International Workshop on Coupled Methods in Numerical Dynamics
IUC, Dubrovnik, Croatia, September 19th-21st *2007*

OpenFOAM: A C++ Library for Complex Physics Simulations

Hrvoje Jasak[1,2], Aleksandar Jemcov[3], Željko Tuković[2]

[1] Wikki Ltd, United Kingdom
[2] Faculty of Mechanical Engineering and Naval Architecture, University of Zagreb, Croatia
[3] Development Department, Ansys/Fluent Inc, USA

Abstract. This paper describes the design of OpenFOAM, an object-oriented library for Computational Fluid Dynamics (CFD) and structural analysis. Efficient and flexible implementation of complex physical models in Continuum Mechanics is achieved by mimicking the form of partial differential equation in software. The library provides Finite Volume and Finite Element discretisation in operator form and with polyhedral mesh support, with relevant auxiliary tools and support for massively parallel computing. Functionality of OpenFOAM is illustrated on three levels: improvements in linear solver technology with CG-AMG solvers, LES data analysis using Proper Orthogonal Decomposition (POD) and a self-contained fluid-structure interaction solver.

Key words: *object oriented, C++, scientific computing, FSI, fluid-structure interaction, POD, proper orthogonal decomposition, iterative solver, CG-AMG*

1. Introduction

R.Landau book: 17.13

Expansion of Computational Continuum Mechanics (CCM) in engineering design is mirrored in the maturity of commercial simulation tools in the market. In terms of numerical techniques, structural analysis is dominated by the Finite Element Method (FEM), while fluid flow is regularly handled using the Finite Volume Method (FVM) of discretisation. Leading software combines accurate and robust numerics with an impressive range of physical models under a general-purpose Graphical User Interface (GUI). Current simulation challenges are related to integration and automation of simulation tools in a Computer Aided Engineering (CAE) environment, including automatic geometry retrieval, surface and volume meshing and sensitivity and optimisation studies.

In terms of solver settings and user expertise, Computational Fluid Dynamics (CFD) is considerably behind structural analysis, making the problems of

*Correspondence to: Hrvoje Jasak, E-mail: h.jasak@wikki.co.uk.

software development and usability more acute. Range and quality of physical models, solver settings and solution algorithms, as well as the lack of robust automatic solution control brings considerable complexity to the user.

Current state of solver development aims to produce monolithic general-purpose tools, trying to tackle all physical problems for all users. A number of consequences arises:

"against monolitic codes"

- Simulation software tends to be exceedingly complex due to interaction between numerous physical models, solution strategies and solver settings. This leads to development bottle-necks and difficulties in testing and validation – with increased code maturity and expanding capability matrix, code development grinds to a halt;

- While it is possible to cater for many "typical" combinations of physical models, user requirements are more general: they may involve experimental material properties, additional equations in the system or coupling with multiple external packages into simulation networks. To answer such needs, user-defined extensions to the built-in functionality are a must. For convenient and efficient user extensions, software must open its architecture to a certain level;

- Monolithic software necessarily implies that for any set of physics only a small subset of functionality is being used. The impact of unused or incompatible model combinations remains, typically in unnecessary memory usage;

- A drawback of monolithic tools is tendency to use identical discretisation and numerics even when they are clearly sub-optimal, simply because they "fit into the framework";

- Economies of scale and price/performance for each new feature dictate which new models will be implemented, with a preference for established physics and recognised customer needs.

very broad

In spite of best efforts, all CCM needs will never be catered for in this manner. This is particularly clear when one examines the true scope of continuum modelling, including non-traditional simulations, *e.g.* electromagnetics, magneto-hydrodynamics; coupled simulations like 1-D/3-D network simulations and fluid-structure interaction [1, 2] and even more general complex transport/stress models [3, 4].

The limiting factor in such cases is not lack of industrial interest or physical or numerical understanding, but a combination of complex software and closed architecture. At the same time, one should appreciate that a bulk of physical modelling, discretisation, numerics and linear solver expertise implemented in commercial CFD is a product of academic research available in open literature. The software provides an efficient solution framework, including geometry handling, mesh generation, solution, post-processing and data analysis, while

implementing known numerics and physical models. Furthermore, software engineering issues like computational efficiency, high performance computing and user support are handled in a centralised manner, reinforcing the complexity and rigidity of the system. As an illustration, while all commercial CFD codes contain efficient linear equation solvers, it is virtually impossible to re-use this component as a stand-alone tool.

In this paper we shall examine how modern programming techniques impact the state-of-the-art in CFD software, extending the reach of continuum modelling beyond its current capabilities. The software of which this study is based is OpenFOAM [5, 6, 7], an Open Source [8] object-oriented library for numerical simulations in continuum mechanics written in the C++ programming language [9, 10]. OpenFOAM is gaining considerable popularity in academic research and among industrial users, both as a research platform and a black-box CFD and structural analysis solver. Main ingredients of its design are:

- Expressive and versatile syntax, allowing easy implementation of complex physical model;
- Extensive capabilities, including wealth of physical modelling, accurate and robust discretisation and complex geometry handling, to the level present in commercial CFD;
- Open architecture and open source development, where complete source code is available to all users for customisation and extension at no cost.

In what follows, we shall start with some basic principles of Object Orientation (OO) in Section 2.. Section 3. presents five main objects of the OpenFOAM class hierarchy and their interaction. Section 4. shows how field algebra implemented in OpenFOAM can be used in complex post-processing on the example of Proper Orthogonal Decomposition (POD). The paper continues with illustrative simulation examples in Section 5., including Large Eddy Simulation (LES), linear equation solver tests, POD post-processing of LES data and fluid-structure interaction using OpenFOAM and is closed with a summary, Section 6.

2. Object Orientation in Numerical Simulation Software

Complexity of monolithic functional software stems from its data organisation model: globally accessible data is operated on by a set of functions to achieve a goal. Here, each added feature interacts with all other parts of the code, potentially introducing new defects (bugs) – with the growing size, the data management and code validation problem necessarily grows out of control.

Object orientation attempts to resolve the complexity with a "divide and conquer" approach, using the techniques listed below.

Data Encapsulation. Object-oriented software design [10] aims to break the complexity by grouping data and functions together and protecting the data from accidental corruption through compilation errors. The driving idea is to

recognise self-contained objects in the problem and place parts of implementation into self-contained types (classes) to be used in building the complexity. A C++ *class* consists of:

- A *public interface*, providing the capability to the user;
- Some *private data*, as needed to provide functionality.

As an example, consider a sparse matrix class. It will store matrix coefficients in its preferred manner (private data) and provide manipulation functions, *e.g.* matrix transpose, matrix algebra (addition, subtraction, multiplication by a scalar *etc.*). Each of these operates on private data in a controlled manner but its internal implementation details are formally independent of its interface.

Operator Overloading. Classes introduce new user-defined type into a problem description language, which encapsulate higher-level functionality in a way which is easy to understand. For clarity, the "look and feel" of operations on new objects needs to closely resemble the existing software. Thus, matrix algebra should have the same syntax as its floating point counterpart: `A + B`, where `A` and `B` are matrices represents a summation operation, just as it does in `a + b` for floating point numbers.

Syntactically, this leads to a set of identical function names with different arguments. Resolving the "correct" functional type from function arguments from the available set is termed *operator overloading*, automatically executed by the C++ compiler.

Object Families and Run-Time Selection. In many situations, one can recognise sets of object that perform the same function, based on identical data. As an example, consider a Gauss-Seidel linear equation solver and a preconditioned Conjugate Gradient solver. Both operate on a linear system $[A][x] = [b]$ to reach its solution $[x]$. Each algorithm will have its own controls (number of sweeps, solution tolerance, type of preconditioner) and different efficiency, but functionally they play the same role in the software.

In object orientation, such cases form *class families*. In C++, they are encoded by *class derivation* and *virtual functions*. Here, the software recognises a common interface and interchangeable functionality of family members and implements them in isolation of each other, further breaking down the complexity. Similar situation in various guises appears throughout the numerical simulation software: consider multiple convection differencing schemes, gradient calculation algorithms, boundary conditions, turbulence models *etc.*

Generic Programming. In some cases, software operations are independent of the actual data type. Consider the example of sorting a list of data using a bubble-sort algorithm: sequence of operations is independent of container content, provided a `compare` and `swap` function is available. The power of generic programming is in the fact that identical code can be used for numerous data

types and can be validated only once. In practice, the C++ compiler operates on the generic code to automatically generate and optimise type-specific implementation.

In numerics, examples of generic programming appear throughout the software. Consider for example, a Dirichlet boundary condition: its functionality is the same, no matter whether it operates on a scalar, vector or tensor field. The same is true for convective and diffusive transport, gradient schemes, field representation (independent of the mesh location and algebraic type) and many others.

Combination of the above makes a strong baseline for efficient and versatile software. It remains to be seen how objects are defined and how they interact with each other in working software.

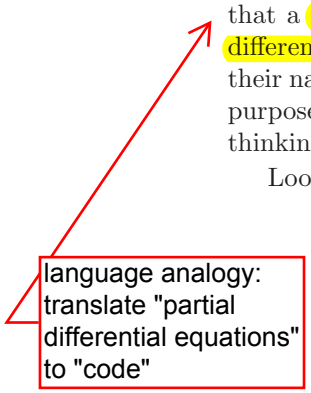
Some Common Myths: How Fast is Your C++? In late 1990s, a discussion on computational overheads of object orientation and implications for "object-oriented numerical simulation software" raged in the scientific community. The basic argument was that code execution overheads associated with new programming techniques make C++ prohibitively expensive for numerically intensive codes.

Over the last 10 years, the exact opposite has been emphatically proven. High-level language built through object orientation allow the software developer to split the problem into several levels and concentrate on efficiency where it should naturally be resolved. Common and well-known rules apply: low-level data layout must follow the computer architecture (packed arrays), loop optimisation is achieved by avoiding forking and function calls, vector- and pipeline instruction and data organisation is enforced. If anything, a high-level type-safe language like C++ makes this job easier: the user of generic container classes can accept optimisation at zero cost as changes appear only in low-level object. Combining this with `inline` instructions in C++ and careful use of virtual functions [10] results in the code which at worst matches the performance of procedural languages like Fortran and C but in a more convenient interface. Ultimately, it is the skill of the programmer and not the programming language that drives the efficiency, which is probably more true in a complex and expressive language like C++.

3. Five Main Objects in OpenFOAM

With the aim of writing a general-purpose CCM simulation tool, we can state that a natural language of continuum mechanics already exists: it is a partial differential equation. Therefore, attempting to represent differential equations in their natural language in software is a reasonable development goal for a general-purpose CCM tool in an object oriented framework. This is the fundamental thinking behind the design of OpenFOAM.

Looking at the example of a turbulence kinetic energy equation in Reynolds



example translation

Averaged Navier-Stokes (RANS) models:

$$\frac{\partial k}{\partial t} + \nabla\bullet(\mathbf{u}\,k) - \nabla\bullet[(\nu + \nu_t)\nabla k] = \nu_t \left[\frac{1}{2}(\nabla\mathbf{u} + \nabla\mathbf{u}^T)\right]^2 - \frac{\epsilon_o}{k_o}\,k, \qquad (1)$$

we shall follow the path to its encoded version in OpenFOAM, Fig. 1.

```
solve
(
   fvm::ddt(k)
 + fvm::div(phi, k)
 - fvm::laplacian(nu() + nut, k)
== nut*magSqr(symm(fvc::grad(U)))
 - fvm::Sp(epsilon/k, k)
);
```

Figure 1. C++ representation of the k-equation, Eqn. (1), in OpenFOAM.

Correspondence between Eqn. (1) and Fig. 1. is clear, even with limited programming knowledge and without reference to object-orientation or C++.

OpenFOAM is a substantial software package with extensive capabilities; analysing it in detail is beyond the scope of this study. In what follows, we shall work on recognising the main objects from the numerical modelling viewpoint appearing in Fig. 1. and encoding them in software.

... and this course ...

3.1. Space and Time

The first point of order in field-based discretisation is to capture the space and time relevant for the simulation. The space is captured as a computational mesh, consisting of a number of non-overlapping elements, while the temporal dimension is split into a finite number of time-steps.

Computational Mesh. Objects needed to describe a computational mesh are points, faces, cells and boundary patches. OpenFOAM implements *polyhedral mesh handling* [11], where a cell is described as a list of faces closing its volume, a face is an ordered list of point labels and points are gathered into an ordered list of (x, y, z) locations. Additionally, boundary faces of a mesh are grouped into *patches* to simplify the specification of boundary conditions.

Recognising objects seems trivial: a `point` or a `vector`, consisting of three floating point values, a `face` and a `cell`. Functionality of each and their private data is also clear: examples would include functions returning a face normal vector or a cell volume.

Furthering the capabilities of the mesh class, there also exists a high-level mesh-related functionality that "belong" to the mesh class. For example, auto-

matic mesh motion, topological mesh changes (*e.g.* sliding interfaces, cell layering *etc.*) and adaptive mesh refinement perform the same function for both the FVM and FEM. Implementing them centrally significantly improves the software design and promotes code re-use.

Time Acting as Object Database. In the temporal dimension, things are simpler: it is sufficient to track the time step count and time increment Δt. Further to this, there is a set of database operations associated with time-marching and with related needs. For example, simulation data output should be performed every n time-steps or x seconds of computational time, solver controls needs to be updated in intervals *etc.* The `time` class in OpenFOAM handles both functions.

3.2. Field Variable

Continuum mechanics operates on fields of scalars, vectors and tensors, each of which is numerically approximated as a list of typed values at pre-defined points of the mesh. The first set of objects involves tensorial types with associated algebra, typically consisting of fixed-length containers. Thus, a `vector` class consists of three floating point numbers and vector operations: addition subtraction, scalar multiplication, magnitude, dot- and cross-products *etc.* Arbitrary rank tensor classes are defined in the same manner.

Field: List with Algebra. In field operations, one regularly operates on collections of identical types, performing the same function. Thus, a field-based dot-product of **a** and **b** involves looping over all elements in a list and storing the result in another list. A `Field` class uses short-hand for implied looping operations: for vector fields `a` and `b`:

```
scalarField c = a & b;
```

will perform the looping, calculate the dot-product and store the result into a new field object called `c`.

Patch Field: Boundary Condition. To describe a boundary condition, a `Field` class needs to carry *behaviour* in addition to its values. For example, a `fixedValue` field carries its values but shall not change on assignment: its value is fixed. Some other cases, like a `fixedGradient` field class can "evaluate" boundary values, given the internal field and a surface-normal gradient.

Note that a collection of patch fields of different types constitutes a family of related classes: each calculates its value based on behaviour, but does the job in its own specific way.

Geometric Field. While automated field algebra is convenient shorthand, a list of values does not provide the complete picture. For visualisation or discretisation, a `Field` needs to be given spatial dependence: how are the values arranged in space and what property they describe. This is encoded by association with

only the list of values/vectors/tensors

a mesh, boundary condition specification (patch fields) and physical dimension set. A combination of the three constitutes a generic `GeometricField`.

Looking at its structure, it quickly becomes clear that field algebra functions follow the same pattern no matter if the field consists of scalar, vector or tensor values or if it is stored on mesh points or cell centres. For further convenience, all field operations are dimensionally checked: summation of a velocity field (m/s) and a temperature gradient field (K/s) is easily flagged as invalid.

3.3. Matrix and Linear System

Functionality and interface of sparse matrix class, a linear system:

$$[A][x] = [b] \tag{2}$$

and associated linear solvers is clear. It suffices to say that code organisation as presented above allows the FEM and FVM discretisation to share sparse matrix implementation and solver technology, resulting in considerable code re-use. True benefit of such code organisation will become obvious when attempting closely coupled simulation: sharing matrix format and implementation between coupled solvers is extremely useful.

3.4. Discretisation Method

Software functionality described above makes no reference to the type or order of discretisation to be implemented. This indicates the classes mentioned above can be re-used without change, with associated benefit of code compactness and modular validation.

Keeping in mind the code layout in Fig. 1. we shall handle the discretisation in OpenFOAM on a *per-operator* basis: if operators can be assembled at will, wealth of continuum mechanics equations can be tackled with the same toolkit. To simplify the discussion, we shall refer to the FVM discretisation as a specific example.

Interpolation, Calculus and Method. A discretisation method can be separated into three basic functions:

- *Interpolation*, which allows us to evaluate the field variable between computational points, based on prescribed spatial and temporal variation (shape function);

numerical derivation/operators

- *Differentiation*, where calculus operations are performed on fields to create new fields. For example, given a FVM pressure field `volScalarField p`,

  ```
  volVectorField g = fvc::grad(p);
  ```

  creates a new FVM vector field of pressure gradient. Its boundary conditions and cell values are calculated from `p`. Calculus operations in Fig. 1. carry the `fvc::` prefix, denoting Finite Volume Calculus;

FVM or FEM, etc

- *Discretisation* operates on differential operators (rate of change, convection, diffusion), and creates a discrete counterpart of the operator in sparse matrix form. Discretisation operators in Fig. 1. carry the `fvm::` prefix.

We can now decode the contents of Fig. 1. with some clarity:

- `k, epsilon` and `U` represent FVM fields of scalars and vectors, storing lists of values associated with a computational mesh. Each field carries its internal (cell) values of appropriate type (`scalar`, `vector`) and boundary conditions on a *per-patch* basis, combining face values and behaviour (Dirichlet, Neumann or mixed boundary condition);

- Field calculus operations are used to create complex field expressions, including field algebra (`nu + nut`) or calculus operations (`fvc::grad(U)`). This is used to assemble the source terms and auxiliary fields;

- Discretisation of each operator (*e.g.* `fvm::laplacian(mu() + nut, k)`) creates a sparse matrix, which are put together using matrix algebra operations to form a linear system of equations;

- The `solve` function chooses an appropriate linear equation solver based on the matrix structure, which evaluates `k` at computational points; this is followed by the boundary condition update, where the behaviour of each condition is dictated by its type.

The identical set of operations is performed in all CFD codes – the difference in OpenFOAM is the clarity and flexibility of its programming implementation, as shown in Fig. 1.

Looking at the high-level numerical simulation language we have assembled, implementation of various complex physical models now looks very simple. Fringe benefits from object orientation as implemented in OpenFOAM imply considerable code re-use (mesh, matrix, field, linear solver, boundary condition) and *layered development*. For example, internal implementation of a linear algebra module may be changed, with minimal impact on the rest of the software – an ideal software development situation.

3.5. Physical Modelling Libraries

Having recognised object families at discretisation level, the exercise can now be repeated in the physics modelling framework. For example, one could clearly see that all RANS turbulence models in effect provide the same functionality: evaluating the Reynolds stress term $\overline{\mathbf{u}'\mathbf{u}'}$ in the momentum equation. Grouping them together guarantees inter-changeability and decouples their implementation from the rest of the flow solver. In such situation, the momentum equation communicates with a turbulence model through a pre-defined interface (the model will contribute a discretisation matrix) and no special knowledge of a particular model is needed. In practical terms, a new turbulence model is implemented by creating a class capable of calculating $\overline{\mathbf{u}'\mathbf{u}'}$ in its own particular

way: standard $k-\epsilon$ model [12] and a Reynolds Stress Transport model [13] differ only in the way *how* they perform their function.

Similar modelling libraries appear throughout the code: Newtonian and non-Newtonian viscosity models, material property models, combustion models *etc.* Model families are grouped into libraries and re-used by all relevant applications.

3.6. Physics Solver

Simulation software described so far resembles a numerical tool-kit used to assemble various physics solvers. Physical modelling capabilities of OpenFOAM include a wide range of fluid and structural analysis models, with emphasis on coupled and non-linear equation sets. In all cases, base mesh handling, field algebra, discretisation, matrix and linear solver support *etc.* is shared.

To avoid unnecessary complexity at the top-level, each physics solver is implemented as a stand-alone tool, re-using library components as the need arises. Capability of such solvers is underpinned by a combination of complex geometry support and parallelisation. The first is supported by the mesh class, while the second remains encapsulated in the linear algebra functionality and discretisation method, with no impact at the top-level physics solver.

Extension to other areas of CCM follows naturally: solving structural analysis equations, including non-linear material properties is a simple modification on the equation set. Other non-standard simulation areas are tackled on a case-by-case basis, building a custom solver from the available tool-kit. A result of such development is a customised and optimised solver for each physics segment, which is both efficient and easy to understand.

4. Data Manipulation: Proper Orthogonal Decomposition

Quite apart from its advantages in complex physics modelling, the tool-kit approach encourages development of auxiliary CCM tools not present in closed-architecture packages or leading commercial software. A good example of this is Proper Orthogonal Decomposition (POD), described below.

Consistency of the mathematical definition of field algebra with its C++ implementation allows arbitrary manipulation of computed fields to produce new data fields. Performing algebraic operations such as addition and subtraction, scaling, scalar products, *etc.* on the result enables easy computations of derived quantities which aid in understanding the solution. In addition to visualising vector components, streamlines and similar, one can easily assemble a reduced order basis using the Proper Orthogonal Decomposition (POD) to represent coherent structures in fluid flow [14, 15].

POD, also known as Principal Component Analysis or Karhunen Loève Expansion, is based on the computation of the best linear basis that describes the dominant dynamics of the problem. Optimality of a POD basis is obtained from the fact that the basis vectors Φ are obtained by maximising the following cost

function:

$$\max_{\Psi} \frac{E\left(|(U,\Psi)|^2\right)}{(\Psi,\Psi)} = \frac{E\left(|(U,\Phi)|\right)}{(\Phi,\Phi)}. \tag{3}$$

Here, symbol $(\,\cdot\,,\,\cdot\,)$ represents the inner product in the L_2 space, and E is the mathematical expectation over a set of ensemble functions. This optimisation problem can be solved by variational methods and the solution is given by:

$$\int_0^{2\pi} R(x,x')\Phi(x')dx' = \lambda\Phi, \tag{4}$$

where

$$R(x,x') = E\left(U(x)U^*(x')\right), \tag{5}$$

where R is the auto-correlation function acting as a kernel of the integral equation in Eqn. (4).

Assuming that the ergodicity hypothesis holds [14], *method of snapshots* is used to compute empirical eigenvectors of the problem in Eqn. (4). Due to the ergodicity hypothesis, eigenvectors of the kernel $R\left(x,x'\right)$ can be represented through a linear combination of snapshots:

$$\Phi = \sum_{i=1}^{M} \beta_i U_i. \tag{6}$$

Coefficients β_i satisfy the eigen-problem:

$$R\beta = \Lambda\beta, \tag{7}$$

where R is obtained by a simple assembly averaging:

$$R = \lim_{M\to\infty} \frac{1}{M} \sum_{i=1}^{M} U_i(x)U_i(x'). \tag{8}$$

Eigenvectors of the auto-correlation R define the POD basis since they are ortho-normal and individual components of the eigenvalue vector Λ define the importance of associated eigenvectors. If the eigenvalues are interpreted as the mean flow energy of the field $U(x,t)$ projected to a subspace spanned by the POD vectors, the relative energy in the i-th basis vector is measured by:

$$e_r = \frac{\lambda_i}{\sum_{k=1}^{M} \lambda_k}. \tag{9}$$

The relative energy e_r is used in the thresholding process by which eigenvectors that represent negligible fraction of the total energy are eliminated, yielding the truncated representation of the flow field:

$$U \approx \sum_{i=1}^{N} \alpha_i \Phi_i, \tag{10}$$

where $N \ll M$.

The newly computed basis in Eqn. (10) is used to represent the flow field in the POD basis. Since the POD eigenvectors are directly related to the energy, and assuming the truncation in Eqn. (10) is performed so that 99% percent (or more) of the flow energy is retained, eigenvectors Φ represent the coherent structures in flow field.

Coherent structures are the representation of the correlation of the given field measured in energy norm, showing important spatial features during the time period in which the snapshots were taken. This fact allows a new look into time dependent computation results by revealing a sequence of coherent structures in the flow that are not available if time averaging is used instead. This is of great importance in DNS and LES data, where so far only time averaging was used to compute correlations. Furthermore, since the POD basis represents the optimal linear basis in which nonlinear problem can be represented, this procedure is quite useful in producing reduced order models of the given problem through Galerkin projection [15].

Advantage of OpenFOAM in the creation of a POD basis for any given problem is the existence of the field algebra that allows consistent implementation of operations described above. Moreover, with the field algebra POD analysis becomes an integral part of the library, removing the need for separate POD development for a specific physics model.

5. Simulation Examples

In what follows we shall illustrate the capability of OpenFOAM at four levels:

- A high-performance linear algebra package, where implementation of new linear equation solvers bring substantial benefit in real simulations;
- A ready-to-use fluid flow solver for complex physics;
- A library of flexible post-processing and data manipulation tools;
- A flexible and efficient development platform for complex and coupled physics simulations.

Results summarised in this section have been reported in other topical publications – the reader is encouraged to consult the references for a more complete description of methodology and results.

5.1. LES over a Forward-Facing Step

The first two sets of results are related to a LES simulation over a forward-facing step at $Re = 10\,000$, Fig. 2. The actual LES performance of OpenFOAM has been reported in multiple publications and is not of particular interest.

Two computational meshes are used. A coarse mesh with 660 000 cells is used in single-processor studies, while a fine mesh consisting of 5 280 000 cells is used

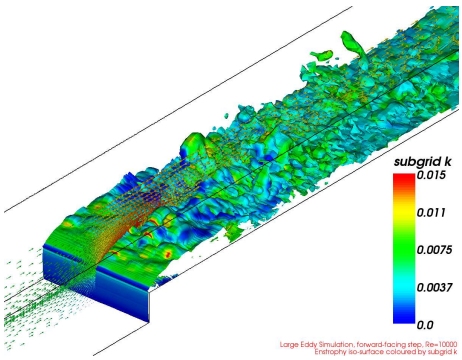


Figure 2. Enstrophy iso-surface coloured by subgrid turbulence kinetic energy for turbulent flow over a forward-facing step.

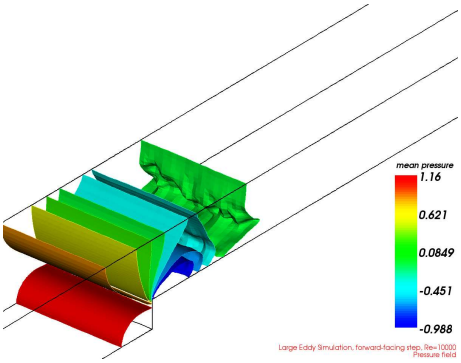


Figure 3. Time-averaged pressure field for turbulent flow over a forward-facing step.

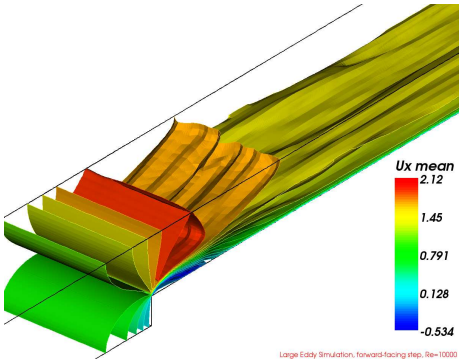


Figure 4. Time-averaged u_x component of velocity for turbulent flow over a forward-facing step.

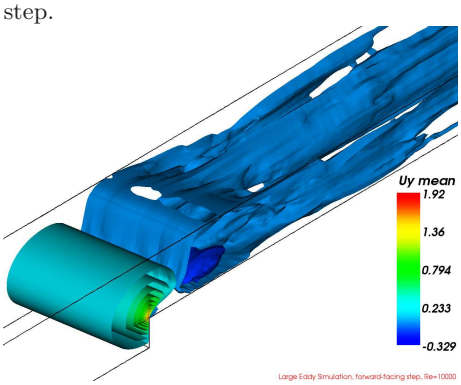


Figure 5. Time-averaged u_y component of velocity for turbulent flow over a forward-facing step.

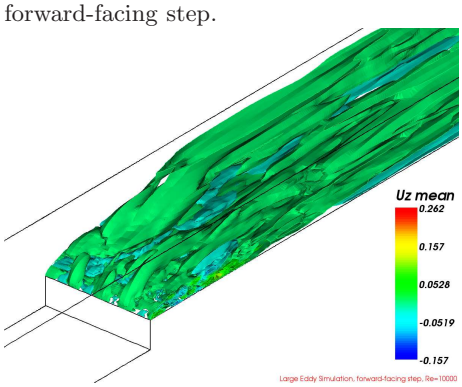


Figure 6. Time-averaged u_z component of velocity for turbulent flow over a forward-facing step.

to evaluate parallel performance. In both cases, the mesh is aggressively graded towards the wall. Sub-grid turbulence is modelled using a 1-equation SGS model [16]. Second-order implicit spatial discretisation is used in conjunction with a second-order temporal scheme. Maximum Courant number is held at unity and 2 PISO correctors are used with full convergence on the pressure equation.

A LES simulation produces a wealth of data, given the fact that $11,000$ time steps were used to simulate the flow. A snapshot of an instantaneous flow field is shown in Fig. 2., represented by enstrophy iso-surface (vorticity magnitude), coloured by the sub-grid turbulence kinetic energy and a centre-line instantaneous velocity slice. Traditional time averaging of velocity and pressure fields provide an insight into a spatial distribution of mean fields as shown in Figs. 3., 4., 5. and 6.

5.2. Linear Solver Performance: CG-AMG Solver

Efficient solution of linear systems of equations stemming from cell-centred Finite Volume Discretisation is critical for computational performance of modern CFD solvers. A good software package will spend 50-80% of execution time in linear solvers, making the matrix inversion a natural place to seek improvements [17, 18]. Two classes of iterative linear equation solvers are traditionally used, namely, Algebraic Multigrid (AMG) and Krylov space methods [19].

AMG relies on rapid removal of high-frequency error in stationary iterative methods and accelerates the removal of smooth error components. This is achieved by creating a hierarchy of progressively coarser linear equation sets by restricting the fine matrix by agglomeration or filtering [20]. Error restriction transforms the low-frequency fine level error into a high-frequency coarse error, where it can be removed efficiently. After smoothing, the coarse level correction is prolongated to the fine level, accelerating the solution. In practice, multiple coarse matrices are created, each responsible for a certain part of the error spectrum. Two single level smoothers has been tested: a symmetric Gauss-Seidel sweep and Incomplete Lower-Upper decomposition with zero fill-in (ILU).

In contrast to stationary iterative methods, Krylov space methods are characterised by iteration-dependent parameters and non-linear convergence behaviour and use preconditioning techniques to achieve convergence. The Krylov space family encompasses a number of solvers, the most notable being Conjugate Gradient (CG), Bi-Conjugate Gradient (BiCG), Generalised Minimum Residual (GMRES) and similar [19]. The idea is that a combination of multigrid and Krylov space method, with a strong smoother may result in an extremely efficient linear equation solver and will be tested on the solution of the LES pressure equation from Fig. 2.

Table 1. summaries the performance of top 5 fastest solvers, including the timing for the first pressure solution and total time per time-step. First 4 positions in ranking belong to CG-AMG, with the fastest solver giving a factor of 2.66 reduction in time per time-step over ICCG. This is in line with "raw" solver speed-up of a factor of 4, since most of the remaining cost is fixed.

	Solver Type	p-Eqn	Time-step
1	CG-AAMG 4 W, ILU 0/2	11.16 s	28.39 s
2	CG-AAMG 4 W, SGS 0/2	13.63 s	31.97 s
3	CG-AAMG 4 W, ILU 2/2	17.29 s	38.18 s
4	CG-SAMG 4 W, ILU 0/2	17.75 s	40.34 s
5	SAMG, V SGS 0/2	18.04 s	40.21 s
	AAMG, 4 W, ILU 2/2	25.82 s	49.42 s
	RRE AMG	14.40 s	34.69 s
	PFE AMG	14.58 s	33.19 s
	MPE AMG	15.94 s	34.34 s
	ICCG	44.41 s	75.62 s
	AMG	54.17 s	93.97 s

Table 1. Coarse mesh solver performance.

ID	Solver	Coarsener	Cycle	Smoother
1	CG-AMG	AAMG, 4	W	SGS 2/2
2	CG-AMG	AAMG, 4	W	ILU 0/2
3	PFEAMG	AAMG, 4	W	ILU 0/2
4	AMG	AAMG, 4	W	SGS 0/2
5	AMG	AAMG, 4	W	ILU 0/2
6	RREAMG	AAMG, 4	W	ILU 0/2
7	AMG	AAMG, 2	V	SGS 0/2
8	ICCG			

Table 2. Solver settings in parallel performance test.

	1 CPU		2 CPUs		4 CPUs	
ID	Iter	Time, s	Iter	Time, s	Iter	Time, s
1	22	357.0	21	177.9	22	71.2
2	16	233.1	21	160.7	21	91.7
3	27	374.5	30	220.9	28	117.6
4	52	495.3	45	229.0	40	120.8
5	35	480.2	36	261.3	35	145.9
6	23	344.1	35	276.8	38	172.6
7	97	849.9	129	575.0	109	254.9
8	884	2571.3	913	1385.8	970	502.6

Table 3. Fine mesh parallel solver performance.

The lower part of the table includes the performance of some other accelerated AMG solvers, namely Projective Forward Extrapolation (PFEAMG), Minimal Polynomial Extrapolation (MPEAMG) and Reduced Rank Extrapolation (RREAMG), [18]. While the RREAMG solver produces substantial performance improvement over standard AMG, it does so at considerable cost: the algorithm allocates storage for 5-10 additional vectors of the size of the solution. The CG-AMG solver is still substantially faster than the fastest vector sequence extrapolation result, without the need for additional storage.

Table 2. lists the fastest linear solver setup for the fine mesh parallel test for a serial, 2- and 4-CPU parallel run. Variation in performance is largest for ICCG, but some parallel degradation is present for most solvers.

5.3. Proper Orthogonal Decomposition of Instantaneous LES Data

Large Eddy Simulation results are usually analysed by collecting time dependent data and performing time averaging to recover mean fields. In this way, statistics of the flow field can be recovered in terms of mean values of the variables and is consistent with the traditional view that required computation of time-averaged data. This approach is in line with the traditional view of turbulence modelling through Reynolds averaged equations.

POD allows a different view of the LES data shown in Figs. 2., 3., 4., 5. and 6. In the case of the time dependent data, POD can be deployed for identification of large coherent structures by performing change of basis as described in Eqn. (10). Figs. 7., 8., 9. and 10. show the most energetic mode for the pressure and components of velocity fields. Comparison between time-averaged and POD decomposed fields reveals that POD modes contains more structure in contrast to time-averaged data which looks smoother. The smooth appearance of time-averaged data is a consequence of the averaging process and is capable of representing only the mean structures in a given time interval. On the other hand, POD data has a more dynamical structure since it represents actual structures in the flow field at a given energy threshold level. This capability of POD modes stems from the use of the auto-correlation function of the kernel in expression in Eqn. (4), whose eigenvectors are shown in Fig. 7. through Fig. 10..

POD decomposed and time-averaged data represent two complementary approaches, leading to a better understanding of the flow features in time dependent simulations. Both approaches represent manipulations of the instantaneous fields with clearly defined mathematical operations consistently implemented in OpenFOAM. Therefore, switching from a time-averaged to a POD view amounts to the selection of the mathematical operations to be performed on the data fields through built-in operators in field algebra.

5.4. Coupled Simulations: Fluid-Structure Interaction

Most Fluid-Structure Interaction (FSI) simulations today involve a combination of solvers, usually with a Finite Volume (FV) solver for the fluid flow, a Finite Element (FE) solver for the structural analysis and a third code for cou-

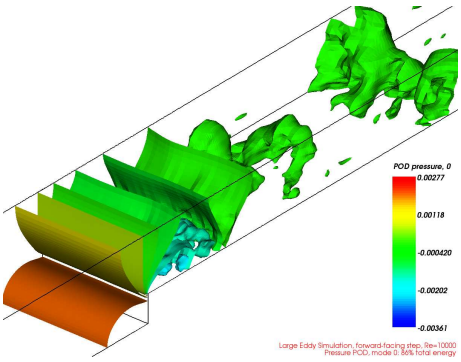


Figure 7. The first POD mode of pressure field for turbulent flow over a forward-facing step.

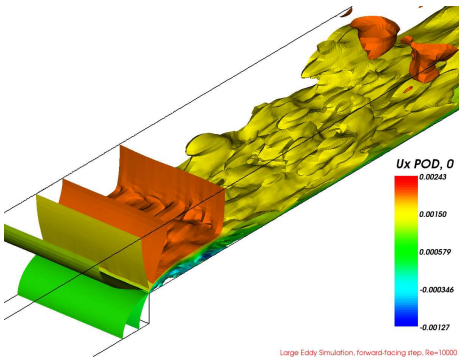


Figure 8. The first POD mode of u_x field for turbulent flow over a forward-facing step.

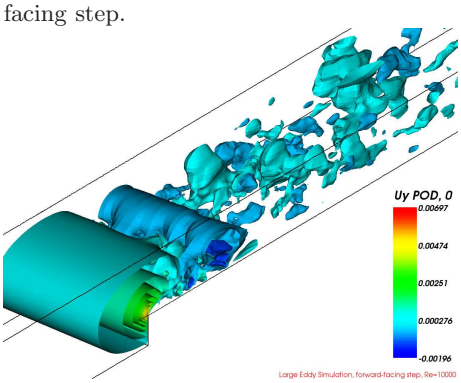


Figure 9. The first POD mode of u_y field for turbulent flow over a forward-facing step.

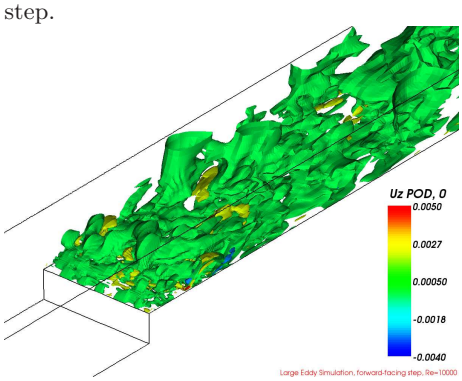


Figure 10. The first POD mode of u_z field for turbulent flow over a forward-facing step.

pling, data interpolation and simulation management. This approach imposes limitations on the mode of coupling and creates problems in model setup.

In this example, OpenFOAM is used to build a self-contained FSI solver, simulating the interaction between an incompressible Newtonian fluid and a St. Venant-Kirchhoff solid. Fluid flow is modelled by the Navier-Stokes equations in an Arbitrary Lagrangian-Eulerian (ALE) formulation, while the large deformation of the solid is described by the geometrically nonlinear momentum equation in an updated Lagrangian formulation.

Spatial discretisation of both models is performed using second-order accurate unstructured FVM, where the fluid model is discretised on the moving mesh [21], while the solid model is discretised on the fixed mesh, updating its configuration in accordance with the displacement from the previous time step. Automatic mesh motion solver [22, 23] is used to accommodate fluid-solid interface deformation. Temporal discretisation for the fluid and solid models is performed

using a fully implicit second-order accurate three-time-levels difference scheme.

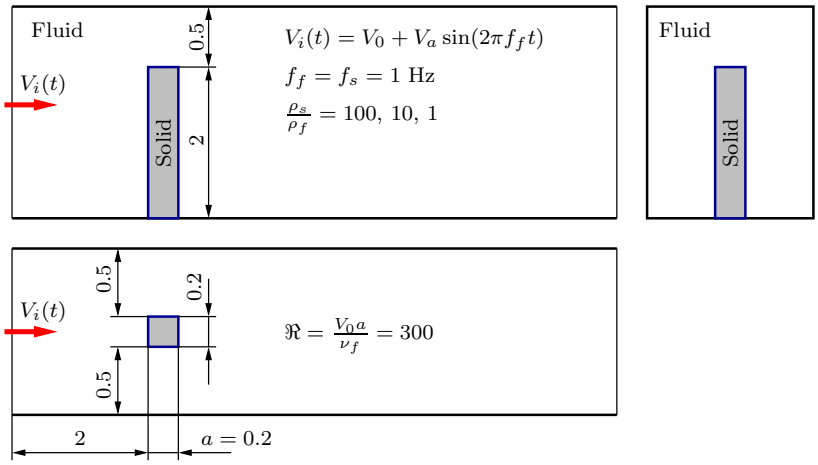


Figure 11. Flow-induced vibration: case setup.

Coupling between the two models is performed using a loosely-coupled staggered solution algorithm, where the force is transfered in one direction and displacement in the opposite. OpenFOAM also provides surface mapping tools used for data mapping between the surfaces, either directly or with second-order interpolation.

The FSI application is tested on the flow past a cantilevered elastic square beam with an aspect ratio of 10 at the Reynolds number 300 in a pulsating velocity field. Test setup is shown in Fig. 11. Ratio between fluid density and Young's modulus of the solid is set to match the first natural frequency of the beam with the frequency of the pulsating velocity field. A snapshot of the solution during the oscillation is shown in Fig. 12.

Tha main advantage of OpenFOAM over traditional FSI implementations lies in the fact that a self-contained executable is easier to manage. Furthermore, shared discretisation, mesh handling and linear solver support allows us to examine matrix-level and even continuum model-level coupling [24], which would otherwise be impossible.

6. Summary and Future Work

In this paper, we have described class layout of OpenFOAM, an object-oriented package for numerical simulation in Continuum Mechanics in C++. On the software engineering side, its advantage over monolithic functional approach is in its modularity and flexibility.

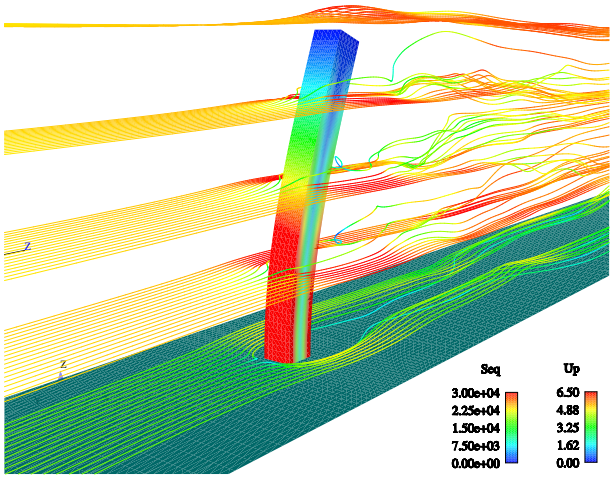


Figure 12. Cantilevered beam vibration in an oscillating flow field.

Object orientation breaks the complexity by building individual software components (classes) which group data and functions together and protect the data from accidental corruption. Components are built in families and hierarchies, where simpler classes are used to build more complex ones. A toolkit approach implemented in OpenFOAM allows the user to easily and reliably tackle complex physical models in software.

Performance and flexibility of OpenFOAM is illustrated on three examples, including improvements of linear solver technology, analysis of LES results using Proper Orthogonal Decomposition (POD) and on a fluid-structure interaction problem.

References

[1] A. K. Slone, K. Pericleous, C. Bailey, M. Cross, and C Bennett. A finite volume unstructured mesh approach to dynamic fluid-structure interaction: an assessment of the challenge of predicting the onset of flutter. *Applied mathematical modelling*, 28:211–239, 2004.

[2] W. A. Wall, C. Forster, M. Neumann, and E. Ramm. Advances in fluid-structure interaction. In K Gurlebeck and C. Konke, editors, *Proceedings of 17th international conference on the application of computer science and mathematics in architecture and civil engineering*, Weimar, Germany, 2006.

[3] M. Lostie, R. Peczalski, and J. Andrieu. Lumped model for sponge cake baking during the crust and crumb period. *J. food eng.*, 65(2):281–286, 2004.

[4] G. Franco. Development of a numerical code for the simulation of the process of refractory concrete drying. Master's thesis, Universitá degli studi di Trieste, Faculty of Engineering, 2005.

[5] OpenFOAM project web pages. http://www.openfoam.org, 2004.

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

[7] H. Jasak. Multi-physics simulations in continuum mechanics. In *Proceedings of 5th International Conference of Croatian Society of Mechanics, Trogir*, page ?? Croatian Society of Mechanics, September 2006.

[8] Open source initiative web pages. http://www.opensource.org, 1987.

[9] Programming languages – C++. ISO/IEC Standard 14822:1998, 1998.

[10] B. Stroustrup. *The C++ programming language*. Addison-Wesley, 3rd edition, 1997.

[11] H. Jasak. *Error analysis and estimation in the Finite Volume method with applications to fluid flows*. PhD thesis, Imperial College, University of London, 1996.

[12] B.E. Launder and D.B. Spalding. The numerical computation of turbulent flows. *Comp. Meth. Appl. Mech. Engineering*, 3:269–289, 1974.

[13] M.M. Gibson and B.E. Launder. Ground effects on pressure fluctuations in the atmospheric boundary layer. *J. Fluid Mechanics*, 86:491, 1978.

[14] L. Syrovich. Turbulence and and the dynamics of coherent structures. *Quart. Applied Math*, XLV(3):561–582, 1987.

[15] P. Holmes, J. L. Lumely, and G. Berkooz. *Turbulence, Coherent Structures, Dynamical Systems and Symmetry*. Cambridge University Press, Cambridge, MA, USA, 1996.

[16] C. Fureby, G. Tabor, H.G. Weller, and A.D. Gosman. A comparative study of subgrid scale models in homogeneous isotropic turbulence. *Phys. Fluids*, 9(5):1416–1429, May 1997.

[17] A. Jemcov, J.P. Maruszewski, and H. Jasak. Performance improvement of algebraic multigrid solver by vector sequence extrapolation. In *CFD 2007 Conference, CFD Society of Canada*, 2007.

[18] H. Jasak, A. Jemcov, and J.P. Maruszewski. Preconditioned linear solvers for large eddy simulation. In *CFD 2007 Conference, CFD Society of Canada*, 2007.

[19] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial & Applied Mathematics,U.S., 2003.

[20] U. Trottenberg, C.W. Oosterlee, and A. Schuller. *Multigrid*. Society for Industrial and Applied Mathematics, 2004.

[21] Ž. Tuković. *Finite Volume Method on Domains of Varying Shape*. PhD thesis, Faculty of Mechanical Engineering and Naval Architecture, University of Zagreb, 2005. In Croatian.

[22] H. Jasak and Ž. Tuković. Automatic mesh motion for the unstructured finite volume method. *Transactions of FAMENA*, 30(2):1–18, 2007.

[23] Ž. Tuković and H. Jasak. Automatic mesh motion in the FVM. 2nd MIT CFD Conference, Boston, June 2003.

[24] C.J. Greenshields, H. G. Weller, and A. Ivanković. The finite volume method for coupled fluid flow and stress analysis. *Computer Modelling and Simulation in Engineering*, 4:213–218, 1999.