

User Guide



Version 1.0

March 31, 2017

License

This document is licensed under
Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License
<http://creativecommons.org/licenses/by-nc-nd/3.0/legalcode>



Acknowledgments

The work leading to the preparation of this document has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013)/ERC Grant agreement n° 307499. The collaboration with Professor Fernando T. Pinho (University of Porto, Portugal), Professor Paulo J. Oliveira (University of Beira Interior, Portugal) and Dr Alexandre Afonso (University of Porto, Portugal) in the development of numerical methods for computational rheology is also acknowledged.

Disclaimer

This offering is not approved nor endorsed by ESI-Group, the producer of the OpenFOAM® software and owner of the OpenFOAM® trademark.

The recommendations expressed in this document are those of the authors and are not necessarily the views of, or endorsement by, third parties named in this document.

RheoTool, where this guide is included, is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY. See the GNU General Public License (<http://www.gnu.org/licenses/>) for more details.

Trademarks

Linux is a registered trademark of Linus Torvalds.

OpenFOAM is a registered trademark of ESI Group.

Paraview is a registered trademark of Kitware.

Typeset in L^AT_EX.

© 2016 Francisco Pimenta, Manuel A. Alves

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Guide organization	2
1.3	Contacts	3
2	Installation	4
2.1	Folder organization	4
2.2	Compatibility with OpenFOAM® and foam-extend versions	5
2.3	System requirements	5
2.4	Downloading Eigen library	6
2.5	Installing <i>rheoTool</i>	7
2.6	Differences between versions	7
3	Theoretical background	9
3.1	Governing equations	9
3.2	Stabilization of viscoelastic fluid flow simulations	10
3.2.1	The both-sides-diffusion technique	10
3.2.2	The log-conformation tensor approach	11
3.3	Coupling algorithms	12
3.3.1	Pressure-velocity coupling	12
3.3.2	Stress-velocity coupling	13
3.4	High-resolution schemes	14
4	Overview of <i>rheoTool</i>	16
4.1	The <i>constitutiveEquations</i> library	16
4.1.1	Available GNF and viscoelastic models	16
4.1.2	A note on FENE-type models	19
4.1.3	Multi-mode modeling	21
4.1.4	Analyzing a code sample	22
4.1.5	Advanced settings	28
4.2	The solvers	28

4.2.1	<i>rheoFoam</i>	29
4.2.2	<i>rheoTestFoam</i>	36
4.2.3	<i>rheoInterFoam</i>	38
4.3	Utilities	39
4.3.1	<i>GaussDefCmpw</i> schemes for convective terms	39
4.3.2	<i>extST</i> boundary condition	41
5	Tutorials	43
5.1	<i>rheoFoam</i>	44
5.1.1	General guidelines	44
5.1.2	A note on <i>coded FunctionObjects</i>	49
5.1.3	Case 1: flow between parallel plates	51
5.1.4	Case 2: lid-driven cavity flow	53
5.1.5	Case 3: flow in a 4:1 planar contraction	55
5.1.6	Case 4: flow around a confined cylinder	58
5.1.7	Case 5: bifurcation in a 2D cross-slot	60
5.1.8	Case 6: blood flow simulation in a real-model aneurysm	65
5.2	<i>rheoTestFoam</i>	68
5.2.1	General guidelines	68
5.2.2	Case I: Herschel-Bulkley model	71
5.2.3	Case II: FENE-CR model	72
5.3	<i>rheoInterFoam</i>	75
5.3.1	General guidelines	75
5.3.2	Case 1: impacting drop	76
	Appendix A Parameters and variables in <i>rheoTool</i>	80
	Bibliography	84

Chapter 1

Introduction

1.1 Motivation

The open-source OpenFOAM[®] toolbox can be used as a versatile finite-volume solver for CFD simulations in general polyhedral grids. A number of constitutive equations for Generalized Newtonian Fluids (GNF) are available in the toolbox since long time. More recently, Favero et al. [1] created a library containing a wide range of constitutive equations to model viscoelastic fluids, along with a solver named *viscoelasticFluidFoam* which makes use of this library. However, *viscoelasticFluidFoam* presents stability issues in certain conditions, such as, for example, in the simulation of high Weissenberg number (Wi) flows or when there is no solvent viscosity contribution (e.g. in the upper-convected Maxwell model).

In ref. [2], we attempted to minimize those issues by modifying critical points in the *viscoelasticFluidFoam* solver and in the handling of viscoelastic models. The modified solver was tested in benchmark flows and second-order accuracy, both in space and time, was observed, in addition to an enhanced stability [2]. The package that we present in this document – *rheoTool* – implements the method described in [2]. Nonetheless, *rheoTool* contains more than a solver and a library. Indeed, in addition to a robust solver for both GNF and viscoelastic fluid flows, we provide also tutorials and utilities that can be useful for the users starting to apply the OpenFOAM[®] toolbox in the simulation of complex fluid flows. In particular, some of the distinguishing features of *rheoTool* are:

- both GNF and viscoelastic models can be selected on run time and applied to single-phase laminar flows. A solver for two-phase flows is also being developed and an experimental (but fully functional) version is already available.
- the log-conformation tensor methodology [3] is available for a wide range of viscoelastic models. This minimizes the numerical instabilities frequently observed for high Weissenberg number flows.

- the transient flow solver, *rheoFoam*, uses the SIMPLEC algorithm for pressure-velocity coupling, instead of the PISO implementation in the default *viscoelasticFluidFoam*. Large time-steps can be used without decoupling problems, and the use of under-relaxation is not required (except for pressure in non-orthogonal grids).
- a stress-velocity coupling term can be selected on run time in order to avoid checkerboard fields under specific conditions, such as in the simulation of the Upper-Convected Maxwell (UCM) model in strong extensional flows.
- high-resolution schemes for convective terms are available in a component-wise and deferred correction approach, avoiding numerical instabilities (see ref. [2] for details). Additional schemes were added to the newly created library, which are not available by default in the OpenFOAM[®] toolbox.
- a solver (*rheoTestFoam*) is provided to compute the relevant material functions of each GNF/viscoelastic model included in the library. The user can select any canonical flow to be tested (shear flow, extensional flow, ...).
- the tool is provided with a user-guide (this document) and a selected set of tutorials reproducing relevant benchmark or real-life flow problems of complex fluids.
- *rheoTool* is available for both ¹OpenFOAM[®] and ²foam-extend versions.

1.2 Guide organization

The remainder of this guide is organized as follows:

- Chapter 2 describes the basic steps to install *rheoTool*.
- Chapter 3 provides a succinct overview of the theory behind the governing equations being solved. More details can be found in refs. [2] and [4].
- Chapter 4 presents an overview of the functionalities available in *rheoTool*, and discusses technical details about the code implementation.
- Chapter 5 contains several tutorials, guiding the reader into the use of *rheoTool*.

¹<http://openfoam.org/>

²<http://www.extend-project.de/>

The language and the content used in this guide assumes that the reader has a basic knowledge on the use of the OpenFOAM[®] toolbox and is familiar with the finite-volume method applied to CFD problems. Thus, it is out the scope of this document to serve as an introduction on those subjects.

Although *rheoTool* is available for different OpenFOAM[®] and foam-extend versions, Chapters 4 and 5 use OpenFOAM[®] version 2.2.2 to describe the contents. However, the small differences among different versions should not be an obstacle to the readers using any other version.

The readers interested in the theory behind *rheoTool* are strongly encouraged to first read ref. [2] before this guide.

1.3 Contacts

rheoTool will be continuously improved in the future and new features will be added. If you have any suggestions, comments or doubts regarding the tool, or if you found a bug or error, feel free to contact us:

✉ F. Pimenta: fpimenta@fe.up.pt

✉ M.A. Alves: mmalves@fe.up.pt

Chapter 2

Installation

2.1 Folder organization

The structure of *rheoTool* cloned or downloaded from the GitHub repository (<https://github.com/fppimenta/rheoTool>) is depicted in Fig. 2.1.

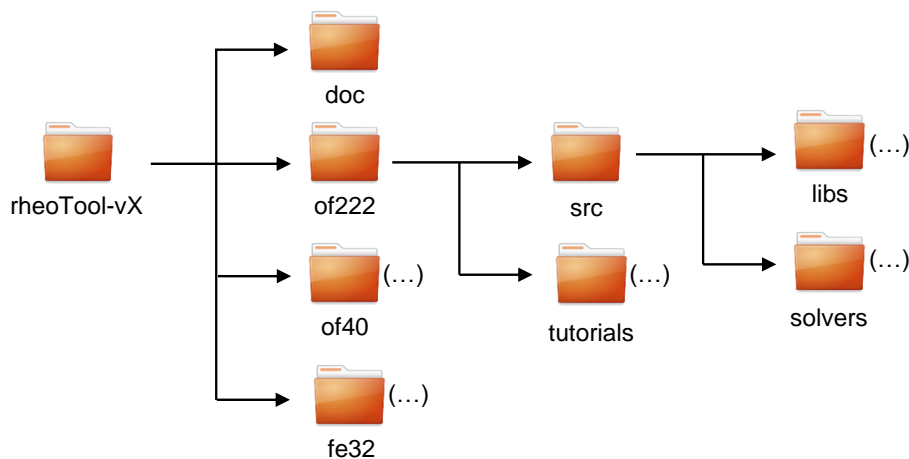


Figure 2.1: Directory organization of *rheoTool*.

The top-level directory of *rheoTool* contains the versions available for different OpenFOAM® (*of*) and foam-extend (*fe*) versions (see next section for compatibility issues). The folder `doc/`, containing the user guide, is also in the top-level directory. Inside the folder for each version, there are two directories: `src/`, where the source-code can be found, and `tutorials/`, containing several tutorial cases, showing the use of *rheoTool*. The `src/` directory is further subdivided in a directory with the applications (`solvers/`) and another one containing libraries

(`libs/`).

After cloning/downloading *rheoTool*, the user is free to remove from the top-level directory all the versions not needed and keep only the one(s) of interest.

2.2 Compatibility with OpenFOAM[®] and foam-extend versions

The development and testing of *rheoTool* was mainly performed in OpenFOAM[®] version 2.2.2. However, an effort has been made to release *rheoTool* also running under other (more recent) versions of OpenFOAM[®] and foam-extend. Thus, compatible versions of *rheoTool* are provided for:

- OpenFOAM[®] v2.2.2 (`of222/`).
- OpenFOAM[®] v4.0 (`of40/`).
- foam-extend 3.2, branch *nextRelease* (`fe32/`).

Note that the list above includes the versions which were tested. This means that a given version of *rheoTool* may be compatible with other OpenFOAM[®] or foam-extend versions not included in this list. The tested versions were run in a Ubuntu (12.04 or 16.04) environment, but other operating systems running OpenFOAM[®] can eventually support some version of *rheoTool*. However, the installation is only described here for a Linux OS. The version compatible with foam-extend 3.2 was successfully tested on commit 6de4e26 of branch *nextRelease* (a precursor of the future foam-extend 4.0 release). There is no compatibility with foam-extend 3.2 on branch *master*.

2.3 System requirements

Only standard requirements are needed to install *rheoTool*:

- a compatible and functional version of OpenFOAM[®] or foam-extend should be already installed.
- the machine should be connected to the Internet.

After ensuring that these conditions are fulfilled, the user is ready to start the installation, which has two major steps: downloading (no install) the open-source Eigen library [5] and installing *rheoTool*.

2.4 Downloading Eigen library

In the top-level directory of *rheoTool*, open a terminal and check that file `etc/bashrc` of your installed OpenFOAM® or foam-extend version has been sourced. This is particularly relevant if you have multiple *alias* variables for different versions of OpenFOAM® or foam-extend. If this is the case, be sure that the alias pointing to the desired version has been typed. Shortly, you should only advance to the next step if a command like `~$ icoFoam -help` is recognized in the terminal. Note that in this document we use the prepending `~$` for any instruction to be typed in the command line (thus, `~$ icoFoam -help` means that you only type `icoFoam -help`). If this check is successful, run the script `downloadEigen` in that terminal:

```
~$ ./downloadEigen
```

This script downloads Eigen version 3.2.9 (the most recent one at the time of writing of this guide, although older versions should also work adequately) from the Internet (using *wget*), extracts it and moves it to directory:

```
$WM_PROJECT_USER_DIR/ThirdParty/Eigen3.2.9
```

Eigen is used in *rheoTool* for computation of the eigenvalues and eigenvectors and there is no need to install the library, since the inclusion of the required headers is enough for our purposes.

However, its location in the system must be defined and exported. This is achieved by attributing to variable `EIGEN_RHEO` – the one used and recognized by *rheoTool* – the actual path of Eigen. The command to do so has been displayed to the terminal after running script `downloadEigen` (if everything was ok) and looks like:

```
~$ echo "export EIGEN_RHEO=/home/user/OpenFOAM/asus-4.0/ThirdParty/Eigen3.2.9">>/home/user/.bashrc
```

Do not copy the command above, it is just an example of what is displayed to the screen. Instead, copy-paste and run the command appearing in your terminal.

If, for some reason, the user wants to move Eigen to another directory (or already has an Eigen version in another directory), then move Eigen to its final location (if already not) and define variable `EIGEN_RHEO` accordingly. **Note that Eigen only needs to be installed once per system.** Even if the user has installed multiple versions of *rheoTool* in the same system, the above procedure only needs to be run once (for the first version being installed).

2.5 Installing *rheoTool*

While Eigen needs to be installed in a specified directory to avoid any change in the code (because an absolute path is used), the folder containing *rheoTool* can be saved and compiled in any location on your machine. Nevertheless, a location with write permission is recommended, otherwise you will need to use *sudo* mode to run all the commands. A good location for *rheoTool* is, for example, directory `$WM_PROJECT_USER_DIR`, which is defined by default when OpenFOAM® or foam-extend is installed.

After you move *rheoTool* to its final location, open a terminal in the top-level directory of *rheoTool* (ensuring that the OpenFOAM® or foam-extend environment has been sourced, as previously) and enter the directory with the version of *rheoTool* that is compatible with your OpenFOAM® or foam-extend version, and then go to directory `src/`. For example, for OpenFOAM® v2.2.2, it would be:

```
~$ cd of222/src
```

Now, run the script `Allwmake` to build the libraries and solvers of *rheoTool*:

```
~$ ./Allwmake
```

When including particular Eigen modules, some warning messages are displayed in the screen during the compilation, which are related with casting-style issues of C++ (trapped by the default flags set for *wmake*). However, none of those warnings is really problematic.

Both the libraries and solvers installed with *rheoTool* can be cleaned by running the script `Allwclean`.

Since the user will probably not need the remaining versions of *rheoTool*, which remain in the top-level directory, they can simply be deleted, if already not.

To check if the installation succeeded, the user should try to run one of the tutorials presented in Chapter 5.

2.6 Differences between versions

In order to make *rheoTool* compatible with each OpenFOAM®/foam-extend version, several modifications were required at the programming level for each case. On the other hand, the user-interface remained almost unchanged among the different versions. The main exception is on the *codedStream FunctionObjects* and *coded* boundary conditions, which are used in the tutorials of Chapter 5. Indeed, while these functionalities are available in OpenFOAM®, it is not the case for foam-extend. Thus, the *coded* boundary conditions and the utilities implemented as *codedStream FunctionObjects* in OpenFOAM® versions had to be hard-coded with the solver in foam-extend.

A second point to be taken into account is that *rheoTool* may perform differently in each OpenFOAM®/foam-extend version, as it may happen with any other solver of OpenFOAM®/foam-extend. This is naturally a consequence of the evolution of the core machinery of OpenFOAM®/foam-extend, transversal to many solvers and libraries. Fortunately, in most of the cases the differences will be small. All the discussion in this guide, including the results presented for the tutorials in Chapter 5, is for OpenFOAM® v2.2.2, as aforementioned. Taking this version as reference, the following issues were detected in the tests that we performed:

- there is some difference in the results for non-orthogonal grids, between *of222* and the other two versions tested. This issue can be observed, for example, in the tutorial of section 5.1.6, where the drag coefficient in a cylinder is computed. The difference is originated by a change which has been introduced since version 2.3.x in the computation of the cell-to-face distance at the boundaries, that is used, for example, in the *laplacian* operator. The change can be found in file `src/finiteVolume/fvMesh/fvPatches/fvPatch/fvPatch.C`, in the function `fvPatch::delta()`, where the newer versions only account for the normal component of the cell-to-face vector, thus correcting the non-orthogonality.
- a given tutorial of *rheoTool* in versions *of40* or *of222* may be run either in serial or parallel while keeping the same numerical settings. However, in the tests using version *fe32*, it was observed that parallel runs are less stable than serial runs, usually requiring a lower time-step or some under-relaxation of the velocity (sometimes as low as 0.97).

Chapter 3

Theoretical background

The equations governing the flow of incompressible complex fluids are first discussed in this chapter, followed by their discretization using the finite-volume framework. Since a thorough discussion on this subject can be found in ref. [2], some intermediate steps are skipped and only the more relevant equations are here presented.

3.1 Governing equations

The basic equations governing isothermal, single-phase, transient flows, under laminar conditions and for incompressible fluids, establish mass conservation (Eq. 3.1) and momentum conservation (Eq. 3.2),

$$\nabla \cdot \mathbf{u} = 0 \quad (3.1)$$

$$\rho \left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \right) = -\nabla p + \nabla \cdot \boldsymbol{\tau}' + \mathbf{f} \quad (3.2)$$

where \mathbf{u} is the velocity vector, t is the time, p is the pressure, $\boldsymbol{\tau}'$ is the extra-stress tensor and \mathbf{f} is any external body-force. To simulate viscoelastic fluid flows, it is a common approach to split the total extra-stress tensor in a solvent contribution ($\boldsymbol{\tau}_s$) and a polymeric contribution ($\boldsymbol{\tau}$), $\boldsymbol{\tau}' = \boldsymbol{\tau} + \boldsymbol{\tau}_s$. In order to have a closed set of equations, a constitutive equation is required for each tensor contribution, which can be generally written as in Eqs. (3.3) and (3.4), for a wide range of models.

$$\boldsymbol{\tau}_s = \eta_s(\dot{\gamma})(\nabla \mathbf{u} + \nabla \mathbf{u}^T) \quad (3.3)$$

$$f(\boldsymbol{\tau})\boldsymbol{\tau} + \lambda(\dot{\gamma}) \frac{\nabla}{\dot{\gamma}} \boldsymbol{\tau} + \mathbf{h}(\boldsymbol{\tau}) = \eta_p(\dot{\gamma})(\nabla \mathbf{u} + \nabla \mathbf{u}^T) \quad (3.4)$$

In Eqs. (3.3) and (3.4), η_s is the solvent viscosity, η_p is the polymeric viscosity coefficient, λ is the relaxation time, $\dot{\gamma}$ is the shear-rate, $f(\boldsymbol{\tau})$ is a general scalar function depending on an invariant of $\boldsymbol{\tau}$, $\mathbf{h}(\boldsymbol{\tau})$ is a tensor-valued function depending on $\boldsymbol{\tau}$ and $\overset{\nabla}{\boldsymbol{\tau}} = \frac{\partial \boldsymbol{\tau}}{\partial t} + \mathbf{u} \cdot \nabla \boldsymbol{\tau} - \boldsymbol{\tau} \cdot \nabla \mathbf{u} - \nabla \mathbf{u}^T \cdot \boldsymbol{\tau}$ represents the upper-convected time derivative, which renders the models frame-invariant. Other constitutive models exist, which can also make use of the lower-convected derivative, but those are not explored here. The constitutive equation for a GNF is limited to Eq. (3.3), since elasticity is not considered ($\boldsymbol{\tau}' = \boldsymbol{\tau}_s$). In Table 4.1 presented in the next Chapter, Eqs. (3.3) and (3.4) are specified for several GNF and viscoelastic models.

Equations (3.1)–(3.4) represent the standard system of equations to be solved. However, due to numerical stability issues, the system is rarely solved in that form. Indeed, several techniques are available for stabilization purposes (see, for instance, ref. [6] for a comparison between the most popular techniques) and the ones used in *rheoTool* are addressed next.

3.2 Stabilization of viscoelastic fluid flow simulations

3.2.1 The both-sides-diffusion technique

The both-sides-diffusion is a technique already incorporated in the *viscoelasticFluidFoam* solver [1]. It consists in adding a diffusive term on both sides of momentum equation (Eq. 3.2), with the difference that one of them (left-hand side) is added implicitly, while the other one (right-hand side) is added explicitly. Once steady-state is reached, both terms cancel each other exactly. Such method increases the ellipticity of the momentum equation and, as such, has a stabilizing effect, mostly when there is no solvent contribution in the extra-stress tensor. Incorporating the terms arising from the both-sides-diffusion in the momentum equation, and making use of Eq. (3.3), then

$$\rho \left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \right) - \nabla \cdot (\eta_s + \eta_p) \nabla \mathbf{u} = -\nabla p - \nabla \cdot (\eta_p \nabla \mathbf{u}) + \nabla \cdot \boldsymbol{\tau} + \mathbf{f} \quad (3.5)$$

Note that the added diffusive terms are scaled by the polymeric viscosity (η_p), which is a common choice in the literature (e.g. ref. [6]), although not mandatory. In order to simplify the reading, the possible dependence of the viscosity and relaxation time on the shear-rate will be dropped in the respective symbols, as already done in Eq. (3.5), although this relation still holds to keep generality.

3.2.2 The log-conformation tensor approach

The log-conformation tensor approach consists in a change of variable when evolving in time the polymeric extra-stress and it was devised to tackle the numerical instability faced at high Weissenberg number flows [3, 7].

The polymeric extra-stress tensor is related with the conformation tensor (\mathbf{A}). For the Oldroyd-B model, for example, this relation is expressed as (see Table 4.1 for other viscoelastic models)

$$\boldsymbol{\tau} = \frac{\eta_p}{\lambda}(\mathbf{A} - \mathbf{I}) \quad (3.6)$$

In the log-conformation tensor methodology, a new tensor ($\boldsymbol{\Theta}$) is defined as the natural logarithm of the conformation tensor

$$\boldsymbol{\Theta} = \ln(\mathbf{A}) = \mathbf{R} \ln(\boldsymbol{\Lambda}) \mathbf{R}^T \quad (3.7)$$

In Eq. 3.7, we can diagonalize the tensor ($\boldsymbol{\Theta}$) because it is positive definite, where \mathbf{R} is a matrix containing in its columns the eigenvectors of \mathbf{c} and $\boldsymbol{\Lambda}$ is a matrix whose diagonal elements are the respective eigenvalues resulting from the decomposition of \mathbf{A} . Equation 3.4 written in terms of ($\boldsymbol{\Theta}$) becomes [3]

$$\frac{\partial \boldsymbol{\Theta}}{\partial t} + \mathbf{u} \cdot \nabla \boldsymbol{\Theta} = \boldsymbol{\Omega} \boldsymbol{\Theta} - \boldsymbol{\Theta} \boldsymbol{\Omega} + 2\mathbf{B} + \frac{1}{\lambda} \mathbf{g}(\boldsymbol{\Theta}) \quad (3.8)$$

where $\mathbf{g}(\boldsymbol{\Theta})$ is a model-specific tensorial function depending on $\boldsymbol{\Theta}$ (see in Table 4.1 this function for several viscoelastic models) and

$$\mathbf{B} = \mathbf{R} \begin{bmatrix} m_{xx} & 0 & 0 \\ 0 & m_{yy} & 0 \\ 0 & 0 & m_{zz} \end{bmatrix} \mathbf{R}^T \quad (3.9)$$

$$\boldsymbol{\Omega} = \mathbf{R} \begin{bmatrix} 0 & \omega_{xy} & \omega_{xz} \\ -\omega_{xy} & 0 & \omega_{yz} \\ -\omega_{xz} & -\omega_{yz} & 0 \end{bmatrix} \mathbf{R}^T \quad (3.10)$$

$$\mathbf{M} = \mathbf{R} \nabla \mathbf{u}^T \mathbf{R}^T = \begin{bmatrix} m_{xx} & m_{xy} & m_{xz} \\ m_{yx} & m_{yy} & m_{yz} \\ m_{zx} & m_{zy} & m_{zz} \end{bmatrix} \quad (3.11)$$

$$\omega_{ij} = \frac{\Lambda_j m_{ij} + \Lambda_i m_{ji}}{\Lambda_j - \Lambda_i} \quad (3.12)$$

After solving Eq. (3.8), $\boldsymbol{\Theta}$ is diagonalized in the form

$$\boldsymbol{\Theta} = \mathbf{R} \boldsymbol{\Lambda}^\Theta \mathbf{R}^T \quad (3.13)$$

and the conformation tensor is recovered by the inverse relation of Eq. (3.7)

$$\mathbf{A} = \exp(\mathbf{\Theta}) = \mathbf{R} \exp(\mathbf{\Lambda}^{\Theta}) \mathbf{R}^T \quad (3.14)$$

Finally, the polymeric extra-stress tensor can be computed from \mathbf{A} (Eq. 3.6) and used in the momentum equation.

It is worth to mention that the log-conformation approach can be considered a particular case of the kernel-conformation method [8]. However, from our experience, the *log* kernel is frequently the optimal kernel (in terms of robustness and accuracy) for generic problems, so that only this one is widely used in *rheoTool*. Nevertheless, for the Oldroyd-B model, the root^k kernel [8] and the square-root transformation [9] are also included in *rheoTool* for demonstration purposes.

3.3 Coupling algorithms

3.3.1 Pressure-velocity coupling

Although the OpenFOAM[®] toolbox is already able to solve linear systems of equations in a coupled way, most of the solvers still rely on segregated solutions (this is a rule for transient solvers). In segregated solvers, the equations for each variable are solved sequentially. Even for a fully-implicit method, if the coupling between variables is weak, then numerical divergence is prone to appear.

In the OpenFOAM[®] toolbox, common algorithms for pressure-velocity coupling are SIMPLE and SIMPLEC for steady-state solvers and either PISO or PIMPLE (a combination of SIMPLE(C) and PISO) for transient solvers. From the benchmark cases performed in ref. [2], it was observed that SIMPLEC was particularly suitable for transient viscoelastic fluid flows at low Reynolds numbers, regarding stability and accuracy.

The continuity equation, implicit in the pressure variable, derived for SIMPLEC (a more detailed derivation is presented in ref. [2]) leads to

$$\nabla \cdot \left(\frac{1}{a_P - H_1} (\nabla p)_P \right) = \nabla \cdot \left[\frac{\mathbf{H}}{a_P} + \left(\frac{1}{a_P - H_1} - \frac{1}{a_P} \right) (\nabla p^*)_P \right] \quad (3.15)$$

where a_P are the diagonal coefficients from the momentum equation, $H_1 = - \sum_{nb} a_{nb}$ is an operator representing the negative sum of the off-diagonal coefficients from momentum equation, $\mathbf{H} = - \sum_{nb} a_{nb} \mathbf{u}_{nb}^* + \mathbf{b}$ is an operator containing the off-diagonal contributions, plus source terms (except the pressure gradient) of the

momentum equation and p^* is the pressure field known from the previous time-step or iteration. Accordingly, the equation to correct the velocity after obtaining the continuity-compliant pressure field from Eq. (3.15) is

$$\mathbf{u} = \frac{\mathbf{H}}{a_P} + \left(\frac{1}{a_P - H_1} - \frac{1}{a_P} \right) (\nabla p^*)_P - \frac{1}{a_P - H_1} (\nabla p)_P \quad (3.16)$$

Importantly, in order to avoid the onset of checkerboard fields, the pressure gradient terms involved in the computation of face velocities, i.e., in Eqs. (3.15) and (3.16), are directly evaluated using the pressure on the cells straddling the face, in a Rhie-Chow-like procedure (more details in ref. [2]). Nonetheless, when Eq. (3.16) is used to correct the cell-centered velocity field, the pressure gradient terms are computed "in the usual way", for example using Green-Gauss integration.

Rhie-Chow methods used to avoid checkerboard fields, as the one described in the previous paragraph, are known to be affected by the use of small time-steps and they also present time-step dependency on steady-state results [10]. In OpenFOAM® solvers, a common strategy to avoid such effects is to add a corrective term to face-interpolated velocities, through functions *ddtPhiCorr()* or *ddtCorr()*. Recently, in foam-extend the time-step dependency was solved in a different way, by removing the transient term contribution from the a_P coefficients of the momentum equation [11]. However, this approach may be problematic when used with the SIMPLEC algorithm, since a division by zero is prone to happen. In *rheoTool*, we keep using the added corrective term, although, as mentioned in ref. [2], this term can be improved in order to more efficiently avoid the small time-step dependency of steady-state solutions.

3.3.2 Stress-velocity coupling

Stress-velocity decoupling problems can arise for similar reasons as those described for pressure-velocity: the cell-centered velocity loses the influence of the forces (either polymeric extra-stress or pressure gradient) of its direct neighborhood (cells sharing a face in common). This usually happens in the interpolation from cell-centered to face-centered fields. In the case of polymeric extra-stresses, it is the divergence term $(\nabla \cdot \boldsymbol{\tau})$ in the momentum equation, when $\boldsymbol{\tau}$ is linearly interpolated from cell centers to face centers, which can be responsible for the decoupling.

In ref. [2], we described a new stress-velocity coupling method, where the polymeric extra-stresses at face centers are computed as

$$\boldsymbol{\tau}_f = \bar{\boldsymbol{\tau}}_f + \eta_p \left[(\nabla \mathbf{u})|_f + (\nabla \mathbf{u})^T|_f - \left(\overline{\nabla \mathbf{u}}|_f + \overline{(\nabla \mathbf{u})^T}|_f \right) \right] \quad (3.17)$$

where terms with an overbar are linearly interpolated from cell-centered values, while the remaining velocity gradient is directly evaluated from the cell-centered

velocities straddling the face. When the definition of $\boldsymbol{\tau}_f$ in Eq. (3.17) is inserted in the momentum equation with the both-sides-diffusion terms already present (Eq. 3.5), then we obtain

$$\rho \left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \right) - \nabla \cdot (\eta_s + \eta_p) \nabla \mathbf{u} = -\nabla p - \overline{\nabla \cdot \eta_p \nabla \mathbf{u}} + \nabla \cdot \bar{\boldsymbol{\tau}} + \mathbf{f} \quad (3.18)$$

where the term $\overline{\nabla \cdot \eta_p \nabla \mathbf{u}}$ is a "special second-order derivative" (different from the Laplacian operator of OpenFOAM[®]), defined as the divergence of the velocity gradient, where the velocity gradient at the faces is obtained by linear interpolation of the velocity gradient evaluated on the cell centers. More details are presented in ref. [2], where it is shown that with mesh refinement Eq. 3.17 approaches $\boldsymbol{\tau}_f = \bar{\boldsymbol{\tau}}_f$ and the additional terms cancel out. Note that when inserting Eq. 3.17 in the momentum equation (resulting in Eq. 3.18), we drop the transpose velocity gradients for simplicity, since continuity imposes $\nabla \cdot \nabla \mathbf{u}^T = \mathbf{0}$.

3.4 High-resolution schemes

The discretization of convective terms within the finite-volume framework leads to

$$\int_V (\mathbf{u} \cdot \nabla \phi) dV = \sum_f \phi_f (\mathbf{u}_f \cdot \mathbf{S}_f) = \sum_f \phi_f F_f \quad (3.19)$$

where ϕ is a generic variable advected, \mathbf{S}_f is the face-area vector and F_f is the volumetric flux crossing face f . While fluxes are known at the faces from the Rhie-Chow-like interpolation (Eq. 3.16), ϕ at face centers need to be interpolated from known values at cell centers. OpenFOAM[®] offers a wide range of schemes to perform such interpolation, from upwind – an unconditionally stable scheme, but only first-order accurate –, to central differences – a conditionally stable, second-order accurate scheme. A good compromise between both extremes is provided by the High-Resolution Schemes (HRSs). When represented in a Normalized Variable Diagram (NVD), several HRSs are piecewise-linear functions and can be defined using the Normalized Weighting Factor (NWF) approach [12]:

$$\tilde{\phi}_f = \alpha \tilde{\phi}_C + \beta \quad (3.20)$$

where the following definitions hold

$$\tilde{\phi}_f = \frac{\phi_f - \phi_U}{\phi_D - \phi_U} \quad (3.21a)$$

$$\tilde{\phi}_C = \frac{\phi_C - \phi_U}{\phi_D - \phi_U} \quad (3.21b)$$

In Eq. (3.20), α and β are scalars specific to each HRS and they can be functions of $\tilde{\phi}_C$. Subscripts in Eqs. (3.21a,b) have the following meaning: for a given face, cell C is the cell from which the flux comes (upstream), cell D (downstream) is the cell to which the flux goes and cell U (far-upstream) is the cell upstream to cell C. In a general unstructured mesh, cell U cannot be identified unequivocally, and ϕ_U in Eqs. (3.21a,b) can be evaluated as [13]

$$\phi_U = \phi_D - 2(\nabla\phi)_C \cdot \mathbf{d}_{CD} \quad (3.22)$$

where \mathbf{d}_{CD} is the vector connecting cell C to cell D. For a deferred correction implementation of HRSs, the upwind part of the HRS is discretized implicitly, while the remaining (difference between the HRS and the upwind differencing scheme) is discretized explicitly (cf. ref. [2]), which, using Eqs. (3.20-3.22), results in

$$\phi_f = [\phi_C]_{\text{implicit}} + [(\alpha - 1)\phi_C + \beta\phi_D + (1 - \alpha - \beta)(\phi_D - 2(\nabla\phi)_C \cdot \mathbf{d}_{CD})]_{\text{explicit}} \quad (3.23)$$

Handling the HRSs in a deferred correction approach avoids, in some cases, numerical instabilities introduced by the central-differencing component of the HRS. Additionally, in ref. [2] it was observed that the usual methodology of OpenFOAM® to apply HRSs to non-scalar variables (tensors and vectors) can locally introduce numerical instabilities in some viscoelastic flow problems. This methodology consists in using a frame-invariant quantity for non-scalar variables, such as the squared magnitude for vectors, or the trace (or double-dot product) for tensors, to compute the α and β parameters in Eq. (3.23). It was observed that such artificial instabilities can be significantly damped with a component-wise handling of non-scalar variables [2], at the cost of losing frame-invariance, which however is very weak and vanishes with grid refinement. Accordingly, non-scalar variables are split into its components and Eq. (3.23) is applied independently to each one of them. Note that this approach still generates one single matrix of coefficients for such variables, since the upwind differencing scheme coefficients are common to all the components (they only depend on the flux). The differentiation between components is only introduced in the explicit part of Eq. (3.23), generating a different source term for each individual tensor/vector component. This is possible due to the use of a deferred correction approach.

Chapter 4

Overview of *rheoTool*

In the previous chapter, the main theoretical points in *rheoTool* were briefly discussed. This chapter focus on the numerical implementation of the governing equations in the OpenFOAM® environment, providing an overview of the functionalities available in *rheoTool*. This is accomplished while presenting the three central pieces of *rheoTool*: a library containing different constitutive models, the solvers and a set of utilities particularly relevant for viscoelastic fluid flow simulations.

4.1 The *constitutiveEquations* library

4.1.1 Available GNF and viscoelastic models

The *constitutiveEquations* library is a main component of *rheoTool*, since it contains all the viscoelastic and GNF constitutive equations, which can then be called from the solvers. It was derived from the *viscoelasticTransportModels* library [1]. However, instead of restricting the library to viscoelastic models, we also extend it to include GNF models, most of them already present in OpenFOAM®. This was done in order to allow accessing both class of models from a single library, hence from a single solver.

The models available in the *constitutiveEquations* library are displayed in Table 4.1, along with the respective expressions to be used in Eqs. (3.3), (3.4), (3.6) and (3.8).

Table 4.1: Available constitutive models in the *constitutiveEquations* library.

Model	¹ TypeName	$\eta_s(\dot{\gamma})$
GNF models		
Newtonian	<i>Newtonian</i>	η
² (Bounded) Power Law	<i>powerLaw</i>	$\max(\eta_{\min}, \min(\eta_{\max}, k \dot{\gamma}^{n-1}))$
Carreau-Yasuda	<i>CarreauYasuda</i>	$\eta_{\infty} + (\eta_0 - \eta_{\infty})[1 + (k\dot{\gamma})^a]^{\frac{n-1}{a}}$
² (Bounded) Herschel-Bulkley	<i>HerschelBulkley</i>	$\min(\eta_0, \tau_0 \dot{\gamma}^{-1} + k\dot{\gamma}^{n-1})$

Model	¹ TypeName	$\eta_s(\dot{\gamma})$	$\eta_p(\dot{\gamma})$	$\lambda(\dot{\gamma})$	Constitutive Equation
Viscoelastic models solved in the standard extra-stress tensor variable					
Oldroyd-B	<i>Oldroyd-B</i>	η_s	η_p	λ	$\boldsymbol{\tau} + \lambda \overset{\nabla}{\boldsymbol{\tau}} = \eta_p(\nabla \mathbf{u} + \nabla \mathbf{u}^T)$
Giesekus	<i>Giesekus</i>	η_s	η_p	λ	$\boldsymbol{\tau} + \lambda \overset{\nabla}{\boldsymbol{\tau}} + \alpha \frac{\lambda}{\eta_p} (\boldsymbol{\tau} \cdot \boldsymbol{\tau}) = \eta_p(\nabla \mathbf{u} + \nabla \mathbf{u}^T)$
sPTT linear	<i>sPTTlinear</i>	η_s	η_p	λ	$\left[1 + \frac{\varepsilon \lambda}{\eta_p} \text{tr}(\boldsymbol{\tau})\right] \boldsymbol{\tau} + \lambda \overset{\nabla}{\boldsymbol{\tau}} = \eta_p(\nabla \mathbf{u} + \nabla \mathbf{u}^T)$
sPTT exponential	<i>sPTTexp</i>	η_s	η_p	λ	$\left[e^{\frac{\varepsilon \lambda}{\eta_p} \text{tr}(\boldsymbol{\tau})}\right] \boldsymbol{\tau} + \lambda \overset{\nabla}{\boldsymbol{\tau}} = \eta_p(\nabla \mathbf{u} + \nabla \mathbf{u}^T)$
FENE-CR	<i>FENE-CR</i>	η_s	η_p	λ	$\left[1 + \lambda \frac{D}{Dt} \left(\frac{1}{f}\right)\right] \boldsymbol{\tau} + \frac{\lambda}{f} \overset{\nabla}{\boldsymbol{\tau}} = \eta_p(\nabla \mathbf{u} + \nabla \mathbf{u}^T)$ where $f = \frac{L^2 + \frac{\lambda}{\eta_p} \text{tr}(\boldsymbol{\tau})}{L^2 - 3}$
FENE-P	<i>FENE-P</i>	η_s	η_p	λ	$\boldsymbol{\tau} + \frac{\lambda}{f} \overset{\nabla}{\boldsymbol{\tau}} = \frac{a\eta_p}{f}(\nabla \mathbf{u} + \nabla \mathbf{u}^T) - \frac{D}{Dt} \left(\frac{1}{f}\right) [\lambda \boldsymbol{\tau} + a\eta_p \mathbf{I}]$ where $f = \frac{L^2 + \frac{\lambda}{a\eta_p} \text{tr}(\boldsymbol{\tau})}{L^2 - 3}$ and $a = \frac{L^2}{L^2 - 3}$
WhiteMetzner (Carreau-Yasuda)	<i>WhiteMetznerCY</i>	η_s	$\eta_p[1 + (K\dot{\gamma})^a]^{\frac{n-1}{a}}$	$\lambda[1 + (L\dot{\gamma})^b]^{\frac{m-1}{b}}$	$\boldsymbol{\tau} + \lambda(\dot{\gamma}) \overset{\nabla}{\boldsymbol{\tau}} = \eta_p(\dot{\gamma})(\nabla \mathbf{u} + \nabla \mathbf{u}^T)$

Model	¹ TypeName	$\Theta \rightarrow \tau$	³ Constitutive Equation
⁴ Viscoelastic models solved with the log-conformation approach			
⁵ Oldroyd-B	<i>Oldroyd-BLog</i>	$\tau = \frac{\eta_p}{\lambda}(e^\Theta - \mathbf{I})$	$\frac{\partial \Theta}{\partial t} + \mathbf{u} \cdot \nabla \Theta = (\mathbf{\Omega} \Theta - \Theta \mathbf{\Omega}) + 2\mathbf{B} + \frac{1}{\lambda}(e^{-\Theta} - \mathbf{I})$
Giesekus	<i>GiesekusLog</i>	$\tau = \frac{\eta_p}{\lambda}(e^\Theta - \mathbf{I})$	$\frac{\partial \Theta}{\partial t} + \mathbf{u} \cdot \nabla \Theta = (\mathbf{\Omega} \Theta - \Theta \mathbf{\Omega}) + 2\mathbf{B} + \frac{1}{\lambda} \left[(e^{-\Theta} - \mathbf{I}) - \alpha e^\Theta (e^{-\Theta} - \mathbf{I})^2 \right]$
sPTT linear	<i>sPTTlinearLog</i>	$\tau = \frac{\eta_p}{\lambda}(e^\Theta - \mathbf{I})$	$\frac{\partial \Theta}{\partial t} + \mathbf{u} \cdot \nabla \Theta = (\mathbf{\Omega} \Theta - \Theta \mathbf{\Omega}) + 2\mathbf{B} + \frac{1}{\lambda} \{1 + \varepsilon [\text{tr}(e^\Theta) - 3]\} (e^{-\Theta} - \mathbf{I})$
sPTT exponential	<i>sPTTexpLog</i>	$\tau = \frac{\eta_p}{\lambda}(e^\Theta - \mathbf{I})$	$\frac{\partial \Theta}{\partial t} + \mathbf{u} \cdot \nabla \Theta = (\mathbf{\Omega} \Theta - \Theta \mathbf{\Omega}) + 2\mathbf{B} + \frac{1}{\lambda} e^{\varepsilon(\text{tr}(e^\Theta) - 3)} (e^{-\Theta} - \mathbf{I})$
FENE-CR	<i>FENE-CRLog</i>	$\tau = \frac{\eta_p f}{\lambda}(e^\Theta - \mathbf{I})$	$\frac{\partial \Theta}{\partial t} + \mathbf{u} \cdot \nabla \Theta = (\mathbf{\Omega} \Theta - \Theta \mathbf{\Omega}) + 2\mathbf{B} + \frac{f}{\lambda}(e^{-\Theta} - \mathbf{I})$ where $f = \frac{L^2}{L^2 - \text{tr}(e^\Theta)}$
FENE-P	<i>FENE-PLog</i>	$\tau = \frac{\eta_p}{\lambda}(f e^\Theta - a \mathbf{I})$	$\frac{\partial \Theta}{\partial t} + \mathbf{u} \cdot \nabla \Theta = (\mathbf{\Omega} \Theta - \Theta \mathbf{\Omega}) + 2\mathbf{B} + \frac{1}{\lambda}(a e^{-\Theta} - f \mathbf{I})$ where $a = \frac{L^2}{L^2 - 3}$ and $f = \frac{L^2}{L^2 - \text{tr}(e^\Theta)}$
⁶ WhiteMetzner (Carreau-Yasuda)	<i>WhiteMetznerCYLog</i>	$\tau = \frac{\eta_p}{\lambda}(e^\Theta - \mathbf{I})$	$\frac{\partial \Theta}{\partial t} + \mathbf{u} \cdot \nabla \Theta = (\mathbf{\Omega} \Theta - \Theta \mathbf{\Omega}) + 2\mathbf{B} + \frac{1}{\lambda(\dot{\gamma})}(e^{-\Theta} - \mathbf{I})$

¹ Corresponds to the name entry identifying the model in the source code.

² In the Power Law and Herschel-Bulkley models special care is taken to avoid division by zero when $\dot{\gamma}$ is zero or very small and $n - 1 < 0$. For $\dot{\gamma} < VSMALL$, the value of $\dot{\gamma} = VSMALL$ is used in the computation of the shear viscosity ($VSMALL = 10^{-300}$ for versions using double precision).

³ The following equivalences hold true: $e^\Theta = \mathbf{A} = \mathbf{R} \mathbf{\Lambda} \mathbf{R}^T$ and $e^{-\Theta} = \mathbf{A}^{-1} = \mathbf{R} \mathbf{\Lambda}^{-1} \mathbf{R}^T$.

⁴ The solvent viscosity, the polymeric viscosity coefficient and the relaxation time for the models solved in variable Θ are the same as those for the models solved in variable τ , in the previous page.

⁵ For this single model, we also included the square-root conformation approach of ref. [9] and the root^k kernel approach in ref. [8], for demonstration purposes.

⁶ The log-conformation tensor approach of the White-Metzner model **is only applicable when** $\frac{\eta_p(\dot{\gamma})}{\lambda(\dot{\gamma})} = \frac{\eta_p}{\lambda}$ **is constant**, i.e., for $K = L$, $a = b$ and $n = m$. The version based on the extra-stress tensor variable is more general and does not have this restriction.

Notes:

- $\dot{\gamma} = \sqrt{\frac{\dot{\gamma} : \dot{\gamma}}{2}}$, with $\dot{\gamma} = \nabla \mathbf{u} + \nabla \mathbf{u}^T$.
- \mathbf{I} is the identity tensor and $\frac{D}{Dt}(\phi) = \frac{\partial \phi}{\partial t} + \mathbf{u} \cdot \nabla \phi$ represents the material derivative of the generic variable ϕ .
- $\overset{\nabla}{\tau} = \frac{\partial \tau}{\partial t} + \mathbf{u} \cdot \nabla \tau - \tau \cdot \nabla \mathbf{u} - \nabla \mathbf{u}^T \cdot \tau$ represents the upper-convected derivative of tensor τ .

In a footnote of Table 4.1, the (invariant) shear-rate used to compute shear-rate dependent variables was defined as

$$\dot{\gamma} = \sqrt{\frac{\dot{\gamma} : \dot{\gamma}}{2}} = \sqrt{2\mathbf{D} : \mathbf{D}}, \text{ with } \dot{\gamma} = \nabla \mathbf{u} + \nabla \mathbf{u}^T \text{ and } \mathbf{D} = \frac{1}{2}\dot{\gamma} \quad (4.1)$$

In the code, the shear-rate is returned by function `strainRate()` as

$$\text{strainRate}() = \text{sqrt}(2.0) * \text{mag}(\text{symm}(\text{fvc}::\text{grad}(U())))$$

and it is equivalent to Eq. (4.1). Indeed,

$$\text{symm}(\text{fvc}::\text{grad}(U())) = \frac{1}{2} (\nabla \mathbf{u} + \nabla \mathbf{u}^T) = \frac{1}{2}\dot{\gamma} = \mathbf{D}$$

thus,

$$\text{sqrt}(2.0) * \text{mag}(\text{symm}(\text{fvc}::\text{grad}(U()))) = \sqrt{2} \sqrt{\frac{1}{2}\dot{\gamma} : \frac{1}{2}\dot{\gamma}} = \sqrt{\frac{\dot{\gamma} : \dot{\gamma}}{2}} = \sqrt{2\mathbf{D} : \mathbf{D}}$$

which is equal to Eq. (4.1) – the definitions of operators `symm()`, `mag()` and `:` (double contraction) can be found in the OpenFOAM® programmers' guide. Note that the invariant computed in Eq. (4.1) is actually the magnitude of the rate-of-strain tensor, which is usually called shear rate or strain rate for shear dominated or extensional dominated flows, respectively.

All the viscoelastic models can be solved in the standard extra-stress tensor $\boldsymbol{\tau}$ (Eq. 3.4) or using the log-conformation approach (Eq. 3.8). The selection is made in dictionary `constitutiveProperties`, which should be located inside the folder `constant` of the case (see more details in section 5.1.1). For the Oldroyd-B model, we provide two additional methods for demonstration purposes. One of them (TypeName: `Oldroyd-BSqrt`) consists in solving the constitutive equation using the square-root of the conformation tensor, according to ref. [9]. The second approach (TypeName: `Oldroyd-BRootk`) allows to apply a general root^k kernel, as described in ref. [8]. Both can be used in 2D or 3D simulations, as any other model in the library. Since both models are only illustrative, their implementation and theory are not described in this guide, although both can be easily understood after a close inspection of the source-code and taking as reference the literature cited for each one. Furthermore, tutorials for both methodologies are included in `rheoTool`.

4.1.2 A note on FENE-type models

The Finite Extendable Non-linear Elastic (FENE) models were originally developed based on the representation of polymer molecules by elastic dumbbells [4]. In such analysis, the end-to-end vector for each molecule is naturally related with the conformation tensor, such that the constitutive equations for this family of models is frequently written and handled as a function of the conformation tensor. The polymeric contribution to the momentum equation is then accounted for by

transforming the conformation tensor (\mathbf{A}) in the extra-stress tensor ($\boldsymbol{\tau}$), using the relations in Table 4.1 (for the models expressed in the log-conformation approach, considering that $e^{\boldsymbol{\Theta}} = \mathbf{A}$).

In order to write the constitutive equation for FENE-type models as a function of $\boldsymbol{\tau}$, some terms arise, which may compromise the numerical stability. Furthermore, the computational cost to evaluate the resulting expression is higher than for the original model. As such, some authors simplify the constitutive equation by neglecting certain terms [14]. For the FENE-CR and FENE-P models, the complete constitutive equation written as a function of \mathbf{A} and $\boldsymbol{\tau}$ and the modified formulation in $\boldsymbol{\tau}$ are:

- FENE-CR

- Complete in \mathbf{A} :

$$\lambda \overset{\nabla}{\mathbf{A}} = -f(\text{tr}(\mathbf{A}))(\mathbf{A} - \mathbf{I}), \text{ where } f(\text{tr}(\mathbf{A})) = \frac{L^2}{L^2 - \text{tr}(\mathbf{A})}$$

- Complete in $\boldsymbol{\tau}$ (see Table 4.1):

$$\left[1 + \lambda \frac{D}{Dt} \left(\frac{1}{f}\right)\right] \boldsymbol{\tau} + \frac{\lambda}{f} \overset{\nabla}{\boldsymbol{\tau}} = \eta_p (\nabla \mathbf{u} + \nabla \mathbf{u}^T), \text{ where } f = \frac{L^2 + \frac{\lambda}{\eta_p} \text{tr}(\boldsymbol{\tau})}{L^2 - 3}$$

- Modified in $\boldsymbol{\tau}$ (usually known as FENE-MCR):

$$\boldsymbol{\tau} + \frac{\lambda}{f} \overset{\nabla}{\boldsymbol{\tau}} = \eta_p (\nabla \mathbf{u} + \nabla \mathbf{u}^T), \text{ where } f = \frac{L^2 + \frac{\lambda}{\eta_p} \text{tr}(\boldsymbol{\tau})}{L^2 - 3}$$

- FENE-P

- Complete in \mathbf{A} :

$$\lambda \overset{\nabla}{\mathbf{A}} = -[f(\text{tr}(\mathbf{A}))\mathbf{A} - a\mathbf{I}], \text{ where } f(\text{tr}(\mathbf{A})) = \frac{L^2}{L^2 - \text{tr}(\mathbf{A})} \text{ and } a = \frac{L^2}{L^2 - 3}$$

- Complete in $\boldsymbol{\tau}$ (see Table 4.1):

$$\boldsymbol{\tau} + \frac{\lambda}{f} \overset{\nabla}{\boldsymbol{\tau}} = \frac{a\eta_p}{f} (\nabla \mathbf{u} + \nabla \mathbf{u}^T) - \frac{D}{Dt} \left(\frac{1}{f}\right) [\lambda \boldsymbol{\tau} + a\eta_p \mathbf{I}], \text{ where } f = \frac{L^2 + \frac{\lambda}{a\eta_p} \text{tr}(\boldsymbol{\tau})}{L^2 - 3}$$

and $a = \frac{L^2}{L^2 - 3}$

- Modified in $\boldsymbol{\tau}$:

$$\boldsymbol{\tau} + \frac{\lambda}{f} \overset{\nabla}{\boldsymbol{\tau}} = \frac{a\eta_p}{f} (\nabla \mathbf{u} + \nabla \mathbf{u}^T), \text{ where } f = \frac{L^2 + \frac{\lambda}{a\eta_p} \text{tr}(\boldsymbol{\tau})}{L^2 - 3} \text{ and } a = \frac{L^2}{L^2 - 3}$$

In *rheoTool*, all the formulations are available and can be used (see section 5.1.1 to know how to select each one). The steady material functions evaluated for canonical flows are the same for all the formulations. However, this is not true when evaluating the transient material functions: the modified formulation has a different behavior comparing with the complete ones, which are themselves similar. For a generic flow, the complete formulations, either in \mathbf{A} or $\boldsymbol{\tau}$, should provide similar results, since they are mathematically equivalent. Due to discretization errors and stability issues, this may not be true. Regarding the modified formulations, they are not expected to behave exactly as the complete ones, even in the limit of highly refined grids.

From our experience, we strongly recommend using the formulations written and solved as a function of \mathbf{A} for FENE-type models. Those are the most stable, the most accurate regarding the original theory presented in [4] and the ones for which there is direct correspondence with the models solved with the log-conformation approach, since those were derived from the constitutive equations written as a function of the conformation tensor. Note that the FENE-CR and FENE-P models available in the *viscoelasticTransportModels* library of *viscoelasticFluidFoam* [1] are expressed in the modified form presented above.

4.1.3 Multi-mode modeling

Similarly to the *viscoelasticTransportModels* library [1], the *constitutiveEquations* library also supports multi-mode modeling for viscoelastic models. In such cases, the total extra-stress tensor is the sum of the extra-stress tensor resulting from each k^{th} mode

$$\boldsymbol{\tau}' = \sum_{k=1}^N (\boldsymbol{\tau}^k + \boldsymbol{\tau}_s^k) \quad (4.2)$$

In practice, this is achieved by assembling and solving one constitutive equation for each k^{th} mode, that is, Eq. (3.3) – solvent contribution – and Eq. (3.4) or (3.8) – polymer contribution – are built and solved N times each time-step. A warning should be made at this point, since this approach is not always the most commonly observed in the literature. Indeed, $\boldsymbol{\tau}_s$ in Eq. (4.2) is commonly placed outside the summation symbol, since multiple modes are only assigned to the polymeric contribution. To achieve this in *rheoTool*, and considering the expression for $\boldsymbol{\tau}_s$ in Eq. (3.3), **the user must split the "single-solvent viscosity" by the N modes considered**, in any way, such that this "single-solvent viscosity" is recovered summing all these N values in Eq. (4.2).

4.1.4 Analyzing a code sample

For the readers still initiating their journey in OpenFOAM®, we will explore in this section the implementation of the Oldroyd-B constitutive model, solved with the log-conformation tensor approach. This example will establish the link between part of the theory described in Chapter 3 and its implementation in the source code.

The source code displayed in Listing 4.1 is taken from file `src/libs/constitutiveEquations/constitutiveEqs/Oldroyd-B/Oldroyd-BLog/Oldroyd_BLog.C`. Let's analyze the most important lines:

- lines **1-90**: this section initializes the variables used in the constitutive model. In terms of field variables, we have (lines 23-84): `tau_` ($\boldsymbol{\tau}$), `theta_` ($\boldsymbol{\Theta}$), `eigVals_` ($\boldsymbol{\Lambda}$) and `eigVecs_` (\boldsymbol{R}). All those fields must be defined by the user when starting a simulation, except `eigVals_` and `eigVecs_`, which can be defined or not. If defined (typical of a restart from a previous simulation), they are used in the first time-step; otherwise, they are both initialized as the identity tensor/matrix, corresponding to a null extra-stress tensor ($\boldsymbol{\tau}$). Afterwards, the fluid properties are read (lines 85-88) from a dictionary, along with a boolean variable, named `uTauCoupling_`. While all the fluid properties must be defined by the user, `uTauCoupling_` is assumed to be *true* if not defined. The purpose of `uTauCoupling_` is to control whether the stress-velocity coupling term (see section 3.3.2) is to be used (*true*), or not (*false*).
- lines **94-151**: this section implements the member function `correct()`, whose purpose is to update the polymeric extra-stress field, by evolving $\boldsymbol{\Theta}$ according to the constitutive equation. From line 96 to 105, variables \mathbf{M} , $\boldsymbol{\Omega}$ and \mathbf{B} , defined in Eqs. (3.9-3.11), are computed. The function `decomposeGradU()` is a member function of the base class `constitutiveEq` (find it in the file `constitutiveEq.C`), since it is used by all the models based on the log-conformation tensor approach. Then, in lines 107-140, the constitutive equation (Eq. 3.8) is built and solved, after which $\boldsymbol{\Theta}$ is diagonalized to compute its eigenvectors/eigenvalues (line 144). The function doing this task (`calcEig()`) is also a member function of the class `constitutiveEq` and the algorithm being used by default for that purpose is the QR method provided by the Eigen library [5]. Another method is also available, as discussed in section 4.1.5. Note that the eigenvalues retrieved by function (`calcEig()`) are already **exponentiated**, so that they effectively correspond to $\boldsymbol{\Lambda} = \exp(\boldsymbol{\Lambda}^\Theta)$. Finally, with the currently computed eigenvectors/eigenvalues, the polymeric extra-stress tensor ($\boldsymbol{\tau}$) is recovered from the conformation tensor (line 148), according to the relation established in Eqs. (3.6) and (3.14) (check Table 4.1 for any model), and will be used in the `divTau()` function described below.

- while in the *viscoelasticTransportModels* library [1] each model was in charge to define its own contribution to the momentum equation, i.e., the term $(\nabla \cdot \boldsymbol{\tau}')$, in the *constitutiveEquations* library there is a default definition of this term in the base class. In fact, the function *divTau()* is now defined in class *constitutiveEq* and can be found in file *constitutiveEq.C*, Listing 4.2. This function starts by distinguishing between GNF and viscoelastic models in line 7. For a GNF (lines 8-14), the extra-stress contribution is $\nabla \cdot \boldsymbol{\tau}' = \nabla \cdot \eta(\dot{\gamma}) \nabla \mathbf{u} + \nabla \mathbf{u} \cdot \nabla \eta(\dot{\gamma})$, divided by the density to be compliant with the usual strategy of OpenFOAM® for single-phase, incompressible fluid flows. Note that the second term is to account for a shear-rate dependent viscosity coefficient. By definition, a GNF fluid has no elasticity, thus $\boldsymbol{\tau} = \mathbf{0}$. For a viscoelastic fluid (lines 17-39), the output depends on *uTauCoupling*: if *false*, only the both-sides-diffusion technique is used and $\nabla \cdot \boldsymbol{\tau} = \nabla \cdot \bar{\boldsymbol{\tau}} - \nabla \cdot \eta_p \nabla \mathbf{u} + \nabla \cdot (\eta_s + \eta_p) \nabla \mathbf{u}$ (Eq. 3.5); otherwise (if *true*), the both-sides-diffusion technique is used in conjunction with the stress-velocity coupling term and $\nabla \cdot \boldsymbol{\tau} = \nabla \cdot \bar{\boldsymbol{\tau}} - \overline{\nabla \cdot \eta_p \nabla \mathbf{u}} + \nabla \cdot (\eta_s + \eta_p) \nabla \mathbf{u}$ (Eq. 3.17). We do not offer the possibility of solving viscoelastic problems without the both-sides-diffusion technique, since the method is beneficial for a wide range of problems and its effect completely disappears in steady-state solutions. Its explicit behavior effects on transient results can be minimized by performing inner iterations at each time-step, a subject discussed later on this guide (see section 4.2.1). In file *constitutiveEq.C*, a function *divTauS()* is also included, which retrieves part of the extra-stress contribution to the momentum equation, when solving two-phase flows (this topic will be discussed later).

```

1 #include "Oldroyd_BLog.H"
  #include "addToRunTimeSelectionTable.H"
3
  // * * * * * Static Data Members * * * * *
  * * * * * //
5
  namespace Foam
7 {
    defineTypeNameAndDebug(Oldroyd_BLog, 0);
9    addToRunTimeSelectionTable(constitutiveEq, Oldroyd_BLog,
      dictionary);
11 }
  // * * * * * Constructors * * * * *
  * * * * * //
13
  Foam::Oldroyd_BLog::Oldroyd_BLog
15 (

```

```

17     const word& name,
18     const volVectorField& U,
19     const surfaceScalarField& phi,
20     const dictionary& dict
21 ) :
22     constitutiveEq(name, U, phi),
23     tau_
24     (
25         IOobject
26         (
27             "tau" + name,
28             U.time().timeName(),
29             U.mesh(),
30             IOobject::MUST_READ,
31             IOobject::AUTO_WRITE
32         ),
33         U.mesh()
34     ),
35     theta_
36     (
37         IOobject
38         (
39             "theta" + name,
40             U.time().timeName(),
41             U.mesh(),
42             IOobject::MUST_READ,
43             IOobject::AUTO_WRITE
44         ),
45         U.mesh()
46     ),
47     eigVals_
48     (
49         IOobject
50         (
51             "eigVals" + name,
52             U.time().timeName(),
53             U.mesh(),
54             IOobject::READ_IF_PRESENT,
55             IOobject::AUTO_WRITE
56         ),
57         U.mesh(),
58         dimensionedTensor
59         (
60             "I",
61             dimless,
62             pTraits<tensor>::I
63         ),
64         zeroGradientFvPatchField<tensor>::typeName

```

```

65     ),
    eigVecs_
67     (
        IOobject
69         (
            "eigVecs" + name,
71            U.time().timeName(),
            U.mesh(),
73            IOobject::READ_IF_PRESENT,
            IOobject::AUTO_WRITE
75        ),
        U.mesh(),
77        dimensionedTensor
        (
79            "I",
            dimless,
81            pTraits<tensor>::I
        ),
83        zeroGradientFvPatchField<tensor>::typeName
    ),
    rho_(dict.lookup("rho")),
    etaS_(dict.lookup("etaS")),
87    etaP_(dict.lookup("etaP")),
    lambda_(dict.lookup("lambda")),
89    uTauCoupling_(dict.lookupOrDefault<Switch>("uTauCoupling",
        true))
{}
91
// * * * * * Member Functions * * * * *
// * * * * * //
93
void Foam::Oldroyd_BLog::correct()
95 {
    // Decompose grad(U).T()
97
    volTensorField L = fvc::grad(U());
99
    dimensionedScalar c1( "zero", dimensionSet(0, 0, -1, 0, 0,
        0, 0), 0.);
101    volTensorField B = c1 * eigVecs_;
    volTensorField omega = B;
103    volTensorField M = (eigVecs_.T() & L.T() & eigVecs_);

105    decomposeGradU(M, eigVals_, eigVecs_, omega, B);

107    // Solve the constitutive Eq in theta = log(c)

109    dimensionedTensor Itensor
    (

```

```

111     "Identity",
112     dimensionSet(0, 0, 0, 0, 0, 0, 0, 0),
113     tensor::I
114 );
115
116 fvSymmTensorMatrix thetaEqn
117 (
118     fvm::ddt(theta_)
119     + fvm::div(phi(), theta_)
120     ==
121     symm
122     (
123         (omega&theta_)
124         - (theta_&omega)
125         + 2.0 * B
126         + (1.0/lambda_)
127         * (
128             eigVecs_ &
129             (
130                 inv(eigVals_)
131                 - Itensor
132             )
133             & eigVecs_.T()
134         )
135     )
136 );
137
138 thetaEqn.relax();
139 thetaEqn.solve();
140
141 // Diagonalization of theta
142
143 calcEig(theta_, eigVals_, eigVecs_);
144
145 // Convert from theta to tau
146
147 tau_ = (etaP_/lambda_) * symm( (eigVecs_ & eigVals_ &
148     eigVecs_.T()) - Itensor);
149
150 tau_.correctBoundaryConditions();
151 }

```

Listing 4.1: Source code for the *Oldroyd-BLog* constitutive model (Oldroyd_BLog.C)

```

1 tmp<fvVectorMatrix> constitutiveEq::divTau
2 (
3     const volVectorField& U

```

```

) const
5 {

7     if (isGNF())
8     {
9         return
10        (
11            fvm::laplacian( eta()/rho(), U, "laplacian(eta,U
12                        )" )
13            + (fvc::grad(U) & fvc::grad(eta()/rho()))
14        );
15    }
16    else
17    {
18        if (coupledUtau())
19        {
20            return
21            (
22                fvc::div(tau()/rho(), "div(tau)")
23                - (etaP()/rho()) * fvc::div(fvc::grad(U))
24                + fvm::laplacian( (etaP() + etaS())/rho(), U,
25                        "laplacian(eta,U)" )
26            );
27        }
28        else
29        {
30            return
31            (
32                fvc::div(tau()/rho(), "div(tau)")
33                - fvc::laplacian(etaP()/rho(), U, "laplacian(
34                        eta,U)" )
35                + fvm::laplacian( (etaP()+ etaS())/rho(), U, "
36                        laplacian(eta,U)" )
37            );
38        }
39    }
}

```

Listing 4.2: Source code of the virtual function *divTau()* defined in file *constitutiveEq.C*

4.1.5 Advanced settings

As aforementioned, the eigenvectors/eigenvalues used in the models based on the log-conformation approach are computed, by default, using the QR algorithm of the Eigen library [5]. However, there is also the possibility to use an iterative Jacobi method [15]. While both options offer good accuracy and stability, the QR algorithm was seen to be slightly faster and this is the reason of being the default option. Switching between either methods is not run time selectable, but hard-coded, instead. This can be controlled in member function *calcEig()* of class *constitutiveEq*, located in file *constitutiveEq.C*. The Jacobi method can be selected by uncommenting the currently commented block (*// Eigen decomposition using the iterative jacobi algorithm*) and commenting the block (*// Eigen decomposition using a QR algorithm of Eigen library*), i.e., all the remaining lines inside the function. The source-code of *jacobi()* function is located in *utils/jacobi.H*. Note that, independently of which method is used in function *calcEig()*, this function will return the eigenvectors of the input tensor, and the **exponential** of the eigenvalues of the same tensor. After re-compiling the library with those modifications, all the log-based models will be affected by those changes.

In order to use library *constitutiveEquations* in any solver other than the ones provided with *rheoTool*, the user should:

- add the header *#include "constitutiveModel.H"* to the main solver.
- create a *constitutiveModel* object by calling the constructor, with the correct arguments, for example: *constitutiveModel constEq(U, phi)*.
- add the library *constitutiveEquations* to the *Make/options*, and specify its path (check the *Make/options* of the solvers in *rheoTool* for an example).

4.2 The solvers

Three main solvers are included in *rheoTool*:

- *rheoFoam* – a general-purpose transient solver for single-phase flows;
- *rheoInterFoam* – a two-phase solver using the VOF method to capture the interface (still under development);
- *rheoTestFoam* – a single-cell solver to test the constitutive equations in simple canonical flows.

4.2.1 *rheoFoam*

The solver *rheoFoam* implements the transient incompressible Navier-Stokes equations for single-phase flows of GNF or viscoelastic fluids. Figure 4.1 displays the solution sequence.

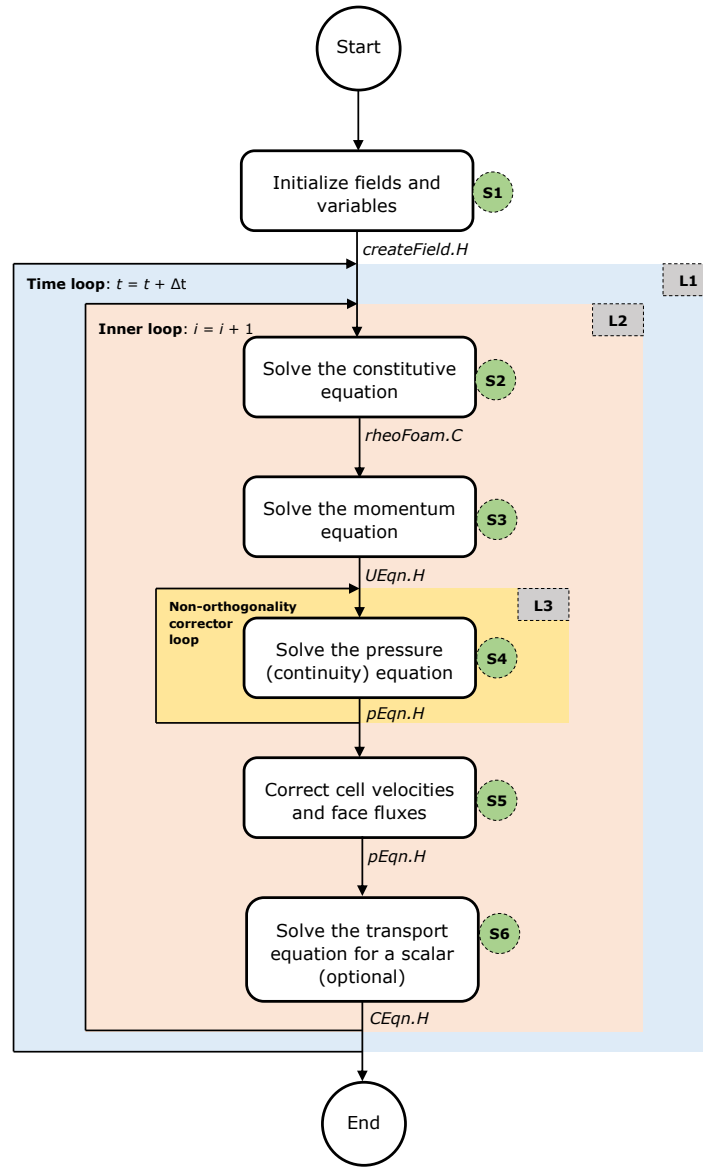


Figure 4.1: Solving sequence of *rheoFoam*.

The solver is composed of three main loops: *L1*, which is advancing the time; *L2*, which is an inner-loop used to converge the solution at each time-step; and *L3*, which can be enabled for non-orthogonal grids, in order to update (inside each time-step and each inner-iteration) the explicit correction of the pressure Laplacian, avoiding stability problems and reducing the error in transient computations. More than understanding each step identified in Fig. 4.1, we want to help the reader to identify them in the source-code and relate them with the theory presented in the previous chapter. With this purpose in mind, we will do a tour to the source-code of the solver and the most important points will be discussed, skipping the lines not essential to understand the algorithm implementation.

The solver *rheoFoam* is composed of one main file (*rheoFoam.C*) and four associated header files (*createFields.H*, *UEqn.H*, *pEqn.H*, *CEqn.H*), which can be found in directory *src/solvers/rheoFoam/*. All the header files are included from the main *.C* file. We will start by digging into *rheoFoam.C*, whose source code is displayed in Listing 4.3.

- lines **1-5**: those *# include* lines load classes used by OpenFOAM® for standard tasks, transversal to much of the OpenFOAM® solver.
- line **7**: this *# include* will allow the solver to get access to the models defined in the *constitutiveEquations* library.
- lines **13-20**: fields, variables, controls and the mesh are created (step *S1* of Fig. 4.1). Line 16 points to the header file *createFields.H*, located in the same directory as *rheoFoam.C*.
- lines **27-62**: represents the time loop, i.e., *L1* of Fig. 4.1. The solver will keep running until the final specified time is reached or once the residuals of the solved variables drop below some tolerance (this dual criterion is specific of *simpleControl* class).
- lines **31-33**: those includes allow automatic time-step adjustment, as a function of a maximum Courant number specified by the user. This control can be switched off by the user (more details in section 5.1.1).
- lines **36-55**: this is the inner loop, *L2*, of Fig. 4.1. Inside this loop, all the conservation equations are solved *nIter* times inside the same time-step. This reduces the explicitness of the method, which exists, for example, in the non-linear convective term of the momentum equation, in the both-sides-diffusion and in several terms of the constitutive equation (for a given equation, only the terms introduced through a *fvm::* operator are implicit). Furthermore, these iterations also strengthen the coupling between velocity and pressure.

- line 43: the function *correct()* of the constitutive model is called. As seen before, this function updates variable $\boldsymbol{\tau}$ by solving the constitutive equation.
- line 46: the momentum equation is solved. The header file `UEqn.H` (Listing 4.4) will be explored next.
- line 47: the pressure equation is solved. The header file `pEqn.H` (Listing 4.5) will be explored later.
- line 51-54: the equation for a passive scalar is optionally solved, depending on a user-defined selection (more details in section 5.1.1). The header file `CEqn.H` (Listing 4.6) will be explored later.

```

#include "fvCFD.H"
2 #include "IFstream.H"
#include "OFstream.H"
4 #include "simpleControl.H"
#include "fvIOoptionList.H"
6
#include "constitutiveModel.H"
8
// * * * * *
//
10
int main(int argc, char *argv[])
12 {
    #include "setRootCase.H"
14     #include "createTime.H"
    #include "createMesh.H"
16     #include "createFields.H"
    #include "createFvOptions.H"
18     #include "initContinuityErrs.H"

20     simpleControl simple(mesh);

22     // * * * * *
    //

24     Info<< "\nStarting time loop\n" << endl;

26     // --- Time loop ---
    while (simple.loop())
28     {
        Info<< "Time = " << runTime.timeName() << nl << endl;

30         #include "readTimeControls.H"
32         #include "CourantNo.H"

```

```

34     #include "setDeltaT.H"

        // --- Inner loop iterations ---
36     for (int i=0; i<nInIter; i++)
    {
38         Info<< "Iteration  " << i << nl << endl;

        // --- Pressure-velocity SIMPLEC corrector
        {
42             // ---- Solve constitutive equation ----
            constEq.correct();

44             // ---- Solve U and p ----
            #include "UEqn.H"
            #include "pEqn.H"
48         }

        // --- Passive Scalar transport
50         if (sPS)
52         {
            #include "CEqn.H"
54         }
56     }

    runTime.write();

58     Info<< "ExecutionTime = " << runTime.elapsedCpuTime()
        << " s"
60         << " ClockTime = " << runTime.elapsedClockTime()
            << " s"
            << nl << endl;
62 }

64 Info<< "End\n" << endl;

66 return 0;
}

```

Listing 4.3: Source code of rheoFoam.C.

The source-code in file `createFields.H` will not be discussed, since it mainly contains standard declaration/initialization of fields and variables. However, it is worth mentioning the creation of a *constitutiveModel* object, named *constEq*, which stores the data of the constitutive equation selected.

According to the SIMPLEC algorithm, the momentum equation is first solved to obtain an estimated velocity field, using the pressure field of the previous inner-iteration or time-step. Then, we solve for the pressure field enforcing continuity. Finally, using the correct pressure field, the previously estimated velocity is cor-

rected, both on faces and cell centers. This is what is being executed in `UEqn.H` and `pEqn.H`. We follow the discussion by analyzing the content of `UEqn.H`, whose code is displayed in Listing 4.4.

- lines **3-10**: this is where the momentum equation (either Eq. 3.5 or 3.18) is built. We can identify the transient term in line 5, the convective term in line 6, an extra momentum source term in line 8 (term \mathbf{f} in Eqs. 3.5 and 3.18) and the extra-stress divergence in line 9 ($\nabla \cdot \boldsymbol{\tau}'$), containing both the solvent and the polymeric contributions (remember the output of `divTau()` function, analyzed in section 4.1.4). The difference between Eqs. (3.5) and (3.18) resides in line 9.
- line **16**: the momentum equation is solved considering the pressure gradient contribution, where the pressure field from the last time-step or inner-iteration is used. This term was not added before, in order for the \mathbf{H} operator of the momentum equation to be free of such contribution, as discussed in section 3.3.1. This is required to avoid the onset of checkerboard fields, since a traditional Rhie-Chow interpolation is not used (c.f. ref. [2]).

```

1  // Momentum predictor
3  tmp<fvVectorMatrix> UEqn
4  (
5      fvm::ddt(U)
6      + fvm::div(phi, U)
7      ==
8      fvOptions(U)
9      + constEq.divTau(U)
10 ) ;
11
12 UEqn().relax();
13
14 fvOptions.constrain(UEqn());
15
16 solve(UEqn() == -fvc::grad(p));
17
18 fvOptions.correct(U);

```

Listing 4.4: Source code of `UEqn.H`.

After having a guessed (non-conservative) velocity field, we will see how is it used inside `pEqn.H`, Listing 4.5.

- lines **1-19**: variables required to solve the pressure equation (Eq. 3.15) are assembled. The sequence of steps can be easily understood, keeping in mind that `UEqn().A()` retrieves diagonal coefficients (a_P) and that `UEqn().H()`

and $UEqn().H1()$ stand for operators \mathbf{H} and H_1 , respectively. As previously discussed in section 3.3.1, pressure gradient terms entering the definition of face fluxes (line 18) are directly evaluated on cell faces to avoid checkerboard fields. Also, in line 9 there is the addition of the corrective term for time-step dependency, described in section 3.3.1.

- lines **24-39**: this is the non-orthogonality corrector loop ($L3$) displayed in Fig. 4.1. The goal is similar to the one of the inner loop: minimize the explicitness of the algorithm. At this point, the reader may be asking why do this loop exists if the inner loop is already there doing a similar task? To clarify that point, it should be noted that the non-orthogonality corrector loop only makes sense to exist for non-orthogonal meshes. For those meshes, the Laplacian operator in line 28 is not completely handled in an implicit way, but an explicit corrective term is added. For highly non-orthogonal meshes, this term has an important contribution and, due to being explicit, the pressure resulting from solving line 37 will not be continuity-compliant, which can afterwards introduce continuity problems and lead the simulation to diverge. For this reason, in such cases the implicitness of the Laplacian term is increased by continuously solving that equation with the updated pressure-field and the fluxes are only corrected on the last iteration of this loop (lines 35-38). Is the non-orthogonal corrector loop absolutely necessary when dealing with non-orthogonal meshes? No, as long as the simulation does not diverge and if only steady-state results are required. Otherwise, this loop should be active. However, even if no non-orthogonal corrector loops are performed, the Laplacian term should still be discretized with the corrective term to keep the accuracy.
- lines **28,33**: the pressure equation (Eq. 3.15) is assembled (line 28) and solved (line 33).
- line **37**: this is the equation which corrects the face fluxes (Eq. 3.16). One more time, pressure gradient terms are directly evaluated on cell faces: the $snGrad(p)$ term in line 18, when building $phiHbyA$, and the one coming from the pressure Laplacian on line 28, from which the $flux()$ operator is derived.
- line **47**: this is the equation which corrects the cell-centered velocity field (Eq. 3.16).

```

2   volScalarField rAU(1.0/UEqn().A());
   volVectorField HbyA("HbyA", U);
   HbyA = rAU*UEqn().H();
4
   surfaceScalarField phiHbyA

```

```

6      (
8          "phiHbyA",
          (fvc::interpolate(HbyA) & mesh.Sf())
          + fvc::ddtPhiCorr(rAU, U, phi)
10      );

12      fvOptions.relativeFlux(phiHbyA);
      adjustPhi(phiHbyA, U, p);
14

16      tmp<volScalarField> rAtU(rAU);

18      rAtU = 1.0/(1.0/rAU - UEqn().H1());
      phiHbyA += fvc::interpolate(rAtU() - rAU)*fvc::snGrad(p)*
          mesh.magSf();
      HbyA -= (rAU - rAtU())*fvc::grad(p);
20

22      UEqn.clear();

24      // Non-orthogonal pressure corrector loop
      while (simple.correctNonOrthogonal())
      {
26          fvScalarMatrix pEqn
          (
28              fvm::laplacian(rAtU(), p, "laplacian(p| (ap-H1))")
              == fvc::div(phiHbyA)
          );

30          pEqn.setReference(pRefCell, pRefValue);

32          pEqn.solve();

34          if (simple.finalNonOrthogonalIter())
          {
36              phi = phiHbyA - pEqn.flux();
38          }
40      }

42      #include "continuityErrs.H"

44      // Explicitly relax pressure for momentum corrector
      p.relax();

46      // Momentum corrector
      U = HbyA - rAtU()*fvc::grad(p);
48      U.correctBoundaryConditions();
      fvOptions.correct(U);

```

Listing 4.5: Source code of pEqn.H.

After pEqn.H is executed, both the pressure and the face fluxes are continuity-

compliant, but not the cell-centered velocity field. The conservative fluxes can now be used to solve any transport equation. In *rheoFoam*, we offer the possibility to solve a transport equation for a passive scalar. The governing equation, included in file `CEqn.H`, is simply a convection-diffusion transport equation, as can be seen in lines 5-11 of Listing 4.6.

```

1  // Transport of passive scalar

3  dimensionedScalar D_ = cttProperties.subDict("
    passiveScalarProperties").lookup("D");

5  fvScalarMatrix CEqn
  (
7      fvm::ddt(C)
    + fvm::div(phi, C)
9      ==
    fvc::laplacian(D_, C)
11 );

13 CEqn.relax();
14 CEqn.solve();
15
16 if (U.time().outputTime())
17 {
18     C.write();
19 }

```

Listing 4.6: Source code of `CEqn.H`.

4.2.2 *rheoTestFoam*

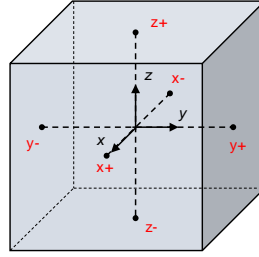
The main purpose of solver *rheoTestFoam* is to evaluate the behavior of the constitutive models for a user-defined $\nabla \mathbf{u}$ tensor. At the same time, it can also be envisaged as a basic debugging tool to check for the correct implementation of the constitutive models, since an analytical or semi-analytical solution usually exists, which can be used for comparison.

Shortly, *rheoTestFoam* solves the polymeric and solvent constitutive equations, Eqs. (3.3) and (3.4), respectively, for a prescribed $\nabla \mathbf{u}$ tensor, assuming homogeneous flow conditions ($\nabla \cdot \boldsymbol{\tau} = \mathbf{0}$; $\mathbf{u} \cdot \nabla \boldsymbol{\tau} = \mathbf{0}$). Since there are no approximations related with spatial discretization, the resulting steady-state solution $\boldsymbol{\tau}' = \boldsymbol{\tau} + \boldsymbol{\tau}_s$ is exact, and OpenFOAM® is simply acting as a nonlinear system solver. To obtain unsteady solutions, the temporal discretization introduces numerical errors during the transient period, which can be reduced using a small time-step.

The computational domain used with this solver is composed of a single cell: a cube with unitary edge length (1 m). The boundary conditions for \mathbf{u} are internally

manipulated inside the code, in order to get the tensor $\nabla \mathbf{u}$ defined by the user, Fig. 4.2. Thus, **the default mesh and boundary conditions should not be changed** by the user when working with *rheoTestFoam*. We note that the following definition holds in this guide (and in OpenFOAM[®], in general):

$$(\nabla \mathbf{u})_{ij} = \frac{\partial u_j}{\partial x_i} = \begin{bmatrix} \frac{\partial u}{\partial x} & \frac{\partial v}{\partial x} & \frac{\partial w}{\partial x} \\ \frac{\partial u}{\partial y} & \frac{\partial v}{\partial y} & \frac{\partial w}{\partial y} \\ \frac{\partial u}{\partial z} & \frac{\partial v}{\partial z} & \frac{\partial w}{\partial z} \end{bmatrix} \quad (4.3)$$



$$\begin{aligned} \mathbf{u}_{x-} &= \left(\kappa_1 - \left(\frac{\partial u_x}{\partial x} \right) \frac{\delta x}{2}, \kappa_2 - \left(\frac{\partial u_y}{\partial x} \right) \frac{\delta x}{2}, \kappa_3 - \left(\frac{\partial u_z}{\partial x} \right) \frac{\delta x}{2} \right) & \mathbf{u}_{x+} &= \left(\kappa_1 + \left(\frac{\partial u_x}{\partial x} \right) \frac{\delta x}{2}, \kappa_2 + \left(\frac{\partial u_y}{\partial x} \right) \frac{\delta x}{2}, \kappa_3 + \left(\frac{\partial u_z}{\partial x} \right) \frac{\delta x}{2} \right) \\ \mathbf{u}_{y-} &= \left(\kappa_1 - \left(\frac{\partial u_x}{\partial y} \right) \frac{\delta y}{2}, \kappa_2 - \left(\frac{\partial u_y}{\partial y} \right) \frac{\delta y}{2}, \kappa_3 - \left(\frac{\partial u_z}{\partial y} \right) \frac{\delta y}{2} \right) & \mathbf{u}_{y+} &= \left(\kappa_1 + \left(\frac{\partial u_x}{\partial y} \right) \frac{\delta y}{2}, \kappa_2 + \left(\frac{\partial u_y}{\partial y} \right) \frac{\delta y}{2}, \kappa_3 + \left(\frac{\partial u_z}{\partial y} \right) \frac{\delta y}{2} \right) \\ \mathbf{u}_{z-} &= \left(\kappa_1 - \left(\frac{\partial u_x}{\partial z} \right) \frac{\delta z}{2}, \kappa_2 - \left(\frac{\partial u_y}{\partial z} \right) \frac{\delta z}{2}, \kappa_3 - \left(\frac{\partial u_z}{\partial z} \right) \frac{\delta z}{2} \right) & \mathbf{u}_{z+} &= \left(\kappa_1 + \left(\frac{\partial u_x}{\partial z} \right) \frac{\delta z}{2}, \kappa_2 + \left(\frac{\partial u_y}{\partial z} \right) \frac{\delta z}{2}, \kappa_3 + \left(\frac{\partial u_z}{\partial z} \right) \frac{\delta z}{2} \right) \end{aligned}$$

Figure 4.2: Boundary conditions manipulation in the single-cell mesh used with *rheoTestFoam* solver. The constants currently used to represent the cell-centered velocity are $\kappa_1 = \kappa_2 = \kappa_3 = 0$, although any other values could be used. The edge length of the cubic cell is set to $\delta x = \delta y = \delta z = 1$.

Two operation modes are available with this solver:

- **ramp mode:** the user defines a list of $\nabla \mathbf{u}$ tensors and the solver will retrieve the steady solution for each entry. In this mode, the solver automatically selects the ideal time-step value to be used and the steady-state condition is also automatically detected (either the relative variation of the extra-stress magnitude drops below 10^{-8} , or after a predefined number of time-steps has been exceeded – this last condition is used to avoid infinite loops).

- **transient mode:** the user defines one single $\nabla \mathbf{u}$ tensor and the solver will return the evolution over time of the monitored variables. In this case, both the time-step and the end time are controlled by the user. This mode allows to determine the transient material functions of the constitutive model selected.

As default behavior, *rheoTestFoam* writes a file named `Report` containing all the components of the total extra-stress stress tensor, $\boldsymbol{\tau}'$ (remember that $\boldsymbol{\tau}' = \boldsymbol{\tau} + \boldsymbol{\tau}_s$ is the total extra-stress tensor, including both solvent and polymeric contributions). In ramp mode, also the status ("Converged" or "Exceed_Niter") is returned, along with the relative error. The user can compute any relevant material function from the tensor components retrieved.

Some viscoelastic models are naturally unbounded under certain flow conditions, such as the UCM and Oldroyd-B models for $Wi \geq 0.5$ in extensional flow. Care should be taken for such situations, since the solver will most likely retrieve a non-physical solution close to those limits and eventually diverge.

Note that *rheoTestFoam* is only adapted to work optimally with the models implemented in the extra-stress tensor variable, which excludes the models solved with the log-conformation approach or with the conformation tensor (FENE-type). However, the material functions for a given model are the same independently of the variable in which it is solved for, as long as all the terms are accounted for. Thus, it is possible to extract the material functions of all the models provided in *rheoTool*.

In a next release of *rheoTool*, we anticipate that the solver *rheoTestFoam* will be able to fit the available constitutive models to experimental data input by the user and return the best-fit model, along with the best-fit parameters (λ, η, \dots). This will allow to run simulations in *rheoFoam* with a numerical model reproducing properly the material functions of the real fluid.

4.2.3 *rheoInterFoam*



Section 4.2.3 is under development.

The solver *rheoInterFoam* is a generalization of *rheoFoam* for two-phase flows, using the Volume of Fluid (VOF) method of OpenFOAM® to represent the interface between the two phases.

Currently, *rheoInterFoam* is solving a constitutive equation for each phase and the extra-stress tensor contributing to the momentum equation is the weighted average of the extra-stress tensor for each phase, where the weighing is ensured by the indicator function used in VOF. This approach allows to have phases represented by different constitutive equations, although it is possibly not the most accurate and stable way to do the computations. Additionally, the SIMPLEC algorithm used for single-phase flows may need to be improved when used in *rheoInterFoam*. Other aspects are still under test, which makes the current version of *rheoInterFoam* still experimental. Nonetheless, this version is fully functional.

4.3 Utilities

4.3.1 *GaussDefCmpw* schemes for convective terms

The component-wise and deferred handling of HRSs, described in section 3.4, is included as a library in *rheoTool*. If the installation procedure presented in Chapter 2 has been followed, this new class of schemes will only be available when using the family of solvers provided with *rheoTool*. To make the schemes available to any solver of OpenFOAM®, there are several ways. One option (not requiring compilation) is to include this library (`libgaussDefCmpwConvectionSchemes.so`) as a *lib* entry of `controlDict` in the case directory. Another option is to compile the class inside library *finiteVolume*, which is included by most OpenFOAM® solvers. To access this class from a specific solver, `lgaussDefCmpwConvectionSchemes` should be added to the `options` file of that solver. We note that the component-wise and deferred handling of HRSs improved significantly the stability of viscoelastic fluid flow simulations [2], but its performance and advantage when used in other type of flows need to be tested (by no way we argue that this is a magic bullet for all purposes).

The new group of HRSs is accessible from class *GaussDefCmpw* and its use is similar to the standard HRSs of OpenFOAM®. For example, the CUBISTA scheme can be used by simply defining in dictionary `fvSchemes`: *GaussDefCmpw cubista*; in front of the divergence term being discretized (remember that keywords in OpenFOAM® are case-sensitive). To obtain a list of all the schemes available, simply type *GaussDefCmpw*; without any further argument and you will obtain all the possibilities, listed in Table 4.2. There is a scheme named *none*, which corresponds to remove the convective term from the equation being discretized. Note that all the limiters implemented in class *GaussDefCmpw* are totally independent from the already existing limiters of OpenFOAM® and all are included in the file `gaussDefCmpwConvectionScheme.C`. For example, you will have now *GaussDefCmpw minmod* and *Gauss Minmod*, which are two different schemes, or,

actually, two different implementations of the same scheme.

Table 4.2: Available High-Resolution schemes for convective terms in class *gauss-DefCmpw*. The schemes are defined using the NWF approach (Eq. 3.20).

Scheme	¹ TypeName	² Equation
Upwind	<i>upwind</i>	$[\alpha, \beta] = [1, 0]$
CUBISTA	<i>cubista</i>	$[\alpha, \beta] = \begin{cases} [1, 0] & \tilde{\phi}_C \leq 0 \vee \tilde{\phi}_C \geq 1 \\ [7/4, 0] & 0 < \tilde{\phi}_C < 3/8 \\ [3/4, 3/8] & 3/8 \leq \tilde{\phi}_C \leq 3/4 \\ [1/4, 3/4] & 3/4 < \tilde{\phi}_C < 1 \end{cases}$
MINMOD	<i>minmod</i>	$[\alpha, \beta] = \begin{cases} [1, 0] & \tilde{\phi}_C \leq 0 \vee \tilde{\phi}_C \geq 1 \\ [3/2, 0] & 0 < \tilde{\phi}_C < 1/2 \\ [1/2, 1/2] & 1/2 \leq \tilde{\phi}_C < 1 \end{cases}$
SMART	<i>smart</i>	$[\alpha, \beta] = \begin{cases} [1, 0] & \tilde{\phi}_C \leq 0 \vee \tilde{\phi}_C \geq 1 \\ [3, 0] & 0 < \tilde{\phi}_C < 1/6 \\ [3/4, 3/8] & 1/6 \leq \tilde{\phi}_C \leq 5/6 \\ [0, 1] & 5/6 < \tilde{\phi}_C < 1 \end{cases}$
WACEB	<i>waceb</i>	$[\alpha, \beta] = \begin{cases} [1, 0] & \tilde{\phi}_C \leq 0 \vee \tilde{\phi}_C \geq 1 \\ [2, 0] & 0 < \tilde{\phi}_C < 3/10 \\ [3/4, 3/8] & 3/10 \leq \tilde{\phi}_C \leq 5/6 \\ [0, 1] & 5/6 < \tilde{\phi}_C < 1 \end{cases}$
SUPERBEE	<i>superbee</i>	$[\alpha, \beta] = \begin{cases} [1, 0] & \tilde{\phi}_C \leq 0 \vee \tilde{\phi}_C \geq 1 \\ [1/2, 1/2] & 0 < \tilde{\phi}_C < 1/2 \\ [3/2, 0] & 1/2 \leq \tilde{\phi}_C \leq 2/3 \\ [0, 1] & 2/3 < \tilde{\phi}_C < 1 \end{cases}$
³ no convection	none	—

¹ Corresponds to the name entry identifying the scheme in the source code.

² See Eq. 3.20.

³ When this option is used, the convective term is deleted.

Details on the implementation of this class of schemes will not be presented in this guide, although the interested reader will easily find the analogy between the equations presented in section 3.4 and the source code in file `gaussDefCmpwConvectionScheme.C`. Nevertheless, for documentation purposes, we summarize

next the operations being executed by each member function of class *GaussDefCmpw*:

- *phifDefC()*: depending on the boolean value of *onlyDCphi*, this function returns either the interpolated variable on the faces – Eq. 3.23, with all the terms explicitly evaluated –, or the deferred correction to the upwind scheme – only the explicit term of Eq. 3.23. An equivalent function (*phifDefCs()*) exists for scalar variables.
- *lims()*: this function retrieves three variables: *alpha*, is a list containing α for each interval of the function defined in Eq. (3.20); *beta* is a list containing β (Eq. 3.20) for each interval of the function defined in Eq. (3.20); *bounds* is a list of $\tilde{\phi}_C$ values, for which there is a change of branch in the function defined in Eq. (3.20). Thus, function *lims()* defines Eq. (3.20) for the selected scheme.
- *fvmDiv()*: this function also exists for *Gauss* schemes and returns the matrix of coefficients and the source term resulting from the discretization of the implicit convective operator *fvm::div()*. The major difference regarding *Gauss* schemes is that we specialize this function for scalar fields (although the template could be made general for all types of fields and *phifDefCs()* would be avoided). Function *phifDefC()* is called from here, whenever the selected scheme is different from *upwind* or *none*. Note that both *Gauss* and *GaussDefCmpw* classes implement the *upwind* scheme in the same way – it is the only scheme for which this happens.
- *interpolate()*: returns face-interpolated values, by simply calling *phifDefC()*, with the adequate boolean value.
- *flux()*: returns the field interpolated on face centers multiplied by the flux on each face (*phi*).

The class *GaussDefCmpw* easily allows modifying or adding a new piecewise-linear HRS, by simply adding a new case or modifying an existing one in function *lims()*. It is also possible to include HRSs not defined as piecewise-linear functions, although this also requires modifying function *phifDefC()*.

4.3.2 *extST* boundary condition

The *extST* boundary condition is a *fixedValue*-derived boundary condition, which linearly extrapolates each tensor component from boundary cells to boundary faces. Shortly, this boundary condition starts by computing the gradient of each

tensor component, at the center of boundary cells (using the previous iteration/time-step known values on the boundary face). Then, with both the value and the gradient of each tensor component at those locations, the tensor components at the boundary faces are linearly extrapolated (see details on ref. [2]). The discretization scheme to compute the gradients enrolled in the process is run time selectable and can be controlled in the *fvSchemes* dictionary, through the entry *extSTGrad* followed by the desired scheme, in the *gradSchemes* subDict. Using linear extrapolation for the polymeric extra-stress tensor on walls is highly recommended, instead of a zero-gradient boundary condition, which has a lower order of accuracy (see e.g. ref. [2]).

The name (type) of this boundary condition for run time selection is *extST* and it can be applied with any type of mesh, but only for symmetric tensors, i.e., $\boldsymbol{\tau}$ or $\boldsymbol{\Theta}$ (generalizing the source code for non-symmetric tensors is straightforward). As with *GaussDefCmpw* schemes, this boundary condition is only accessible from the solvers packed with *rheoTool* and the methods to get full access to this utility – compiled in library *libBCRheoTool.so* – are also similar.

Chapter 5

Tutorials

In this chapter, we provide a step-by-step guide on how to use the solvers provided with *rheoTool*. For each solver, general guidelines are first discussed, regarding the new fields and dictionaries required by those applications. Then, specific tutorials are presented, which will illustrate the application of *rheoTool* to relevant problems. These tutorials cover the full process to obtain results: from the mesh generation to the post-processing stage.

The approach used in this chapter assumes that the reader is familiar with the folder organization of OpenFOAM® cases and has basic knowledge on how to run simulations in OpenFOAM®.

⚠ Some of the tutorials included in *rheoTool*, and described in this chapter, make use of either *coded FunctionObjects* or *codedFixedValue* functionalities of OpenFOAM®, which allow to use run time compilable code from within the case directory. In some (older) versions of OpenFOAM® (e.g. v2.2.2), those features are not allowed to be executed by default and an error reporting this problem is retrieved. The instructions to change the default behavior are displayed in the error description printed in the screen. Shortly, the user should switch on variable *allowSystemOperations* in file `etc/controlDict` located in the OpenFOAM® installation directory.

i The tutorials in this chapter are mainly intended for learning purposes. It is not our primary goal to achieve highly accurate results with such examples, but solely to show how to run the solvers, preferably using fast-running cases. Higher accuracy can be obtained in all the cases by increasing the resolution in space and time.

5.1 *rheoFoam*

5.1.1 General guidelines

Before proceeding, we note that the sequence of operations required to prepare a case in OpenFOAM® do not need to be ordered as presented next (`constant/` \rightarrow `0/` \rightarrow `system/`). This sequence was selected in such a way to be (hopefully) logic and easy to follow and execute by any OpenFOAM® user.

`constant/`

Inside folder `constant/` there are two main components of the simulation: the mesh, in folder `polyMesh/`, and the dictionary `constitutiveProperties`, which is a dictionary specific of *rheoTool*. Since the mesh is an element required by almost all OpenFOAM® solvers, it will not be discussed here and we assume that a valid mesh already exists in folder `polyMesh/`.

The dictionary `constitutiveProperties` used with *rheoFoam* holds information on the constitutive model and on the passive scalar transport which can optionally be activated in the simulation (Listing 5.1).

```

1 parameters
2 {
3     type                Oldroyd-BLog;
4
5     rho                 rho [1 -3 0 0 0 0 0] 1.;
6     etaS                etaS [1 -1 -1 0 0 0 0] 0.01;
7     etaP                etaP [1 -1 -1 0 0 0 0] 0.99;
8     lambda              lambda [0 0 1 0 0 0 0] 1.;
9
10    uTauCoupling        on;
11 }
12
13 passiveScalarProperties
14 {
15     solvePassiveScalar  off;
16     D                   D [ 0 2 -1 0 0 0 0 ] 1e
17                          -9;
18 }

```

Listing 5.1: Example of a `constitutiveProperties` dictionary used with *rheoFoam*.

The dictionary `constitutiveProperties` has two different sub-dictionaries (subDict), which must necessarily exist: *parameters*, with the information of the

constitutive model, and *passiveScalarProperties*, related with the scalar-transport equation.

Regarding subDict *parameters*, in line 3 we define the *TypeName* of the constitutive model to be used, that can be found in Table 4.1. In the example displayed in Listing 5.1, we are using the Oldroyd-B model, solved with the log-conformation approach (*Oldroyd-B + Log*). If we would like to use the same model without solving it with the log-conformation approach (i.e. solving the constitutive equation for the extra-stress tensor), then the type would be simply *Oldroyd-B* – this naming rule is valid for all the viscoelastic models. Lines 5 to 8 specify the fluid properties required by the constitutive model being solved. The density is a property common to all models (it is not related with the constitutive equation) and should always be present, while the model-dependent properties can be checked in Table 4.1. Anyway, if some required parameter is not specified, the solver will retrieve an error complaining for its absence.

The reader might be surprised with the unphysical parameters displayed in Listing 5.1 (even just being an example), particularly the density of the fluid, which is not realistic for any known viscoelastic liquid. The use of a unitary density ($\rho = 1 \text{ kg/m}^3$) and many other unitary variables is simply to facilitate the calculations. In these tutorials, we frequently make use of this kind of approach, since the computation of dimensionless parameters does not require physically realistic quantities.

At line 10, the user can control the inclusion (*on*, *true* or *yes*) or not (*off*, *false* or *no*) of the stress-velocity coupling term, discussed in sections (3.3.2) and (4.1.4). Note that this option is only valid for viscoelastic models, since no such coupling term is available for GNF models.

For multi-mode viscoelastic models, the *TypeName* is *multimode* and lines 3-10 need to be included for each mode, enclosed in a dictionary identified with the name of the mode. A case example is provided in the tutorials directory (`tutorials/rheoFoam/OtherTests/`).

For the FENE-type models not using the log-conformation approach, several formulations are available, as discussed in section 4.1.2. The selection between them is also performed in subDict *parameters*. If nothing is specified, FENE models are evaluated using the (complete) formulation in **A**. In order to solve the complete formulation in **τ** , the keyword *solveInTau* should be defined and set to *true*. If the modified formulation in **τ** is intended, both *solveInTau* and *modified-Form* keywords should be defined and set to *true*. A case example is provided in the tutorials directory (`tutorials/rheoFoam/OtherTests/`). Note that the selection between formulations is not achieved by specifying different *TypeNames* for each one: the *TypeName* is the same for all the formulations within the same model (FENE-CR or FENE-P) and the selection is based on the previously-

mentioned keywords.

Focusing now on subDict *passiveScalarProperties*, only two entries are present. In line 15, the user can decide to solve (*on*, *true* or *yes*) or not (*off*, *false* or *no*) the transport equation of a passive scalar. If the equation is solved, then line 16 should specify the diffusion coefficient and field *C* (the name of the scalar being transported) should be defined in the folder corresponding to the start-time. Otherwise, none of these two actions is required. Importantly, if the option to solve the transport equation is enabled, but field *C* is not defined nor initialized, then *rheoFoam* will solve a transport equation (you can confirm it on the solver output) for a scalar not present on the domain (its concentration will remain null during all the simulation time), since this is how the field is internally initialized when there is no entry for it in the start-time folder.

0/

At this point, both the mesh and the fluid are defined and some decisions have been made about the numerical method. It is now time to create and define the internal field and boundary conditions for the variables used in the simulation, which will depend on the constitutive equation selected. At least three scenarios are possible:

- **GNF fluid:** those cases only require defining pressure, p (in this guide represented by p), and velocity, U (in this guide represented by \mathbf{u}) fields. For all the GNF models, except for the Newtonian fluid, the solver will automatically write the shear-rate dependent viscosity at subsequent times.
- **viscoelastic model using the standard extra-stress approach:** those cases require defining pressure, p , velocity, U , and the polymeric extra-stress field, τ (in this guide represented by $\boldsymbol{\tau}$). The novelty relative to the GNF cases is on variable τ , which is of type *symmTensor*. All the three variables will be automatically written at future times. When a multi-mode viscoelastic model is used, each mode owns a variable τ , which should be present in folder 0/. The name given to each variable should be consistent with the names attributed to each mode in *constitutiveProperties*, i.e., this name should be appended at the end of name τ . For example, having defined mode names *M1* and *M2*, then the names for the respective τ should be $\tau M1$ and $\tau M2$.
- **viscoelastic model using the log-conformation approach:** comparing with the previous case, it requires defining the additional variable θ , which represents the natural logarithm of the conformation tensor (in this guide represented by $\boldsymbol{\Theta}$), which is also a *symmTensor*. In order to define boundary conditions for θ , we suggest the reader to take a look at Eqs.

(3.6) and (3.7). For example, if the polymeric extra-stress (τ) is a null tensor, then variable θ is also a null tensor. At subsequent times, the solver will automatically write fields p , U , τ , θ and both the eigenvectors, $eigVecs$ (in this guide represented by \mathbf{R}), and eigenvalues, $eigVals$ (in this guide represented by $\mathbf{\Lambda}$), which are obtained from the diagonalization of the conformation tensor. Note that the fields $eigVecs$ and $eigVals$ do not need to be present to start a simulation, although they are read if they are present (for example, to restart a simulation from the exact point where it finished). For a multi-mode case, the same considerations previously described are applied, inclusively for variable θ .

When using FENE-type models solved in the conformation tensor, without the logarithm transformation (see section 4.1.2), the conformation tensor field (\mathbf{A}) can be optionally defined in folder 0/, being read by the solver in that case. However, if not defined, the solver automatically initializes the conformation tensor field from τ (that should always be present). Independently of being or not present in the folder corresponding to the initial time, field \mathbf{A} will be written to the case directory for the remaining of the simulation.

For any of the previous cases, if the option to solve the transport equation of a passive scalar has been enabled, then the field C should also be present in folder 0/. The utility *setFields* of OpenFOAM® can be particularly helpful to initialize this field, since it allows to assign different values of C in different regions of the domain (the Cross-slot tutorial in this Chapter exemplifies such situation).

system/

The last steps before starting the simulation are related to the dictionaries located in folder *system/*, which mainly control the numerical method. In particular, we will focus our attention on the following dictionaries: *controlDict*, *fvSchemes* and *fvSolution*. All the three dictionaries must be present for the simulation to run, as required by most of the OpenFOAM® solvers. Since most of the entries in those dictionaries are transversal to both *rheoFoam* and any OpenFOAM® solver, we will limit our description to the new features introduced by *rheoTool*.

In *controlDict* dictionary, the options allowing to automatically control the time-step by imposing a Courant number limit are available in *rheoFoam* and can be used (following the same principles of other OpenFOAM® solvers). Those options are *adjustTimeStep* (*on/off*), *maxCo* (the value of the limiting Courant number) and *maxDeltaT* (the maximum admissible time-step). Furthermore, and although not being a feature exclusive of *rheoFoam*, *coded functionObjects* can be defined in *controlDict* and used with *rheoFoam* to extract and monitor quantities of interest (this is not possible when using foam-extend versions; this

issue is discussed in the next section). This kind of functions are frequently used in the tutorials of this Chapter.

Regarding dictionary `fvSchemes`, we remember that *GaussDefCmpw* schemes (section 4.3.1) are available for selection and can be used to discretize any convective term of type: *div(phi,variable)*, where *variable* is either of *U*, *tau*, *theta* or *C*. Still in the *divSchemes* subDict, the term *div(grad(U))* is part of the stress-velocity coupling algorithm (see line 23 of Listing 4.2) and should (always) be discretized using a central differencing scheme (*Gauss linear*), if used. In the *gradSchemes* subDict, the entry *extSTGrad* is for the gradient of the tensor components when using linear extrapolation of polymeric extra-stress at a given boundary, as discussed in section 4.3.2. Apart from this, the remaining entries in `fvSchemes` should be familiar to the user and the selection of appropriate discretization schemes for each one is essential to keep the numerical method accurate and stable.

The dictionary `fvSolution` is the last one remaining to be adjusted before running the simulation. In subDict *solvers*, the matrix solver for each equation being solved should be specified (remember that there will be *N* equations to solve for *theta/tau* in a model using *N* modes; wildcard characters are useful in those cases). If the user forgets to specify any, the solver will retrieve an error message asking for it. Only the pressure equation results in a symmetric matrix of coefficients, while all the others generate non-symmetric matrices. The only exception is the momentum equation without the convective term included, which also results in a symmetric matrix. This should be taken into account when selecting the type of matrix solver, since some are specific for some type of matrices. In the *SIMPLE* subDict, there is a new entry specific of *rheoFoam*, which is *nInIter*. This variable was defined in section 4.2.1 and controls the number of inner iterations (see Fig. 4.1). If not defined, the solver will execute 1 inner iteration as the default behavior. Still in the *SIMPLE* subDict, the entry *residualControl* allows the solver to automatically stop the simulation once the residuals for all the specified variables drop below the prescribed value. This is mainly a characteristic of steady-state solvers of OpenFOAM® and it can be also used in *rheoFoam*. However, if the goal is to run *rheoFoam* until the *endTime* specified in `controlDict`, simply leave this entry empty. The last subDict in `fvSolution` is the *relaxationFactors*, that determines the amount of explicit (*fields*) and implicit (*equations*) under-relaxation for each field or equation. When using the SIMPLEC algorithm for pressure-velocity coupling, as a rule of thumb, the pressure does not need to be explicitly under-relaxed to correct the velocity (see ref. [2]). The only exception occurs for non-orthogonal grids, where a small amount of under-relaxation is needed for pressure. The under-relaxation factor, ranging between 1 (no under-relaxation) and 0 (total under-relaxation, pressure does not evolve in time), is case-specific and should be as high as possible (it will be conditioned by

stability issues). Regarding implicit under-relaxation (*equations*), it only makes sense to be used with steady-state solvers, where the absence of a time-derivative term requires under-relaxation for stability reasons. Since *rheoFoam* is by default a transient solver, where time-derivatives are present in all the transport equations, implicit under-relaxation is not needed and the under-relaxation factors should be kept at 1 for all variables. In practice, it is possible to run *rheoFoam* without those time-derivatives by selecting a default steady-state discretization scheme in the *ddtSchemes* subDict of *fvSchemes* and, by this way, *rheoFoam* will run as a typical steady-state solver of OpenFOAM[®], requiring implicit under-relaxation. However, in some situations (viscoelastic models solved using the log-conformation approach) the user will probably face stability issues, due to the poor diagonal dominance of the base matrix of coefficients. For this reason, unless the user is experienced and knows what is doing, we strongly recommend to use *rheoFoam* in transient mode. Note that, if it exists, the steady-state will be reached after some time of a transient simulation (typically after several relaxation times, of the order of 10 or above for higher Wi). If the transient analysis is not important, a high Courant number ($\gg 1$) can be defined to reach this state faster, as long as the computations remain stable.

In all the tutorials presented next, a steady-state is reached for the range of parameters used in the examples. However, in none of them we use the residuals as the termination criteria. Indeed, we prefer to set a very long *endTime* and monitor a relevant variable at sensitive points over time. It can be the extra-stress near a singular point, the drag coefficient over a patch, a vortex length, or any other relevant quantity for the problem at hand. This is usually achieved either through a *probe* (when the variable exists within the solver) or a *coded FunctionObject* (when the variable does not exist within the solver and needs to be computed), in dictionary *controlDict*. The termination criteria is then based on this variable. Of course, we set the *endTime* in the tutorials based on that. The residuals displayed to the screen should not be used alone as the termination criteria.

5.1.2 A note on *coded FunctionObjects*

Most of the tutorials presented next use a *coded FunctionObject*. This is a run time compilable code, executing run time functions coded by the user, which can be defined in dictionary *controlDict*.

These functions allow to access almost all the data of the case, from field variables to information about the mesh. The frequency at which they are evaluated can be controlled using the keywords *outputControl* and *outputInterval*. It is also possible to disable these functions by setting the keyword *enabled* to *off*.

The *coded FunctionObject* included in the tutorials are usually divided in three

sections: a section which reads data, a section which computes quantities of interest from this data and a write section. Usually, we create a dynamic list to accommodate the data to be written, so that any extra quantity can be added to the list. This also eases the writing step. The meaning of each column of the data being written is usually displayed as a comment in the *coded FunctionObject*, being of course dependent on the tutorial case.


Among the several possibilities to write the variables computed by *coded FunctionObjects*, we chose to use a *.sh* executable, named `wriTeData`. This executable simply receives as arguments the name of the file to write to (the user can change it in the *codedStream functionObjects*), along with the data to be written, both provided by a system call from the *coded FunctionObject*. If the executable is not present in the case directory, but it is being called by the *FunctionObject*, a warning is displayed in the terminal informing about this situation (it is also possible to copy this script to a location loaded by default in each OpenFOAM® session, in order to avoid the need of having it in the case directory). Keep in mind that this *.sh* executable is only writing to a file the data it receives as argument, so that it is unlikely that the user would need to change it.

One main advantage of *coded FunctionObjects* is that they are case-specific, instead of solver-specific, which means that the code of the solver does not need to be changed. They are probably also a good starting point to begin programming in OpenFOAM®, since the compilation steps are automatically handled.

Note for foam-extend users: as mentioned in section 2.6, *coded* objects are not available in foam-extend. For this reason, a class of post-processing utilities (*ppUtil*) was assembled in a library (*libpostProcessingRheoTool.so*), which basically implement the same functions as the corresponding *coded FunctionObjects* in OpenFOAM® versions. Selecting the utility to be used, as well as its controls, is performed in (a new) subDict *PostProcessing* located in `fvSolution`. Three controls should be defined: *enabled*, which is a on/off switch that can be *true* or *false*; *evaluateInterval*, representing the number of **time-steps** after which the utility is evaluated; *funcType* should take the name of the *ppUtil* to be used (to see all the possibilities, just type any random word; use *none* if none of them is to be used or simply do not define this subDict). The boundary conditions implemented in OpenFOAM® versions as *coded* functions were added to library *libBCRheoTool.so* in the foam-extend version.

i For all the tutorials presented next, the commands required to run them are specified. It is instructive for less experienced users to type each one in the command line, in order to exactly know what is being done. However, in the directory for each tutorial we provide also a script named `Allrun` that automatically runs all these commands. On the other hand, the script `Allclean` also present cleans the directory, deleting everything that has been created.

5.1.3 Case 1: flow between parallel plates

 `tutorials/rheoFoam/Channel/Oldroyd-BLog/`

♥ Overview

In this tutorial, the flow between two infinite parallel plates is simulated for an Oldroyd-B fluid. Although apparently simple, this case can pose formidable difficulties to UCM and Oldroyd-B models at high Weissenberg numbers [16]. This example also shows that the log-conformation approach is effective in solving this stability issue, while retrieving the predicted analytical profiles.

Following ref. [17], the Reynolds number for this problem is defined as $Re = \frac{\rho U w}{\eta_0}$ and the Weissenberg number as $Wi = \frac{\lambda U}{w}(1 - \beta)$, where $\beta = \frac{\eta_s}{\eta_s + \eta_p} = \frac{\eta_s}{\eta_0}$. The case reproduced in the tutorial is for $Re = 0$ (the convective term in the momentum equation is suppressed), $Wi = 0.99$ and $\beta = 0.01$.

♥ Geometry & Mesh

The geometry is a planar channel (two parallel plates) with half-width w , Fig. 5.1. The mesh is composed of 50 cells in the x -direction and 60 cells in the y -direction, uniformly distributed in both directions.

♥ Boundary conditions

This flow is 2D, being solved in the xy -plane. A uniform velocity profile (U) is set at the inlet, along with zero-gradient for pressure and polymeric extra-stress components. At the outlet, fully-developed flow conditions are assumed (zero-gradient for all variables, except pressure, which is fixed to a constant value, $p = 0$). A no-slip boundary condition is assigned at the walls (velocity is null, polymeric extra-stress components are linearly extrapolated and a zero-gradient is assumed for pressure in the normal direction to the wall).

♥ Command-line

1–Build the mesh:

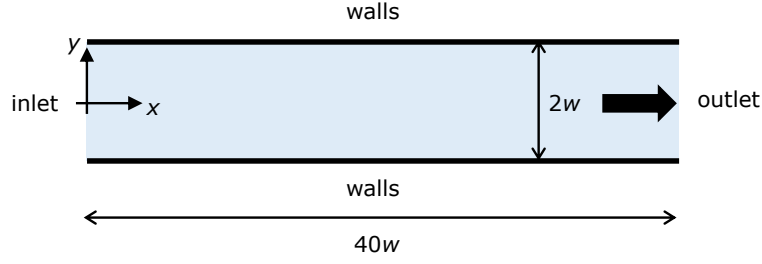


Figure 5.1: Planar channel geometry.

```
~$ blockMesh
```

2–Run the solver:

```
~$ rheoFoam
```

3–Extract profiles for \mathbf{u} and $\boldsymbol{\tau}$ along line $x = 35$:

```
~$ sample
```

♥ Results

Figure 5.2 presents the fully-developed profiles at line $x = 35$. The variables were normalized as follows: length is normalized with w , time with w/U , velocity with U and polymeric components of the extra-stress with $\frac{\eta_0 U}{w}$, as in ref. [17]. A good agreement is observed between the numerical results and the analytical solution written in dimensionless form [17]:

$$\begin{aligned} u_x &= \frac{3}{2} (1 - y^2) \\ \tau_{xy} &= -3(1 - \beta)y \\ \tau_{xx} &= 18Wi(1 - \beta)y^2 \end{aligned}$$

The user can test the solver with a UCM fluid ($\beta = 0$) and confirm that an accurate solution is still achieved, without facing any numerical issue. However, running the same cases without the log-conformation approach leads to solver divergence (try it!).

This tutorial probes a point in the flow over time (to check for convergence), which has been specified in `controlDict` dictionary. The data is written to a directory named `probes/`, whose location in the case directory depends on the OpenFOAM® version.

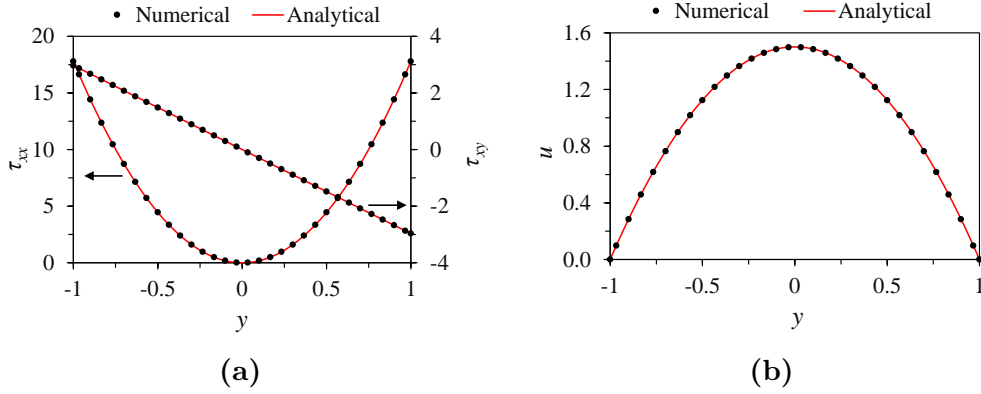



Figure 5.2: (a) Polymeric extra-stress components and (b) velocity, at $x = 35$, for $Re = 0$, $Wi = 0.99$ and $\beta = 0.01$.

5.1.4 Case 2: lid-driven cavity flow

 `tutorials/rheoFoam/Cavity/Oldroyd-BLog/`

📌 Overview

The flow in a lid-driven cavity is a common benchmark for numerical solvers, for both Newtonian and viscoelastic fluids, being one of the mostly used geometries for such purposes. One reason explaining the popularity is its simple geometry: a square in 2D or a cube in 3D.

For viscoelastic fluids, stress boundary layers develop at the walls and, at high Deborah numbers, the flow becomes time-dependent. An equivalent behavior is observed with Newtonian fluids for high Reynolds numbers.

The case reproduced in this tutorial is for an Oldroyd-B fluid with $\beta = 0.5$. The Deborah number is defined here as $De = \frac{\lambda U}{L}$, while the Reynolds number is $Re = \frac{\rho U L}{\eta_0}$. In this tutorial, we set $De = 1$ and $Re = 0.01$, so that the creeping flow assumption is still adequate, notwithstanding the finite Re .

📌 Geometry & Mesh

The planar lid-driven cavity is simply a square, with side length L , Fig. 5.3. The coordinate axis is fixed at the bottom-left corner. The mesh consists of one single block with 127 cells uniformly distributed in both directions.

📌 Boundary conditions

The flow is assumed to be 2D, being solved in the xy -plane. For the three stationary walls, a no-slip boundary condition is assigned, with null velocity, linearly extrapolated polymeric extra-stresses and zero normal gradient for pressure. At the moving lid wall, the same boundary conditions are used for pressure and

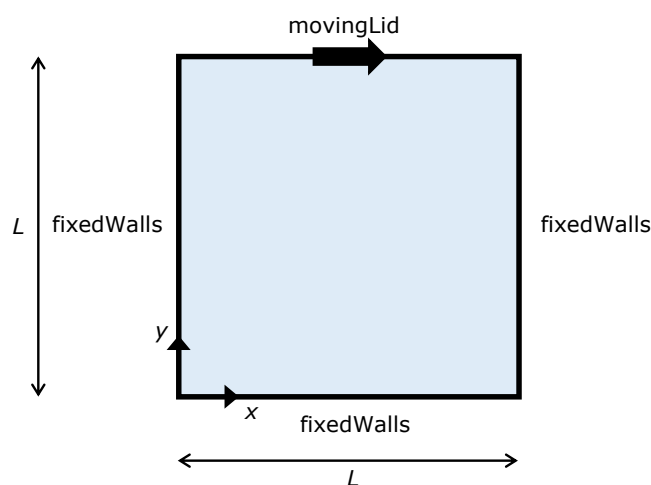


Figure 5.3: Geometry for the lid-driven cavity.

polymeric extra-stresses. Regarding the velocity, a time-space dependent condition is employed in order to impose a smooth start of the flow and to avoid a local singularity with infinite acceleration at the top-right and top-left corners [7]:

$$U_{\text{lid}}(x, t) = 8U [1 + \tanh \{8(t - 0.5)\}] x^2(1 - x)^2 \quad (5.1)$$

Equation 5.1 is directly implemented as a *codedFixedValue* boundary condition in file U, located in folder 0/. The variables were normalized as follows: length is normalized with L , time with L/U , velocity with U , and polymeric extra-stresses with $\frac{\eta_0 U}{L}$.

♥ Command-line

1–Build the mesh:

```
~$ blockMesh
```

2–Run the solver:

```
~$ rheoFoam
```

3–Extract profiles for \mathbf{u} and $\boldsymbol{\tau}$ along lines $x = 0.5$ and $y = 0.75$:

```
~$ sample
```

♥ Results


Figure 5.4 presents spatial profiles for the x -component of the velocity and for Θ_{xy} , along with the evolution over time of the volume-averaged "kinetic energy", defined as

$$E_k = \frac{1}{2V_t} \int |\mathbf{u}|^2 dV = \frac{1}{2V_t} \sum_{k=1}^N |\mathbf{u}_k|^2 V_k = \frac{1}{2N} \sum_{k=1}^N |\mathbf{u}_k|^2 \quad (5.2)$$

where N is the number of cells of the mesh and $V_t = NV_k$ for a uniform mesh. A good agreement is observed between the results obtained by *rheoFoam* and the reference data [7], which shows the good accuracy of the solver, both in space and time. The contour maps for the components of Θ are also provided in Fig. 5.5, together with the flow streamlines.

The "kinetic energy" is written on run time to the case directory, using a *codedFunctionObject*, in the `controlDict` dictionary. The reader can check in this function how Eq. (5.2) has been implemented.

5.1.5 Case 3: flow in a 4:1 planar contraction

 `tutorials/rheoFoam/Contraction41/Oldroyd-BLog/`

♥ Overview

The 4:1 planar contraction is another traditional benchmark flow problem for viscoelastic fluid flow solvers. The existence of singular points at the re-entrant corners, where stresses grow exponentially as the corner is approached, make this problem challenging from a numerical perspective.

This tutorial exactly reproduces the work that we developed in ref. [2] using an early version of *rheoFoam*, where an Oldroyd-B fluid ($\beta = \frac{1}{9}$) was studied for $De = 0 - 12$. The `constitutiveProperties` dictionary is set to reproduce the case for $De = 1$ and $Re = 0.01$.

♥ Geometry & Mesh

The geometry for this case is reproduced in Fig. 5.6. The mesh used corresponds to mesh M1 of ref. [2].

♥ Boundary conditions

The boundary conditions used are described in ref. [2]. It is worth mentioning that the time-varying inlet velocity is implemented as a *codedFixedValue* boundary condition in file `U`, located in folder `0/`. The function is implemented as

$$\mathbf{u}(t) = \begin{cases} \frac{1-\cos(\pi t)}{fac} \mathbf{dirN} & t \leq tlim \\ \mathbf{Uav} & t > tlim \end{cases} \quad (5.3)$$

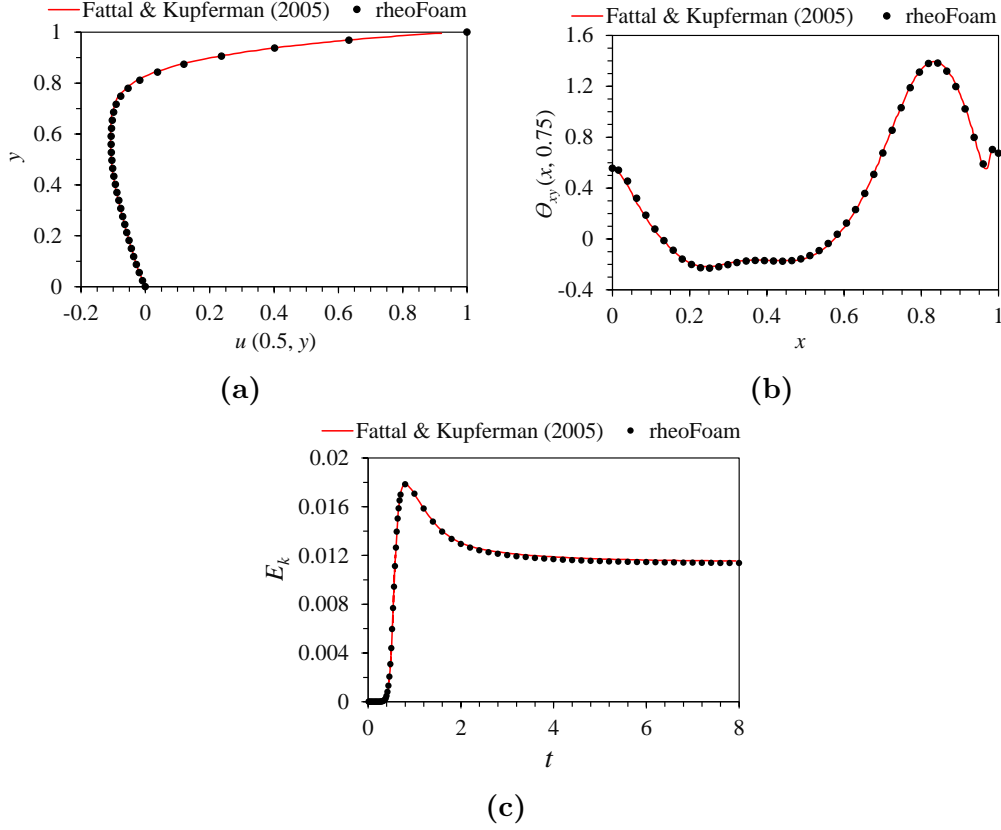


Figure 5.4: (a) Velocity profile along line $x = 0.5$ (at $t = 8$), (b) Θ_{xy} profile along line $y = 0.5$ (at $t = 8$) and (c) evolution of the average "kinetic energy" over time. All the results are for $Re = 0.01$, $De = 1$ and $\beta = 0.5$.

where \mathbf{Uav} and \mathbf{dirN} are vectors, and fac and $tlim$ are scalar parameters. For $\mathbf{Uav} = (0.25, 0, 0)$, $\mathbf{dirN} = (1, 0, 0)$, $fac = 8$ and $tlim = 1$, this generates an inlet velocity profile aligned with the x -axis and whose magnitude increases from 0 at $t = 0$ to 0.25 at $t = 1$.

♥ Command-line

1-Build the mesh:

```
~$ blockMesh
```

2-Run the solver:

```
~$ rheoFoam
```

3-Extract \mathbf{u} and $\boldsymbol{\tau}$ at the cell centers immediately upstream and downstream of vertical line $x = 0$:

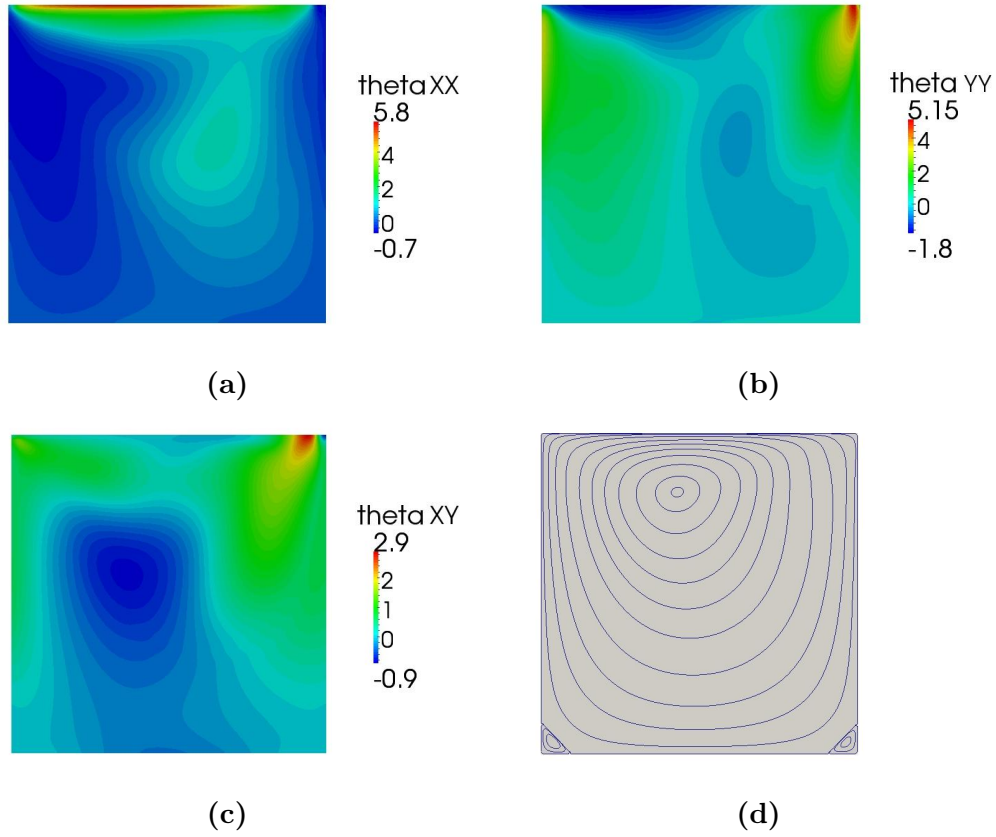


Figure 5.5: Contours of (a) θ_{xx} , (b) θ_{yy} and (c) θ_{xy} . In (d), the streamlines are plotted. All the results are for $Re = 0.01$, $De = 1$, $\beta = 0.5$ and $t = 8$.

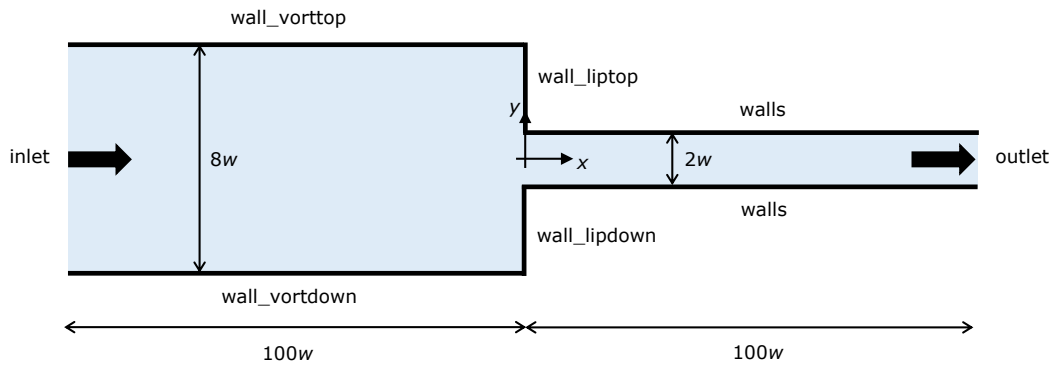


Figure 5.6: Geometry for the 4:1 planar contraction.


~\$ sample

♥ Results

The results obtained with mesh M1 can be found in ref. [2].

A *coded FunctionObject* returns the points where the wall-parallel velocity component changes sign (for both the walls near to the upper lip and corner vortices). Those points are delimiting the lip and corner vortices (if present). The user can easily add two extra *coded FunctionObjects* for the vortices in the lower-half of the contraction.

5.1.6 Case 4: flow around a confined cylinder

 tutorials/rheoFoam/Cylinder/Oldroyd-BLog/

♥ Overview

The planar flow past a confined cylinder is another traditional benchmark problem in computational rheology. Since this flow has no singular points and because extra-stresses can grow significantly in the wake of the cylinder, this problem is particularly well-suited to test the accuracy and stability of numerical methods. Furthermore, it is also a good problem to test the non-orthogonality handling by the algorithms, both in terms of accuracy and stability, since the grids used for this problem usually require some degree of non-orthogonality.

The Oldroyd-B model is used in this tutorial, since a reasonable amount of data is available in the literature for comparison purposes. The Reynolds number for this flow is defined as $Re = \frac{\rho UR}{\eta_0}$ and the Weissenberg number as $Wi = \frac{\lambda U}{R}$. In order to establish comparable conditions with refs. [18, 19], the solvent viscosity ratio (β) is fixed to 0.59, the blockage ratio (diameter of cylinder/width of the channel) is 50 % and $Re = 0$ (the convective term in the momentum equation is removed). The case at $Wi = 0.7$ is simulated in this tutorial.

♥ Geometry & Mesh

The geometry used in this case is composed of a channel with a cylinder of radius R vertically centered between its walls – spaced apart $4R$ –, and placed at a distance of $20R$ from the inlet, Fig. 5.7. The length of the channel downstream of the cylinder is $60R$. The mesh for this geometry is composed of 8 blocks, with a high cell-density near the wall of the cylinder. The minimum cell length in the radial direction of the cylinder is $0.0049R$, while in the tangential direction it is $0.0053R$.

♥ Boundary conditions

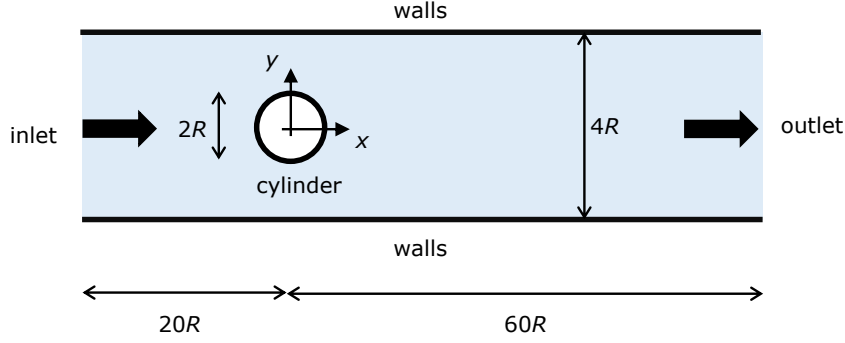


Figure 5.7: Cylinder vertically centered in a planar channel with 50 % blockage ratio.

The flow is assumed to be 2D, being solved in the xy -plane. At inlet, a uniform velocity profile with magnitude U is imposed, the polymeric extra-stresses are null and zero-gradient is assigned to pressure. Channel and cylinder walls are static (velocity is null, polymeric extra-stresses are linearly extrapolated to the walls and a zero-gradient is imposed for pressure). At the outlet, fully-developed conditions are assumed: zero-gradient for all variables, except pressure, which is fixed to a constant value, $p = 0$.

♥ Command-line

1–Create half of the mesh:

```
~$ blockMesh
```

2–Reflect the half-mesh using plane xz as mirror, to obtain the full mesh:

```
~$ mirrorMesh -noFunctionObjects
```

3–Run the solver:

```
~$ rheoFoam
```

♥ Results

Figure 5.8 presents the contour plots for the first normal stress difference and for the velocity magnitude (with superposed streamlines), at $Re = 0$, $Wi = 0.7$ and $\beta = 0.59$, using the Oldroyd-B model with the log-conformation approach. Note that the velocity is normalized with U , time with R/U and polymeric extra-stresses with $\frac{\eta_0 U}{R}$. The drag coefficient obtained in such conditions is $C_d = 117.357$, which is in good agreement with $C_d = 117.323$ [19] and $C_d = 117.315$ [18]. Refining


the mesh would further increase the accuracy of the numerical solution. The drag coefficient was computed as

$$C_d = \frac{1}{\eta_0 U H} \int_S \left(p \mathbf{I} + \boldsymbol{\tau}' \right) \cdot \hat{\mathbf{i}} \, dS = \frac{1}{\eta_0 U H} \sum_{k=1}^{Nf} \mathbf{S}_f \cdot \left(p_f \mathbf{I} + \boldsymbol{\tau}'_f \right) \cdot \hat{\mathbf{i}}$$

where \mathbf{S}_f is a vector normal to each face of the cylinder boundary, whose magnitude is equal to the face's area, H is the depth of the cylinder in the neutral (empty) direction and $\hat{\mathbf{i}}$ is a unitary vector aligned with the streamwise direction. The drag coefficient is retrieved (over time) by a *coded FunctionObject*, which can be found in `controlDict` dictionary.

It should be noted that, although the mesh displays some amount of non-orthogonality (run *checkMesh* to confirm it), the non-orthogonality corrector loop is not used and the pressure under-relaxation factor is kept relatively high (0.9). These features show the enhanced stability of the solver. However, to study for example the time evolution of the drag coefficient, the use of the non-orthogonality corrector loop, combined with inner iterations, is suggested in order to increase the time accuracy (by decreasing the explicitness of the method).

5.1.7 Case 5: bifurcation in a 2D cross-slot

 `tutorials/rheoFoam/CrossSlot/Oldroyd-BLog/`

Overview

While the previous tutorials were based on traditional benchmark flow problems, the case selected for this tutorial is a recent benchmark: the 2D cross-slot [20]. For sufficiently high Deborah numbers, it can be observed that the flow in such geometry becomes asymmetric (steady or unsteady) [20]. At the stagnation point, which is generated by the two counter-flowing streams, fluid elements can remain for a virtually infinite amount of time. Because the local strain-rate is non-zero, the accumulated strain is high, which is especially problematic for models considering springs with an infinite extension – the tensile normal stress grows exponentially over time near that point. Thus, a singular point exists in this case, although not being located at a wall, as commonly seen in other geometries with singularities.

In directory `tutorials/rheoFoam/CrossSlot/`, there are four tutorials for this case, each one using a different model or solution method. The tutorial described here is the one solving the Oldroyd-B model with the log-conformation approach (`Oldroyd-BLog/`). The remaining cases are: `Oldroyd-BRootk/`, which solves the Oldroyd-B model using the root^k kernel, with $k = 8$; `Oldroyd-BSqrt/`, which solves the Oldroyd-B model using the square-root transformation approach; and `sPTTlinearLog/`, which solves the linear sPTT model with the

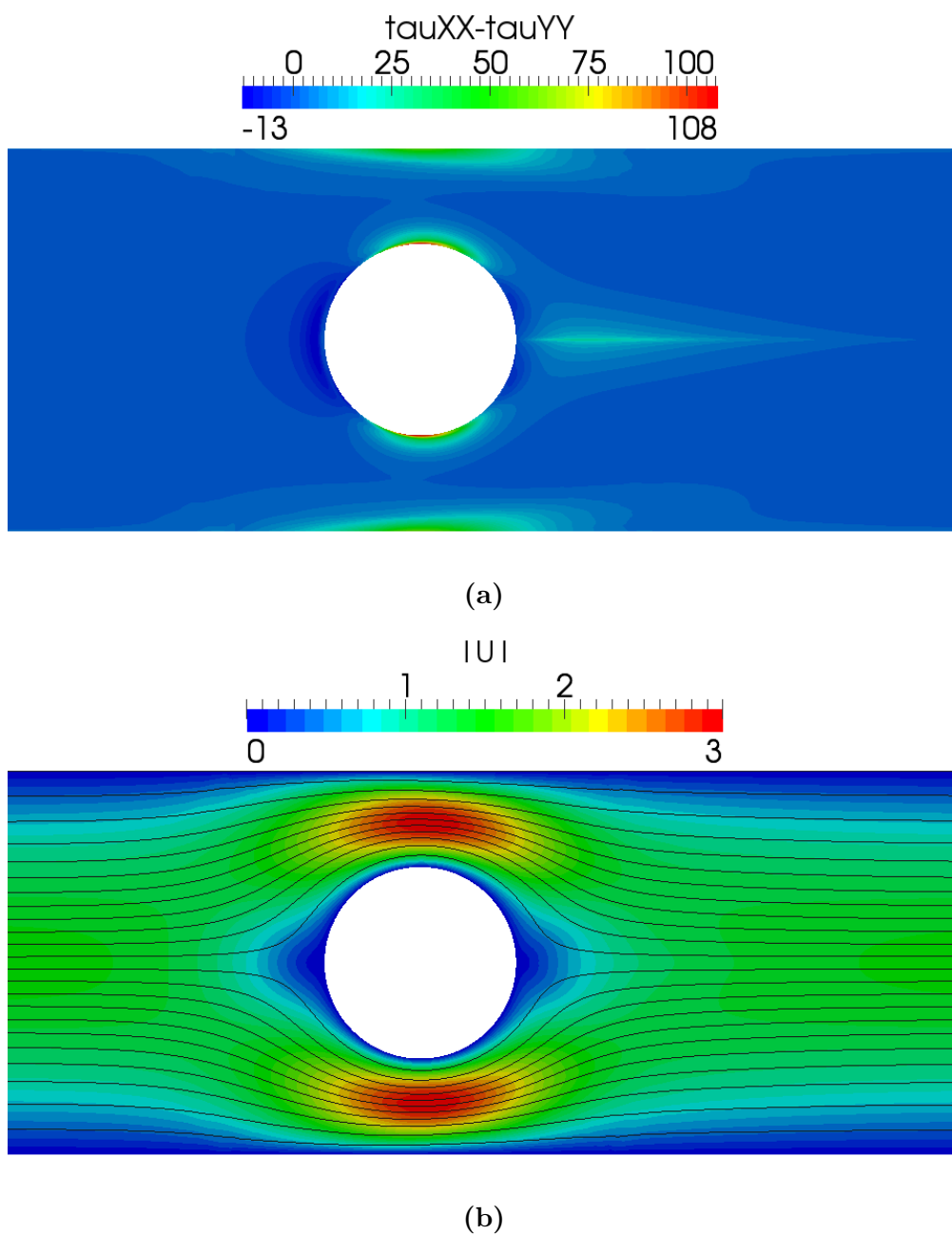


Figure 5.8: (a) First-normal stress difference and (b) velocity magnitude contours with superposed streamlines, at $t = 15$, for $Wi = 0.7$, $Re = 0$ and $\beta = 0.59$.

log-conformation approach. All the tutorials with the Oldroyd-B model are for the same conditions: $De = 0.33$, $Re = 0$ and $\beta = 0$ (UCM fluid). The tutorial

solving the linear sPTT model is for $De = 0.6$, $Re = 0$ and $\beta = 1/9$. This group of tutorials also solves the transport equation for a passive scalar, exemplifying how this extra-feature can be used.

The dimensionless numbers for this problem are defined as: $Re = \frac{\rho U H}{\eta_0}$, $Wi = \frac{\lambda U}{H}$ and $Pe = \frac{HU}{D}$. The Péclet number (Pe) is only relevant for the scalar transport equation and D is the diffusion coefficient of the passive tracer. We set $Pe = 500$ in all the cases (consider, for example, rhodamine-B in water flowing in a 200 μm wide channel, at 1 mm/s).

♥ Geometry & Mesh

The cross-slot geometry for this tutorial is depicted in Fig. 5.9. It consists of four identical arms (width H ; length $10H$), where the two vertically opposite arms are inlets and the remaining are outlets. Each arm is meshed as a single block with 60 cells in the streamwise direction (cells are compressed near the origin) and 51 cells uniformly distributed in the transverse direction. As a consequence, the central square block, $(x, y) \in [-0.5H, 0.5H]$, has 51x51 uniformly spaced cells. The use of an odd number of cells in both directions of the central square generates a cell exactly centered at the stagnation point, which is advantageous for the post-processing of quantities of interest at this location.

♥ Boundary conditions

The flow is assumed to be 2D, being solved in the xy -plane. At both inlets, a uniform velocity profile is specified, with magnitude U and pointing to the origin, so that those two streams are flowing in opposite directions. The polymeric extra-stresses are null and for pressure a zero-gradient is used. The walls are static, thus velocity is null, polymeric extra-stresses are linearly extrapolated and the pressure is assumed to not change in the normal direction. Fully-developed flow conditions are assigned at the outlets: null normal gradient for all variables, except pressure, which is fixed at $p = 0$.

A passive scalar (tracer field C) is added to the problem, which requires the assignment of initial and boundary conditions. We impose a continuous injection of C at *inlet_north* ($C = 1$), while no tracer is injected at *inlet_south* ($C = 0$). In the remaining boundaries, we impose null normal gradient (meaning no flux of C across the walls and fully-developed flow conditions at both outlets). At time $t = 0$, when the simulation is started, the y -positive portion of the cross-slot is filled with the tracer ($C = 1, y > 0 \wedge t = 0$).

♥ Command-line

1-Build the mesh:

```
~$ blockMesh
```

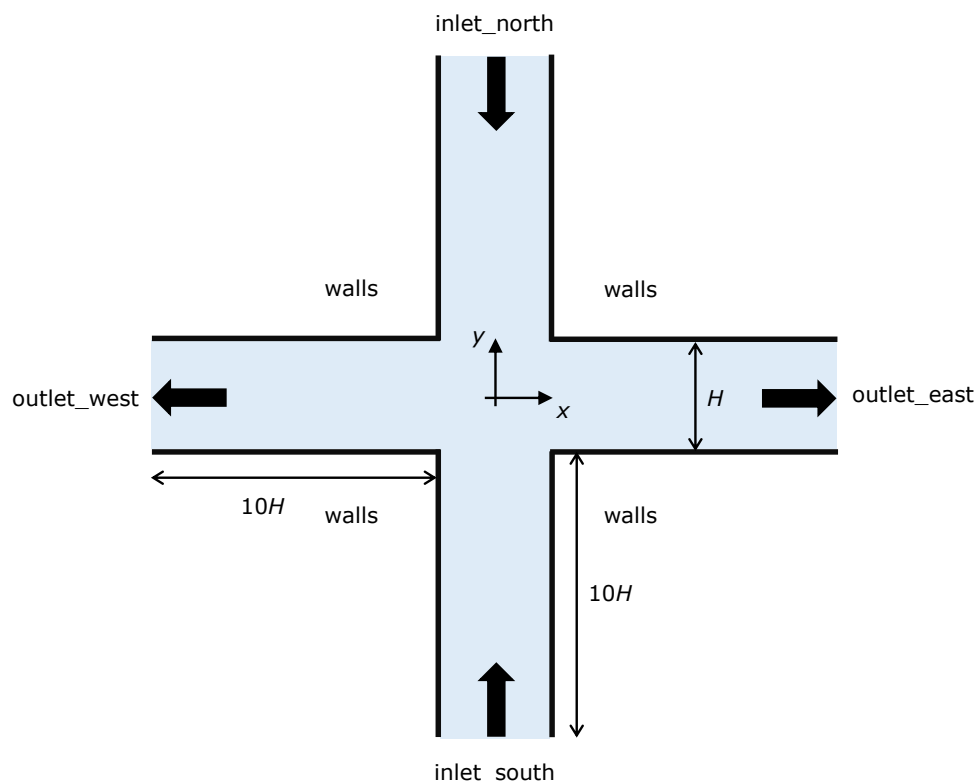


Figure 5.9: Cross-slot geometry composed of 4 arms: two inlets and two outlets.

2—Create field C by copying one already present, which is not initialized in the interior domain ($C = 1, y > 0 \wedge t = 0$):

```
~$ cp 0/C.org 0/C
```

3—Initialize field C in the interior domain:

```
~$ setFields
```

4—Run the solver:

```
~$ rheoFoam
```

📌 Results

The contour plots for some important variables are displayed in Fig. 5.10, for $Re = 0$, $Wi = 0.33$, $\beta = 0$ (UCM model) and $Pe = 500$. The variables are normalized with U (velocity), H/U (time) and $\frac{\eta_0 U}{H}$ (stresses).

The local Weissenberg number at the origin, defined in ref. [20] as the product of the relaxation time by the velocity gradient magnitude on the stagnation point streamlines, is $Wi_0 = 0.523$, which is relatively close to the benchmark value obtained in a similar mesh, $Wi_0 = 0.509$ (ref. [20], for mesh M1). The local Weissenberg number at the origin is retrieved by the solver to the case directory, through a *coded FunctionObject*, which can be found in `controlDict` dictionary.

The importance of the stress-velocity coupling term in this case can be evaluated by re-running the tutorial with this option disabled (keyword *uTauCoupling* in dictionary `constitutiveProperties`). Checkerboard fields easily develop in such conditions (this is a critical case due to the use of a UCM fluid).

Note that this tutorial makes use of a variable time-step, controlled by a maximum Courant number fixed at 0.4. This strategy is used because only steady-state results are of interest. This is also the reason to use a high tolerance in the sparse matrix solvers, in `fvSolution` dictionary – the steady asymmetry in the flow develops faster with these conditions, since the accumulated numerical error is higher.

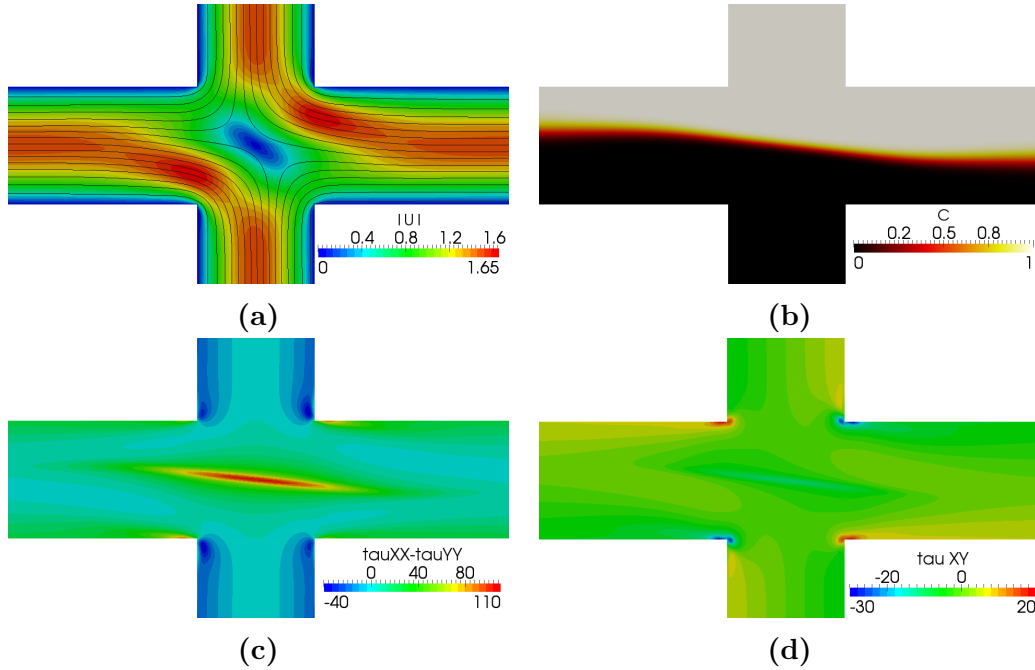


Figure 5.10: (a) Velocity magnitude contours with superimposed streamlines, (b) contours of C , (c) first-normal stress difference and (d) τ_{xy} contours, for $Re = 0$, $Wi = 0.33$, $\beta = 0$ and $Pe = 500$.

5.1.8 Case 6: blood flow simulation in a real-model aneurysm

 `tutorials/rheoFoam/Aneurysm/HerschelBulkley/`

♥ Overview

The tutorial presented in this section addresses the blood flow in a real-model aneurysm. Contrarily to the previous tutorials, this case is based on a 3D polyhedral (non-orthogonal) mesh and uses a GNF model. Furthermore, the flow is simulated for moderate Reynolds number, which will test the robustness of the solver for such conditions, where inertia already plays a special role.

The Herschel-Bulkley model was selected to simulate the blood rheology, following ref. [21]. The generalized Reynolds number for this model, assuming $\tau_0 = 0$ – the power law limit –, is [21]:

$$Re_{GN} = \frac{\rho(2R_{in1})^n \bar{U}^{2-n}}{k \left(\frac{3n+1}{4n}\right)^n 8^{n-1}}$$

This tutorial simulates the flow at $Re_{GN} = 420$.

♥ Geometry & Mesh

The STL file of the aneurysm surface was downloaded from a repository with model cerebral aneurysms [22], extracted from 3D rotational angiographies of diseased patients. From the list of available models, case ID *C0005* was selected, which refers to an aneurysm in the internal carotid artery (ICA).

The surface is composed of one main entry vessel (*in1*, the ICA), which bifurcates into two smaller vessels (*out1* and *out2*), Fig. 5.11a. The aneurysm is located near the bifurcation point. Due to the long extension of vessels upstream and downstream of the aneurysm in the original STL file, all the vessels were shortened and a cylindrical extension was connected to each one, in order to minimize entry/exit effects in the region near the aneurysm. The locations where those connections were performed are highlighted in Fig. 5.11a using red arrows. The transition length between the tube connectors and the vessels is typically 10 % of the vessel radius and the radius of those tubes is equal to the equivalent radius of the vessels at the connection point. The radius of each inlet/outlet is: $R_{in1} = 1.66$ mm, $R_{out1} = 1.17$ mm and $R_{out2} = 0.95$ mm.

The mesh for the geometry was built using ¹*cfMesh*, a meshing tool available in foam-extend since version 3.2. The maximum cell size was limited to 5 mm and boundary cell layers were generated near to the vessel walls in order to accurately solve the gradients developed there, Fig. 5.11b. The mesh provided with this tutorial has around 280 kcells.

¹<http://cfmesh.com/>

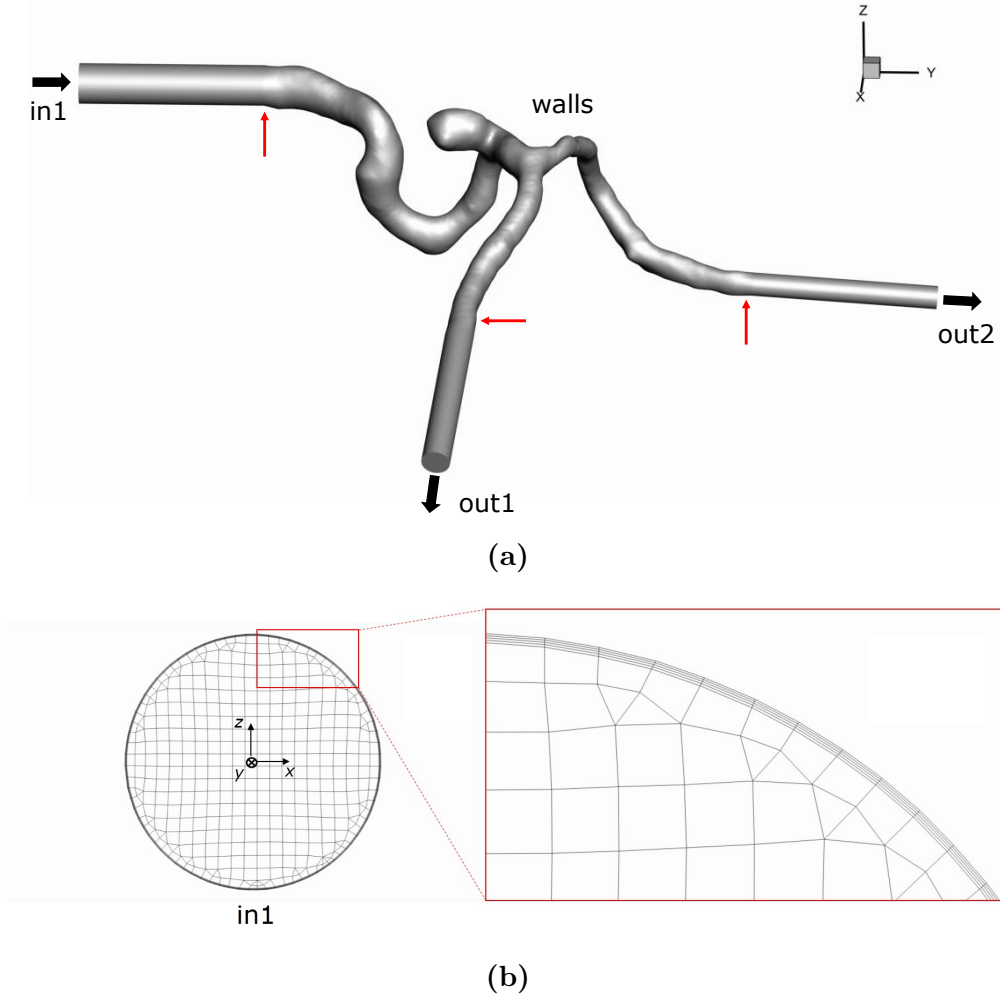


Figure 5.11: (a) Geometry of the aneurysm considered in the tutorial. Red arrows point to the transition regions between the aneurysm and the cylindrical extensions. The radius of each inlet/outlet is: $R_{in1} = 1.66$ mm, $R_{out1} = 1.17$ mm and $R_{out2} = 0.95$ mm. (b) Detailed view of the mesh on patch *in1*, zooming the cell layers near the wall. The reference axis for the geometry is centered on patch *in1*, with the normal vector of the patch pointing in the negative direction of the *y*-axis.

♥ Boundary conditions

The boundary conditions and fluid properties used in this tutorial are based in ref. [21], where also ICA aneurysms were studied. Accordingly, blood is modeled with a Herschel-Bulkley model, with $\tau_0 = 0.0175$ Pa, $k = 8.9721 \times 10^{-3}$ Pa.s^{*n*}, $n = 0.8601$ and $\eta_0 = 0.15$ Pa.s. Note that η_0 (see Table 4.1) is a parameter

characteristic of the model implementation in *rheoTool*, which limits the viscosity for low strain-rate values – otherwise it would generate infinite values at points with zero shear-rate. A relatively high value was attributed to η_0 in order to not affect the results. Furthermore, a density of 1050 kg/m^3 is considered.

For this fluid model, a fully-developed velocity profile is imposed at inlet, along with a null pressure gradient. For small $\boldsymbol{\tau}_0$, we can use (approximately) the fully-developed velocity profile for a Power law fluid:

$$U = \bar{U} \frac{3n+1}{n+1} \left[1 - \left(\frac{r}{R} \right)^{\frac{n+1}{n}} \right] \quad (5.4)$$

where \bar{U} is the mean velocity. In our case, \bar{U} is constant over time, thus steady conditions are simulated, instead of the cardiac cycle (this is to shorten the simulation time, since transient solutions would require a lower time-step, leading to a higher computational time). Regarding the walls, a no-slip boundary condition is imposed. At the outlets, the pressure is fixed to zero and the velocity is assumed to be fully-developed (zero gradient for velocity). Since the Reynolds number used in the tutorial is well below the critical value for transition to the turbulent regime, no special conditions need to be defined regarding turbulence modeling.

♥ Command-line

The mesh is already built and can be found in folder `polyMesh/`.

1–Decompose the case among two processors to speed-up the computations (with two processors, it takes around 1h to reach convergence in a laptop with an Intel i5-3210M processor, 2.5 GHz):

```
~$ decomposePar
```

2–Run the solver in parallel, using 2 processors:

```
~$ mpirun -np 2 rheoFoam -parallel
```

3–Reconstruct the last time-step of the case for post-processing:

```
~$ reconstructPar -latestTime
```

♥ Results

The results obtained at $Re_{GN} = 420$ are displayed in Fig. 5.12, where both the streamlines and the wall shear-stress magnitude (*WSSmag*) contours are shown. The wall shear-stress magnitude was computed as

$$WSSmag = \text{mag} \left[\eta(\dot{\gamma}) \mathbf{n} \cdot (\nabla \mathbf{u} + \nabla \mathbf{u}^T) \right] \quad (5.5)$$

where \mathbf{n} is the vector normal to the wall boundary and $\eta(\dot{\gamma})$ is the shear-rate dependent viscosity given by the Herschel-Bulkley model. The reader can check its implementation as a *coded FunctionObject* in dictionary `controlDict`.

This tutorial is defined to run with an adjustable time-step, controlled by a maximum Courant number, which we fix to 50. The high Courant number is to quickly achieve the steady-state, without the need to cancel the time-derivatives. The non-orthogonality corrector loop was turned on, with 1 iteration *per* time-step (the pressure is also under-relaxed with a factor of 0.9), in order to avoid possible numerical issues arising from the use of such a high Courant number. The convergence can be monitored with the data retrieved over time by a probe located at one of the exit vessels, downstream of the aneurysm.

5.2 *rheoTestFoam*

5.2.1 General guidelines

In section 4.2.2, *rheoTestFoam* was presented as a testing application for the constitutive models implemented in *rheoTool*, being not a general-purpose solver. For this reason, some of the traditional steps required to setup a simulation in OpenFOAM® are not necessary with *rheoTestFoam*, while, on the other hand, extra-inputs need to be specified.

constant/

One main difference of *rheoTestFoam* cases regarding, for example, *rheoFoam* cases is in the mesh: the user should always use the same single-cell unitary mesh when working with *rheoTestFoam*. Thus, changing the mesh from case to case is unnecessary.

The dictionary `constitutiveProperties` is composed of two subDict: `parameters` and `rheoTestFoamParameters`, as displayed in Listing 5.2. The entries in `parameters` have exactly the same meaning, as previously discussed for *rheoFoam*. However, there are two entries which remain inactive in *rheoTestFoam*: `rho` and `uTauCoupling`. Remember from section 4.2.2, that *rheoTestFoam* is only solving the constitutive equations for a given $\nabla \mathbf{u}$ tensor, thus those two parameters related with momentum equation are meaningless. Nevertheless, they should be present (with any assigned value) to avoid a run time error. `rheoTestFoamParameters` is a subDict specific to *rheoTestFoam*, in the same way as `passiveScalarProperties` is a particular subDict of *rheoFoam*. The keyword `ramp` stands for the desired operation mode (see section 4.2.2): `ramp` (`true`) or transient (`false`). The other two entries define tensor $\nabla \mathbf{u}$, since we consider $\nabla \mathbf{u} = \text{gammaEpsilonDotL}[i] \cdot \mathbf{gradU}$, where i is the index representing each entry of list `gammaEpsilonDotL`. If `ramp = false`, the mode is transient and only one entry is expected in `gammaEpsilonDotL` – the solver is testing the transient behavior of the constitutive model, for a (single) given $\nabla \mathbf{u}$. On the other hand, for ramp mode (`ramp = true`), `gammaEpsilonDotL` may have as many entries as

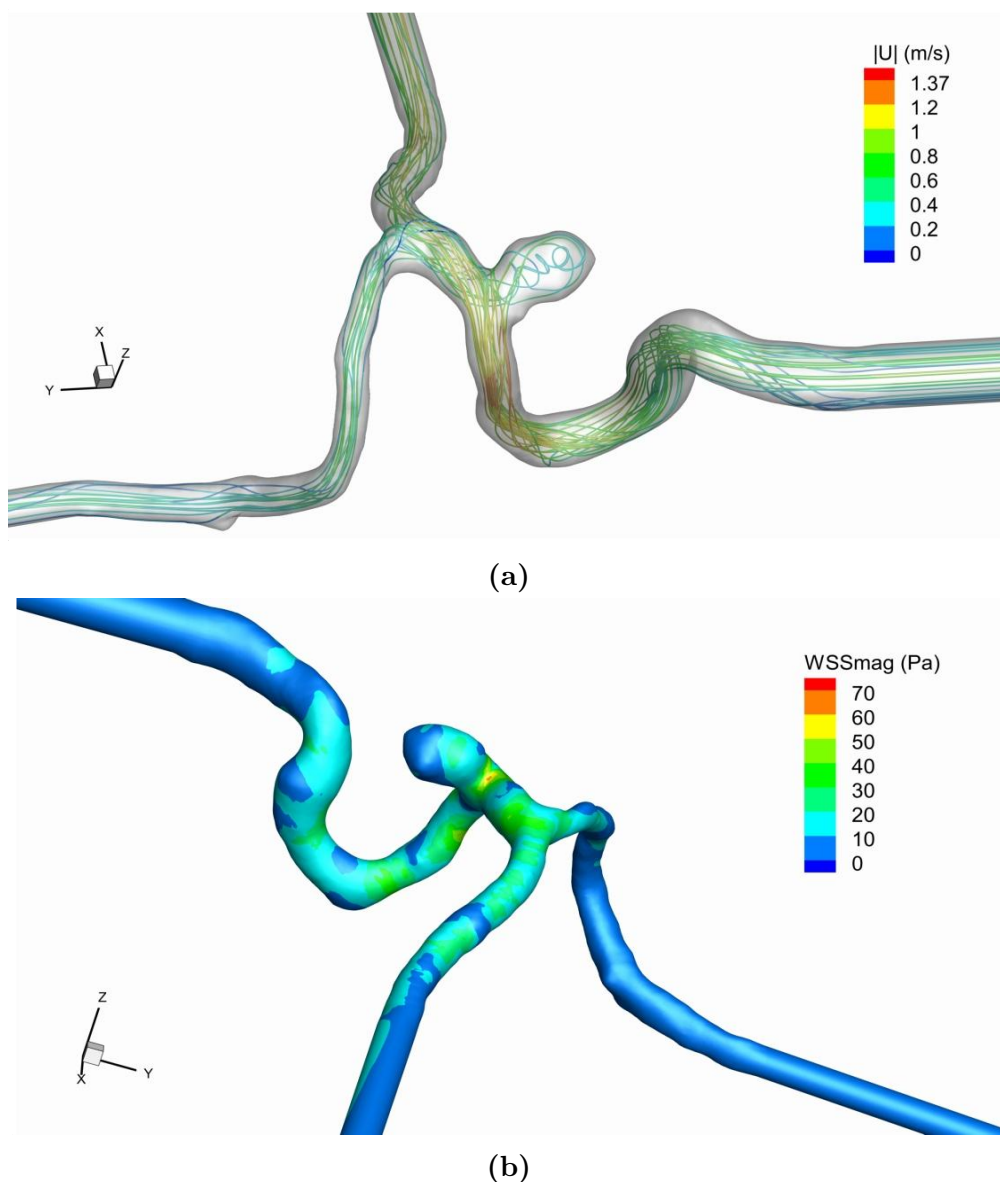


Figure 5.12: (a) Streamlines (colored with the velocity magnitude) and (b) wall shear-stress magnitude contours, at steady-state and for $Re_{GN} = 420$.

desired by the user and steady-state variables will be returned by the solver for each entry. Any combination of components is admissible for $gradU$, although only some correspond to canonical rheometric flows. The one displayed in Listing 5.2 is for a pure-shear flow: $\mathbf{u} = (\dot{\gamma}y, 0, 0)$, where $\dot{\gamma}$ is the *gammaEpsilonDotL* value.

```
1 parameters
{
```

```

3      type                Oldroyd-B;

5      etaS                etaS [1 -1 -1 0 0 0 0] 1.;
      etaP                etaP [1 -1 -1 0 0 0 0] 1.;
7      lambda              lambda [0 0 1 0 0 0 0] 0.1;

9  // Place-holder variables in rheoTestFoam
      uTauCoupling        false;
11     rho                 rho [1 -3 0 0 0 0 0] 0.;
13 }

rheoTestFoamParameters
15 {
17     ramp                false;

19     gradU               (0.    0.    0.
21                        1.    0.    0.
23                        0.    0.    0.);

25     gammaEpsilonDotL    (
                        1.
                        );
}

```

Listing 5.2: Example of a `constitutiveProperties` dictionary used with *rheoTestFoam*.

0/

When using *rheoTestFoam*, only the boundary conditions for the velocity field are required, since they will be needed to define $\nabla \mathbf{u}$ when using the standard discretization tools of OpenFOAM® (the remaining fields should also be defined, but any value can be assigned to internal/boundary fields). Independently of the user's options in dictionary `constitutiveProperties`, the velocity at all the 6 faces of the unitary single-cell constituting should be kept at $\mathbf{u} = (0,0,0)$, although assigning any other value would not effect the results. The solver will internally manipulate those values in order to fulfill the specified $\nabla \mathbf{u}$ (Fig. 4.2). Shortly, both the mesh and folder 0/ provided in the tutorials can be readily applied to any fluid, without any change.

system/

When running *rheoTestFoam* in ramp mode, the user does not have control on Δt (time step), nor on the *endTime*. The time step is automatically set based on the relaxation time and strain-rate values for viscoelastic fluids or is simply set to 1 s for GNF (in this case, the value is not important since no equation is solved implicitly). The *endTime* in ramp mode is not important, since the

stopping criteria is based on an hard-coded threshold for the residuals and for the number of iterations. On the other hand, in transient mode, both variables should be specified by the user in `controlDict`. Regarding discretization schemes (`fvSchemes` dictionary), only time-derivatives and $\text{grad}(U)$ are used by the solver. The discretization of $\text{grad}(U)$ should be kept as *Gauss linear*, while any valid time-scheme can be selected (except steady-state), although in ramp mode this should not make any difference, since we are looking for steady-state solutions. In dictionary `fvSolution`, the matrix solvers required by the constitutive equations must be defined and the number of inner iterations may also be manipulated if running in transient mode. Note that since the mesh has only one cell, a good time accuracy can be achieved by selecting a small time step, without compromising the CPU time (in general, simulations will be always fast). The use of under-relaxation is not needed, as long as time-derivatives are not disabled in `fvSchemes` (this is our recommendation).

5.2.2 Case I: Herschel-Bulkley model

 `tutorials/rheoTestFoam/HerschelBulkley/`

♥ Overview

This tutorial illustrates the behavior of the Herschel-Bulkley model used in tutorial *Case 6* to model the blood rheology. A steady shear flow is considered for this purpose.

♥ Geometry & Mesh

The geometry used with *rheoTestFoam* is always the same (see sections 5.2.1 and 4.2.2). The mesh is already built (do not change it).

♥ Boundary conditions

The boundary conditions to be used with *rheoTestFoam* are always the same (see sections 5.2.1 and 4.2.2). Folder `0/` should not be changed.

♥ Command-line

1–Run the solver:

```
~$ rheoTestFoam
```

The file `Report` is created in the case directory, which contains the results.

♥ Results

For a GNF model, only the ramp mode of *rheoTestFoam* makes sense to be used, since thixotropy is not included in any of the GNF models implemented. A steady shear flow is used, thus $\frac{\partial u}{\partial y}$ is the only non-zero component of tensor $\nabla \mathbf{u}$.

For the range of shear-rates between 0.01 s^{-1} and 10000 s^{-1} , the Herschel-Bulkley model behavior is displayed in Fig. 5.13. The model predicts a shear-thinning behavior for $\dot{\gamma} > \dot{\gamma}_0$, where $\dot{\gamma}_0$ is the critical strain-rate at which $\eta = \eta_0$, being $\dot{\gamma}_0 = 0.13 \text{ s}^{-1}$ for the parameters defined in this example.

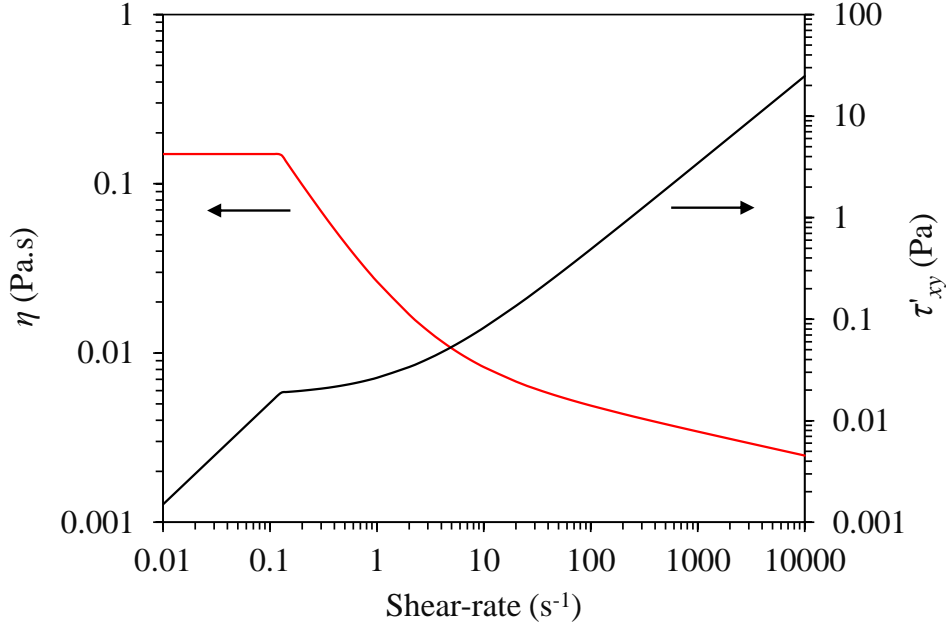



Figure 5.13: Shear viscosity and τ'_{xy} (the only non-zero component of the symmetric extra-stress tensor) as a function of the shear-rate, in a steady shear flow, for the Herschel-Bulkley model with parameters: $\tau_0 = 0.0175 \text{ Pa}$, $k = 8.9721 \times 10^{-3} \text{ Pa.s}^n$, $n = 0.8601$ and $\eta_0 = 0.15 \text{ Pa.s}$.

5.2.3 Case II: FENE-CR model

 tutorials/rheoTestFoam/FENE-CR/

Overview

This tutorial exemplifies the use of *rheoTestFoam*, both in transient and ramp modes, with a constitutive equation for a viscoelastic fluid. The FENE-CR model is selected and its behavior will be assessed for uniaxial extensional flow.

The uniaxial extensional flow may be described by the following velocity gradient

$$\nabla \mathbf{u} = \dot{\epsilon} \begin{bmatrix} 1 & 0 & 0 \\ 0 & -\frac{1}{2} & 0 \\ 0 & 0 & -\frac{1}{2} \end{bmatrix}$$

where $\dot{\epsilon}$ is the extensional rate. The Weissenberg number, $Wi = \lambda\dot{\epsilon}$, is the dimensionless group controlling the rate of stretch induced in the fluid and it was varied between 0.01 and 100, by increasing the extensional rate from 0.01 to 100 s^{-1} . The fluid properties used in the FENE-CR model are: $\eta_s = 0.1$ Pa.s, $\eta_p = 0.9$ Pa.s, $\lambda = 1$ s and different values of L^2 were tested (10, 100 and 1000). In such conditions, the extensional viscosity, defined as $\eta_E = \frac{\tau_{xx} - \tau_{yy}}{\dot{\epsilon}}$, is given by [14]

$$\eta_E = 3\eta_s + \eta_p \left(\frac{2}{1 - 2\lambda\dot{\epsilon}/f} + \frac{1}{1 + \lambda\dot{\epsilon}/f} \right) \quad (5.6)$$

where f is the solution of the cubic equation

$$(L^2 - 3)f^3 - [(\lambda\dot{\epsilon})(L^2 - 3) + L^2] f^2 - [2(\lambda\dot{\epsilon})^2(L^2 - 3) - (\lambda\dot{\epsilon})L^2 + 6(\lambda\dot{\epsilon})^2] f + 2(\lambda\dot{\epsilon})^2 L^2 = 0 \quad (5.7)$$

♥ Geometry & Mesh

The geometry used with *rheoTestFoam* is always the same (see sections 5.2.1 and 4.2.2). The mesh is already built (do not change it).

♥ Boundary conditions

The boundary conditions to be used with *rheoTestFoam* are always the same (see sections 5.2.1 and 4.2.2). Folder 0/ should not be changed.

♥ Command-line

By default, the tutorial will run in ramp mode.

1-Run the solver:

```
~$ rheoTestFoam
```

Take a look to file *Report* created in the case directory, which contain the results.

♥ Results

The results computed by *rheoTestFoam* and displayed in Fig. 5.14a, clearly illustrate that the FENE-CR model is correctly implemented, since the difference to the analytical solution is negligible.

In addition to the "steady" results in Fig. 5.14a, also the transient evolution of the extensional viscosity (commonly denoted as η_E^+ in the literature) can be obtained with *rheoTestFoam*. For that purpose, simply switch the keyword *ramp* (in *constitutiveProperties*) from *true* to *false* and define the desired extensional rate as the first entry of list *gammaEpsilonDotL* (the remaining entries can be left, since they will not be read). The results obtained for $Wi = 2, 5$ and 10 are displayed in Fig. 5.14b.

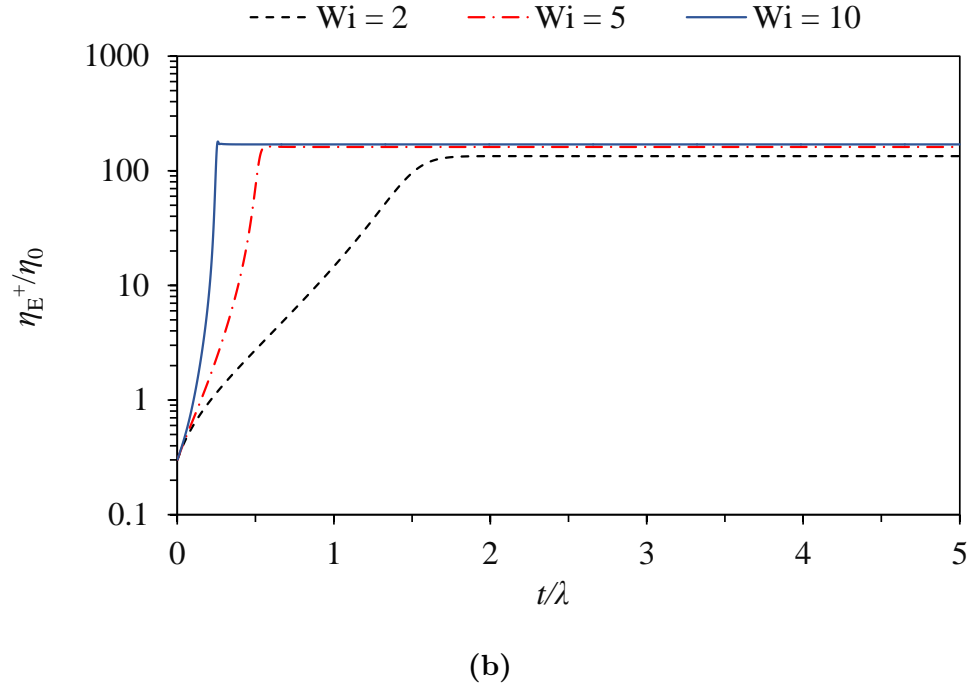
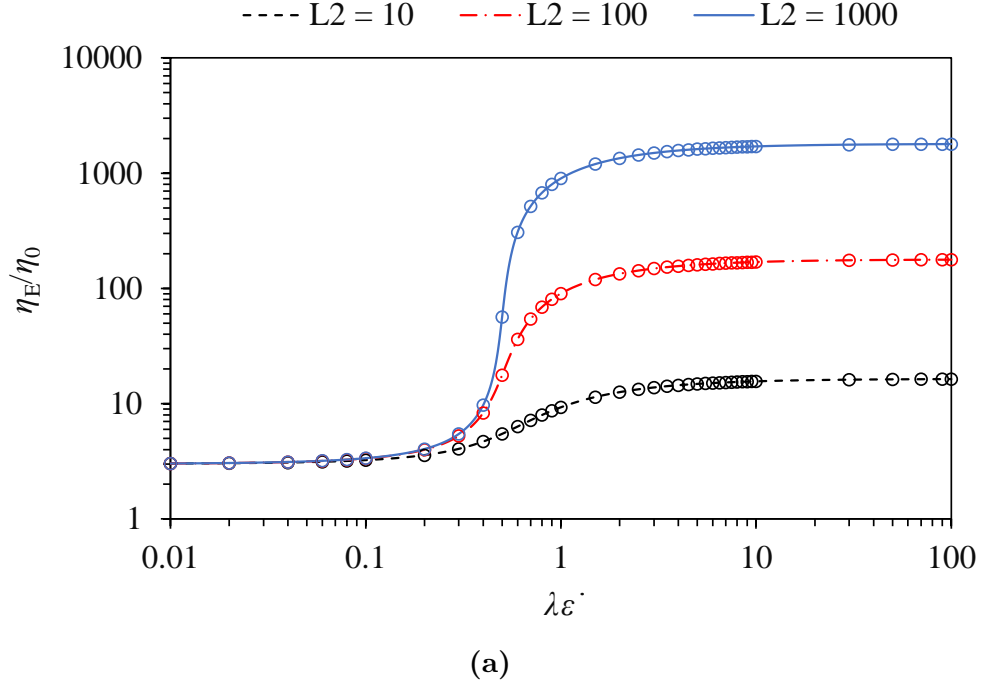


Figure 5.14: (a) Steady extensional viscosity (η_E) as a function of $Wi = \lambda \dot{\epsilon}$, for different values of L^2 (points represent the numerical results of *rheoTool* and the lines correspond to the analytical solution of Eq. 5.6); (b) transient extensional viscosity η_E^+ for different Wi , at fixed $L^2 = 100$. The remaining parameters of the FENE-CR model are $\eta_s = 0.1$ Pa.s and $\eta_p = 0.9$ Pa.s.

5.3 *rheoInterFoam*



Section 5.3 is under development.

5.3.1 General guidelines

Since most of the steps required to set up a case for *rheoInterFoam* are the same as for *rheoFoam*, only the major differences will be pointed out.

constant/

The dictionary `constitutiveProperties` should contain the same information as detailed for *rheoFoam* (5.1.1), for each phase. The principle is the same as for the default two-phase solvers of OpenFOAM® (e.g. *interFoam*), where each phase owns a dictionary defining its physical properties. Importantly, `subDict passiveScalarProperties`, related with the transport of a passive scalar, is general for the two phases and should only be defined once, outside each phase. Finally, the surface tension between the two phases should also be present and it is named *sigma*.

Note that when using default names *phase1* and *phase2* for each phase, without specifying the name of the phases in a *wordList*, then it is automatically assumed that the phases are labeled 1 and 2, respectively. Recent versions of OpenFOAM® have a slightly different behavior regarding phases labeling.

0/

In folder 0/, the internal and boundary field for the indicator (color) function used by the VOF method should be defined. The indicator has a value of 1 for one of the phases and 0 for the other phase. Ideally, and assuming that boundary conditions were correctly assigned, the indicator should remain bounded in this range, since the contrary would signify non-conservativeness. The name given to the file representing the indicator field should be consistent with the naming in dictionary `constitutiveProperties`. If the default names *phase1* and *phase2* were used, then the indicator function should be named *alpha1*. If other name was used instead, then the indicator would be named *alpha* suffixed with that name, without spaces in-between (recent versions of OpenFOAM® require a separation point). Although there are always two phases, only one indicator field should be defined, since the indicator for the other phase is computed from this one, due to its conservation.

Given that a constitutive equation is being defined and solved individually for each phase, variables *tau* and *theta* should be labeled (suffixed) with the respective phase name. Considering a viscoelastic model for each phase and default naming of phases, we would have *tau1*, *theta1* and *tau2*, *theta2*. If a multimode model is assigned to a given phase, then the name of each mode should also be appended.

system/


The main novelty regarding *rheoFoam* is that when using *rheoInterFoam* the user has the possibility to choose between PIMPLE and SIMPLEC for pressure-velocity coupling. This is controlled in dictionary *fvSolution*, in subDict *PIMPLE*, where the keyword *SIMPLEC* can be assigned to *true* or *false*. When in *PIMPLE* mode, the variable *nCorrectors* works in its usual way (looping the pressure equation) and the variable *nInIter* assumes the same function as *nOuterCorrectors*. In SIMPLEC mode, only *nInIter* is functional. Independently of the choice, the momentum equation is always solved. In a future release of *rheoInterFoam*, this workflow will most likely change. There are currently these two options because it is still not clear which one is more advantageous. Still in dictionary *fvSolution*, other keywords must be assigned in subDict *PIMPLE*, which are related with the VOF method and that the reader can find in the tutorials provided.

In dictionary *fvSchemes*, the discretization schemes for the two phases should be defined (only the convective term is differentiated), as well the discretization schemes related with the VOF method, which can also be found in the tutorials.

Besides the Courant number, the solvers using VOF, as *rheoInterFoam*, can also restrict the time-step based on an interface Courant number, which should be defined in dictionary *controlDict*.

The dictionary *setFields*, used to initialize parts of the domain with specified values, should also be present in folder *system/* whenever used.

5.3.2 Case 1: impacting drop

 **tutorials/rheoInterFoam/Oldroyd-BLog/ImpactingDrop/**

Overview

In this tutorial, a liquid drop composed of a viscoelastic fluid falls under gravity and its shape (the drop width more precisely) is monitored once the drop impacts a rigid plate. Under such conditions, the drop width oscillates after the impact. This problem has been used in the literature as a benchmark case for viscoelastic two-phase flow solvers (e.g. [23, 24]).

The configuration adopted in the tutorial aims to reproduce the conditions of ref. [23], where an axisymmetric geometry has been used. The dimensionless numbers governing the flow are: $Fr = \frac{U_0}{\sqrt{gD}}$, $Re = \frac{\rho U_0 D}{\eta_0}$, $Wi = \frac{\lambda U_0}{D}$ and $\beta = \frac{\eta_s}{\eta_s + \eta_p}$, where U_0 is the initial velocity of the drop, g is the gravitational acceleration, D is the drop diameter, ρ is the fluid density, η_0 is the total viscosity (polymer + solvent) and λ is the relaxation time (all these fluid properties are for the drop phase). The problem was simulated for $Fr = 2.26$, $Re = 5$, $Wi = 1$ and $\beta = 0.1$. Note that this is assumed to be a free surface flow, so that the surface tension is set to zero and we assign low (but finite) density and viscosity values to the fluid surrounding the drop.

♥ Geometry & Mesh

The geometry is composed by a plate on its bottom, while the other patches simply act as open boundaries, representing the atmosphere. Axisymmetry is considered around the y -axis. All the dimensions are expressed as a function of the drop diameter, Fig. 5.15, where the dimensions assigned to the square domain were selected in order to ensure independence of the results on the open boundaries location.

The domain is meshed uniformly with 120 cells in both the radial and axial directions.

♥ Boundary conditions

At the plate, no-slip boundary conditions are imposed with a null velocity, linearly extrapolated polymeric extra-stresses and zero normal gradient for the pressure and the indicator field. The patches representing the open boundaries (atmosphere) are assumed to not interfere with the dynamics of the drop, so that zero-gradient is assumed for all variables, except the pressure, which is fixed $p = 0$.

Following ref. [23], the drop has an initial velocity U_0 , in the vertical direction, and its center of mass is at a distance $2D$ from the plate.

♥ Command-line

1–Build the mesh:

```
~$ blockMesh
```

2–Initialize the indicator and velocity fields in the drop region:

```
~$ cp 0/alpha1.org 0/alpha1
```

```
~$ cp 0/U.org 0/U
```

```
~$ setFields
```

3–Run the solver:

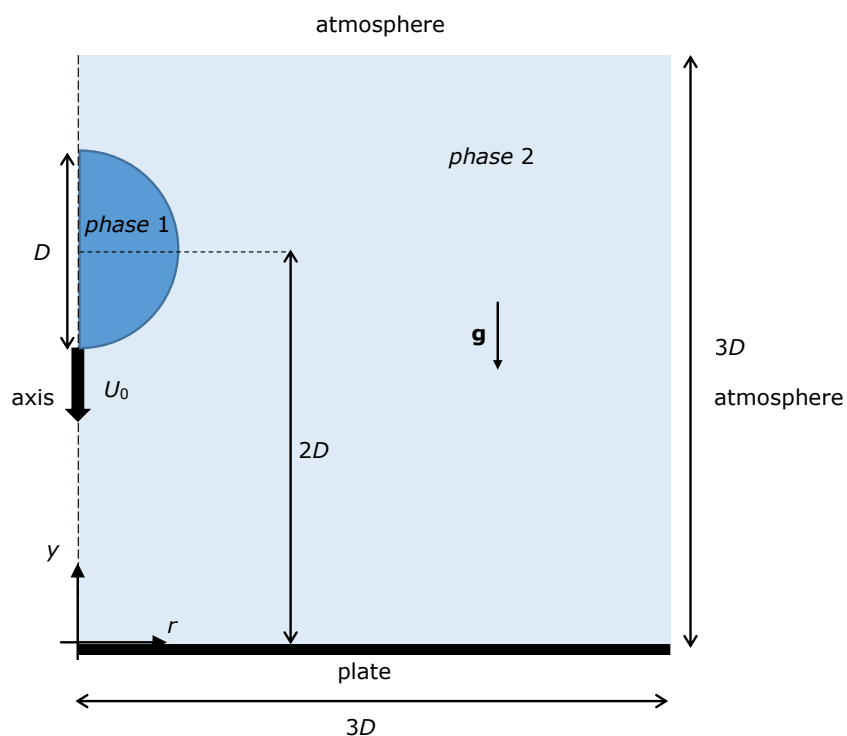


Figure 5.15: Geometry for the impacting drop problem.

~\$ rheoInterFoam

♥ Results

The evolution of the drop width over time is plotted in Fig. 5.16. The width is normalized with D (its initial diameter) and time with D/U_0 . The drop width is written to a file in the case directory, using a *coded FunctionObject*.

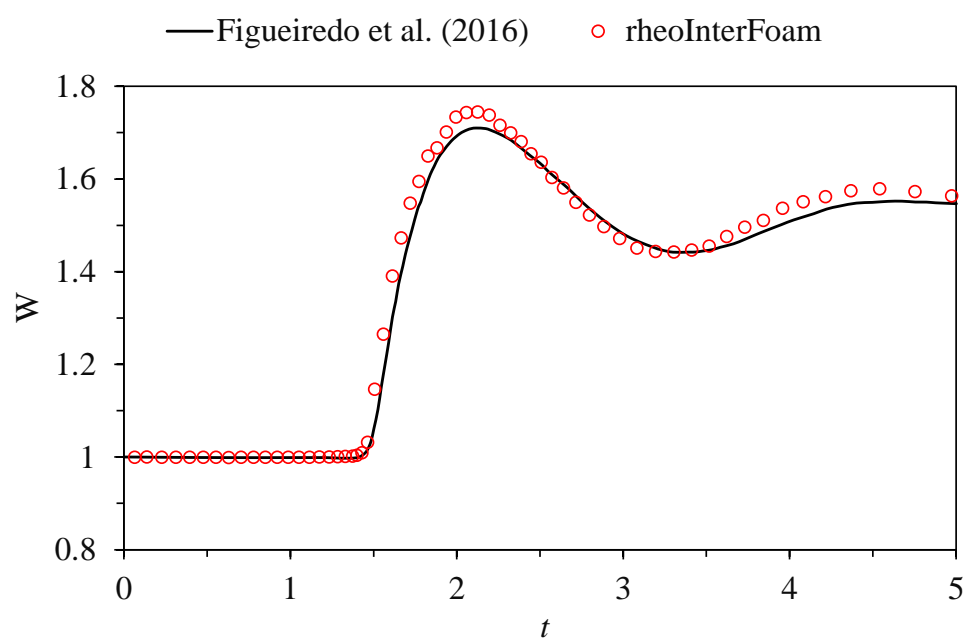


Figure 5.16: Evolution of the drop diameter over time for $Fr = 2.26$, $Re = 5$, $Wi = 1$ and $\beta = 0.1$. The profile obtained with *rheoInterFoam* is compared with the data in ref. [23].

Appendix A

Parameters and variables in *rheoTool*

Table A.1: List of some relevant parameters/variables used by *rheoTool* and correspondence with the nomenclature used in this guide.

Name in the guide	Name in the code	Dimensions [kg m s K mol A cd]	Definition
α	alpha	[0 0 0 0 0 0 0]	Mobility parameter of Giesekus model (scalar)
α	alpha	[0 0 0 0 0 0 0]	Parameter of the piecewise-linear HRS functions (scalar)
α	alpha	[0 0 0 0 0 0 0]	Indicator/Color function of the VOF method (scalar)
A	A	[0 0 0 0 0 0 0]	Variable of FENE-P model (scalar)
a	a	[0 0 0 0 0 0 0]	Dimensionless parameter of Carreau-Yasuda and White-Metzner models (scalar)
a	a	[0 0 0 0 0 0 0]	Variable of FENE-P model (scalar)
β	beta	[0 0 0 0 0 0 0]	Parameter of the piecewise-linear HRS functions (scalar)

b	b	[0 0 0 0 0 0 0]	Dimensionless parameter of White-Metzner model (scalar)
-	b	[0 0 0 0 0 0 0]	Square-root conformation tensor of ref. [9], for the <i>Oldroyd-BSqrt</i> model (symmTensor)
-	C	[0 0 0 0 0 0 0]	Passive scalar (scalar)
\mathbf{c}	-	[0 0 0 0 0 0 0]	Conformation tensor (symmTensor)
D	D	[0 2 -1 0 0 0 0]	Diffusion coefficient in the the passive scalar transport equation (scalar)
ε	epsilon	[0 0 0 0 0 0 0]	Extensibility parameter of sPTT-type models (scalar)
η	eta	[1 -1 -1 0 0 0 0]	Newtonian viscosity (scalar)
$\eta(\dot{\gamma})$	eta	[1 -1 -1 0 0 0 0]	Shear-rate dependent viscosity output by a GNF model (scalar)
η_0	eta0	[1 -1 -1 0 0 0 0]	Zero shear-rate viscosity (scalar)
η_0	eta0	[1 -1 -1 0 0 0 0]	Limiting viscosity in the Herschel-Bulkley model (scalar)
η_∞	etaInf	[1 -1 -1 0 0 0 0]	Infinite shear-rate viscosity (scalar)
η_{\max}	etaMax	[1 -1 -1 0 0 0 0]	Upper bound for the viscosity in the Power law model (scalar)
η_{\min}	etaMin	[1 -1 -1 0 0 0 0]	Lower bound for the viscosity in the Power law model (scalar)
η_p	etaP	[1 -1 -1 0 0 0 0]	Polymeric viscosity coefficient (scalar)
η_s	etaS	[1 -1 -1 0 0 0 0]	Solvent viscosity (scalar)
$\mathbf{\Lambda}$	eigVals	[0 0 0 0 0 0 0]	Eigenvalues obtained in the diagonalization of \mathbf{c} (tensor)

\mathbf{R}	eigVecs	[0 0 0 0 0 0 0]	Eigenvectors obtained in the diagonalization of $\mathbf{\Theta}$ and \mathbf{c} (tensor)
f	f	[0 0 0 0 0 0 0]	Variable of FENE-type models (scalar)
k	k	[1 -1 -1 0 0 0 0]	Consistency index (scalar)
k	k	[0 0 1 0 0 0 0]	Time-scale in the Carreau-Yasuda model (scalar)
K	K	[0 0 1 0 0 0 0]	Time-scale for η_p in the White-Metzner model (scalar)
λ	lambda	[0 0 1 0 0 0 0]	Relaxation time (scalar)
L	L	[0 0 1 0 0 0 0]	Time-scale for λ in the White-Metzner model (scalar)
L^2	L2	[0 0 0 0 0 0 0]	Extensibility parameter of FENE-type models (scalar)
m	m	[0 0 0 0 0 0 0]	Dimensionless parameter of White-Metzner model (scalar)
n	n	[0 0 0 0 0 0 0]	Behavior index for shear-rate dependent viscosity models (scalar)
$\mathbf{u} \cdot \mathbf{S}$	phi	[0 3 -1 0 0 0 0]	Face fluxes (scalar)
-	p	[1 -1 -2 0 0 0 0]	Modified pressure, in <i>rheoInterFoam</i> (scalar)
$\frac{p}{\rho}$	p	[0 2 -2 0 0 0 0]	Pressure divided by the density, in <i>rheoFoam</i> (scalar)
ρ	rho	[1 -3 0 0 0 0 0]	Density (scalar)
$\dot{\gamma}$	strainRate()	[0 0 -1 0 0 0 0]	Shear-rate (scalar)
$\boldsymbol{\tau}$	tau	[1 -1 -2 0 0 0 0]	Polymeric extra-stress tensor (symmTensor)
τ_0	tau0	[1 -1 -2 0 0 0 0]	Yield stress (scalar)

-	tauMF	[1 -1 -2 0 0 0 0]	Polymeric extra-stress tensor weighted by the indicator function in <i>rheoInterFoam</i> (symmTensor)
Θ	theta	[0 0 0 0 0 0 0]	Logarithm of the conformation tensor (symmTensor)
\mathbf{u}	U	[0 1 -1 0 0 0 0]	Velocity (vector)

Bibliography

- [1] J. Favero, A. Secchi, N. Cardozo, and H. Jasak, “Viscoelastic fluid analysis in internal and in free surface flows using the software OpenFOAM,” *Computers & Chemical Engineering*, vol. 34, no. 12, pp. 1984 – 1993, 2010. 10th International Symposium on Process Systems Engineering, Salvador, Bahia, Brasil, 16-20 August 2009.
- [2] F. Pimenta and M. Alves, “Stabilization of an open-source finite-volume solver for viscoelastic fluid flows,” *Journal of Non-Newtonian Fluid Mechanics*, vol. 239, pp. 85 – 104, 2017.
- [3] R. Fattal and R. Kupferman, “Constitutive laws for the matrix-logarithm of the conformation tensor,” *Journal of Non-Newtonian Fluid Mechanics*, vol. 123, no. 2–3, pp. 281 – 285, 2004.
- [4] R. Bird and O. Hassager, *Dynamics of Polymeric Liquids: Fluid mechanics*, vol. 1-2 of *Dynamics of Polymeric Liquids*. Wiley, 1987.
- [5] G. Guennebaud, B. Jacob, *et al.*, “Eigen v3.” <http://eigen.tuxfamily.org>, 2010.
- [6] X. Chen, H. Marschall, M. Schäfer, and D. Bothe, “A comparison of stabilisation approaches for finite-volume simulation of viscoelastic fluid flow,” *International Journal of Computational Fluid Dynamics*, vol. 27, no. 6-7, pp. 229–250, 2013.
- [7] R. Fattal and R. Kupferman, “Time-dependent simulation of viscoelastic flows at high weissenberg number using the log-conformation representation,” *Journal of Non-Newtonian Fluid Mechanics*, vol. 126, no. 1, pp. 23 – 37, 2005.
- [8] A. Afonso, F. Pinho, and M. Alves, “The kernel-conformation constitutive laws,” *Journal of Non-Newtonian Fluid Mechanics*, vol. 167–168, pp. 30 – 37, 2012.

- [9] N. Balci, B. Thomases, M. Renardy, and C. R. Doering, “Symmetric factorization of the conformation tensor in viscoelastic fluid models,” *Journal of Non-Newtonian Fluid Mechanics*, vol. 166, no. 11, pp. 546 – 553, 2011. XVIth International Workshop on Numerical Methods for Non-Newtonian Flows.
- [10] Ž. Tuković and H. Jasak, “A moving mesh finite volume interface tracking method for surface tension dominated interfacial fluid flow,” *Computers & Fluids*, vol. 55, pp. 70 – 84, 2012.
- [11] H. Jasak, “Pressure-velocity coupling in FOAM, Consistent derivation for steady and transient flow solvers.” OFW11, Guimarães, Portugal, 2016.
- [12] F. Moukalled, A. A. Aziz, and M. Darwish, “Performance comparison of the NWF and DC methods for implementing high-resolution schemes in a fully coupled incompressible flow solver,” *Applied Mathematics and Computation*, vol. 217, no. 11, pp. 5041 – 5054, 2011.
- [13] H. Jasak, H. Weller, and A. Gosman, “High resolution NVD differencing scheme for arbitrarily unstructured meshes,” *International Journal for Numerical Methods in Fluids*, vol. 31, no. 2, pp. 431 – 449, 1999.
- [14] P. J. Oliveira, “Asymmetric flows of viscoelastic fluids in symmetric planar expansion geometries,” *Journal of Non-Newtonian Fluid Mechanics*, vol. 114, no. 1, pp. 33 – 63, 2003.
- [15] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, “Numerical recipes (FORTRAN version),” *Press Syndicate of the University of Cambridge, Cambridge, UK*, 1989.
- [16] S. Xue and G. W. Barton, “An unstructured finite volume method for viscoelastic flow simulations with highly truncated domains,” *Journal of Non-Newtonian Fluid Mechanics*, vol. 233, pp. 48 – 60, 2016. Papers presented at the Rheology Symposium in honor of Prof. R. I. Tanner on the occasion of his 82nd birthday, in Vathi, Samos, Greece.
- [17] Y. Na and J. Y. Yoo, “A finite volume technique to simulate the flow of a viscoelastic fluid,” *Computational Mechanics*, vol. 8, no. 1, pp. 43–55, 1991.
- [18] M. A. Hulsen, R. Fattal, and R. Kupferman, “Flow of viscoelastic fluids past a cylinder at high weissenberg number: Stabilized simulations using matrix logarithms,” *Journal of Non-Newtonian Fluid Mechanics*, vol. 127, no. 1, pp. 27 – 39, 2005.

- [19] M. Alves, F. Pinho, and P. Oliveira, “The flow of viscoelastic fluids past a cylinder: finite-volume high-resolution methods,” *Journal of Non-Newtonian Fluid Mechanics*, vol. 97, no. 2–3, pp. 207 – 232, 2001.
- [20] F. Cruz, R. Poole, A. Afonso, F. Pinho, P. Oliveira, and M. Alves, “A new viscoelastic benchmark flow: Stationary bifurcation in a cross-slot,” *Journal of Non-Newtonian Fluid Mechanics*, vol. 214, pp. 57 – 68, 2014.
- [21] A. Valencia, A. Zarate, M. Galvez, and L. Badilla, “Non-newtonian blood flow dynamics in a right internal carotid artery with a saccular aneurysm,” *International Journal for Numerical Methods in Fluids*, vol. 50, no. 6, pp. 751–764, 2006.
- [22] Aneurisk-Team, “AneuriskWeb project website, <http://ecm2.mathcs.emory.edu/aneuriskweb>.” Web Site, 2012.
- [23] R. Figueiredo, C. Oishi, A. Afonso, I. Tasso, and J. Cuminato, “A two-phase solver for complex fluids: Studies of the weissenberg effect,” *International Journal of Multiphase Flow*, vol. 84, pp. 98 – 115, 2016.
- [24] C. Oishi, F. Martins, M. Tomé, and M. Alves, “Numerical simulation of drop impact and jet buckling problems using the extended pom–pom model,” *Journal of Non-Newtonian Fluid Mechanics*, vol. 169–170, pp. 91 – 103, 2012.