gdbOF: A Debugging Tool for OpenFOAM®

Santiago Márquez Damián[a], Juan M. Giménez[a], Norberto M. Nigro[a]

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

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 is needed concerning with classes structure, for value storage in geometric fields and also for matrices resulting from equation systems, becoming a hard task for debugging.

To help in this process a new tool, called gdbOF, attachable to gdb (GNU Debugger) is presented in this paper. It allows to analyze classes structures at debugging time. This application is implemented by gdb macros, these macros can access to code classes and also to their data in a transparent way, giving the requested information. This tool is tested for different application cases, such as the assemble 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 checked, such as: internal and boundary vector and scalar values from solution fields, fluxes in cell faces, boundary patches and boundary conditions. As additional features of this tool data dumping to file and a graphical monitoring of fields are presented.

All these capabilities give to gdbOF a wide range of use not only in academic tests but also in real problems.

Keywords: Data Structures, debugging, OpenFOAM, gdbOF

1. INTRODUCTION

OpenFOAM® is a CFD library that allows users to program solvers and tools (for pre-processing or post-processing) in a high-level specific language. This high-level language refers to the fact of writing in a notation closer to the 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., the manipulation of individual floating-points values. Thus, in order to achieve the abstraction from the low-level coding it is necessary to use another approach, so that the Object-Oriented Programming (OOP) paradigm is selected. This methodology produces code which is easier to write, to validate and to maintain compared with purely procedural techniques. OpenFOAM® is completely written is C++ language. It is less rigorously object-oriented than the others languages (such as SmallTalk or Eiffel), due to the inclusion of some characteristics that are not strictly object-based. The main add-on is operator overloading, which is essential to work with tensor, vector and scalar fields objects concepts as in the mathematical notation. In addition, it is a multiplatform language and, being based on C, is as fast as any procedural languages [1].

There are five fundamental concepts in OOP, whereby OpenFOAM® achieves its objectives: *modularization*, *abstraction*, *encapsulation*, *inheritance* and *polimorphism*[2], widely used in the code. Polymorphism is a key concept in OpenFOAM®, which is clearly demonstrated by the proliferation of virtual methods (methods that must be implemented in child classes). 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 the representation of tensor fields: in this case `geometricField` is the parent class and various tensor fields inherit from this: `scalarField` (rank 0), `vectorField` (rank 1) and `tensorField` (rank 2), each one implements the interface provided by the parent class in different ways.

In addition to these OOP features, there are other tools of the C++ language which are not strictly object-based and those are used in OpenFOAM®. They are the aforementioned operator overloading and the use of preprocessor macros. Macros allow 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 code readability [3].

As it was mentioned, using these techniques a library oriented to high-level development is generated, ensuring that the user only has to take care about the model to solve and not other details of coding [4]. On the other hand, some problems could arise in the application creation stage yielding to undesired results. There begins the code debugging work, and this includes monitoring values corresponding to variables involved in the resolution, such as, tensors, vectors and/or scalar fields defined at cell or face centers, coefficients in the system matrix, and many other examples. In addition, debugging is not ever motivated by problems, but simply for exploratory or control purposes [5].

From the side of debugging tools in GNU-Linux platforms, *gdb* (GNU-debugger) is the `defacto` standard. It includes a variety of tools for code analysis and data inspection at run-time [6] which gives a successful environment for OpenFOAM® debugging. gdb offers a powerful print command likely to inspect arrays in memory, nevertheless is can be used directly only in simple data structures like `lists` or `Fields`. Data examination get hard when viewing the desired data involving polymorphism and inheritance connected with the virtual methods used by the library. This work requires to walk through the general class tree looking for the attributes which are wanted to be inspected. Moreover, once desired attributes are found, these maybe do not directly represent the information required by the developer. In the case of the matrices generated by `fvm` methods, they store the coefficients using the LDU Addressing technique (See gdbOF User's Manual, 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.

The main objective of the *gdbOF* tool is for solving problems like those explained in the previous paragraph. This tool is implemented by gdb macros and it is based on an implementation of gdb macros for STL (Standard Library for C++) debugging [7]. These macros simplify the task of debugging the OpenFOAM® libraries, performing the work 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 an user readable format. Moreover, *gdbOF* includes the option of writing the output into a file on disk and to view it graphically. 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 inspection at debugging time.

In this work the design concept of the tools will be presented and several cases will be solved as examples of use. These problems not only emerge in an academic context 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 a non-orthogonal correction methods in purely diffusive tests; and the third is an analysis of multiphase solvers based on Volume of Fluid Method. The last examples are focused in volumetric and surface data inspection both in array and graphical format.

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
```

Nevertheless, as it was pointed out in previous section, data inspection in OpenFOAM® requires often more complex sentences. A typical example is to verify at 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), sentence presented in Example 2 is needed to observe the values on this patch, where `vSF` is a `volScalarField`.

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

[1]In-lining is an optimization which inserts a copy of the function body directly in

Example 2 View Boundary Field values.

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

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
```

There are many examples in OpenFOAM® like the previous one in which the necessity of a tool that simplifies the access to the complex class diagram can be useful. Note that in the last example it wasn't mentioned how the index of the desired patch was known. Usually OpenFOAM® user knows only the string that represents the patch, but not the index by which it is ordered in the list of patches. Here *gdbOF* simplifies the task again, providing the `ppatchlist` command which displays the list of patches with each corresponding index. Regarding to other basic gdfOF tools please refer to the User's Manual (CITA XXXXXXXXXXXXXXXXXXXXXXXXX).

---

each calling, instead of jumping to a shared routine. gdb displays in-lined functions just like non in-lined functions. To support in-lined functions in gdb, the compiler must record information about in-lining into debug information. gcc uses the dwarf 2 format to achieve this goal like several other compilers. On the other hand gdb only supports in-lined 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 in-lined function calls with earlier versions of gcc. [8]

3. Advanced Debugging

3.1. System matrix

Increasing the complexity of debugging, it can be found cases involving not only the search and dereference of some plain variables. A typical case is the dumping of the linear system, $Ax = b$, generated by the discretization of a set of differential equations which are being solved. This is stored using *LDUAddressing* technique (see **??**) which takes advantage of the sparse matrix format and saves the coefficients in an unusual way. This storing format and the necessity of accessing to individual matrix coefficients leads to trace the values one by one and to apply a decoding algorithm. There are two commands to do this task, one to dump the data as full matrices and the other to dump the data as sparse matrices.

In order to implement the necessary loops over the matrix elements, gdb provides a C-like syntax to use 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 the calculation is performed in another language through calls to the shell. Thus, 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. This is performed by the `pfvmatrixfull/pfvmatrixsparse` commands whose structure is presented in Pseudo-code 1

3.2. Mesh Search

Another group of macros are those which 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 the searching tasks using external tools. In order to circumvent this issue OpenFOAM®'s mesh methods are used to accomplish these tasks. Thus *gdbOF* includes *ad hoc* stand-alone applications to which call at debugging time to search in the mesh. Even though this way means creating a new instance of the mesh in memory, the cost in time and development is lower than that required to accomplish the search on the mesh in gdb, implementing the loops in the gdb C-like syntax, or in another language such as python. These OpenFOAM® applications are

Pseudo-code 1 Structure of gdbOF Command `pfvmatrixfull/pfvmatrixsparse`.

1. Get parameters
2. Get upper and lower arrays with gdb
3. Redirect data to an auxiliary file
4. Format the 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 **??**)
   (d) Complete with zeros
6. Format auxiliary files: python format → gdb format
7. Show output or/and save file in octave format. Add header (sparse case)

included in *gdbOF* package and they are compiled when the *gdbOF* installer is run.

Cases of searching on the 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 at 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 (via `pfindcell` command), whose centroid is nearest to it. The corresponding `volFields` command returns two indexes: the index of the cell that contains the point, and the index of the cell which has the nearest centroid. 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`.

A Pseudo-code of this tool is presented in 2, where it may be noted that it doesn't exist 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.

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

1. Get parameters
2. Call FOAM app. to make the search
   (a) Start new case
   (b) Do search (as it is explained in **??**)
   (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 back-end 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 the patch at which the cell belongs to and which is the local index of the face within the patch. With this information it is possible to obtain the field's value at that face. For more information see appendix gdbOF User's Manual 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.

In `pfindcell`, the result stored on disk 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 3 presents this implementation.

Note that the temporal gdb macro is generated on the fly and it 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

Pseudo-code 3 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) 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)

the cell is transparent to the user and it is not a problem for debugging.

3.3. Graphical debugging

Having in mind that the aim of these tools is the debugging of field manipulation software, the most powerful tool is finally presented. It consists on the spatial visualization of fields in a graphical way.

This is a widely spread concept which remind us the first efforts in graphical debugging [9]. An usual application of graphical debugging are general data structures [10, 11], and particularly linked-lists [12] and graphs [13]. Data Display Debugger [14, 15] can be cited as an useful and general tool for these purposes. Respect to the field manipulation software debugging, it requires mesh manipulation and more sophisticated data analysis tools which drives to specific implementations [16, 17].

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 for exporting (via `foamToVtk` tool) and Paraview® [18] on the fly running. The main details of this Pseudo-code 4 are the following:

Pseudo-code 4 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' `surfaceField`s.
8. Close file.
9. Call optional parameters (`.vtk` exporting and Paraview® running)

4. Tests

4.1. Scalar Transport Test

The first test consists of the unsteady advective-diffusive equation, in a two dimensional geometry with a mesh of 3 × 3 cells, which is shown in Figure 1.

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)- (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 $\phi^0 = 273[\text{K}]$ uniform along the whole domain as initial solution.

In the Finite Volume Method, each cell is discretized as it is shown in equation (5). [19]

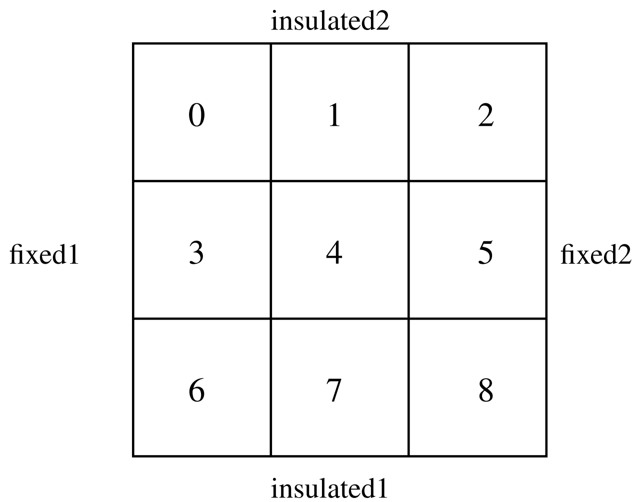

Figure 1: Geometry and patches in scalar transport test (numbers identify cells)

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

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`.

Establishing a breakpoint in the proper code line, and calling the *gdbOF* `pfvmatrixfull` command, the system matrix **A** is printed on the console. This matches the manually generated system, showing the right performance of the tool (data can be additionally saved to a file compatible with Octave and Matlab®).

Example 4 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 command and others, is the ability to export data in 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 exporting the values in the correct format, using rows, columns and values in `[row,col,coeff]` format. `pfvmatrixsparse` exports the matrix of the system in this format which has a header that identifies the file as a sparse matrix. This method greatly reduces the size needed to store the matrices in the case of medium or large meshes.

Regarding to patch commands this example is also useful to show their potentiality. Suppose that checking a boundary condition is wanted, for example the value $\phi = 373$[2] in the *fixed1* patch. First of all, it is necessary to know the index of this patch. Once the patch index is known, it is possible to see its values (See Example 5). The output is an array with three values corresponding to the boundary condition on each one of the three faces that make up this patch.

Example 5 View patches list with *gdbOF*

```
(gdb) ppatchlist T
PatchName   -->   Index to Use
FIXED1    -->   0
FIXED2    -->   1
INSULATED2    -->   2
INSULATED1    -->   3
FRONT_AND_BACK    -->   4
(gdb)
(gdb) ppatchvalues T 0
(gdb) $1 = {373,373,373}
```

Appendix B of the gdbOF User's Manual shows how the internal and boundary values (in `volFields` and in `surfaceFields`) are stored in OpenFOAM®.

[2]In the case, T is used to represent the `scalarField` instead of ϕ because OpenFOAM® preserves ϕ for a `surfaceScalarField` which represents the flux through each face ($\phi = S_f \cdot U_f$)

4.2. Laplacian Test

In this problem, *gdbOF* is used to monitor the field values and the resulting equations system, in order to realize how the correction method for non-orthogonal mesh used in OpenFOAM® works [3][19, 20].

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

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

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

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

$$\phi_{\text{fixed2}} = \phi_{\text{fixed1}} \tag{10}$$

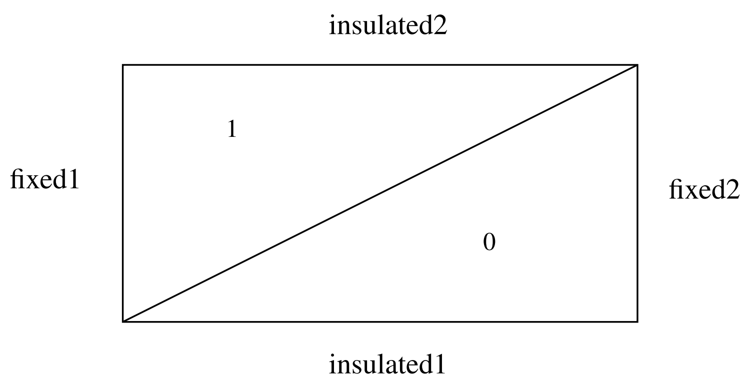

Figure 2: Geometry and patches in Laplacian test (numbers identifies cells).

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

[3]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 forms to find $\mathbf{\Delta_f}$.

Example 6 allows to verify the proper initialization of the internal field. The list shown presents the values of the field.

Example 6 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 $[\mathbf{k_f} \cdot (\nabla\phi)_f = 0]$. It allows us to compare the systems generated by the different approaches in comparison with the obtained in OpenFOAM®, and to determine which one is used as default.

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

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 3), and filled with a light fluid as initial condition and the

Example 7 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}
```

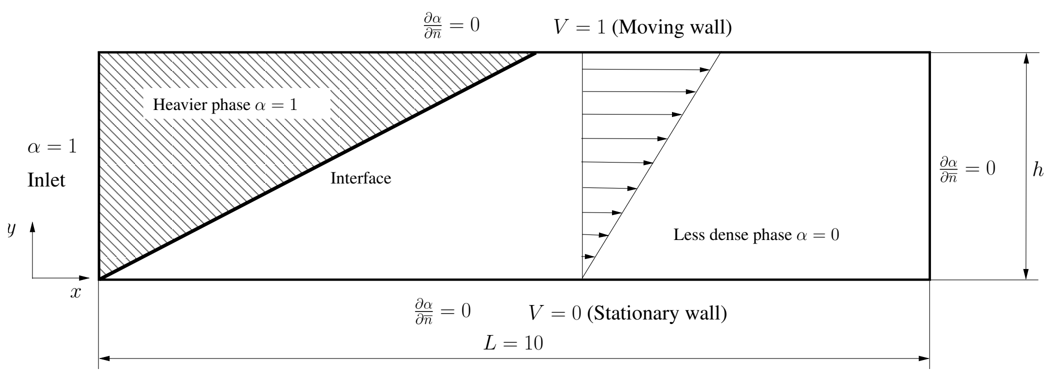

Figure 3: Geometry in `interFoam` test

domain inlet with a heavy fluid in all its extension. The problem to solve is the evolution of the heaviest phase through the domain along the time.

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) [21]

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

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

In this case, $\mathbf{g} = 0$ and it can be shown that ∇p_d and κ are both null (no pressure gradient is needed in a velocity driven flow and curvature vanishes due a linear interface). Taking this in mind, an 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 heaviest phase advances more quickly as streamlines are closer at the top region, giving a linear interface front (See Figure 3). This advancement is governed by an advective equation for the indicator function which includes an extra term, suitable to compress the interface [22].

Using Finite Volume Method Equation 12 can be discretized as in Equation 14 [23]

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

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 time-step. $\mathbf{U}_r$ is the compressive velocity and is computed 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 with δ_n as a stabilization factor [21]. ϕ_{rf} values are variable only vertically in this example and will be checked at debugging time against those calculated from theory, using *gdbOF* tools. In this case, because of how the advective terms are calculated it is necessary to show values at faces.

Domain was meshed as a 3D geometry due to OpenFOAM® requirements [24] with a $100 \times 10 \times 1$ elements in the 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 the bottom edge of the domain, $y = 0.05$, $y = 0.45$ and $y = 0.95$, values for ϕ_{rf} 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.

Again, it is necessary to find the indices of three cells with such y coordinates, taking for example $x = 0.05$, and using `pFindCell` tool the results shown in Example 8 can be obtained.

Example 8 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 using only the index of the cell is not enough to address the values in a `surfaceField` of a given field. Each cell has as many surface values as faces in the cell, therefore it is necessary to show all these values, extracting the information from faces whose indexes are 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, but it indicates which of these faces has an empty boundary condition (see gdbOF User's Manual, Appendix C or the Subsection 3.2).

So that, applying this command to the cell previously found, make it possible to show ϕ_{rf} in all faces of that cell (See Example 9). Results are consistent with the 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 faces with normals aligned with the velocity and the flux values are those predicted theoretically for $y = 0.45$. Values have different sign due to the normals orientation.

Regarding graphical debugging presented in Section 3.3 `pexportfoamformat`

Example 9 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
```

i is a useful tool to inspect the α field as in Figure 3. To do so, command is invoked as in Example 10 and results are shown in Figure 4.

Example 10 Field exporting to `.vtk` by means of `pexportfoamformat`. Paraview® is invoked as well

```
(gdb) pexportfoamformat alpha1 VTK Paraview
Including internal field...
Including boundary field(s)
fixedWall
movingWall
inlet
outlet
frontAndBackPlanes
--------------------------------
Field saved to gdbOF_dump/alpha1.0.dump
--------------------------------
Exporting to VTK...
Launching Paraview...
```

5. Conclusions

OpenFOAM® is a free software tool that enables high-level programming using a similar sintaxis as the mathematical expression that solves the problem. The use of C++ programming language and all its object-oriented

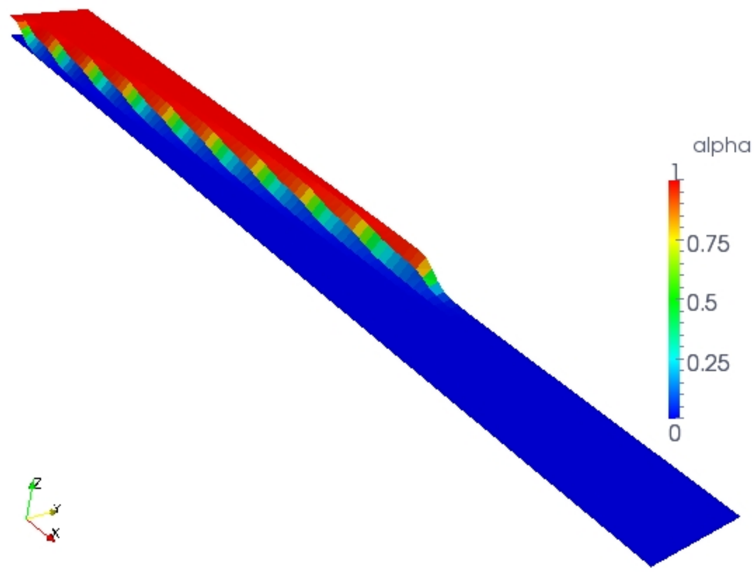

Figure 4: α field representation in Paraview® using `pexportfoamformat` (with `VTK` option)

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 tools like *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 these 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 different 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.

Finally a graphical debugging tool was presented. This goal is achieved by means of an interface between gdb an Paraview® which is built up based on more basic *gdbOF* tools. Once data are properly exported all the benefits of graphical tool can be resorted in order to analyze the results.

All of these *gdbOF* features described above, allow to the user to have greater efficiency and flexibility at debugging OpenFOAM® applications.

6. Acknowledgments

Authors want to give thanks to CONICEC, UNL (CAI+D PI 65-333) and ANPCyT (PICT 2008-1645) for their financial support and to Dr. Lisandro Dalcín for his valuable comments about this work. An special acknowledgment is given to OpenFOAM®, gdb, octave and Paraview® developers and users community for their contribution to free software.

References

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

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

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

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

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

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

[7] D. Marinescu, Stl-views-1.0.3.gdb, 2008.

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

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

[10] V. Waddle, Graph layout for displaying data structures, in: Graph Drawing, Springer, pp. 98–103.

[11] J. Korn, P. U. D. of Computer Science, Abstraction and visualization in graphical debuggers, Princeton University Princeton, NJ, USA, 1999.

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

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

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

[15] D. Cruz, P. Henriques, M. Pereira, Alma versus ddd (2008).

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

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

[18] A. Squillacote, J. Ahrens, The paraview guide, Kitware, 2006.

[19] H. Jasak, 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.

[20] H. Versteeg, W. Malalasekera, An introduction to computational fluid dynamics: the finite volume method, Prentice Hall, 2007.

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

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

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

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