1 APPFRAME

AppFrame was originally developed by Dr. Guido Kanschat at Texas A & M and it is not continued to be developed by the ESDLab group. AppFrame is a framework to develop a chain of applications. The idea of these application classes is the possibility to build a chain out of these in order to have several predefined nested solvers. For this, we distinguish applications roughly in two classes, both derived from ApplicationBase:

- Terminal applications, which implement the real finite element code like computing residuals on mesh cells, assembling matrices and solving linear systems.
- Non-terminal applications derived from ApplicationCopy; these usually implement a new solve() function as an iterative solver around another application. They implement all functions of the ApplicationBase interface, either forwarding them to the next inner application by ApplicationCopy or by providing their own implementation.

Usually a class in a chain communicates values with its next outer class through function arguments. Nevertheless, at least the terminal application will require values from even outer applications in the chain in order to compute residuals and matrices correctly. For these, the mechanism of named data provided by ApplicationData was introduced. Each class can store auxiliary data under a unique name there for use in inner iterations.

Class DoFApplication

This class it the parent of all terminal applications. It is the base class for applications requiring a Triangulation and a handler for degrees of freedom.

The mesh as well as the dof handler may be created by this class, which is the default, or they may be provided by another object, in which case they must be specified in the constructor.

Note that in this class the received or created dof handler is associated to the finite element given by the argument "Element" on the parameter file. Therefore, this class in not responsible to generate the system of equations to be solved, only to initialize the dof handler "Element" can either be a single element

of a FESystem. In the latter case, the nomenclature used in the paramter file is: set Element = FESystem[element1_type(element1_degree)^number_of_elements1-...-elementN_type(elementN_degree)^number_of_elementsN] Example: set Element = FESystem[FE_DGQ(0)-FE_Q(1)^2]

Class BlockMatrixApplication

Text needed here

7.8.2 `COLDAE` Interface

The class `DAESolver`, declared in `DAE_solver.h` and defined in `DAE_solver.cc`, provides an interface to the `Fortran 77` boundary-value differential algebraic equations (DAEs) code `COLDAE`. The code solves DAEs the consists of a system of mixed-order ODEs

$$u_i^{(m_i)} = f_i(x; z(u(x)), y(x)), \quad i = 1, ..., c,$$

and algebraic constraints

$$0 = f_i(x; z(u(x)), y(x)), \quad i = c+1, ..., c+d,$$

for $a < x < b$. The DAE is subject to a system of mixed-point boundary conditions

$$g_j(\zeta_j; z(u(\zeta_j))) = 0, \quad j = 1, ..., m^*,$$

where

$$a \leq \zeta_1 \leq \zeta_2 \leq \cdots \leq \zeta_{m^*} \leq b,$$

and

$$m^* = \sum_{i=1}^{c} m_i.$$

The code `COLDAE` returns an exact solution

$$z(u(x)) = (u_1^{(1)}(x), u_1^{(m_1-1)}(x), \ldots, u_c^{(m_c-1)}(x),$$

where $u_i^{(m_i)}$ is the ith derivative of u_i.

The `COLDAE` interface is demonstrated by solving the DAEs

$$z''(x) = y(x) + \sin\left(\frac{1}{1+x}\right) + \frac{2}{(1+x)^3}, \quad 0 < x < 1,$$
$$0 = y(x) + \sin(z(x)),$$

subject to the boundary conditions

$$z(0) = 1,$$
$$z(1) = \frac{1}{2}.$$

The DAE along with the boundary conditions are defined by a series of functions.

Beginning with the DAE:

```
1 void fsub(double &x, double z[], double y[], double f[])
2 {
3   f[0] = y[0] + sin(1.0/(1.0+x)) + 2.0/pow(1.0+x,3);
4   f[1] = y[0] + sin(z[0]);
5 }
```

In the above function, the functions `pow` and `sin` are declared in `math.h`. The Jacobian matrix of the DAE must also be defined:

```
void dfsub(double &x, double z[], double y[], double df[])
{
  // Declare an array of pointers for the two dimensional array
  // that will hold the Jacobian matrix .
      double** dfc;
  dfc = new double*[2];
  for (int i=0; i < 2; ++i)
  {
    dfc[i] = new double[3];

  }

  // Define the Jacobian matrix.
  dfc[0][0] = 0.0;
  dfc[0][1] = 0.0;
  dfc[0][2] = 1.0;
  dfc[1][0] = cos(z[0]);
  dfc[1][1] = 0.0;
  dfc[1][2] = 1.0;

  // Convert the matrix dfc to a one dimensional
  // array that Fortran will understand.
      AppFrame::c_to_for_matrix(2,3,dfc,df);
}
```

In the function `dfsub`, the function `AppFrame::c_to_for_matrix` converts the `C/C++` two-dimensional array `dfc` to a one-dimensional array `df`. The array `df` is then sent to `Fortran` and read as a two-dimensional array. Similar to the other math functions, `cos` is declared in `math.h`.

The boundary conditions can be defined as:

```
void gsub (int &i, double z[], double &g)
{
  if (i == 1) g = z[0] - 1.0;
  else if (i == 2) g = z[0] - 1.0/2.0;
}
```

In the above function, `i` refers to the location, between $0 \leq x \leq 1$, of the ith boundary condition. In this case, `i==1` refers to $x = 0$ and `i==2` refers to $x = 1$. Similar to `fsub`, the partial derivatives must be defined for `gsub` in a separate function.

```
void dgsub (int &i, double z[], double dg[])
{
  if (i==1 || i == 2 )
  {
    dg[0] = 1.0;
    dg[1] = 0.0;
  }

}
```

Once the functions are defined, a few variables must be defined that contains additional information about the boundary-value DAE.

```
  //Declare an array that holds the
  // order of each of the ODEs.
  int *mm = new int[1];
 mm[0] = 2;

  //Declare the location of each boundary point.
  double *zeta = new double[2];
  zeta[0] = 0.0;
  zeta[1] = 1.0;
```

An instance of `DAESolver` can now be created by calling the constructor.

```
//Create an instance of the DAE solver class.
AppFrame::DAESolver *prob = new AppFrame::DAESolver
(1,  //Number of ODES
1, // Number of algebraic constraints
mm, // Array that holds the order of each of the ODEs
0.0, // Leftmost boundary point
1.0, // Rightmost boundary point
zeta, // Location of boundary points
fsub, // ptr to ODE function
dfsub, //ptr to Jacobian of ODE function
gsub, //ptr to boundary-condition function
dgsub, //ptr to derivatives of boundary-condition function
);
```

Optional `COLDAE` parameters can be set by calling corresponding member function of the `DAESolver` class. For example, the number of points in the initial mesh can be set.

```
prob->set_initial_mesh_size(20);
```

These methods must be called before the boundary-value DAE is solved. See the `Doxygen` documentation for `DAESolver` for a complete list of methods.

Once all desired `COLDAE` parameters are set, the boundary-value DAE can be solved.

```
int flag = prob->DAE_solve();
```

The variable `flag` contains an integer value that reports the success of `COLDAE`; see `Doxygen` documentation. If `COLDAE` is successful, a continuous solution can be accessed by the use of the `DAE_solution` method. For example, suppose a solution is required for $x = 0.5$:

```
double x = 0.5;
double z[2];
double y[1];
prob->DAE_solution(x,z,y);
```

In the above code, `z` contains the solution for the ODEs and `y` contains the solution for the algebraic constraints.

7.8.3 Adding a new version of a contribution library to the repository

In order to be in the save side, follow these steps: Step 1 - Rename current contrib_folder folder to contrib_folder.old. For example, if you are trying to update deal.II, type the following commands on the terminal:

```
$svn mv deal.II deal.II.old
$svn commit -m"Deal directory for 7.0 moved to deal.II.old"
```

Step 2 - Download new version of contrib_folder from net

Step 3 - Make sure the new code and FCST compile and work fine together

Step 4 - Update subversion to now hold the "new" version of contrib_folder

```
$svn mv deal.II deal.II.old
$svn commit -m"Deal directory for 7.0 moved to deal.II.old"
```

Step 5 - Delete contrib_folder.old directory tree

Chapter 8

Coding Guidelines (DRAFT)

The purpose of this chapter is to specify coding guidelines for developers of the FCST 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.

8.1 Class and Member Naming Conventions

Naming conventions are defined it 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 the meaning and/or purpose of the object they represent.

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.

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

Function naming definition:

Function names should be written in camel-case with their first letter in lower case. 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.

Should we use this name convection or the following get_potatoes( ) which is what we have been using thus far. If we use getPotatoes we need to recode every single class... Examples:

```
1 generateInverse(double numToInvers): //Good
2
3 compute_I(double a);     //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.

```
1 ... anodeKinetics //Good
2
3 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.

```
SPEED_OF_LIGHT   //Good
```

FCST contains a file with many constants already available named `fcst_constants.h`.

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.

8.2 Class and Member Document Strings

Document strings (doc strings) are comments which accompany class, function and variable definitions in the header files. They provide information to aid developers wishing to understand and utilize other's code. Documentation packages such as Doxygen can parse doc strings to produced styled, easily readable documentation with minimal developer effort. The following are doc string templates that should be implemented by FCST developers when creating new classes, functions or variables.

Class Doc String:

```
/**
   *Authors    : List of name authors
   *
   *Description : A brief description of the classes purpose
   *
   *Use cases    : A list of intended uses of the class\
   *
   *Notes    : Other important information
   *
*/
```

Function Doc String:

```
/**
   *Description : A brief description of the purpose
   *
   *Use cases    : A list of intended uses
   *
   *Access rules: Public/Private/Protect
   *
   *Inputs   : Variable descriptions and Types
   *
   *Outputs    : Description of output
   *
   *Notes    : Other important information
   *
*/
```

Variable Doc String:

```
/**
   *Description : A brief description of the purpose, , units (if applicable)
   *
   *Use cases    : A list of intended uses
   *
   *Access rules: Public/Private/Protect
   *
   *Notes    : Other important information
   *
*/
```

Please note that the correct input syntax for Doxygen:

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

8.3 Assertations and exception handling

OpenFCST includes many assertations 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 expect is received by the class. OpenFCST uses two types of assertations

- Assert: Checks that the desired information is provided. This assertation will only work in debug mode. This means that when running on 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 assertations check that the parameters in the input file are correct. Such assertations 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 9

Developing Documentation in FCST

FCST documentation consists of two documents, the User and Developer's Reference Guide that you are currently reading (located in `/fcst/doc/RefGuide/`) and the HTML class documentation (located in `/fcst/doc/html/`). In order to have access to the latter, the documentation needs to be compiled using DOxygen by issuing the command `make doc` in a LINUX terminal. In the following section, the guidelines for developing the two documents are discussed.

9.1 Developing the User and Developer's Reference Guide

9.2 Developing DOxygen documentation

THIS SECTION STILL NEEDS TO BE DEVELOPED

9.2.1 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 by Peter Dobson 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
```

9.2.2 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 function and(...)
3  */
```

Chapter 10

Development Process

This section outlines first the approach that developers should take when approaching a new project, i.e. modify the code directly or create a new branch of the code using Subversion that we can then merge into the main trunk. Next, the section outlines a development method know 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 explain in the following sections.

10.1 Proposed Development Cycle

In order to reduce the number of undesired bugs in FCST, the following development cycle should be used when modifying FCST. If you decide to modify FCST you have three options depending on the purpose of the changes:

- Modifications to `trunk` directory. If only minor changes are made to the FCST interface, i.e. you are developing new classes or applications that are not core components of FCST, changes can be performed directly in the `trunk` of FCST. Before committing any changes make sure the code compiles AND that the code passes all FCST tests by running the test script `run_tests`.
- Creation of `branches`. If you expect to modify any core classes of FCST such as a base class or AppFrame, it is recommended that the developer creates a new branch for the code, makes all desired modifications, makes sure the code compiles in several operating systems such as OpenSUSE and Fedora, makes sure the code passes all FCST tests by running the test script `run_tests` and then, merge the branch with the trunk.
- `stable_release` creation. Once a year, a new release of FCST will be developed. A stable_release is the version of FCST that will be available on the OpenFCST website.

10.2 Test Driven Development

Test Driven Development (TDD) is a software development methodology which is rather different compared to the typical development process generally acquire 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 test. 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

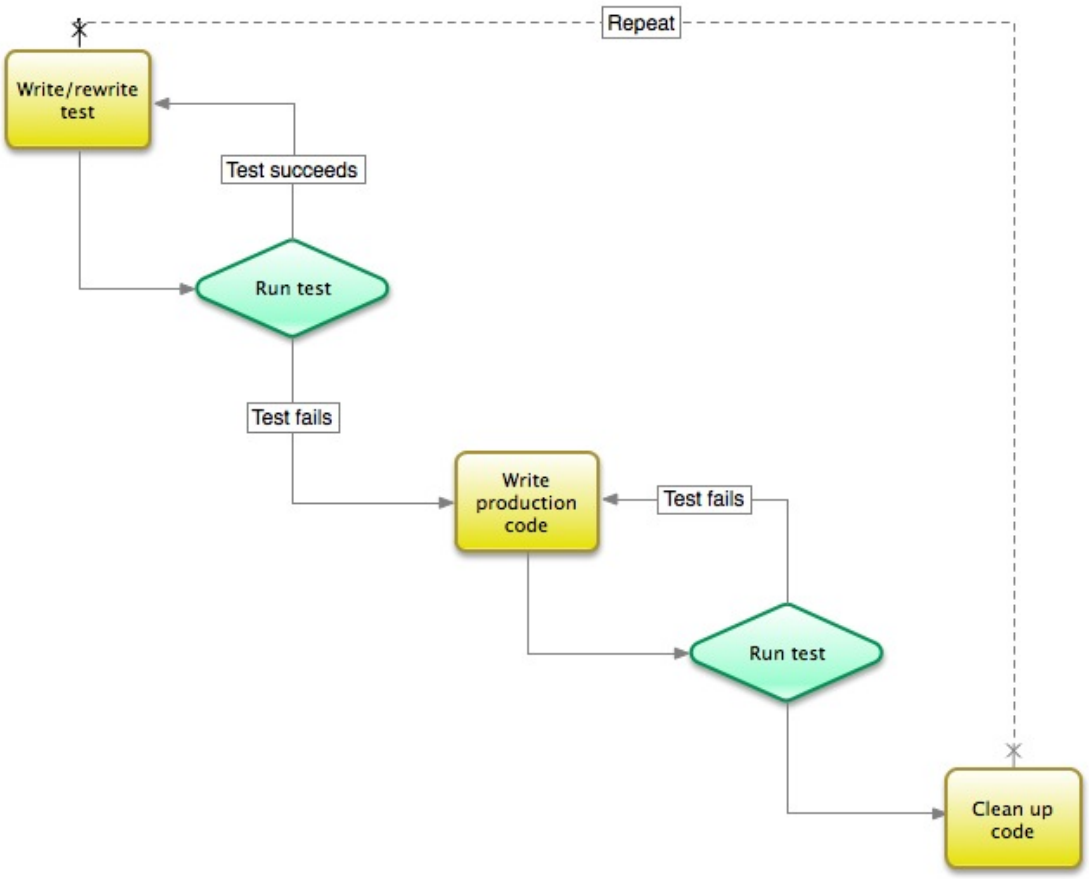

Figure 10.1: TDD Cycle

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 ?? :

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 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 judgment.

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.

4. Units 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

10.2.1 Unit Tests

Unit tests as already mentioned are tests that determine if individual units of source code are 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 is not.
7
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 recieve 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"
```

10.2.2 TDD Implementation in the FCST

The unit testing structure that is implemented in the FCST 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 10.2. Firstly unit tests are run (which will tests individual components of various classes), then system level tests (similar to the tests already implemented by Michael Moore).

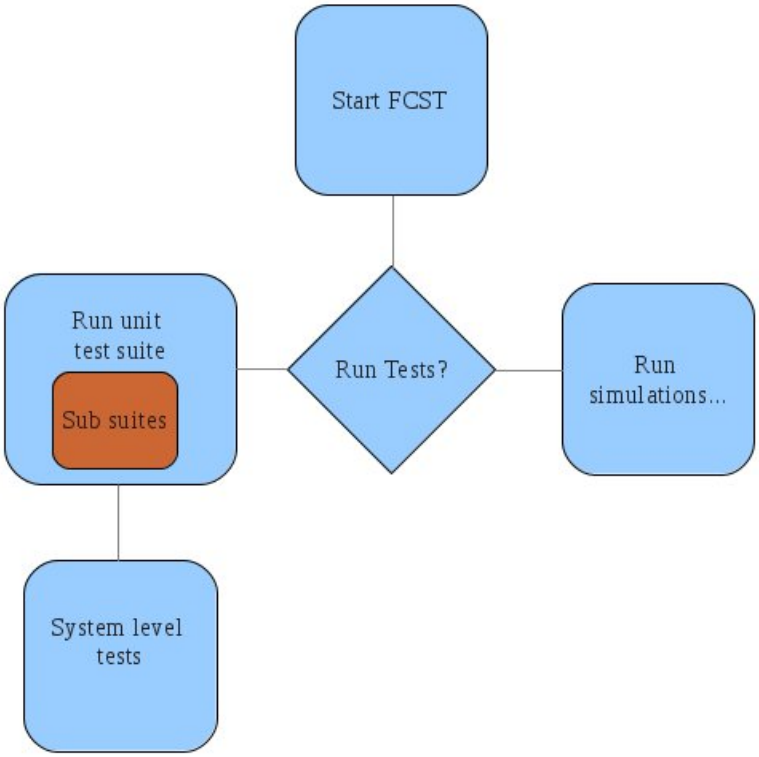
Start FCST
Run unit
test suite
Sub suites
Run Tests?
Run
simulations...
System level
tests

Figure 10.2

The operation is as follows:
The “Run tests” parameter is set in the main parameter file.

```
set Run tests = true
```

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

```
    void SimulatorBuilder<dim>::run_test()
    {
  deallog << "==================Running Unit Tests==================" << std::endl;

  FcstTestSuite::run_tests();

  deallog << "================System Tests Complete=================" << std::endl;
```

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

```
bool FcstTestSuite::run_tests()
{
  Test::Suite ts;

  //add sub tests suites
  ts.add(std::auto_ptr<Test::Suite>(new FcstsUnitsTestSuite));
  ts.add(std::auto_ptr<Test::Suite>(new IonomerAgglomerate3Test));

  Test::TextOutput output(Test::TextOutput::Verbose);
  return ts.run(output);

}
```

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.

```
#ifndef _FCST_UNITS_TESTSUITE
#define _FCST_UNITS_TESTSUITE

#include <cpptest.h>
#include "fcst_units.h"



class FcstsUnitsTestSuite: public Test::Suite
{
  public:
      FcstsUnitsTestSuite()
      {
  //Add a number of tests that will be called during Test::Suite.run()
  //Generic cases
  TEST_ADD(FcstsUnitsTestSuite::perBigToSmallTest);
  TEST_ADD(FcstsUnitsTestSuite::bigToSmallTest);
  TEST_ADD(FcstsUnitsTestSuite::perSmallToBig);
  TEST_ADD(FcstsUnitsTestSuite::smallToBig);

    //specific Cases
  TEST_ADD(FcstsUnitsTestSuite::btuToKwh);
  TEST_ADD(FcstsUnitsTestSuite::kwhToBtu);
      }
  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  }
```

10.2.3 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 ab 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;")

10.2.4 Refactoring

Refactoring is a technique for restructuring existing code so to improve it's readability and user understanding, whithout changing the behaviour of the code in anyway. 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 make 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 primatives 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):

```
void sendMessage(Message dataToSend, string phoneNumber, string networkOperator){

  string areaCode = "213";

  if (networkOperator == "Rogers"){
    string _phoneNumber = areaCode + "4" + phoneNumber;
    MessageBuffer b;

    for(int i=0; i < dataToSend.length(), i++)
      b.pack(dataToSend[i]);

    send(b, _phoneNumber)

  }
  else if(networkOperator == "Telus"){

    string _phoneNumber = areaCode + "9" + phoneNumber;
    MessageBuffer b;

    for(int i=0; i < dataToSend.length(), i++)
      b.pack(dataToSend[i]);

    send(b, _phoneNumber)

  }
}
```

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.

```
void sendMessage(Message dataToSend, string phoneNumber, string networkOperator){

  string fullPhoneNumer = PhoneBook::getAreaCode() + PhoneBook::getOperatorCode(
      networkOperator) + phoneNumber;
  MessageBuffer buffer = package(dataToSend);
  send(buffer, _phoneNumer)''

}
```

```
 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 not the change of variable names. The new names do a better job at describing their purpose.

10.2.5 Unit Standards

This section will specify standards on what physical units are typically used in the FCST.

Chapter 11

Tracking and Ticketing System for FCST

The ESDLab group maintains a Tracking and Ticketing System for FCST at the following site. In addition, TODO items can be added to the HTML documentation. See the HTML documentation for a complete list.

11.1 Tracking and Ticketing System Overview

11.2 Using the Ticketing System

Chapter 12

Daily FCST testing suite: CTest and CDash

12.1 Testing your code in your local directory

Before committing any changes to the repository, you should make sure that your working copy does not have any errors. You can test your working copy of FCST by typing the following in the command line in the FCST main folder:

```
1 $./run_tests
```

All test cases that have been programmed will be executed and the results will appear in two files in the main FCST folder. The first file created is the tests_summary.log, which will give a quick summary of the results of the tests, this will be opened automatically by the default text editor. The second file created is the full output from the simulation, this will contained in tests_output.log. The summary from each test will be appended to the end of the output from each test.

At present there are two tests implemented, the app_cathode and app_pemfc test. The tests comprise of running the cases contained in the testing folder for each case (i.e. ./data/cathode/testing and ./data/pemfc/testing). Each test case has a data file containing expected results that will be compared to the results from the test. If they are not in agreement, an error message will be printed. To add another test to be run by ctest, first add the case to the list of tests. This list is contained in the ./test/CTestTestfile.cmake file. The command ADD_TEST is used:

```
1 ADD_TEST(app_cathode "tests/app_cathode_case.sh")
```

The first item is simply a name and can be anything that will describe the test being run. The second item is the location of the script that contains the test that is to be run. These scripts should always be put in the ./tests/ folder. The tests themselves are simply bash scripts that contain at least the call to the executable with the correct input data file. Additional capability is added that includes the printing of additional messages and comparison of expected and simulation results. To create a new bash script containing a test, please view the app_cathode.sh test case as an example (the case is given in the appendix):

The main features of a testing script should include:

- Runs the FCST executable using the correct data file.
- Compares the results to expected results.
- Determines whether the run was successful.
- Prints the results to the tests_summary.log file which will be copied to the main FCST folder.

Note that if you develop a new test case and feel it should be added to the nightly tests, then please speak to a CDASH administrator before committing the test script to the repository.

Chapter 13

Useful Programming Tips

13.1 Memory Leak Detection

Despite taking all precautions to avoid memory leaks, it is still a possibility. It is recommended that developers make use of the program Valgrind. It is a freely available, open-source software, and can be found in the software packaged with most Linux distributions. Valgrind checks memory operations of any program, without having to modify the program in any way. It will typically make the program run 20-50 times slower than normal. To run, simply call `valgrind --options` before your normal program call. E.g.

```
1 valgrind --leak-check=full --show-below-main=no --show-possibly-lost=no
2 --show-reachable=no ./fuel_cell-2d.bin simulation.prm
```

will run FCST with a full memory leak check, while ignoring losses outside of the main program (system memory) and ignoring possible memory leak areas. This tool is particularly useful at pinpointing segmentation faults, etc.

See the Valgrind documentation for usage.

13.2 Working with pointers

As the complexity of FCST grows, it is easy to lose track of the dependencies between the applications and the classes that define the physics of the problem. FCST makes frequent use of pointers to pass information from one area of the code to another. Without careful tracking of pointers, it is possible to program memory leaks and memory faults into the program. It is recommended that two tools be used. First, the Boost Libraries implementation of smart pointers. Using these pointers ensures that memory will not be free'd if it is still in use. Furthermore, there is no need to explicitly `delete` data contained within a pointer, reducing the risk of a memory leak. The task to convert regular pointers to smart pointers is underway. For implementation, see the Boost documentation

13.3 Including files to the include files

In order to make sure that the include files in FCST are looked at first, please include files as follows: #include"example.h" instead of #include¡example.h¿ for all FCST files.

13.4 Subversion tips

13.4.1 Setting the Id Tag in Subversion

Subversion can be configured to set the tag when you check in a file.

This configuration has to be done client side. All developers must go through this procedure. Configure subversion to automatically add the $Id: $ tag do the following [6]

- Update your /.subversion/config by including the lines below. This configuration will automatically apply the svn:keywords property (which sets the *Id* tag) to all new files.

```
[miscellany]
enable-auto-props = yes

[auto-props]
*.h = svn:keywords=Id
*.cc = svn:keywords=Id
Makefile.in = svn:keywords=Id
```

- Set props on existing files. For existing files, you have to set the property manually. Go to the root of your source tree, and run this command:

- **NOTE:** For "*.h", "*.cc" and "Makefile.in" files, the property has already been set in the repository. So, it isn't required to do "propset" on these files. The following example code is for future use, for eg. setting property for new code files like Python script etc.

```
find . \( -name "*.h" -o -name "*.cc" \) -exec svn propset svn:keywords Id {} \;
```

- Go through each file and add the prototype *Id* tag. Do this at the comments header of each file as shown below

```
//-----------------------------------------------------------------------
//
//    FCST: Fuel Cell Simulation Toolbox
//
//    Copyright (C) 2013 by Energy Systems Design Laboratory, University of Alberta
//
//    This software is distributed under the MIT License.
//    For more information, see the README file in /doc/LICENSE
//
//    - Class: electron_transport_equation.h
//    - Description:
//    - Developers: M. Secanell
//    - Id: $Id: $
//
//-----------------------------------------------------------------------
```

13.5 Troubleshooting

13.5.1 Including new virtual functions in already existing classes

Please note that if you include a new virtual function to a class, you **must** perform a `make clean` before compiling the code again since all the classes that depend on the class you modified must be recompiled. Currently Make does not recognize this, therefore you must to it manually; otherwise you will get a `Segmentation Fault` error when running the program.

13.5.2 Corrupted double-linked list error

Sometimes, during code execution, following error is detected:
**** glibc detected *** /home/madhur/FCST/lib/fuelcell 2d.bin: corrupted double-linked list ****

This happens mostly due to compiler messing up the compiled files. When this happens, try cleaning the compiled files of FCST, using **make clean** and then re-compiling using **make**.

Bibliography

[1] M. Secanell, V. Zingan, M. Bhaiya, P. Wardlaw, M. Moore, K. Domican and P. Dobson, Fuel Cell Simulation Toolbox, User's Guide, 2013. 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] http://www.startupcto.com/server-tech/subversion/setting-the-id-tag. Accessed on June 28, 2013.

13.6 Appendix

13.6.1 Example test script

```
#!/bin/bash

#############################################################################################
# This file will run the app_cathode test that is contained in the data/cathode/testing
# folder. The script is run by ctest, keep in mind that the results that would normally
# be printed to screen will be suppressed by ctest and printed to its own file.

# The script will (by line number):
# (1) first navigate to the folder where the test is
# (2) will run the code with the correct data files.
# (3) The result from the simulation is queried to see if it ran without error by
#      checking ${PIPESTATUS[0]}. A zero means the test ran without error.
# (4) If a non-zero result is returned by the simulation, the code will print out a
#      message saying that there was an error. As this will also be suppressed by the
#      code, the message is also printed to the tests_summary.log file. The first line
#      containing a 'tee' command will create the file, subsequent calls will append
#      their output to the end of the file, so as not to overwrite it.
# (5) Before exiting, the test_summary.log file is copied to the fcst main folder
#      where it will be opened by the run_tests script.
# (6) If the simulation ran correctly, then the results from the simulation are
#      compared to expected results. Both sets of results are stored in a texts files,
#      appended with .dat. The test_comparison script is a python file that will read in
#      the two files and compare them. If they are not within a reasonable agreement the
#      python script will return a non-zero (7) and an error is printed. If they are in
#      reasonable agreement, a zero is returned indicated that all is well and a message
#      is printed. Again the message is captured by ctest, so it is also appended to the
#      tests_summary.log file.
#############################################################################################
(1)cd ../data/cathode/testing
(2)../../../lib/fuel_cell-2d.bin main_app_cathode_test.prm
(3)if [ "${PIPESTATUS[0]}" != "0" ]; then
(4) echo  2>&1 | tee tests_summary.log
    echo  "Results summary from app_cathode test:" 2>&1 | tee --append tests_summary.log
    echo  2>&1 | tee --append tests_summary.log
    echo "-------------------------------------" 2>&1 | tee --append tests_summary.log
    echo "The simulation did not run correctly. " 2>&1 | tee tests_summary.log
    echo  Please review the tests_output.log file"  2>&1 | tee tests_summary.log
    echo "-------------------------------------" 2>&1 | tee --append tests_summary.log
(4) echo  2>&1 | tee --append tests_summary.log
(5) cp tests_summary.log ../../../tests_summary.log
    exit 2
   else
(6) ../../../test/test_comparison.py dakota_tabular.dat test_results.dat
    if [ "${PIPESTATUS[0]}" != "0" ]; then
     echo  2>&1 | tee tests_summary.log
     echo "Results summary from app_cathode test:" 2>&1 | tee --append tests_summary.log
     echo 2>&1 | tee --append tests_summary.log
     echo "-------------------------------------" 2>&1 | tee --append tests_summary.log
     echo "Results from the test do not match " 2>&1 | tee --append tests_summary.log
     echo "expected results" 2>&1 | tee --append tests_summary.log
     echo "Please check results in dakota_tabular.dat "2>&1 | tee --append tests_summary.log
     echo "against that of test_results.dat" 2>&1 | tee --append tests_summary.log
     echo "-------------------------------------" 2>&1 | tee --append tests_summary.log
     echo 2>&1 | tee --append tests_summary.log
     cp tests_summary.log ../../../tests_summary.log
(7)  exit 2
    else
     echo  2>&1 | tee tests_summary.log
     echo  "Results summary from app_cathode test:" 2>&1 | tee --append tests_summary.log
     echo 2>&1 | tee --append tests_summary.log
```

```
61      echo "-------------------------------------------------" 2>&1 | tee --append tests_summary.log
62      echo "Results from the test match expected results" 2>&1 | tee --append tests_summary.log
63      echo  "-------------------------------------------------" 2>&1 | tee --append tests_summary.log
64      echo 2>&1 | tee --append tests_summary.log
65      cp tests_summary.log ../../../tests_summary.log
66      exit 0
67    fi
68  fi
```