# gdbOF: A Debugging Tool for OpenFOAM®

Juan M. Giménez     Santiago Márquez Damián
Norberto M. Nigro
International Center for Computational Methods in Engineering (CIMEC),
INTEC-UNL/CONICET,
Güemes 3450, Santa Fe, Argentina,`http://www.cimec.org.ar`

November 10, 2011

# Contents

# Preface

OpenFOAM® libraries are a great contribution to CFD community and a powerful way to create solvers and other tools. Nevertheless in this creative process a deep knowledge is needed respect of classes structure, both for value storage in geometric fields and for matrices resulting from equation systems, becoming a big challenge for debugging.
gdbOF is a new tool, attachable to gdb (GNU Debugger) which allows to analyze class structures in debugging time. This application is implemented by gdb macros, these macros can access to code classes and its data transparently, giving so, the requested information. This manual presents the key concepts of this tool and different application cases, such as assembling and storage of matrices in a scalar advective-diffusive problem, non orthogonal correction methods in purely diffusive tests and multiphase solvers based on Volume of Fluid Method. In these tests several type of data are inspected, like internal and boundary vector and scalar values from solution fields, fluxes in cell faces, boundary patches and boundary conditions. As additional features data dumping to file and graphical viewing is presented.
All these capabilities give to gdbOF a wide range of use not only in academic tests but also in real problems.

# Chapter 1

# Requirements and Installation

To install gdbOF you must have already installed OpenFOAM®[1] and compiled it in Debug mode. In `$WM_PROJECT_DIR/etc/bashrc` the environment variable `$WM_COMPILE_OPTION` can be set to Debug. That is what you need to do if you want to compile using the debug flag, or use the Debug version.

In order to leave this flag set as a default please add the following line in your `$HOME\.bashrc` file.

```
. $WM_PROJECT_DIR/etc/bashrc
```

Once you open a terminal again you can check that you are using the Debug mode by typing:

```
which icoFoam
```

which should point to a path containing the `linuxGccDPDebug` string.

Now you can compile or run the whole distribution or parts of OpenFOAM® in Debug mode. Note that you may consider not compiling ThirdPartyProducts in Debug mode, and simply make sure that the Opt version of those are used also for the Debug mode[2].

So, if you have already created your user folder (`$mkdir -p $FOAM_RUN`), you only need to decompress the downloaded tar.gz, put it into the folder and execute the installer:

```
$ sh installgdbOF.sh
```

If it was succesfull, the message `End Installation..` will be presented and gdbOF will be ready to use.

[1] gdbOF also depends on python 2.6 and gawk

[2] Hakan Nilsson, Chalmers/Applied Mechanics/Fluid Dynamics

# Chapter 2

# Basic debugging

One of the most common tasks in the debugging process is to look at the values stored in an array, that is possible in gdb with the command of Example 1, where `v` is the array to analyze.

**Example 1** View array.

```
$(gdb) p *v@v_size
```

When analyzing class attributes is required, it is necessary to know the class inheritance tree. It allows to interpret classes that contains other classes as attributes. To get the desired information it is necessary to navigate through the pointers to find an specific attribute. A typical example is to verify in debugging time that a certain boundary condition is being satisfied (typically when the boundary condition is coded directly in the solver and the next field information is obtained after solving the first time-step). Boundary conditions in OpenFOAM® are given for each patch in a `GeometricField`, then, assuming that the inspected patch is indexed as 0 (the attribute `BoundaryField` has information of all the patches), to observe the values on this patch sentence presented in Example 2 is needed, where `vSF` is a `volScalarField`.

**Example 2** View Boundary Field values.

```
$(gdb) p *(vSF.boundaryField_.ptrs_.v_[0].v_)
    @(vSF.boundaryField_.ptrs_.v_[0].size_)
```

Note that the statement in Example 2 doesn't include any call to inline functions, which could generate some problems in gdb[1], giving even more complex access to information.

*gdbOF* solves the inconvenience of knowing the attribute's place and using long statements. Using *gdbOF* commands, as it is shown in Example 3, the same results are obtained. Note the simplification of the statement, this is the *gdbOF* spirit, reducing the work needed to debug and perform the same tasks more simply and transparently.

An extra feature allows to define print limits. Choosing starting and ending indexes, only the desired value range is printed. The *gdbOF* command is called `ppatchvalueslimits` (there is

[1]Inlining is an optimization which inserts a copy of the function body directly in each calling, instead of jumping to a shared routine. gdb displays inlined functions just like non-inlined functions. To support inlined functions in gdb, the compiler must record information about inlining into debug information. gcc uses the dwarf 2 format to achieve this goal like several other compilers. On the other hand gdb only supports inlined functions by means of dwarf 2. Versions of gcc before 4.1 do not emit two of the required attributes (DW_AT_call_file and DW_AT_call_line) so that gdb does not display inlined function calls with earlier versions of gcc. [14]

**Example 3** View Boundary Field values with gdbOF.

```
$(gdb) ppatchvalues vSF 0
```

a similar command called `pinternalvalueslimits`). In Pseudo-code 1 the scheme of command implementation is presented.

**Pseudo-code 1** Structure of *gdbOF* Commands `ppatchesvalueslimits` and `pinternalvalueslimits`.

```
1. Get parameters: field name, limits and patchindex (only in patchvalueslimits)
2. Corroborate limits to print
3. Detect field type (Vol-Surface and scalar-vector-tensor)
4. Print the field values in its respective format
```

There are many examples in OpenFOAM® like the previous one in which the necessity of a tool that simplifies access to the intricate class diagram can be pondered. Note that in the last example it wasn't mentioned how the index of the desired patch is known. Usually OpenFOAM® user knows only the string that represents the patch, but not the index by which is ordered in the list of patches. Here *gdbOF* simplifies the task again, providing a command that displays the list of patches with the respective index. The used command is presented in Example 4.

**Example 4** View patches list with gdbOF.

```
$(gdb) ppatchlist
```

Another important thing to take into account at debugging time is the scope of validity of variables or object instances. To watch the values in a field or system of equations, it is necessary to generate a gdb *break* statement in a line belonging to the scope of the analyzed variable. This requires a previous code analysis prior to debugging or, at least, to recognize the object whose variables are being tested. Here OpenFOAM® introduces a further degree of complexity, and it is the inclusion of *macro C++ functions* in the code, within which gdb cannot insert *breaks*. So, to watch at the variables defined in this scope, it requires successive jumps in the code using the commands *step*, *next* and *finish*.

Here, it was only the presentation of the problem and how the tool simplifies the debugging work. For a more complete reference about other *gdbOF* macros, *gdbOF* documentation is a valuable reference.

# Chapter 3

# Advanced Debugging

## 3.1 System matrix

Increasing the complexity of debugging, it can be found cases in which not only looking for an attribute and dereference it is the solution of the problem. A typical case is the presentation of the system, $Ax = b$, generated by the discretization of the set of differential equations that are being solved and stored using *LDUAddressing* technique (see Appendix A). This technique takes advantage of the sparse matrix format and stores the coefficients in an unusual way. This storing format and the necessity of accessing and dereferencing the values forces to trace the values one by one and, at every step, assemble the matrix in the desired format. There are two commands to do this task, one for full matrices and other for sparse matrices.

In order to implement the necessary loops over the matrix elements, gdb provides a C-like syntax to implement iterative (while, do-while) and control structures (if, else). These commands have a very low performance, so the iteration over large blocks of data must be done externally. *gdbOF* becomes independent of gdb for the assembly of matrix using another platform: the `lduAddressing` vectors are exported to auxiliary files, and through calls to the shell the calculation is performed in another language. In *gdbOF*, python is chosen due to its ability to run scripts from console and having a simple file management, both to load and to save data.

It should be stressed that *gdbOF* macros for arrays have more complex options including not only to see the complete matrix ($M \times N$), but a submatrix determined by a starting $[row, col]$ pair and another finishing $[row, col]$ pair. Respect to the code, it doesn't requires more than taking care in defining the limits of the loop that reorder the matrix. Next a diagram that explains how the command `pfvmatrixfull` works (with or without limits) is presented in the Pseudo-code 2 and the diagram for the command `pfvmatrixsparse` (with o without limits) is presented in the Pseudo-code 3.

## 3.2 Mesh Search

Another group of macros are those that search in the mesh. The aforementioned inability of gdb to perform loops on large blocks of data extents to the case of meshes, forcing thus to do searching using external tools. Taking advantage that OpenFOAM® contains a battery of methods to accomplish these tasks, *gdbOF* chooses to create stand-alone applications to which call in debugging time to do the job. Even though this way means creating a new instance of the mesh in memory, the cost in time and development is lower than would be required to conduct the search in the mesh in gdb, implementing the loops in the gdb C-like syntax, or

**Pseudo-code 2** Structure of gdbOF Command `pfvmatrixfull`.

```
1. Get paramaters
2. Get upper and lower arrays with gdb
3. Redirect data to aux file
4. Format auxiliary files: gdb format → python format
5. Call python script to assemble the matrix
   (a) Read auxiliary files
   (b) Set limits
   (c) Do lduAddressing (See appendix A)
   (d) Complete with zeros
6. Format auxiliary files: python format → gdb format
7. Show in output and save file in octave format
```

**Pseudo-code 3** Structure of gdbOF Command `pfvmatrixsparse`.

```
1. Get parameters
2. Get upper and lower arrays with gdb
3. Redirect data to aux file
4. Format aux files: gdb format → python format
5. Call python script to assemble the matrix
   (a) Read aux files
   (b) Do lduAddressing for sparse matrix
   (c) Generate sparse file header
6. Format aux files: python format → gdb format
7. Show in output or/and save file in octave format adding the header to the body
```

in another language such as python. These OpenFOAM® applications are included in *gdbOF* package and are compiled when the *gdbOF* installer is run.

Cases of search in mesh typically covered by *gdbOF* are those which start with a point defined by $[x, y, z]$, returning a cell index or values in some field, either in the center of cell (volFields) or in each of its faces (surfaceFields).

Regarding to obtain the value of a field at some point there is no more inconvenient that finding the index of the cell or index of the cell containing the point, whose centroid is nearest of it. To do this, *gdbOF* uses a call to one of the applications that are compiled at installation time, but the user only needs to call the a simple command as is shown in Example 5, where `x`, `y`, and `z` are the parameters passed by the user in the command call representing the $(x, y, z)$ coordinates of the point. That command returns two indexes: the index of the cell that contains the point, and the index of the cell which has the nearest centroid. Afterward, the user put one of these indexes in the command `pinternalvalueslimits` to extract the field value in the cell centroid, or to observe the equation assembled for that cell with the command `pfvmatrix`.

A Pseudo-code of this tool is presented in Pseudo-code 4, note that it doesn't exists any

**Example 5** View cell index.

```
$(gdb) pfindCell x y z
```

communication between gdb and other platforms more that the shell call. The return of the results is through temporal files, which must be generated in a particular format to be readable by *gdbOF*. This technique is used because it is not possible to access to values in memory from one process to another process.

**Pseudo-code 4** Structure of *gdbOF* Command `pfindcell`.

```
1. Get parameters
2. Call FOAM app. to make the search
    (a) Start new case
    (b) Do search (how is explained in Appendix C)
    (c) Save results in a temporal file
3. Read temporal file using a shell script
4. Show the indexes by standard output
```

Another kind of searching through the mesh is to find a list of indices of faces belonging to a cell, this task operates in similar way. The user invokes a *gdbOF* command and this uses a backend application. Nevertheless the simplicity of using the commands, the code is more intricate because the storage of faces in a cell is not correlated, and the faces are subdivided in internal or boundary faces (this requires walking through the list of faces in the mesh). It is also needed to identify whether these faces are in the `internalField` or in one of the patches in the `boundaryField`: the last option requires seeking what is the patch which the cell is belonging to and what is the local index of the face within the patch. With this information is possible to obtain the field's value in that face. For more information see appendix C.

The *gdbOF* command `psurfacevalues` performs this search: given a cell, find the indices of the faces that make up it and the value of the chosen field in each of these faces. See Example 6.

**Example 6** View surface values

```
$(gdb) psurfacevalues surfaceField cellIndex
```

In `pfindcell`, the result stored on disk application was only necessary to parse and display it on console, but in this case, the indexes that returns the application should be used to access to an array containing the values of the field. To do that, this implementation requires to generate a temporal gdb macro (using a shell script) because it is not possible in gdb to assign the result of extracted data from a file to a variable. The Pseudo-code 5 presents this implementation.

Note that the temporal gdb macro is generated on the fly and is only functional for the parameters generated in the temporal code of the macro (Field name and location of the desired value), then the loop in all faces of the cell is transparent to the user and it is not a problem for debugging.

**Pseudo-code 5** Structure of *gdbOF* Command `psurfacevalues`.

```
1. Get parameters and check if it is a surfaceField
2. Call FOAM app. to make the search
    (a) Start new case
    (b) Do search (how is explained in appendix C)
    (c) Save results in a temporal file
3. Read temporal file using a shell script
4. Through each index:
    (a) Generate temporal macro
    (b) Call macro (this macro prints the results)
```

## 3.3 Graphical debugging

Having in mind the aim of these tools is debugging of field manipulation software, the capstone tool is finally presented. It consists in the spatial visualization of such fields in graphical form.

This is a widely spread concept which reminds us the first efforts in graphical debugging [5]. An usual application of graphical debugging are general data structures [15, 8], and particularly linked-lists [12] and graphs [11]. Data Display Debugger [16, 4] can be cited as an useful and general tool for these purposes. Respect of field manipulation software debugging, it requires mesh manipulation and more sophisticated data analisis tools which drives to specific implementations [6, 1].

In the *gdbOF* particular case, this objective summarizes previously presented tools, and it is particularly tailored for `volField` debugging. Basically it consists in an OpenFOAM® format data dump tool callable from any debugging point with optional `.vtk` file format exporting (via `foamToVtk` tool) and Paraview® [13] on the fly running. Steps to achieve this goal are presented in Pseudo-code 6

**Pseudo-code 6** Structure of *gdbOF* Command `pexportfoamformat`.

```
1. Get parameters and check if it is a volField
2. OS environment setting (first run)
    (a) Creation of data dump directories
    (b) Symbolic linkage of constant/ and system/ to avoid data duplication
3. Get actual time-step and last data written name
4. Write OpenFOAM® file format header and set field dimensions
5. Write internalField
6. Identification of boundary patches via ppatchlist calling.
7. For each patch, write boundaries' surfaceFields.
8. Close file.
9. Call optional parameters (.vtk exporting and Paraview® running)
```

# Chapter 4

# Tests

## 4.1 Scalar Transport Test

The first study case consists in the unsteady advective-diffusive equation, in a bi-dimensional mesh with $3 \times 3$ cells, which is shown in Figure 4.1.

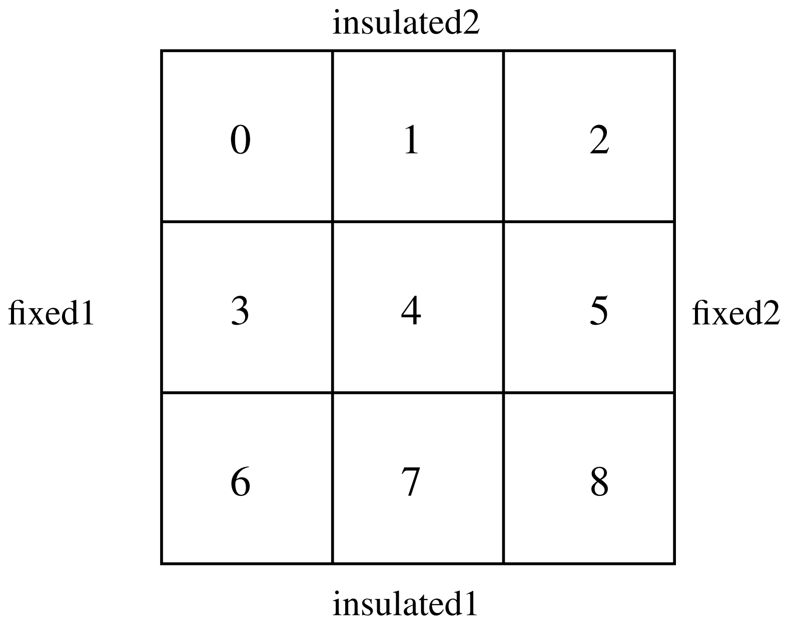

Figure 4.1: Geometry and patches in scalar transport test (numbers identify cells)

Partial differential equation is presented in Equation (4.1).

$$\frac{\partial \rho\phi}{\partial t} + \nabla \cdot (\rho \mathbf{U} \phi) - \nabla \cdot (\rho \Gamma_\phi \nabla \phi) = S_\phi(\phi) \tag{4.1}$$

with the boundary conditions shown in Equations (4.2), (4.3) and (4.4).

$$\nabla\phi \cdot \mathbf{n}|_{\text{insulated}} = 0 \tag{4.2}$$
$$\phi_{\text{fixed1}} = 373[\text{K}] \tag{4.3}$$
$$\phi_{\text{fixed2}} = 273[\text{K}] \tag{4.4}$$

To solve this problem, the following parameters are selected: $\mathbf{U} = [1,0][\frac{\mathrm{m}}{\mathrm{s}}]$, $\Delta t = 0.005[\mathrm{s}]$, $\rho = 1[\frac{\mathrm{kg}}{\mathrm{m}^3}]$, $\Gamma_\phi = 0.4[\frac{\mathrm{m}^2}{\mathrm{s}}]$, $S_\phi(\phi) = 0$ and $\phi^0 = 273[\mathrm{K}]\ \forall\ \Omega$ as initial solution.

In the Finite Volume Method, each cell is discretized as is shown in equation (4.5). [7]

$$\frac{\phi_p^n - \phi_p^0}{\Delta t} V_p + \sum_f F\phi_f^n - \sum_f \Gamma_\phi \mathbf{S_f} (\nabla\phi)_f^n = 0 \tag{4.5}$$

It is known that the assembly of a problem that includes convection using the upwind method, results in a non-symmetric matrix, in addition, increasing the diffusive term and decreasing the time step, this matrix will tend to be diagonal dominant.

Assembling the equation (4.5) in each cell for the initial time ($t = 0.005$), the system of equations presented in (4.6) is obtained.

$$\begin{aligned}
202.6\phi_0 - 0.4\phi_1 - 0.4\phi_3 &= 55271.4\\
-1.4\phi_0 + 202.2\phi_1 - 0.4\phi_4 &= 54600\\
-1.4\phi_1 + 201.6\phi_2 - 0.4\phi_5 &= 54545.4\\
-0.4\phi_0 + 203\phi_3 - 0.4\phi_4 - 0.4\phi_6 &= 55271.4\\
-0.4\phi_1 - 1.4\phi_3 + 202.6\phi_4 - 0.4\phi_5 - 0.4\phi_7 &= 54600\\
-0.4\phi_2 - 0.14\phi_4 + 202\phi_5 - 0.4\phi_8 &= 54545.4\\
-0.4\phi_3 + 202.6\phi_6 - 0.4\phi_7 &= 55271.4\\
-0.4\phi_4 - 1.4\phi_6 + 202.2\phi_7 - 0.4\phi_8 &= 54600\\
-0.04\phi_5 - 1.4\phi_7 + 201.6\phi_8 &= 54545.4
\end{aligned} \tag{4.6}$$

## OpenFOAM® Assembly

The above system, which was assembled manually, can be compared with the system obtained by running the OpenFOAM® solver `scalarTransportFoam`. First of all a directory is generated with the case described and solver is run in debug mode (`$ gdb scalarTransportFoam`). Then, a `break` is set in a line of some class that is within the scope of the object `fvScalarMatrix` containing the system of equations as is mentioned in the section 2.

Establishing a `break` in line `144` of the file `fvScalarMatrix.C`, and calling the *gdbOF* `pfvmatrixfull` command, the matrix **A** of the system is printed on the console. This coincides with the manually generated system, showing the the right performance of the tool.

An additional feature of this and other commands, is the ability to export data to a file format compatible with the calculation software Octave and Matlab®. To do this only one more parameter is needed in the command invocation, indicating the file name. *gdbOF* is responsible for export in the correct format, which can be not only rows and columns of values, but also, in `[row,col,coeff]` format. `pfvmatrixsparse` exports the matrix of the system in this format which has a header that identifies the file as sparse matrix. This method greatly reduces the size needed to store the matrices in the case of medium or large meshes.

Expanding the explanation of the last section, here it is shown the use of patch commands.

**Example 7** View system matrix with *gdbOF*

```
$(gdb) b fvScalarMatrix.C:144
$(gdb) run
$(gdb) pfvmatrixfull this fileName.txt
$(gdb) shell cat fileName.txt
202.60    -0.40     0.00   -0.40   ...
-1.40     202.20   -0.40    0.00   ...
0.00      -1.40    201.60   0.00   ...
-0.40      0.00     0.00   203.00  ...
...        ...      ...     ...     ...

(gdb) p *totalSource.v_@9
{55271.4, 54600, 54545.4, 55271.4 ...
```

Suppose that is wanted to verify if the condition $\phi = 373$[1] in the patch called *fixed1* is correctly set. First, it is necessary to know the index of this patch, as it is shown in Example 8.

**Example 8** View patches list with *gdbOF*

```
  (gdb) ppatchlist T
  PatchName   -->   Index to Use
  FIXED1    -->   0
  FIXED2    -->   1
  INSULATED2    -->   2
  INSULATED1    -->   3
  FRONT_AND_BACK    -->   4
```

Knowing the patch index, it is possible to see its values, how it is shown in Example 9. There is an array with three values corresponding to the boundary condition on each of the three faces that make up this patch.

**Example 9** View patch values with *gdbOF*

```
  (gdb) ppatchvalues T 0
  (gdb) $1 = {373,373,373}
```

Appendix B shows how the internal and boundary values (in `volFields` and in `surfaceFields`) are stored in OpenFOAM®.

## 4.2 Laplacian Test

In this problem, *gdbOF* is used to observe the fields values and the resulting equations system, in order to infer the correction method for non-orthogonality of the mesh used in OpenFOAM® (Jasak presents in his thesis [7] three methods to determine the non-orthogonal contribution in diffusion term discretization: minimum correction, orthogonal correction and over-relaxed correction[2]).

[1]In the case, $T$ is used to represent the `scalarField` in instead of $\phi$ , because OpenFOAM® uses $\phi$ for a `surfaceScalarField` that represents the flux thought each face ($\phi = S_f \cdot U_f$)

[2]The diffusive term in a non-orthogonal mesh is discretized in the following way: $\mathbf{S_f} \cdot (\nabla\phi)_f = \mathbf{\Delta_f} \cdot (\nabla\phi)_f + \mathbf{k_f} \cdot (\nabla\phi)_f$, where $\mathbf{S_f} = \mathbf{\Delta_f} + \mathbf{k_f}$. The correction methods propose different ways to find $\mathbf{\Delta_f}$.

The problem to solve is defined in the Equation (4.7), with the boundary conditions shown in (4.8), (4.9) and (4.10), and the non-orthogonal mesh presented in Figure 4.2.

$$\nabla \cdot (\rho \Gamma_\phi \nabla \phi) = 0 \tag{4.7}$$
$$\nabla \phi \cdot \mathbf{n}|_{\text{insulated}} = 0 \tag{4.8}$$
$$\phi_{\text{fixed1}} = 273[\text{K}] \tag{4.9}$$
$$\phi_{\text{fixed2}} = \phi_{right}[\text{K}] \tag{4.10}$$

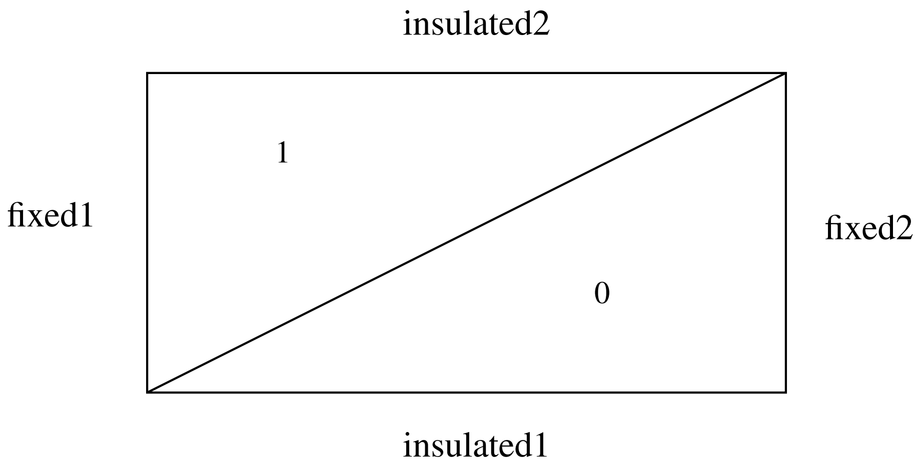

Figure 4.2: Geometry and patches in Laplacian test (numbers identifies cells).

Constants and initial conditions are: $\rho = 1$, $\Gamma_\phi = 1$ and $\phi^0 = 0[K]\ \forall\ \Omega$.

Example 10 allows to verify the proper initialization of the internal field. The list shown presents the values of the field.

**Example 10** View internalField values with *gdbOF*

```
(gdb) pinternalvalues T
(gdb) $1 = {0,0}
```

It can be shown analytically that the solution to this problem is a linear function $\phi(x) = ax + b$, and if $\phi_{\text{fixed2}} = \phi_{\text{fixed1}} \;\Rightarrow\; a = 0$ and the solution is constant, doing unnecessary the second term in non-orthogonal correction ( $\mathbf{k_f} \cdot (\nabla\phi)_f = 0$), but allows to compare the systems generated by the different approaches in comparison with the generated in OpenFOAM®, and to determine which one is the used method.

Using minimum-correction approach ($\mathbf{\Delta_f} = \frac{\mathbf{d}\cdot\mathbf{S}}{|\mathbf{d}|}\mathbf{d}$):

$$-3.29\phi_0 + 1.79\phi_1 = -409.5$$
$$1.79\phi_0 + -3.29\phi_1 = -409.5$$

Using orthogonal-correction approach ($\mathbf{\Delta_f} = \frac{\mathbf{d}}{|\mathbf{d}|}|\mathbf{S}|$):

$$-4.5\phi_0 + 3\phi_1 = -409.5$$
$$3\phi_0 + -4.5\phi_1 = -409.5$$

Using over-relaxed approach ($\mathbf{\Delta_f} = \frac{\mathbf{d}}{\mathbf{d \cdot S}}|\mathbf{S}|^2$):

$$-5.25\phi_0 + 3.75\phi_1 = -409.5$$
$$3.75\phi_0 + -5.25\phi_1 = -409.5$$

Example 11 shows how *gdbOF* extracts the equation system. Here, the reader can verify that the over-relaxed approach is implemented in OpenFOAM®.

**Example 11** Equation System debugging in LaplacianTest

```
$(gdb) b fvScalarMatrix.C:144
Breakpoint 1 at 0xb71455dc: file fvMatrices/fvScalarMatrix... line 144
$(gdb) run
...
$(gdb) pfvmatrixfull this this.txt
Saved correctly!
$(gdb) shell cat this.txt
  -5.25    3.75
  3.75     -5.25
(gdb) p *totalSource.v_@2
{-409.5, -409.5}
```

## 4.3 Multiphase Test

As the last example, a multiphase solver, namely `interFoam` is used showing *gdbOF* functionality. In this case a 2D reference problem is solved, which has analytical solution. Let be a rectangular domain with a Couette velocity profile (see Figure 4.3), and filled with a light fluid as initial condition and a domain inlet with a heavy fluid in all extension. The problem to solve is the evolution of the heavy phase through the domain along the time.

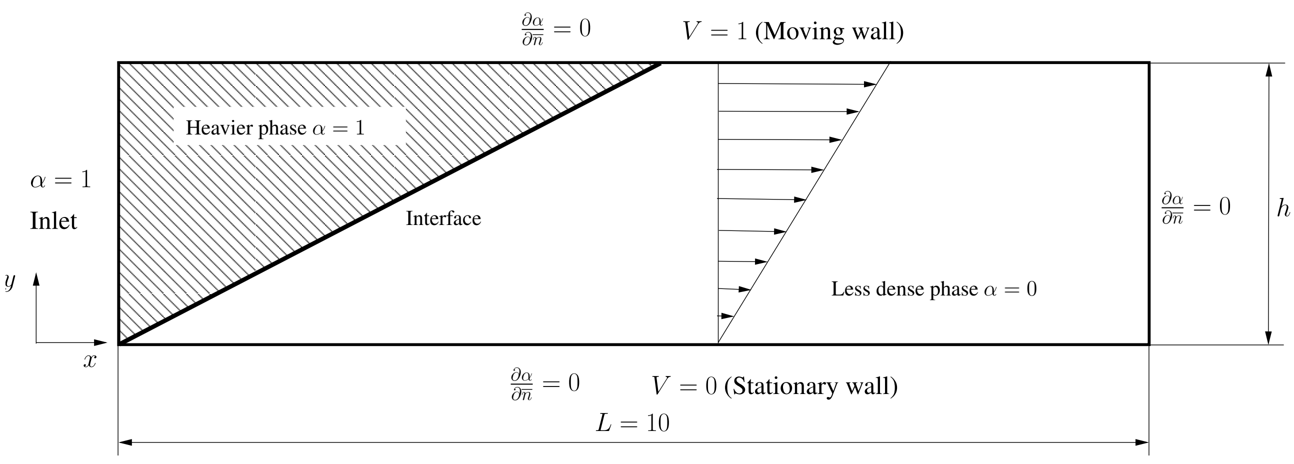
$\frac{\partial\alpha}{\partial n} = 0$
$V = 1$ (Moving wall)
Heavier phase $\alpha = 1$
$\alpha = 1$
Inlet
Interface
$\frac{\partial\alpha}{\partial n} = 0$
$h$
Less dense phase $\alpha = 0$
$y$
$x$
$\frac{\partial\alpha}{\partial n} = 0$
$V = 0$ (Stationary wall)
$L = 10$

Figure 4.3: Geometry in `interFoam` test

This two phase system is solved by means of a momentum equation (See Equation 4.11) and an advection equation for the void fraction function $\alpha$ (See Equation 4.12) [2]

$$\frac{\partial \rho \mathbf{U}}{\partial t} + \nabla\bullet(\rho \mathbf{U}\mathbf{U}) - \nabla\bullet(\mu \nabla \mathbf{U}) - (\nabla \mathbf{U})\bullet\nabla\alpha = -\nabla p_d - \mathbf{g}\bullet\mathbf{x}\nabla\rho + \sigma\kappa\nabla\alpha \tag{4.11}$$

$$\frac{\partial \alpha}{\partial t} + \nabla\bullet(\mathbf{U}\alpha) + \nabla\bullet[\mathbf{U}_r\alpha(1-\alpha)] = 0 \tag{4.12}$$

In this case, $\mathbf{g} = 0$ and it can be shown that $\nabla p_d$ and $\kappa = 0$ (no pressure gradient is needed in a velocity driven flow and curvature vanishes due a linear interface). Taking this in account, initial linear velocity profile is an spatial solution of Equation 4.11 so it reduces to Equation 4.13.

$$\frac{\partial \mathbf{U}}{\partial t} = 0 \tag{4.13}$$

From this conclusion it is clear that streamlines are horizontal, and the heavier phase advances more quickly as streamlines are closer to the top, giving a linear interface front (See Figure 4.3). This advancement is governed by an advective equation for the indicator function which includes an extra term, suitable to compress the interface [9].

Using Finite Volume Method Equation 4.12 can be discretized as in Equation 4.14 [3]

$$\frac{\alpha^{n+1} - \alpha^n}{\Delta t} V + \sum_f \left[\alpha_f^n \phi_f^n + \alpha_f^n \left(1 - \alpha_f^n\right) \phi_{rf}^n\right] = 0 \tag{4.14}$$

where $\phi_f^n = \mathbf{U}^n \cdot \mathbf{S}_f$, $\phi_{rf}^n = \mathbf{U}_r^n \cdot \mathbf{S}_f$ and superindex $n$ implies the time-step. $\mathbf{U}_r$ is the compressive velocity and is calculated directly as a flux: $\phi_{rf} = n_f \min\left[C_\alpha \frac{|\phi|}{|\mathbf{S}_f|}, \max\left(\frac{|\phi|}{|\mathbf{S}_f|}\right)\right]$. $C_\alpha$ is an adjustment constant and $n_f = \frac{(\nabla\alpha)_f}{\left|(\nabla\alpha)_f + \delta_n\right|}\bullet\mathbf{S}_f$ is the face unit normal flux with $\delta_n$ as a stabilization factor [2]. $\phi_{rf}$ values are variable only vertically in this example and will be checked at debugging time against those can be calculated from theory, using *gdbOF* tools. In this case, due how advective terms are calculated there is necessary to show values at faces.

Domain was meshed as a 3D geometry due to OpenFOAM® requirements [10] with a $100 \times 10 \times 1$ elements grid, so each hexahedron has edges of 0.1 units in size. Since its definition and taking $C_\alpha = 1$, $|\mathbf{U}_r| = |\mathbf{U}|$, therefore $\phi_{rf} = \mathbf{U}_r \cdot \mathbf{S}_f = 0.01\,|\mathbf{U}_r|\,\left(\check{\mathbf{U}}_r \cdot \check{\mathbf{S}}_f\right)$. So taking three distances from bottom edge of the domain, $y = 0.05$, $y = 0.45$ and $y = 0.95$, values for $\phi$ in faces with $\mathbf{S}_f$ aligned with $x$ direction must be $|\phi_{rf}| = 0.005$, $|\phi_{rf}| = 0.045$ and $|\phi_{rf}| = 0.095$ respectively.

As the first stage, it is necessary to find the indices of three cells with such a $y$ coordinates, taking for example $x = 0.05$, and using `pFindCell` tool results shown in Example 12 can be obtained.

**Example 12** View cell index in multiphase problem.

```
(gdb) pfindcell 0.05 0.45 0.05
RESULTS:
Nearest cell centroid cell number: 400
Containing point cell number (-1=out) : 400
```

As it was explained in Subsection 3.2 the only index of the cell is not enough to address the values in the `internalField` of a given field. Each cell has as many surface values as faces in the cell, therefore is necessary to show all these values, and each face has an addressing index not necessarily correlative.

`psurfacevalues` *gdbOF* command simplifies this task. Knowing the index of the cell to analyze, it returns the information on each face about the field indicated in the command line parameters: boundary face or internal face (categorized according to whether it has a neighbour or not) and field value. If it is working with a 2D mesh, information is also returned as in a 3D mesh (6 faces for a hexahedron), but it indicates on which of these the condition is empty.

To perform this task, different methods of some of the classes in charge of managing OpenFOAM® mesh are called by means of various *OpenFOAM® applications* (included in *gdbOF* package) running on the backend and returning the results (index of cells, or faces) at *gdbOF* macros. Then these are responsible for finding the value of the field in debugging time (see Appendix C or the Subsection 3.2).

So that, applying this command to the previous found cell it is possible to show $\phi$ in all faces of that cell (See Example 13)

**Example 13** Example of usage of `psurfacevalues` for face defined field.

```
(gdb) psurfacevalues phir 400
internal Face:
$5 = 0
internal Face:
$6 = -0.0045
internal Face:
$7 = 0
empty Face
empty Face
boundary Face:
$8 = 0.0045
```

Results are consistent with original problem. Two faces are marked as *empty* because the mesh has only one cell in depth. This boundary condition is used by OpenFOAM® to represent no variability in direction perpendicular to the face, allowing a 2D calculation. Faces 5 and 7 corresponds to top and bottom faces of the cell where flux is null. Finally, faces 6 and 8 have face normals aligned with the velocity and flux values are that were predicted theoretically for $y = 0.45$. Values have different sign due to the faces have opposite normals direction.

Regarding graphical debugging presented in Section 3.3 `pexportfoamformat` is a useful tool to inspect the $\alpha$ field as in Figure 4.3. To do so, command is invoked as in Example 14 and results are shown in Figure 4.4.

**Example 14** Field exporting to `.vtk` by means of `pexportfoamformat`. Paraview® is invoked as well

```
(gdb) pexportformat alpha1 VTK Paraview
      Including internal field...
      Saved correctly!
      Including fixedWall boundary field...
      Empty Condition, no values
      Including movingWall boundary field...
      Saved correctly!
      Including inlet boundary field...
      Saved correctly!
      Including outlet boundary field...
      Saved correctly!
      Including frontAndBackPlanes boundary field...
      Saved correctly!
      Exporting to VTK...
      Launching Paraview...
```

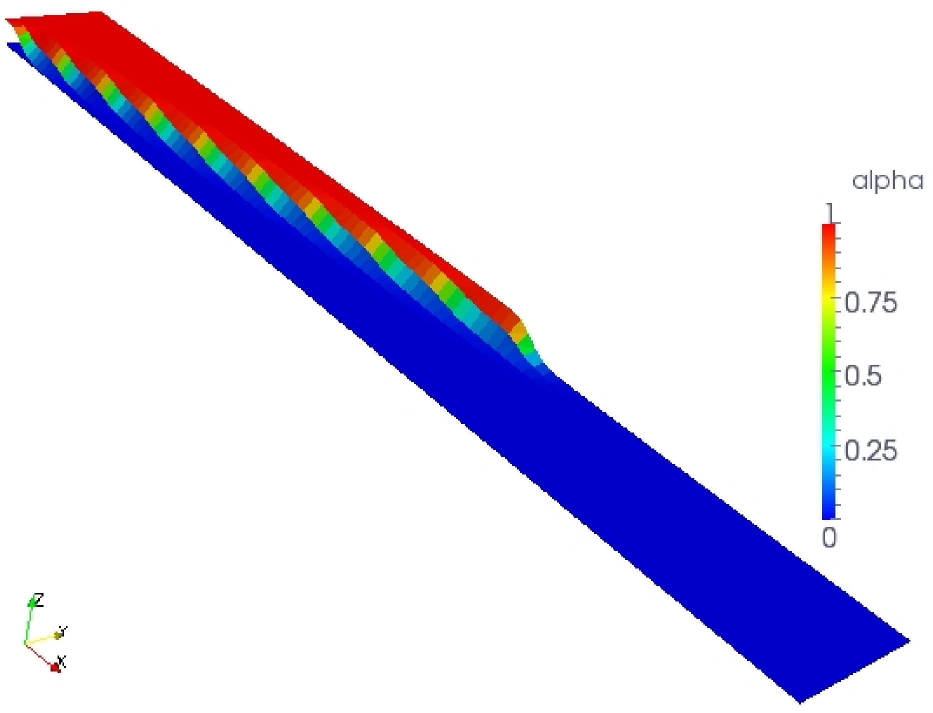

Figure 4.4: $\alpha$ field representation in Paraview® using `pexportfoamformat`

# Bibliography

[1] D. Abramson, I. Foster, J. Michalakes, and R. Sosič. Relative debugging: A new methodology for debugging scientific applications. *Communications of the ACM*, 39(11):69–77, 1996.

[2] E. Berberovic, N.P. Van Hinsberg, S. Jakirlic, I.V. Roisman, and C. Tropea. Drop impact onto a liquid layer of finite thickness: Dynamics of the cavity evolution. *Physical Review E*, 79, 2009.

[3] P. Bohorquez R. de M. *Study and Numerical Simulation of Sediment Transport in Free-Surface Flow.* PhD thesis, Málaga University, Málaga, 2008.

[4] D. Cruz, P. Henriques, and M.J. Pereira. Alma versus ddd. 2008.

[5] A.D. Dewar and J.G. Cleary. Graphical display of complex information within a prolog debugger. *International Journal of Man-Machine Studies*, 25(5):503–521, 1986.

[6] V. Grimm. Visual debugging: A way of analyzing, understanding and communicating bottom-up simulation models in ecology. *Natural Resource Modeling*, 15(1):23–38, 2002.

[7] H. Jasak. *Error analysis and estimation for the finite volume method with applications to fluid flows.* PhD thesis, Department of Mechanical Engineering Imperial College of Science, Technology and Medicine, 1996.

[8] J.L. Korn and Princeton University. Dept. of Computer Science. *Abstraction and visualization in graphical debuggers.* Princeton University Princeton, NJ, USA, 1999.

[9] OpenCFD. OpenCFD Technical report no. TR/HGW/02 (unpublished), 2005.

[10] OpenCFD. *OpenFOAM, The Open Source CFD Toolbox, User Guide.* OpenCFD Ltd., 2009.

[11] G. Parker, G. Franck, and C. Ware. Visualization of large nested graphs in 3d: Navigation and interaction. *Journal of Visual Languages and Computing*, 9(3):299–317, 1998.

[12] T. Shimomura and S. Isoda. Linked-list visualization for debugging. *Software, IEEE*, 8(3):44–51, 1991.

[13] A.H. Squillacote and J. Ahrens. *The paraview guide.* Kitware, 2006.

[14] R. Stallman, R. Pesch, and S. Shebs. *Debugging with GDB: The GNU Source-Level debugger.* GNU Press, Free Software Foundation Inc., 9th edition, 2002.

[15] V. Waddle. Graph layout for displaying data structures. In *Graph Drawing*, pages 98–103. Springer, 2001.

[16] A. Zeller and D. Lutkehaus. DDD A free graphical front-end for unix debuggers. *ACM Sigplan Notices*, 31(1):22–27, 1996.

# Appendix A

# Matrix Storage in OpenFOAM®

The discretization of a set of differential equations, can be described in matrix form

$$\mathbf{A}\, x = b \tag{A.1}$$

where **A** is a sparse block matrix, that can be inverted to solve the system. However OpenFOAM® do not use the typical sparse storage form, but uses another form of storage that is called **LDU Addressing**. This technique consists in decompose the matrix coefficients in three arrays: one for diagonal coefficients called *diag*, and the others two for the non-zero coefficients in lower and upper triangulars, called *lower* and *upper* respectively. Also, exists other two arrays that indicates the position in the matrix of each coefficient, they are called *lowerAddr* and *upperAddr*, where `lowerAddr[i]` represents the smallest cell's index (in the lower triangular will the row index and in the upper triangular will the column index), meanwhile `upperAddr[i]` represents the biggest index.

A pseudocode to assemble the full matrix is presented in Code 7.

**Pseudo-code 7** Assembly with LDU Addressing.

```
for k : sizeDiag
  A[k][k] = diag[k]
end for

for k : sizeAddr
  i = lowerAddr[k]
  j = upperAddr[k]
  A[i][j] = upper[k]
  A[j][i] = lower[k]
end for
```

It should be stressed that in case of a symmetric matrix, the `upper` and `lower` vectors are identical, so that only one of them is stored. To access to the complementary vector, the original one is referenced.

## OpenFOAM® Files References

- ~/OpenFOAM/OpenFOAM-<version>/src/finiteVolume/fvMatrices/fvMatrix/fvMatrix.H
- ~/OpenFOAM/OpenFOAM-<version>/src/finiteVolume/fvMatrices/fvMatrix/fvMatrix.C
- ~/OpenFOAM/OpenFOAM-<version>/src/matrices/lduMatrix/lduAddressing/lduAddressing.H
- ~/OpenFOAM/OpenFOAM-<version>/src/matrices/lduMatrix/lduAddressing/lduAddressing.C

# Appendix B

# volFields and surfaceFields

`volFields` contain values in each cell centroid, this make up the `internalField`, and the index in this array is equivalent to the cell index in the mesh. Each patch is represented with a `surfaceField`, and all together make up the `boundaryField`.

**Pseudo-code 8** Recovering Internal and Boundary values

```
l = myVolField.internalField.size_
i = 0
while(i<l)
  intFieldValue = myVolField.internalField.v_[i]
  //do something
  i++

l = myVolField.boundaryField.ptrs_.size_
while(i<l)
  patch = myVolField.boundaryField.ptrs_.v_[i]
  l2 = patch.size_
  j = 0
  while(j<l2)
    patchFieldValue = patch.v_[j]
    //do something
    j++
  i++
```

It should be mentioned that the value of field can be `scalar`, `vector` or `tensor`, depending on the specialization of `volField` (`volScalarField`, `volVectorField` or `volTensorField`).

The difference between `volFields` and `surfaceFields`, is that the first one stores in `internalField` the field values at the centroids of each cell, while the second stores in the `internalField` the field values at the internal faces (those which have `owner` and `neighbour`). Retrieving the values is similar to that was previously presented (Pseudo-code 8), but require some manipulation of the values to determine the correspondence cell faces, since the field values on each face for a given cell are not contiguous in the array. (see Appendix C).

## OpenFOAM® Files References

- ~/OpenFOAM/OpenFOAM-<version>/src/OpenFOAM/fields/GeometricFields/GeometricField.H
- ~/OpenFOAM/OpenFOAM-<version>/src/OpenFOAM/fields/GeometricFields/GeometricField.C
- OpenFOAM® Programmer's Guide, chapter 2.3: Discretisation of the solution domain.

# Appendix C

# Values in cell faces of surfaceFields

This appendix explains how the fields values in cells and in each face that make up the cell are stored.

Given a point and using mesh search methods (implemented with octrees) provided OpenFOAM® it is possible to find the index of the cell whose centroid is closest to the point or in which this point is contained.

This index is directly used in collecting the field value (in `volFields`) for that cell. An example of this technique is presented in the Pseudo-code 9.

**Pseudo-code 9** Recovering field value

```
cellIndex = mesh.searchCellIndex(point)
fieldValue = field.internalField.v_[cellIndex]
```

Nevertheless, in the case of values on the faces of the cell a disadvantage arises, it is that there is no data structure to map *cellIndex* => $facesIndex$ (where $facesIndex$ is a vector with faces indexes in a `surfaceField`) so that the search should be done through the faces list, checking if the cell is the face's *owner* or *neighbour*.

In the case of inner faces, the face index found is that allows to access the internal `surfaceField` to extract the value of that field in the face. But in the case of boundary faces, a distinction have to be done because belonging to a patch implies a local index of the face within the patch. OpenFOAM® includes methods that simplifies the search of the local index to the simply calling of a function.

A pseudocode for the case of `surfaceField` is presented in the Pseudo-code 10.

## OpenFOAM® Files References

- ~/OpenFOAM/OpenFOAM-<version>/src/meshTools/meshTools/meshTools.H
- ~/OpenFOAM/OpenFOAM-<version>/src/meshTools/meshTools/meshTools.C
- OpenFOAM® Programmer's Guide, chapter 2.3: Discretisation of the solution domain.

**Pseudo-code 10** Recovering field faces values

```
cellIndex = mesh.searchCellIndex(point)
for f : nFaces
  fieldFaceValue = false
  if isInternalFace(f)
    if owner[f]==cellIndex || neighbour[f]==cellIndex
      fieldFaceValue = field.internalField[f]
  else
    if owner[f]==cellIndex
      patchIndex = whichPatch(f)
      f_local = whichFace(f,patchIndex)
      fieldFaceValue = field.boundaryField[patchIndex][f_local]
  if(fieldFaceValue)
    //do something with fieldFaceValue
end for
```

# Appendix D

# gdbOF commands table

| **Command** | **Description** |
|---|---|
| ppatchlist | Prints a list with patches' names and IDs |
| pinternalvalues | Prints the internal field values of geometric fields |
| pinternalvalueslimits | Prints the internal field values of geometric fields in a specified range |
| pfieldvalues | Prints the values of simple fields |
| pfieldvalueslimits | Prints the values of simple fields in a specified range |
| ppatchvalues | Prints the field values on the selected patch |
| ppatchvalueslimits | Prints the field values on the selected patch in a specified range |
| pfvmatrixfull | Dumps a fvMatrix in Octave/Matlab® format |
| pfvmatrixsparse | Dumps a fvMatrix in Octave/Matlab® sparse format |
| pfindcell | Prints the nearest cell centroid index and the cell centroid index that contains a given point |
| pfindface | Prints the nearest patchID from a point, and the nearest faceID in this patch |
| psurfacevalues | Prints the field values on the faces of a given cell |
| pexportfoamformat | Exports vol*Field in FOAM format. It allows converting it to VTK and open with Paraview |