Structural and Multidisciplinary Optimization (2021) 64:4409–4440
https://doi.org/10.1007/s00158-021-03061-4

EDUCATIONAL PAPER

Flexible framework for fluid topology optimization with OpenFOAM® and finite element-based high-level discrete adjoint method (FEniCS/dolfin-adjoint)

Diego Hayashi Alonso[1] · **Luis Fernando Garcia Rodriguez**[1] · **Emílio Carlos Nelli Silva**[1]

Received: 11 December 2020 / Revised: 16 July 2021 / Accepted: 19 August 2021 / Published online: 26 September 2021

Abstract
In order to implement the topology optimization method, it is necessary to simulate the fluid flow dynamics and also obtain the sensitivities with respect to the design variable (such as through the adjoint method). However, more complex fluid flows, such as turbulent, non-Newtonian, and compressible flows, may turn the implementation of these two aspects difficult and non-intuitive. In order to solve this deadlock, this work proposes the combination of two well-known and established open-source softwares: OpenFOAM® and FEniCS/dolfin-adjoint. OpenFOAM® already provides efficient implementations for various fluid flow models, while FEniCS, when combined with the dolfin-adjoint library, provides an efficient and automatic high-level discrete adjoint model. There have been various attempts for obtaining the adjoint model directly in OpenFOAM® , but they mostly rely on the following: (1) manually deducing the adjoint equations, which may become a hard and cumbersome task for complex models; (2) C++ automatic differentiation tools, which are generally computationally inefficient; and (3) finite differences, which have been developed for shape optimization (not topology optimization, where there are many more design variable values). Nonetheless, these approaches generally do not provide an easy setup, and may be fairly complex to consider. The FEniCS platform does not provide any fluid flow model out of the box, but makes it fairly simple to "simplistically" define them. The main problem of the FEniCS implementation and even implementations "by hand" (such as in C++, Matlab® or Python) is the convergence of the simulation, which would possibly require fairly complex adjustments in the implementation in order to reach convergence. Therefore, the combination proposed in this work (OpenFOAM® and FEniCS/dolfin-adjoint) is a simpler but efficient approach to consider more complex fluid flows, countering the difficult adjoint model implementation in OpenFOAM® and also the convergence issues in FEniCS. The implemented framework, referred as "FEniCS TopOpt Foam", can perform the coupling between the two softwares. Numerical examples are presented considering laminar and turbulent flows (Spalart-Allmaras model) for 2D, 2D axisymmetric, and 3D domains.

Keywords Fluid topology optimization · Discrete adjoint method · Turbulence · OpenFOAM® · FEniCS · dolfin-adjoint

1 Introduction

Topology optimization is the optimization method which relies on distributing a given design variable (which, in this work, represents the solid/fluid material) over a design domain. This method was originally considered for structural optimization (Rozvany et al. 1992; Rozvany 2001), but was later introduced in fluid flow problems (Borrvall and Petersson 2003). The first approach that has been considered in topology optimization is the "pseudo-density approach", but there are also other approaches, such as the "level-set method" (Duan et al. 2016; Zhou and Li 2008), and topological derivatives (Sokolowski and Zochowski 1999; Sá et al. 2016). In this work, topology optimization is considered through the "pseudo-density approach".

From the initial work of topology optimization for fluids, various other types of fluid flow types have been considered, such as Stokes flows (Borrvall and Petersson 2003), Navier-Stokes flows (Evgrafov 2004; Olesen et al. 2006), Darcy-Stokes flows (Guest and Prévost 2006; Wiker et al.

Responsible Editor: Qing Li

✉ Emílio Carlos Nelli Silva
ecnsilva@usp.br

[1] Department of Mechatronics and Mechanical Systems Engineering, Polytechnic School of the University of São Paulo, São Paulo, SP, Brazil

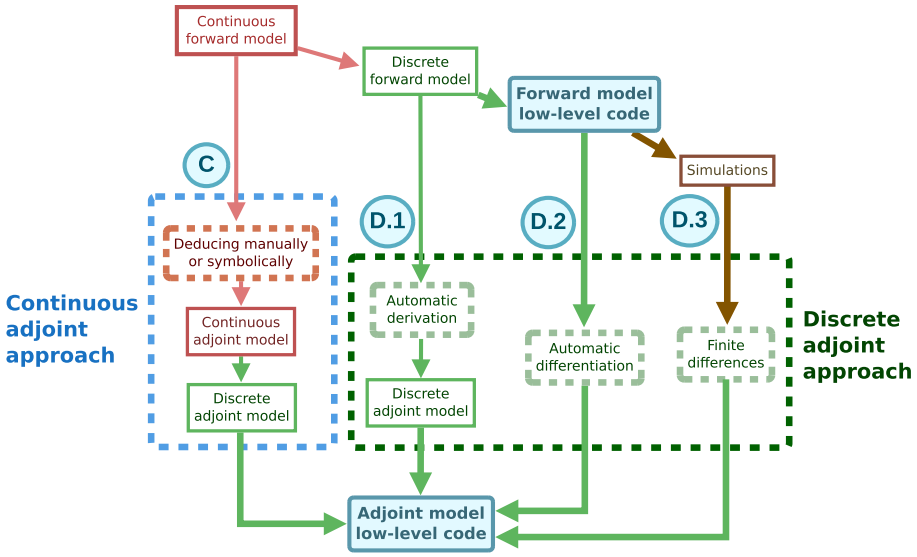


Fig. 1 Diagram illustrating the continuous adjoint approach and some possibilities of the discrete adjoint approach [figure based on Farrell et al. (2013) and Funke (2013)]

2007), compressible flows (Sá et al. 2021), non-Newtonian flows (Pingen and Maute 2010; Hyun et al. 2014; Alonso et al. 2020), thermal-fluid flows (Sato et al. 2018; Ramalingom et al. 2018; Lv and Liu 2018), turbulent flows (Papoutsis-Kiachagias et al. 2011, 2015; Yoon 2016; Dilgen et al. 2018), 2D swirl flows (Alonso et al. 2018, 2019), unsteady flows (Nørgaard et al. 2016; Hasund 2017) etc. Also, various fluid flow devices can be designed through topology optimization, such as valves (Song et al. 2009; Sato et al. 2017), mixers (Andreasen et al. 2009; Deng et al. 2018), rectifiers (Jensen et al. 2012), and flow machine rotors (Romero and Silva 2014, 2017; Zhang et al. 2016).

When performing topology optimization, it is necessary to compute the sensitivities for all of the distributed design variable values inside the design domain. One way to efficiently compute them is by considering the adjoint model. For this, there are essentially two approaches: the continuous adjoint approach and the discrete adjoint approach (see Fig. 1).

The continuous adjoint approach (indicated by the label "C" in Fig. 1) consists of directly specifying the adjoint equations and may be implemented by deriving the adjoint equations manually ("by hand") [or symbolically, by using, for example, the SymPy library (Meurer et al. 2017)]. However, this approach is specific to each problem (Papoutsis-Kiachagias et al. 2011, 2015), may be laborious (Funke 2013), and even when it is symbolically derived, the adjoint equations may be presented in a format that is not computationally efficient. In this last case, the equations would normally require further manipulation in order to get to a computationally efficient format. The implementation of the adjoint model may become highly non-intuitive, especially when considering more complex fluid flow modeling, such as turbulent, non-Newtonian, and compressible flows. When considering the finite volume method, the resulting continuous adjoint model equations are normally solved in the same way as the simulation, such as from the iterative SIMPLE (Semi-Implicit Method for Pressure-Linked Equations) algorithm (Patankar 1980; OpenFOAM Wiki 2014). It can also be mentioned that it should also be possible to derive the continuous adjoint model equations for a coupled approach (i.e., a single equation) in OpenFOAM® (Mangani et al. 2014).

The discrete adjoint approach would consist of using, for example, a low-level approach, from C++ generic automatic differentiation (AD) tools [such as CoDiPack (Sagebaum et al. 2018) and Adept (Adept 2021)] (indicated by the label "D.2" in Fig. 1), which are normally considered to be non-intuitive and may be computationally inefficient (since the low-level C++ code would have to be automatically differentiated at each iteration of the optimization). More into the implementation in OpenFOAM®, Towara and Naumann (2013) use a SIMPLE iterative scheme to solve the adjoint model and obtain the adjoint variables. An alternative is by performing finite differences (He et al. 2018, 2020) (indicated by the label "D.3" in Fig. 1), which is automated, but there may be a significant increase in the computational cost of the topology optimization.

Another way is by considering the finite element method for a single equation (coupled pressure-velocity formulation), by automatically deriving the adjoint equations in a high-level approach (i.e., in a high-level representation of the equations) (indicated by the label "D.1" in Fig. 1) (Farrell et al. 2013; Funke 2013). This way, the resulting linear system of equations can be solved directly, without the need of any iterative method such as the SIMPLE algorithm. In this work, the discrete adjoint approach is considered in this high-level representation.

The well-known and established open-source software FEniCS (based on finite elements) (Logg et al. 2012; Farrell et al. 2013; Mitusch et al. 2019) can be used for fluid flow simulations (Mortensen et al. 2011) and, when coupled with the dolfin-adjoint library, can provide an efficiently computed discrete adjoint solution from a defined forward model (indicated by the label "D.1" in Fig. 1). However, more complex fluid flow modeling may require various possibly non-intuitive adjustments to the implementation for convergence and may result in an implementation that is less efficient than what OpenFOAM® provides (Mortensen et al. 2011).

The also well-known and established open-source software OpenFOAM® (based on finite volumes) (Weller et al. 1998; Chen et al. 2014) is capable of performing efficient fluid flow simulations, but its main drawback is the computation of the adjoint model (required for computing the sensitivities), which can be a highly demanding task for the programmer (indicated by the label "C" in Fig. 1) or may result in loss of computational efficiency (indicated by the labels "D.2" and "D.3" in Fig. 1).

Therefore, this work proposes using two well-known and established open-source softwares, combining the automated method provided by FEniCS/dolfin-adjoint with the simulation computed by OpenFOAM®. In terms of implementation, this approach only requires the specification of both simulation solvers (in FEniCS and OpenFOAM®), which makes it relatively simpler to implement than the other approaches, and should be, therefore, interesting for performing fluid flow topology optimization. In relation to the continuous adjoint approach, the proposed solution using the high-level discrete adjoint approach shows an inherent computational cost due to the interfacing between OpenFOAM® and FEniCS/dolfin adjoint. However, in relation to a continuous adjoint model in OpenFOAM®, it does not require an iterative procedure (SIMPLE) to solve the adjoint model.

In the point of view of the OpenFOAM® software, the automation of the generation of the adjoint model means that any model (such as any objective function, any turbulent/compressible/non-Newtonian model) may be considered with only an additional implementation consisting of specifying the forward model both in finite elements and finite volumes, which is much easier than deriving the adjoint model by hand for a complex model. In the point of view of FEniCS/dolfin-adjoint, the fluid simulation may be computed more efficiently by using OpenFOAM® (Mortensen et al. 2011), while significantly reducing the need of complex implementations and adjustments for convergence in the FEniCS/dolfin-adjoint implementation (Mortensen et al. 2011).

Therefore, the main objective of this work is to present a framework for topology optimization by using OpenFOAM® and finite element-based high-level discrete adjoint method (FEniCS/dolfin-adjoint). The numerical examples consider the traditional material model of fluid topology optimization (Borrvall and Petersson 2003). Three types of computational domains are illustrated: 2D, 2D axisymmetric, and 3D domains. Laminar or turbulent (Spalart-Allmaras model) flows are considered. The design variable is assumed to be nodal. The objective function is the energy dissipation. OpenFOAM® (Weller et al. 1998; Chen et al. 2014) is used for the finite volume simulation, while the sensitivities are computed by the adjoint model generated by FEniCS/dolfin-adjoint (Logg et al. 2012; Farrell et al. 2013; Mitusch et al. 2019), and IPOPT (Interior-Point Optimization algorithm) is used as the optimization algorithm (Wächter and Biegler 2006). The "FEniCS TopOpt Foam" library used in the implementation of this work is to be made available in a git repository.[1]

This paper is organized as follows: in Sect. 2, the fluid flow model is described; in Sect. 3, the weak formulation (finite element method) of the problem is presented; in Sect. 4, the finite element/volume modeling is presented; in Sect. 5, the topology optimization problem is stated; in Sect. 6, the numerical implementation is described, along with the interfacing between OpenFOAM® and FEniCS/dolfin-adjoint; in Sect. 7, numerical examples are presented; and in Sect. 8, some conclusions are inferred.

2 Equilibrium equations

In this work, in order to exemplify the approach of interfacing OpenFOAM® with FEniCS/dolfin-adjoint, the fluid flow modeling is performed for incompressible fluid, and steady-state regime (Munson et al. 2009; White 2011). Therefore, the continuity and linear momentum (Navier-Stokes) equations considered are:

$$\nabla \cdot \boldsymbol{v} = 0 \tag{1}$$

$$\rho \nabla \boldsymbol{v} \cdot \boldsymbol{v} = \nabla \cdot (\boldsymbol{T} + \boldsymbol{T}_R) + \rho \boldsymbol{f} - \kappa(\alpha) \boldsymbol{v}_{\text{mat}} \tag{2}$$

where $\boldsymbol{v}$ is the fluid velocity, p is the fluid pressure, ρ is the fluid density, μ is the fluid dynamic viscosity, $\rho \boldsymbol{f}$ is the body force per unit volume acting on the fluid, $\boldsymbol{f}_r(\alpha) = -\kappa(\alpha)\boldsymbol{v}_{\text{mat}}$ is the resistance force of the porous medium used in topology optimization ($\kappa(\alpha)$ is the inverse permeability ("absorption coefficient"), and $\boldsymbol{v}_{\text{mat}} = \boldsymbol{v} - \boldsymbol{v}_{\text{material}}$ is the velocity in relation to the porous material – when $\boldsymbol{v}_{\text{material}} = \boldsymbol{0}$ (i.e., the solid material is stationary), $\boldsymbol{v}_{\text{mat}} = \boldsymbol{v}$), α is the pseudo-density, which assumes values from 0 (solid) to 1 (fluid) (and is the design variable in topology optimization), and $\boldsymbol{T}$ is the fluid stress tensor given by

[1] https://github.com/diego-hayashi/fenics_topopt_foam.

$$\boldsymbol{T} = 2\mu\boldsymbol{\epsilon} - p\boldsymbol{I}, \boldsymbol{\epsilon} = \frac{1}{2}(\nabla\boldsymbol{v} + \nabla\boldsymbol{v}^T) \tag{3}$$

The term $\boldsymbol{T}_R$ in Eq. (2) is the Reynolds (turbulent) stress tensor, which appears in RANS (Reynolds-Averaged Navier-Stokes) formulations. When considering a RANS formulation, the velocity ($\boldsymbol{v}$) and pressure (p) fields refer to statistical time-averaged values.

In this work, the Spalart-Allmaras model is used for considering turbulence. The Spalart-Allmaras model (Spalart and Allmaras 1994; Bueno-Orovio et al. 2012; Wilcox 2006) is a single-equation turbulence RANS model, which is said to be adequate for mild boundary layer separations (Ansys 2006). According to Bardina et al. (1997), the Spalart-Allmaras model does not require a finer mesh resolution near walls in wall-bounded flows as two-equation turbulence models (such as k-ε and k-ω models), and shows good convergence for simpler flows. Also, it is said to show improvements in the prediction of fluid flows under adverse pressure gradients (when the pressure increases toward the outlet) when compared to the standard k-ε and k-ω models (Bardina et al. 1997). There are various modifications that have been proposed in the Spalart-Allmaras model along the years (NASA 2019). In this work, the modifications that are considered are based in the OpenFOAM® (OpenFOAM Foundation 2020) implementation. An additional term based on Yoon (2016), Dilgen et al. (2018), and Papoutsis-Kiachagias and Giannakoglou (2016) is included in order to take the effect of the modeled solid material (of topology optimization) into account. This way, the Spalart-Allmaras model is given by (OpenFOAM Foundation 2020):

$$\boldsymbol{T}_R = \mu_{\mathrm{T}}(\nabla\boldsymbol{v} + \nabla\boldsymbol{v}^T),\ \mu_{\mathrm{T}} = \rho f_{v1}\tilde{\nu}_{\mathrm{T}} \tag{4}$$

$$\begin{aligned}\rho\boldsymbol{v}\cdot\nabla\tilde{\nu}_{\mathrm{T}} = &\underbrace{c_{b1}\rho\tilde{S}\tilde{\nu}_{\mathrm{T}}}_{\text{Production}} + \underbrace{\left[-c_{w1}f_w\rho\left(\frac{\tilde{\nu}_{\mathrm{T}}}{\ell_w}\right)^2\right]}_{\text{Destruction}} \\ &+ \underbrace{\frac{1}{\sigma}\nabla\cdot((\rho(\nu+\tilde{\nu}_{\mathrm{T}})\nabla\tilde{\nu}_{\mathrm{T}})}_{\text{Diffusion (conservative)}} \\ &+ \underbrace{\frac{c_{b2}}{\sigma}\rho\nabla\tilde{\nu}_{\mathrm{T}}\cdot\nabla\tilde{\nu}_{\mathrm{T}}}_{\text{Diffusion (non-conservative)}} \\ &+ \underbrace{\left[-\lambda_{\tilde{\nu}_{\mathrm{T}}}\kappa(\alpha)\tilde{\nu}_{\mathrm{T,\,mat}}\right]}_{\text{Attenuation of turbulence in the porous medium}}\end{aligned} \tag{5}$$

where $\tilde{\nu}_{\mathrm{T}}$ is the auxiliary turbulent viscosity of the Spalart-Allmaras model, μ_{T} is the turbulent viscosity (i.e., the flow parameter which accounts for the statistical time-averaged effect of turbulence in the stress tensor), $\tilde{\nu}_{\mathrm{T,\,mat}} = \tilde{\nu}_{\mathrm{T}} - \tilde{\nu}_{\mathrm{T,\,wall}}$ is the auxiliary turbulent viscosity in relation to its "wall value" ($\tilde{\nu}_{\mathrm{T,\,wall}}$, which is assumed as equal to 0 m^2/s), and $\lambda_{\tilde{\nu}_{\mathrm{T}}}$ is an adjustable parameter for the intensity of the attenuation of turbulence inside the solid material (it can be chosen, for example, as $\lambda_{\tilde{\nu}_{\mathrm{T}}} = 1$). The other terms of Eq. (5) are specified as follows:

$$\begin{aligned}&c_{b1} = 0.1355,\ c_{b2} = 0.6220 \\ &c_{v1} = 7.1, \sigma = \frac{2}{3}, f_\Omega = 0.3 \\ &c_{w1} = \frac{c_{b1}}{\kappa^2} + \frac{1+c_{b2}}{\sigma}, c_{w2} = 0.3, c_{w3} = 2 \\ &\chi = \frac{\tilde{\nu}_{\mathrm{T}}}{\nu} \\ &\tilde{S} = \max\left[S + \frac{\tilde{\nu}_{\mathrm{T}}}{\kappa^2\ell_w^2}f_{v2}, f_\Omega\Omega_{\mathrm{m}}\right] \\ &S = \Omega_{\mathrm{m}}, \Omega_{\mathrm{m}} = \sqrt{2\boldsymbol{\Omega}\cdot\boldsymbol{\Omega}}, \boldsymbol{\Omega} = \frac{1}{2}(\nabla\boldsymbol{v} - \nabla\boldsymbol{v}^T) \\ &f_{v1} = \frac{\chi^3}{\chi^3 + c_{v1}^3}, f_{v2} = 1 - \frac{\chi}{1+\chi f_{v1}} \\ &f_w = g\left(\frac{1+c_{w3}^6}{g^6+c_{w3}^6}\right)^{1/6}, g = r_i + c_{w2}(r_i^6 - r_i) \\ &r_i = \min\left[\frac{\tilde{\nu}_{\mathrm{T}}}{\tilde{S}_r\kappa^2\ell_w^2}, 10\right], \tilde{S}_r = \max\left[\tilde{S}, 10^{-6}\right]\end{aligned} \tag{6}$$

where $\kappa = 0.41$ is the von Kármán constant, ℓ_w is the wall distance, and $\nu = \frac{\mu}{\rho}$ is the kinematic viscosity.

In fluid topology optimization, the walls change according to the distribution of the pseudo-density (α), which means that the wall distance (ℓ_w) also changes accordingly. Thus, in order to consider such changes in the simulation and in the adjoint model, a modified Eikonal equation (Yoon 2016) is considered, which is given as:

$$\ell_w = \frac{1}{G} - \frac{1}{G_0}, G_0 = \frac{1}{\ell_{\mathrm{ref}}} \tag{7}$$

$$\begin{aligned}&\underbrace{\nabla G\cdot\nabla G}_{\substack{\text{From the original}\\ \text{Eikonal equation}}} + \underbrace{\sigma_w G\left(\nabla^2 G\right)}_{\substack{\text{Elliptic diffusion}\\ \text{for allevaiting}\\ \text{non - linearities}}} = \\ &\underbrace{\left(1+2\sigma_w\right)}_{\substack{\text{For satisfying}\\ \text{inverse linear}\\ \text{behaviour}}}\ \underbrace{G^4}_{\substack{\text{From the original}\\ \text{Eikonal equation}}} + \underbrace{\gamma(\alpha)\left(G - G_0\right)}_{\substack{\text{Porous medium}\\ \text{penalization}}}\end{aligned} \tag{8}$$

where G is the reciprocal wall distance, ℓ_{ref} is a reference value for the wall distance [which leads ℓ_w to emphasize objects that are larger than it, and can be chosen, for example, as the maximum size of the elements of the mesh (largest of the maximum distances between two vertices

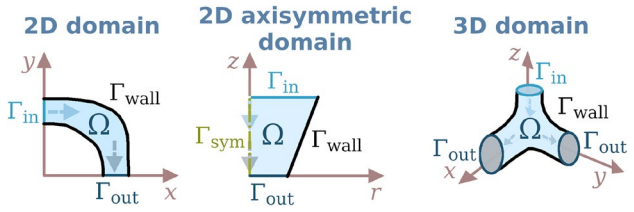


Fig. 2 Examples of boundaries for 2D, 2D axisymmetric, and 3D domains

of an element)], $\gamma(\alpha)$ is the wall penalization, which varies according to the pseudo-density (α) in order to consider the presence of a wall changing during the topology optimization, and σ_w is a relaxation factor for the wall distance computation.

2.1 Boundary value problem

The three types of computational domain considered in this work are shown in Fig. 2. It can be reminded that the definition of the differential operators and coordinates are different in the 2D axisymmetric domain (due to axisymmetry and cylindrical coordinates) (Alonso et al. 2018). Generically, a 2D axisymmetric domain may include the symmetry axis or not. The boundary value problem is specified for the three types of computational domain considered in this work as:

$$
\begin{aligned}
\rho\nabla\boldsymbol{v}\cdot\boldsymbol{v} &= \nabla\cdot(\boldsymbol{T}+\boldsymbol{T}_R)+\rho\boldsymbol{f}-\kappa(\alpha)\boldsymbol{v}_{\text{mat}} && \text{in } \Omega\\
\nabla\cdot\boldsymbol{v} &= 0 && \text{in } \Omega\\
\rho\boldsymbol{v}\cdot\nabla\tilde{\nu}_{\text{T}} &= c_{b1}\rho\tilde{S}\tilde{\nu}_{\text{T}}+\\
&\left[-c_{w1}f_w\rho\left(\frac{\tilde{\nu}_{\text{T}}}{\ell_w}\right)^2\right]+\frac{1}{\sigma}\nabla\cdot((\rho(\nu+\tilde{\nu}_{\text{T}})\nabla\tilde{\nu}_{\text{T}})+\\
&\frac{c_{b2}}{\sigma}\rho\nabla\tilde{\nu}_{\text{T}}\cdot\nabla\tilde{\nu}_{\text{T}}+\left[-\lambda_{\tilde{\nu}_{\text{T}}}\kappa(\alpha)\tilde{\nu}_{\text{T, mat}}\right] && \text{in } \Omega\\
\nabla G\cdot\nabla G+\sigma_w G(\nabla^2 G) &= \\
(1+2\sigma_w)G^4+\gamma(\alpha)(G &- G_0) && \text{in } \Omega\\
\boldsymbol{v}=\boldsymbol{v}_{in} \text{ and } \tilde{\nu}_{\text{T}} &= \tilde{\nu}_{\text{T},in} \text{ and } \nabla G\cdot\boldsymbol{n}=0 && \text{on } \Gamma_{\text{in}}\\
\boldsymbol{v}=\boldsymbol{0} \text{ and } \tilde{\nu}_{\text{T}} &= \tilde{\nu}_{\text{T},_{wall}} \text{ and } G=G_0 && \text{on } \Gamma_{\text{wall}}\\
v_r &= 0\\
\text{and } \frac{\partial v_r}{\partial r}=\frac{\partial v_z}{\partial r}=\frac{\partial p}{\partial r}=\frac{\partial\tilde{\nu}_{\text{T}}}{\partial r}=\frac{\partial G}{\partial r}&=0 && \text{on } \Gamma_{\text{sym}}\\
(\boldsymbol{T}+\boldsymbol{T}_R)\cdot\boldsymbol{n}=\boldsymbol{0} &\text{ and } \nabla\tilde{\nu}_{\text{T}}\cdot\boldsymbol{n}=0\\
\text{and } \nabla G\cdot\boldsymbol{n}&=0 && \text{on } \Gamma_{\text{out}}
\end{aligned}
\tag{9}
$$

where Ω, Γ_{in}, Γ_{wall}, Γ_{sym} , and Γ_{out} can be visualized in Fig. 2. The inlet boundary (Γ_{in}) consists of an inlet velocity profile ($\boldsymbol{v}_{\text{in}}$), an imposed auxiliary turbulent viscosity value ($\tilde{\nu}_{\text{T,in}}$), and a zero normal flux boundary condition for the reciprocal wall distance (G). On the walls (Γ_{wall}), the no-slip condition is imposed for the velocity, a fixed value is imposed for the auxiliary turbulent viscosity ($\tilde{\nu}_{\text{T, wall}} = 0\ \text{m}^2/\text{s}$), and a fixed value is imposed for the reciprocal distance (G_0). The outlet boundary (Γ_{out}) consists of an outlet stress free condition (i.e., open to the atmosphere) for the pressure-velocity formulation, where $\boldsymbol{n}$ is the normal vector to the boundaries, which points outside the computational domain. On the outlet boundary (Γ_{out}), a developed auxiliary turbulent viscosity is imposed (through zero normal flux) and a zero normal flux boundary condition is imposed for the reciprocal wall distance (G). In the 2D axisymmetric domain, if there is a symmetry axis (Γ_{sym}) bordering it, the derivatives toward the r coordinate are imposed to be zero, as well as the radial velocity.

In the boundary value problem [Eq. (9)], the wall distance may be computed separately during topology optimization, since it only depends on the current distribution of the pseudo-density (α). However, it has to be later included in the adjoint model.

3 Finite element method

In order to automatically derive the adjoint model, it is needed to specify the weak form of the finite element method in FEniCS. The weak form is defined as follows.

3.1 Weak form

In the finite element method, the equilibrium equations are modeled by a corresponding weak form. In the following equations, the computational domain is represented as $d\Pi$, and the boundary of the computational domain is represented as $d\Gamma_{\Pi}$. For 2D and 3D flows, $d\Pi = d\Omega$ and $d\Gamma_{\Pi} = d\Gamma$, while, for 2D axisymmetric flow, $d\Pi = 2\pi r d\Omega$ and $d\Gamma_{\Pi} = 2\pi r d\Gamma$. By considering the weighted-residual and Galerkin methods for the mixed (velocity-pressure) formulation, (Reddy and Gartling 2010; Alonso et al. 2018)

$$
R_c = \int_{\Pi}[\nabla\cdot\boldsymbol{v}]w_p d\Pi \tag{10}
$$

$$
\begin{aligned}
R_m = &\int_{\Pi}\left[\rho\nabla\boldsymbol{v}\cdot\boldsymbol{v}-\rho\boldsymbol{f}\right]\boldsymbol{w}_v d\Pi+\int_{\Pi}\boldsymbol{T}\cdot\left(\nabla\boldsymbol{w}_v\right)d\Pi\\
&-\oint_{\Gamma_{\Pi}}\left(\boldsymbol{T}\cdot\boldsymbol{w}_v\right)\cdot\boldsymbol{n}d\Gamma_{\Pi}-\int_{\Pi}\boldsymbol{f}_r(\alpha)\cdot\boldsymbol{w}_v d\Pi
\end{aligned}
\tag{11}
$$

$$
\begin{aligned}
R_{SA} = & \int_{\Pi} \left[\rho \boldsymbol{v} \cdot \nabla \tilde{\nu}_{\mathrm{T}} - c_{b1} \rho \widetilde{S} \tilde{\nu}_{\mathrm{T}} + c_{w1} f_w \rho \left(\frac{\tilde{\nu}_{\mathrm{T}}}{\ell_w} \right)^2 - \frac{c_{b2}}{\sigma} \rho \nabla \tilde{\nu}_{\mathrm{T}} \right] w_{\tilde{\nu}_{\mathrm{T}}} d\Pi \\
& + \int_{\Pi} \frac{1}{\sigma} \left(\rho \left(\nu + \tilde{\nu}_{\mathrm{T}} \right) \nabla \tilde{\nu}_{\mathrm{T}} \right) \cdot \nabla w_{\tilde{\nu}_{\mathrm{T}}} d\Pi \\
& - \oint_{\Gamma_{\Pi}} \frac{1}{\sigma} \boldsymbol{n} \cdot \left(\rho \left(\nu + \tilde{\nu}_{\mathrm{T}} \right) \nabla \tilde{\nu}_{\mathrm{T}} w_{\tilde{\nu}_{\mathrm{T}}} \right) d\Gamma_{\Pi} \\
& - \int_{\Pi} \left[-\lambda_{\tilde{\nu}_T} \kappa(\alpha) \tilde{\nu}_{\mathrm{T,\,mat}} \right] w_{\tilde{\nu}_{\mathrm{T}}} d\Pi
\end{aligned}
\tag{12}
$$

$$
\begin{aligned}
R_w = & \int_{\Pi} \left[\nabla G \cdot \nabla G - \left(1 + 2\sigma_w \right) G^4 \right] w_G d\Pi \\
& - \int_{\Pi} \left[(\nabla G) \cdot \nabla \left(\sigma_w G w_G \right) \right] d\Pi \\
& + \oint_{\Gamma_{\Pi}} \boldsymbol{n} \cdot \left[(\nabla G) \left(\sigma_w G w_G \right) \right] d\Gamma_{\Pi} \\
& - \int_{\Pi} \left[\gamma(\alpha) \left(G - G_0 \right) \right] w_G d\Pi
\end{aligned}
\tag{13}
$$

where the subscripts "c", "m", "SA" and "w" refer to the "continuity" equation, the "linear momentum" (Navier-Stokes) equations, the "Spalart-Allmaras" equation and the "wall distance" equation (modified Eikonal equation), respectively. The test functions of the state variables (p, $\boldsymbol{v}$, $\tilde{\nu}_{\mathrm{T}}$ and G) are given by w_p, $\boldsymbol{w}_v$, $w_{\tilde{\nu}_{\mathrm{T}}}$ and w_G, respectively. Under 2D axisymmetric flow, since the integration domain ($2\pi r d\Omega$) has a constant multiplier (2π), which does not influence when solving the weak form, Eqs. (10), (11), (12) and (13) may be optionally divided by 2π (Alonso et al. 2018, 2019).

From the mutual independence of the test functions, the equations of the weak form can be summed to a single equation:

$$
F = R_c + R_m + R_{\mathrm{SA}} + R_w = 0 \tag{14}
$$

where it is also possible to solve $R_w = 0$ separately, because the computation of the wall distance is uncoupled from the other equations, depending only on the pseudo-density (α). In such case, which is considered in this work, the two weak forms may be sequentially solved:

$$
F_1 = R_w = 0 \tag{15}
$$

$$
F_2 = R_c + R_m + R_{\mathrm{SA}} = 0 \tag{16}
$$

4 Finite element/finite volume modeling

The LBB (Ladyžhenskaya-Babuška-Brezzi) condition is a necessary condition for the numerical stability of the fluid flow simulation when considering the finite element formulation (Brezzi and Fortin 1991; Reddy and Gartling 2010; Langtangen and Logg 2016). The main effect of respecting the LBB condition is numerical, in which the pressure distribution becomes consistent with the velocity field. Some LBB-stable elements are Taylor-Hood and MINI elements. In this work, MINI elements (linear elements enriched by a bubble function) (Arnold et al. 1984; Logg et al. 2012) are used for the velocity-pressure formulation (see Fig. 3) (in 3D, the order of the bubble enrichment is increased to 4), due to their lower computational cost in relation to Taylor-Hood elements. The auxiliary turbulent viscosity of the Spalart-Allmaras model ($\tilde{\nu}_{\mathrm{T}}$) and the wall distance (ℓ_w and, therefore, G) are selected with 1st degree interpolation (P_1 element). The pseudo-density (design variable) is chosen with 1st degree interpolation (P_1 element), which also enables the possible use of a Helmholtz filter in topology optimization if needed (Lazarov and Sigmund 2010), due to the fact that this filter requires the existence of the first derivative (nonexistent for element-wise (dP_0, "DG0") variables). As can be noticed, there may be some "loss" of precision when converting between finite element and finite volume methods, due to the different interpolation schemes. In such case, it is also possible to consider different discretizations/resolutions for the OpenFOAM® and FEniCS meshes, but, in this work, for simplicity, they are assumed to be the same.

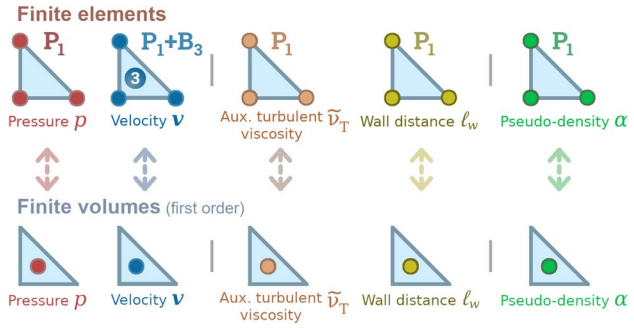


Fig. 3 Finite elements and volumes choice for the state variables: pressure, velocity, auxiliary turbulent viscosity of the Spalart-Allmaras model ($\tilde{\nu}_{\mathrm{T}}$), wall distance (ℓ_w and, therefore, G), and pseudo-density (design variable) (α)

Although Fig. 3 shows a 2D representation of the finite elements/volumes as triangles, they are implemented differently for each computational domain shown in Fig. 2 while taking into account Fig. 6: for the 2D case, the FEniCS mesh is composed of triangles, while the OpenFOAM® mesh is composed of prisms; for the 2D axisymmetric case, the FEniCS mesh is composed of triangles, while the OpenFOAM® mesh is composed of prisms/tetrahedrons/pyramids; and, for the 3D case, the FEniCS mesh is composed of tetrahedrons, as well as the OpenFOAM® mesh. The conversion between the variables in FEniCS and OpenFOAM® is detailed in Sect. 6.2.

5 Formulation of the topology optimization problem

5.1 Material model for the inverse permeability

The material model in fluid topology optimization aims to block fluid flow, while aiming to obtain a sufficiently discrete distribution for the pseudo-density (α) inside the design domain (with values 0 for solid, and 1 for fluid). The subtle transition between solid (0) and fluid (1) (binary values) is normally relaxed for better numerical conditioning, allowing an intermediate porous medium ("gray", with a pseudo-density between 0 and 1) (real values). The amount of "strength" to block the fluid is referred as "inverse permeability", which, as the name says, provides an opposite behavior to that of permeability. Borrvall and Petersson (2003) consider a convex interpolation function for the inverse permeability, given by:

$$\kappa(\alpha) = \kappa_{\max} + (\kappa_{\min} - \kappa_{\max})\alpha\frac{1+q}{\alpha+q} \tag{17}$$

where $\kappa_{\max}$ and $\kappa_{\min}$ are, respectively, the maximum and minimum values of the inverse permeability of the porous medium. The parameter $q > 0$ is a penalization parameter that controls the convexity (i.e., the relaxation) of the material model, where large values of q lead to a less relaxed material model. There is no clear rule on how q should be chosen, since the specific fluid flow topology optimization problem may behave better with either one value or another. In general, it is better not to leave the material model overly relaxed (i.e., $q \leqslant 0.01$), at least in the last optimization iterations, due to the consequently worse fluid flow blocking capacity.

5.2 Material model for the wall penalization

For the modified Eikonal equation, the material model may be based on Eq. (17), being given as the wall penalization

$$\gamma(\alpha) = \gamma_{\max} + (\gamma_{\min} - \gamma_{\max})\alpha\frac{1+q}{\alpha+q} \tag{18}$$

where $\gamma_{\max}$ and $\gamma_{\min}$ are, respectively, the maximum and minimum values of the wall penalization of the porous medium, and q is the same as in Eq. (17).

5.3 Topology optimization problem

The topology optimization problem can be formulated as follows.

$$\begin{aligned}
&\min_{\alpha}\ J(p(\alpha), \boldsymbol{v}(\alpha), \tilde{\nu}_{\mathrm{T}}(\alpha), \ell_w(\alpha), \alpha)\\
&\textbf{such that}\\
&\quad \text{Fluid volume constraint: } \int_{\Pi_\alpha} \alpha(d\Pi_\alpha) \leqslant\ fV_0\\
&\quad \text{Box constraint of } \alpha\text{: } 0\ \leqslant\ \alpha \leqslant\ 1
\end{aligned} \tag{19}$$

where f is the specified volume fraction, $V_0 = \int_{\Pi_\alpha} d\Pi_\alpha$ is the volume of the design domain (represented as Π_α), $J(p(\alpha), \boldsymbol{v}(\alpha)\tilde{\nu}_{\mathrm{T}}(\alpha), \ell_w(\alpha), \alpha)$ is the objective function, and $p(\alpha)$, $\boldsymbol{v}(\alpha)$, $\tilde{\nu}_{\mathrm{T}}(\alpha)$ and $\ell_w(\alpha)$ are the state variables obtained by solving the boundary value problem [Eq. (9)], which features an indirect dependency with respect to the design variable α.

5.4 Objective function

The objective function (J) is chosen as the energy dissipation (Φ) (Borrvall and Petersson 2003) including the turbulence effect [as in Yoon (2016)]. The energy dissipation is closely related to the head loss (Borrvall and Petersson 2003), and generally behaves well in fluid topology optimization. By considering zero external body forces,

$$\begin{aligned}
\Phi = &\int_{\Pi} \left[\frac{1}{2}(\mu + \mu_{\mathrm{T}})(\nabla\boldsymbol{v} + \nabla\boldsymbol{v}^T)\cdot(\nabla\boldsymbol{v} + \nabla\boldsymbol{v}^T)\right] d\Pi\\
&+ \int_{\Pi} \boldsymbol{\kappa}(\alpha)\boldsymbol{v}_{\mathrm{mat}}\cdot\boldsymbol{v} d\Pi
\end{aligned} \tag{20}$$

5.5 Sensitivity analysis

The sensitivity is given by the adjoint method from the finite element matrices and automatic differentiation as

$$\left(\frac{dJ}{d\alpha}\right)^* = \left(\frac{\partial J}{\partial\alpha}\right)^* - \left(\frac{\partial F}{\partial\alpha}\right)^* \boldsymbol{\lambda}_J \tag{21}$$

$$\left(\frac{\partial F}{\partial(p, \boldsymbol{v}, \tilde{\nu}_{\mathrm{T}}, \ell_w)}\right)^* \boldsymbol{\lambda}_J = \left(\frac{\partial J}{\partial(p, \boldsymbol{v}, \tilde{\nu}_{\mathrm{T}}, \ell_w)}\right)^* \tag{22}$$

(adjoint equation)

where $J = \Phi$ is the objective function, which is the energy dissipation, the weak form equation is given by $F = 0$, "*" represents conjugate transpose, and $\boldsymbol{\lambda}_J$ is the adjoint variable (Lagrange multiplier of the weak form) for this case. If the uncoupled form given by Eqs. (15) and (16) is considered, the two weak form dependencies need to be sequentially combined into a new equation for the sensitivity.

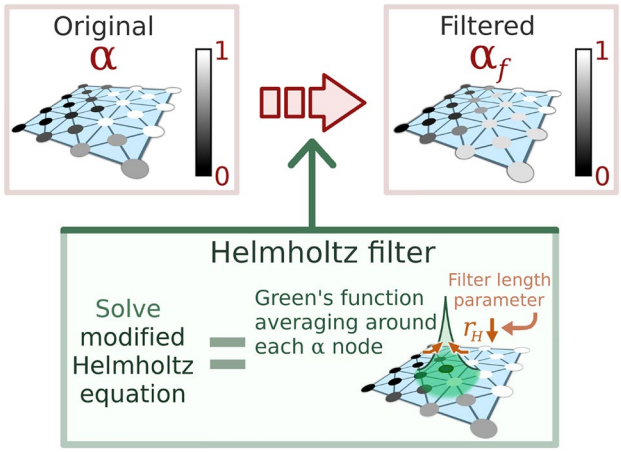


Fig. 4 Application of a Helmholtz filter

5.6 Helmholtz pseudo-density filter

Some of the topology optimization results in this work consider the use of a regularization. Regularizations are a common mechanism in topology optimization in order to counter possible numerical instabilities due to the lack of smoothness in the finite element equations (Kawamoto et al. 2013), which would possibly lead to mesh dependency and local minima (Sigmund and Petersson 1998; Bendsøe and Sigmund 2003; Sigmund 2007). The regularization that is considered is the use of a Helmholtz filter, which is a PDE-based topology optimization pseudo-density filter, having been proposed by Lazarov and Sigmund (2010). It is schematically shown in Fig. 4, where α is the original design variable and α_f is the filtered design variable.

Figure 4 illustrates the fact that the Helmholtz filter consists of weighting all values of the original design variable (α) with a Green's function, which is a function that is always positive and whose integral is equal to 1 ("100%") (Lazarov and Sigmund 2010). When choosing smaller values for the filter length parameter (r_H), this function approaches a Dirac's delta function $\left(\alpha_f \xrightarrow{r_H \to 0^+} \alpha\right)$. This "Green's function" averaging is the same as solving a modified Helmholtz equation with homogeneous Neumann boundary conditions, whose boundary value problem is given by (Lazarov and Sigmund 2010; Zauderer 1989)

$$\begin{aligned} -r_H^2 \nabla^2 \alpha_f + \alpha_f &= \alpha \qquad \text{in } \Pi \\ \frac{\partial \alpha_f}{\partial \boldsymbol{n}} &= \mathbf{0} \qquad \text{on } \Gamma_\Pi \end{aligned} \tag{23}$$

where α is the original design variable, α_f is the filtered design variable, and r_H is the filter length parameter.

The weak form is obtained by multiplying Eq. (23) by the test function w_{HF} and integrating in the whole design domain, which leads to

$$\begin{aligned} r_H^2 \int_\Pi (\nabla \alpha_f) \cdot \nabla w_{HF} d\Pi + \int_\Pi \alpha_f w_{HF} d\Pi \\ - \int_\Pi \alpha w_{HF} d\Pi = 0 \end{aligned} \tag{24}$$

When a Helmholtz filter is considered, the value given by α_f is used in the place of α in all other equations, and the sensitivities need to include the dependency of α_f in relation to α [i.e., from the chain rule for derivatives $(\frac{\mathrm{d}J}{\mathrm{d}\alpha} = \frac{\mathrm{d}J}{\mathrm{d}\alpha_f}\frac{\mathrm{d}\alpha_f}{\mathrm{d}\alpha})$] (Lazarov and Sigmund 2010).

6 Numerical implementation of the optimization problem

The fluid flow simulation is solved in the finite volumes software OpenFOAM® (version from "The OpenFOAM foundation") (Weller et al. 1998; Chen et al. 2014), by using the SIMPLE (Semi-Implicit Method for Pressure-Linked Equations) algorithm (Patankar 1980; OpenFOAM Wiki 2014). The implementation of the SIMPLE algorithm is practically the same as the "`simpleFoam`" solver from OpenFOAM®, but including the additional inverse permeability term shown in Eq. (2). Then, the additional inverse permeability term is also included in the Spalart-Allmaras model in OpenFOAM®. The adjoint model is computed in the finite elements software FEniCS (Logg et al. 2012) through dolfin-adjoint (Farrell et al. 2013; Mitusch et al. 2019). The topology optimization problem is solved with IPOPT (Wächter and Biegler 2006), from the interface provided by the dolfin-adjoint library.

6.1 Interfacing OpenFOAM® with FEniCS/dolfin-adjoint

The main idea for performing an interfacing between OpenFOAM® (finite volume method) with FEniCS/dolfin-adjoint (finite element method) is for efficiently computing the fluid flow simulation in OpenFOAM®, while the adjoint model can be automatically derived and computed in FEniCS/dolfin-adjoint.

FEniCS (Logg et al. 2012) is a finite element software implemented in C++ that uses automatic differentiation and a high-level language (UFL) for representing the weak form and functionals for the finite element matrices. From its high-level notation, the adjoint model can be automatically derived from the weak form and objective functions by the dolfin-adjoint library (Farrell et al. 2013; Mitusch et al.

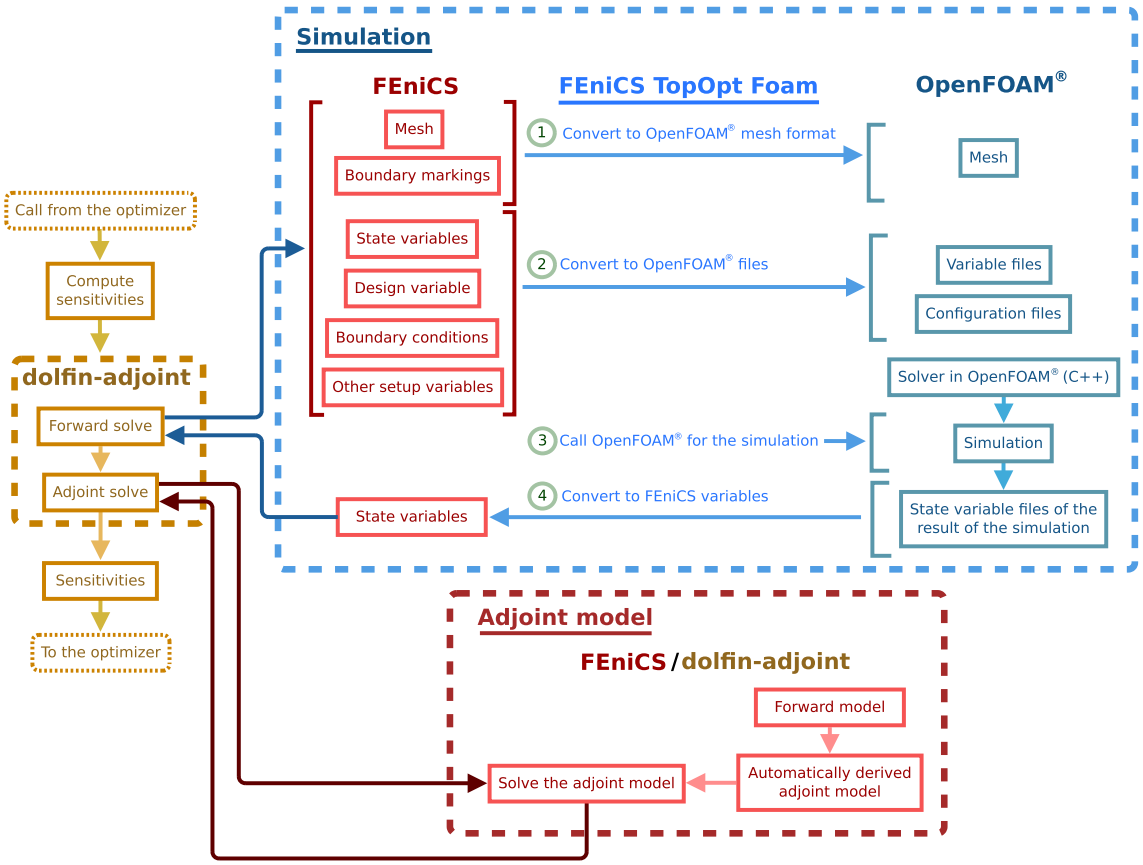


Fig. 5 Diagram illustrating the computation of the sensitivities when using OpenFOAM® and the "FEniCS TopOpt Foam" library for the fluid flow simulation

2019). The dolfin-adjoint library is restricted to the Python interface of FEniCS.

OpenFOAM® (Weller et al. 1998; Chen et al. 2014) is an open-source CFD (Computational Fluid Dynamics) software written in C++, in which the syntax for specifying the finite volume equations is, as in the case of FEniCS UFL, close to the representation of the equations themselves. Since OpenFOAM® operates in the lowest degree of finite volumes (element-wise), the simulation should become less computationally expensive than when using finite elements with the traditional Taylor-Hood elements or MINI elements for a same discretization (although the numerical precision should be lower due to the lower interpolation degree of the finite volumes in OpenFOAM®). Also, the finite volume method is based on the local conservation of fluxes (i.e., between finite volumes), which is different from the finite element method [i.e., based on the global conservation of fluxes – except for Discontinuous Galerkin finite elements (Li 2006)]. The main drawback regarding the use of OpenFOAM® in topology optimization is the derivation of the adjoint model, which was mentioned in Sect. 1.

Since dolfin-adjoint is a Python-only library, OpenFOAM®'s C++ and shell script functionalities should be made accessible in Python. The interfacing between FEniCS/dolfin-adjoint and OpenFOAM®, for topology optimization, is performed through a library developed in this work ("FEniCS TopOpt Foam").

6.2 Interfacing OpenFOAM® with dolfin-adjoint for computing the sensitivities

The objective function is computed directly with FEniCS after the simulation with OpenFOAM® is performed, while the computation of the sensitivities uses the simulation result for later solving the adjoint model equations. A diagram illustrating the computation of the sensitivities is shown in Fig. 5.

The diagram of Fig. 5 starts with a call from the optimizer for dolfin-adjoint to compute the sensitivities. The first step is computing the forward model (i.e., the simulation). It starts by passing the mesh (FEniCS "`Mesh`"), together with a boundary marking (FEniCS "`MeshFunction`") (i.e., names of each group of facets [edges (2D/2D

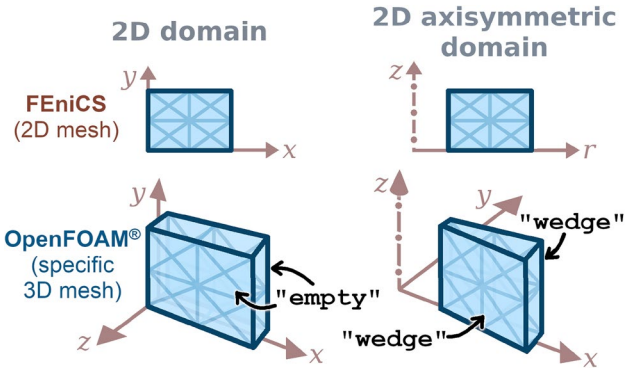


Fig. 6 Representation of 2D and 2D axisymmetric domains in FEniCS and OpenFOAM® (Obs. The specific boundary conditions "`wedge`" are presented separately, because they are imposed separately on each "almost parallel" face with respect to the 2D plane in OpenFOAM®)

axisymmetry) or faces (3D)] of the boundary), to "FEniCS TopOpt Foam" to convert to the OpenFOAM® mesh format. It can be mentioned that OpenFOAM® operates only in 3D meshes/coordinates, but allows simulating for 2D and 2D axisymmetric flows if the mesh has a specific construction [i.e., one-element uniform thickness (for the 2D mesh), and one-element "wedge" thickness (i.e., thickness linearly varying from zero radius, for a sufficiently small wedge angle) (for the 2D axisymmetric mesh)] and specific boundary conditions ["`empty`" for the parallel faces with respect to the 2D plane (of the 2D mesh), and "`wedge`" for the parallel faces with respect to the 2D plane (of the 2D axisymmetric mesh)] (see Fig. 6). Since, in OpenFOAM®, the boundary conditions are applied on the external faces of the 3D mesh, the symmetry axis boundary condition (from 2D axisymmetry) is implicitly considered when applying the "`wedge`" boundary conditions in OpenFOAM®. A similar scheme of using a 3D mesh for 2D/2D axisymmetric simulation is also used in Ansys®CFX. If the mesh is the same during all iterations of the topology optimization, this conversion can be performed a single time.

Then, the state variables (FEniCS "`Function`" 's), the design variable (FEniCS "`Function`"), the boundary conditions (specified as required by OpenFOAM®) and other setup variables are converted by "FEniCS TopOpt Foam" to variable and configuration files. The variable and configuration files in OpenFOAM® are located in three subfolders: "`0`" (initial guess for the simulation), "`constant`" (mesh and properties) and "`system`" (solver parameters). With the OpenFOAM® files prepared, a specific solver for OpenFOAM®, which corresponds to the simulation defined in FEniCS, is selected for using in the simulation. In case the simulation includes the design variable, the "default" OpenFOAM® solvers can not be used without an adjustment that includes the design variable in it (i.e., a "new" solver

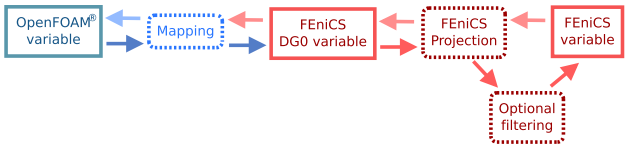


Fig. 7 Diagram illustrating the conversion of variables between OpenFOAM® and FEniCS

has to be programmed). Then, the OpenFOAM® simulation is performed. After the simulation, the state variable files of the result of the OpenFOAM® simulation are converted to the state variables in FEniCS. With the simulation result, dolfin-adjoint is now used to compute the adjoint model that is automatically generated from the forward model specified in FEniCS. The conversion from the OpenFOAM® files to the FEniCS variables (see Fig. 7) is performed by first mapping the internal values of the OpenFOAM® variables to element-wise variables in FEniCS (dP_0, "DG0"). Then, the element-wise variables are projected (FEniCS "`project`") into the interpolation that is being used in the adjoint model. The isolated state variables are then joined together in a single state vector by using a "`FunctionAssigner`" in FEniCS. In the case of turbulent variables, it may be needed to guarantee that their conversion to FEniCS is strictly positive and non-zero (compensating any numerical error that may appear in the conversions), because some turbulence models rely on some specific square-roots/divisions, and some other specific square-roots/divisions may arise due to the automatic differentiation performed by FEniCS. After this imposition, a small-radius Helmholtz filter (Lazarov and Sigmund 2010) may be applied in the turbulent variables in order to slightly filter ("alleviate") some consequent sharp transitions which may hinder post-processing operations in FEniCS. An additional step is reimposing the original Dirichlet boundary conditions (FEniCS "`DirichletBC`") onto the state vector, because the converted values from OpenFOAM® to FEniCS correspond only to the internal values of each cell and not to the external facets, which may generate numerical error on the boundaries. For the sake of completeness, the weak form that corresponds to a projection (FEniCS "`project`" function) is:

$$\int_{\Pi} a_{\text{orig}} w_p d\Pi = \int_{\Pi} a_p w_p d\Pi \tag{25}$$

where a_{orig} is the function that is being projected, while a_p is the projected function [obtained from solving Eq. (25)] and w_p is the corresponding test function for the projection.

The interfacing of the simulation with dolfin-adjoint requires "overloading" a specific internal function of the solver object in the dolfin-adjoint library, regarding the

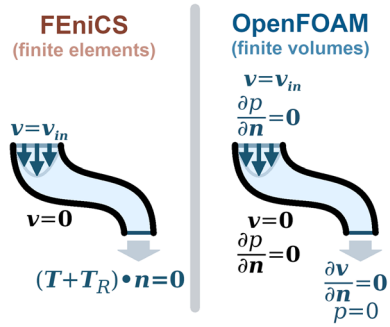


Fig. 8 Correspondence of boundary conditions for velocity and pressure between finite elements (FEniCS) and finite volumes (OpenFOAM®) considered in this work

"forward simulation" (which is called "`_forward_solve`", and is located inside the "`SolveBlock`" class).

In terms of a parallel computation of the simulation and optimization, both OpenFOAM® and FEniCS provide independent implementations of parallelism out of the box, which means that both softwares may partition the mesh differently according to their needs and what is set up by the user, and also independently call MPI operations. In the current version of "FEniCS TopOpt Foam", it is possible to consider both parallelisms independently, which means that FEniCS may be set to run in parallel, such as from "`mpiexec -n 2 python my_code.py`" (for 2 processes), while OpenFOAM® may be set up to run in parallel from "FEniCS TopOpt Foam" functions independently.

6.3 Choice of boundary conditions in OpenFOAM®

The boundary conditions that are possible to impose in OpenFOAM® may be different from the ones that are imposed in FEniCS due to the different solution methods and systems of equations (of finite volumes and finite elements, respectively). Therefore, the boundary conditions should be chosen to be with a close resemblance for corresponding simulation results. Although other variations are possible, one possibility for velocity and pressure is shown in Fig. 8. For the auxiliary turbulent viscosity of the Spalart-Allmaras model ($\tilde{\nu}_\mathrm{T}$), the boundary conditions are the same as the ones used in Eq. (9) (i.e., the same as in the finite element method). The wall distance [ℓ_w, from Eq. (8)] is computed through the finite element method and is later imported into OpenFOAM® – This procedure avoids having to implement and solve a similar equation that should aim to attain the same wall distance value from FEniCS in OpenFOAM®.

Although in finite elements (FEniCS), no boundary conditions need to be explicitly imposed for the pressure, and for the outlet velocity (because of the stress free boundary condition), OpenFOAM® (finite volumes) requires all boundary conditions to be explicitly imposed.

On the walls, the normal gradient of the pressure is set to zero ($\frac{\partial p}{\partial \boldsymbol{n}} = \mathbf{0}$) in OpenFOAM® (Neumann boundary condition). This boundary condition is originated from Prandtl's boundary layer equations (Schlichting 1979), where, inside the boundary layer, $\frac{1}{\rho}\frac{\partial p}{\partial \boldsymbol{n}} = \mathcal{O}(\delta_\mathrm{BL}) \approx \mathbf{0}$, where δ_BL is the thickness of the boundary layer, $\mathcal{O}(\delta_\mathrm{BL})$ represents the order of magnitude (i.e., in the "big O notation") of δ_BL, and the fluid is assumed to be attached to the wall. Particularly when the fluid is incompressible, $\frac{\partial p}{\partial \boldsymbol{n}} \approx \mathbf{0}$. Therefore, setting the normal gradient of the pressure to zero is an approximation. In reality, $\frac{\partial p}{\partial \boldsymbol{n}}$ is non-zero (Rempfer 2006), but the "correct" boundary condition would lead to a mathematically ill-posed problem (Rempfer 2006). According to Rempfer (2006), due to the approximation, the "pressure" value used in finite volumes numerical methods [such as the SIMPLE algorithm (Patankar 1980; OpenFOAM Wiki 2014)], would, in reality, correspond to an "articial pressure" value, which should attain a systematic deviation from the "correct" pressure value, and may be corrected due the execution of the SIMPLE algorithm.

The normal gradient of the pressure is set to zero ($\frac{\partial p}{\partial \boldsymbol{n}} = \mathbf{0}$) on the inlet in OpenFOAM® (Neumann boundary condition), because the velocity profile is already specified (Dirichlet boundary condition) and no previous knowledge outside the computational domain is known.

The outlet boundary condition in OpenFOAM® is given by imposing zero normal gradient for the velocity ($\frac{\partial \boldsymbol{v}}{\partial \boldsymbol{n}} = \mathbf{0}$) (Neumann boundary condition) and a fixed pressure value ($p = 0$) (Dirichlet boundary condition). In FEniCS, the corresponding boundary condition is selected as "stress free": $(\boldsymbol{T} + \boldsymbol{T}_R)\cdot\boldsymbol{n} = \mathbf{0}$, which corresponds to a weak imposition of a fixed zero pressure value ($p = 0$).

6.4 Topology optimization loop

The topology optimization loop is schematized in Fig. 9, showing the interconnection between the software packages. The topology optimization starts with an initial guess for the design variable (pseudo-density). Then, the forward model defined in FEniCS is "annotated" ("stored") in dolfin-adjoint for the automatic derivation of the adjoint model. The optimization loop is started with IPOPT, which interacts with dolfin-adjoint for the computation of the objective function, constraints and sensitivities from the adjoint method. The solver that includes all computations of the forward and adjoint models is referred in Fig. 9, for simplicity, as "Solver". In order to obtain the sensitivities, it is necessary to compute the forward model, which is given from the following steps: (1) The wall distance is computed in FEniCS; (2) The computed wall distance is transferred to OpenFOAM® by using the "FEniCS TopOpt Foam" library; (3) The fluid flow simulation is executed in OpenFOAM®;

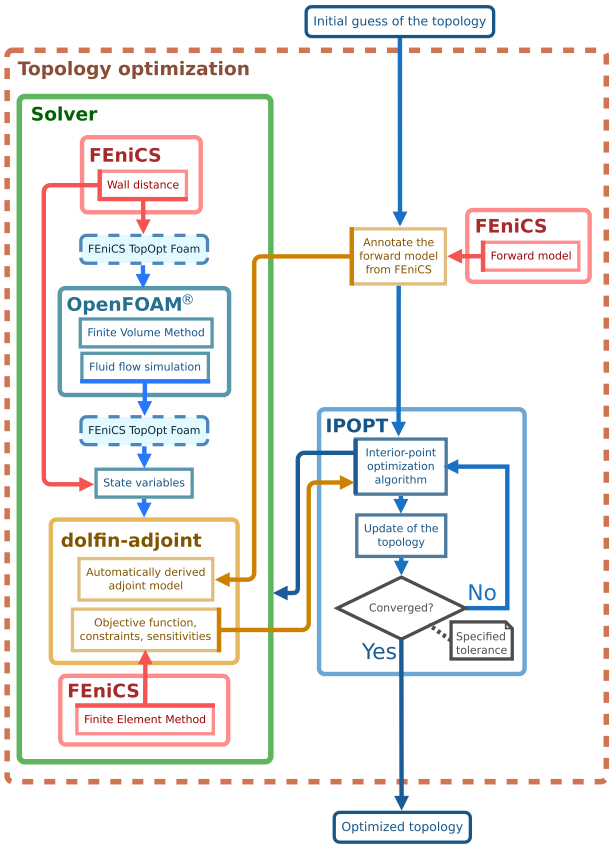


Fig. 9 Flowchart representing the topology optimization loop implemented with OpenFOAM® and FEniCS/dolfin-adjoint

(4) The fluid flow variables computed in OpenFOAM® are converted to FEniCS; (5) The converted variables and the computed wall distance are sent to dolfin-adjoint, for assembling the adjoint model. Then, the objective function, constraints and sensitivities are computed in dolfin-adjoint by using FEniCS. In each loop of the IPOPT algorithm, the values of the design variable are updated, defining new topologies. The optimization loop proceeds until a specified tolerance is reached (convergence criterion).

The computed sensitivities (of the objective function and constraint) are adjusted by the volume of each element. This is similar to considering the use of a Riesz map in the sensitivity analysis, which leads to mesh independency in the computed sensitivities. This mesh independency is particularly interesting in the case of considering non-uniform meshes, where the non-adjusted sensitivity distribution may achieve a seemingly less-smooth distribution, which may hinder the topology optimization process. For a nodal design variable, the adjusted sensitivity is given by:

$$\left.\frac{dJ}{d\alpha}\right|_{\text{adjusted}} = \frac{1}{V_{\substack{\text{neighbor elements}\\ \text{of the node}}}} \frac{dJ}{d\alpha} \underbrace{\left[\frac{\sum_{\text{nodes}} V_{\substack{\text{neighbor elements}\\ \text{of the node}}}}{n_{\text{nodes}}}\right]}_{\text{Average neighbor elements' volume}} \tag{26}$$

where $V_{\substack{\text{neighbor elements}\\ \text{of the node}}}$ is the summed volume of the neighbor elements touching a node/vertex in the mesh, and n_{nodes} is the number of nodes/vertices in the mesh. In the 2D case, the volume computations ($V_{\substack{\text{neighbor elements}\\ \text{of the node}}}$) are substituted by their area counterparts ($A_{\substack{\text{neighbor elements}\\ \text{of the node}}}$), while in the 2D axisymmetric case, the volume computations are performed considering axisymmetry (i.e., "ring-shaped" element volumes).

A comparison of the computed sensitivities from dolfin-adjoint with respect to finite differences is presented in "Appendix A".

7 Numerical examples

In the following numerical examples (with the exception of Sect. 7.1), the fluid is considered as water, with a dynamic viscosity (μ) of 0.001 Pa s, and a density (ρ) of 1000.0 kg/m^3.

An initial numerical example is performed for 2D laminar flow for checking the implementation. Then, three numerical examples (for 2D, 2D axisymmetric and 3D domains) are presented in order to illustrate the application of topology optimization with the coupling between OpenFOAM® and FEniCS/dolfin-adjoint.

The inlet velocity profiles are considered to be parabolic for the laminar flow examples, but are considered to be turbulent velocity profiles for the turbulent flow examples (see Fig. 10). The turbulent velocity profiles are implemented according to De Chant (2005), in which the velocity profile is analytically deduced from a simplified fluid flow model. The difference of this turbulent velocity profile with respect to the $1/7^{\text{th}}$ power law (Munson et al. 2009) is that the derivative is zero in the middle of the velocity profile (see the highly enlarged view of the difference in derivatives in Fig. 10). It can be reminded that this zero derivative in the middle of the turbulent velocity profile is expected for turbulent fluid flows (Munson et al. 2009). For reference, a turbulent velocity profile in the y direction, between a minimum (x_{min}) and a maximum (x_{max}) coordinate becomes (De Chant 2005):

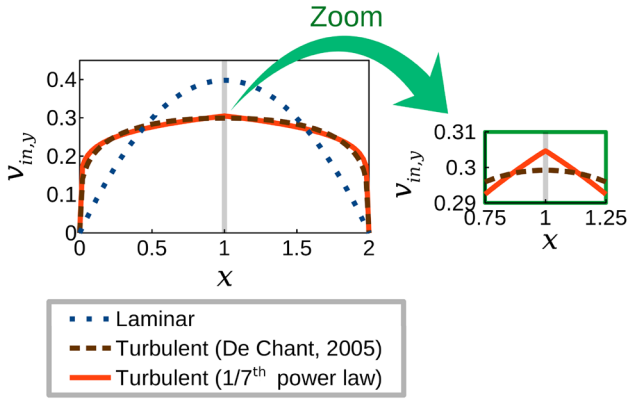


Fig. 10 Laminar and turbulent velocity profiles for the same "2D flow rate" (area below the curves)

$$v_{in,y} = v_{in,y,\max}\sqrt{\sin\left(\frac{\pi}{2}\sqrt{1-\left|\frac{x-x_{\text{middle}}}{x_1}\right|}\right)} \tag{27}$$

where $x_{\text{middle}} = \frac{x_{\max}+x_{\min}}{2}$ is the coordinate of the middle of the velocity profile, $x_1 = \frac{x_{\max}-x_{\min}}{2}$ is an auxiliary coordinate, and $v_{in,y,\max}$ is the maximum velocity of the turbulent velocity profile (computed from numerical integration for a given flow rate).

The optimization loop considers the convergence criterion as a tolerance of 10^{-10} for the optimality error of the IPOPT barrier problem, which consists of the maximum norm of the KKT conditions (Wächter and Biegler 2006).

The external body force term ($\rho \boldsymbol{f}$) is not considered in the numerical examples ($\rho \boldsymbol{f} = \boldsymbol{0}$). The porous medium is considered to be stationary ($\boldsymbol{v}_{\text{mat}} = \boldsymbol{v}$). The minimum value of the inverse permeability is considered as zero ($\kappa_{\min} = 0$ kg/(m^3 s)). The parameter $\lambda_{\tilde{\nu}_T}$ is chosen as 1.0.

The reference value for the wall distance (ℓ_{ref}) is used as the maximum size of the elements of the mesh (largest of the maximum distances between two vertices of an element), and the relaxation factor for the wall distance computation (σ_w) is chosen as 0.1. The minimum value of the wall penalization of the porous medium is considered as zero ($\gamma_{\min} = 0$ m^{-3}).

The mesh is post-processed after topology optimization has been performed (i.e., for the optimized topology), from the values of the design variable, from a threshold (step) function:

$$\alpha_{\text{th}} = \begin{cases} 1 \text{ (fluid), if } \alpha \geqslant 0.5 \\ 0 \text{ (solid), if } \alpha < 0.5 \end{cases} \tag{28}$$

where α_{th} is the thresholded function. The resulting thresholded design variable (α_{th}) is cut in order to remove the solid material ($\alpha = 0$) from the computational domain (see Fig. 11). Therefore, the final simulations are performed with the fluid flow equations without the effect of the porous medium. In all of the optimized topologies, the final values of the design variable (pseudo-density) are close to the variable bounds (0 and 1).

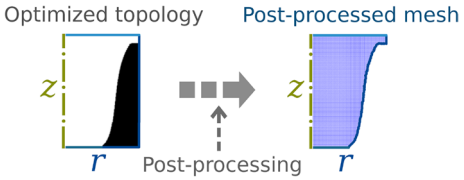


Fig. 11 Post-processing applied to an optimized topology

The post-processed simulations are computed entirely in OpenFOAM®, which means that a "default" OpenFOAM® wall distance calculation method can be used in this case (such as "`meshWave`").

The inlet values for the turbulent variable ($\tilde{\nu}_{\text{T},in}$) are given from the turbulence intensity (I_T) and the turbulence length scale (ℓ_T) based on the mean absolute velocity on the inlet ($|\boldsymbol{v}_{\text{abs},in}|$), as:

$$\tilde{\nu}_{\text{T},in} = \sqrt{\frac{n_v}{2}} I_T \ell_T \overline{|\boldsymbol{v}_{\text{abs},in}|} \tag{29}$$

where $\overline{|\boldsymbol{v}_{\text{abs},in}|} = \frac{\int_{\Gamma_{\text{II,in}}} |\boldsymbol{v}_{\text{abs},in}| d\Gamma_{\text{II,in}}}{\int_{\Gamma_{\text{II,in}}} d\Gamma_{\text{II,in}}}$ is the mean absolute velocity on the inlet, and n_v is the number of velocity components (for 2D, $n_v = 2$; for 2D axisymmetry and 3D, $n_v = 3$).

The maximum inlet Reynolds number (considering only the inlet velocity) and the maximum local Reynolds number (considering the local velocities) are defined as, respectively,

$$\text{Re}_{\text{in,max}} = \frac{\mu |\boldsymbol{v}_{\text{abs},in}|_{\max} L_{\text{ref}}}{\rho} \tag{30}$$

$$\text{Re}_{\text{ext},\,\ell,\,\max} = \frac{\mu |\boldsymbol{v}_{\text{abs}}|_{\max} L_{\text{ref}}}{\rho} \tag{31}$$

where L_{ref} is a characteristic length given, in this work, as the inlet diameter (in the 2D case, it is given as the width of the inlet).

In order to accelerate the execution of the optimization, the OpenFOAM® simulation for each optimization step reuses the simulation result from the immediately previous optimization step. A maximum number of SIMPLE iterations per optimization step is also considered, which is set, in this work, as 500~2000.

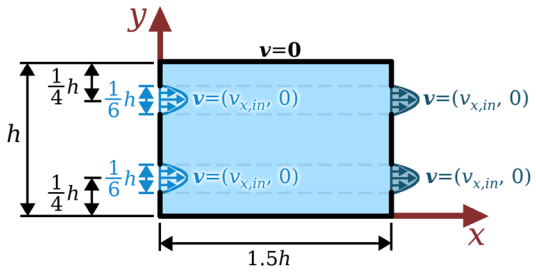


Fig. 12 Design domain for the laminar flow 2D double pipe (Borrvall and Petersson 2003)

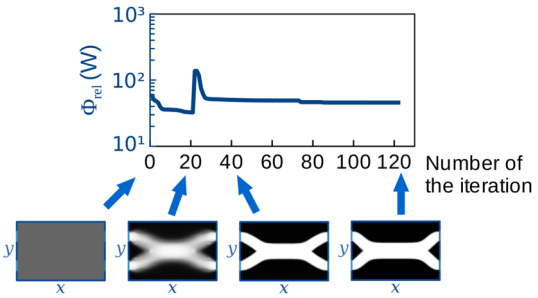


Fig. 14 Convergence curve for the laminar flow 2D double pipe

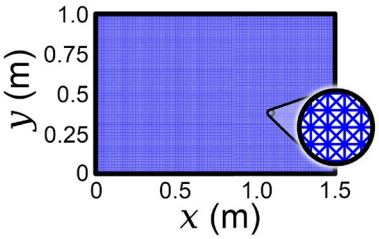


Fig. 13 Mesh used for the laminar flow 2D double pipe (check Fig. 6 for the correspondence of meshes between FEniCS and OpenFOAM®)

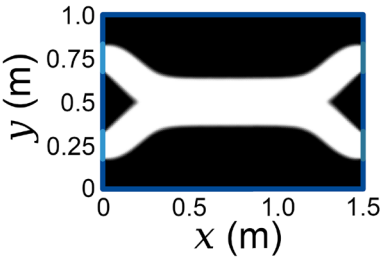


Fig. 15 Optimized topology for the laminar flow 2D double pipe

7.1 Laminar flow 2D double pipe

This initial example is for checking the implemented framework for the classical laminar flow 2D double pipe (Borrvall and Petersson 2003) (see Fig. 12). Differently from the other numerical examples, the fluid properties, topology optimization setup, boundary conditions, and dimensions are set according to Borrvall and Petersson (2003): $\mu = 1$ Pa s; $\rho = 1\,\text{kg/m}^3$; $\kappa_{\text{max}} = 2.5 \times 10^4 \mu$; $\kappa_{\text{min}} = 2.5 \times 10^{-4} \mu$; q is set as 0.01 for 20 iterations, and then changed to 0.1; the specified fluid volume fraction (f) is selected as $\frac{1}{3}$; parabolic velocity profiles are imposed (also including outlet velocity profiles) with the maximum value of the parabolas set as 1 m/s; and $h = 1$ m. Particularly, in this work, the more generic Navier-Stokes flow implementation is considered, which should not deviate much from the original Stokes flow results, since the Navier-Stokes equations tend to the Stokes equations when the Reynolds number is much smaller than 1 (in this case, the maximum inlet Reynolds number is equal to 0.17). The initial guess for topology optimization is chosen as "fluid fraction" ($\alpha = f - 1\%$, where 1% is a margin, in order to avoid the fluid volume constraint to be violated due to numerical precision). The mesh is composed of 30,251 nodes and 60,000 elements (see Fig. 13).

The convergence curve for the laminar flow 2D double pipe is shown in Fig. 14.

The optimized topology for the laminar flow 2D double pipe is shown in Fig. 15. As can be seen, the optimized topology is the same as Borrvall and Petersson (2003), which shows that the proposed framework is able to achieve the classical laminar flow 2D double pipe optimized topology.

7.2 2D bend channel

The second example is the design of the classical 2D bend channel. This numerical example has been extensively treated in topology optimization, such as for Stokes flow (Borrvall and Petersson 2003), Navier-Stokes flow (Gersborg-Hansen 2003; Dai et al. 2018), and turbulent flows (Dilgen et al. 2018; Yoon 2016). The 2D bend channel is illustrated in Fig. 16.

The mesh is composed of 5101 nodes and 10,000 elements (see Fig. 17). The input parameters and geometric dimensions of the design domain that are used are shown in Table 1. The inlet flow rates correspond to maximum inlet Reynolds numbers of 12.5 (for the laminar flow) and 8460.0 (for the turbulent flow). The initial guesses are chosen as "full fluid" ($\alpha = 1$) for the laminar flow case, and "fluid fraction" ($\alpha = f - 1\%$) for the turbulent flow case. The specified fluid volume fraction (f) is selected as 30%. For the wall distance computation, $\gamma_{\text{max}} = 10^{10}\,\text{m}^{-3}$. The inverse permeability (κ_{max}) and the penalization parameter (q) are selected, respectively, as $2.5 \times 10^8 \mu$ [kg/(m^{3}s)] and 0.1, for

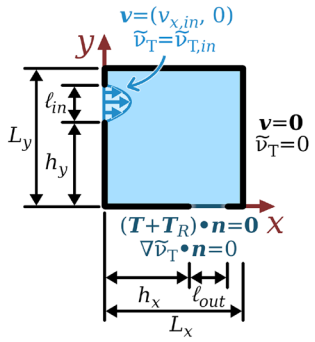


Fig. 16 Design domain for the 2D bend channel

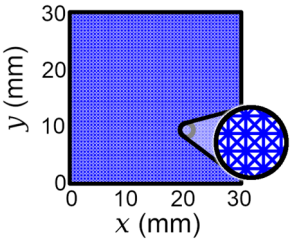


Fig. 17 Mesh used for the 2D bend channel (check Fig. 6 for the correspondence of meshes between FEniCS and OpenFOAM®)

the laminar flow; and as $1.5 \times 10^9 \mu$ [kg/(m^{3}s)] and 0.1, for the turbulent flow.

The optimized topology for laminar flow is consistent with Borrvall and Petersson (2003), because the optimized topology directly connects the inlet to the outlet, in almost a straight line. In the optimized topology for turbulent flow, due to this same fact, and also due to the optimized channel slight bulging toward the origin ((0, 0) coordinates), it bears some resemblance to some of the results from Yoon (2016), but is essentially different mainly because of the different volume fraction (Yoon (2016) considered $f = 20\%$), different problem dimensions, fluid properties, boundary conditions and Reynolds numbers.

The convergence curves for the 2D bend channel are shown in Fig. 18.

The simulation results for the post-processed meshes are shown in Fig. 19. The maximum local Reynolds numbers are computed as 143 (for the laminar flow) and 2.7×10^5 (for the turbulent flow). The energy dissipation values in the post-processed meshes are 6.66×10^{-8} W/m (for the laminar flow) and 1.48 W/m (for the turbulent flow). The difference in magnitude of the energy dissipation values is expected, because the fluid velocities are much higher in the turbulent flow, and also because of the presence of the turbulent viscosity in Eq. (20), for turbulent flow. As can be noticed in Fig. 19, the topology optimization results show different formats for both cases: the optimized topology for the laminar flow case shows a direct connection between inlet and outlet, with a small bulging toward the origin ((0, 0) coordinates) of the left side of the channel, due to the change of direction near the inlet, probably in order to redirect the fluid flow toward the outlet; the optimized topology for the turbulent flow case is more bent to the left, which is probably due to the higher viscosity (due to the turbulent viscosity) that is formed to the left of the channel. For reference, the maximum turbulent viscosity ratio, which is a simple measure of the influence of the turbulence in the simulation, is given as $\max(\frac{\mu_T}{\mu}) = 40$, which shows that the effect of the turbulent viscosity is high in at least a part of the computational domain.

7.3 2D axisymmetric nozzle

The third example is a design that relies on 2D axisymmetry, which is considered in the design of a nozzle. A nozzle is a device that is used to control the fluid flow characteristics

Table 1 Parameters used for the topology optimization of the 2D bend channel

Input parameters (laminar flow)	
Inlet flow rate (Q)	0.0022 L/min*
Inlet velocity profile	Parabolic
Input parameters (turbulent flow)**	
Inlet flow rate (Q)	2.5 L/min*
Inlet velocity profile	Turbulent
I_T	5.0%
ℓ_T	0.186 mm
Dimensions	
$L_y = L_x$	30 mm
$h_y = h_x$	18.75 mm
$\ell_{in} = \ell_{out}$	7.5 mm

*Flow rates computed assuming that the width of the inlet (ℓ_{in}) corresponds to an "inlet diameter" (in 3D)

**The turbulent case is optimized considering a Helmholtz pseudo-density filter (Sect. 5.6), where r_H is set as 0.3 mm

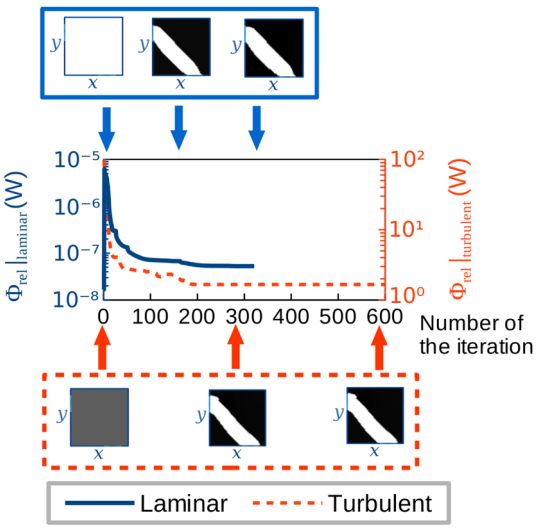
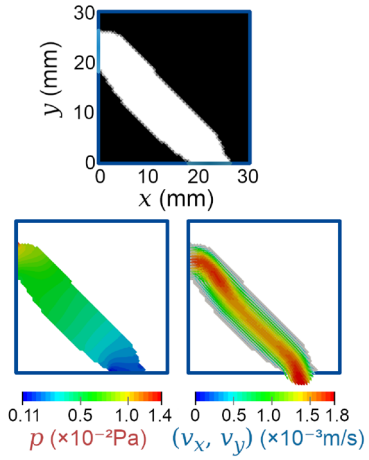
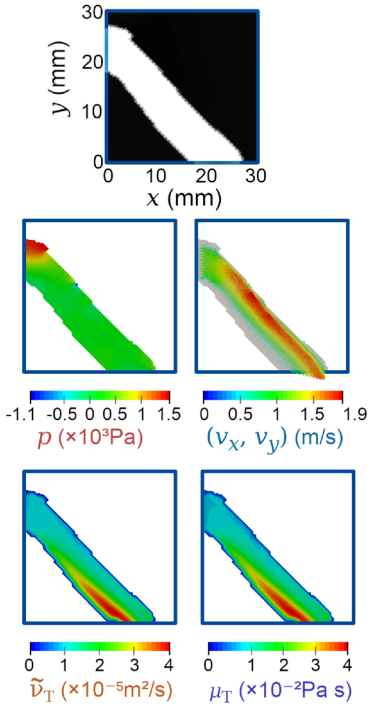


Fig. 18 Convergence curves for the 2D bend channel



(a) Optimization for laminar flow.



(b) Optimization for turbulent flow.

Fig. 19 Optimized topologies, pressure, and velocity for the 2D bend channel

entering or leaving another fluid device. This type of design is here analyzed for 2D axisymmetric flow, but has already been considered for 2D flow in Borrvall and Petersson (2003) and 2D swirl flow in Alonso et al. (2018).

In this work, as opposed to Alonso et al. (2018), where the size of the fluid flow outlet was left to be determined according to the specified fluid volume fraction (f), the size of the fluid flow outlet is fixed with a radius R_{out} (see Fig. 20). Also, in order to avoid any issue of the topology optimization blocking the low velocity part of the inlet velocity profile [as can be seen in Borrvall and Petersson (2003)], a small non-optimizable inlet height is included before the design domain.

The mesh is composed of 19,401 nodes and 38,400 elements (see Fig. 21). The input parameters and geometric dimensions of the design domain that are used are shown in Table 2. The inlet flow rates correspond to maximum inlet Reynolds numbers of 325 (for the laminar flow) and 3,253 (for the turbulent flow). In order to facilitate the convergence of the topology optimization, a "conical" initial guess (i.e., connecting the inlet (R) of the design domain ($H - h_{in}$) directly to the outlet (R_{out}) with a straight line) is considered for α. The specified fluid volume fraction (f) is selected as 50%. For the wall distance computation, $\gamma_{\max} = 10^{10}\text{m}^{-3}$. The inverse permeability ($\kappa_{\max}$) and the penalization parameter (q) are selected, respectively, as $2.5 \times 10^7 \mu$ [kg/(m^{3}s)] and 1.0, for the laminar flow; and as $5 \times 10^8 \mu$ [kg/(m^{3}s)] and 1.0, for the turbulent flow.

The convergence curves for the 2D bend channel are shown in Fig. 22.

The simulation results for the post-processed meshes are shown in Fig. 23. The maximum local Reynolds numbers are computed as 505 (for the laminar flow) and 12,023 (for the turbulent flow). The energy dissipation values in the post-processed meshes are 1.04×10^{-7} W (for the laminar flow) and 3.10×10^{-4} W (for the turbulent flow). The difference in magnitude of the energy dissipation values is expected, as in the 2D bend channel example, because of the higher fluid velocities in relation to the turbulent flow, and also because of the presence of the turbulent viscosity in Eq. (20) for turbulent flow. As can be noticed in Fig. 23a, the laminar case topology features a small bump near the low

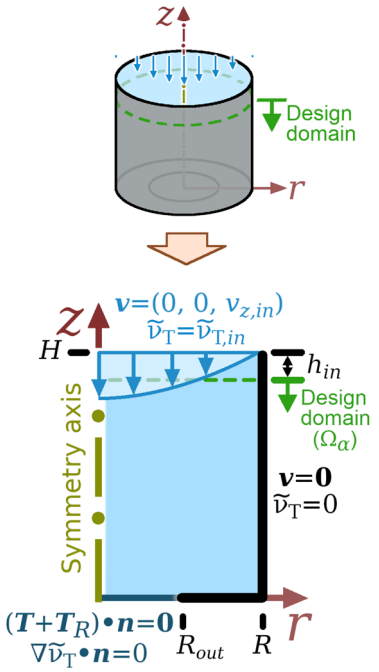


Fig. 20 Design domain for the 2D axisymmetric nozzle

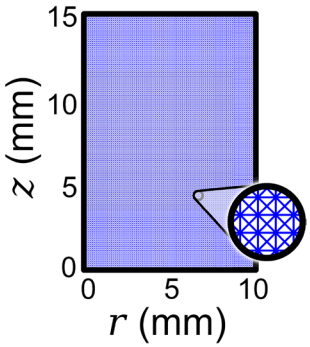


Fig. 21 Mesh used for the 2D axisymmetric nozzle (check Fig. 6 for the correspondence of meshes between FEniCS and OpenFOAM®)

velocity part of the parabolic inlet velocity profile. This small velocity means that this zone of the fluid flow is given a lower importance with respect to the objective function in relation to the rest of the computational domain. A similar effect is also observed in Borrvall and Petersson (2003)'s nozzle example. In the optimized topology for tubulent flow (Fig. 23b), the inlet of the optimized topology becomes smoother than the optimized topology for the laminar flow case. This is probably due to the different inlet velocity profile (turbulent velocity profile), which features higher velocity values at larger radii than the parabolic velocity profile, and the inlet turbulence value, which influences the objective function near the inlet. For reference, the maximum turbulent viscosity ratio is given as $\max(\frac{\mu_T}{\mu}) = 0.73$, which shows that the effect of the turbulent viscosity is comparable to the fluid (water) viscosity in at least a part of the computational domain.

7.4 3D channel

The fourth example is based on a 3D model, for the design of a channel that bifurcates into other two. Fig. 24 shows the computational domain with the inlet channel and the two outlet channels. The inlet and outlet channels are left outside the design domain.

The mesh is composed of 18,308 nodes and 102,254 tetrahedral elements (see Fig. 25), whose quantities are slightly increased for the turbulent case (18,344 nodes and 102,720 tetrahedral elements). The input parameters and geometric dimensions of the design domain that are used are shown in Table 3. The inlet flow rates correspond to maximum inlet Reynolds numbers of 1,062 (for the laminar flow) and 2,603 (for the turbulent flow). The initial guess for the laminar case is chosen as "fluid fraction" ($\alpha = f - 1\%$), while the initial guess for the turbulent case is chosen as the optimized topology of the laminar case. The specified fluid volume fraction (f) is selected as 20%. For the wall distance computation, $\gamma_{max} = 10^8 m^{-3}$. The inverse permeability (κ_{max}) and the penalization parameter (q) are selected, respectively, as $5.0 \times 10^7 \mu$ [kg/(m^{3}s)] and 1, for the laminar flow; and as $8.0 \times 10^7 \mu$ [kg/(m^{3}s)] and 1000, for the turbulent flow.

The convergence curves for the 3D channel are shown in Fig. 26. It can be highlighted that there is a maximum number of SIMPLE iterations per optimization step (which is set, in this work, as 500), which means that the "quality" of the simulation is lower in the first iterations of the topology optimization.

The simulation results for the post-processed meshes are shown in Fig. 27, where only a slice of the scalar fields (p, $\tilde{\nu}_T$, μ_T) is plotted, for illustrative purposes. The maximum local Reynolds numbers are computed as 1,254 (for the laminar flow) and 5,645 (for the turbulent flow). The energy dissipation values in the post-processed meshes are 1.08×10^{-5} W (for the laminar flow) and 3.65×10^{-4} W (for the turbulent flow). The difference in magnitude of the energy dissipation values is expected as mentioned in the other numerical examples. It can be noticed, when comparing Fig. 27a and b, that the channels are thicker in the laminar flow case, which is probably due to the effect of the lower velocities and the higher effect of the viscosity of the fluid. In the turbulent flow case, the channels are thinner, which is probably due to the higher velocities and turbulent viscosity effect in the turbulent flow case. Also, the channels are split near the outlet in the laminar flow case, while they are split near the inlet in the turbulent flow case. This may be due to the fact that, if the channel is split near the outlet in the turbulent flow case, the fluid will be at a higher velocity, meaning that

Table 2 Parameters used for the topology optimization of the 2D axisymmetric nozzle

Input parameters (laminar flow)*	
Inlet flow rate (Q)	0.05 L/min
Inlet velocity profile	Parabolic
Input parameters (turbulent flow)*	
Inlet flow rate (Q)	2.5 L/min
Inlet velocity profile	Turbulent
I_T	5.0%
ℓ_T	0.25 mm
Dimensions	
R	10 mm
R_{out}	5 mm
H	15 mm
h_{in}	1 mm

*The optimized cases consider a Helmholtz pseudo-density filter (Sect. 5.6), where r_H is set as 0.0625 mm

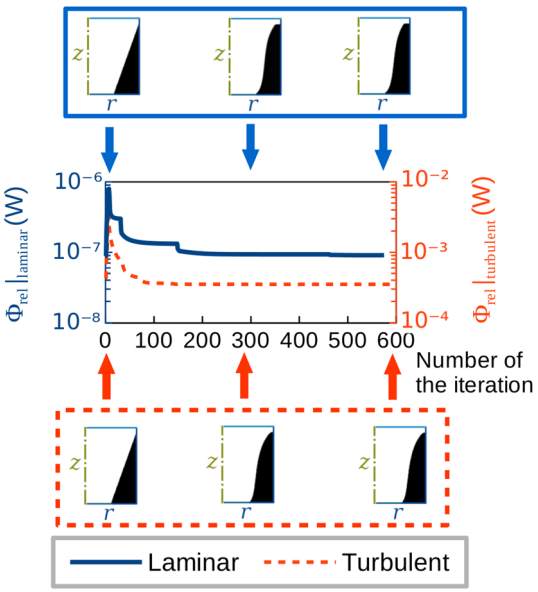
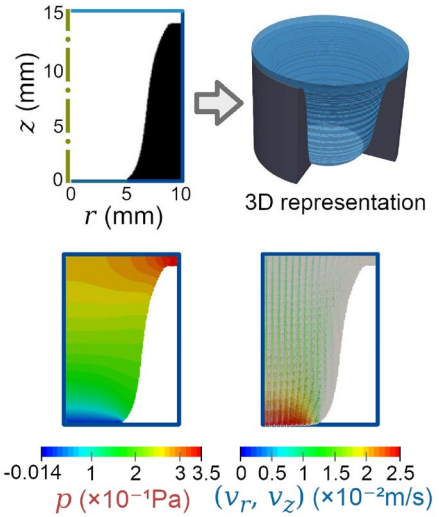


Fig. 22 Convergence curves for the 2D axisymmetric nozzle

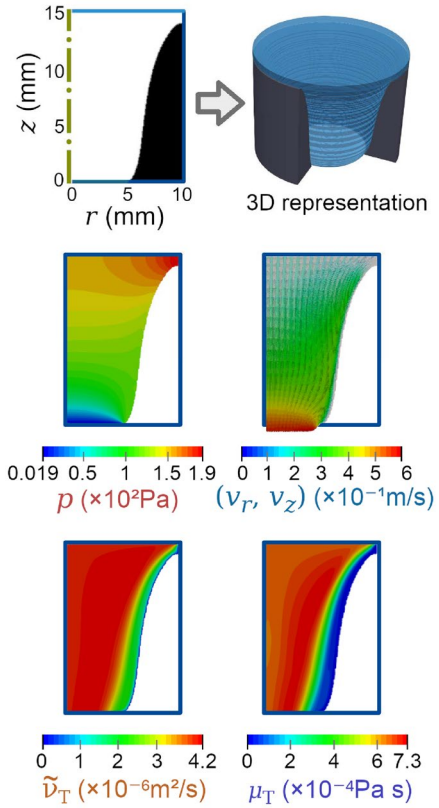
the energy dissipated in the "collision" with the "splitting edge" would become higher. One more observation is that the fluid volume is different in both optimized topologies, which is acceptable, since the constraint that is being imposed is a maximum fluid volume constraint [Eq. (9)]. For reference, the maximum turbulent viscosity ratio is given as $\max(\frac{\mu_T}{\mu}) = 6.24$, which shows that the effect of the turbulent viscosity is higher than the fluid (water) viscosity in at least a part of the computational domain.

8 Conclusions

This work presents the approach of using the OpenFOAM® infrastructure for the computation of an efficient fluid flow simulation, while the adjoint model is automatically derived in an efficient manner by FEniCS/dolfin-adjoint. Although an even higher computational efficiency would be possible to be achieved through manually deriving the continuous adjoint model and adjusting its implementation (such as through reordering the terms/operations, block matrices, local preconditionings etc.), this procedure may become a hard and cumbersome task, especially for complex models. Therefore, this work presents a more convenient and comprehensive approach of obtaining the automatically derived adjoint model in an efficient manner when considering OpenFOAM®. In the point of view of OpenFOAM®, this means that the adjoint equations do not need to be derived by hand, while, in the point of view of FEniCS, the fluid flow simulation may be computed more efficiently, without needing to implement various adjustments for convergence of the algorithm. In terms of work required in the implementation, the additional work is to write the material model terms in the equations inside the OpenFOAM® solver and write the weak forms and boundary conditions in FEniCS. The required additional work for this implementation is far from having to derive the adjoint equations by hand, and even saves time when testing, since the derivation of the adjoint model is automated. In terms of computational cost, the implemented algorithm is able to deploy OpenFOAM® and FEniCS to run in parallel (independently), which may help in reducing the required computational time. Since the adjoint equations are linear, the resulting matrix system needs to be solved a single time at each iteration, and the computational cost is mostly due to the interpolation degrees of the state variables in finite elements (see Fig. 3), which



(a) Optimization for laminar flow.



(b) Optimization for turbulent flow.

Fig. 23 Optimized topologies, 3D representation, pressure, and velocity for the 2D axisymmetric nozzle

the authors tried reducing by considering the use of MINI elements instead of Taylor-Hood elements. It is also possible to use linear finite elements by including a stabilization term in the fluid flow equations (Reddy and Gartling 2010; Logg et al. 2012; Elhanafy et al. 2017; Langtangen et al. 2002; Franca 1992).

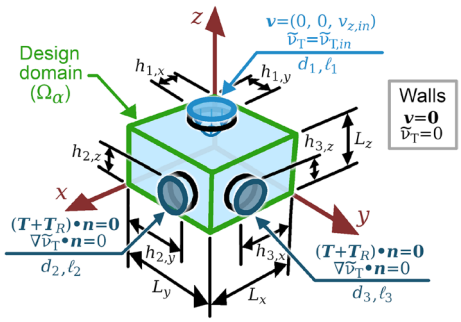


Fig. 24 Design domain for the 3D channel

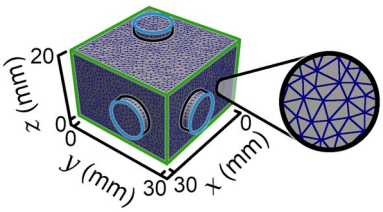


Fig. 25 Mesh used for the 3D channel (laminar case)

It is also possible to extend the implemented approach to any type of optimization method implemented in the FEniCS platform, by including the adequate conversions to OpenFOAM®simulations by using the "FEniCS TopOpt Foam" library. Although this work is focused in topology optimization for fluid flow, this approach is extensible to any kind of physics that is modellable in OpenFOAM®.

As future work, it is suggested to consider this scheme for investigating topology optimization for turbulent, compressible, and non-Newtonian flows.

9 Replication of results

The part of the implementation that is performed in the FEniCS platform is direct from the description that is provided of the equations and numerical implementation in this article. This is because FEniCS is based on a high-level description for the variational formulation (UFL), which automates the generation of the necessary matrix equations. It may be reminded that, in the 2D axisymmetric case, the coordinates are considered to be cylindrical (i.e., the differential operators ("`grad`", "`curl`", "`div`") must be programmed by hand by using the "`Dx(var,component_num)`" or "`var.dx(component_num)`" functions, because the default operators available in FEniCS consider Cartesian coordinates).

Table 3 Parameters used for the topology optimization of the 3D channel

Input parameters (laminar flow)	
Inlet flow rate (Q)	0.5 L/min
Inlet velocity profile	Parabolic
Input parameters (turbulent flow)*	
Inlet flow rate (Q)	2 L/min
Inlet velocity profile	Turbulent
I_T	5.0%
ℓ_T	0.25 mm
Dimensions	
$L_x = L_y$	30 mm
L_z	20 mm
$d_1 = d_2 = d_3$	10 mm
$\ell_1 = \ell_2 = \ell_3$	2.5 mm
$h_{1,x} = h_{1,y}$	10 mm
$h_{2,y} = h_{3,x}$	20 mm
$h_{2,z} = h_{3,z}$	10 mm
H	15 mm

*The turbulent case is optimized considering a Helmholtz pseudo-density filter (Sect. 5.6), where r_H is set as 0.457 mm

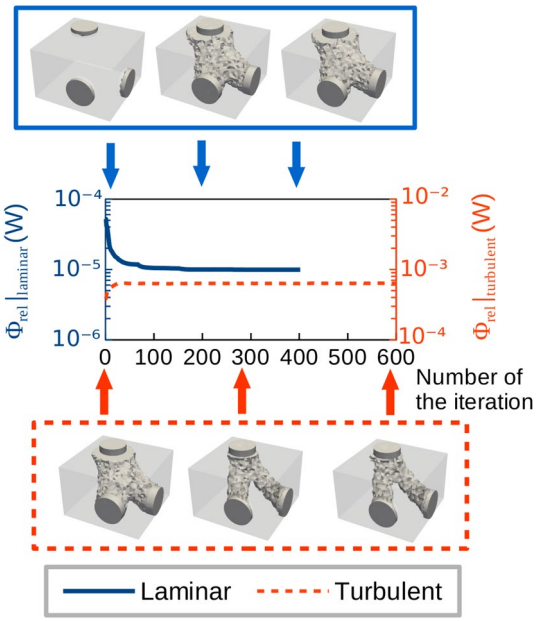


Fig. 26 Convergence curves for the 3D channel (Obs. For ease of visualization of the optimized topology, only the values of α with $\alpha \geqslant 0.5$ are shown in nontransparent color). It can be highlighted that the optimized topologies (in the final iterations) are highly discrete

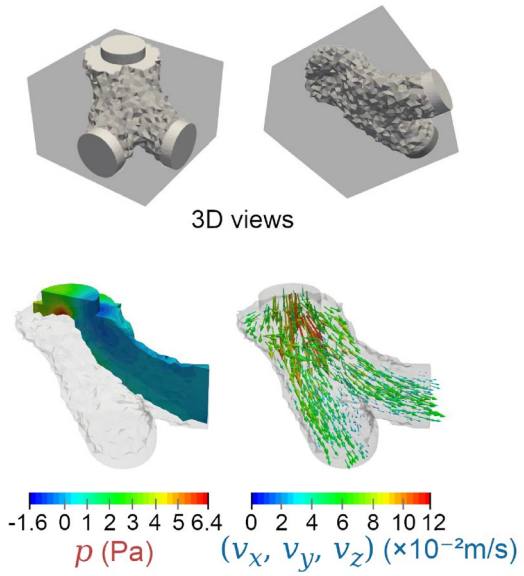
The part of the implementation that is performed in OpenFOAM® is, as mentioned in Sect. 6, including the additional inverse permeability term in the "`simpleFoam`" solver from OpenFOAM® (referred as "`CustomSimpleFoam`" in this work) [see Eq. (2)], and also in the "`SpalartAllmaras`" turbulence model (referred as "`CustomSpalartAllmaras`" in this work) [see Eq. (5)]. Another necessary implementation is to create an additional type of wall distance computation, which loads the wall distance from a file (referred as "`Custom_externalImport`" in this work).

The "FEniCS TopOpt Foam" library used in the implementation of this work is to be made available in a git repository[2]. It also includes sample implementations of "`CustomSimpleFoam`", "`CustomSpalartAllmaras`", and "`Custom_externalImport`". An implementation of a code by using "FEniCS TopOpt Foam" for a sample 2D bend channel topology optimization (slightly different from Sect. 7.2 in order to be simpler and easier to understand) is shown step by step in the following subsections. In the following code excerpts, when a line of code is split due to lack of space, its continuation is shown in the next line, preceded by an arrow ("↪").

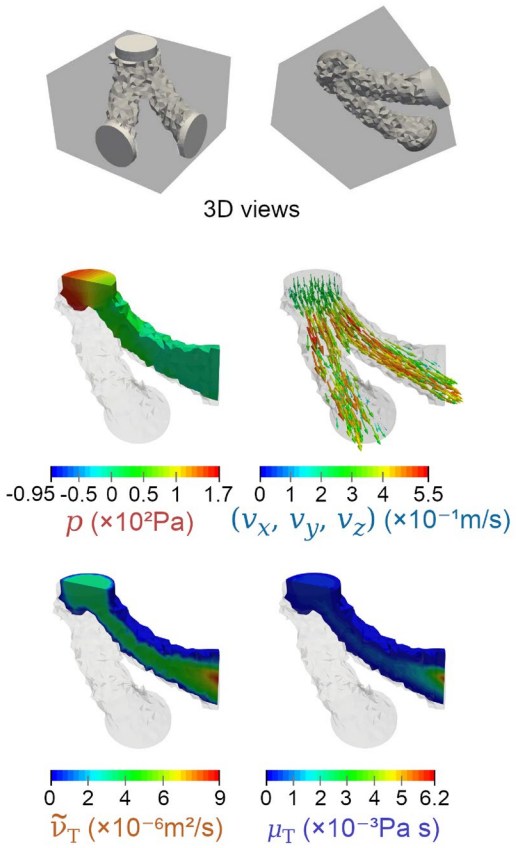
9.1 Sample 2D bend channel problem

In this section, the 2D bend channel problem is considered through a sample implementation, where ρ is set as 1.0, μ is set as 0.1, and the inlet velocity is defined as such that the maximum velocity of the inlet parabola is 1.0, while the computational domain is a 1×1 square. The implementation is performed by leaving a variable to set which flow regime

[2] https://github.com/diego-hayashi/fenics_topopt_foam.



(a) Optimization for laminar flow.



(b) Optimization for turbulent flow.

Fig. 27 Optimized topologies, pressure, and velocity for the 3D channel

Table 4 Variable naming in the equations of this article and the implementations in FEniCS and OpenFOAM®.

Equations of this article	FEniCS	OpenFOAM®
$\boldsymbol{v}$	`v`	`U`
p	`p`	`p`
$\tilde{\nu}_\mathrm{T}$	`nu_T_aux`	`nuTilda`
ℓ_w	`l_wall`	`yWall_to_load`
α	`alpha`	`alpha_design`

(laminar or turbulent) is being considered ("`flow_regime`" variable) and another variable is left to set whether to consider OpenFOAM® in parallel or not ("`run_openfoam_in_parallel`"). The optimization parameters are prepared for the laminar and turbulent cases, but their specific values are set for a laminar flow topology optimization, and may be adjusted by the user for a turbulent flow case. Table 4 presents the main variable naming differences between this article and the implementations in FEniCS and OpenFOAM®.

9.2 Necessary imports

The necessary imports should be included in the beginning of the code.

```
# Necessary imports
import os
import numpy as np
import matplotlib
matplotlib.use('Agg')
from fenics import *
import ufl
from dolfin_adjoint import *
import pyadjoint
from pyadjoint.tape import no_annotations
import mpi4py

# Some flags for FEniCS
parameters["form_compiler"]["optimize"] = True
parameters["form_compiler"]["cpp_optimize"] = True
parameters["form_compiler"]["cpp_optimize_flags"] =
 ↪ "-O3 -ffast-math -march=native"
parameters['allow_extrapolation'] = True # Allow
 ↪ small numerical differences in the boundary
 ↪ definition.

# Quadrature degree in FEniCS  (sometimes, the "
 ↪ automatic" determination of the quadrature
 ↪ degree becomes excessively high, meaning that it
 ↪  should be manually reduced)
parameters['form_compiler']['quadrature_degree'] = 5

# FEniCS TopOpt Foam imports
import fenics_topopt_foam
from fenics_topopt_foam.dolfin_adjoint_extensions
 ↪ import UncoupledNonlinearVariationalSolver,
 ↪ getWallDistanceAndNormalVectorFromDolfinAdjoint
```

9.3 General configurations

The general configurations can be set as follows: First, an additional variable ("`run_openfoam_in_parallel`")

is set in order to control whether OpenFOAM® should run in parallel or not.

```
# Run OpenFOAM in parallel?
run_openfoam_in_parallel = False
```

Then, the fluid properties are set alongside the corresponding inlet values and the flow regime.

```
# Fluid flow setup
rho_ = 1.0; mu_ = 0.1 # Density and dynamic viscosity
width_inlet_outlet = 1.0/5.0 # Inlet/outlet width
x_min = 0.0; x_max = 1.0 # x dimensions
y_min = 0.0; y_max = 1.0 # y dimensions
v_max_inlet = 1.0       # Inlet velocity
nu_T_aux_inlet = 0.0001 # Inlet turbulent variable value
flow_regime = 'laminar' # Flow regime: 'laminar' or 'turbulent (Spalart-Allmaras)'
```

9.4 Set topology optimization-related parameters

The topology optimization-related parameters are defined.

```
# Topology optimization setup
k_max = 1.E4*mu_; k_min = 0.0; q = 0.1
k = lambda alpha : k_max + (k_min - k_max) * alpha * (1. + q) / (alpha + q)
gamma_max = 1.E3; gamma_min = 0.0
gamma = lambda alpha : gamma_max + (gamma_min - gamma_max) * alpha * (1. + q) / (alpha + q)
lambda_kappa_v = 1.0
f_V = 0.3 # Volume fraction
```

9.5 Create the output folder

A folder for including the results is created.

```
# Output folders
output_folder = 'output'
problem_folder = "%s/foam_problem" %(output_folder)
if (MPI.comm_world.Get_size() == 1) or (MPI.comm_world.Get_rank() == 0):
 if not os.path.exists(output_folder):
  os.makedirs(output_folder) # Create the output folder if it still does not exist
```

9.6 Create the 2D mesh in FEniCS

The mesh is created in FEniCS and saved to file for visualization. It can be mentioned that any mesh or mesh generation scheme in FEniCS may be considered, such as from FEniCS itself, from an external mesh imported to FEniCS, and from "`mshr`" (additional meshing module from FEniCS).

```
# Create the 2D mesh and plot it
N_mesh = 50
delta_x = x_max - x_min; delta_y = y_max - y_min
mesh = RectangleMesh(Point(x_min, y_min), Point(x_max, y_max), int(N_mesh*delta_x/delta_y), N_mesh, diagonal = "crossed")
File('%s/mesh.pvd' %(output_folder)) << mesh
```

9.7 Define the function spaces for FEniCS

The FEniCS implementation requires the definition of the function spaces for the state and design variables.

```
# Function spaces -> MINI element (2D)
V1_element =  FiniteElement('Lagrange', mesh.ufl_cell(), 1)
B_element = FiniteElement('Bubble', mesh.ufl_cell(), 3)
V_element = VectorElement(NodalEnrichedElement(V1_element, B_element)) # Velocity
P_element = FiniteElement('Lagrange', mesh.ufl_cell(), 1) # Pressure
if flow_regime == 'turbulent (Spalart-Allmaras)':
 NU_T_AUX_element = FiniteElement('Lagrange', mesh.ufl_cell(), 1) # Turbulent variable
if flow_regime == 'laminar':
 U_element = MixedElement([V_element, P_element])
elif flow_regime == 'turbulent (Spalart-Allmaras)':
 U_element = MixedElement([V_element, P_element, NU_T_AUX_element])
U = FunctionSpace(mesh, U_element) # Mixed function space
A_element = FiniteElement('Lagrange', mesh.ufl_cell(), 1)
A = FunctionSpace(mesh, A_element) # Design variable function space (nodal)
```

9.8 Prepare the boundary definition in FEniCS

The boundaries of the computational domain are given names in FEniCS, which will also be used in OpenFOAM®, and saved to file, for visualization.

```
# Prepare the boundary definition
class Inlet(SubDomain):
 def inside(self, x, on_boundary):
  return on_boundary and x[0] == x_min and ((y_min +
   ↪  4.0/5*delta_y - width_inlet_outlet/2) < x[1]
   ↪ < (y_min + 4.0/5*delta_y + width_inlet_outlet
   ↪ /2))
class Outlet(SubDomain):
 def inside(self, x, on_boundary):
  return on_boundary and x[1] == y_min and ((x_min +
   ↪  4.0/5*delta_x - width_inlet_outlet/2) < x[0]
   ↪ < (x_min + 4.0/5*delta_x + width_inlet_outlet
   ↪ /2))
class Walls(SubDomain):
 def inside(self, x, on_boundary):
  return on_boundary # * It will be set before the
   ↪ other boundaries
marker_numbers = {'unset' : 0, 'wall' : 1, 'inlet' :
 ↪  2, 'outlet' : 3}
boundary_markers = MeshFunction('size_t', mesh, mesh
 ↪ .topology().dim() - 1)
boundary_markers.set_all(marker_numbers['unset'])
Walls().mark(boundary_markers, marker_numbers['wall'
 ↪ ])
Inlet().mark(boundary_markers, marker_numbers['inlet
 ↪ '])
Outlet().mark(boundary_markers, marker_numbers['
 ↪ outlet'])
File("%s/markers.pvd" %(output_folder)) <<
 ↪ boundary_markers
```

9.9 Prepare boundary values (for Dirichlet Boundary conditions) in FEniCS

Some of the boundary values that will be used for Dirichlet Boundary conditions are defined.

```
# Boundary values (for Dirichlet Boundary conditions
 ↪ )
class InletVelocity(UserExpression):
 def eval(self, values, x):
  for i in range(len(values)):
   values[i] = 0.0 # Initialize all values with
    ↪ zeros
  if x[0] == x_min and (4.0/5*delta_y -
   ↪ width_inlet_outlet/2) < x[1] < (4.0/5*delta_y
   ↪ + width_inlet_outlet/2):
   y_local = x[1] - 4.0/5*delta_y; values[0] =
    ↪ v_max_inlet*(1 - (2*y_local/
    ↪ width_inlet_outlet)**2)
 def value_shape(self):
  return (2,)
inlet_velocity_expression = InletVelocity(element =
 ↪ V_element)
wall_velocity_value = Constant((0,0))
if flow_regime == 'turbulent (Spalart-Allmaras)':
 inlet_nu_T_aux_value = Constant(nu_T_aux_inlet)
```

9.10 Function to set "FEniCS TopOpt Foam"

The function "`prepareFEniCSFoamSolverWithUpdate`" is created in order to prepare the whole setup for the OpenFOAM® simulation from "FEniCS TopOpt Foam". First, the boundary data are gathered in a format that is more closely related to OpenFOAM® definitions.

```
# Function to set FEniCS TopOpt Foam
@no_annotations
def prepareFEniCSFoamSolverWithUpdate(u, alpha, mesh
 ↪ , boundary_markers, marker_numbers, bcs):

 # Gather the boundary data
 boundary_data = {
  'mesh_function' : boundary_markers,
  'mesh_function_tag_to_boundary_name' : {value :
   ↪ key for key, value in marker_numbers.items()},
   ↪  # Invert dictionary
  'boundaries' : {
   'wall'    : {'type' : 'wall', 'inGroups' : ['wall'
    ↪ ]},
   'inlet'   : {'type' : 'patch',},
   'outlet' : {'type' : 'patch',},
  },
 }
```

Then, the basic parameters necessary for defining a solver in "FEniCS TopOpt Foam" are defined.

```
 # Parameters for fenics_topopt_foam.FoamSolver
 foam_parameters = {
  'domain type' : '2D',  # Domain type according to
   ↪ what the model implemented in FEniCS
  'error_on_nonconvergence' : False,
  'problem_folder' : problem_folder,
  'solver' : {
   'type' : 'custom',
   'openfoam' : {'name' : 'simpleFoam'},
   'custom' : {
    'name' : 'CustomSimpleFoam', # simpleFoam with
     ↪ material model
    'location' : '%s/cpp_openfoam_modules/
     ↪ foam_cpp_solvers' %(os.path.dirname(
     ↪ fenics_topopt_foam.__file__)),
   },
  },
  'compile_modules_if_needed' : True,
 }
```

Following, it is necessary to prepare the OpenFOAM® dictionary entries for "`controlDict`", "`fvSchemes`", and "`fvSolution`". These three dictionaries are required by OpenFOAM® for any simulation and are essential for controlling how these simulations will be executed, which means that they should be completely defined by the user. It should be reminded, though, that "`writeFormat`" (from "`controlDict`") needs to be set to "`ascii`" for "FEniCS TopOpt Foam".

```
# Configurations for OpenFOAM
foam_configurations_dictionary = {
 'controlDict' : {
  'application' :         foam_parameters['solver'
   ↪ ][foam_parameters['solver']['type']]['name'],
  'startFrom' :           'startTime',
  'startTime' :           0,
  'stopAt' :              'endTime',
  'endTime' :             2000,
  'deltaT' :              1,
  'writeControl' :        'timeStep',
  'writeInterval' :       100,
  'purgeWrite' :           2,
  'writeFormat' :         'ascii',
  'writePrecision' :      12,
  'writeCompression' :    'off',
  'timeFormat' :          'general',
  'timePrecision' :       6,
  'graphFormat' :         'raw',
  'runTimeModifiable' :   'true',
 },
 'fvSchemes' : {
  'ddtSchemes' : {
   'default' : 'steadyState',
  },
  'gradSchemes' : {
   'default' : ('Gauss', 'linear'),
   'grad(nuTilda)' : ('cellLimited', 'Gauss', '
    ↪ linear', '1.0'),
  },
  'divSchemes' : {
   'default' : 'none',
   'div(phi,U)' : ('bounded', 'Gauss', '
    ↪ linearUpwind', 'grad(U)'),
   'div(phi,nuTilda)' : ('bounded', 'Gauss', '
    ↪ limitedLinear', 1),
   'div((nuEff*dev2(T(grad(U))))' : ('Gauss', '
    ↪ linear'),
   'div(nonlinearStress)' : ('Gauss', 'linear'),
  },
  'laplacianSchemes' : {
   'default' : ('Gauss', 'linear', 'corrected'),
  },
  'interpolationSchemes' : {
   'default' : 'linear',
  },
  'snGradSchemes' : {
   'default' : 'corrected',
  },
  'wallDist' : {
   'method' : 'Custom_externalImport',
  },
 },
 'fvSolution' : {
  'solvers' : {
   'p' : {
    'solver' :          'GAMG',
    'tolerance' :       1.E-06,
    'relTol':           0.1,
    'maxIter' :         1000,
    'preconditioner' : 'none',
    'smoother' :       'GaussSeidel',
   },
   'U' : {

    'type' :            'segregated',
    'solver' :          'smoothSolver',
    'tolerance' :       1e-05,
    'relTol' :          0.1,
    'maxIter' :         1000,
    'preconditioner' : 'none',
    'smoother' :       'symGaussSeidel',
    'nSweeps' :         2,
   },
   'nuTilda' : {
    'solver' :       'smoothSolver',
    'tolerance' :       1e-05,
    'relTol' :      0.1,
    'maxIter' :         1000,
    'preconditioner' : 'none',
    'smoother' :       'symGaussSeidel',
    'nSweeps' :         2,
   },
  },
  'SIMPLE' : {
   'nNonOrthogonalCorrectors' : 3,
   'consistent' : 'yes',
   'residualControl' : {
    'p' :       1e-2,
    'U' :       1e-3,
    'nuTilda' : 1e-3,
   },

  },

  'relaxationFactors' : {
   'fields' : {
    'p' :        0.9,
    'U' :        0.9,
    'nuTilda' : 0.9,
    '".*"' :     0.9,
   },
   'equations' : {
    'p' :        0.9,
    'U' :        0.9,
    'nuTilda' : 0.9,
    '".*"' :     0.9,
   },
  },
 },
}
```

The "libs" entry from "controlDict" is set to consider some C++ OpenFOAM® libraries provided by "FEniCS TopOpt Foam" (i.e., the OpenFOAM® libraries that are already mentioned in the beginning of Sect. 9), but the user may include any user-made library in this entry.

```
# Set to use the libs provided in FEniCS TopOpt
 ↪ Foam in OpenFOAM
libs_folder = '%s/cpp_openfoam_modules/
 ↪ foam_cpp_libs' %(os.path.dirname(
 ↪ fenics_topopt_foam.__file__))
lib_names = fenics_topopt_foam.
 ↪ compileLibraryFoldersIfNeeded(libs_folder)
lib_names_string = '('
for i in range(len(lib_names)):
 if i != 0: lib_names_string += ' '
 lib_names_string += "\"%s\"" %(lib_names[i])
lib_names_string += ')'
foam_configurations_dictionary['controlDict']['libs
 ↪ '] = lib_names_string
```

9.11 Solver that interacts with FEniCS and OpenFOAM®

Now, the solver can be created (called "FEniCSFoamSolverWithUpdate") with the previously defined parameters, variables, mesh, boundary conditions, and the fluid properties.

```
# Solver that interacts with FEniCS/dolfin-adjoint
 ↪ and OpenFOAM
class FEniCSFoamSolverWithUpdate():
 def __init__(self, u, alpha):

  # Setups and initializations
  self.u = u; self.u.vector().apply('insert'); self
   ↪ .u_array_copy = self.u.vector().get_local()
  self.alpha = alpha
  self.fenics_foam_solver = fenics_topopt_foam.
   ↪ FEniCSFoamSolver(
   mesh, boundary_data,
   foam_parameters, self.getPropertiesDictionary(),
   foam_configurations_dictionary,
   use_mesh_from_foam_solver = False,
   python_write_precision =
    ↪ foam_configurations_dictionary['controlDict'
    ↪ ]['writePrecision'],
   configuration_of_openfoam_measurement_units = {'
    ↪ pressure' : 'rho-normalized pressure'},
   )
  if flow_regime == 'turbulent (Spalart-Allmaras)':
   self.fenics_foam_solver.initFoamVector('nut', '
    ↪ volScalarField', skip_if_exists = True)
   self.fenics_foam_solver.initFoamVector('nuTilda'
    ↪ , 'volScalarField', skip_if_exists = True)
   self.fenics_foam_solver.initFoamVector('
    ↪ yWall_to_load', 'volScalarField',
    ↪ skip_if_exists = True)
   self.fenics_foam_solver.initFoamVector('
    ↪ nWall_to_load', 'volVectorField',
    ↪ skip_if_exists = True)
   self.flag_set_boundary_values = True
  self.setBoundaryConditionsForOpenFOAM()
```

The parallelism in OpenFOAM® is set here (if "run_openfoam_in_parallel = True"), where the "parallel_data" dictionary needs to be set according to OpenFOAM® conventions, and the value set for the "numberOfSubdomains" entry also corresponds to the number of processes for OpenFOAM® parallelism.

```
    # Set to run OpenFOAM in parallel
    if run_openfoam_in_parallel == True:
     parallel_data = {
      'numberOfSubdomains' : 2,
      'method' : 'simple',
      'simpleCoeffs' : {
       'n' : '(2 1 1)',
       'delta' : 0.001,
      },
     }
     self.fenics_foam_solver.foam_solver.
      ↪ setToRunInParallel(parallel_data)
```

The boundary conditions are set as follows:

```
  def setBoundaryConditionsForOpenFOAM(self):
   self.fenics_foam_solver.setAdditionalProperty('
    ↪ rho', rho_) # rho is necessary to convert '
    ↪ rho-normalized pressure' (used in OpenFOAM)
    ↪ to 'pressure' (used here)
   self.fenics_foam_solver.setFoamBoundaryCondition(
    ↪ 'U', 'outlet', 'zeroGradient', None)
   self.fenics_foam_solver.setFoamBoundaryCondition(
    ↪ 'U', 'inlet', 'fixedValue',
    ↪ inlet_velocity_expression)
   self.fenics_foam_solver.setFoamBoundaryCondition(
    ↪ 'U', 'wall', 'noSlip', None)
   self.fenics_foam_solver.setFoamBoundaryCondition(
    ↪ 'p', 'outlet', 'fixedValue', np.array([0.0],
    ↪ dtype = 'float'))
   self.fenics_foam_solver.setFoamBoundaryCondition(
    ↪ 'p', 'inlet', 'zeroGradient', None)
   self.fenics_foam_solver.setFoamBoundaryCondition(
    ↪ 'p', 'wall', 'zeroGradient', None)
   if flow_regime == 'turbulent (Spalart-Allmaras)':
    self.fenics_foam_solver.setFoamBoundaryCondition
     ↪ ('nuTilda', 'inlet', 'fixedValue',
     ↪ inlet_nu_T_aux_value)
    self.fenics_foam_solver.setFoamBoundaryCondition
     ↪ ('nuTilda', 'wall', 'fixedValue', np.array
     ↪ ([0]))
    self.fenics_foam_solver.setFoamBoundaryCondition
     ↪ ('nuTilda', 'outlet', 'zeroGradient', None)
    self.fenics_foam_solver.setFoamBoundaryCondition
     ↪ ('nut', 'inlet', 'calculated', np.array([0])
     ↪ )
    self.fenics_foam_solver.setFoamBoundaryCondition
     ↪ ('nut', 'outlet', 'calculated', np.array
     ↪ ([0]))
    self.fenics_foam_solver.setFoamBoundaryCondition
     ↪ ('nut', 'wall', 'calculated', np.array([0]))
```

The fluid flow properties are set as follows:

```
  def getPropertiesDictionary(self):
   if flow_regime == 'laminar':
    simulationType = 'laminar'
    turbulence_switch = 'off'
   elif flow_regime == 'turbulent (Spalart-Allmaras
    ↪ ':
    simulationType = 'RAS'
    turbulence_switch = 'on'
   foam_properties_dictionary = {
    'materialmodelProperties' : {
     'q_penalization' : (fenics_topopt_foam.
      ↪ convertToFoamUnitSpecification('
      ↪ dimensionless'), q),
     'k_max' : (fenics_topopt_foam.
      ↪ convertToFoamUnitSpecification('inverse
      ↪ permeability'), k_max),
     'k_min' : (fenics_topopt_foam.
      ↪ convertToFoamUnitSpecification('inverse
      ↪ permeability'), k_min),
     'rho_density' : (fenics_topopt_foam.
      ↪ convertToFoamUnitSpecification('density'),
      ↪ rho_), # * This is because we will need to
      ↪ divide k(alpha) by the density inside
      ↪ CustomSimpleFoam!
    },
    'turbulenceProperties' : {
     'simulationType' : simulationType,
     'RAS' : {
      'turbulence' : turbulence_switch,
      'RASModel' : 'CustomSpalartAllmaras',
      'CustomSpalartAllmaras' : {
       'lambda_v_design' : lambda_kappa_v,
      },
      'printCoeffs' : 'on',
     },
    },
    'transportProperties' : {
     'transportModel' : 'Newtonian',
     'nu' : (fenics_topopt_foam.
      ↪ convertToFoamUnitSpecification('kinematic
      ↪ viscosity'), mu_/rho_),
    },
   }
   return foam_properties_dictionary
```

The “plotResults” function from “FEniCSFoamSolver” can be left more readily accessible.

```
  @no_annotations
  def plotResults(self, *args, **kwargs):
   self.fenics_foam_solver.plotResults(*args, **
    ↪ kwargs)
```

The main function for solving the simulation can then be defined as follows. First, the variables are retrieved from dolfin-adjoint (“replace_map”), and an initial guess for the state vector (called “u”) is set.

```
@no_annotations
def solve(self, replace_map = {}):

 # Get variables from replace_map and load the initial guess for the simulation
 if type(replace_map).__name__ == 'NoneType' or len(replace_map) == 0:
  u = self.u; alpha = self.alpha
 else:
  u = replace_map[self.u]; alpha = replace_map[self.alpha]
 u.vector().set_local(self.u_array_copy); u.vector().apply('insert')
```

Then, the wall distance is computed in FEniCS and set to OpenFOAM®.

```
 # Wall distance in FEniCS
 if flow_regime == 'turbulent (Spalart-Allmaras)':

  global l_wall
  L_WALL = FunctionSpace(mesh, 'Lagrange', 1)
  N_WALL = VectorFunctionSpace(mesh, 'Lagrange', 1)
  (l_wall_projected, normal_a_paredes) = getWallDistanceAndNormalVectorFromDolfinAdjoint(l_wall, L_WALL, N_WALL, domain_type = '2D', replace_map = replace_map)
  self.fenics_foam_solver.setFEniCSFunctionToFoamVector(l_wall_projected, foam_variable_name = 'yWall_to_load')
  self.fenics_foam_solver.setFEniCSFunctionToFoamVector(normal_a_paredes, foam_variable_name = 'nWall_to_load')

  if self.flag_set_boundary_values == True:
   self.fenics_foam_solver.setFoamBoundaryCondition('yWall_to_load', 'wall', 'fixedValue', l_wall_projected)
   self.fenics_foam_solver.setFoamBoundaryCondition('nWall_to_load', 'wall', 'fixedValue', normal_a_paredes)
   self.fenics_foam_solver.setFoamBoundaryCondition('yWall_to_load', 'inlet', 'zeroGradient', None)
   self.fenics_foam_solver.setFoamBoundaryCondition('nWall_to_load', 'inlet', 'zeroGradient', None)
   self.fenics_foam_solver.setFoamBoundaryCondition('yWall_to_load', 'outlet', 'zeroGradient', None)
   self.fenics_foam_solver.setFoamBoundaryCondition('nWall_to_load', 'outlet', 'zeroGradient', None)
   self.flag_set_boundary_values = False
```

Following, the properties are optionally updated (if a continuation scheme in the property values is desired during topology optimization).

```
 # Update properties
 foam_properties_dictionary = self.getPropertiesDictionary()
 if type(foam_properties_dictionary).__name__ != 'NoneType':
  for key in foam_properties_dictionary:
   self.fenics_foam_solver.setFoamProperty(key, foam_properties_dictionary[key])
```

The variables are set to OpenFOAM®.

```
 # Set all variables to fenics_foam_solver
 u_split_deepcopy = u.split(deepcopy = True)
 self.fenics_foam_solver.setFEniCSFunctionToFoamVector(u_split_deepcopy[0], foam_variable_name = 'U')
 self.fenics_foam_solver.setFEniCSFunctionToFoamVector(u_split_deepcopy[1], foam_variable_name = 'p')
 if flow_regime == 'turbulent (Spalart-Allmaras)':
  self.fenics_foam_solver.setFEniCSFunctionToFoamVector(u_split_deepcopy[2], foam_variable_name = 'nuTilda')
 self.fenics_foam_solver.setFEniCSFunctionToFoamVector(alpha, foam_variable_name = 'alpha_design', set_calculated_foam_boundaries = True, ensure_maximum_minimum_values_after_projection = True)
```

The OpenFOAM® simulation can now be performed. In this case, in order to help monitoring the residuals from the simulation, the parameter "`continuously_plot_residuals_from_log`" is set to "`True`". This means that, inside the OpenFOAM® simulation folder (called "`foam_problem`" in Sect. 9.3), there will be a "`logs`" folder which will contain the plots made with Matplotlib (image files, ".png") for each residual. These plots are renewed at each optimization iteration. In order for Matplotlib to be able to plot, it is essential that "`matplotlib.use('Agg')`" is used in the beginning of the code, as shown in Sect. 9.3, because Matplotlib is set to create the plots simultaneously to the simulation in OpenFOAM® by spawning a child process, because it requires Matplotlib to be using a non-interactive backend (such as "`Agg`"), which is able to directly generate image files, but disables the capacity of Matplotlib opening GUI windows.

```
 # Solve the problem with OpenFOAM and plot residuals
 self.fenics_foam_solver.solve(
  silent_run_mode = False,
  num_logfile_lines_to_print_in_silent_mode = 0,
  continuously_plot_residuals_from_log = True,
  continuously_plot_residuals_from_log_time_interval = 5,
  continuously_plot_residuals_from_log_x_axis_label = 'Iteration',
  continuously_plot_residuals_from_log_y_axis_scale = 'log/symlog',
  )
```

After the simulation, the computed variables are set back to FEniCS/dolfin-adjoint.

```
    # Set the state variables from the fenics_foam_solver
    self.fenics_foam_solver.setFoamVectorToFEniCSFunction(u_split_deepcopy[0], foam_variable_name = 'U')
    self.fenics_foam_solver.setFoamVectorToFEniCSFunction(u_split_deepcopy[1], foam_variable_name = 'p')
    if flow_regime == 'turbulent (Spalart-Allmaras)':
     self.fenics_foam_solver.setFoamVectorToFEniCSFunction(u_split_deepcopy[2], foam_variable_name = 'nuTilda')
    fenics_topopt_foam.assignSubFunctionsToFunction(to_u_mixed = u, from_u_separated_array = list(u_split_deepcopy))
    [bc.apply(u.vector()) for bc in bcs]
    self.u_array_copy = u.vector().get_local()
    return u
```

With "FEniCSFoamSolverWithUpdate" defined, it is now created.

```
# Create fenics_foam_solver_with_update
fenics_foam_solver_with_update = FEniCSFoamSolverWithUpdate(u, alpha)
return fenics_foam_solver_with_update
```

9.12 Forward model in FEniCS

A function that prepares the forward model in FEniCS from a design variable distribution ("alpha") has to be defined, because it will be used by dolfin-adjoint for the automatic derivation of the adjoint model. First, the state vector and test functions are defined, alongside some auxiliary definitions.

```
# Function to solve the forward problem
global fenics_foam_solver_with_update, l_wall, mu_T
fenics_foam_solver_with_update = None; l_wall = None; mu_T = 0
def solve_forward_problem(alpha):
 global fenics_foam_solver_with_update, l_wall, mu_T

 # Set the state vector and test functions
 u = Function(U); u.rename("StateVariable", "StateVariable")
 u_split = split(u); v = u_split[0]; p = u_split[1]
 if flow_regime == 'turbulent (Spalart-Allmaras)': nu_T_aux = u_split[2]
 w = TestFunction(U)
 w_split = split(w); w_v = w_split[0]; w_p = w_split[1]
 if flow_regime == 'turbulent (Spalart-Allmaras)': w_nu_T_aux = w_split[2]

 # Additional definitions
 n = FacetNormal(mesh)
 nu_ = mu_/rho_
```

In the case of using a turbulence model (Spalart-Allmaras model), the computation of the wall distance is performed.

```
 # Wall distance computation (modified Eikonal equation)
 if flow_regime == 'turbulent (Spalart-Allmaras)':
  G_space = FunctionSpace(mesh, 'Lagrange', 1)
  G = Function(G_space); w_G = TestFunction(G_space)
  mesh_hmax = MPI.comm_world.allreduce(mesh.hmax(), op = mpi4py.MPI.MAX)
  G_initial = interpolate(Constant(1./mesh_hmax), G_space); G.assign(G_initial)

  sigma_wall = 0.1
  G_ref = 1./mesh_hmax
  F_G = inner(grad(G), grad(G))*w_G*dx - inner(grad(G), grad(sigma_wall*G*w_G))*dx - (1. + 2*sigma_wall)*G**4 * w_G*dx - gamma(alpha)*(G - G_ref) * w_G*dx
  bcs_G = [DirichletBC(G_space, G_ref, boundary_markers, marker_numbers['wall'])]

  dF_G = derivative(F_G, G); problem_G = NonlinearVariationalProblem(F_G, G, bcs_G, dF_G)
  solver_G = NonlinearVariationalSolver(problem_G)
  solver_G.solve(annotate = True)
  l_wall = 1./G - 1./G_ref
```

Then, the remaining weak forms and boundary conditions for FEniCS are defined and combined.

```
# Weak form of the turbulent equations (Spalart-Allmaras model)
if flow_regime == 'turbulent (Spalart-Allmaras)':
 k_von_Karman = 0.41; c_v1 = 7.1; c_b1 = 0.1355; c_b2 = 0.6220; c_w2 = 0.3; c_w3 = 2.0; sigma = 2./3
 adjustment = DOLFIN_EPS_LARGE # Adjustment for numerical precision (1.E-14)
 Chi = nu_T_aux/nu_; f_v1 = (Chi**3)/(Chi**3 + c_v1**3)
 nu_T = f_v1*nu_T_aux; mu_T = rho_*nu_T
 Omega = 1/2. * (grad(v) - grad(v).T); Omega_m = sqrt(2.*inner(Omega, Omega)); S = Omega_m
 f_v2 = 1. - Chi/(1 + Chi*f_v1); S_tilde = ufl.Max(S + nu_T_aux/(k_von_Karman**2*(l_wall**2 + adjustment))*f_v2, 0.3*Omega_m)
 S_tilde_para_r = ufl.Max(S_tilde, 1.E-6)
 r_i = ufl.Min(nu_T_aux/(S_tilde_para_r*k_von_Karman**2*(l_wall**2 + adjustment)), 10.0)
 g_i = r_i + c_w2*(r_i**6 - r_i)
 f_w = g_i*((1. + c_w3**6)/(g_i**6 + c_w3**6))**(1./6)
 c_w1 = c_b1/k_von_Karman**2 + (1 + c_b2)/sigma
 F_SA = inner(v, rho_*grad(nu_T_aux))*w_nu_T_aux*dx - (c_b1*rho_*S_tilde*nu_T_aux)*w_nu_T_aux*dx + inner( rho_*(nu_ + nu_T_aux)*grad(nu_T_aux), grad(w_nu_T_aux)/sigma)*dx - c_b2/sigma*rho_*inner(grad(nu_T_aux), grad(nu_T_aux))*w_nu_T_aux*dx - ( - c_w1*f_w*rho_*(nu_T_aux**2)/(l_wall**2 + adjustment) )*w_nu_T_aux*dx + lambda_kappa_v*k(alpha)*(nu_T_aux - 0.0)*w_nu_T_aux*dx
else:
 mu_T = 0

# Weak form of the pressure-velocity formulation
I_ = as_tensor(np.eye(2)); T = -p*I_ + (mu_ + mu_T)*(grad(v) + grad(v).T); v_mat = v
F_PV = div(v) * w_p*dx + inner( grad(w_v), T )*dx + rho_ * inner( dot(grad(v), v), w_v )*dx + inner(k(alpha) * v_mat, w_v)*dx

# Full weak form
if flow_regime == 'laminar':
                                F = F_PV
elif flow_regime == 'turbulent (Spalart-Allmaras)':
 F = F_PV + F_SA

# Boundary conditions
bcs = [DirichletBC(U.sub(0), wall_velocity_value, boundary_markers, marker_numbers['wall']), DirichletBC(U.sub(0), inlet_velocity_expression, boundary_markers, marker_numbers['inlet'])]
if flow_regime == 'turbulent (Spalart-Allmaras)':
 bcs += [DirichletBC(U.sub(2), Constant(0.0), boundary_markers, marker_numbers['wall']), DirichletBC(U.sub(2), inlet_nu_T_aux_value, boundary_markers, marker_numbers['inlet'])]
```

Then, the "`prepareFEniCSFoamSolverWithUpdate`" is created and used as an input parameter for "`UncoupledNonlinearVariationalSolver`", which will perform the coupling between the optimization and the simulation.

```
# Prepare the FEniCSFoamSolver a single time
if type(fenics_foam_solver_with_update).__name__ == 'NoneType':
 fenics_foam_solver_with_update = prepareFEniCSFoamSolverWithUpdate(u, alpha, mesh, boundary_markers, marker_numbers, bcs)
else:
 fenics_foam_solver_with_update.u = u; fenics_foam_solver_with_update.alpha = alpha

# Solve the simulation
dF = derivative(F, u); problem = NonlinearVariationalProblem(F, u, bcs, dF)
nonlinear_solver = UncoupledNonlinearVariationalSolver(problem, simulation_solver = fenics_foam_solver_with_update)
u = nonlinear_solver.solve()
return u
```

9.13 Preparations for topology optimization

The initial setup for topology optimization is performed.

```
# Initial setup for topology optimization
alpha = interpolate(Constant(f_V), A) # Initial guess for the design variable
set_working_tape(Tape())              # Clear all annotations and restart the adjoint model
```

An initial simulation is performed for dolfin-adjoint to prepare the automatic derivation of the adjoint model.

```
# Solve the simulation
u = solve_forward_problem(alpha)
```

Some visualization files are prepared for visualizing the optimized topology during the topology optimization iterations.

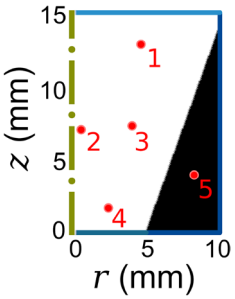


Fig. 28 Topology considered for the finite differences comparison

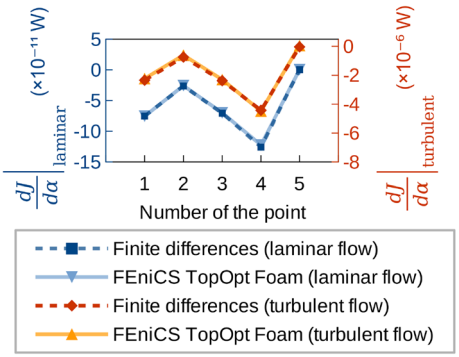


Fig. 29 Sensitivity values computed with the "FEniCS TopOpt Foam" approach (from dolfin-adjoint) and from finite differences, for laminar and turbulent flows

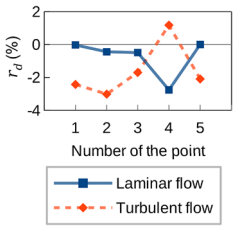


Fig. 30 Relative differences for the cases shown in Fig. 29

```
# Visualization files
alpha_pvd_file = File("%s/alpha_iterations.pvd" %(output_folder)); alpha_viz = Function(A, name = "AlphaVisualisation")
dj_pvd_file = File("%s/dj_iterations.pvd" %(output_folder)); dj_viz = Function(A, name = "dJVisualisation")
```

In order to continuously save the visualization files, it is necessary to create a callback for dolfin-adjoint, such as immediately after the computation of the sensitivities ("`derivative_cb_post`").

```
# Callback during topology optimization
global current_iteration
current_iteration = 0
def derivative_cb_post(j, dj, current_alpha):
 global current_iteration
 print("\n [Iteration: %d] J = %1.7e\n" %(current_iteration, j)); current_iteration += 1
 # Save for visualization
 alpha_viz.assign(current_alpha); alpha_pvd_file << alpha_viz
 dj_viz.assign(dj); dj_pvd_file << dj_viz
```

9.14 Topology optimization

The topology optimization problem can now be defined, as well as the IPOPT solver can be instantiated from dolfin-adjoint.

```
# Objective function
u_split = split(u); v = u_split[0]
J = assemble((1/2.*(mu_ + mu_T)*inner(grad(v) + grad(v).T, grad(v) + grad(v).T) + inner(k(alpha) * v, v))*dx)
print(" Current objective function value: %1.7e" %(J))

# Set the topology optimization problem and solver
alpha_C = Control(alpha)
Jhat = ReducedFunctional(J, alpha_C, derivative_cb_post = derivative_cb_post)
problem_min = MinimizationProblem(Jhat, bounds = (0.0, 1.0), constraints = [UFLInequalityConstraint((f_V - alpha)*dx, alpha_C)])
solver_opt = IPOPTSolver(problem_min, parameters = {'maximum_iterations': 100})
```

To finalize, the topology optimization is performed.

```
# Perform topology optimization
alpha_opt = solver_opt.solve()
alpha.assign(alpha_opt); alpha_viz.assign(alpha)
alpha_pvd_file << alpha_viz
```

9.15 Plot the simulation

The simulation from OpenFOAM® may be plotted as follows.

```
# Plot a simulation
u = solve_forward_problem(alpha)
fenics_foam_solver_with_update.plotResults(file_type = 'VTK', tag_folder_name = '_final')
u_split_deepcopy = u.split(deepcopy = True)
v_plot = u_split_deepcopy[0]
p_plot = u_split_deepcopy[1]
File("%s/simulation_final_v.pvd" %(output_folder)) << v_plot
File("%s/simulation_final_p.pvd" %(output_folder)) << p_plot
if flow_regime == 'turbulent (Spalart-Allmaras)':
 nu_T_aux_plot = u_split_deepcopy[2]
 File("%s/simulation_final_nu_T_aux.pvd" %(output_folder)) << nu_T_aux_plot
```

9.16 Running the code

The resulting code may be run as: (1) totally in serial mode, (2) with only OpenFOAM® in parallel, (3) with only FEniCS in parallel, or (4) with FEniCS and OpenFOAM® in parallel. Parallelism in OpenFOAM® is enabled by setting "`run_openfoam_in_parallel = True`" (Sect. 9.3) and adequately setting (depending on your computational resources) the "`parallel_data`" dictionary (Sect. 9.11). Parallelism in FEniCS is set directly in the Python call, such as "`mpiexec -n 2 python my_code.py`" (for 2 processes). The number of processes for each type of parallelism is set as desired by the user and in a value allowed by the user's computational resources (such as 2,3,4 etc.). For no parallelism in OpenFOAM® and FEniCS ("serial mode"), set "`run_openfoam_in_parallel = False`" and run

the code as "`python my_code.py`". The plots, which contain the extensions .pvd and .vtk, may be visualized with the ParaView software.

Appendix A: Comparison of sensitivities with finite differences

In this appendix, a comparison of the computed sensitivities from dolfin-adjoint with finite differences is presented. The comparison is performed for the initial guess of the 2D axisymmetric nozzle (Sect. 7.3). A set of points is selected in the computational domain for comparison with finite differences (see Fig. 28): one near the inlet, one near the symmetry axis, one near the middle of the computational domain, one near the outlet, and a last one inside the solid material. The comparison is performed for the same configurations considered for laminar and turbulent flows in Sect. 7.3, by restricting the simulation to 6000 SIMPLE iterations. The finite differences are considered through the backward difference approximation (for $\alpha = 1$): $\frac{\mathrm{d}J}{\mathrm{d}\alpha} = \frac{J(\alpha)-J(\alpha-\Delta\alpha)}{\Delta\alpha}$, where $J = \Phi$. The finite difference approximation is changed to forward difference approximation for point number 5 ($\alpha = 0$): $\frac{\mathrm{d}J}{\mathrm{d}\alpha} = \frac{J(\alpha+\Delta\alpha)-J(\alpha)}{\Delta\alpha}$, where $J = \Phi$. A better approximation would be the use of a central finite difference approximation ($\frac{\mathrm{d}J}{\mathrm{d}\alpha} = \frac{J(\alpha+\Delta\alpha)-J(\alpha-\Delta\alpha)}{2\Delta\alpha}$), which is, however, inadequate for $\alpha = 0$ and $\alpha = 1$ (bounds of α). The computed sensitivities are shown in Fig. 29, for a step size of 10^{-3}. As can be seen, the computed sensitivities for the "FEniCS TopOpt Foam" approach (from dolfin-adjoint) and finite differences are close to each other. In order to get a better insight about the differences between the two sensitivities, Fig. 30 shows the relative differences as defined below, which resulted small.

$$r_d\big|_{\text{laminar}} = \left. \frac{\frac{dJ}{d\alpha}\Big|_{\text{FD}} - \frac{dJ}{d\alpha}\Big|_{\text{FTF}}}{\max\left|\frac{dJ}{d\alpha}\right|_{\text{FTF, all points}}} \right|_{\text{laminar}} \tag{32}$$

$$r_d\big|_{\text{turbulent}} = \left. \frac{\frac{dJ}{d\alpha}\Big|_{\text{FD}} - \frac{dJ}{d\alpha}\Big|_{\text{FTF}}}{\max\left|\frac{dJ}{d\alpha}\right|_{\text{FTF, all points}}} \right|_{\text{turbulent}} \tag{33}$$

where the subscript "FTF" indicates the "FEniCS TopOpt Foam" approach (from dolfin-adjoint) and "FD" indicates "Finite Differences".

Funding This research was partly supported by CNPq (Brazilian Research Council) and FAPESP (São Paulo Research Foundation). The authors thank the supporting institutions. The first author thanks the financial support of FAPESP under Grant 2017/27049-0. The second author thanks the financial support of CAPES. The third author thanks the financial support of CNPq (National Council for Research and Development) under grant 302658/2018-1 and of FAPESP under Grant 2013/24434-0. The authors also acknowledge the support of the RCGI (Research Centre for Gas Innovation), hosted by the University of São Paulo (USP) and sponsored by FAPESP (2014/50279-4) and Shell Brazil.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

Adept (2021) Adept—a combined automatic differentiation and array library for c++. http://www.met.reading.ac.uk/clouds/adept/. Accessdd May 2021

Alonso DH, de Sá LFN, Saenz JSR, Silva ECN (2018) Topology optimization applied to the design of 2d swirl flow devices. Struct Multidisc Optim 58(6):2341–2364. https://doi.org/10.1007/s00158-018-2078-0

Alonso DH, de Sá LFN, Saenz JSR, Silva ECN (2019) Topology optimization based on a two-dimensional swirl flow model of tesla-type pump devices. Comput Math Appl 77(9):2499–2533. https://doi.org/10.1016/j.camwa.2018.12.035

Alonso DH, Saenz JSR, Silva ECN (2020) Non-Newtonian laminar 2d swirl flow design by the topology optimization method. Struct Multidisc Optim 62(1):299–321. https://doi.org/10.1007/s00158-020-02499-2

Andreasen CS, Gersborg AR, Sigmund O (2009) Topology optimization of microfluidic mixers. Int J Numer Methods Fluid 61:498–513. https://doi.org/10.1002/fld.1964

Ansys I (2006) Modeling turbulent flows - introductory fluent training. www.southampton.ac.uk/~nwb/lectures/GoodPracticeCFD/Articles/Turbulence_Notes_Fluent-v6.3.06.pdf. Accessed Nov 2020

Arnold D, Brezzi F, Fortin M (1984) A stable finite element method for the stokes equations. Calcolo 21:337–344

Bardina JE, Huang PG, Coakley TJ (1997) Turbulence modeling validation, testing and development. Tech. rep, NASA Technical Memorandum, p 110446

Bendsøe MP, Sigmund O (2003) Topology optimization: theory, methods, and applications, 2nd edn. Springer, Berlin

Borrvall T, Petersson J (2003) Topology optimization of fluids in stokes flow. Int J Numer Methods Fluids 41(1):77–107. https://doi.org/10.1002/fld.426

Brezzi F, Fortin M (1991) Mixed and hybrid finite element methods. Springer, Berlin

Bueno-Orovio A, Castro C, Palacios F, Zuazua E (2012) Continuous adjoint approach for the spalart-allmaras model in aerodynamic optimization. AIAA J 50(3):631–646

Chen G, Xiong Q, Morris PJ, Paterson EG, Sergeev A, Wang Y (2014) Openfoam for computational fluid dynamics. Not AMS 61(4):354–363

Dai X, Zhang C, Zhang Y, Gulliksson M (2018) Topology optimization of steady Navier-Stokes flow via a piecewise constant level set method. Struct Multidisc Optim 57(6):2193–2203

De Chant LJ (2005) The venerable 1/7th power law turbulent velocity profile: a classical nonlinear boundary value problem solution and its relationship to stochastic processes. J Appl Math Comput Mech 161:463–474

Deng Y, Zhou T, Liu Z, Wu Y, Qian S, Korvink JG (2018) Topology optimization of electrode patterns for electroosmotic micromixer. Int J Heat Mass Transf 126:1299–1315

Dilgen CB, Dilgen SB, Fuhrman DR, Sigmund O, Lazarov BS (2018) Topology optimization of turbulent flows. Comput Methods Appl

Mech Eng 331:363–393. https://doi.org/10.1016/j.cma.2017.11.029

Duan X, Li F, Qin X (2016) Topology optimization of incompressible Navier-Stokes problem by level set based adaptive mesh method. Comput Math Appl 72(4):1131–1141. https://doi.org/10.1016/j.camwa.2016.06.034

Elhanafy A, Guaily A, Elsaid A (2017) Pressure stabilized finite elements simulation for steady and unsteady Newtonian fluids. J Appl Math Comput Mech 16(3):17–26

Evgrafov A (2004) Topology optimization of navier-stokes equations. Nordic MPS 2004. The Ninth Meeting of the Nordic Section of the Mathematical Programming Society, vol 014. Linköping University Electronic Press, pp 37–55

Farrell PE, Ham DA, Funke SW, Rognes ME (2013) Automated derivation of the adjoint of high-level transient finite element programs. SIAM J Sci Comput 35(4):C369–C393

Franca LP (1992) Stabilized finite element methods: II. The incompressible Navier-Stokes equations. Comput Methods Appl Mech Eng 99(258):209–233

Funke S (2013) The automation of PDE-constrained optimisation and its applications. PhD thesis. Imperial College London

Gersborg-Hansen A (2003) Topology optimization of incompressible Newtonian flows at moderate Reynolds numbers. Master's thesis, Technical University of Denmark

Guest JK, Prévost JH (2006) Topology optimization of creeping fluid flows using a Darcy-Stokes finite element. Int J Numer Methods Eng 66(3):461–484. https://doi.org/10.1002/nme.1560

Hasund KES (2017) Topology optimization for unsteady flow with applications in biomedical flows. Master's thesis, NTNU

He P, Mader CA, Martins JRRA, Maki KJ (2018) An aerodynamic design optimization framework using a discrete adjoint approach with OpenFOAM. Comput Fluids 168:285–303. https://doi.org/10.1016/j.compfluid.2018.04.012

He P, Mader CA, Martins JRRA, Maki KJ (2020) DAFoam: an open-source adjoint framework for multidisciplinary design optimization with OpenFOAM. AIAA J 10(2514/1):J058853

Hyun J, Wang S, Yang S (2014) Topology optimization of the shear thinning non-Newtonian fluidic systems for minimizing wall shear stress. Comput Math Appl 67(5):1154–1170. https://doi.org/10.1016/j.camwa.2013.12.013

Jensen KE, Szabo P, Okkels F (2012) Topology optimizatin of viscoelastic rectifiers. Appl Phys Lett 100(23):234102

Kawamoto A, Matsumori T, Kondoh T (2013) Regularization in topology optimization. ROKS 2013, Leuven

Langtangen HP, Logg A (2016) Solving PDEs in minutes —The FEniCS Tutorial Volume I. https://fenicsproject.org/book/. Accessed Nov 2020

Langtangen HP, Mardal KA, Winther R (2002) Numerical methods for incompressible viscous flow. Adv water Resour 25(8–12):1125–1146

Lazarov BS, Sigmund O (2010) Filters in topology optimization based on Helmholtz-type differential equations. Int J Numer Methods Eng 86(6):765–781

Li BQ (2006) Discontinuous finite elements in fluid dynamics and heat transfer. Springer, London

Logg A, Mardal KA, Wells G (2012) Automated solution of differential equations by the finite element method: The FEniCS book, vol 84. Springer, https://fenicsproject.org/book/

Lv Y, Liu S (2018) Topology optimization and heat dissipation performance analysis of a micro-channel heat sink. Meccanica 53(15):3693–3708

Mangani L, Buchmayr M, Darwish M (2014) Development of a novel fully coupled solver in openfoam: Steady-state incompressible turbulent flows. Numer Heat Transf B 66(1):1–20. https://doi.org/10.1080/10407790.2014.894448

Meurer A, Smith CP, Paprocki M, Čertík O, Kirpichev SB, Rocklin M, Kumar A, Ivanov S, Moore JK, Singh S, Rathnayake T, Vig S, Granger BE, Muller RP, Bonazzi F, Gupta H, Vats S, Johansson F, Pedregosa F, Curry MJ, Terrel AR, Roučka Š, Saboo A, Fernando I, Kulal S, Cimrman R, Scopatz A (2017) Sympy: symbolic computing in python. Peer J Comput Sci. https://doi.org/10.7717/peerj-cs.103

Mitusch S, Funke S, Dokken J (2019) dolfin-adjoint 2018.1: automated adjoints for Fenics and Firedrake. J Open Sour Softw 4(38):1292. https://doi.org/10.21105/joss.01292

Mortensen M, Langtangen HP, Wells GN (2011) A Fenics-based programming framework for modeling turbulent flow by the Reynolds-averaged Navier-Stokes equations. Adv Water Resour 34(9):1082–1101. https://doi.org/10.1016/j.advwatres.2011.02.013

Munson BR, Young DF, Okiishi TH (2009) Fundamentals of fluid mechanics, 6th edn. Wiley, New York

NASA (2019) Turbulence modeling resource—the spalart-allmaras turbulence model . https://turbmodels.larc.nasa.gov/spalart.html

Nørgaard S, Sigmund O, Lazarov B (2016) Topology optimization of unsteady flow problems using the lattice Boltzmann method. J Comput Phys 307(C):291–307. https://doi.org/10.1016/j.jcp.2015.12.023

Olesen LH, Okkels F, Bruus H (2006) A high-level programming-language implementation of topology optimization applied to steady-state Navier-stokes flow. Int. J. Numer. Methods Eng. 65(7):975–1001

OpenFOAM Foundation (2020) Official openfoam repository. https://github.com/OpenFOAM

OpenFOAM Wiki (2014) Openfoam guide/the simple algorithm in openfoam. http://openfoamwiki.net/index.php/The_SIMPLE_algorithm_in_OpenFOAM

Papoutsis-Kiachagias E, Kontoleontos E, Zymaris A, Papadimitriou D, Giannakoglou K (2011) Constrained topology optimization for laminar and turbulent flows, including heat transfer. CIRA, editor, EUROGEN, Evolutionary and Deterministic Methods for Design, Optimization and Control, Capua, Italy

Papoutsis-Kiachagias E, Zymaris A, Kavvadias I, Papadimitriou D, Giannakoglou K (2015) The continuous adjoint approach to the k-ϵ turbulence model for shape optimization and optimal active control of turbulent flows. Eng Optim 47(3):370–389. https://doi.org/10.1080/0305215X.2014.892595

Papoutsis-Kiachagias EM, Giannakoglou KC (2016) Continuous adjoint methods for turbulent flows, applied to shape and topology optimization: industrial applications. Arch Comput Methods Eng 23(2):255–299

Patankar SV (1980) Numerical heat transfer and fluid flow, 1st edn. McGraw-Hill, New York

Pingen G, Maute K (2010) Optimal design for non-Newtonian flows using a topology optimization approach. Comput Math Appl 59(7):2340–2350

Ramalingom D, Cocquet PH, Bastide A (2018) A new interpolation technique to deal with fluid-porous media interfaces for topology optimization of heat transfer. Comput Fluids 168:144–158. https://doi.org/10.1016/j.compfluid.2018.04.005

Reddy JN, Gartling DK (2010) The finite element method in heat transfer and fluid dynamics, 3rd edn. CRC Press, Boca Raton

Rempfer D (2006) On boundary conditions for incompressible Navier-Stokes problems. Appl Mech Rev 59(3):107–125. https://doi.org/10.1115/1.2177683

Romero J, Silva E (2014) A topology optimization approach applied to laminar flow machine rotor design. Comput Methods Appl Mech Eng 279(Supplement Supplement C):268–300. https://doi.org/10.1016/j.cma.2014.06.029

Romero JS, Silva ECN (2017) Non-Newtonian laminar flow machine rotor design by using topology optimization. Struct Multidisc Optim 55(5):1711–1732

Rozvany G (2001) Aims, scope, methods, history and unified terminology of computer-aided topology optimization in structural mechanics. Struct Multidisc Optim 21(2):90–108. https://doi.org/10.1007/s001580050174

Rozvany GIN, Zhou M, Birker T (1992) Generalized shape optimization without homogenization. Struct Optim 4(3):250–252. https://doi.org/10.1007/BF01742754

Sá LFN, Amigo RCR, Novotny AA, Silva ECN (2016) Topological derivatives applied to fluid flow channel design optimization problems. Struct Multidisc Optim 54(2):249–264. https://doi.org/10.1007/s00158-016-1399-0

Sá LFN, Okubo Jr CM, Silva ECN (2021) Topology optimization of subsonic compressible flows. Struct Multidisc Optim 64:1–22. https://doi.org/10.1007/s00158-021-02903-5

Sagebaum M, Albring T, Gauger NR (2018) Expression templates for primal value taping in the reverse mode of algorithmic differentiation. Optim Methods Softw 33(4–6):1207–1231. https://doi.org/10.1080/10556788.2018.1471140

Sato Y, Yaji K, Izui K, Yamada T, Nishiwaki S (2017) Topology optimization of a no-moving-part valve incorporating pareto frontier exploration. Struct Multidisc Optim 56(4):839–851. https://doi.org/10.1007/s00158-017-1690-8

Sato Y, Yaji K, Izui K, Yamada T, Nishiwaki S (2018) An optimum design method for a thermal-fluid device incorporating multiobjective topology optimization with an adaptive weighting scheme. J Mech Des 140(3):031402

Schlichting H (1979) Boundary-layer theory, 7th edn. McGraw-Hill, New York

Sigmund O (2007) Morphology-based black and white filters for topology optimization. Struct Multidisc Optim 33(4):401–424. https://doi.org/10.1007/s00158-006-0087-x

Sigmund O, Petersson J (1998) Numerical instabilities in topology optimization: a survey on procedures dealing with checkerboards, mesh-dependencies and local minima. Struct Optim 16(1):68–75

Sokolowski J, Zochowski A (1999) On the topological derivative in shape optimization. SIAM J Control Optim 37(4):1251–1272

Song XG, Wang L, Baek SH, Park YC (2009) Multidisciplinary optimization of a butterfly valve. ISA Trans 48(3):370–377

Spalart PRA, Allmaras S (1994) A one-equation turbulence model for aerodynamic flows. Cla Recherche Aérospatiale 1:5–21

Towara M, Naumann U (2013) A discrete adjoint model for openfoam. Proced Comput Sci 18:429–438. https://doi.org/10.1016/j.procs.2013.05.206

Wächter A, Biegler LT (2006) On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. Math Program 106(1):25–57

Weller HG, Tabor G, Jasak H, Fureby C (1998) A tensorial approach to computational continuum mechanics using object-oriented techniques. Comput Phys 12(6):620–631

White FM (2011) Fluid mechanics, 7th edn. McGraw-Hill, New York

Wiker N, Klarbring A, Borrvall T (2007) Topology optimization of regions of Darcy and Stokes flow. Int Jo Numer Methods Eng 69(7):1374–1404

Wilcox DC (2006) Turbulence modeling for CFD, 3rd edn. DCW Industries Inc., La Canada

Yoon GH (2016) Topology optimization for turbulent flow with Spalart-Allmaras model. Comput Methods Appl Mech Eng 303:288–311. https://doi.org/10.1016/j.cma.2016.01.014

Zauderer E (1989) Partial differential equations of applied mathematics, 2nd edn. Wiley, Hoboken

Zhang B, Liu X, Sun J (2016) Topology optimization design of non-Newtonian roller-type viscous micropumps. Struct Multidisc Optim 53(3):409–424

Zhou S, Li Q (2008) A variationals level set method for the topology optimization of steady-state Navier-Stokes flow. J Comput Phys 227(24):10178–10195