CBC.RANS – a new flexible, programmable software framework for computational fluid dynamics

Mikael Mortensen[1,2]**, Hans Petter Langtangen**[2,3] **and Jørgen Myre**[4]

[1]Norwegian Defence Research Establishment (FFI)
Corresponding author: Mikael.Mortensen@ffi.no

[2]Center for Biomedical Computing
Simula Research Laboratory

[3]Department of Informatics
University of Oslo

[4]Department of Mathematics
University of Oslo

Summary CBC.RANS (https://launchpad.net/cbc.rans) is a new flexible, programmable software framework for assisting with numerical experiments in computational turbulence. The framework targets Reynolds-averaged Navier-Stokes (RANS) models discretized by finite element methods. The novel implementation makes use of Python and the FEniCS package, a combination that allows model- and solver-specific code to closely resemble the mathematical formulation of equations and algorithms.
CBC.RANS is designet to be a valuable tool for developers of turbulence models. A 'naive' implementation of a new model can be realized very quickly since this process pretty much boils down to correctly typing the variational forms of the equations and derived quantities being solved for. The user has total control over the solution algorithms, linearizations (e.g., Picard or Newton), coupling of systems of equations, discretization (e.g., continuous or discontinuous Galerkin methods of arbitrary order) and implicit vs. explicit treatment of terms. Finding the most robust and efficient solution algorithm and discretization for a given turbulence model has always relied heavily on experimentation, and through CBC.RANS this process has become substantially easier compared to existing CFD software frameworks.
This short paper takes the form of a tutorial for implementing two state-of-the-art turbulence models, $v^2 - f$ and a model based on elliptic relaxation. The original $v^2 - f$ turbulence model has a reputation for being notoriously difficult to implement in standard commercial CFD packages due to certain limitations not inherent in CBC.RANS. The elliptic relaxation model, on the other hand, is a highly complex Reynolds stress model that, in addition to solving a regular $k - \varepsilon$ model, also requires the solution of transport equations for two symmetric second rank tensors, that, to top it off, should be solved in a coupled manner due to complex boundary conditions. We illustrate in this paper how the process of implementing and solving these models is made relatively easy through CBC.RANS, and as proof of concept we show results from simulations of a plane channel and a diffusor.

Introduction

Most flows in nature and man-made constructions are turbulent. Unfortunately, finding the proper turbulence model for a given flow case is demanding. There exists a large number of different turbulence models, yet constructing efficient and robust iteration techniques is highly model- and problem-dependent, and hence subject to extensive experimentation. Flexible software tools can greatly enhance the researcher's productivity when experimenting with models

and numerical methods. The present paper describes a new generation of such flexible software tools.

Direct Numerical Simulation (DNS) and Large Eddy Simulations (LES), though being accurate, are often considered too expensive for the simulation of turbulent flows in many practical applications. A computationally efficient approach to turbulent flows is to work with Reynolds-averaged Navier-Stokes (RANS) models. RANS models involve solving the incompressible Navier–Stokes (NS) equations in combination with a set of highly nonlinear transport equations for statistical turbulence quantities. The uncertainty in RANS models lies in the extra transport equations, and for a given flow problem it is a challenge to pick an appropriate model. Extensive experimentation is usually the way to go to arrive at firm conclusions on the physics of a problem.

Most commercial computational fluid dynamics (CFD) packages contain a limited number of turbulence models, but allow users to add new models through *user subroutines*. The implementation of such routines can be difficult, and many new models might not fit easily with the design of the package and the given interface to the user subroutine. Therefore, there is a need for CFD software with a flexible design so that new partial differential equation (PDE) systems can be added quickly and reliably. Moreover, new PDE systems may also require composition of new solution approaches. We are convinced that the most cost-effective way of meeting such demands is to have a *programmable* framework, where the models and numerics are defined in terms of a compact, high-level computer language with a syntax that is close to the mathematical language and abstractions.

A software system for RANS modeling must provide higher-order spatial discretizations, fine-grained control of linearizations, support for both Picard and Newton type iteration methods, under-relaxation, restart of models, combinations of models and the easy implementation of new nonlinear systems of PDEs. Standard building blocks needed in PDE software, such as computing and solving linear systems, can act as black boxes for a researcher in computational turbulence. To the authors' knowledge, there is little software with the mentioned flexibility for incompressible CFD. There are, however, many programmable environments for solving PDEs in general [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**]. Only a few of the cited packages have been extensively used for turbulent flow. OpenFOAM [**?**] is a well-structured and widely used object-oriented C++ framework for CFD, based on finite volume methods, where new models can quite easily be added through high-level C++ statements. FEniCS [**?**] is a recent C++/Python framework, where systems of PDEs and corresponding discretization and iteration strategies can be defined in terms of a few high-level Python statements which mimic the mathematical structure of the problem. The approach advocated in this work utilizes FEniCS tools. All FEniCS components are freely available under GNU general public licenses [**?**].

Traditional simulation software packages are usually implemented in Fortran, C, or C++ because of the need for high computational performance. A consequence is that these packages are less user-friendly and flexible, but far more efficient, than similar projects implemented in scripting languages such as Matlab or Python. In FEniCS, scripting is combined with symbolic mathematics and code generation to provide both user-friendliness and efficiency. Specifically, the Unified Form Language (UFL), a domain-specific language for the specification of variational formulations of PDEs, is embedded within the programming language Python. Variational formulations are then just-in-time compiled into C++ code for efficiency. The generated C++ code can be expected to outperform hand-written code since special-purpose PDE compilers [**?**, **?**, **?**] are employed. UFL has built-in support for automatic differentiation, which makes

it particularly useful for complicated and coupled PDE problems.

FEniCS supports finite element methods, including discontinuous Galerkin methods [**?**], for spatial discretization. Many finite volume methods can be constructed as low-order discontinuous Galerkin methods using FEniCS [**?**]. Several successful finite element methods methods for incompressible laminar flow and LES models have been constructed over the years. However, finite element methods have to a little extent been applied to RANS models, but some work in this direction exists [**?**, **?**, **?**, **?**]. Many CFD practitioners question the suitability of finite elements for RANS models. One of the aims of CBC.RANS is to clear up this misunderstanding and demonstrate the general applicability of finite elements for RANS models. For some problems, higher-order finite element discretizations provide means to overcome challenges that cannot be met by the dominating lower-order finite volume methods. However, there are some theoretical difficulties in applying finite element methods to RANS models, but these have been resolved within CBC.RANS.

A number of applications that make use of the FEniCS software have been published [**?**]. For instance, CBC.Solve [**?**] is a framework for solving several classes of problems: the incompressible Navier-Stokes equations (CBC.Flow), the equations for large-strain hyperelasticity (CBC.Twist), the equation of slow viscous mantel flow (CBC.Rock), and ALE formulations of fluid-structure interaction problems (CBC.Swing). CBC.Solve applications share some of the features of CBC.RANS, but at this time of writing CBC.RANS targets even more complicated systems of PDEs and aims at higher flexibility and ease of use compared with the mentioned applications. The present paper reports work in the extension of the description of CBC.RANS in [**?**].

Most of the popular CFD packages offer several types of RANS models. Nevertheless, two of the most promising models, the (original) $v^2 - f$ model with 8 PDEs and the elliptic relaxation model with 18 PDEs, have been viewed as extremely difficult to implement, and as a result, these models are hardly used by practitioners. This paper shows, however, that with sufficiently flexible programming tools, both models can be quickly implemented and be made available in user-friendly form to a large audience.

The remainder of this paper is organized as follows. We first describe the Reynolds averaged Navier-Stokes equations as well as two models, $v^2 - f$ and elliptic relaxation, for closing the unknown Reynols stresses. We then illustrate how these two models have been implemented in CBC.RANS. Finally, as proof of concept, the implementations are tested for two channel configurations.

Reynolds averaged Navier-Stokes equations

Turbulent flows are described by the Navier-Stokes (NS) equations. On a domain $\Omega \subset \mathbb{R}^d$ for time $t \in (0, t_s]$, the incompressible NS equations read

$$\frac{\partial \boldsymbol{U}}{\partial t} + \boldsymbol{U} \cdot \nabla \boldsymbol{U} = -\frac{1}{\varrho} \nabla P + \nabla \cdot \nu (\nabla \boldsymbol{U} + \nabla \boldsymbol{U}^T) + \boldsymbol{f}, \tag{1}$$

$$\nabla \cdot \boldsymbol{U} = 0, \tag{2}$$

where $\boldsymbol{U}(\boldsymbol{x}, t)$ is the velocity, $P(\boldsymbol{x}, t)$ is the pressure, ν is the kinematic viscosity, ϱ is the mass density and $\boldsymbol{f}$ represents body forces. A boldface font is used throughout this paper to identify a tensor. The incompressible NS equations must be complemented by initial and appropriate boundary conditions to complete the problem.

Simulations of turbulent flows are usually computationally expensive because of the need for extreme resolution in both space and time. However, in most applications the average quantities are of interest. In the statistical modeling of turbulent flows, the velocity and pressure are viewed as random fields which can be decomposed into mean and fluctuating parts: $\boldsymbol{U} = \boldsymbol{u} + \boldsymbol{u}'$ and $P = p + p'$, where $\boldsymbol{u}$ and p are ensemble averages of $\boldsymbol{U}$ and P, respectively, and $\boldsymbol{u}'$ and p' are the corresponding random fluctuations about the mean field. Inserting for these decompositions into (1) and (2) and averaging results in a system of equations for the mean quantities $\boldsymbol{u}$ and p:

$$\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u} \cdot \nabla \boldsymbol{u} = -\frac{1}{\varrho}\nabla p + \nabla \cdot \nu(\nabla \boldsymbol{u} + \nabla \boldsymbol{u}^T) - \nabla \cdot \overline{\boldsymbol{u}' \otimes \boldsymbol{u}'} + \overline{\boldsymbol{f}}, \tag{3}$$

$$\nabla \cdot \boldsymbol{u} = 0, \tag{4}$$

where $\boldsymbol{R} = \overline{\boldsymbol{u}' \otimes \boldsymbol{u}'}$, known as the Reynolds stress tensor, is the ensemble average of $\boldsymbol{u}' \otimes \boldsymbol{u}'$. The Reynolds stress tensor is unknown and solving equations (3) and (4) requires approximating $\boldsymbol{R}$ in terms of $\boldsymbol{u}$, $\nabla \boldsymbol{u}$, or other computable quantities.

Turbulence shows dispersive nature, and mixing on a large scale is much more efficient than in laminar flows. It is then natural to think that that the Reynolds stress tensor $\boldsymbol{R}$ can be related to to the strain rate tensor of the mean velocity field, $\boldsymbol{S} = (\nabla \boldsymbol{u} + \nabla \boldsymbol{u}^T)/2$:

$$\boldsymbol{R} = -2\nu_T \boldsymbol{S} + \frac{2}{3}k\boldsymbol{I}, \tag{5}$$

where ν_T is the "turbulent viscosity", $k = \overline{\boldsymbol{u}' \cdot \boldsymbol{u}'}/2$ is the turbulent kinetic energy and $\boldsymbol{I}$ is the identity tensor. Models that utilize (5) are usually referred to as *eddy-viscosity* models. Many models have been proposed for the turbulent viscosity. The most commonly employed "one-equation" turbulence model is the Spallart Allmaras model [**?**]. It involves a transport equation for a "viscosity" parameter, coupled to 11 derived quantities with 9 model parameters.

Two-equation eddy-viscosity models represent the largest class of RANS models, providing two transport equations for the turbulence length and time scales. This family of models includes the k–ε models [**?**, **?**] and the k–ω models [**?**]. Due to severe mesh resolution requirements these models usually involve the use of wall functions instead of regular boundary conditions on solid walls. Support for the use of wall functions has been implemented in CBC.RANS and special near-wall modifications are employed both for the standard k–ε model, the four equation v^2–f model [**?**] and the elliptic relaxation model [**?**]. Implementation aspects of these wall modifications, though, involve a level of detail that is beyond the scope of the current presentation.

Reynolds stress models belong to a higher level of sophistication than eddy-viscosity models. The Reynolds stress models do not utilize (5), but instead solves for a separate transport equation for the second rank tensor $\boldsymbol{R}$. In FEniCS there is support for working with second rank tensors, and in principle a tensor PDE for a second rank tensor can be implemented with one single line of code, just like scalar transport equations.

Reynolds stress models

The exact transport equation for the Reynolds stresses $\boldsymbol{R}$ can be derived from the Navier-Stokes equations (1). The equation for $\boldsymbol{R}$ is most conveniently written using index notation with summation implied by repeated indices:

$$\frac{\partial R_{ij}}{\partial t} + u_k \frac{\partial R_{ij}}{\partial x_k} + \frac{\partial T_{kij}}{\partial x_k} = \mathbb{P}_{ij} + \mathbb{G}_{ij} - \varepsilon_{ij}. \tag{6}$$

The production term $\mathbb{P}_{ij}$ is closed and defined as

$$\mathbb{P}_{ij} = -R_{ik}\frac{\partial u_j}{\partial x_k} - -R_{jk}\frac{\partial u_i}{\partial x_k}. \tag{7}$$

The dissipation term ε_{ij} is defined as

$$\varepsilon_{ij} = 2\nu\overline{\frac{\partial u_i}{\partial x_k}\frac{\partial u_j}{\partial x_k}}. \tag{8}$$

The dissipation rate is unclosed and requires modeling. One option is to assume that the dissipation is isotropic, i.e., $\varepsilon_{ij} = 1/3\varepsilon_{kk}\delta_{ij}$, where a separate closure is required for $\varepsilon = 0.5\varepsilon_{kk}$. A Reynolds stress model is often accompanied by a k-ε model to provide plausible time and lengthscales. More sophisticated models, like the elliptic relaxation model [**?**] presented below, incorporates the inhomogeneous part of ε_{ij} into a model for a modified $\mathbb{G}_{ij}$.

The third rank Reynolds stress flux tensor T_{kij} is defined as

$$T_{kij} = \overline{u_i' u_j' u_k'} - \nu\frac{\partial R_{ij}}{\partial x_k} + \frac{1}{\rho}\left(\overline{u_i' p}\delta_{jk} + \overline{u_j' p}\delta_{ik}\right). \tag{9}$$

The first term of T_{kij} is usually closed by invoking a gradient diffusion model, whereas the second term is already closed. The last part is usually incorporated into the model for $\mathbb{G}_{ij}$.

The pressure-rate-of-strain tensor $\mathbb{G}_{ij}$ serves to redistribute energy among the Reynolds stresses. For accurate modeling this term is generally considered to be the most important to close. $\mathbb{G}_{ij}$ is defined as

$$\mathbb{G}_{ij} = \overline{\frac{p'}{\rho}\left(\frac{\partial u_i'}{\partial x_j} + \frac{\partial u_j'}{\partial x_i}\right)}. \tag{10}$$

A familiy of pressure-rate-of-strain models can be written as

$$\mathbb{G}_{ij} = \varepsilon\sum_{n=1}^{8}\alpha^{(n)}\mathbb{T}_{ij}^{(n)}, \tag{11}$$

where the non-dimensional, symmetric, deviatoric tensors $\mathbb{T}_{ij}^{(n)}$ are given in Table 11.3 of Pope [**?**]. Several suggestions for the model parameters $\alpha^{(n)}$ are given in Table 11.4 of [**?**].

Durbin [**?**] does not use an algebraic closure for $\mathbb{G}_{ij}$. Instead the tensor is closed through an additional elliptic PDE for another second rank tensor f_{ij}. The elliptic relaxation model of Durbin reads

$$\mathbb{G}_{ij} - \left(\varepsilon_{ij} - \frac{R_{ij}}{k}\varepsilon\right) = kf_{ij}, \tag{12}$$

$$A_{ij} = \frac{R_{ij}}{2k} - \frac{\delta_{ij}}{3}, \tag{13}$$

$$L^2\nabla^2 f_{ij} - f_{ij} = -\frac{\mathbb{G}_{ij}^h}{k} - \frac{2A_{ij}}{T}, \tag{14}$$

where the time and lengthscales T and L can be computed as

$$T = \max\left\{\frac{k}{\varepsilon}, 6\sqrt{\frac{\nu}{\varepsilon}}\right\}, \tag{15}$$

$$L = C_L\min\left\{\frac{k^{\frac{3}{2}}}{\varepsilon}, C_\eta\left(\frac{\nu^3}{\varepsilon}\right)^{\frac{1}{4}}\right\}, \tag{16}$$

where $C_L = 0.25$ and $C_\eta = 80$. Note that several alternatives have been presented for closure of the right hand side of the equation for f_{ij}. In principle, any of the models presented in Table 11.4 of [**?**] can be used for the homogeneous term $\mathbb{G}^h_{ij}$. The model used in this work is the LRR-IP model defined as

$$\mathbb{G}^h_{ij} = -\frac{2C_R}{T}A_{ij} - C_2\left(\mathbb{P}_{ij} - \frac{2}{3}\mathbb{P}\delta_{ij}\right), \tag{17}$$

where $C_R = 1.8$, $C_2 = 0.6$ and $\mathbb{P} = 0.5\mathbb{P}_{ii}$.

The $v^2 - f$ model

The $v^2 - f$ model of Durbin [**?**] was introduced prior to the elliptic relaxation model, but is commonly viewed as a simplification of this. The $v^2 - f$ model is a four equation eddy-viscosity model that uses one scalar transport equation for a parameter v^2, which can be interpreted as the wall normal Reynolds stress, and a pressure distribution parameter f for the stress in the wall normal direction. Hence it is a simplification of the elliptic relaxation model, retaining only the most important terms for parallel shear flows. The model reads

$$\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u}\cdot\nabla\boldsymbol{u} = \nabla\cdot\nu_u(\nabla\boldsymbol{u} + \nabla\boldsymbol{u}^T) - \frac{1}{\varrho}\nabla p + \overline{\boldsymbol{f}}, \tag{18}$$

$$\nabla\cdot\boldsymbol{u} = 0, \tag{19}$$

$$\frac{\partial k}{\partial t} + \boldsymbol{u}\cdot\nabla k = \nabla\cdot(\nu_k\nabla k) + \mathbb{P} - \varepsilon, \tag{20}$$

$$\frac{\partial \varepsilon}{\partial t} + \boldsymbol{u}\cdot\nabla\varepsilon = \nabla\cdot(\nu_\varepsilon\nabla\varepsilon) + \left(C_{\varepsilon 1}\mathbb{P} - C_{\varepsilon 2}\varepsilon\right)/T, \tag{21}$$

$$\frac{\partial v^2}{\partial t} + \boldsymbol{u}\cdot\nabla v^2 = \nabla\cdot(\nu_{v^2}\nabla v^2) + kf - v^2\frac{\varepsilon}{k}, \tag{22}$$

$$L^2\nabla^2 f = f + \frac{C_1 - 1}{T}\left(\frac{v^2}{k} - \frac{2}{3}\right) - C_2\frac{\mathbb{P}}{k} \tag{23}$$

$$\nu_\alpha = \nu + \nu_T/\sigma_\alpha, \text{ for } \alpha = u, k, \varepsilon, v^2, \tag{24}$$

$$\nu_T = C_{\mu 1}v^2 T. \tag{25}$$

The main motivation for the $v^2 - f$ model is the turbulent viscosity model (25) that has shown much closer agreement with direct numerical simulations and experiments of near wall turbulence than the common $\nu_T = C_\mu kT$, used in most two-equation turbulence models.

The parameters of the $v^2 - f$ model vary somewhat in the literature, but are most commonly set to $C_{\mu 1} = 0.22$, $C_{ed} = 0.045$, $C_{\varepsilon 2} = 1.9$, $C_1 = 1.4$, $C_2 = 0.3$, $\sigma_u = \sigma_k = \sigma_{v^2} = 1.0$, $\sigma_\varepsilon = 1.3$. To avoid using the distance to the nearest wall directly, the final parameter $C_{\varepsilon 1}$ is most often computed as

$$C_{\varepsilon 1} = 1.4\left(1 + C_{ed}\sqrt{\frac{k}{v^2}}\right), \tag{26}$$

which is a slight modification of the original model.

The pressure p in the NS equations is a modified pressure that includes the kinetic energy from the model for the Reynolds stresses. On no slip boundaries (walls) in parallel shear flows it is well known that k, v^2 (here interpreted as the wall normal stress), ε and f behave asymptotically as [**?**]

$$k \sim O(y^2), \quad v^2 \sim O(y^4), \quad \varepsilon \sim O(1), \quad f \sim O(1), \tag{27}$$

as the wall is approached ($y \longrightarrow 0$). Here y denotes the coordinate normal to the wall. In other words as $y \longrightarrow 0$ the proper wall boundary conditions are $k = 0$ and $v^2 = 0$. By considering Eqs. (20) and (22) asymptotically near the wall we can also obtain

$$\varepsilon \longrightarrow 2\nu\frac{k}{y^2} \tag{28}$$

and

$$kf \longrightarrow -5\frac{v^2\varepsilon}{k} \text{ or } f \longrightarrow -\frac{20\nu^2 v^2}{\varepsilon y^4}. \tag{29}$$

Note that the boundary condition for f involves y^4 and ε in the denominator. This boundary condition has been discussed by several authors as being responsible for numerical instabilities, due to the strong coupling with v^2. As a consequence, the $v^2 - f$ model has usually had to be implemented with a coupled solver (solving systems of equations, k and ε together and then v^2 and f together), and not the more common segregated solvers found in most commercial CFD software. (Segregated here means that the individual equations in a PDE system is solved one by one, not fully coupled.) The need for a coupled solver has motivated several alternative "code-friendly" versions of the $v^2 - f$ model [**?**, **?**, **?**], where f in general has been redefined such that $f = 0$ on walls. Since coupling of PDEs is trivial in FEniCS, the original model can be implemented with a coupled solver that implicitly sets the boundary conditions, and the common troubles with numerical instability associated with boundary conditions are thus never encountered.

Implementation details

In this section we present some details of the implementations in CBC.RANS of the $v^2 - f$ model and the elliptic relaxation model. For a deeper understanding of the examples, the reader should be familiar with some of the inner workings of CBC.RANS, as presented in [**?**]. However, we hope the examples can give newcomers to CBC.RANS a glimpse of how the framework is operated. Some familiarity with object-oriented programming will be beneficial in the further reading.

All turbulence models in CBC.RANS are implemented by subclassing a common superclass `TurbSolver`. A new model is required to at least provide (i) a system composition describing the names of primary unknowns (e.g., 'k' and 'e' for a $k - \varepsilon$ model) and how they are coupled, (ii) a list of model parameters and their values, (iii) a list of mathematical expressions for derived quantities (e.g., turbulent viscosity) and (iv) variational forms to represent the various PDEs entering the model. Each RANS model can be used with any NS solver and each `TurbSolver` instance contains a reference to a `NSSolver` class that at least provides the mean velocity and pressure. In CBC.RANS there are two main families of NS solvers, fully coupled (`NSCoupled`) or segregated (`NSSegregated`). The coupled solvers solve for the velocity and pressure simultaneously. The segregated solvers decouple the pressure from the velocity and among the many solvers implemented we mention the Chorin projection scheme [**?**] and incremental pressure correction schemes [**?**]. Each solver typically has several alternative schemes with different choice of discretization for the convective and diffusive terms. Crank-Nicholson and Adams-Bashforth schemes are used most commonly to discretize in time. The convective term can be implemented using standard form `inner(v, grad(u)*w)*dx`, divergence form `inner(v, div(outer(u, w)))*dx` or skew form, which is a combination of the two former. In these formulas `v` and `u` are typically the test- and trialfunctions for velocity, whereas `w` is a

velocity form suitable for the chosen discretization. For a Newton method we can choose `w=u` and obtain a nonlinear form. For a Picard method `w` must be computed from known velocity fields (`Functions`) in order for convection to be a bilinear or linear form. The operators `inner, div, grad, outer` are key ingredients of the Unified Form Language (UFL) that perform the stated task on a finite element variational form or functional. For example, `grad(u)` is the gradient of the `TrialFunction u` and the form `grad(u)` represents a second rank tensor. One might think of UFL as a flexible user interface for defining finite element spaces and expressions for weak forms in a notation close to mathematical notation. The operators `inner, div, grad, outer` will be used heavily in the construction of variational forms representing the PDEs of the turbulence models described in the previous section.

Implementation details for the $v^2 - f$ model

The $v^2 - f$ model contains four primary unknowns (k, ε, v^2 and f) represented by Eqs. (20) - (23) and naturally there are many different ways of coupling them. We could for example solve for all parameters in a fully coupled system, or for two subsystems (k and ε) and (v^2 and f). The variable `system_composition` determines the coupling. We implement a semi-coupled system in the solver class `V2F_2Coupled` by feeding the `system_composition = [['k', 'e'], ['v2', 'f']]` to the superclass `TurbSolver`. The composition is a list of subsystems in the total PDE system, and each subsystem lists its unknowns (or more precisely, the equations related to these unknowns). As an example the coupled NS solvers have `system_composition = [['u', 'p']]`, whereas the segregated solvers use `[['u'], ['p']]`. An iteration is set up between the subsystems, but each subsystem is solved in a fully coupled fashion.

The superclass automatically generates neccessary function spaces, test- and trial functions and functions for holding the various components of the total solution of the problem. For example, the k unknown has trial and test functions `k` and `v_k`, whereas the finite element function holding the latest approximation to the solution is represented with an underscore, as in `k_`. This means that we can very easily switch between treating a primary unknown like k explicitly or implicitly in a term: with `k` we have a trial function and implicit treatment (k is unknown), while when adding an underscore, `k_`, the quantity is treated explicitly (k is known, meaning that the most recently computed finite element field for k is used).

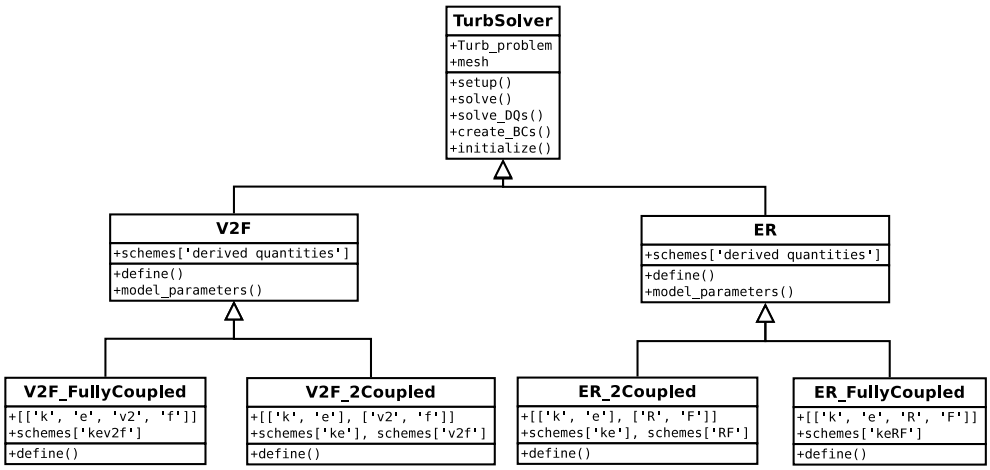


Figure 1: UML sketch of the V2F and elliptic relaxation (ER) solvers in CBC.RANS. The arrows with triangular heads indicate that classes are derived from the classes that they point towards. For better illustration we show the value of the `system_composition` attribute lists.

Model parameters and derived quantities are common to all possible variations of the system composition and as such we create a `V2F` superclass, where these can be implemented. The class hierarchy for the v^2–f and elliptic relaxation solvers is illustrated in Figure 1. Note that we merely need to implement two new methods (`model_parameters` and `define`) for any new solver. The remaining work, including routines for solving the PDE system and generating boundary conditions, is performed by the `TurbSolver` superclass.

The `define` method is responsible for setting up the variational forms and derived quantities representing the model [i.e., Eqs. (20)-(26), (15) and (16)]. An example of a variational form for the $k-\varepsilon$ subsystem of the v^2-f model is shown in Figure 2.

```
class Steady_ke_1(TurbModel):

    def form(self, k, e, v_k, v_e, k_, e_, nu, nut_, u_, Ce1_,
             P_, T_, Ce2, e_d, sigma_e, **kwargs):

        Fk = (nu + nut_)*inner(grad(v_k), grad(k))*dx \
            + inner(v_k, inner(u_, grad(k)))*dx \
            - P_*v_k*dx + (e*e_d + k*e_/k_*(1. - e_d))*v_k*dx

        Fe = (nu + nut_*(1./sigma_e))*inner(grad(v_e), grad(e))*dx \
            + inner(v_e, inner(u_, grad(e)))*dx \
            - (Ce1_*P_ - Ce2*e)*(1/T_)*v_e*dx

        return Fk + Fe
```

Figure 2: A variational form for $k-\varepsilon$ subsystem in the v^2-f model. `TurbModel` is a superclass for all turbulence model schemes, see [**?**].

The corresponding variational form in mathematical notation, derived from (20) and (21) and presented in stationary form to save space, reads: find $k \in V_k$ such that

$$F_k \equiv \int_\Omega \boldsymbol{u}\cdot\nabla k v_k dx + \int_\Omega \nu_k \nabla k\cdot\nabla v_k dx - \int_\Omega (\mathbb{P}-\varepsilon)v_k dx = 0 \quad \forall\, v_k \in V_k, \tag{30}$$

and find $\varepsilon \in V_\varepsilon$ such that

$$F_\varepsilon \equiv \int_\Omega \boldsymbol{u}\cdot\nabla \varepsilon v_\varepsilon dx + \int_\Omega \nu_\varepsilon \nabla\varepsilon\cdot\nabla v_\varepsilon dx - \int_\Omega \left(C_{\varepsilon 1}\mathbb{P} - C_{\varepsilon 2}\varepsilon\right) T^{-1} v_\varepsilon dx = 0 \quad \forall\, v_\varepsilon \in V_\varepsilon, \tag{31}$$

where V_k and V_ε are suitably defined function spaces (e.g., linear or quadratic triangular elements).

A key theme is to linearize the equations and select the terms to be treated implicitly. The `k` and `e` variables (trial functions) implies implicit treatment, while an underscore indicates explicit treatment, using the most recently computed values. For example, in Figure 2 the production term `P_*v_k*dx` is explicit, whereas convection and diffusion are treated implicitly. The dissipation term ε in the k equation is naturally explicit if the k equation is solved for in a segregated manner. However, this term often gives rise to instability, so an implicit treatment is desired. This can be achieved by rewriting the term as $k\varepsilon/k$ and treating ε/k as known, coded as `k*e_/k_`. A more sophisticated approached is invoked in Figure 2 where we introduce a weighting of this implicit rewrite and the original term, in implicit form (`e`), with `e_d` as weighting factor: `e*e_d + k/k_*e_*(1 - e_d)`. For `e_d` greater than zero we also bring in `e`, which then couples implicitly to `k`. We think this example of treating a simple term like ε in the original PDE illustrates the challenges with linearization of RANS models and the highly desired software flexibility. In CBC.RANS one can just add and remove underscores to adjust

the implicit treatment of variables. Replacing `e*e_d` by `e_*e_d` effectively splits the k and ε equations and implies a segregated approach (although we technically still solve for k and ε simulataneously).

The derived quantities descibed in (15), (16), (25) and (26), as well as $\mathbb{P}$ and the stress tensor $\boldsymbol{S}$ are implemented in the base class `V2F` as shown in Figure 3. Note that the derived quantity `Ce1` (26) uses a wall function approach, since `k_/v2_` is undefined on a solid wall. To achieve this we create a class `Ce1` as a subclass of `DerivedQuantity` and overload the `create_BC` method (details not shown). A `DerivedQuantity` enforces homogeneous Dirichlet boundary conditions on solid walls, whereas the `DerivedQuantity_NoBC` does nothing, which is appropriate for, e.g., the shear stress $\boldsymbol{S}$.

```
class V2F(TurbSolver):

    def define(self):
        """define derived quantities for V2F model."""
        V,  NS = self.V['dq'], self.Turb_problem.NS_solver # Short forms
        DQ, DQ_NoBC = DerivedQuantity, DerivedQuantity_NoBC
        NS.schemes['derived quantities'] = [
            DQ_NoBC(NS, 'Sij_', NS.S, "epsilon(u_)", dict(u_=NS.u_), bounded=False)]
        self.Sij_ = NS.Sij_
        ns = vars(self)
        self.schemes['derived quantities'] = [
            DQ_NoBC(self, 'T_', V, "max_(k_*(1./e_), 6.*sqrt(nu*(1./e_)))", ns),
            DQ_NoBC(self, 'L_', V, "CL*max_(Ceta*(nu**3/e_)**(0.25), k_**(1.5)/e_", ns),
            DQ     (self, 'nut_', V, "Cmu*v2_*T_", ns),
            DQ     (self, 'P_', V, "2.*inner(Sij_, grad(u_))*nut_", ns, bounded=False),
            Ce1    (self, 'Ce1_', V, "1.4*(1. + Ced*sqrt(k_/v2_))", ns)
        ]
```

Figure 3: Implementation of derived quantities for the $v^2 - f$ model. The derived quantity `Ce1_`, given in (26), uses a wall function approach since `k_/v2_` is undefined on a solid wall. The wall function is implemented in class `Ce1` that is a subclass of `DerivedQuantity`.

Implementation details for the elliptic relaxation model

The elliptic relaxation model requires about the same level of coding as the $v^2 - f$ model. We define a `system_composition` = `[['k', 'e'], ['R', 'F']]` and solve one coupled $k - \varepsilon$ subsystem as well as a coupled subsystem consisting of the Reynolds stress equation for $\boldsymbol{R}$ and the quantity $\boldsymbol{F}$, representing the second rank tensor f_{ij}. One significant difference is that we need to tell the superclass `TurbSolver` that the parameters $\boldsymbol{R}$ and $\boldsymbol{F}$ represent two second rank tensors and not scalars that is assumed as default. The implementation that achives this is shown in Figure 4.

Derived quantities used in the elliptic relaxation model are implemented in the base class `ER` as shown in Figure 5. Note that we implement both an implicit `Gh` and an explicit `Gh_`. The explicit form is implemented as a `DerivedQuantity` because we can then choose to use the formula as it is written, or we could compute a `Function` `Gh_` by projecting the formula on the `TensorFunctionSpace` `V['R']`. The latter approach has the advantage that we can introduce underrelaxation, and combined with the fact that projection is a process where the solution is filtered through the mass matrix, this will often have a stabilizing effect on the solution process. All CFD softwares today use underrelaxation for primary variables in turbulence models, and most also for at least the turbulent viscosity. Underrelaxation is used in CBC.RANS for any derived quantity that has keyword `apply` set to 'project'. The underrelaxation factor can be specified in the parameters dictionary `prm` of any `DerivedQuantity` instance by setting the value of the keyword `omega`.

```
class ER(TurbSolver):

    def __init__(self, system_composition, problem, parameters):
        parameters['space']['R'] = TensorFunctionSpace
        parameters['space']['F'] = TensorFunctionSpace
        dim = problem.NS_problem.mesh.geometry().dim()
        # The second rank tensors are symmetric, thus
        symmetry = dict(((i,j), (j,i))
                        for i in range(dim) for j in range(dim) if i > j )
        parameters['symmetry']['R'] = symmetry
        parameters['symmetry']['F'] = symmetry
        TurbSolver.__init__(self,
                    system_composition=system_composition,
                    problem=problem,
                    parameters=parameters)

class ER_2Coupled(ER):

    def __init__(self, problem, parameters):
        ER.__init__(self,
                    system_composition=[['k', 'e'], ['R', 'F']],
                    problem=problem,
                    parameters=parameters)
```

Figure 4: Declaration of $\boldsymbol{R}$ and $\boldsymbol{F}$ as symmetric `TensorFunctionSpace` objects. The systems composition is specified in the child class, here represented by `ER_2Coupled`, whereas the common code for all elliptic relaxation solvers is put in the parent class `ER`.

```
class ER(TurbSolver):

    def define(self):
        """define derived quantities for V2F model."""
        V,  NS = self.V['dq'], self.Turb_problem.NS_solver # Short forms
        DQ, DQ_NoBC = DerivedQuantity, DerivedQuantity_NoBC
        self.dij = Identity(self.V['R'].cell().d)
        # Define 'implicit' derived quantities
        self.A = self.R*(0.5/self.k_) - 1./3.*self.dij
        self.Gh = -self.CR*(1./self.T_)*2.*self.A*self.k_ \
                  -self.C2*(self.P_ - 1./3.*tr(self.P_)*self.dij)
        # Define 'explicit' derived quantities
        ns = vars(self)
        NS.schemes['derived quantities'] = [
        DQ_NoBC(self, 'A_', NS.S, "R_*(0.5/k_) - 1./3.*dij", ns,
                              bounded=False)
        DQ_NoBC(self, 'P_', self.V['R'], "- dot(R_, grad(u_).T) - \
                             dot(grad(u_), R_.T)", ns, bounded=False)
        DQ_NoBC(self, 'Gh_', self.V['R'], "-CR*(1./T_)*2.*A_*k_ \
                            - C2*(P_ - 1./3.*tr(P_)*dij)", ns, bounded=False)
        ...]
```

Figure 5: Implementation of derived quantities for the elliptic relaxation model.

A 'naive' (simplest possible) implementation of the variational forms of the PDEs governing the elliptic relaxation model is shown in Figure 6. Note how $\boldsymbol{F}$ is made implicit in the Reynolds stress equation through the third term in Fr, whereas the Reynolds stress is made implicit through the use of `A` and not `A_` in the expression for the implicit form `Gh`. Note that the $k-\varepsilon$ implementation differ from that used in the v^2-f model, see Figure 2, because the Reynolds stresses are used to close the turbulent transport terms (the last term in both `Fe` and `Fk`).

Running a channel test case with the new models

When a new model has been added we need to make CBC.RANS aware of it. For the previously defined models this is accomplished by making the following modifications to the `__init__.py` file in the `turbsolvers` directory

```
from V2F_2Coupled import V2F_2Coupled
from V2F_FullyCoupled import V2F_FullyCoupled
```

```
class Steady_ke_1(KEBase):
    def form(self, k, e, v_k, v_e, k_, e_, R_, P_, nu, u_, Ce1_, T_, Ce2,
                   Cmu, e_d, sigma_e, nut_, **kwargs):
        Fk = nu*inner(grad(k), grad(v_k))*dx \
             + inner(dot(u_, grad(k)), v_k )*dx \
             - inner(0.5*tr(P_), v_k)*dx \
             + (e*e_d + k*(1./k_)*e_*(1. - e_d))*v_k*dx \
             + inner(Cmu*T_*dot(R_, grad(k)), grad(v_k))*dx

        Fe = nu*inner(grad(e), grad(v_e) )*dx \
             + inner(dot(u_, grad(e)), v_e)*dx \
             - inner(0.5*Ce1_*(1./T_)*tr(P_), v_e)*dx \
             + Ce2*(1./T_)*e*v_e*dx \
             + inner(Cmu*T_*(1./sigma_e)*dot(R_, grad(e)), grad(v_e))*dx

        return Fk + Fe

class Steady_RF_1(TurbModel):

    def form(self, R, R_, v_R, k_, e_, P_, nu, u_, F, F_, v_F,
             A_, Gh, Cmu, T_, L_, **kwargs):

        Fr = nu*inner(grad(R), grad(v_R))*dx \
             + inner(dot(grad(R), u_), v_R)*dx \
             - inner(k_*F, v_R)*dx \
             - inner(P_, v_R )*dx \
             + inner(R*e_*(1./k_), v_R)*dx \
             + inner(Cmu*T_*dot(grad(R), R_), grad(v_R) )*dx

        Ff = inner(grad(F), grad(L_**2*v_F))*dx \
             + inner(F , v_F)*dx \
             - (1./k_)*inner(Gh , v_F)*dx \
             - (2./T_)*inner(A_ , v_F)*dx

        return Fr + Ff
```

Figure 6: Variational forms for k, ε, the Reynolds stress $\boldsymbol{R}$ and the quantity $\boldsymbol{F}$ used with the elliptic relaxation model.

```
    from ER_2Coupled import ER_2Coupled
    from ER_FullyCoupled import ER_FullyCoupled
```

The new models are now ready to be used with any of the model problems already defined in `cbc/rans/turbproblems/`. However, for accurate simulations most of these problems require precomputed profiles for both velocity and turbulence parameters used for specifying inlet conditions to the problem[1]. The fully developed channel problem, on the other hand, is driven by a constant forcing field computed from the given turbulent Reynolds number and thus periodic boundary conditions for in- and outlets are applied. As such, the channel problem requires only an initial guess to get started. The solution obtained from the channel case can then be used to set inlet profiles for the more complicated geometries, like the adverse pressure gradient flow past a bump (`nsproblems/apbl.py`) or the backwards facing step (`nsproblems/Bfs.py`).

The initial guess for the new parameters (k, ε, etc.) must be specified in the global dictionary `initial_turbulence_constants` located in the top section of the `turbproblems/channel.py` module. That is all there is to it. If the parameter is not specified, then a default value of `1e-3` is used for all components. The boundary conditions discussed briefly in (28) and (29) have been implemented in `common/Wall.py`. However, this level of detail, which is specific to the $v^2 - f$ and elliptic relaxation models, is not shown here.

The channel problem is now ready and can be run using the test script `/turbproblems/test.py`. The user can request 100 iterations with the `V2F_2Coupled` solver by executing the following command in a terminal or in a Python shell (`ipython`):

[1] Precomputed profiles are not required by CBC.RANS, but the use of precomputed channel solutions is simply considered more accurate than specifying a plug flow or similar.

```
run test --p channel --vd 2 --n V2F_2Coupled --m Original_V2F --Nx 6 --Ny 60 \
         --max_iter 100 --Ret 395. --wu 0.8 --wt 0.8
```

We are here using the original formulation of the $v^2 - f$ model. Alternatively, we could also choose the `LienKalizin` [**?**] model that uses homogeneous Dirichlet boundary conditions for f. We use 60 elements for half the channel height and the elements are skewed towards the wall using an inverse tangent function. Details are found in `nsproblems/channel.py`. We use a coupled NS solver (default) with Taylor-Hood P2-P1 elements. The second order P2 element is specified through the `--vd` switch (velocity polynomial degree). The pressure uses the default P1 element. The turbulent Reynolds number based on friction velocity $Re_\tau = u_\tau H/\nu$ is set to 395 and the final arguments `wu` and `wt` specify the underrelaxation factors used for `NSSolver` and `TurbSolver` classes, respectively. The same underrelaxation parameters will then be applied to both the PDEs collected in the `schemes` dictionary and projected `derived quantities` (see [**?**]). Note that all parameters can be tweaked from the command line during the process of finding a solution and we could easily request 10 iterations, then increase the underrelaxation factor for the NS solver and request 10 more iterations. The user has complete control over this process. The given problem should converge with normalized residual errors less than 1e-7 in 36 iterations, starting from constant initial guesses for all turbulence parameters and a laminar velocity profile.

The elliptic relaxation model requires somewhat lower underrelaxation factors, and we need to solve for the whole channel height since the Reynolds stress $\overline{uv}$ and $\boldsymbol{F}_{12}$ are antisymmetric about the center of the channel (otherwise this would require some tweaking of boundary conditions). The total height of the channel is specified in `nsproblems/channel.py` and the problem is run by executing

```
run test --p channel --vd 2 --n ER_2Coupled --m LRR-IP --Nx 6 --Ny 120 \
         --max_iter 100 --wu 0.6 --wt 0.4 --Ret 395.
```

Using the LRR-IP model (17) for $\mathbb{G}^h_{ij}$, the solution converges in about 90 iterations. The results are similar to the $v^2 - f$ model, which also makes use of the LRR-IP model to close for f. The major difference stems from the fact that whereas the $v^2 - f$ model simply uses an eddy-viscosity model in the NS equation, the elliptic relaxation model uses the whole Reynolds stress tensor in the term $\nabla \cdot \boldsymbol{R}$. If we were to define an equivalent eddy-viscosity model for the elliptic relaxation model (e.g., $\nu_T = C_\mu T \boldsymbol{R} : n \otimes n$, where n is the wall normal vector) and choose to add only the eddy-viscosity model to the NS equation, then stabilization is greatly enhanced (we can use higher underrelaxation factors) and results are even closer to the $v^2 - f$ model.

When the channel problem is solved, one-dimensional profiles can be stored through the command

```
Problem.tospline()
```

that stores spline approximations of the profiles in `cbc/rans/data/channel_LRR-IP_395.0.ius` (i.e., profiles are parameterized by problem, model and turbulent Reynolds number).

With these profiles in place we are now in a position where we can run any of the problems that require inlet profiles. The diffusor problem is a channel flow characterized by an expanding section that sets up an adverse pressure gradient, see Figure 7. The basic problem with mesh and boundary conditions is set up in `nsproblems/diffusor.py` and we use the same parameters as for the channel case (otherwise we would have to rerun the channel to get the correct

inlet profiles). Since the elliptic relaxation model contains components that are antisymmetric across the channel, we choose to simulate the whole diffusor geometry even though it is in fact symmetric about the x-axis. For the same reason we also need to tweak the general initialization routine slightly in `turbproblms/diffusor.py`, to ensure that the antisymmetric inlet profiles are generated correctly (all other scalars are symmetric and because of statistical symmetry we store only results for half the channel height). The diffusor flow is simulated by running the command

```
run test --p diffusor --vd 2 --n ER_2Coupled --m LRR-IP --Nx 120 --Ny 60 \
         --max_iter 100 --wu 0.6 --wt 0.4 --Ret 395.
```

and the problem converges nicely in about 100 iterations. For illustration, the flow field and some of the components of $\boldsymbol{R}$ and $\boldsymbol{F}$ are shown in figures 7 and 8.

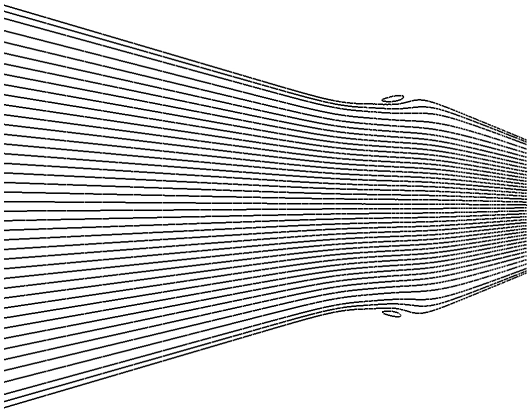
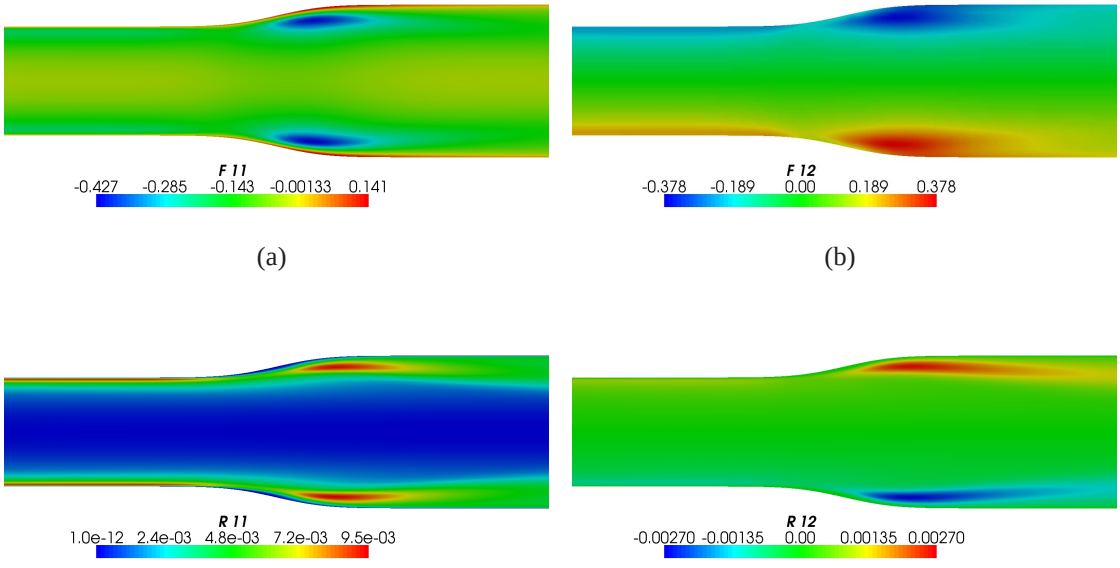


Figure 7: Streamlines for the diffusor.



(a) (b)

(c) (d)

Figure 8: Contour plots for some of the components of $\boldsymbol{F}$ and $\boldsymbol{R}$.

Conclusions

CBC.RANS (https://launchpad.net/cbc.rans) is a new flexible, programmable software framework that can help researchers in computational turbulence enhance their productivity with model experimentation. Compact, easy-to-read Python code specifying the mathematical model automatically generates problem-specific, efficient C++ code for lage-scale simulations. In this paper we have illustrated possibilities and concepts of CBC.RANS by outlining the implementation of two highly complicated state-of-the-art turbulence models, the $v^2 - f$ model and the elliptic relaxation model (which is a Reynolds stress model). For the elliptic relaxation model we solve a system of equations consisting of the Reynolds-Averaged Navier-Stokes equations, a regular $k - \varepsilon$ model and two second rank tensors – the Reynolds stress itself and a quantity $\boldsymbol{F}$ used to close the pressure-strain-rate in the Reynolds stress equation. The Reynolds stress and $\boldsymbol{F}$ are solved in a fully coupled system to enable implicit (and thus stable) treatment of boundary conditions. The resulting implementations are freely available as part of CBC.RANS and easy to use.

References

[1] M. S.Alnæs and K.-A.Mardal On the efficiency of symbolic computations combined with code generation for finite element methods *ACM Trans. Math. Software*, **vol.37**(1), 2010.

[2] W.Bangerth, R.Hartmann and G.Kanschat deal.II – A general-purpose object-oriented finite element library *ACM Trans. Math. Software*, **vol.33**(4), 2007.

[3] A. C.Benim Finite element analysis of confined turbulent swirling flows *Int. J. Num. Methods in Fluids*, **vol.11**, 697–717, 2005.

[4] Software package `http://www.cactuscode.org/`, 2011.

[5] Software package `https://launchpad.net/cbc.solve`, 2011.

[6] A. J.Chorin Numerical solution of the navier-stokes equations *Math. Comp.*, **vol.22**, 745 – 762, 1968.

[7] Software package `http://www.comsol.com`, 2011.

[8] Software package `http://www.dealii.org`, 2011.

[9] Software package `http://www.diffpack.com`, 2011.

[10] P.Dular and C.Geuzaine GetDP: a General environment for the treatment of Discrete Problems `http://www.geuz.org/getdp/`, 2011.

[11] The distributed and unified numerics environment `http://www.dune-project.org/dune.html`, 2011.

[12] P. A.Durbin Near-wall turbulence closure modeling without damping functions *Theoret. Comp. Fluid Dyn.*, **vol.3**, 1–13, 1991.

[13] P. A.Durbin and B. A.Pettersson Reif *Statistical Theory and Modeling for Turbulent Flows* John Wiley and Sons, Chichester, 2nd edition, 2009.

[14] Software package `http://www.fenics.org`, 2011.

[15] Software package `http://home.gna.org/getfem/`, 2011.

[16] J. L.Guermond, P.Minev and J.Shen An overview of projection methods for incompressible flows *Comput. Methods Appl. Mech. Eng.*, **vol.41**, 112–134, 2006.

[17] K.Hanjalić, M. P.M and Hadvziabdić A robust near-wall elliptic-relaxation eddy-viscosity turbulence model for CFD *International journal of heat and fluid flow*, **vol.25**, 1047–1051, 2004.

[18] F.Ilinca, D.Pelletier and A.Garon An adaptive finite element method for a two-equation turbulence model in wall-bounded flows *Int. J. Num. Methods in Fluids*, **vol.24**, 101–120, 2005.

[19] W. P.Jones and B. E.Launder The prediction of laminarization with a two-equation model of turbulence *Int. J. Heat Mass Transfer*, **vol.15**, 301–314, 1972.

[20] R. C.Kirby and A.Logg Benchmarking domain-specific compiler optimizations for variational forms *ACM Trans. Math. Software*, **vol.35**(2), 1–18, 2008.

[21] B. E.Launder and B. I.Sharma Application of the energy-dissipation model of turbulence to the calculation of flow near a spinning disc *Letters in Heat and Mass Transfer*, **vol.1**(2), 131–138, 1974.

[22] D. R.Laurence, J. C.Uribe and S. V.Utyuzhnikov A robust formulation of the v2-f model *Flow, Turb. and Comb.*, **vol.73**, 169–185, 2004.

[23] F. S.Lien and P. A.Durbin Non-linear k-e-v2 modelling with application to high lift In *Proceeding of the Summer Program, Stanford University*, 1995.

[24] F. S.Lien and G.Kalitzin Computations of transonic flow with the $v^2 - f$ turbulence model *International journal of heat and fluid flow*, **vol.22**, 53–61, 2001.

[25] A.Logg and G. N.Wells DOLFIN: Automated finite element computing *ACM Trans. Math. Software*, **vol.37**(2), 20:1–20:28, 2010.

[26] C.Lübon, M.Kessler and S.Wagner A parallel CFD solver using the discontinuous Galerkin approach In S.Wagner, M.Steinmetz, A.Bode and M.Brehm, editors, *High Performance Computing in Science and Engineering*. Springer, 2009.

[27] M.Mortensen, H. P.Langtangen and G.Wells A FEniCS-based programming framework for modeling turbulent flow by the Reynolds-averaged Navier-Stokes equations *Advances in Water Resources*, 2011 DOI: 10.1016/j.advwatres.2011.02.013.

[28] K. B.Ølgaard, A.Logg and G. N.Wells Automated code generation for discontinuous Galerkin methods *SIAM J. Sci. Comput.*, **vol.31**(2), 849–864, 2008.

[29] K. B.Ølgaard and G. N.Wells Optimisations for quadrature representations of finite element tensors through automated code generation *ACM Transactions on Mathematical Software*, **vol.37**(1), 8:1–8:23, 2010.

[30] The open source CFD toolbox `http://www.openfoam.com`, 2011.

[31] Software package `https://computation.llnl.gov/casc/Overture`, 2011.

[32] S. B.Pope *Turbulent flows* Cambridge University Press, 2000.

[33] Software package `https://adh.usace.army.mil/proteus/`, 2011.

[34] Software package `https://computation.llnl.gov/casc/SAMRAI`, 2011.

[35] R. M.Smith A practical method of two-equation turbulence modelling using finite elements *Int. J. Num. Methods in Fluids*, **vol.4**, 321–336, 2005.

[36] P. R.Spalart and S. R.Allmaras A one-equation turbulence model for aerodynamic flows *AIAA Journal*, pages 92–439, 1992.

[37] G. N.Wells Multiphase flow through porous media In A.Logg, K.-A.Mardal and G. N.Wells, editors, *Automated Scientific Computing*. Springer, 2011 To appear, `https://launchpad.net/fenics-book`.

[38] D.Wilcox Re-assessment of the scale-determining equation for advanced turbulence models *AIAA Journal*, **vol.26**, 1414–1421, 1988.

[39] Software package `https://launchpad.net/fenics-group`, 2011.