Asociación Argentina de Mecánica Computacional

Mecánica Computacional Vol XXIX, págs. 3441-3460 (artículo completo)
Eduardo Dvorkin, Marcela Goldschmit, Mario Storti (Eds.)
Buenos Aires, Argentina, 15-18 Noviembre 2010

GDBOF: A DEBUGGING TOOL FOR OPENFOAM®

Juan M. Gimenez[a], **Santiago Márquez Damián**[a,b] **and Norberto M. Nigro**[a,b]

[a]*Facultad de Ingeniería y Ciencias Hídricas - Universidad Nacional del Litoral. Ciudad Universitaria. Paraje "El Pozo". Santa Fe. Argentina. jmarcelogimenez@gmail.com, http://www.fich.unl.edu.ar*

[b]*International Center for Computational Methods in Engineering (CIMEC), INTEC-UNL/CONICET, Guemes 3450, Santa Fe, Argentina, http://www.cimec.org.ar*

Keywords: Data Structures, Debugging, OpenFOAM, gdbOF

Abstract. 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 of classes structure, both for value storage in geometric fields and for matrices resulting from equation systems is needed, becoming a big challenge for beginners.
To help in this process a new tool, called gdbOF, attachable at gdb (GNU Debugger) is presented. It 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. Tool is tested in 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 an additional feature data dumping to file is presented.
All these capabilities give to gdbOF a wide range of use not only in academic tests but also in real problems.

1 INTRODUCTION

OpenFOAM® is a CFD library that allows to the user to program different solvers and tools (for pre-processing or post-processing) in a high-level specific language, understanding for high-level the fact of writing in a notation closer to mathematical description of the problem, releasing the user from the internal affairs of the code.

This programming approach contrasts with procedural languages approach, such as Fortran, that are widely used in academic and scientific environments but oriented to the low-level problem resolution, i.e., to the manipulation of individual floating-points values. Then, to abstract from the low-level coding is necessary to use other approach, being adopted in this work the Object-Oriented Programming (OOP) methodology. This methodology produces code that is easier to write, validate and maintain than with procedural techniques. OOP paradigm exists in many languages, such as SmallTalk or Eiffel, but for the development of this library C++ was chosen. It is less rigorously object-oriented than the others languages, because allows to include different characteristics that are not strictly object-based. The main addon is operator overloading, which is essential to interface the standard numerical notation in order to work with tensorial, vector and scalar fields objects concept. Also, its is a multiplatform language and, being based on C, is so fast as procedural languages (Cary et al., 1997).

Exists five fundamentals concepts in OOP, whereby OpenFOAM® achieves its goal: *modulation*, *abstraction*, *encapsulation*, *inheritance* and *polimorphism*. (Weller et al., 1998).

The modulation allows to subdivide the original problem into several subproblems. The abstraction consists in extracting essential aspects of an entity and ignore their accidental or irrelevant properties, therefore determines the level of detail. Encapsulation separates the contractual interface of the abstraction from the implementation, then abstraction and encapsulation are two complementary concepts: the first focuses on the observable behavior of an object, whereas the second focuses on the implementation that leads to such behavior. These two features allow to create abstract data types that represent real entities, i.e. tensor fields.

In addition, inheritance is the mechanism to define a new class (subclass) from another class (superclass), so that the subclass inherits the structure (state) of the superclass through variables, and the behavior through the methods. This allows to create new classes that include new behavior without the need to modify existing classes. Polymorphism is the ability of objects to provide same external interface but different internal implementations, thus complementing the inheritance. Focusing on OpenFOAM® code, there is a proliferation of virtual methods (methods that must be implemented in child classes), which allow the concept of polymorphism. Examples of this include the implementation of boundary conditions, which inherit from a base class `patchField`, so they have the same interface but different implementations. Another example is representation of tensorial fields: `geometricField` is the parent class and various tensor fields inherit from it: with rank 0 called `scalarField`, rank 1 called `vectorField` and rank 2 called `tensorField`, each implementing the interface that provides the parent class in different ways[1].

In addition to these used OOP features, there are other tools of C++ language which are not strictly object-based and are widely used in OpenFOAM®, like the aforementioned opera-

[1]Simplification of structure, really more classes are involved.

tor overloading and the use of macros. Macros allows to insert code directly in the program, avoiding the overhead of invoking a function (passing parameters to the stack, do a jump, take parameters), without losing the readability of it (Eckel, 2000).

As it was mentioned, using these techniques generates a library oriented to high-level development, ensuring that the user only have to take care about the model to solve and not in other details of coding (Mangani et al., 2007). Nevertheless, for developers, problems arise in applications creation which not yield the desired results. There begins the code debugging work, and this includes monitoring values corresponding to variables involved in the resolution, such as, float values in each of the points (cell centers or faces) of tensor fields, vector and/or scalar values, coefficients in the system matrix, and many other examples. On the other hand, debugging is not ever motivated by problems, but simply for exploratory or control purposes. (Ewer et al., 1995)

The debugger software for excellence in GNU-Linux platforms is *gdb* (GNU-debugger), which includes numerous tools for analyzing the code at runtime (Matloff and Salzman, 2008). The developer, almost intuitively, links a series of numerical values (such as the values of a field) to a contiguous array in memory. For these cases gdb offers a powerful print command, which together with the dereference operators "*" and "@", let to see this sequence of values in memory in a very simple way. The main problem for the developer, who wants to track variables, is the needing of knowing the structure of the class tree of OpenFOAM®. This problem gets worse when exploring data involves the polymorphism and inheritance with the virtual methods widely used by the library. Exploring the general class tree looking for the attribute class of the object which is wanted to observe the values, is a task that demands a considerable loss of time. Moreover, once desired attributes are found, these maybe do not directly represent the information required by the developer. In the case of the matrix generated by `fvm`, it stores the coefficients using the technique of LDU Addressing (see appendix A), so it is necessary to apply a decoding algorithm to transform it into the traditional format (full or sparse), and to control and to operate with their values.

Solving the problems explained in the previous paragraph is the main objective of the tool presented in this work, which is called *gdbOF*. This tool is implemented by gdb macros and is based on an implementation of gdb macros for debugging the STL (Standard Library for C++) (Marinescu, 2008). These macros simplify the task of debugging the OpenFOAM® libraries, performing the above actions transparently to the user: the simple call of a gdb macro from console triggers a sequence of actions that include: navigate the OpenFOAM® class tree, collect information and reorder it for representation in a format readable by the user. Moreover, *gdbOF* includes the option of writing the output into a file on disk. This output is formatted appropriately to be imported in numerical computation software such as Octave or Matlab®, thus allowing the developer to expand the possibilities of data control in debugging time.

In this document several cases will be treated, all of them justifies the use of this new tool as a quick way to check the flow of data and Finite Volume techniques by which some typical tasks of CFD problems are solved. These problems not only emerge in academic circles but also occur in real application environments: the first consists in a scalar advective-diffusive problem, in which the emphasis will be placed on the assembling and storage of matrices, the second consists in non orthogonal correction methods in purely diffusive tests, and the third is an analysis of Multiphase solvers based on Volume of Fluid Method.[2]

2 PROBLEMS WITH 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 established on each patch in each `GeometricField`, then, assuming that the patch of interest is one whose index is 0 (the attribute `BoundaryField` has information of all the patches), to observe the values on this patch the command 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 function, which can generate some problems in gdb[3], 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.

Example 3 View Boundary Field values with gdbOF.

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

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` (exists a

[2]gdbOF is tested in gdb 7.0 or earlier versions. It is known a issue with the version 7.1 about C string functions, it produces crashes in some gdbOF commands.

[3]Inlining is an optimization that inserts a copy of the function body directly at 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 in the debug information gcc using the dwarf 2 format does this, and several other compilers do also. gdb only supports inlined functions when using dwarf 2. Versions of gcc before 4.1 do not emit two required attributes (DW_AT_call_file and DW_AT_call_line); gdb does not display inlined function calls with earlier versions of gcc. (Stallman et al., 2002)

similar called `pinternalvalueslimits`). In Pseudo-code 1 a pseudo-code of that 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 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 tasks. For a more complete reference about other *gdbOF* macros, *gdbOF* documentation is a valuable reference.

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 (to understand how the system is stored by OpenFOAM® please read the appendix A, which presents the *LDUAddressing* technique). 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 include more complex options including not to see the complete matrix ($M \times N$), but only a submatrix determined by a starting pair $[row, col]$ and another finishing pair $[row, col]$. Respect to the code, it doesn't represent more than taking care in defining the limits of the loop that reorder the matrix. Next a diagram that explains the command works `pfvmatrixfull` (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.

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
```

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 other 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 debugging, implementing the loops in the gdb C-like sintax, or in another language such as python. These OpenFOAM® applications are included in *gdbOF* package and are compiled when the *gdbOF* installer is run.

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
```

Cases of search in mesh that are typically covered by *gdbOF* are those which starts 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 call the command 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 nearest cell. Afterwards, 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`.

Example 5 View cell index.

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

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

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

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 C)
    (c) Save results in a temporal file
3. Read temporal file using a bash script
4. Show in output the indexes
```

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, using a bash script, a temporal gdb macro because it is not possible in gdb to assign the result of extracting data from a file to a variable. The Pseudo-code 5 presents this implementation.

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 bash script
4. Through each index:
    (a) Generate temporal macro
    (b) Call macro (this macro prints the results)
```

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), but the loop in all faces of the cell is transparent to the user and it is not a problem for debugging.

4 TESTS

4.1 Scalar Transport Test

The first study case consists of the unsteady advective-diffusive equation, in a bidimensional mesh with 3×3 cells, which is shown in Figure 1.

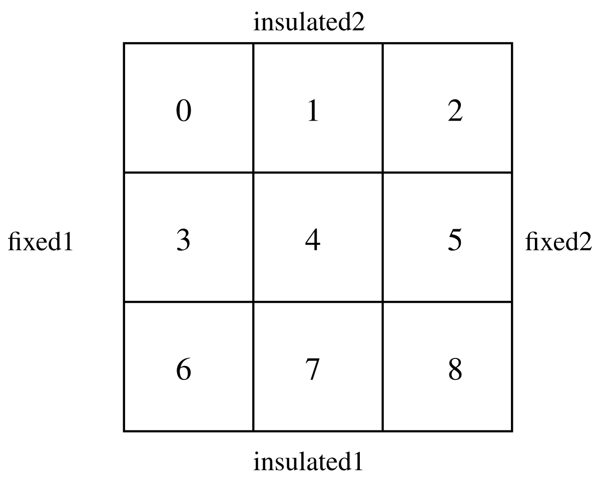

Figure 1: Geometry and patches in scalar transport test (numbers idenfies cells

Partial differential equation is presented in Equation (1).

$$\frac{\partial \rho\phi}{\partial t} + \nabla \cdot (\rho \mathbf{U} \phi) - \nabla \cdot (\rho \Gamma_\phi \nabla \phi) = S_\phi(\phi) \tag{1}$$

with the boundary conditions shown in Equations (2), (3) and (4).

$$\nabla\phi \cdot \mathbf{n}|_{\text{insulated}} = 0 \tag{2}$$
$$\phi_{\text{fixed1}} = 373[\text{K}] \tag{3}$$
$$\phi_{\text{fixed2}} = 273[\text{K}] \tag{4}$$

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

In the Finite Volume Method, each cell is discretized as is shown in equation (5). (Jasak, 1996)

$$\frac{\phi_p^n - \phi_p^0}{\Delta t} V_p + \sum_f F \phi_f^n - \sum_f \Gamma_\phi \mathbf{S_f} (\nabla\phi)_f^n = 0 \tag{5}$$

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 (5) in each cell for the initial time ($t = 0.005$), the system of equations presented in (6) is obtained.

$$
\begin{aligned}
202.6\phi_0 - 0.4\phi_1 - 0.4\phi_3 &= 55271.4\\
-1.4\phi_0 + 202.2\phi_1 - 0.4\phi_4 &= 54600\\
-1.4\phi_1 + 201.6\phi_2 - 0.4\phi_5 &= 54545.4\\
-0.4\phi_0 + 203\phi_3 - 0.4\phi_4 - 0.4\phi_6 &= 55271.4\\
-0.4\phi_1 - 1.4\phi_3 + 202.6\phi_4 - 0.4\phi_5 - 0.4\phi_7 &= 54600\\
-0.4\phi_2 - 0.14\phi_4 + 202\phi_5 - 0.4\phi_8 &= 54545.4\\
-0.4\phi_3 + 202.6\phi_6 - 0.4\phi_7 &= 55271.4\\
-0.4\phi_4 - 1.4\phi_6 + 202.2\phi_7 - 0.4\phi_8 &= 54600\\
-0.04\phi_5 - 1.4\phi_7 + 201.6\phi_8 &= 54545.4
\end{aligned}
\tag{6}
$$

4.1.1 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` called, 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.

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 ...
```

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 `ppatchs` commands. Suppose that is wanted to verify if the condition $\phi = 373$[4] 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 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 (Jasak, 1996) three methods to determine the non-orthogonal contribution in diffusion term discretisation: minimum correction, orthogonal correction and over-relaxed correction[5]).

The problem to solve is defined in the Equation (7), with the boundary conditions shown in (8), (9) and (10), and the non-orthogonal mesh presented in Figure 2.

$$\nabla \cdot (\rho \Gamma_{\phi} \nabla \phi) = 0 \tag{7}$$

$$\nabla \phi \cdot \mathbf{n}|_{\text{insulated}} = 0 \tag{8}$$

$$\phi_{\text{fixed1}} = 273[\text{K}] \tag{9}$$

$$\phi_{\text{fixed2}} = \phi_{right}[\text{K}] \tag{10}$$

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

[5]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 $\mathbf{\Delta_f}$.

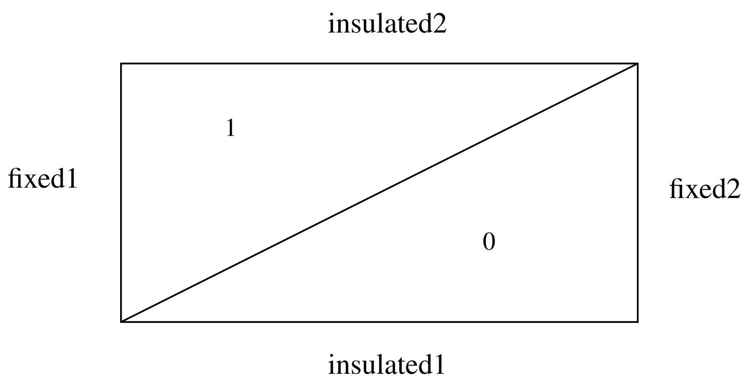

Figure 2: Geometry and patches in Laplacian test (numbers identifies cells).

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 ($\mathbf{\Delta_f} = \frac{\mathbf{d}}{\mathbf{d}\cdot\mathbf{S}}|\mathbf{S}|^2$):

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

Example 11 shows how *gdbOF* extracts the equations system was shown. 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 thought the domain along the time.

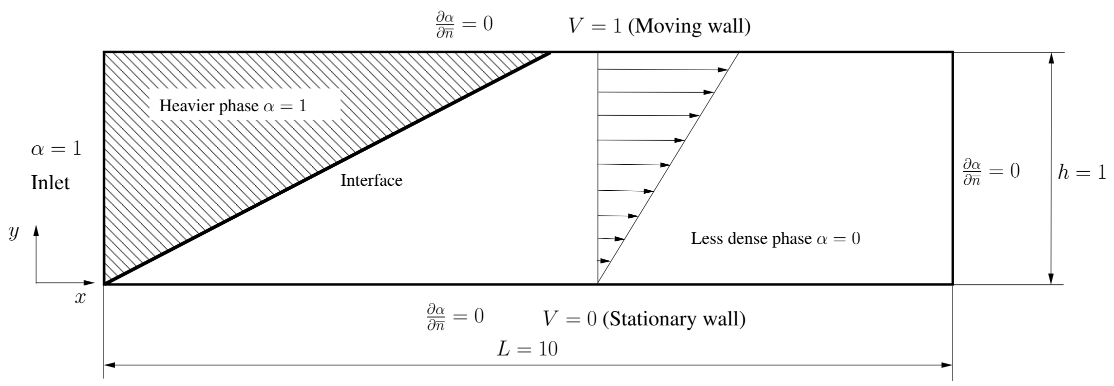
$\frac{\partial\alpha}{\partial n}=0$
$V=1$ (Moving wall)
Heavier phase $\alpha=1$
$\alpha=1$
Inlet
Interface
$\frac{\partial\alpha}{\partial n}=0$
$h=1$
Less dense phase $\alpha=0$
y
x
$\frac{\partial\alpha}{\partial n}=0$
$V=0$ (Stationary wall)
$L=10$

Figure 3: Geometry in `interFoam` test

This two phase system is solved by means of a momentum equation (See Equation 11) and an advection equation for the void fraction function α (See Equation 12) (Berberovic et al., 2009)

$$\frac{\partial\rho\mathbf{U}}{\partial t}+\nabla\bullet(\rho\mathbf{U}\mathbf{U})-\nabla\bullet(\mu\nabla\mathbf{U})-(\nabla\mathbf{U})\bullet\nabla\alpha=-\nabla p_d-\mathbf{g}\bullet\mathbf{x}\nabla\rho+\sigma\kappa\nabla\alpha \tag{11}$$

$$\frac{\partial\alpha}{\partial t}+\nabla\bullet(\mathbf{U}\alpha)+\nabla\bullet[\mathbf{U}_r\alpha(1-\alpha)]=0 \tag{12}$$

In this case, $\mathbf{g}=0$, $\rho=1$ and it can be shown that ∇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 11 so it reduces to Equation 13.

$$\frac{\partial\mathbf{U}}{\partial t}=0 \tag{13}$$

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 indicator function advective equation which includes an extra term, suitable to compress the interface (OpenCFD, 2005).

Using Finite Volume Method Equation 12 can be discretized as in Equation 14 (Bohorquez R. de M., 2008)

$$\frac{\alpha^{n+1}-\alpha^{n}}{\Delta t}V+\sum_{f}\left[\alpha_f^n\,\phi_f^n+\alpha_f^n\left(1-\alpha_f^n\right)\phi_{rf}^{\,n}\right]=0 \tag{14}$$

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 timestep. $\mathbf{U}_r$ 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_α 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 δ_n as a stabilization factor (Berberovic et al., 2009). ϕ_{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 (OpenCFD, 2009) 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 ϕ 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 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: 400
Cell Centroid in which the point is contained (-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 necesarelly correlative.

`psurfacevalues` *gdbOF* command simplifies this task. Knowing the index of the cell to analyze, it returns the info on each face about the field indicated in the command line parameters: boundary face or internal face (categorized according to whether it has a neighbor 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 ϕ 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.

5 CONCLUSIONS

OpenFOAM® is a free software tool that enables high-level programming at the same level as the mathematical expression that solves the problem. The use of C++ programming language and all its object-oriented machinery allows a fast and expandable code improving the performance of procedural languages. The OOP approach provides greater simplicity for maintenance and code expansion, since it uses five main features: modularity, abstraction, encapsulation, inheritance and polymorphism. To complete the machinery, the flexibility of C++ allows to include non object-based elements, such as operator overloading and the definition of macros instead of functions.

The downside of all these benefits is an intricate code, which is difficult to learn and analyze. These difficulties arise, for example, when there are problems with unwanted results, or simply looking for analyzing if a procedure is properly executed. These tasks require debugging with *gdb*, but given the complexity of the aforementioned code it is necessary to expand the capabilities of the debugger with a set of commands that allow simply navigation through the code and class inspection, giving the place to *gdbOF*.

So, *gdbOF* was presented simplifying the tasks to a single line. In addition, due to the use of parameters in each command, *gdbOF* offers the versatility of adapting to different objects that exists in a typical OpenFOAM® simulation, such as `volFields` and `surfaceFields` each with its derivatives `Scalar`, `Vector` and `Tensor`. Another benefit presented and tested was the ability to do geometric searches in the mesh at debugging time, being able to find the index of a particular cell or face, or the list of faces surrounding a cell. All of this tasks were achieved through calls to backend applications which are included in the *gdbOF* distribution package. In addition to this list of benefits it may be included the ability to obtain the system of equations associated to the discrete version of the problem, allowing to see it in various formats and exporting it to disk to be manipulated with other software, and the possibility of extracting only a sub-matrix to analyze only a specific part of it.

All of *gdbOF* features described above, allows to the user to have greater efficiency and flexibility in debugging OpenFOAM®. All the tools included in *gdbOF* may be a of great to developers, therefore it is considered that this package can be widely used in the community.

REFERENCES

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

Bohorquez R. de M. P. *Estudio y Simulación Numérica del Transporte de Sedimentos en Flujos con Superficie Libre*. Ph.D. thesis, Málaga University, Málaga, 2008.

Cary J., Shasharina S., and Cummings J. Comparison of C++ and fortran 90 for object-oriented scientific programming. *Computer Physics Communications*, 105(1):20–36, 1997.

Eckel B. *Thinking in C++*, volume 1 Introduction to Standard C++. Prentice Hall Inc., 2nd. edition, 2000.

Ewer J., Knight B., and Cowell D. Case study: an incremental approach to re-engineering a legacy fortran computational fluid dynamics code in C++. *Advances in Engineering Software*, 22(3):153–168, 1995.

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

Mangani L., Bianchini C., Andreini A., and Facchini B. Development and validation of a C++ object oriented CFD code for heat transfer analysis. In *ASME-JSME, Thermal Engineering and Summer Heat Transfer Conference*. 2007.

Marinescu D. Stl-views-1.0.3.gdb. 2008.

Matloff N. and Salzman P. *The Art of Debugging with GDB, DDD, and Eclipse*. No Starch Press, 1st edition, 2008.

OpenCFD. OpenCFD Technical report no. TR/HGW/02 (unpublished). 2005.

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

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

Weller H.G., Tabor G., Jasak H., and Fureby C. A tensorial approach to computational continuum mechanics using object-oriented techniques. *Computer in Physics*, 12(6):620–631, 1998.

A APPENDIX A: MATRIX STORAGE IN OPENFOAM®

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

$$\mathbf{A}\,x = b \tag{15}$$

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 6.

Pseudo-code 6 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.

A.1 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

B APPENDIX B: VOLFIELDS AND SURFACEFIELDS

`volFields` contains 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 7 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 that have `owner` and `neighbor`). Retrieving the values is similar to that was previously presented (Pseudo-code 7), but require some manipulation of these values to determine which values correspond to which faces of the cell, since the field values on each face for a given cell are not contiguous in the array. (see Appendix C).

B.1 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.

C 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 8.

Pseudo-code 8 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 *neighbor*.

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.

An pseudocode for the case of `surfaceField` is presented in the Pseudo-code 9.

Pseudo-code 9 Recovering field faces values

```
cellIndex = mesh.searchCellIndex(point)
for f : nFaces
  fieldFaceValue = false
  if isInternalFace(f)
    if owner[f]==cellIndex || neighbor[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
```

C.1 OpenFOAM® Files References

- ∼/OpenFOAM/OpenFOAM-<version>/src/meshTools/meshTools/meshTools.H

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