



# Automating adjoint sensitivity analysis for multidisciplinary models involving partial differential equations

Ru Xiang<sup>1</sup> · Sebastiaan P. C. van Schie<sup>1</sup> · Luca Scotzniovsky<sup>1</sup> · Jiayao Yan<sup>1</sup> · David Kamensky<sup>2</sup> · John T. Hwang<sup>1</sup>

Received: 13 July 2023 / Revised: 21 May 2024 / Accepted: 5 July 2024 / Published online: 14 August 2024  
© The Author(s), under exclusive licence to Springer-Verlag GmbH Germany, part of Springer Nature 2024

## Abstract

We present a general framework for incorporating partial differential equation (PDE)-based models into gradient-based optimization of multidisciplinary systems by integrating FEniCSx with the recently developed Computational System Design Language (CSDL). CSDL is a domain-specific language designed to facilitate adjoint sensitivity analysis multidisciplinary design optimization (MDO). We use CSDL's abstractions to link together sub-models representing different disciplines, not all of which are necessarily modeled by PDEs. For the subsystems which are modeled by PDEs, we use FEniCSx to compute partial derivatives of problem residuals, which CSDL can combine with derivatives from other disciplines using the chain rule and the adjoint method. The development of this framework is motivated by the problem of optimizing designs of electric vertical takeoff and landing (eVTOL) aircraft where, due to the relative novelty of this class of vehicle, there is currently a large, unexplored design space. Predicting the performance of eVTOL aircraft requires PDE-based modeling of various sub-problems, including electric motors, composite shell structures, and thermoelasticity and electrochemistry of battery packs. For system-level analysis and optimization, these must be coupled to non-PDE components, such as low-fidelity aerodynamic models, geometric design tools, and lumped-parameter battery and circuit models. In this work, we demonstrate the modeling flexibility and efficiency of this framework through classic optimal control and topology optimization problems with known solutions, and also challenging eVTOL-related applications including shape optimization of an electric motor, and aeroelastic coupling for gust response of an eVTOL wing. Given the generality of this framework, we expect it to facilitate research on a wide range of PDE-constrained MDO problems beyond eVTOL applications.

**Keywords** Finite element method · Gradient-based optimization · Multidisciplinary design optimization · Aeroelasticity · Electric motor design

## 1 Introduction

The design and optimization of electrical vertical takeoff and landing (eVTOL) aircraft poses significant challenges due to the complexity in modeling and analyzing multiphysics systems. This includes predicting the stability and performance of eVTOL aircraft under different flight conditions, which requires evaluating a system-level model that integrates sub-models from various disciplines, such as weights,

battery design, flight dynamics, and aeroelasticity. In order to achieve a reasonable turnaround time while maintaining sufficient accuracy for global evaluation, the system-level model may contain both high-fidelity PDE-based sub-models and low-fidelity non-PDE sub-models, highlighting the need for a multifidelity computational tool.

Multidisciplinary design optimization (MDO) is a widely used approach in this kind of problems, as it can incorporate all of the disciplines simultaneously, in conjunction with the numerical optimization algorithms (de Weck et al. 2007; Martins and Lambe 2013). MDO stands as a promising modeling technique used to solve complex engineering challenges such as coupled subsystems involving multiple disciplines, especially in the early stage of the design process. Its applications span a broad spectrum, including aircraft wing design (Benaouali and Kachel 2019), wind turbines (Ashuri et al. 2014), and spacecraft design (Taylor 2000; Jilla

---

Responsible editor: Seonho Cho

✉ Ru Xiang  
rxiang@ucsd.edu

<sup>1</sup> Department of Mechanical and Aerospace Engineering,  
University of California San Diego, La Jolla, CA 92093, USA

<sup>2</sup> Coreform LLC, Orem, UT 84097, USA

and Miller 2002). In the context of eVTOL aircraft design, gradient-based MDO is preferable over gradient-free methods, as evaluating high-fidelity sub-models can be extremely expensive in terms of computational cost.

To numerically solve the PDE components in gradient-based MDO problems, we can utilize the finite element method (FEM) to approximate the desired PDE solution in a finite-dimensional function space over a discretized domain. While the finite element method is widely used in structural analysis, it is also applicable to problems in any discipline that can be described by PDEs, such as heat transfer simulations of battery packs (Nelson et al. 2011), fluid–structure interaction of blood vessels (Kamensky 2021), or even weather predictions (Cotter and Shipton 2012).

However, existing tools for finite element analysis and MDO are not naturally compatible with each other and do not fulfill all the requirements mentioned above. Commercial finite element tools, such as Nastran, may come with fully integrated optimization modules, like Nastran SOL200 (Siemens 2014). However, it is challenging to export essential gradient information from a closed-source commercial solver for coupling with other sub-models, such as a low-fidelity particle-based aerodynamic solver built on purely algebraic equations.

Open-source finite element tools are easier to extract derivative information from, but not general enough to suit all disciplines involving PDE models. For example, while TACS (Kennedy and Martins 2014) specializes in structural optimization with beams and shells, it is challenging to be extended to multidisciplinary problems like aeroelasticity, as it can only provide finite element solutions for existing disciplines implemented in its C++ library, which currently supports structures and heat transfer.

FEniCS (Logg et al. 2011) is an open-source tool for solving PDEs that automates the efficient implementation of the finite element method through its abstraction of mathematical modeling and its automatic code generation capability. It uses the Python interface of the Unified Form Language, UFL (Alnæs et al. 2014a) to provide a user-friendly front end, where users only need to provide the mathematical model as a variational form of the PDE residual, and FEniCS will call the FEniCS Form Compiler, FFC (Ølgaard and Wells 2010) to generate finite element code in C++ and solve the problem in the backend of DOLFIN (Logg and Wells 2010).

The library dolfin-adjoint (Mitusch et al. 2019) is an adjoint-based optimization extension of FEniCS that minimizes the effort required to build an adjoint model based on a forward solver. However, due to its dependency on FEniCS, it requires that all sub-models be implemented in the Python interface of FEniCS or Firedrake (Rathgeber et al. 2016), limiting the use of non-PDE components like models represented by algebraic equations for preprocessing and postprocessing steps.

CSDL (Gandarillas et al. 2024) is a recently developed algebraic modeling language that automates the exact derivative computation for its native operations including array operations such as reshaping, reordering, and matrix–vector products, and fundamental operations such as logarithmic, exponential, and trigonometric functions. CSDL is designed to be compatible with external solvers using customized operations, requiring users to define the operations and the associated partial derivatives in the format that CSDL specifies. This allows CSDL to connect the external solvers with the rest of the model automatically and compute the total derivatives needed for the optimization using the chain rule and the adjoint method.

The aforementioned FEniCS is a prototypical external solver that would extend the capability of CSDL for PDE-based high-fidelity modeling. In this paper, we use FEniCSx (Scroggs et al. 2022; Baratta et al. 2023), the new version of the FEniCS library with significant improvements, including support for a broader range of element types and complex number computation, among others. We will refer to them as FEniCSx and the legacy FEniCS, respectively, in the remainder of this paper. In this work, we present a general framework that integrates FEniCSx with CSDL for efficient modeling, simulation and optimization of multidisciplinary and multifidelity systems involving some, but not exclusively, PDE-based subsystems.

We present FEMO,<sup>1</sup> a generalized tool that enables PDE solutions for multiphysics problems and provides automated derivative computation for optimization process. FEMO offers the following key features that enhance research in PDE-constrained MDO:

- (1) Highly modularized design that can be extended to interface external solvers, meeting the needs of large-scale multidisciplinary modeling and optimization
- (2) User-friendly API that minimizes the barrier of FEniCSx programming for optimization users, allowing them to focus on the modeling itself
- (3) Demonstrations of eVTOL-related applications, showcasing efficient modeling for coupling aerodynamic and structural solvers of different levels of fidelity, as well as innovative approach to incorporate shape parametrization into PDE solutions in the motor design application

The remainder of this paper is organized as follows. Section 2 introduces a model problem—namely, the optimal control of a nonlinear elliptic PDE—to illustrate the proposed methodology. It is followed with the implementation of FEMO in Sect. 3. In Sect. 4, we demonstrate the

<sup>1</sup> FEMO stands for **F**inite **E**lements for **M**ultidisciplinary **O**ptimization. The source code of FEMO is available at <https://github.com/RuruX/femo>.

capabilities of this approach through a wide range of applications relevant to eVTOL subsystems, including topology optimization of a cantilever beam, shape optimization of an electric motor, and gust response simulation of an eVTOL wing and the associated sensitivity analysis. Section 5 draws conclusions and discusses possible future extensions of this work and anticipated challenges of applying it to system-level eVTOL aircraft design and optimization.

## 2 Methodology

We start with the definition of a PDE-constrained MDO problem in Sect. 2.1, and introduce a specific model problem—the optimal control of a nonlinear elliptic PDE. This model problem is used to illustrate the workflow of solving this optimization problem, which includes formulating the finite element equation with Nitsche’s method, and obtaining derivatives for gradient-based optimization through the automatic code generation capabilities in FEniCSx.

### 2.1 A glimpse at a PDE-constrained MDO problem

The diagram in Fig. 1 illustrates a representative PDE-constrained MDO process, which involves one or two high-fidelity PDE sub-models, and multiple mid- to low-fidelity algebraic sub-models. The sub-model that computes objective function could be a weighted combination of several sub-functions such as efficiency, cost, weight, or maximum loads.

In this process, coupling can occur in various ways among the sub-models, including one-way coupling and two-way coupling between disciplines. For instance, the two-way coupling between Discipline 1 and Discipline 2 could be used for load and displacement updates between an aerodynamic solver and a structural solver in aeroelastic analysis; the one-way coupling between Discipline  $n$  and Discipline  $n - 1$  could represent postprocessing for electric power losses based on solutions of the Maxwell’s equations in electromagnetic modeling.

### 2.2 A model problem: optimal control of a nonlinear elliptic PDE

The methodology for solving a PDE-constrained MDO problem is presented as a generalized approach. To illustrate the workflow, we walk through the modeling process using the optimal control of a nonlinear elliptic PDE as the model problem (Pörner and Wachsmuth 2017). This problem is commonly referred to as an inverse problem in the mathematical community, as the goal involves determining the source term given the analytical solution of the PDE. The mathematical model of this problem is formulated as

$$\text{minimize } J(u, f) = \frac{1}{2} \|u - g\|_{L^2}^2 + \frac{\alpha}{2} \|f\|_{L^2}^2$$

with respect to  $f$  (1a)

$$\text{such that } \begin{aligned} -\Delta u + u^3 &= f & \text{in } \Omega \\ u &= g & \text{on } \Gamma, \end{aligned}$$

(1b)

where  $\Omega$  is the domain of interest,  $\Gamma$  is the outer boundary of  $\Omega$ . The state variable  $u : \Omega \rightarrow \mathbb{R}$  and the control variable (or design variable)  $f : \Omega \rightarrow \mathbb{R}$  are constrained by a boundary value problem (BVP) represented by (1b). The BVP consists of a nonlinear elliptic equation, specifically a nonlinear Helmholtz equation in this case, and subject to a Dirichlet boundary condition. The objective function, defined in (1a), is formulated as the sum of the  $L^2$  norm of the difference between the state variable  $u$  and the analytical solution  $g$ , and an  $L^2$  regularization term that depends on the control variable  $f$ .

The goal of this optimal control problem is to determine the optimal values of  $f$  that lead to a state variable  $u$  with the desired profile matching the analytical solution  $g$ , while satisfying the nonlinear elliptic PDE associated with  $f$ . This model problem is relatively simple and easily abstracted in the optimization problem setting, but due to the nonlinearity in the PDE it still presents a challenge.

### 2.3 Finite element solution of the model problem

Recall the strong form of the nonlinear Helmholtz equation in (1b). Find  $u$ , such that

$$\begin{aligned} -\Delta u + u^3 &= f & \text{in } \Omega, \\ u &= g & \text{on } \Gamma. \end{aligned}$$

The weak form is formulated as below. Find  $u \in \mathcal{V}_g$ , such that  $\forall v \in \mathcal{V}_0$ ,

$$\int_{\Omega} \nabla u \cdot \nabla v \, d\Omega + \int_{\Omega} u^3 v \, d\Omega = \int_{\Omega} f v \, d\Omega, \tag{2}$$

where  $\mathcal{V}_g$  and  $\mathcal{V}_0$  are the trial and the test spaces defined as

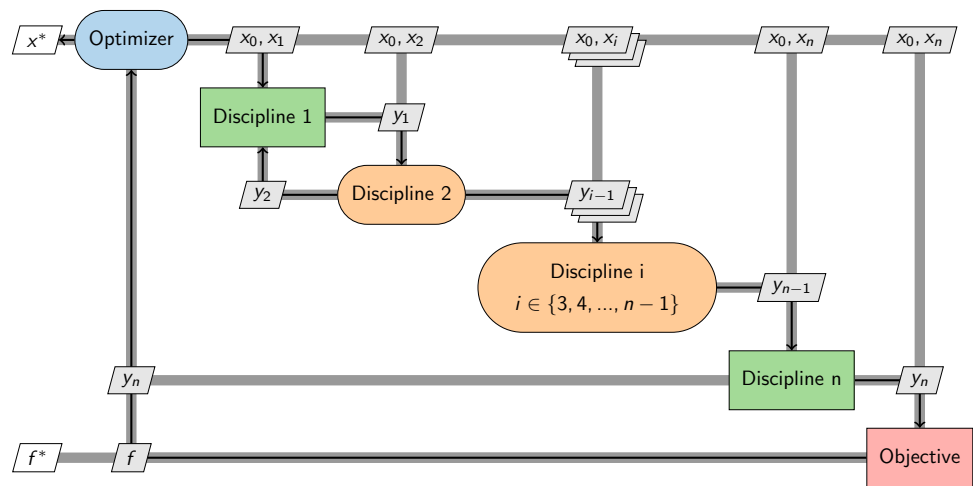
$$\begin{aligned} \mathcal{V}_g &:= \left\{ u \in H^1(\Omega) : u|_{\partial\Omega} = g \right\}, \\ \mathcal{V}_0 &:= \left\{ v \in H^1(\Omega) : v|_{\partial\Omega} = 0 \right\}. \end{aligned} \tag{3}$$

Let  $u_h, v_h$  be the Galerkin approximation of  $u, v$ . The discrete problem then becomes find  $u_h \in \mathcal{V}_h$ , such that  $\forall v_h \in \mathcal{V}_h$ ,

$$\int_{\Omega} \nabla u_h \cdot \nabla v_h \, d\Omega + \int_{\Omega} u_h^3 v_h \, d\Omega = \int_{\Omega} f v_h \, d\Omega, \tag{4}$$

where  $\mathcal{V}_h$  is a finite-dimensional subspace of  $H^1(\Omega)$ .

**Fig. 1** A representative MDO process that encapsulates the most challenging aspects typically encountered in such problems: (a) coupling multiple high-fidelity PDE sub-models; (b) combining sub-models for disciplines with and without PDEs. The orange boxes represent the high-fidelity PDE sub-models, while the green boxes represent the mid- to low-fidelity non-PDE sub-models. The pink box represents the sub-model that computes the objective function. (Color figure online)



In order to handle the Dirichlet boundary condition in a more systematic manner throughout the optimization process, Nitsche’s method (1971) is employed for weak enforcement. This approach involves introducing additional boundary terms to both sides of (4). By doing so, the boundary condition is automatically incorporated into the weak form and its associated partial derivatives with respect to the state variable  $u$ . With the boundary terms added to (4), we have

$$\begin{aligned}
 & \int_{\Omega} \nabla u_h \cdot \nabla v_h \, d\Omega + \int_{\Omega} u_h^3 v_h \, d\Omega - \int_{\Gamma} (\nabla u_h \cdot \mathbf{n}) v_h \, d\Gamma \\
 & \mp \int_{\Gamma} (\nabla v_h \cdot \mathbf{n}) u_h \, d\Gamma + \int_{\Gamma} \frac{C_{\text{pen}}}{h} u_h v_h \, d\Gamma \quad (5) \\
 & = \int_{\Omega} f v_h \, d\Omega \mp \int_{\Gamma} (\nabla v_h \cdot \mathbf{n}) g \, d\Gamma + \int_{\Gamma} \frac{C_{\text{pen}}}{h} g v_h \, d\Gamma,
 \end{aligned}$$

where  $\mathbf{n}$  is the unit normal vector pointing outwards of the outer boundary  $\Gamma$ ,  $h$  is a parameter indicating the refinement of the mesh (i.e., the cell diameter of the discretized domain), and  $C_{\text{pen}}$  is a positive constant independent of the mesh refinement. Rearranging terms we obtain the residual form of the nonlinear Helmholtz equation:

$$\begin{aligned}
 R(u_h, f) &= \int_{\Omega} \nabla u_h \cdot \nabla v_h \, d\Omega + \int_{\Omega} u_h^3 v_h \, d\Omega \\
 & - \int_{\Omega} f v_h \, d\Omega - \int_{\Gamma} (\nabla u_h \cdot \mathbf{n}) v_h \, d\Gamma \\
 & \mp \int_{\Gamma} (\nabla v_h \cdot \mathbf{n}) (u_h - g) \, d\Gamma \\
 & + \int_{\Gamma} \frac{C_{\text{pen}}}{h} (u_h - g) v_h \, d\Gamma. \quad (6)
 \end{aligned}$$

The  $\mp$  sign in (6) corresponds to the symmetric (–) and non-symmetric (+) variants of Nitsche’s method. In this paper’s applications, the symmetric variant is utilized. It is widely accepted by the finite element community and offers optimal accuracy in both  $L^2$  and  $H^1$  norms (Annavarapu et al. 2012;

Jiang et al. 2015). The nonsymmetric Nitsche’s method has also gained increasing attention for its independence of the choice of penalty parameter  $C_{\text{pen}}$  (Schillinger et al. 2016). However, we do not delve into the detailed comparison of these two variants as it is beyond the scope of this paper.

### 2.4 Automated derivative computation for gradient-based optimization

Gradient-based optimization is a preferred approach in MDO problems that involves evaluating high-fidelity sub-models, such as finite element solvers. It uses derivative information from the model to determine the direction of design variable updates, offering efficiency and accuracy. However, a major challenge in gradient-based optimization is ensuring that each component of the model has the required partial derivatives of its outputs with respect to its inputs. These derivatives are essential for computing the total derivative of the objective function with respect to the design variables.

To address this challenge, FEMO automates derivative computation as much as possible. It leverages the automatic differentiation capabilities of FEniCSx to compute partial derivatives of PDE residuals with respect to function inputs (i.e.,  $\partial R/\partial u$  and  $\partial R/\partial f$ ), and utilizes CSDL for partial derivatives of algebraic operations, such as matrix–vector multiplications, occupying in preprocessing and postprocessing code.

For the model problem in (1b), the total derivative of the objective function  $J$  with respect to the inputs  $f$  can be computed using the chain rule

$$\frac{dJ}{df} = \frac{\partial J}{\partial f} + \frac{\partial J}{\partial u} \frac{du}{df}. \quad (7)$$

With  $\partial R/\partial u$  and  $\partial R/\partial f$  provided by FEniCSx, we can obtain  $du/df$  by solving the linear system in (8), and insert it to (7) to compute the total derivative  $dJ/df$ .

**Listing 1** A CSDL model computing  $y = x_1^3 + \sin(x_2) * \cos(x_2)$  with build-in operations

```

class ExampleModel(csd1.Model):
    def define(self):
        # Declare inputs with default values
        x1 = self.declare_variable('x1', val=1.)
        x2 = self.declare_variable('x2', val=1.)

        # Compute the output with built-in operations
        y = x1**3 + sin(x2)*cos(x2)
        self.register_output('y', y)

# Run the simulation with the Python backend
sim = Simulator(ExampleModel())
sim.run()

```

$$R(u, f) = 0 \Rightarrow \frac{dR}{df} = 0 \Rightarrow \frac{\partial R}{\partial u} \frac{du}{df} = -\frac{\partial R}{\partial f}. \quad (8)$$

The assembly of partial derivatives to obtain total derivatives is handled automatically by CSDL, where it employs the direct or adjoint method based on the computational cost.

### 3 Implementation

In this section, we present a detailed explanation of how FEMO is used to solve PDE-constrained MDO problems. We begin with examples of FEniCSx and CSDL implementation, and then walk through the steps involved in solving the model problem using FEMO.

In Sect. 3.1, we introduce the native operations in CSDL such as matrix–vector products, and the `Custom Operation` class object, which serves as a wrapper for FEniCSx related operations. In Sect. 3.2, we provide a detailed explanation of FEniCSx programming workflow, accompanied by code examples for essential functionalities, such as computing partial derivatives and solving linear or nonlinear systems. In Sect. 3.3, we demonstrate how all the components are assembled together using the Python API of FEMO.

#### 3.1 System-level modeling construction using CSDL

CSDL is a domain-specific language for constructing computational models in MDO. It provides access to a diverse set of operations such as summation, averaging, trigonometric functions, and array operations. Listing 1 demonstrates a simple model that utilizes built-in CSDL operations. In CSDL, the computation of partial derivative associated with built-in operations is automated. The execution of CSDL models is performed by the `Simulator` object in the Python backend of CSDL, which is implemented using Numpy.

In addition to the built-in operations, CSDL also supports user-defined custom operations, including *implicit* and *explicit* custom operations. *Implicit* custom operations typically involve solving a linear or nonlinear system to compute the output, making them suitable for wrapping PDE solvers. We will demonstrate how we utilize custom operations as wrappers for FEniCSx operations in Sect. 3.3. A CSDL model can contain multiple sub-models, where each of them has its own operation defined, whether built-in or custom. The total derivatives of models with coupled sub-models are computed using either direct or adjoint methods, depending on the number of design variables and the number of objectives. If there are more design variables than objectives, the adjoint method is executed, which is often the case for MDO problems.

#### 3.2 Automated finite element solutions and derivatives with FEniCSx

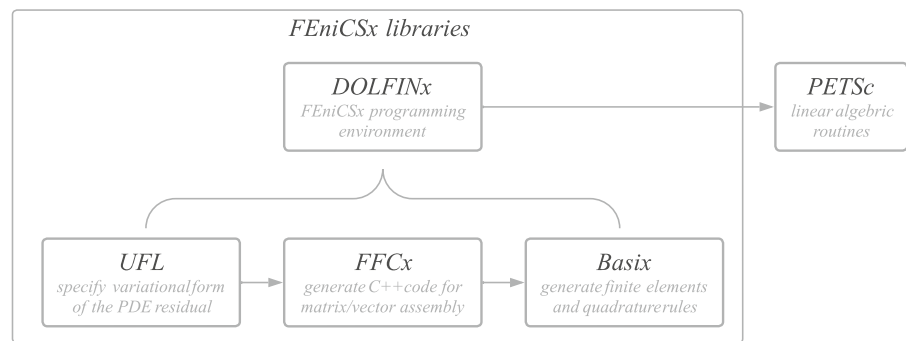
FEniCSx is a collection of scientific libraries (UFL, DOLFINx, FFCx, Basix) written in Python and C++, where each library has a specific functionality in solving finite element problems. Figure 2 illustrates the workflow between these libraries and their respective functionalities.

In the workflow for writing a FEniCSx program in Python, the user begins by importing the mesh using the XDMF file reader in DOLFINx. Next, they specify the variational form of the PDE residual using the Uniform Form Language, UFL (Alnæs et al. 2014b), which is a domain-specific language for variational expressions. DOLFINx then automatically interfaces with FFCx to generate high-performance finite element code in C++ on the backend.

For example, when solving the model problem, we first create a unit square mesh with triangular elements as the



Fig. 2 FEniCSx workflow



analysis domain. We then define a Lagrange finite element space for the solution  $u$  of the nonlinear elliptic PDE of  $C^1$  continuity, as shown in Listing 2.

PETSc arrays. These arrays can be further converted to Numpy or Scipy arrays to interface with CSDL. This process is generalized by FEMO as described in Sect. 3.3.

Listing 2 Runscript for solving the model problem

```

N = 16
# Create a 2D mesh with N elements on each edge
mesh = dolfinx.mesh.create_unit_square(N, N)
# Create a Lagrange function space for state variables
# with C^1 continuity
V = dolfinx.fem.FunctionSpace(mesh, ('CG', 1))
u = dolfinx.fem.Function(V)
  
```

Next, we express the interior terms and the boundary terms with symmetric Nitsche's method of the PDE residual using UFL operations. The code snippet in Listing 3 corresponds to the mathematical formulation described in (6).

FEniCSx offers convenient build-in linear and nonlinear solvers for users and allows for the flexibility of custom solvers tailored to specific problem requirements. In

Listing 3 (Continued) runscript for solving the model problem

```

# Define the weak form of the nonlinear Helmholtz equation
interior_residual = inner(grad(u), grad(v))*dx \
    + inner(u**3, v)*dx \
    - inner(f, v)*dx
boundary_residual = - inner(dot(grad(u), n), v)*ds \
    - inner(u-g, dot(grad(v), n))*ds \
    + C_pen*h_E**(-1)*inner(u-g, v)*ds
R = interior_residual + boundary_residual
  
```

The UFL also provides analytical computation of partial derivatives, such as  $\partial R/\partial u$  and  $\partial R/\partial f$  for the PDE residual  $R$  defined in Listing 3, i.e.,

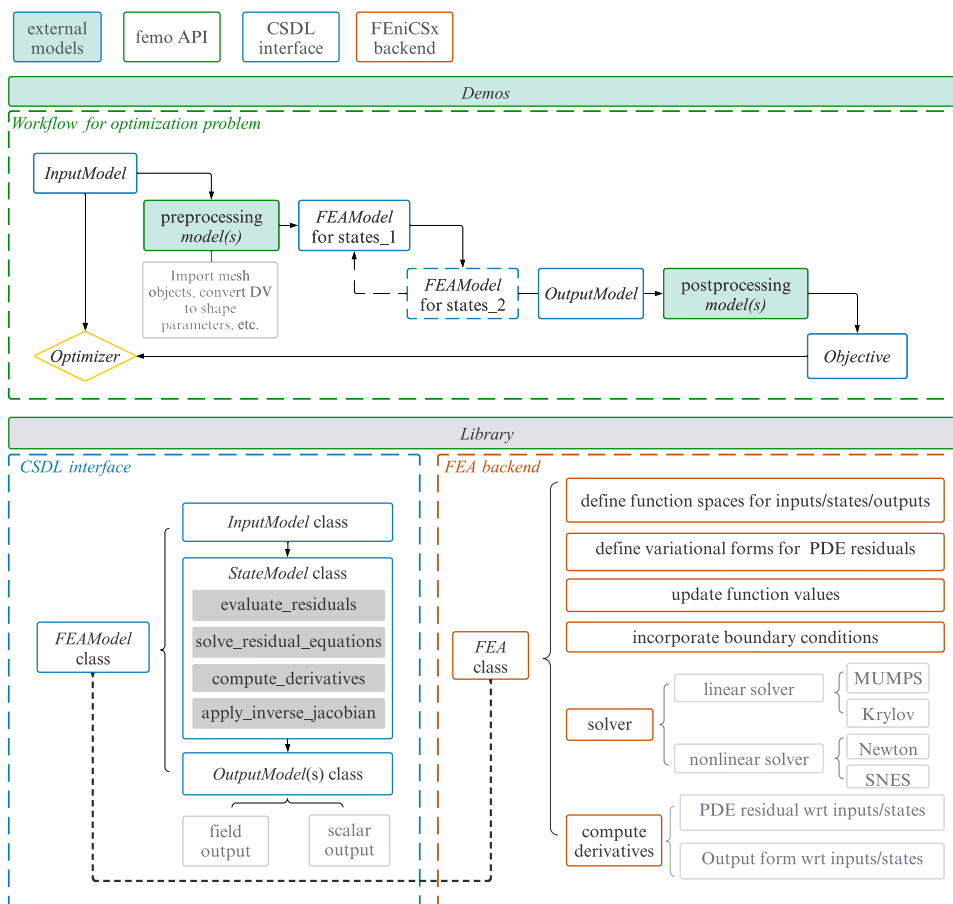
```

dRdu = ufl.derivative(R, u) #
    is a variational form
  
```

The analytical derivatives computed by UFL can be assembled into discrete form using DOLFINx, resulting in

FEMO, we utilize customized nonlinear solvers such as NewtonSolver and SNESSolver with MUMPS as the linear solver from PETSc. The code in FEMO is designed to be case-independent, enabling the solution of PDE problems in any discipline. The implementation details of the solvers, as well as other modularized components such as

Fig. 3 FEMO framework



derivative form assembly and function value extraction, will be discussed in Sect. 3.3.

### 3.3 Coupling of FEniCSx and CSDL through FEMO

Integrating FEniCSx and CSDL for PDE-constrained MDO can be challenging due to their different domain-specific languages and data storage methods. FEniCSx uses UFL to express variational terms, such as integration and differentiation and stores data as PETSc sparse arrays. On the other hand, CSDL utilizes Numpy arrays for algebraic operations. Translating between these two software can be a tedious task, but it can be streamlined through the use of generic class objects. FEMO addresses this challenge by providing a generalized modeling framework for FEA-related computations and automating the data communication between the two libraries, as illustrated in Fig. 3.

The code example below showcases two convenient utility functions provided by FEMO: `assembleMatrix` converts analytical expression to discrete form, and `solveNonlinear` solves the finite element problem. These functions can be easily utilized within the CSDL modeling environment.

```

dRdu_array = assembleMatrix(
    dRdu)
solveNonlinear(R, u, solver='
    Newton')
    
```

By integrating all the utility functions within FEMO's FEA class object, we can provide essential functionalities required by CSDL's CustomOperation. This integration enables the construction of a generic `StateModel` for computing PDE solutions, and an `OutputModel` for assembling variational forms. Listing 4 is a code snippet illustrating the implementation of the generic `StateModel` in FEMO's source code. It takes a user-defined FEA object, along with input and output names, as parameters.

**Listing 4** FEMO source code for the `StateModel` class

---

```

class StateModel(Model):
    def initialize(self):
        self.parameters.declare('fea', types=femo.FEA)
        self.parameters.declare('output_name', types=str)
        self.parameters.declare('input_name_list', types=list)

    def define(self):
        e = StateOperation(fea=self.fea,
                          input_name_list=input_name_list,
                          output_name=output_name)
        state = csdl.custom(*input_name_list, op=e)
        self.register_output(output_name, state)

# Define an implicit operation
class StateOperation(csdl.CustomImplicitOperation):
    ...

```

---

For the model problem, we utilize these generic models to create problem-specific CSDL custom operation models. These models can take FEniCSx `Function` or `Form` objects (i.e., `u` is a `Function` object defined in Listing 2) as the inputs and use generalized FEniCSx operations in FEMO to compute the outputs, as shown in Listing 5. The `type="scalar"` argument specifies that this model returns a scalar output, i.e., an integral over a certain domain. There is also a "field" type output, which computes distributed field outputs on the finite element mesh, i.e., element-wise and point-wise properties. The `FEAModel` class is employed to group finite element-related operations together. It provides the flexibility to connect multiple PDE systems by including multiple `FEA` objects in its list argument, i.e., `FEAModel(fea=[fea1, fea2])`. This is used in the motor example in Sect. 4.3.

The coupling of sub-models in FEMO is automatically handled based on the names of the variables. Each input or output variable in the sub-models has a unique name associated with the object that stores its values. These names are automatically propagated throughout the system by CSDL and are used to connect to sub-models of other disciplines. Therefore, it is crucial to maintain consistency in variable names throughout the modeling process.

Finally, we utilize the SNOPT library—a powerful software for solving nonlinear optimization problems (Gill et al. 2005)—as the optimizer for our numerical tests. We leverage the Python binding of SNOPT in `modOpt`,<sup>2</sup> which can interface with CSDL by creating a `CSDLProblem` object with the simulator, as demonstrated in Listing 6.

<sup>2</sup> <https://github.com/LSDOlab/modopt>.



**Listing 5** (Continued) runscript for solving the model problem

---

```

from femo.fea.fea_dolfinx import FEA # FEniCSx operations
from femo.csdل_opt.fea_model import FEAModel # CSDL interface
fea = FEA(mesh)
fea.add_input(name='f', function=f)
fea.add_state(name='u', function=u, residual_form=R,
              arguments=['f'])
fea.add_output(name='objective',
               # Can be either scalar or field variable
               type='scalar',
               # Defined as a UFL form object
               form=output_form,
               arguments=['u', 'f'])

# Can contain multiple fea objects for coupled PDE system
# Consists of one state model and several output models
fea_model = FEAModel(fea=[fea])

```

---

**Listing 6** (Continued) runscript for solving the model problem

---

```

from modopt.csdل_library import CSDLProblem
from modopt.snopt_library import SNOPT

# Create a CSDL problem object in modOpt
prob = CSDLProblem(problem_name='nonlinear_Helmholtz_opt',
                  simulator=sim)
# Create an optimizer object in modOpt and solve
optimizer = SNOPT(prob, Major_iterations = 1000,
                  Major_optimality = 1e-9)
optimizer.solve()

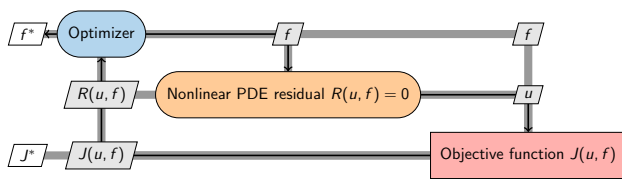
```

---

## 4 Numerical examples

In this section, we demonstrate the versatility and effectiveness of FEMO through four numerical examples that span various disciplines and complexity levels in modeling. The problem set includes three well-posed optimization problems and one sensitivity analysis example. In Sect. 4.1, we continue using the model problem discussed in previous sections as an example of a pure PDE-constrained problem, as there is no other algebraic sub-models coupled with it. In Sect. 4.2, we solve a classic density-based topology optimization problem for a 2D cantilever beam. This example demonstrates the coupling of a PDE with non-PDE components, specifically

the use of a density filter as a preprocessor. We compare its runscript with pure FEniCS implementation in “Appendix A” to show the differences between them. In Sect. 4.3, we tackle a more complex real-world problem for shape optimization of an electric motor. This problem involves solving two coupled PDEs: one for mesh deformation and the other for Maxwell’s equations. Additionally, the PDE components are coupled with external preprocessors implemented in CSDL for shape parametrization. In Sect. 4.4, we construct a multifidelity aeroelastic solver that combines the Vortex Lattice Method, VLM (Anderson 2010) implemented in CSDL with the Reissner–Mindlin shell solver implemented in FEniCSx. We utilize this aeroelastic solver in a dynamic test case simulating the gust response of an eVTOL wing. We also



**Fig. 4** The MDO process for the optimal control of a nonlinear Helmholtz equation. The orange box represents the finite element solver, and corresponds to the `StateModel` in FEMO; the pink box represents the objective function written as a variational form, corresponding to the `OutputModel` in FEMO. (Color figure online)

showcase the automatic computation of derivatives for this coupled system in a static test case.

All three optimization examples utilize the Python binding of SNOPT, accessible through `modOpt`, as the optimizer. We tune the optimizer using two settings: the `(major) optimization tolerance`, closely linked to the objective gradient, and the `(major) feasibility tolerance`, determining the tightness of the constraint enforcement. The values of these two tolerances along with the number of iterations are discussed in the numerical results of each example.

## 4.1 Optimal control of a nonlinear elliptic PDE

### 4.1.1 Mathematical formulation

The MDO process for the model problem described in (1b) is illustrated in Fig. 4. The optimal solution of the problem is denoted as  $f^*$ , and the associated state variable is represented by  $u^*$ . The objective function value at the end of the optimization iterations is denoted as  $J^*$ , i.e.,  $J^* = J(u^*, f^*)$ .

### 4.1.2 Numerical results

For numerical analysis, we consider a unit square domain  $\Omega = [0, 1] \times [0, 1]$  and choose  $g(x) = \sin(2\pi x_1) \cos(\pi x_2)$  as the analytical solution of  $u$  on  $\Omega$ . We use symmetric Nitsche's method to weakly enforce the Dirichlet boundary condition  $u = g$  on  $\Gamma$ .  $\alpha$  is a dimensionless constant used in the regularization term of the objective function in (1a). For our specific case, we have selected  $\alpha = 6 \times 10^{-7}$ . With the known analytical solution  $g$ , the optimal solution of  $f$  can be calculated analytically using the strong form of the nonlinear Helmholtz equation in (1b). Specifically, we have  $f_d = -\Delta g + g^3$ , which can be easily implemented using UFL as

$$f\_d = -ufl.div(ufl.grad(g)) + g**3$$

where  $g$  is a UFL form object defined beforehand.

The optimization is terminated when the optimality drops below a predefined tolerance. At the end of the optimization,

we expect that the results for  $u$  and  $f$  will closely approximate their respective analytical solutions, i.e.,  $u^* \rightarrow g$  and  $f^* \rightarrow f_d$ . To evaluate the success of the optimization problem, we calculate the  $L_2$  norms for both pairs. Figure 5 illustrates the evolution of the control variable  $f$  and the state variable  $u$  during the optimization test with a  $128 \times 128$  mesh. The initial guess for all degrees of freedom in  $f$  is set to 0.1. This optimization run requires 570 iterations to converge to a tolerance of  $10^{-11}$ . In this problem, the control variable  $f$  is represented as a discontinuous Lagrange function of degree 0. Initially, the distribution of  $f$  exhibits a scattering profile for  $N = 200$ . As the optimization progresses, it gradually becomes smoother. This smoothing effect can be attributed to the presence of  $f$  in the regularization term of the objective function  $J$ .

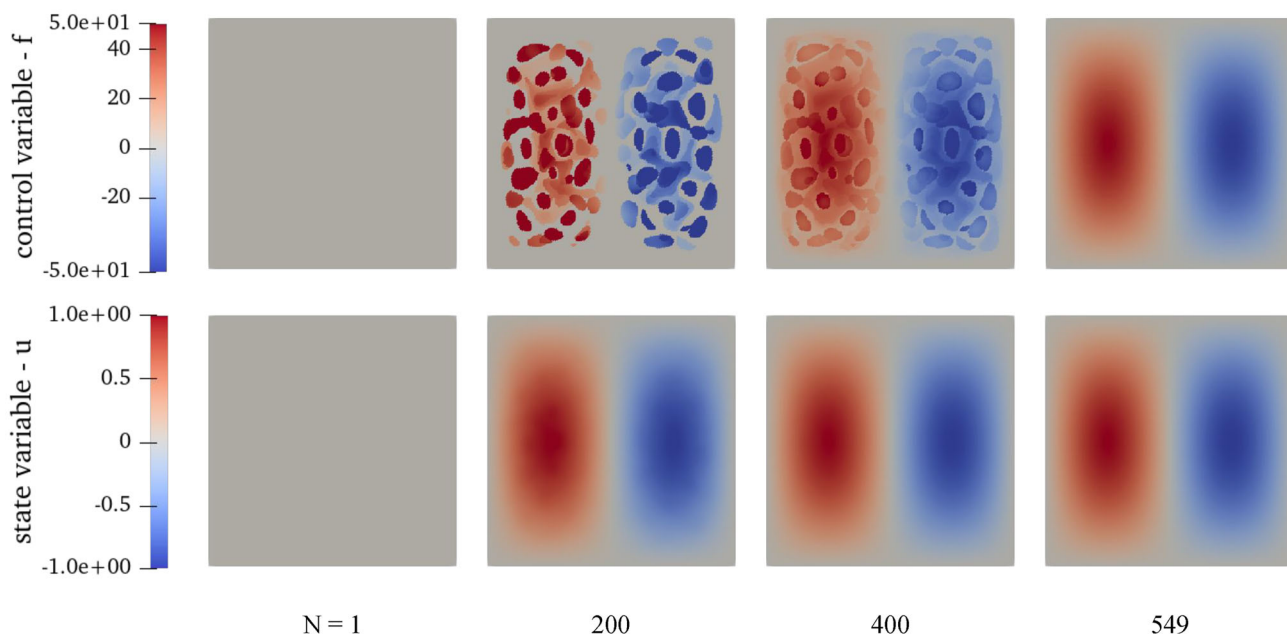
We perform a convergence study on mesh sensitivity by running optimization tests with different meshes. The results are summarized in Table 1. As observed, all the values in the last three columns decrease as the mesh becomes finer. Specifically, the error of the control variable  $f^*$  exhibits the expected first-order convergence, as reported in Arada et al. (2002). Another observation is that as the mesh size decreases, tightening the optimality tolerance becomes necessary, along with an increased number of SNOPT optimization iterations, to ensure this first-order convergence of the error in control. We showcase the convergence history for the test case with a  $16 \times 16$  mesh in Fig. 6 by plotting the objective function  $J$  and optimality over the optimization iterations. This plot demonstrates a monotonic convergence for the objective function, which serves as verification for both accuracy and efficiency of the adjoint computation.

## 4.2 Topology optimization of a cantilever plate

We demonstrate the versatility of this software by applying it to a classical 2D cantilever beam topology optimization problem using a density-based approach. This problem is a well-known benchmark in the field of topology optimization, and has been extensively studied using various optimization approaches. Notably, Chung et al. (2019) implemented the problem using OpenMDAO (Gray et al. 2019), a NASA developed MDO framework, while Yan et al. (2022) automated gradient-based topology optimization by coupling the legacy FEniCS and OpenMDAO in their previous work.

### 4.2.1 Mathematical formulation

We aim to optimize the density distribution of the material, denoted as  $\rho$ , subjected to a constant volume constraint. The structural behavior of the beam is governed by the PDE of linear elasticity. The objective of this optimization problem is to minimize the compliance of the beam, which is defined



**Fig. 5** Snapshots of the optimization process for the model problem with a  $128 \times 128$  mesh. The control variable  $f$  has a uniform value of 0.1 across the entire domain at the first optimization iteration ( $N = 1$ ) and the optimization converges at  $N = 570$

**Table 1** A mesh sensitivity study for the optimal control of the nonlinear Helmholtz problem

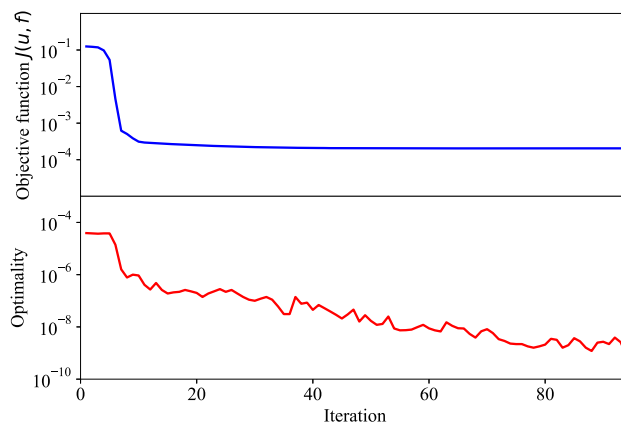
Mesh	Optimality tolerance	SNOPT iterations	$\ f^* - f_d\ _{L_2}$	$\ u^* - g\ _{L_2}$	$J^*$
$4 \times 4$	$10^{-9}$	74	24.017097	0.075343	0.003312
$8 \times 8$	$10^{-9}$	115	6.999477	0.016861	0.000374
$16 \times 16$	$10^{-9}$	93	2.824094	0.004063	0.000205
$32 \times 32$	$10^{-9}$	299	1.376501	0.001253	0.000190
$64 \times 64$	$10^{-10}$	517	0.673929	0.000806	0.000187
$128 \times 128$	$10^{-11}$	570	0.331941	0.000767	0.000187

Each row corresponds to a successfully converged optimization problem with the associated mesh. The  $L_2$  norms,  $\|f^* - f_d\|_{L_2}$ , and  $\|u^* - g\|_{L_2}$ , represent the differences between the optimized control and state variables and their respective analytical solutions.  $J^*$  is the objective function value by the end of the optimization

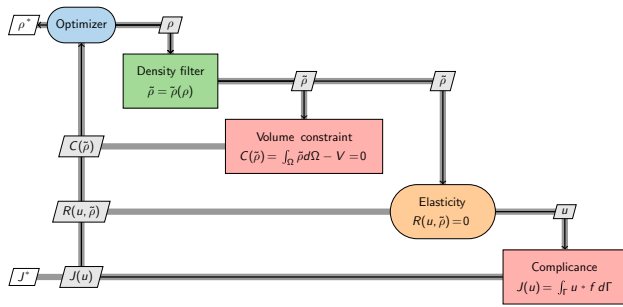
as the inverse of the stiffness, as indicated in the first equation of (9).

$$\begin{aligned}
 &\text{minimize } J(\mathbf{u}) = \int_{\Gamma} \mathbf{u} \cdot \mathbf{f} \, d\Gamma \\
 &\text{with respect to } \rho \\
 &\text{such that } \tilde{\rho} = \tilde{\rho}(\rho) \\
 &\int_{\Omega} \tilde{\rho} \, d\Omega = \text{constant} \\
 &-\nabla \cdot \sigma(\mathbf{u}, \tilde{\rho}) = \mathbf{f} \\
 &\rho_{\min} < \rho \leq 1.
 \end{aligned} \tag{9}$$

To smooth the density distribution and prevent checkerboard patterns, which can cause instability, a distance-based method is employed to map the original density  $\rho$  to the filtered density  $\tilde{\rho}$ . For each element  $i$ , its filtered density is



**Fig. 6** Convergence history for the model problem with a  $16 \times 16$  mesh. The optimization converges after 93 iterations with a total runtime of 28 s



**Fig. 7** The MDO process for topology optimization of a cantilever beam

computed by summing the weighted contributions of nearby elements, given by the equation:

$$\tilde{\rho}_i = \sum_j w_{ij} \rho_j, \tag{10}$$

where  $\tilde{\rho}_i$  is the filtered density of element  $i$ ,  $\rho_j$  is the unfiltered density of a nearby element  $j$ , and  $w_{ij}$  is the weight function that connects element  $i$  and  $j$ , which satisfies  $\sum_j w_{ij} = 1$ . In this example, we use a simple conic weight function to compute  $w_{ij}$  based on the distance between element  $i$  and  $j$ , whose formula can be found in Svanberg and Svård (2013).

The solid isotropic material with penalization (SIMP) method is employed to incorporate the density distribution into the PDE constraint, which was initially developed by Bendsøe and Kikuchi (1988). In this specific application, the local material property is proportional to the power product of the local density, i.e.,  $E_i(\tilde{\rho}_i) = \tilde{\rho}_i^3 E_0$ . Additionally, a minimum value of density, denoted as  $\rho_{\min}$ , is imposed to avoid material singularity and maintain numerical stability during the finite element analysis. Figure 7 shows the MDO process of this problem. The distance-based density filter sub-model is implemented with CSDL linear algebra operations, and marked in green as it is considered as an external component, while the finite element sub-models are marked in pink (OutputModel) and orange (StateModel).

### 4.2.2 Numerical results

The physical setting of the boundary value problem is shown in Fig. 8a. The compliance of a cantilever beam is minimized with a volume fraction constraint of 0.4. The beam has a clamped boundary condition on the left edge, and a downward traction force applied in the middle of the right edge. The beam has a length of 160 and a height of 80. It is discretized into a structured quadrilateral mesh, with user-defined number of elements on each side. Linear elasticity is assumed in this example, but it can be extended to non-linear cases as discussed in Yan et al. (2022). The Young’s

modulus is set to 1.0 and the Poisson’s ratio is set to 0.3. The traction force, denoted as  $F$ , has a magnitude of 0.25 and is directed downwards.  $\rho_{\min}$  is set to  $10^{-4}$  as the lower bound of the design variable. Note that all of the parameters are dimensionless.

We showcase the optimization results and convergence history for the test case with a  $80 \times 40$  mesh in Fig. 8b and c, which demonstrates a generally monotonic convergence for the compliance. The optimization terminates within 338 major iterations with a total runtime of 177 s. The optimality tolerance is set to be  $10^{-8}$  and the feasibility tolerance for enforcing the volume constraint is set to be  $10^{-6}$  in SNOPT. Given that the feasibility is always zero for most of the iterations (indicating that the linear constraints are always satisfied), the focus of Fig. 8c is on the convergence history of the objective function and the optimality. The optimized solution in Fig. 8b for the penalized density  $\tilde{\rho}$  exhibits a tapered shape as anticipated in prior studies (Chung et al. 2019; Yan et al. 2022). Furthermore, due to the symmetric boundary condition and load, the density distribution displays a symmetric profile with linear elastic material model.

## 4.3 Electric motor design

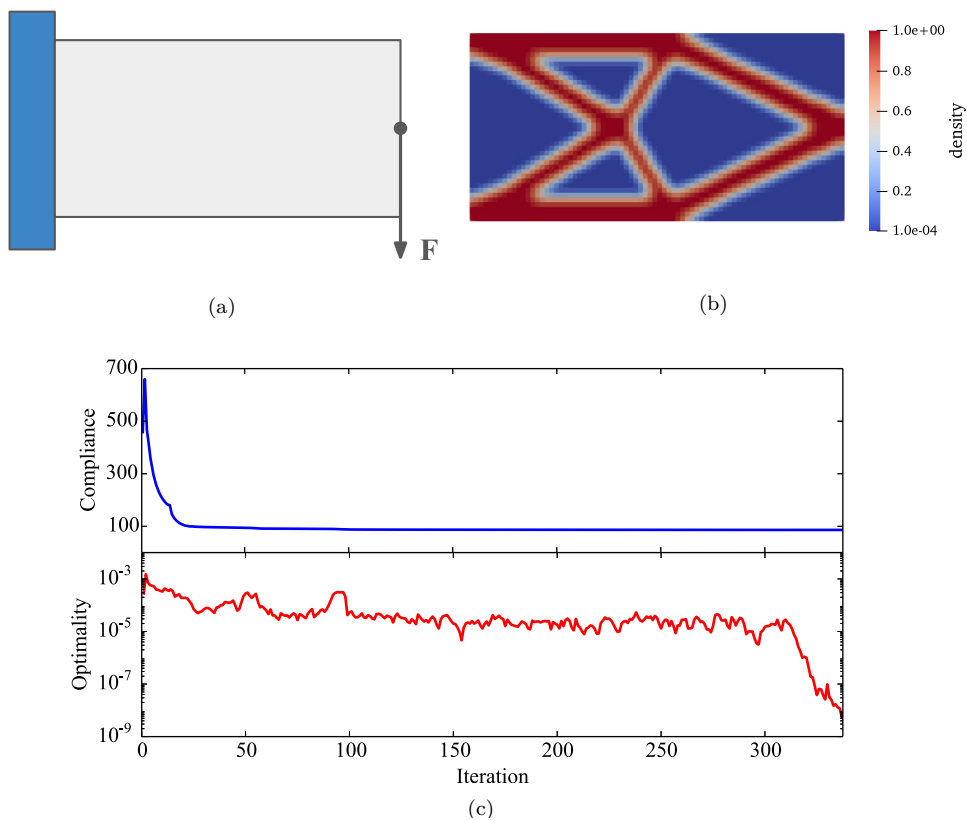
In this section, we tackle a practical problem including the shape optimization of an electric motor. This problem is an essential part for the system-level modeling and optimization of eVTOL aircraft. Using the presented tool, we are able to build a high-fidelity model that can accurately capture the precise mesh movement near the motor components and their impact on the electromagnetic performance of the motor. This is achieved by formulating the relevant processes as coupled PDEs and connecting them with an external solver for shape parametrization. Specifically, we focus on studying a permanent magnet synchronous motor (PMSM), and analyze the behavior of a single static phase within its periodic operation.

### 4.3.1 Optimization problem definition

The optimization problem is formulated shown in (11), where  $\mathbf{x}$  represents a vector of design variables that define the shape of the motor.  $\Omega_{bc}$  is a vector of boundary movements on the finite element mesh.  $\Omega$  denotes the original mesh and  $\tilde{\Omega}$  represents the moved mesh. The objective function  $J(\cdot)$  depends on the deformed mesh  $\tilde{\Omega}$  and the solution of Maxwell’s equations  $A_z$ —the z-component of magnetic potential vector  $A$ .

$$\begin{aligned} & \text{minimize} && J(A_z, \tilde{\Omega}) \\ & \text{with respect to} && \mathbf{x} \quad \text{— design variables} \\ & \text{such that} && S(\mathbf{x}) = \Omega_{bc} \quad \text{— shape parametrization} \\ & && M(\tilde{\Omega}, \Omega, \Omega_{bc}) = \mathbf{0} \quad \text{— mesh movement} \end{aligned}$$

**Fig. 8** Topology optimization of a cantilever beam under linear elasticity with a  $80 \times 40$  mesh. **a** The problem setting, **b** the optimized density distribution  $\tilde{\rho}$ , and **c** convergence history of the objective function and the optimality



$$\begin{aligned}
 R(A_z, \tilde{\Omega}) &= \mathbf{0} && \text{— Maxwell's equations} \\
 \mathbf{x}_{\min} &\leq \mathbf{x} \leq \mathbf{x}_{\max}. && (11)
 \end{aligned}$$

The problem is governed by two nonlinear PDEs: one for mesh movement (hyperelasticity) and another for electromagnetic modeling (Maxwell's equations). These PDEs are coupled in a way that the solution of the former—the nodal deformation—is incorporated in the PDE residual of the latter. Hence the nodal deformation can have an impact on the final solution of the magnetic flux density. The corresponding MDO process is abstracted in Fig. 9.

### 4.3.2 Shape parametrization with free-form deformation

We start with an initial configuration of the motor geometry as shown in Fig. 10a. We use the vector of design variables  $\mathbf{x}$  to describe the changes in the geometric parameters of the motor, such as the inner radius of the rotor and the positions of the magnets.

To incorporate these changes into the physical domain, we employ free-form deformation (FFD), which is a technique commonly used for modeling the deformation of rigid objects and is analogous to parametrization with B-splines (Kenway et al. 2010; Sederberg and Parry 1986). FFD allows us to map the geometric changes  $\mathbf{x}$  to the movements of a specific

group of boundary-defining points in the physical domain. These points are denoted as  $\Omega_{bc}$ .

It is important to note that this problem involves numerous modeling specifications and steps in the preprocessing and postprocessing stages. A comprehensive discussion on the FFD method can be found in Scotzniovsky et al. (2024), which uses FEMO for geometric design of electric motor. However, for the purpose of this paper, these details are omitted, and the focus is primarily on the PDE-related aspects of the formulation.

### 4.3.3 Automated mesh movement

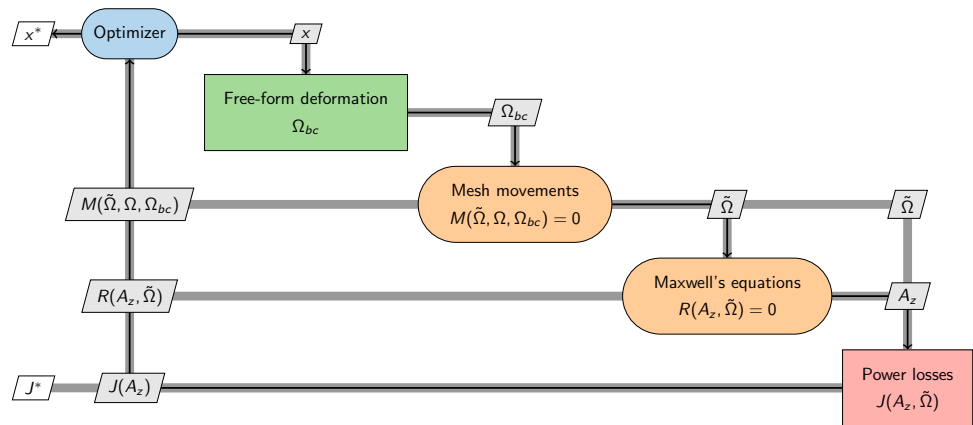
For the sake of notation convenience, we introduce  $\mathbf{u}_{bc}$  to represent the boundary movement  $\Omega_{bc}$  on the finite element domain. Additionally, we denote  $\mathbf{u}$  as the displacements for all the mesh vertices in the entire domain:

$$\mathbf{u}(\mathbf{X}) = \phi(\mathbf{X}) - \mathbf{X}, \tag{12}$$

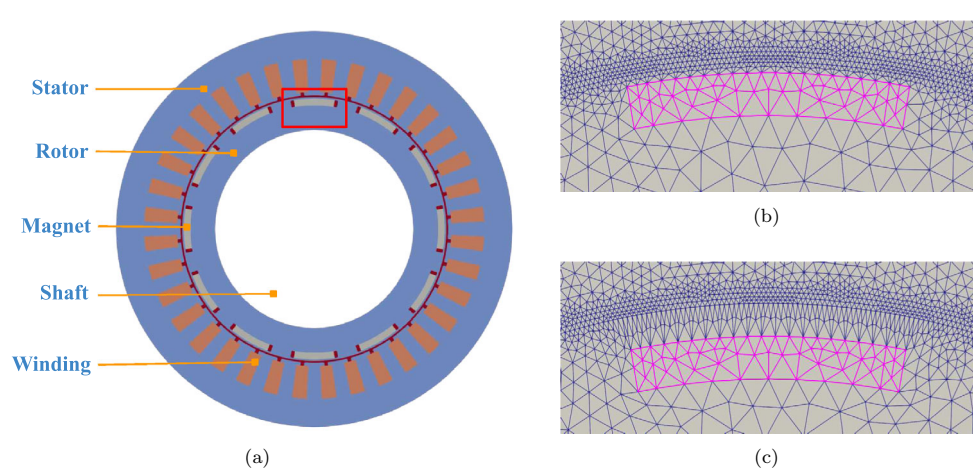
where  $\phi: \mathbf{X} \rightarrow \tilde{\mathbf{X}}$  represents the mapping from the original domain  $\Omega$  to the deformed domain  $\tilde{\Omega}$ . This approach for updating mesh based on the mesh movement is analogous to the approach used in fluid–structure interaction applications (Neighbor et al. 2023).



**Fig. 9** The MDO process for shape optimization of an electric motor



**Fig. 10** **a** The baseline geometry of the permanent magnet synchronous motor model. The red box highlights 1 of the 12 magnets. **b** The zoomed view of the red-boxed region with the undeformed mesh. The magnet is highlighted in fuchsia color. **c** The zoomed view of the red-boxed region with the deformed mesh. The magnet highlighted in fuchsia has been moved inwards along the radial direction



To simulate the mesh movement, we formulate a BVP where the displacements of the boundary vertices  $\mathbf{u}_{bc}$  are imposed as Dirichlet boundary conditions. The BVP is governed by a hyperelastic equation. By solving this BVP, we obtain the smooth movements  $\mathbf{u}$  throughout the entire domain, including both the on-boundary and off-boundary vertices. The strong form of the hyperelasticity problem is as follows

$$-\nabla \cdot \mathbf{P}(\mathbf{u}) = \mathbf{f} \quad \text{in } \Omega, \quad (13)$$

$$\mathbf{u} = \mathbf{u}_{bc} \quad \text{on } \Gamma_{bc},$$

where  $\mathbf{f}$  represents the body force, which is assumed to be zero in our case. The first Piola–Kirchhoff stress  $\mathbf{P}$  is given by

$$\mathbf{P} := J\sigma \mathbf{F}^{-T}, \quad (14)$$

where  $\sigma$  represents the Cauchy stress,  $\mathbf{F}$  is the deformation gradient, and  $J$  is its determinant.  $\mathbf{F}$  can be computed by

$$\mathbf{F} = \frac{\partial \phi}{\partial \mathbf{X}} = \nabla \mathbf{u} + \mathbf{I}, \quad (15)$$

which can be conveniently implemented in UFL as follows:

```
def F(u):
    return ufl.grad(u) + ufl.Identity(2) # 2D problem
```

For the material model, we adopt the modified St. Venant–Kirchhoff constitutive model, where the second Piola–Kirchhoff stress is defined as

$$\mathbf{S}(\mathbf{F}) = 2\mu \left[ \mathbf{E} - \frac{\text{tr}(\mathbf{E})}{3} \mathbf{I} \right] + K \text{tr}(\mathbf{E}) \mathbf{I}. \quad (16)$$

Here,  $\mathbf{E} = \frac{1}{2}(\mathbf{F}^T \mathbf{F} - \mathbf{I})$  is the Green–Lagrange strain.  $\mu$  and  $K$  are the material parameters that can be chosen based on the deformation to introduce artificial stiffness in regions of the mesh that experience crushing. In our numerical tests, we set  $\mu = K = J^{-p}$ , where  $p$  is a positive integer (set to 3).

Then the first Piola–Kirchhoff stress can be computed by

$$\mathbf{P}(\mathbf{F}) = \mathbf{F} \cdot \mathbf{S}(\mathbf{F}). \quad (17)$$

In the finite element formulation, the boundary condition is weakly enforced to the weak form of (13) using the symmetric Nitsche’s method applied on the subdomains. To



specify the relevant facets for the boundary condition, we define an integral operator using UFL, such as

```
dss = ufl.Measure('ds', domain=
    mesh, subdomain_data=
    boundary_data)
```

Here,  $ds$  represents the integral operator on facets in the finite element domain that are not on the outer boundary, and  $dss$  is a user-specified integral operator on the selected facets based on the provided boundary data, which contains the indices of the desired facet entities. The group of those facets corresponds to  $\Gamma_{bc}$  in (13), which is a subset of the total boundary  $\Gamma$ .

Figure 10 illustrates how the mesh deforms when moving a magnet inwards along the radial direction. Figure 10b represents the undeformed mesh. Nonzero displacements are assigned to the vertices located on the outer edges of the magnet, while the vertices on the unmoved boundaries, such as the air gap (the finely discretized area above the magnet), are restrained to remain unmoved. Figure 10c shows the deformed mesh obtained by warping the domain using the solution of the hyperelastic problem. The deformed mesh captures the movements of the vertices both inside and surrounding the magnet, resulting in a smooth mesh of the new configuration.

We note that the mesh is not actually moved during the optimization process. Instead, the mesh movement is incorporated into Maxwell’s equations, allowing us to compute the electromagnetic solution as if it were on the deformed mesh. This coupling method will be further explained in the next subsection.

An important advantage of this approach is that the number of vertices and their connectivity relationship remain constant throughout the optimization process. By computing the displacements of the vertices and incorporating them into the subsequent steps, we eliminate the need to regenerate or move the mesh, resulting in reduced computational cost.

To enhance the robustness of this algorithm for handling large deformation during the optimization process, we employ a customized nonlinear solver with load-stepping to incrementally apply the subdomain movements. The number of steps is dynamically determined based on the mesh size and the maximum expected deformation. This adaptive approach helps prevent inverted elements, which can cause the subsequent electromagnetic solver to fail.

### 4.3.4 Magnetostatic problem on the deformed mesh

Maxwell’s equations are

$$\begin{aligned} \nabla \times \mathbf{H} &= \mathbf{J}_w + \mathbf{J}_m, \\ \nabla \cdot \mathbf{B} &= 0, \end{aligned} \tag{18}$$

where  $\mathbf{J}_w$  and  $\mathbf{J}_m$  are source terms representing the effective current densities within the stator windings and the magnets, respectively.  $\mathbf{B}$  represents the magnetic flux density, and  $\mathbf{H}$  represents the magnetizing field.

The relationships between  $\mathbf{B}$ ,  $\mathbf{H}$ , and  $\mathbf{A}$  (the magnetic vector potential) are given by

$$\begin{aligned} \mathbf{B} &= \mu \mathbf{H}, \\ \mathbf{B} &= \nabla \times \mathbf{A}, \end{aligned} \tag{19}$$

where  $\mu$  is the magnetic permeability, which is a nonlinear function of  $B$  and depends on the local material properties.

Rewriting Maxwell’s equations in (18) by using  $\mathbf{A}$  as the solution gives us

$$\nabla \times (\mu^{-1} \nabla \times \mathbf{A}) = \mathbf{J}_w + \mathbf{J}_m. \tag{20}$$

This can be reduced by considering only the z-component of  $\mathbf{A}$  as the other two components are both zero

$$\begin{aligned} -\nabla \cdot (\mu^{-1} \nabla A_z) &= \mathbf{J}_z \text{ in } \Omega, \\ A_z &= g \text{ on } \Gamma. \end{aligned} \tag{21}$$

We choose  $g = 0$  for a homogeneous boundary condition of  $A_z$ . Then,  $\mathbf{B}$  can be obtained from  $A_z$  in the postprocessing steps by

$$\mathbf{B}(x, y) = \left( \frac{\partial A_z}{\partial y}, -\frac{\partial A_z}{\partial x} \right). \tag{22}$$

The weak form of (21) can be formulated as

$$\int_{\Omega} \mu^{-1} \nabla A_z \cdot \nabla v \, d\Omega = \int_{\Omega} J_z v \, d\Omega, \tag{23}$$

where  $v$  is a test function of  $A_z$ . By moving the right-hand side term to the left and applying the Dirichlet boundary condition with Nitsche’s method, we obtain the PDE residual in variational form on the undeformed mesh:

$$\begin{aligned} R(A_z, \Omega) &= \int_{\Omega} \mu^{-1} \nabla A_z \cdot \nabla v \, d\Omega - \int_{\Omega} J_z v \, d\Omega \\ &\quad - \int_{\Gamma} \mu^{-1} (\nabla A_z \cdot \mathbf{n} \cdot v \mp \nabla v \cdot \mathbf{n} \cdot (A_z - g)) \, d\Gamma \\ &\quad + \int_{\Gamma} \frac{C_{pen}}{h} \mu^{-1} v (A_z - g) \, d\Gamma. \end{aligned} \tag{24}$$

To obtain the variational form on the deformed mesh, we adopt a seamless coupling method for hyperelasticity and electromagnetism by incorporating  $\mathbf{u}$  into the weak form in (24). This is inspired by the body-fitted method for Fluid–Structure Interaction (FSI) problems (Bazilevs et al. 2008),

where the integral and differential operators are modified by Nanson's formula into the weak forms. For example,

$$\begin{aligned} d\Omega &\rightarrow Jd\Omega, \\ nd\Gamma &\rightarrow J\mathbf{F}^{-T}nd\Gamma, \\ d\Gamma &\rightarrow |J\mathbf{F}^{-T}\mathbf{n}|d\Gamma, \\ \nabla v &\rightarrow \nabla v \cdot \mathbf{F}^{-1} \\ &\dots \end{aligned} \quad (25)$$

Here,  $J$  represents the determinant of  $\mathbf{F}$ , which is computed by  $J = \det(\mathbf{F})$ . These modifications can be easily handled by UFL, i.e.,  $\nabla v \cdot \mathbf{F}^{-1}$  can be conveniently implemented as

```
def gradx(f, u):
    return ufl.dot(ufl.grad(f),
                  ufl.inv(F(u)))
```

where  $F(u)$  is the predefined function as described in Sect. 4.3.3.

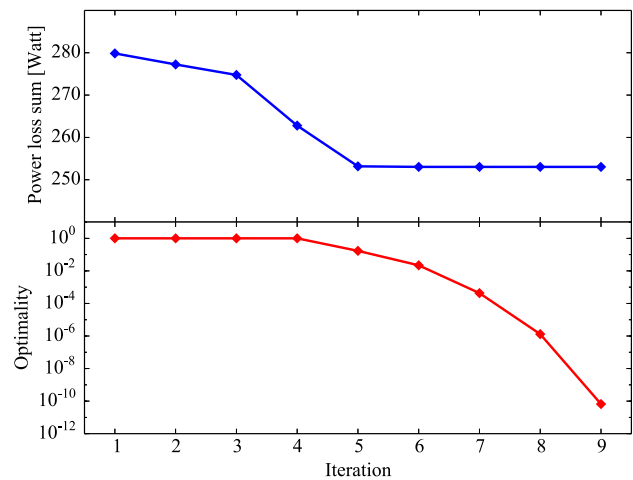
#### 4.3.5 Numerical results

For the numerical tests, we set up a simplified optimization problem with a single design variable—the change of magnet position in the radial direction. The objective is to minimize the natural core losses due to the magnets during no-load operation. In the simplified setting, the total core loss  $P_L$  is given by the sum of the eddy current loss  $P_{ec}$  and the hysteresis loss  $P_h$ , i.e.,  $P_L = P_{ec} + P_h$ . The losses are computed by

$$\begin{aligned} P_{ec} &= 2\pi^2 f^2 l K_{ec} \int_{\Omega_{ec}} |\mathbf{B}|^2 d\Omega \quad \text{and} \\ P_h &= 2\pi f l K_h \int_{\Omega_h} |\mathbf{B}|^\beta d\Omega, \end{aligned} \quad (26)$$

where  $f$  is the frequency of the input signal in Hertz,  $l$  is the motor length, and they are both set to unit values.  $K_{ec}$  is the eddy current coefficient, and  $K_h$  is the Steinmetz hysteresis coefficient.  $\Omega_{ec}$  and  $\Omega_h$  represent the subdomains where the power loss effects are observed (typically the rotor and the stator).  $\beta$  is the Steinmetz exponent that depends on material, ranging from 1.5 to 2.5.  $\beta$  is set to 1.76835 for the chosen Hiperco 50 the alloy material, which is the material used for the rotor and the stator. Again, we need to convert the integral operators in (26) to the deformed mesh using the substitutions given in (25).

As shown in Fig. 11, this single-variable single-objective optimization converges successfully. The optimality drops below  $10^{-8}$  within 9 iterations. The problem is always feasible throughout the optimization process as the linear constraints, i.e., the bounds on the design variable, are always



**Fig. 11** Convergence history for the shape optimization of the electric motor

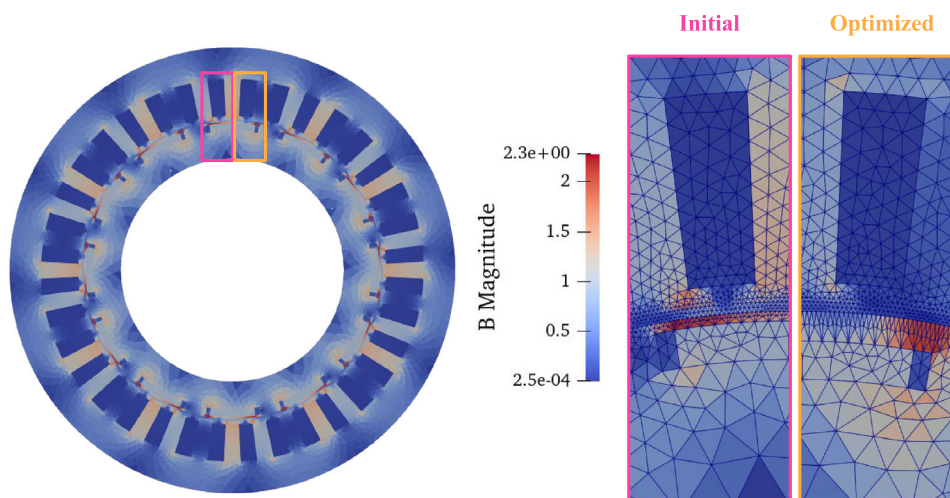
satisfied. The total runtime for this problem is 571 s, involving the evaluation of the coupled PDE system on a mesh with 28,410 elements as well as the adjoint computation.

The magnitude of the magnetic flux density  $\mathbf{B}$  is plotted on both sides of the motor in Fig. 12. The left side represents the magnetic flux solution field on the undeformed mesh, while the right half shows the results obtained on the deformed mesh. Although it is not obvious to notice the difference of the two sides as they use the same color scale, it is worth mentioning that the maximum magnitude of  $\mathbf{B}$  on the right is 2.1, which is 8.7% smaller than the maximum value of 2.3 on the left as in the color legend. Also, in the optimized configuration, a lighter color of  $\mathbf{B}$  magnitude can be observed in the stator area between the stator teeth. This indicates a reduction in the power loss in this region, leading to improved performance. The decrease in peak magnetic flux density can be explained by the enlarged area of the rotor between the magnets and the air gap. The movement of the magnets increases the area for flux to permeate through this tight zone, which decreases the effective flux density. The movement of the magnets also decreases the flux that permeates from the rotor to the stator, reducing the flux density magnitudes in the stator teeth.

#### 4.4 Aeroelastic analysis on an eVTOL wing

This application employs aeroelastic coupling of a high-fidelity structural solver based on Reissner–Mindlin shell theory and a low-fidelity aerodynamics solver: the vortex lattice method (VLM). A solver-independent method (Van Schie et al. 2023) is used for coupling these two solvers. Numerical tests include a static aeroelastic analysis with analytical derivative computation and a dynamic case for gust response on an eVTOL wing. We use the Uber eCRM-001

**Fig. 12** The magnetic flux density on the initial configuration (pink) and the optimized configuration (orange). (Color figure online)



model in the following test cases, which is an electric common reference model (eCRM) for electric air taxi.

#### 4.4.1 Aerodynamic solver with vortex lattice method

VLM is a widely used approach for analyzing aerodynamic performance, such as force or pressure distribution on lifting surfaces (Anderson 2010). VLM makes a potential flow assumption, which means the flow can be characterized by an irrotational velocity field given by

$$\nabla^2 \Phi = 0, \tag{27}$$

where  $\Phi$  is the fluid potential.

Other assumptions of VLM include that (1) the fluid is incompressible; and (2) the lifting surfaces are thin and have a small angle of attack and sideslip.

The VLM model is built in CSDL to enable automated derivative computation. It is implemented in an inertial frame with a constant freestream velocity and assumes steady-state conditions during each mission segment. Figure 13 demonstrates the inputs, model, and outputs of the VLM model. The inputs of the VLM model are the meshes and the states of the lifting surfaces. Then, the model creates structured quadrilateral panels and uses vortex rings located at the quarter-chord of each panel for computation. The system of equations to be solved in (28) corresponds to the flow-tangency condition meaning zero normal velocity on the lifting surface panels.

$$\mathbf{A}\Gamma_b + \mathbf{B}\Gamma_w + \mathbf{v} \cdot \mathbf{n} = 0, \tag{28}$$

where  $\mathbf{v}$  is the freestream velocity, and  $\mathbf{n}$  is the normal vector of the lifting surface panels.  $\Gamma_b$  and  $\Gamma_w$  represent the bound and the wake vortex circulation strengths, respectively.  $\mathbf{A}$  and  $\mathbf{B}$  are the aerodynamic influence coefficient matrices.

The outputs of the VLM model are the traction forces on each panel given by the panel forces divided by the areas. The panel forces are calculated using the Kutta–Joukowski theorem (Katz and Plotkin 2001) as below

$$\mathbf{F} = \rho \Gamma_b \mathbf{v}_{ind} \times \mathbf{s}, \tag{29}$$

where  $\rho$  is the density of air,  $\mathbf{v}_{ind}$  is the induced velocity evaluated at the bound vortex collocation points, and  $\mathbf{s}$  represents the bound vector.

#### 4.4.2 Thin-shell structural solver with Reissner–Mindlin plate theory

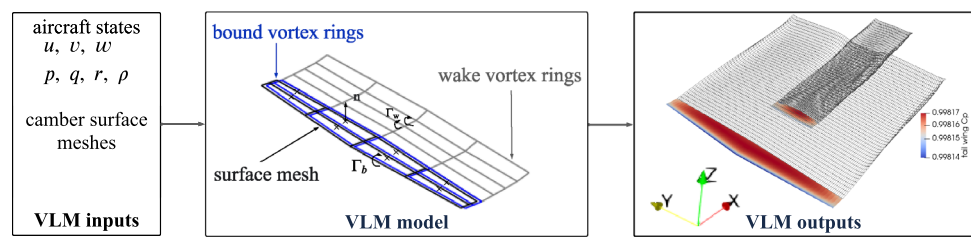
We implement the Reissner–Mindlin shell formulation in FEniCSx,<sup>3</sup> based on the geometrically linear, quadrilateral-element variant of the fully nonlinear unconventional shell formulation developed by Campello et al. (2003). This formulation considers six-parameter solution fields (three for displacements and three for rotations).

There are other openly available implementations of this shell formulation in the FEniCS community, such as the pedagogical example by Bleyer (2018), with slight difference in the formulation of the drilling stabilization term. Moreover, the existing implementation is implemented in legacy FEniCS with limited support for quadrilateral elements, while our shell solver is implemented in FEniCSx, which provides robust support for quadrilateral elements.

While our shell solver can deal with both triangular and quadrilateral elements, it currently cannot handle meshes containing multiple types of elements, i.e., meshes con-

<sup>3</sup> The source code of this FEniCSx-based shell solver is available on Github at [https://github.com/RuruX/shell\\_analysis\\_fenicsx](https://github.com/RuruX/shell_analysis_fenicsx) with numerous examples and the associated mesh files.

**Fig. 13** VLM workflow for aerodynamic analysis



taining both triangular and quadrilateral elements, due to a limitation in the FEniCSx mesh importing module.

#### 4.4.3 Work-conservative aeroelastic coupling

The VLM aerodynamic solver is loosely coupled with the FEniCSx structural solver by linearly mapping the aerodynamic loads and structural displacements between both solvers. We use the approach presented in Van Schie et al. (2023) to construct these linear load and displacement maps. In this approach, we start by expressing the aeroelastic work with each solver. This is achieved by defining a matrix  $M$  for each solver, such that the aeroelastic work can be computed as the product  $\mathbf{u}^T M \mathbf{F}$ . Here,  $\mathbf{u}$  and  $\mathbf{F}$  are the vectors containing the degrees of freedom of the displacements and the loads, respectively. The product  $\mathbf{u}^T M \mathbf{F}$  encodes all solver-specific information used in computing aeroelastic work.

Next, we define a linear map to transfer the displacements from the structural solver to the VLM solver. Then, we construct the linear map that transfers aerodynamic loads from the VLM solver to the structural solver following the expressions of aeroelastic work and the displacement map. The resulting load map ensures that the load-displacement mapping pair conserves aeroelastic work and can also be designed to conserve the total aerodynamic load in each principal direction. In this application, both the displacement map and the load map are linear, which allows them to be used in gradient-based optimization algorithms.

#### 4.4.4 Static aeroelastic simulation with constant wind velocity

We start with a simple static case of aeroelasticity, where a constant wind velocity  $V_\infty = 50$  m/s is applied. We consider an angle of attack  $\text{AoA} = 6^\circ$ , and the air density is set to the International Standard Atmosphere air density at sea level,  $\rho = 1.225$  kg/m<sup>3</sup>. The aerodynamic solver initiates with an original VLM mesh and outputs the aerodynamic loads, which are then projected onto the structural domain. The structural solver computes the displacements, which in turn lead to updates in the VLM mesh. This iterative process continues until the updates in the VLM mesh coordinates between the previous and the current iterations fall below  $10^{-6}$ . We use an embedded nonlinear block CG solver for

this iterative coupling relationship. The results of the von Mises stress and the aerodynamic load are shown in Fig. 14.

#### 4.4.5 Dynamic aeroelastic simulation of wind gust response

For the dynamic scenario, we study the response of the eVTOL wing subjected to gust excitation. The eVTOL aircraft is initially cruising at a constant speed of  $v_0 = 50$  m/s under steady wind in the  $x$  direction,  $V_\infty = 50$  m/s. Then, we apply a gust wind in the  $z$ -direction with a sinusoidal profile given by  $V_{\text{gust}} = v_0 \times (1 - \cos(2\pi(t - T_0)/T_1))$ , where  $T_0$  is the time when the gust is activated and  $T_1$  is the duration of the gust application.

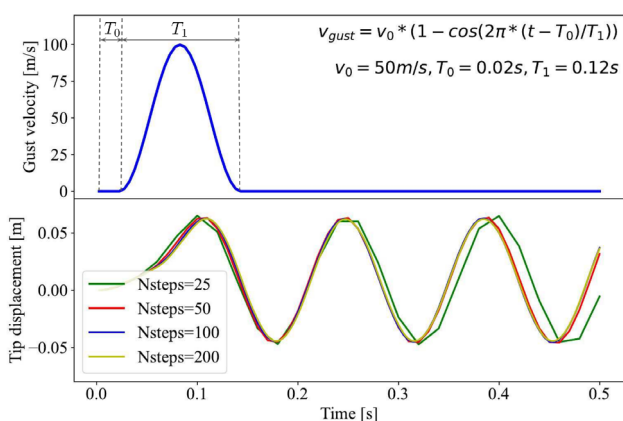
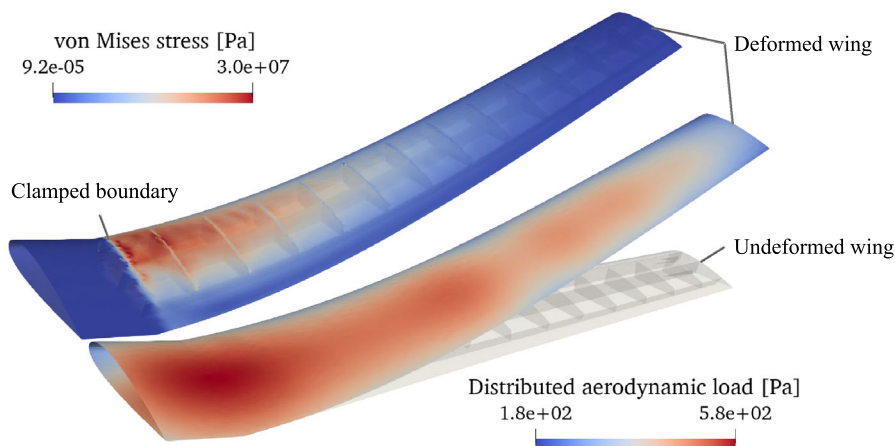
We simulate the aeroelastic response of the wing under this gust excitation, considering three phases: the steady state, the gust application period, and the oscillation after gust removal. We choose the implicit midpoint rule in FEniCSx as the time integration method for the dynamic structural solver, and the steady VLM solver. In Fig. 15, the gust velocity and the tip displacement of the eVTOL wing are shown over time. The peak tip displacement occurs slightly after the maximum gust velocity. This aligns with our expectation since the velocity of wing tip is not zero yet at the maximum gust velocity, leading to a continued upward movement of the wing until the velocity of the wing itself decreases to zero. The amplitude of the tip displacement is constant during the oscillation phase because there is no damping in the elastic model. In the plot of tip displacements, a convergence study of time step size is performed and the solution converges when  $\Delta t = 0.001$  (Nsteps=50).

#### 4.4.6 Sensitivity analysis and validation of compliance with respect to thickness

As the aeroelastic solver is fully integrated with FEMO, it allows us to compute the total derivatives of any specified outputs with respect to design variables automatically. The MDO process is shown in Fig. 16. These automated derivatives are the key to solve any optimization problems involving aeroelasticity efficiently. In Fig. 17, we demonstrate the capability of automatic adjoint computation for aeroelasticity by performing sensitivity analysis of structural compliance with respect to thickness distribution in a static scenario. We represent the thickness  $t$  using a first-order continuous polynomial



**Fig. 14** Static aeroelastic analysis of the eCRM-001 wing model with constant wind velocity and clamped boundary condition at the wing root. The deformation of the wing structure is scaled by a factor of 40 for visualization



**Fig. 15** Numerical results for wind gust response of the eCRM-001 wing model under a sinusoidal gust excitement. The plot on top shows the velocity of the gust over time. The plot on the bottom shows the corresponding aeroelastic response of the wing measured by the tip displacement with different time step sizes

function, which is equivalent to nodal thickness. A constant thickness of 0.03 m is assigned as the initial value. The compliance  $C$  in this problem is defined as the dot product of the force vector  $f$  and the displacement vector  $u$  (Christensen

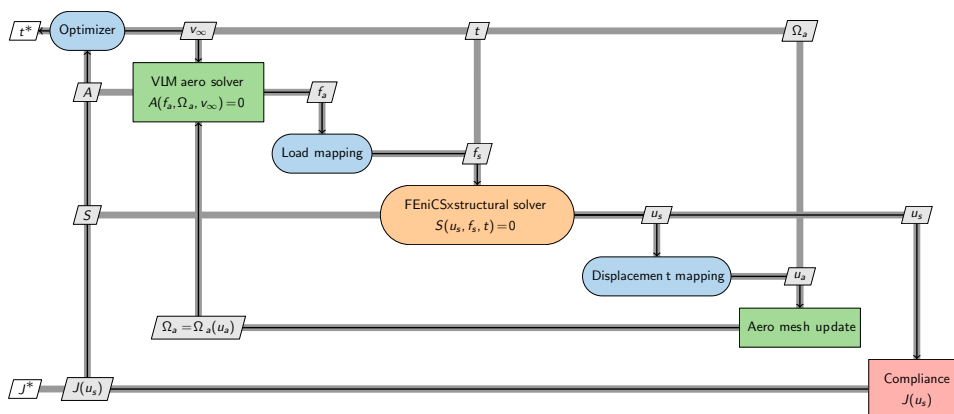
and Klarbring 2008),

$$C = f^T u, \tag{30}$$

which estimates the inverse of the stiffness in the structure model. The runscript for sensitivity analysis is shown in Listing 7, “Appendix B,” where we also provide a comprehensive comparison of estimated implementation efforts involved in utilizing FEMO versus employing Nastran. It is worth mentioning that while compliance and thickness are structural parameters, the aerodynamic solver is also involved in this derivative computation. This is because the aerodynamic mesh will be affected by structural deformation, leading to changes in the aerodynamic loads, which in turn impact the compliance, computed from the displacements.

We validate the gradients computed through FEMO’s automatic adjoint method by comparing them with finite difference results at selected nodes. Specifically, we use the first-order forward difference approximation for validation. Thirteen nodes along the span-wise direction of the wing skin are chosen for this comparison, with their locations shown in Fig. 17. Given the gradients’ varying magnitudes ranging from  $10^{-4}$  to  $10^{-1}$ , the choice of perturbation step size

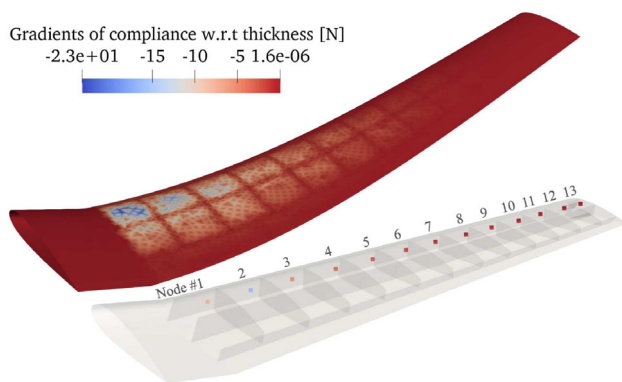
**Fig. 16** The MDO process for thickness sensitivity analysis of the eCRM-001 wing structure under aerodynamic loads



**Table 2** Adjoint validation at selected nodes by comparing the gradients computed, respectively, by FEMO and finite difference method for the two-way coupled aeroelastic model

Node	FEMO $dC/dt$	Finite difference $dC/dt$	Absolute error	Relative error  (%)
1	-7.547996	-7.482387	0.065609	0.87
2	-16.785037	-16.606966	0.178071	1.06
3	-5.036602	-4.988461	0.048140	0.96
4	-3.405216	-3.367557	0.037659	1.11
5	-2.525660	-2.500194	0.025466	1.01
6	-1.823823	-1.805237	0.018586	1.02
7	-0.883301	-0.873788	0.009513	1.08
8	-0.291701	-0.288946	0.002755	0.94
9	-0.599661	-0.593118	0.006543	1.09
10	-0.136978	-0.135707	0.001271	0.93
11	-0.022173	-0.022037	0.000136	0.61
12	-0.005538	-0.005501	0.000038	0.68
13	-0.000846	-0.000835	0.000011	1.31

We consider the gradients of compliance with respect to thickness in this case, denoted as  $dC/dt$ . A perturbation step size of  $10^{-4}$  is used for the finite difference method



**Fig. 17** The gradients of compliance with respect to the thickness distribution are represented as a first-order continuous function in the top contour plot. The locations of the nodes used for adjoint validation are shown in the bottom plot

$h$  can significantly affect the finite difference results. After conducting several numerical experiments with different step sizes, we opt for a step size of  $10^{-4}$  to achieve a good balance of accuracy and stability. Detailed results are provided in Fig. 18, “Appendix C.” The gradients at the selected nodes

are summarized in Table 2, demonstrating good agreement between the two approaches, with a maximum relative error of 1.31%.

## 5 Conclusion

In this work, we have presented a generalized approach for modeling PDE-constrained MDO problems, and developed the computational tool FEMO, with the goal to minimize the time, effort, and expertise required for development. FEMO leverages the powerful finite element solver FEniCSx to effectively and accurately solve PDEs, and integrates with CSDL, a domain-specific language for MDO modeling, to enable coupling with non-PDE components and automatic adjoint sensitivity analysis.

The effectiveness of this approach has been demonstrated through verification using benchmark problems, including the optimal control of a nonlinear elliptic PDE and topology optimization of a cantilever beam, both of which have known analytical solutions. Additionally, its capability to tackle real-world problems, such as the design and optimization of electric vertical takeoff and landing (eVTOL) aircraft, which motivated this research, has been established.



The shape optimization of an electric motor has converged successfully, which couples two nonlinear PDEs—the hyperelastic equation and Maxwell's equations—to simulate the magnetic flux density field under geometry deformation, and then coupled with non-PDE postprocessing model to compute the power loss as the objective function.

We applied this method to large-scale aeroelastic analysis of an eVTOL wing, where there are thousands of design variables corresponding to node-wise structural thickness. Here, the FEniCSx-based structural solver is loosely coupled with the CSDL-based VLM solver through FEMO, and both static and dynamic cases have been discussed. For the static case, we also performed sensitivity analysis for structural compliance of the wing with respect to thickness distribution, which provided qualitatively reasonable results for the gradients. In the dynamic case, the structural response of the wing under gust excitation was studied, and the wing tip displacement profile achieved convergence through the reduction of the time step size.

While the presented methodology and implementation have shown promising results, there are still some challenges to overcome. Expertise is required for applying case-dependent Dirichlet boundary conditions weakly with Nitsche's method, including formulation, tuning, and testing. Additionally, the efficiency needs to be improved for large-scale problems with a large number of design variables. Potential solutions could be reduced-order modeling, parallel computing, or tuning for nonlinear/linear solvers. For future work, we plan to address the above-mentioned challenges, and extend FEMO to include additional features and applications, such as sensitivity analysis for dynamic aeroelastic models. Another goal could be to incorporate other finite element modules based on the legacy FEniCS to fulfill spe-

cific analysis requirements, such as using PENGoLINS for isogeometric analysis with non-matching shells (Zhao et al. 2022).

## Appendix A: Code comparison of FEMO and FEniCS for topology optimization problem

We utilize the topology optimization problem described in Sect. 4.2 to compare FEMO code<sup>4</sup> with pure FEniCS code.<sup>5</sup> To elucidate the structures of the two platforms, we have streamlined both scripts by removing secondary lines, including import statements, unnecessary comments, and postprocessing. In addition, the `GeneralFilterModel` class in FEMO code is also omitted, as it primarily consists of simple CSDL-based algebraic operations. Even in this simple example, several benefits of using FEMO become apparent:

- (1) FEMO naturally separates the finite element analysis and optimization code through a modular design, facilitating easy debugging and modification. For instance, in this case, only Lines 115–117 in the FEMO code need to be changed if a different density filter is desired.
- (2) The FEMO code incorporates fewer FEniCS implementations (Lines 12–50) compared to pure FEniCS (throughout the entire script). This is attributed to the automation of FEniCS utility functions, such as variable updates in the backend of FEMO.
- (3) The remaining sections of the FEMO code are straightforward to implement, mainly involving filling in the blank spaces of parameters in the dictionary object, such as in `fea.add_input()`, `fea.add_state()` and `fea.add_output()` (Lines 81–93).

<sup>4</sup> Full FEMO code: [https://github.com/RuruX/femo/blob/main/examples/beam\\_topo\\_opt/run\\_topo\\_opt\\_cantilever\\_beam.py](https://github.com/RuruX/femo/blob/main/examples/beam_topo_opt/run_topo_opt_cantilever_beam.py).

<sup>5</sup> Full FEniCS code: [https://comet-fenics.readthedocs.io/en/latest/demo/topology\\_optimization/simp\\_topology\\_optimization.html](https://comet-fenics.readthedocs.io/en/latest/demo/topology_optimization/simp_topology_optimization.html).

## FEMO code

```

1 num_el_x = 80
2 num_el_y = 40
3 LENGTH_X = 160.
4 LENGTH_Y = 80.
5 mesh = createRectangleMesh(np.array([0.0,0.0]),
6                             np.array([LENGTH_X, LENGTH_Y]),
7                             num_el_x,
8                             num_el_y)
9 #####
10 ##### FEA code #####
11 #####
12 DOLFIN_EPS = 3E-16
13 def TractionBoundary(x):
14     return np.logical_and(
15         abs(x[1] - LENGTH_Y/2) < LENGTH_Y/num_el_y \
16             + DOLFIN_EPS*1e10,
17         abs(x[0] - LENGTH_X) < DOLFIN_EPS*1e10)
18 fdim = mesh.topology.dim - 1
19 traction_facets = locate_entities_boundary(mesh, fdim,
20                                         TractionBoundary)
21 facet_tag = meshtags(mesh, fdim, traction_facets,
22                     np.full(len(traction_facets),
23                             100, dtype=np.int32))
24 metadata = {"quadrature_degree":4}
25 ds_ = ufl.Measure('ds',
26                 domain=mesh,
27                 subdomain_data=facet_tag,
28                 metadata=metadata)
29 def pdeRes(u, v, rho_e, f, E = 1, dss = ds, method='SIMP'):
30     if method == 'SIMP':
31         C = rho_e**3
32     else:
33         C = rho_e/(1 + 8. * (1. - rho_e))
34     E = 1. * C # C is the design variable from 0 to 1
35     nu = 0.3 # Poisson's ratio
36     lambda_ = E * nu/(1. + nu)/(1 - 2 * nu)
37     mu = E / 2 / (1 + nu) # lame's parameters
38     w_ij = 0.5 * (grad(u) + grad(u).T)
39     v_ij = 0.5 * (grad(v) + grad(v).T)
40     d = len(u)
41     sigm = lambda_*div(u)*Identity(d) + 2*mu*w_ij
42     res = inner(sigm, v_ij) * dx - dot(f, v) * dss
43     return res
44 def averageFunc(func):
45     volume = assemble(Constant(mesh,1.0)*dx)
46     func1 = Function(func.function_space)
47     func1.vector.set(1/volume)
48     return inner(func, func1)*dx
49 def compliance(u, f, dss=ds):
50     return dot(u, f)*dss

```

## FEniCS code

```

niternp = 20 # number of non-penalized iterations
niter = 80 # total number of iterations
pmax = 4 # maximum SIMP exponent
# minimum number of steps between exponent update
exponent_update_frequency = 4
# tolerance on mass when finding Lagrange multiplier
tol_mass = 1e-4
thetamin = 0.001 # minimum density modeling void
# Problem parameters
thetamoy = 0.4 # target average material density
E = Constant(1)
nu = Constant(0.3)
lamda = E*nu/(1+nu)/(1-2*nu)
mu = E/(2*(1+nu))
f = Constant((0, -1)) # vertical downwards force
# Mesh
mesh = RectangleMesh(Point(-2, 0), Point(2, 1),
                    50, 30, "crossed")
# Boundaries
def left(x, on_boundary):
    return near(x[0], -2) and on_boundary
def load(x, on_boundary):
    return near(x[0], 2) and near(x[1], 0.5, 0.05)
facets = MeshFunction("size_t", mesh, 1)
AutoSubDomain(load).mark(facets, 1)
ds = Measure("ds", subdomain_data=facets)
# Function space for density field
V0 = FunctionSpace(mesh, "DG", 0)
# Function space for displacement
V2 = VectorFunctionSpace(mesh, "CG", 2)
# Fixed boundary conditions
bc = DirichletBC(V2, Constant((0, 0)), left)
p = Constant(1) # SIMP penalty exponent
exponent_counter = 0 # exponent update counter
# Lagrange multiplier for volume constraint
lagrange = Constant(1)
thetaold = Function(V0, name="Density")
thetaold.interpolate(Constant(thetamoy))
coeff = thetaold*p
theta = Function(V0)
volume = assemble(Constant(1.)*dx(domain=mesh))
# initial average density
avg_density_0 = assemble(thetaold*dx)/volume
avg_density = 0.
# We now define some useful functions for formulating
# the linear elastic variational problem.
def eps(v):
    return sym(grad(v))
def sigma(v):
    return coeff*(lamda*div(v)*Identity(2)+2*mu*eps(v))

```

## FEMO code

```

51 #####
52 ##### FEMO code #####
53 #####
54 fea = FEA(mesh)
55 # Add input to the PDE problem:
56 input_name = 'density'
57 input_function_space = FunctionSpace(mesh, ('DG', 0))
58 input_function = Function(input_function_space)
59 gradient_function = Function(input_function_space)
60 # Add state to the PDE problem:
61 state_name = 'displacements'
62 state_function_space = VectorFunctionSpace(mesh, ('CG', 1))
63 state_function = Function(state_function_space)
64 v = TestFunction(state_function_space)
65 method = 'SIMP'
66 f = Constant(mesh, (0, -1/4))
67 residual_form = pdeRes(state_function,
68                        v,
69                        input_function,
70                        f,
71                        dss=ds_(100),
72                        method=method)
73 # Add output to the PDE problem:
74 output_name_1 = 'avg_density'
75 output_form_1 = averageFunc(input_function)
76 output_name_2 = 'compliance'
77 output_form_2 = compliance(state_function,
78                            f,
79                            dss=ds_(100))
80 fea.record = True
81 fea.add_input(input_name, input_function)
82 fea.add_state(name=state_name,
83              function=state_function,
84              residual_form=residual_form,
85              arguments=[input_name])
86 fea.add_output(name=output_name_1,
87               type='scalar',
88               form=output_form_1,
89               arguments=[input_name])
90 fea.add_output(name=output_name_2,
91               type='scalar',
92               form=output_form_2,
93               arguments=[state_name])
94 ubc = Function(state_function_space)
95 ubc.vector.set(0.0)
96 locate_BC1 = locate_dofs_geometrical(
97     (state_function_space,
98     state_function_space),
99     lambda x: np.isclose(
100         x[0], 0., atol=1e-6))
101 locate_BC_list = [locate_BC1]
102 fea.add_strong_bc(ubc, locate_BC_list, state_function_space)
103 #####
104 ##### Optimization code #####
105 #####
106 fea_model = FEAModel(fea=[fea])
107 pre_processor_name = 'general_filter_model'
108 coords = input_function_space.tabulate_dof_coordinates()
109 tdim = mesh.topology.dim
110 num_cells = mesh.topology.index_map(tdim).size_local
111 h = dolfinx.cpp.mesh.h(mesh, tdim, range(num_cells))
112 h_avg = (h.max() + h.min())/2
113 nel = mesh.topology.index_map(mesh.topology.dim).size_local
114 # Case-to-case preprocessor model
115 pre_processor_model = GeneralFilterModel(nel=nel,
116                                         coordinates=coords,
117                                         h_avg=h_avg)
118 # Coupling is done as simple as model.add()
119 fea_model.add(pre_processor_model, name=pre_processor_name)
120 np.random.seed(0)
121 fea_model.create_input("{}".format('density_unfiltered'),
122                       shape=nel,
123                       val=np.random.random(nel) * 0.86)
124 fea_model.add_design_variable('density_unfiltered',
125                              upper=1.0,
126                              lower=1e-4)
127 fea_model.add_objective('compliance')
128 fea_model.add_constraint('avg_density', upper=0.40)
129 sim = Simulator(fea_model, analytics=True)
130 # Set up optimizer and run
131 sim.run()
132 prob = CSDLProblem(
133     problem_name='beam_topo_opt',
134     simulator=sim,
135 )
136 optimizer = SNOPT(prob,
137                  Major_iterations = 100000,
138                  Major_optimality = 1e-8,
139                  Major_feasibility=1e-6,
140                  append2file=True)
141 optimizer.solve()
142 #####
143 #
144 #
145 # Empty lines to match FEniCS code
146 #
147 #
148 #####

```

## FEniCS code

```

def energy_density(u, v):
    return inner(sigma(u), eps(v))
# Inhomogeneous elastic variational problem
u_ = TestFunction(V2)
du = TrialFunction(V2)
a = inner(sigma(u_), eps(du))*dx
L = dot(f, u_)*ds(1)
def local_project(v, V):
    dv = TrialFunction(V)
    v_ = TestFunction(V)
    a_proj = inner(dv, v_)*dx
    b_proj = inner(v, v_)*dx
    solver = LocalSolver(a_proj, b_proj)
    solver.factorize()
    u = Function(V)
    solver.solve_local_rhs(u)
    return u
def update_theta():
    theta.assign(local_project(
        (p*coeff*energy_density(u, u)
         /lagrange)**(1/(p+1)), V0))
    thetav = theta.vector().get_local()
    theta.vector().set_local(np.maximum(
        np.minimum(1, thetav), thetamin))
    theta.vector().apply("insert")
    avg_density = assemble(theta*dx)/volume
    return avg_density
# We now define a function for finding the correct value
# of the Lagrange multiplier $lambda$.
def update_lagrange_multiplier(avg_density):
    avg_density1 = avg_density
    # Initial bracketing of Lagrange multiplier
    if (avg_density1 < avg_density_0):
        lagmin = float(lagrange)
        while (avg_density < avg_density_0):
            lagrange.assign(Constant(lagrange/2))
            avg_density = update_theta()
        lagmax = float(lagrange)
    elif (avg_density1 > avg_density_0):
        lagmax = float(lagrange)
        while (avg_density > avg_density_0):
            lagrange.assign(Constant(lagrange*2))
            avg_density = update_theta()
        lagmin = float(lagrange)
    else:
        lagmin = float(lagrange)
        lagmax = float(lagrange)
    # Dichotomy on Lagrange multiplier
    inddico=0
    while ((abs(1.-avg_density/avg_density_0)) > tol_mass):
        lagrange.assign(Constant((lagmax+lagmin)/2))
        avg_density = update_theta()
        inddico += 1;
        if (avg_density < avg_density_0):
            lagmin = float(lagrange)
        else:
            lagmax = float(lagrange)
    print("Dichotomy iterations:", inddico)
# Finally, the exponent update strategy is implemented:
def update_exponent(exponent_counter):
    exponent_counter += 1
    if (i < niternp):
        p.assign(Constant(1))
    elif (i >= niternp):
        if i == niternp:
            print("\nStarting penalized iterations\n")
            if ((abs(compliance-old_compliance) \
                 < 0.01*compliance_history[0]) and
                (exponent_counter > exponent_update_frequency) ):
                # average gray level
                gray_level = assemble((theta-thetamin) \
                                       *(1.-theta)*dx)*4/volume
                p.assign(Constant(min(float(p) \
                                       *(1+0.3**((1.+gray_level/2))), pmax)))
                exponent_counter = 0
                print("Updated SIMP exponent_uPu=", float(p))
            return exponent_counter
u = Function(V2, name='Displacement')
old_compliance = 1e30
ffile = XDMFFile("topology_optimization.xdmf")
ffile.parameters["flush_output"]=True
ffile.parameters["functions_share_mesh"]=True
compliance_history = []
for i in range(niter):
    solve(a == L, u, bc, solver_parameters={
        "linear_solver": "cg",
        "preconditioner": "hybre_amg"})
ffile.write(thetaold, i)
ffile.write(u, i)
compliance = assemble(action(L, u))
compliance_history.append(compliance)
print("Iteration {}: compliance_u=".format(i), compliance)
avg_density = update_theta()
update_lagrange_multiplier(avg_density)
exponent_counter = update_exponent(exponent_counter)
# Update theta field and compliance
thetaold.assign(theta)
old_compliance = compliance

```

## Appendix B: Implementation effort comparison of FEMO and Nastran for aeroelasticity

We compare the implementation efforts involved in using FEMO and Nastran for aeroelasticity analysis. We present the full FEMO runsript in Listing 7 for sensitivity analysis of an eVTOL wing under static aeroelastic coupling. The FEniCS-based dolfin-adjoint library is excluded from our comparison due to its inability to interface with other non-FEniCS-based solvers, such as low-/mid-fidelity aerodynamic solvers.

Given that Nastran primarily serves as a finite element analysis solver for structural and thermal analysis, while FEMO offers a generic PDE solver platform coupled with an optimization environment, our comparison focuses on two main aspects: the shell solver integrated with FEMO versus Nastran, and the respective workflows of using them in aeroelasticity. It is worth noting that most source code using Nastran for high-fidelity aeroelastic coupling are either inaccessible to public or challenging to reproduce because of the lack of maintenance or license constraints (Kenway and Martins 2014; Benaouali and Kachel 2019). For this reason, discussions of implementation efforts are estimated based the authors' experience with these software tools and the literature.

The breakdown of implementation efforts is as follows:

- *Learning curve* we use the time required to master the shell solver as the learning curve estimation. Nastran, being a widely accepted finite element solver in aerospace and other engineering fields, typically has a lower learning curve for its pre-compiled commercial

software products. In contrast, FEMO's FEA capability relies on FEniCSx, necessitating the users to possess basic knowledge of FEniCSx and Python programming. Consequently, the learning curve for the FEMO shell solver is slightly steeper

- *Coupling* we specifically consider the integration of forward simulation for this aspect, as the coupling of derivatives is addressed separately in the next item. In contrast to FEMO's automated coupling capability as easy as `model.add(submodel)`, as shown in Lines 302–309 in Listing 7, Nastran demands significant effort as well as expertise in compiling multi-language code, given that the original solver is written in Fortran.<sup>6</sup> While tools like pyNastran<sup>7</sup> exist, they are primarily utilized for pre/postprocessing of Nastran.
- *Derivatives* similar to coupling, derivative computation for the coupled system is significantly easier in FEMO. While Nastran does provide derivatives for the structure solver, computing derivatives for the coupled system requires manual derivation and implementation, whereas FEMO utilizes automatic adjoint methods for derivative computation.
- *Lines of code* since FEMO automates coupling and derivative computation, it can save hundreds to thousands of lines of code for those processes.
- *Code maintenance* code maintenance is considerably easier in FEMO. It is fully open-source and can be easily extended to other application scenarios by simply changing the sub-model to be added to the full model. Additionally, thanks to its utilization of dictionaries, FEMO code is much easier to read.

<sup>6</sup> <https://github.com/nasa/NASTRAN-95>.

<sup>7</sup> <https://pynastran-git.readthedocs.io/en/latest/>

## Listing 7 FEMO runscript for sensitivity analysis of aeroelasticity

```

1  # FEMO import statements
2  from femo.fea.fea_dolfinx import *
3  from femo.csdل_opt.fea_model import FEAModel
4  from femo.csdل_opt.state_model import StateModel
5  from femo.csdل_opt.output_model import OutputModel
6  import csdl
7  from python_csdل_backend import Simulator
8
9  # Structure solver import statements
10 from shell_analysis_fenicsx import *
11 from shell_analysis_fenicsx.read_properties import readCLT, sortIndex
12
13 # Aerodynamic solver import statements
14 from FSI_coupling.VLM_sim_handling import *
15 from FSI_coupling.shellmodule_utils import *
16 from FSI_coupling.NodalMapping import *
17 from FSI_coupling.NodalMapping import *
18 from FSI_coupling.mesh_handling_utils import *
19 from FSI_coupling.array_handling_utils import *
20 from FSI_coupling.shellmodule_csdل_interface import (
21     DisplacementMappingImplicitModel,
22     ForceMappingModel,
23     VLMForceIOModel,
24     VLMMeshUpdateModel
25 )
26 #####
27 ##### Structural inputs #####
28 #####
29
30 s_mesh_file_name = "eVTOL_wing_half_tri_107695_136686.xdmf"
31 f_mesh_file_name = 'vlm_mesh_nx2_ny10.npy'
32 path = "../evtol_wing_mesh/"
33 solid_mesh_file = path + s_mesh_file_name
34 vlm_mesh_file = path + f_mesh_file_name
35
36 with XDMFFile(MPI.COMM_WORLD, solid_mesh_file, "r") as xdmf:
37     solid_mesh = xdmf.read_mesh(name="Grid")
38 nel = solid_mesh.topology.index_map(solid_mesh.topology.dim).size_local
39 nn = solid_mesh.topology.index_map(0).size_local
40
41 # define structural properties
42 E = 6.8E10 # unit: Pa (N/m^2)
43 nu = 0.35
44 h_val = 3E-3 # overall thickness (unit: m)
45 y_bc = 0.6
46 PENALTY_BC = True
47
48 element_type = "CG2CG1" # with quad/tri elements
49
50 element = ShellElement(solid_mesh, element_type, )
51 dx_inplane, dx_shear = element.dx_inplane, element.dx_shear
52
53 def pdeRes(h, w, E, f, CLT, dx_inplane, dx_shear, dss=ds, dSS=dS,
54     penalty=False, g=None):
55     elastic_model = ElasticModel(solid_mesh, w, CLT)
56     elastic_energy = elastic_model.elasticEnergy(E, h, dx_inplane, dx_shear)
57     return elastic_model.weakFormResidual(elastic_energy, f,
58     penalty=penalty, dss=dss, dSS=dSS, g=g)
59
60 def compliance(u_mid, f, dxx):
61     return inner(u_mid, f) * dxx
62
63 def volume(h):
64     return h * dx
65
66 def elastic_energy(w, CLT, E, h, dx_inplane, dx_shear):

```

```

67     elastic_model = ElasticModel(solid_mesh,w,CLT)
68     elastic_energy = elastic_model.elasticEnergy(E, h, dx_inplane,dx_shear)
69     return elastic_energy
70
71 # Getting facets of the LEFT and the RIGHT edge
72 DOLFIN_EPS = 3E-16
73 def ClampedBoundary(x):
74     return np.less_equal(x[1], y_bc)
75 def rightChar(x):
76     return np.greater(x[1], 5.2) # measure deflection near wing tip
77 fdim = solid_mesh.topology.dim - 1
78 facets_1 = locate_entities_boundary(solid_mesh,fdim,ClampedBoundary)
79 facets_11 = locate_entities(solid_mesh,fdim,ClampedBoundary)
80 facets_2 = locate_entities_boundary(solid_mesh,fdim,rightChar)
81 # Defining measures
82 facet_tag_1 = meshtags(solid_mesh, fdim, facets_1,
83                       np.full(len(facets_1),100,dtype=np.int32))
84 facet_tag_11 = meshtags(solid_mesh, fdim, facets_11,
85                       np.full(len(facets_11),100,dtype=np.int32))
86 metadata = {"quadrature_degree":4}
87 ds_1 = ufl.Measure('ds',domain=solid_mesh,subdomain_data=facet_tag_1,
88                   metadata=metadata)
89 dS_1 = ufl.Measure('dS',domain=solid_mesh,subdomain_data=facet_tag_11,
90                   metadata=metadata)
91
92 area_2 = dolfinx.mesh.locate_entities(solid_mesh,fdim+1,rightChar)
93 area_tag_2 = meshtags(solid_mesh,fdim+1,area_2,
94                       np.full(len(area_2),10,dtype=np.int32))
95 dx_2 = ufl.Measure('dx',domain=solid_mesh,subdomain_data=area_tag_2,
96                   metadata=metadata)
97
98 fea = FEA(solid_mesh)
99 fea.PDE_SOLVER = "Newton"
100 fea.initialize = True
101 fea.record = False
102 fea.linear_problem = True
103 # Add input to the PDE problem:
104 input_name_1 = 'thickness'
105 input_function_space_1 = FunctionSpace(solid_mesh, ("CG", 1))
106 input_function_1 = Function(input_function_space_1)
107 # Add input to the PDE problem:
108 input_name_2 = 'F_solid'
109 input_function_space_2 = VectorFunctionSpace(solid_mesh, ("CG", 1))
110 input_function_2 = Function(input_function_space_2)
111
112 # Add state to the PDE problem:
113 state_name = 'disp_solid'
114 state_function_space = element.W
115 state_function = Function(state_function_space)
116 g = Function(state_function_space)
117 with g.vector.localForm() as uloc:
118     uloc.set(0.)
119 # Simple isotropic material
120 material_model = MaterialModel(E=E,nu=nu,h=input_function_1)
121 residual_form = pdeRes(input_function_1,state_function,E,input_function_2,
122                       material_model.CLT,dx_inplane,dx_shear,
123                       penalty=PENALTY_BC,dss=ds_1(100),dSS=dS_1(100),g=g)
124
125 # Add output to the PDE problem:
126 output_name_1 = 'compliance'
127 u_mid, theta = split(state_function)
128 output_form_1 = compliance(u_mid,input_function_2, ufl.dx)
129 output_name_2 = 'volume'
130 output_form_2 = volume(input_function_1)
131 output_name_3 = 'elastic_energy'
132 output_form_3 = elastic_energy(state_function,material_model.CLT,E,

```



```

133             input_function_1, dx_inplane, dx_shear)
134
135 with input_function_1.vector.localForm() as uloc:
136     uloc.set(h_val)
137 V0 = assemble(output_form_2)
138
139 fea.add_input(input_name_1, input_function_1)
140 fea.add_input(input_name_2, input_function_2)
141 fea.add_state(name=state_name,
142              function=state_function,
143              residual_form=residual_form,
144              arguments=[input_name_1, input_name_2])
145 fea.add_output(name=output_name_1,
146               type='scalar',
147               form=output_form_1,
148               arguments=[state_name, input_name_1])
149 fea.add_output(name=output_name_2,
150               type='scalar',
151               form=output_form_2,
152               arguments=[input_name_1])
153 fea.add_output(name=output_name_3,
154               type='scalar',
155               form=output_form_3,
156               arguments=[input_name_1, state_name])
157
158 # Set the BCs for the airplane model
159 locate_BC1 = locate_dofs_geometrical((state_function_space.sub(0),
160                                     state_function_space.sub(0).collapse()[0]),
161                                     lambda x: np.less(x[1], y_bc))
162 locate_BC2 = locate_dofs_geometrical((state_function_space.sub(1),
163                                     state_function_space.sub(1).collapse()[0]),
164                                     lambda x: np.less(x[1], y_bc))
165 ubc = Function(state_function_space)
166 with ubc.vector.localForm() as uloc:
167     uloc.set(0.)
168
169 # Strongly enforced boundary conditions
170 if not PENALTY_BC:
171     fea.add_strong_bc(ubc, [locate_BC1], state_function_space.sub(0))
172     fea.add_strong_bc(ubc, [locate_BC2], state_function_space.sub(1))
173
174 #####
175 ##### Aerodynamic inputs #####
176 #####
177 # define vlm input parameters
178 V_inf = 50. # freestream velocity magnitude in m/s
179 AoA = 6. # Angle of Attack in degrees
180 AoA_rad = np.deg2rad(AoA) # Angle of Attack converted to radians
181 # International Standard Atmosphere air density at sea level
182 rho = 1.225 #unit: kg/m^3
183
184 conv_eps = 1e-6 # Convergence tolerance for iterative solution approach
185 iterating = True
186
187 # Construct Aerodynamic mesh
188 # Import a preconstructed vlm mesh
189 vlm_mesh = load_mesh(vlm_mesh_file, np.array([4.28, 0., 2.96]))
190 vlm_mesh_mirrored = mirror_mesh_around_y_axis(vlm_mesh)
191 vlm_mesh_baseline_2d = reshape_3D_array_to_2D(vlm_mesh)
192
193 vlm_mesh_baseline_2d_mirrored = reshape_3D_array_to_2D(
194     vlm_mesh_mirrored)
195
196 # Define force functions and aero-elastic coupling object
197 coupling_obj = FEniCSx_vortexmethod_coupling(solid_mesh, vlm_mesh,
198     state_function_space, RBF_width_par=2.)

```

```

199
200 vlm_mesh_displaced_2d_mirrored = deepcopy(vlm_mesh_baseline_2d_mirrored)
201 vlm_mesh_displaced_3d_mirrored = np.reshape(vlm_mesh_displaced_2d_mirrored,
202       (vlm_mesh_mirrored.shape[0],
203        vlm_mesh_mirrored.shape[1], 3),
204        order='F')
205 vlm_mesh_transposed = construct_VLM_transposed_input_mesh(
206     vlm_mesh_displaced_3d_mirrored)
207
208 # Define CSDL mapping models for force and displacement input/output
209 # management
210 panel_forces_shape = ((vlm_mesh_mirrored.shape[0]-1) \
211     *(vlm_mesh_mirrored.shape[1]-1),
212     vlm_mesh_mirrored.shape[2])
213 panel_forces_3d_shape = ((vlm_mesh_mirrored.shape[0]-1),
214     (vlm_mesh_mirrored.shape[1]-1),
215     vlm_mesh_mirrored.shape[2])
216 starboard_panel_forces_3d_shape = ((vlm_mesh.shape[0]-1),
217     (vlm_mesh.shape[1]-1),
218     vlm_mesh.shape[2],)
219 starboard_panel_forces_shape = ((vlm_mesh.shape[0]-1)*
220     (vlm_mesh.shape[1]-1),
221     vlm_mesh.shape[2],)
222 panel_force_vector_shape = ((vlm_mesh.shape[0]-1)*(vlm_mesh.shape[1]-1) \
223     *vlm_mesh.shape[2],)
224
225 vlm_class = VLM_CADDEE([vlm_mesh_transposed], AoA,
226     V_inf*np.array([np.cos(AoA_rad), 0., np.sin(AoA_rad)]),
227     rho=rho)
228 vlm_model = vlm_class.model # A CSDL model contains VLM as the submodel
229
230 #####
231 ##### The two-way couple problem #####
232 #####
233 # Set up coupling models
234 vlm_force_reshape_model = VLMForceIOModel(input_name='panel_forces',
235     output_name_2d_array='starboard_panel_force_array',
236     output_name_vector='F_aero',
237     input_shape=panel_forces_shape,
238     starboard_3d_shape=starboard_panel_forces_3d_shape,
239     output_vector_length=panel_force_vector_shape,
240     full_3d_shape=panel_forces_3d_shape)
241
242 force_map_model = ForceMappingModel(coupling=coupling_obj,
243     input_name='F_aero',
244     state_name='F_solid',
245     input_shape=(coupling_obj.P_map.shape[1]*3),
246     output_shape=(coupling_obj.Mat_f_sp.shape[0],))
247
248 # Set up structure models
249 solid_model = StateModel(fea=fea, debug_mode=False, state_name=state_name,
250     arg_name_list=fea.states_dict[state_name]['arguments'])
251 compliance_model = OutputModel(fea=fea, output_name=output_name_1,
252     arg_name_list=fea.outputs_dict[output_name_1]['arguments'])
253 volume_model = OutputModel(fea=fea, output_name=output_name_2,
254     arg_name_list=fea.outputs_dict[output_name_2]['arguments'])
255 elastic_energy_model = OutputModel(fea=fea, output_name=output_name_3,
256     arg_name_list=fea.outputs_dict[output_name_3]['arguments'])
257
258 disp_map_model = DisplacementMappingImplicitModel(coupling=coupling_obj,
259     input_name='disp_solid',
260     state_name='disp_fluid',
261     output_name='r_disp_fluid',
262     input_shape=(state_function.vector.size),
263     output_shape=(vlm_mesh.shape[0]*vlm_mesh.shape[1] \
264         *vlm_mesh.shape[2],))

```

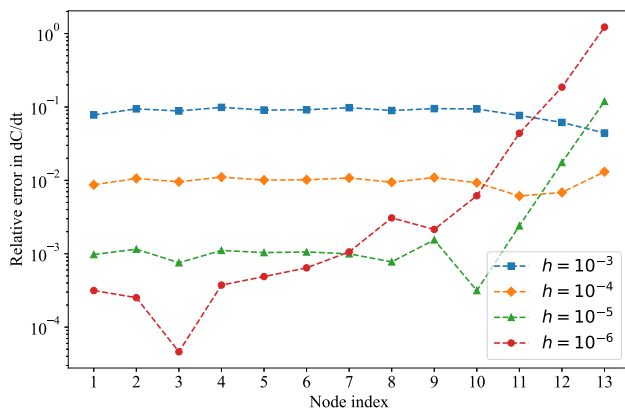
```

265
266 vlm_mesh_update_model = VLMMeshUpdateModel (
267     base_vlm_mesh_2d=vlm_mesh_baseline_2d,
268     starboard_mesh_3d_shape=vlm_mesh.shape,
269     input_name='disp_fluid',
270     output_name_2d='vlm_disp_mat',
271     output_name_3d = 'surf_0',
272     input_shape=(vlm_mesh.shape[0]*vlm_mesh.shape[1] \
273                 *vlm_mesh.shape[2]),
274     output_shape=vlm_mesh_transposed.shape)
275
276 # Connect the models
277 disp_map_model.add(vlm_model, name='vlm_model')
278 disp_map_model.add(vlm_force_reshape_model, name='vlm_force_reshape_model')
279 disp_map_model.add(force_map_model, name='force_map_model')
280 disp_map_model.add(solid_model, name='solid_model')
281 disp_map_model.add(vlm_mesh_update_model, name='vlm_mesh_update_model')
282 disp_map_model.declare_variable('thickness',
283                                 shape=fea.inputs_dict[input_name_1]['shape'],
284                                 val=h_val)
285 model = csdl.Model()
286
287 # Set up Newton solver for the two-way coupled system
288 solve_fixed_point_iteration = model.create_implicit_operation(disp_map_model)
289 solve_fixed_point_iteration.declare_state('disp_fluid',
290                                           residual='r_disp_fluid')
291 solve_fixed_point_iteration.nonlinear_solver = csdl.NonlinearBlockGS(
292     maxiter=100, atol=1e-6, rtol=1e-6)
293 thickness = model.declare_variable('thickness',
294                                   shape=fea.inputs_dict[input_name_1]['shape'],
295                                   val=h_val)
296 x = solve_fixed_point_iteration(thickness)
297
298 # Final assembly
299 model.add(vlm_mesh_update_model, name='vlm_mesh_update_model')
300 model.add(vlm_model, name='vlm_model')
301 model.add(vlm_force_reshape_model, name='vlm_force_reshape_model')
302 model.add(force_map_model, name='force_map_model')
303 model.add(solid_model, name='solid_model')
304 model.add(compliance_model, name='compliance_model')
305 model.add(volume_model, name='volume_model')
306 model.add(elastic_energy_model, name='elastic_energy_model')
307
308 # Test the forward solve
309 sim.run()
310
311 # Compute and output the total derivatives
312 path = "solutions"
313 totals = sim.compute_totals(of=['compliance'], wrt=['thickness'])
314 dCdT = totals[('compliance', 'thickness')]
315 dCdT_function = Function(input_function_space_1)
316 dCdT_function.vector.setArray(dCdT)
317 with XDMFFile(MPI.COMM_WORLD, path+"/gradient_dCdT.xdmf", "w") as xdmf:
318     xdmf.write_mesh(solid_mesh)
319     xdmf.write_function(dCdT_function)
320
321 # Check the total derivatives
322 # It takes too long to run finite difference gradients for this problem.
323 # Though it is easy to add once we reduce the number of design variables.
324
325 error_dict = sim.check_totals(of=['compliance'], wrt=['thickness']
326                              , compact_print=True)
327 dCdT = error_dict[('compliance', 'thickness')]['analytical_jac']
328 dCdT_fd = error_dict[('compliance', 'thickness')]['fd_jac']
329 dCdT_error = error_dict[('compliance', 'thickness')]['error_jac']

```

## Appendix C: Numerical experiments on finite difference step size in the aeroelastic example

For adjoint validation of the aeroelastic model, we use the finite difference method to compare the gradients of compliance with respect to thickness ( $dC/dt$ ) at selected nodes. Figure 18 shows the relative errors in  $dC/dt$  for various finite difference step sizes  $h$ . The results show that smaller  $h$  generally reduces errors, but increases errors significantly for some nodes. Specifically,  $h = 10^{-5}$  and  $h = 10^{-6}$  have smaller errors for the first ten nodes but increase for node 11–13 due to round-off error from lower magnitudes of the gradients at these nodes. Therefore,  $h = 10^{-4}$  is optimal, balancing low errors and stability. The accuracy of the adjoint method in FEMO has been verified as the relative errors for the first ten nodes (whose gradients' magnitude exceeds 0.1) remain reasonable across different step sizes.



**Fig. 18** Relative errors for the gradients of compliance with respect to thickness ( $dC/dt$ ) using different finite difference step sizes  $h$

**Acknowledgements** The work presented in this paper is supported by National Aeronautics and Space Administration Award Number 80NSSC21M0070.

### Declarations

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

**Replication of results** The source code of FEMO, and the example problems discussed in this work are provided in our open-source GitHub repository (<http://github.com/RuruX/femo>).

### References

Alnæs MS, Logg A, Ølgaard KB, Rognes ME, Wells GN (2014) Unified form language: a domain-specific language for weak formulations

- of partial differential equations. *ACM Trans Math Softw* 40(2):1–37
- Alnæs MS, Logg A, Ølgaard KB, Rognes ME, Wells GN (2014) Unified form language: a domain-specific language for weak formulations of partial differential equations. *ACM Trans Math Softw* 40(2):1–37
- Anderson JD Jr (2010) *Fundamentals of aerodynamics*. Tata McGraw-Hill Education, New York
- Annarapu C, Hautefeuille M, Dolbow JE (2012) A robust Nitsche's formulation for interface problems. *Comput Methods Appl Mech Eng* 225–228:44–54
- Arada N, Casas E, Tröltzsch F (2002) Error estimates for the numerical approximation of a semilinear elliptic control problem. *Comput Optim Appl* 23:201–229
- Ashuri T, Zaaijer M, Martins J, van Bussel G, van Kuik G (2014) Multidisciplinary design optimization of offshore wind turbines for minimum levelized cost of energy. *Renew Energy* 68:893–905
- Baratta IA, Dean JP, Dokken JS, Habera M, Hale JS, Richardson CN, Rognes ME, Scroggs MW, Sime N, Wells GN (2023) DOLFINx: the next generation FEniCS problem solving environment
- Bazilevs Y, Calo VM, Hughes TJR, Zhang Y (2008) Isogeometric fluid–structure interaction: theory, algorithms, and computations. *Comput Mech* 43:3–37
- Benaouali A, Kachel S (2019) Multidisciplinary design optimization of aircraft wing using commercial software integration. *Aerosp Sci Technol* 92:766–776
- Bendsøe MP, Kikuchi N (1988) Generating optimal topologies in structural design using a homogenization method. *Comput Methods Appl Mech Eng* 71(2):197–224
- Bleyer J (2018) Numerical tours of computational mechanics with FEniCS
- Campello EMB, Pimenta PM, Wriggers P (2003) A triangular finite shell element based on a fully nonlinear shell formulation. *Comput Mech* 31:505–518
- Christensen P, Klarbring A (2008) *An introduction to structural optimization. Solid mechanics and its applications*. Springer, Dordrecht
- Chung H, Hwang JT, Gray JS, Kim HA (2019) Topology optimization in OpenMDAO. *Struct Multidisc Optim* 59(4):1385–1400
- Cotter C, Shipton J (2012) Mixed finite elements for numerical weather prediction. *J Comput Phys* 231(21):7076–7091
- de Weck O, Agte J, Sobieszczanski-Sobieski J, Arendsen P, Morris A, Spieck M (2007) State-of-the-art and future trends in multidisciplinary design optimization. In: *Collection of technical papers—AIAA/ASME/ASCE/AHS/ASC structures. Structural dynamics and materials conference, 2007, vol 3, p 04*
- Gandarillas V, Joshy AJ, Sperry MZ, Ivanov AK, Hwang JT (2024) A graph-based methodology for constructing computational models that automates adjoint-based sensitivity analysis. *Struct Multidisc Optim* 67:76
- Gill PE, Murray W, Saunders MA (2005) SNOPT: an SQP algorithm for large-scale constrained optimization. *SIAM Rev* 47:99–131
- Gray JS, Hwang JT, Martins JRR, Moore KT, Naylor BA (2019) OpenMDAO: an open-source framework for multidisciplinary design, analysis, and optimization. *Struct Multidisc Optim* 59:1075–1104
- Jiang W, Annarapu C, Dolbow J, Harari I (2015) A robust Nitsche's formulation for interface problems with spline-based finite elements. *Int J Numer Methods Eng* 104(7):676–696
- Jilla C, Miller D (2002) A multiobjective, multidisciplinary design optimization methodology for the conceptual design of distributed satellite systems. *J Spacecr Rockets* 41:09
- Kamensky D (2021) Open-source immersogeometric analysis of fluid–structure interaction using FEniCS and tiGA. *Comput Math Appl* 81:634–648
- Katz J, Plotkin A (2001) *Low-speed aerodynamics, vol 13*. Cambridge University Press, Cambridge

- Kennedy GJ, Martins JR (2014) A parallel finite-element framework for large-scale gradient-based design optimization of high-performance structures. *Finite Elem Anal Des* 87:56–73
- Kenway G, Martins J (2014) Multi-point high-fidelity aerostructural optimization of a transport aircraft configuration. *J Aircr* 51:144–160
- Kenway G, Kennedy G, Martins J (2010) A CAD-free approach to high-fidelity aerostructural optimization. In: 13th AIAA/ISSMO multidisciplinary analysis optimization conference, 2010
- Logg A, Wells GN (2010) DOLFIN: automated finite element computing. *ACM Trans Math Softw* 37(2):1–28
- Logg A, Wells G, Mardal K.-A (2011) Automated solution of differential equations by the finite element method: the FEniCS book. *Lecture notes in computational science and engineering*, vol 84. Springer, Berlin
- Martins JR, Lambe AB (2013) Multidisciplinary design optimization: a survey of architectures. *AIAA J* 51:2049–2075
- Mitusch SK, Funke SW, Dokken JS (2019) DOLFIN-adjoint 2018.1: automated adjoints for FEniCS and Firedrake. *J Open Source Softw* 4(38):1292
- Neighbor GE, Zhao H, Saraeian M, Hsu M-C, Kamensky D (2023) Leveraging code generation for transparent immersogeometric fluid–structure interaction analysis on deforming domains. *Eng Comput* 39:1019–1040
- Nelson PA, Gallagher K, Bloom ID, Dees DW (2011) Modeling the performance and cost of lithium-ion batteries for electric-drive vehicles. Technical Report ANL-11/32. Argonne National Laboratory, Lemont
- Nitsche J (1971) Über ein Variationsprinzip zur Lösung von Dirichlet-Problemen bei Verwendung von Teilräumen, die keinen Randbedingungen unterworfen sind. *Abhandlungen Math Semin Univ Hamburg* 36:9–15
- Ølgaard KB, Wells GN (2010) Optimizations for quadrature representations of finite element tensors through automated code generation. *ACM Trans Math Softw* 37(1):1–23
- Pörner F, Wachsmuth D (2017) Tikhonov regularization of optimal control problems governed by semi-linear partial differential equations. *Math Control Relat Fields* 8(1):315–335
- Rathgeber F, Ham DA, Mitchell L, Lange M, Luporini F, Mcrae ATT, Bercea G-T, Markall GR, Kelly PHJ (2016) Firedrake: automating the finite element method by composing abstractions. *ACM Trans Math Softw* 43(3):1–27
- Schillinger D, Harari I, Hsu M-C, Kamensky D, Stoter SK, Yu Y, Zhao Y (2016) The non-symmetric Nitsche method for the parameter-free imposition of weak boundary and coupling conditions in immersed finite elements. *Comput Methods Appl Mech Eng* 309:625–652
- Scotzniovsky L, Xiang R, Cheng Z, Rodriguez G, Kamensky D, Mi C, Hwang JT (2024) Geometric design of electric motors using adjoint-based shape optimization (Preprint). <https://doi.org/10.21203/rs.3.rs-3941981/v1>
- Scroggs MW, Dokken JS, Richardson CN, Wells GN (2022) Construction of arbitrary order finite element degree-of-freedom maps on polygonal and polyhedral cell meshes. *ACM Trans Math Softw* 48(2):1–23
- Sederberg T, Parry S (1986) Free-form deformation of solid geometric models. *ACM SIGGRAPH Comput Graph* 20:151–160
- Siemens (2014) NX Nastran 10: optimization user’s guide. Siemens. [https://docs.plm.automation.siemens.com/data\\_services/resources/nxnastran/10/help/en\\_US/tDocExt/pdf/optimization.pdf](https://docs.plm.automation.siemens.com/data_services/resources/nxnastran/10/help/en_US/tDocExt/pdf/optimization.pdf)
- Svanberg K, Svärd H (2013) Density filters for topology optimization based on the Pythagorean means. *Struct Multidisc Optim* 48:859–875
- Taylor E (2000) Evaluation of multidisciplinary design optimization techniques as applied to spacecraft design. In: 2000 IEEE aerospace conference. Proceedings (cat. no. 00TH8484), 2000, vol 1, pp 371–384
- Van Schie SPC, Zhao H, Yan J, Xiang R, Hwang JT, Kamensky D (2023) Solver-independent aeroelastic coupling for large-scale multidisciplinary design optimization. In: AIAA Scitech 2023 Forum
- Yan J, Xiang R, Kamensky D, Tolley MT, Hwang JT (2022) Topology optimization with automated derivative computation for multidisciplinary design problems. *Struct Multidisc Optim* 65:151
- Zhao H, Liu X, Fletcher AH, Xiang R, Hwang JT, Kamensky D (2022) An open-source framework for coupling non-matching isogeometric shells with application to aerospace structures. *Comput Math Appl* 111:109–123

**Publisher’s Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.