Structural and Multidisciplinary Optimization
https://doi.org/10.1007/s00158-019-02281-z

RESEARCH PAPER

Automated shape differentiation in the Unified Form Language

David A. Ham[1] · Lawrence Mitchell[2] · Alberto Paganini[3] · Florian Wechsung[3]

Received: 7 December 2018 / Revised: 4 April 2019 / Accepted: 11 April 2019
© The Author(s) 2019

Abstract

We discuss automating the calculation of weak shape derivatives in the Unified Form Language (JAMA 40(2):9:1–9:37 2014) by introducing an appropriate additional step in the pullback from physical to reference space that computes Gâteaux derivatives with respect to the coordinate field. We illustrate the ease of use with several examples.

Keywords Shape derivatives · Finite elements

1 Introduction

Physical models often involve functionals that assign real values to the solutions of partial differential equations (PDEs). For instance, the compliance of a structure is a function of the solution to the elasticity equations, and the drag of a rigid obstacle immersed in a fluid is a function of the solution to the Navier-Stokes equations.

This type of functional depends on the PDE parameters, and it is often possible to compute the derivative of a functional with respect to a chosen set of parameters. This derivative can in turn be used to perform sensitivity analysis and to run optimization algorithms with respect to parameters in the PDE.

Responsible Editor: Ji-Hong Zhu

✉ Alberto Paganini
paganini@maths.ox.ac.uk

David A. Ham
david.ham@imperial.ac.uk

Lawrence Mitchell
lawrence.mitchell@durham.ac.uk

Florian Wechsung
wechsung@maths.ox.ac.uk

1 Department of Mathematics, Imperial College London, London, SW7 2AZ, UK

2 Department of Computer Science, Durham University, Durham, DH1 3LE, UK

3 Mathematical Institute, University of Oxford, Oxford, OX2 6GG, UK

The shape of the physical domain that is part the PDE-model (like the shape of the rigid obstacle mentioned above) is a parameter that is not straightforward to handle. Although we can compute the shape derivative of a functional following shape calculus rules (Delfour and Zolésio 2011), this differentiation exercise is often tedious and error prone. In Schmidt (2018), Schmidt introduces the open-source library FEMorph: an automatic shape differentiation toolbox for the Unified Form Language (UFL, Alnæs et al. 2014). The library FEMorph is based on refactoring UFL expressions and applying shape calculus differentiation rules recursively. It can compute first- and second-order shape derivatives (both in so-called weak and strong form), and it has been successfully employed to solve shape optimization problems (Schmidt et al. 2018).

This article presents an alternative approach to automated shape differentiation. The key idea is to rely solely on pullbacks and standard Gâteaux derivatives. This approach is more generic and robust, because it does not require handling of special cases. In particular, it circumvents the implementation of shape calculus rules, and required only a minor modification of UFL, because UFL supports Gâteaux derivatives with respect to functions. As a result, UFL is now capable of shape differentiating any integral that can be expressed in it.

This article is organized as follows. In Section 2, we revisit shape calculus and describe how to shape differentiate using standard finite element software. In Section 3, we consider three test cases and show how to compute shape derivatives using Firedrake and UFL. In Section 4, we describe code validation of this new UFL feature. In Section 5, we solve a PDE-constrained shape optimization test case with a descent algorithm implemented in Firedrake and UFL.

Published online: 02 August 2019

Finally, in Section 6, we summarize the contribution of this article.

2 Shape differentiation on the reference element

A *shape functional* is a map $\mathrm{J} : \mathcal{U} \to \mathbb{R}$ defined on the collection of domains $\mathcal{U}$ in $\mathbb{R}^d$, where $d \in \mathbb{N}$ denotes the space dimension.

We follow the perturbation of identity approach (Simon 1980) and for a vector field $\mathbf{V} \in W^{1,\infty}(\mathbb{R}^d, \mathbb{R}^d)$, we consider the family of transformations $\{\mathbf{P}_s\}_{s\geq 0}$ defined by $\mathbf{P}_s(\mathbf{x}) = \mathbf{x} + s\mathbf{V}(\mathbf{x})$ for every $\mathbf{x} \in \mathbb{R}^d$. Note that $\mathbf{P}_s$ is a diffeomorphism for any sufficiently small $s \in \mathbb{R}^+$.

For a domain $\Omega \in \mathcal{U}$, let $\Omega_s := \mathbf{P}_s(\Omega)$ and assume that $\Omega_s \in \mathcal{U}$ for s sufficiently small. The *shape directional derivative* of J at Ω in direction $\mathbf{V}$ is the derivative

$$\mathrm{dJ}(\Omega)[\mathbf{V}] := \lim_{s\searrow 0} \frac{\mathrm{J}(\Omega_s) - \mathrm{J}(\Omega)}{s} .$$

We say that J is *shape differentiable in* Ω if the directional derivative $\mathrm{dJ}(\Omega)[\mathbf{V}]$ exists for every direction $\mathbf{V} \in W^{1,\infty}(\mathbb{R}^d, \mathbb{R}^d)$, and if the associated map $\mathrm{dJ}(\Omega) : W^{1,\infty}(\mathbb{R}^d, \mathbb{R}^d) \mapsto \mathbb{R}$ is linear and continuous. In this case, the linear operator $\mathrm{dJ}(\Omega)$ is called the *shape derivative* of J in Ω.

To illustrate the shape differentiation of a shape functional, we consider the prototypical example

$$\mathrm{J}(\Omega_s) = \int_{\Omega_s} u_{\mathbf{P}_s} \,\mathrm{d}\boldsymbol{x} , \tag{1}$$

where $u_{\mathbf{P}_s}$ is a scalar function.[1] The subscript $\mathbf{P}_s$ highlights the possible dependence of $u_{\mathbf{P}_s}$ on the domain Ω_s.

The standard procedure to compute dJ is to employ transformation techniques and rewrite

$$\mathrm{J}(\Omega_s) = \int_{\Omega} (u_{\mathbf{P}_s} \circ \mathbf{P}_s) \det(\mathbf{DP}_s) \,\mathrm{d}\boldsymbol{x} , \tag{2}$$

where $\mathbf{DP}_s$ denotes the Jacobian matrix of $\mathbf{P}_s$ and $u_{\mathbf{P}_s} \circ \mathbf{P}_s$ denotes the composition of $u_{\mathbf{P}_s}$ with $\mathbf{P}_s$, that is, $(u_{\mathbf{P}_s} \circ \mathbf{P}_s)(\mathbf{x}) = u_{\mathbf{P}_s}(\mathbf{P}_s(\mathbf{x}))$ for every $\mathbf{x} \in \Omega$. Note that $\det(\mathbf{DP}_s) > 0$ for s sufficiently small. Then, by linearity of the integral, the shape derivative dJ is given by

$$\begin{aligned}\mathrm{dJ}(\Omega)[\mathbf{V}] &= \int_{\Omega} \mathrm{d_s}((u_{\mathbf{P}_s} \circ \mathbf{P}_s) \det \mathbf{DP}_s) \,\mathrm{d}\boldsymbol{x} \\ &= \int_{\Omega} \mathrm{d_s}(u_{\mathbf{P}_s} \circ \mathbf{P}_s) + u_{\mathbf{P}_0} \mathrm{div}(\mathbf{V}) \,\mathrm{d}\boldsymbol{x} ,\end{aligned} \tag{3}$$

where $\mathrm{d_s}(\cdot)$ denotes the derivative with respect to s at $s = 0$. The term $\mathrm{d_s}(u_{\mathbf{P}_s} \circ \mathbf{P}_s)$ is often called the *material derivative* (Berggren 2010). Its explicit formula depends on whether the function $u_{\mathbf{P}_s}$ does or does not dependent on Ω_s (see Section 3).

Next, we repeat the derivation of (3) in the context of finite elements and derive an alternative formula for dJ. Let $\{K_i\}_{i\in\mathcal{I}}$ be a partition of Ω such that $\dot{\cup}_i \overline{K}_i = \overline{\Omega}$ and such that the elements K_i are non-overlapping. Additionally, let $\{\mathbf{F}_i\}_{i\in\mathcal{I}}$ be a family of diffeomorphisms such that $\mathbf{F}_i(\hat{K}) = K_i$ for every $i \in \mathcal{I}$, where $\hat{K}$ denotes a reference element. This induces a partition $\{\mathbf{P}_s(K_i)\}_{i\in\mathcal{I}}$ of Ω_s. To evaluate (1), standard finite element software rewrites it as

$$\begin{aligned}\mathrm{J}(\Omega_s) &= \sum_{i\in\mathcal{I}} \int_{\mathbf{P}_s(K_i)} u_{\mathbf{P}_s} \,\mathrm{d}\boldsymbol{x} \\ &= \sum_{i\in\mathcal{I}} \int_{\hat{K}} (u_{\mathbf{P}_s} \circ \mathbf{P}_s \circ \mathbf{F}_i) |\det(\mathbf{D}(\mathbf{P}_s \circ \mathbf{F}_i))| \,\mathrm{d}\hat{\boldsymbol{x}} .\end{aligned} \tag{4}$$

Let $\mathbf{F}_i^{-1}$ denote the inverse of $\mathbf{F}_i$, that is, $\mathbf{F}_i^{-1}(\mathbf{F}_i(\mathbf{x})) = \mathbf{x}$ for every $\mathbf{x} \in \hat{K}$, and $\mathbf{F}_i(\mathbf{F}_i^{-1}(\mathbf{x})) = \mathbf{x}$ for every $\mathbf{x} \in K_i$. Since $\mathbf{P}_s = (\mathbf{P}_s \circ \mathbf{F}_i) \circ \mathbf{F}_i^{-1}$ and $\mathbf{P}_s \circ \mathbf{F}_i = \mathbf{F}_i + s\mathbf{V} \circ \mathbf{F}_i$, we can rewrite (4) as follows:

$$\begin{aligned}\mathrm{J}(\Omega_s) = \sum_{i\in\mathcal{I}} \int_{\hat{K}} \Big(u_{(\mathbf{F}_i + s\mathbf{V}\circ\mathbf{F}_i)\circ\mathbf{F}_i^{-1}} \circ (\mathbf{F}_i + s\mathbf{V} \circ \mathbf{F}_i)\Big) \\ |\det(\mathbf{D}(\mathbf{F}_i + s\mathbf{V} \circ \mathbf{F}_i))| \,\mathrm{d}\hat{\boldsymbol{x}} .\end{aligned} \tag{5}$$

Let $\{g_i\}_{i\in\mathcal{I}}$ be the collection of maps defined by

$$g_i(\mathbf{T}) := (u_{\mathbf{T}\circ\mathbf{F}_i^{-1}} \circ \mathbf{T}) |\det(\mathbf{DT})| .$$

Then, formula (5) can be rewritten as

$$\mathrm{J}(\Omega_s) = \sum_{i\in\mathcal{I}} \int_{\hat{K}} g_i(\mathbf{F}_i + s\mathbf{V} \circ \mathbf{F}_i) \,\mathrm{d}\hat{\boldsymbol{x}} ,$$

and taking the derivative of (5) with respect to s implies

$$\mathrm{dJ}(\Omega)[\mathbf{V}] = \sum_{i\in\mathcal{I}} \int_{\hat{K}} \mathrm{d_s}(g_i(\mathbf{F}_i + s\mathbf{V} \circ \mathbf{F}_i)) \,\mathrm{d}\hat{\boldsymbol{x}} . \tag{6}$$

Equation (6) gives an alternative and equivalent expression for the shape derivative (3). However, to derive formula (3), it is necessary to follow shape calculus rules by hand, which is often a tedious and error prone exercise. Equation (6), by contrast, can be derived automatically with finite element software. Indeed, to evaluate $\mathrm{J}(\Omega)$, standard finite element software rewrites it as

$$\mathrm{J}(\Omega) = \sum_{i\in\mathcal{I}} \int_{\hat{K}} g_i(\mathbf{F}_i) .$$

In UFL, the maps $\{g_i\}_{i\in\mathcal{I}}$ are constructed symbolically and in an automated fashion. Therefore, it is possible to evaluate $\mathrm{dJ}(\Omega)[\mathbf{V}]$ by performing the steps necessary for the assembly of $\mathrm{J}(\Omega)$ and, at the appropriate time, differentiating the maps $\{g_i\}_{i\in\mathcal{I}}$. To be precise, this differentiation corresponds

[1] Shape functionals that involve vector fields or boundary integrals can be treated following the same steps and employing suitable pullbacks.

to a standard Gâteaux directional derivative, because the integrand in (6) corresponds to the following limit

$$\mathrm{d_s}(g_i(\mathbf{F}_i + s\mathbf{V} \circ \mathbf{F}_i)) = \lim_{s \searrow 0} \frac{g_i(\mathbf{F}_i + s\mathbf{V} \circ \mathbf{F}_i) - g_i(\mathbf{F}_i)}{s},$$

which can be interpreted as the Gâteaux directional derivative of g_i at $\mathbf{T} = \mathbf{F}_i$ in the direction $\mathbf{V} \circ \mathbf{F}_i$ (Hinze et al. 2009, Def. 1.29). This viewpoint is important to correctly implement this differentiation step in the existing pipeline in UFL (see Fig. 1). We emphasize that this also enables computing higher order shape derivatives by simply taking higher order Gâteaux derivatives in (6).

Remark 1 Lagrange finite element global basis functions are obtained by gluing local parametric basis functions, that is, basis functions $\{b_m^i\}_{m \in \mathcal{M}}$ defined only on K_i and of the form $b_m = \hat{b}_m \circ \mathbf{F}_i^{-1}$, where $\{\hat{b}_m\}_{m \in \mathcal{M}}$ is the set of reference local basis functions, which are defined only on the reference element $\hat{K}$. If $\mathbf{V}$ lives in a Lagrange finite element space built on the partitioning $\{K_i\}_{i \in \mathcal{I}}$, it is possible to evaluate $\mathrm{dJ}(\mathbf{T})[\mathbf{V}]$ by computing the Gâteaux derivative in (6) in the direction of the reference local basis functions $\{\hat{b}_m\}_{m \in \mathcal{M}}$ (instead of in the direction $\mathbf{V} \circ \mathbf{F}_i$) and summing these values. This allows us to fully rely on the symbolic differentiation capabilities of UFL.

Remark 2 The approach does not rely on the element being affinely mapped, but extends to elements that are mapped using a Piola transform such as the Raviart-Thomas or Nedelec elements. However, the current implementation does not work for elements such as the Hermite element that require different pullbacks for point evaluation and derivative degrees of freedom.

3 Examples

In this section, we consider three examples based on (1) that cover most applications. For these examples, we give

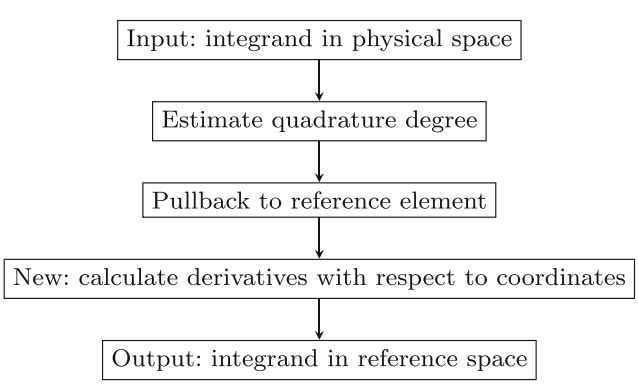

Fig. 1 Symbolic workflow in UFL to transform integrals from physical to reference space

```
from firedrake import *
mesh = UnitSquareMesh(10, 10)
x, y = X = SpatialCoordinate(mesh)
J = (x*x + y*y - 1) * dx
dJ = assemble(derivative(J, X))
```

Listing 1 Firedrake code to compute dJ from example 1 when $u(x, y) = x^2 + y^2 - 1$

explicit expressions of dJ using (3) and (6) and show how to compute dJ with the finite element software Firedrake[2] (Rathgeber et al. 2016). To shorten the notation, we define $\mathbf{V}_i := \mathbf{V} \circ \mathbf{F}_i$.

Example 1 Let the integrand be independent of Ω, i.e., $u_{\mathbf{P}_s} = u$ for some function u. Then, the chain rule implies that $\mathrm{d_s}(u_{\mathbf{P}_s} \circ \mathbf{P}_s) = \mathrm{d_s}(u \circ \mathbf{P}_s) = \nabla u \cdot \mathbf{V}$. Recalling $\mathrm{d_s}(\det \mathbf{DP}_s) = \mathrm{div}(\mathbf{V})$, we conclude that (3) becomes

$$\mathrm{dJ}(\Omega)[\mathbf{V}] = \int_\Omega \nabla u \cdot \mathbf{V} + u\,\mathrm{div}(\mathbf{V})\,\mathrm{d}\boldsymbol{x}\,. \tag{7}$$

On the other hand, inserting $u_{\mathbf{P}_s} = u$ into (6), we obtain the equivalent expression:

$$\mathrm{dJ}(\Omega)[\mathbf{V}] = \sum_{i \in \mathcal{I}} \int_{\hat{K}} \Big(\mathbf{V}_i \cdot (\nabla u \circ \mathbf{F}_i) + (u \circ \mathbf{F}_i)\mathrm{tr}(\mathbf{DV}_i \mathbf{DF}_i^{-1})\Big) \,|\det(\mathbf{DF}_i)|\,\mathrm{d}\hat{\boldsymbol{x}}\,. \tag{8}$$

Example code is shown in Listing 1. Functionals with domain independent integrands are used in applications including image segmentation (Hintermüller and Ring 2004) or, when $u \equiv 1$, to enforce volume constraints in shape optimization.

Example 2 Let $\{V_h(\Omega_s)\}_s$ be a family of scalar finite element spaces such that the global basis functions $\{B_s^i\}_i$ of $V_h(\Omega_s)$ are of the form $B_s^i = B^i \circ \mathbf{P}_s^{-1}$, where $\{B^i\}_i$ are basis functions of $V_h(\Omega)$ and $\mathbf{P}_s^{-1}$ is the inverse of $\mathbf{P}_s$, that is, $\mathbf{P}_s^{-1}(\mathbf{P}_s(\mathbf{x})) = \mathbf{P}_s(\mathbf{P}_s^{-1}(\mathbf{x})) = \mathbf{x}$ for every $\mathbf{x} \in \mathbb{R}^d$. Let $v_{\mathbf{P}_s} \in V_h(\Omega_s)$ and $u_{\mathbf{P}_s} = v_{\mathbf{P}_s} + \|\nabla v_{\mathbf{P}_s}\|^2$. Since

$$v_{\mathbf{P}_s} \circ \mathbf{P}_s + \|(\nabla v_{\mathbf{P}_s}) \circ \mathbf{P}_s\|^2 = v_{\mathbf{P}_0} + \|\mathbf{DP}_s^{-T} \nabla v_{\mathbf{P}_0}\|^2\,, \tag{9}$$

equation (3) becomes

$$\mathrm{dJ}(\Omega)[\mathbf{V}] = \int_\Omega (v_{\mathbf{P}_0} + \|\nabla v_{\mathbf{P}_0}\|^2)\mathrm{div}(\mathbf{V}) - 2\nabla v_{\mathbf{P}_0} \cdot (\mathbf{DV}^T \nabla v_{\mathbf{P}_0})\,\mathrm{d}\boldsymbol{x}\,. \tag{10}$$

On the other hand, note that for any $\hat{\mathbf{x}} \in \hat{K}$ and for $i \in \mathcal{I}$, it holds $v_{\mathbf{P}_s}(\mathbf{P}_s(\mathbf{F}_i(\hat{\boldsymbol{x}}))) = v_i(\hat{\boldsymbol{x}})$, where v_i is a linear

[2]Examples using FEniCS (Logg and Wells 2010; Logg et al. 2012) will be almost identical, modulo small differences in setting up initial conditions

```
from firedrake import *
mesh = UnitSquareMesh(10, 10)
V = FunctionSpace(mesh, "CG", 1)
x, y = X = SpatialCoordinate(mesh)
v = interpolate(sin(x) * cos(y), V)
J = v * dx + inner(grad(v), grad(v)) * dx
dJ = assemble(derivative(J, X))
```

Listing 2 Firedrake code to compute dJ from example 2. Note that in this case, v does not depend explicitly on x and y

combination of the local basis functions $\{\hat{b}_m\}_{m\in\mathcal{M}}$ defined on the reference element $\hat{K}$. Therefore,

$$\begin{aligned} &v_{\mathbf{P}_s} \circ \mathbf{P}_s \circ \mathbf{F}_i + \|(\nabla v_{\mathbf{P}_s}) \circ \mathbf{P}_s \circ \mathbf{F}_i\|^2 \\ &= \hat{v}_i + \|\mathbf{D}(\mathbf{P}_s \circ \mathbf{F}_i)^{-T} \nabla \hat{v}_i\|^2 \quad \text{on } \hat{K}\,, \end{aligned} \tag{11}$$

and (6) becomes

$$\begin{aligned} \mathrm{dJ}(\Omega)[\mathbf{V}] = &\sum_{i\in\mathcal{I}} \int_{\hat{K}} \Big((\hat{v}_i + \|\mathbf{DF}_i^{-T} \nabla v_i\|^2) \mathrm{tr}(\mathbf{DV}_i \mathbf{DF}_i^{-1}) \\ &-2\nabla \hat{v}_i \cdot \mathbf{DF}_i^{-1} \mathbf{DV}_i \mathbf{DF}_i^{-1} \mathbf{DF}_i^{-T} \nabla \hat{v}_i \Big) \,|\det(\mathbf{DF}_i)|\, \mathrm{d}\hat{\boldsymbol{x}}. \end{aligned} \tag{12}$$

Listing 2 shows code for this case, using as $v_{\mathbf{P}_0}$, the piecewise affine Lagrange interpolant of $\sin(x)\cos(y)$.

Example 3 Let $u_{\mathbf{P}_s}$ be the finite element solution to the boundary value problem

$$-\Delta u_{\mathbf{P}_s} + u_{\mathbf{P}_s} = f \quad \text{in } \Omega_s\,, \quad \nabla u_{\mathbf{P}_s} \cdot \mathbf{n} = 0 \quad \text{on } \partial\Omega_s\,. \tag{13}$$

In this case, the functional (1) is said to be PDE-constrained, and computing its shape derivative is less straightforward. The standard procedure is to introduce an appropriate Lagrangian functional (Delfour and Zolésio 2011, Ch. 10, Sect. 5). For this example, the Lagrangian is

$$\mathrm{L}_s(u_{\mathbf{P}_s}, v_s) := \mathrm{J}(\Omega_s) + \mathrm{e}_s(u_{\mathbf{P}_s}, v_s), \tag{14}$$

where

$$\mathrm{e}_s(u_{\mathbf{P}_s}, v_s) := \int_{\Omega_s} \nabla u_{\mathbf{P}_s} \cdot \nabla v_s + u_{\mathbf{P}_s} v_s - f v_s \,\mathrm{d}\boldsymbol{x} \tag{15}$$

stems from the weak formulation of the PDE constraint (13). The shape derivative dJ is equal to the shape derivative of $\mathrm{L}_s(u \circ \mathbf{P}_s^{-1}, p \circ \mathbf{P}_s^{-1})$, where u is the solution to (15) for $s = 0$ and $p \in V_h(\Omega)$ is the solution to an adjoint boundary value problem. The shape derivative of $\mathrm{L}_s(u \circ \mathbf{P}_s^{-1}, p \circ \mathbf{P}_s^{-1})$ can be computed as in example 2. The result is

$$\begin{aligned} \mathrm{dJ}(\Omega)[\mathbf{V}] = &\int_{\Omega} (u + \nabla u \cdot \nabla p + up - fp)\mathrm{div}(\mathbf{V}) \\ &- p\nabla f \cdot \mathbf{V} - \nabla u \cdot (\mathbf{DV} + \mathbf{DV}^T)\nabla p \,\mathrm{d}\boldsymbol{x}\,. \end{aligned} \tag{16}$$

For this example, we omit the equivalent formula on the reference element because of its length. However, as Listing 3 shows, UFL removes the tedium of deriving the shape derivative, and we can easily compute dJ.

```
1  from firedrake import *
2  mesh = UnitSquareMesh(10, 10)
3  V = FunctionSpace(mesh, "CG", 1)
4  x, y = X = SpatialCoordinate(mesh)
5  u, p, v = Function(V),Function(V),TestFunction(V)
6  e = inner(grad(u), grad(v))*dx + (u*v - x*y*v)*dx
7  J = u * dx
8  solve(e == 0, u)
9  solve(adjoint(derivative(e,u)) == -derivative(J,u,v),p)
10 L = replace(e, {v: p}) + J
11 dJ = assemble(derivative(L, X))
```

Listing 3 Firedrake code to compute dJ from example 3 when $f(x, y) = xy$ in (15)

Remark 3 With appropriate modifications, the same code can be use for functionals constrained to boundary value problems with Neumann or Dirichlet boundary conditions. For the Neumann case, it is sufficient to add the Neumann forcing term in line 6 of Listing 3. For the Dirichlet case, one needs to replace `u` with `u+g` in lines 6 and 7 (where `g` is the function defined in terms of `X` that describes the Dirichlet boundary condition) and impose homogeneous Dirichlet boundary conditions in lines 8 and 9.

Remark 4 To evaluate the action of the shape Hessian of a PDE-constrained functional, one can follow the instructions given in Hinze et al. (2009, p. 65). Note that by computing shape derivatives as in (6), it is straightforward to combine shape derivatives of $\mathrm{L}_s(u \circ \mathbf{P}_s^{-1}, p \circ \mathbf{P}_s^{-1})$ with standard Gâteaux derivatives with respect to $u \circ \mathbf{P}_s^{-1}$.

4 Code validation

We validate our implementation by testing that the Taylor expansions truncated to first and second order satisfy the asymptotic conditions

$$\delta_1(\mathrm{J}, \mathrm{s}) = \mathrm{O}(\mathrm{s}^2) \quad \text{and} \quad \delta_2(\mathrm{J}, \mathrm{s}) = \mathrm{O}(\mathrm{s}^3)\,, \tag{17}$$

where

$$\delta_1(\mathrm{J}, \mathrm{s}) := \|\mathrm{J}(\Omega_\mathrm{s}) - \mathrm{J}(\Omega) - \mathrm{s}\mathrm{dJ}(\Omega)[\mathbf{V}]\|$$

and

$$\begin{aligned} \delta_2(\mathrm{J}, \mathrm{s}) := \|&\mathrm{J}(\Omega_\mathrm{s}) - \mathrm{J}(\Omega) \\ &- s\mathrm{dJ}(\Omega)[\mathbf{V}] - \frac{1}{2}s^2 \mathrm{d}^2\mathrm{J}(\Omega)[\mathbf{V}, \mathbf{V}]\|\,. \end{aligned}$$

In Fig. 2, we plot the values of δ_1 and δ_2 for $s = 2^{-1}, 2^{-2}, \ldots, 2^{-10}$, and J as in examples 1 and 3 from the previous section (we denote these functionals J_1 and J_2 respectively). The vector field $\mathbf{V}$ is chosen randomly. This experiment clearly displays the asymptotic rates predicted by (17).

We have repeated this numerical experiment for many other test cases, including functionals that are not linear in u, functionals given by integrals over $\partial\Omega$ involving the normal

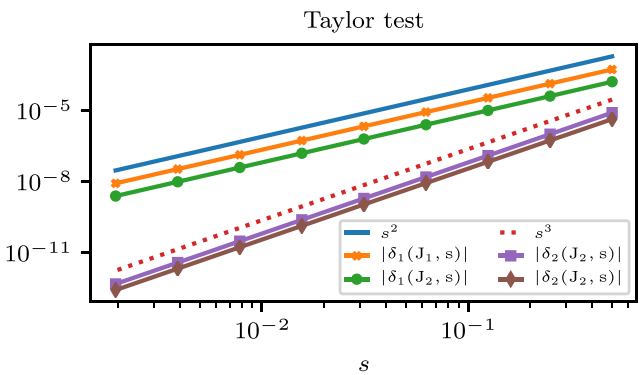

Fig. 2 Taylor test for examples 1 and 3. The convergence rates match the expected convergence

n, and functionals that are constrained to linear and nonlinear boundary value problems with nonconstant right-hand sides and nonconstant Neumann and Dirichlet boundary conditions. In every instance, we have observed the asymptotic rates predicted by (17). The code for these numerical experiments is available at "Software used in 'Automated shape differentiation in the Unified Form Language' (2019)".

5 Shape optimization of a pipe

In this section, we show how to use Firedrake and the new UFL capability to code a PDE-constrained shape optimization algorithm. As test case, we consider the optimization of a pipe to minimize the dissipation of kinetic energy of the fluid into heat. This example is taken from Schmidt (2010, Sect. 6.2.3). To simplify the exposition, we use a very simple optimization strategy. At the end of the section, we will comment on possible improvements.

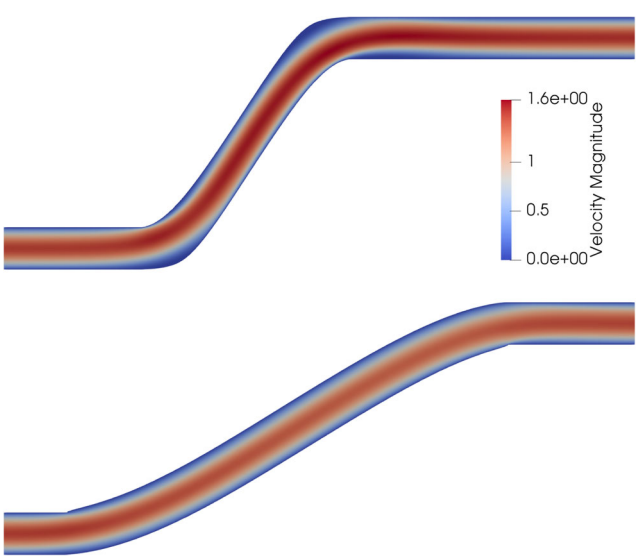

Fig. 3 Initial (top) and optimized (bottom) shape of a pipe connecting a given inflow and outflow

The initial design of the pipe is shown in Fig. 3 (top). The pipe contains viscous fluid (with viscosity ν), which flows in from the left and is modeled using the incompressible Navier-Stokes equations. To be precise, let Ω be the shape of the pipe, $\Gamma \subset \partial\Omega$ be the outflow boundary of the pipe (that is, the end of the pipe on the right), and $\mathbf{u}$ and p be the velocity and the pressure of the fluid, respectively. Then, $\mathbf{u}$ and p satisfy

$$\begin{aligned} -\nu\Delta\mathbf{u} + \mathbf{u}\nabla\mathbf{u} + \nabla p &= 0 && \text{in } \Omega\,, \\ \operatorname{div}\mathbf{u} &= 0 && \text{in } \Omega\,, \\ \mathbf{u} &= \mathbf{g} && \text{on } \partial\Omega \setminus \Gamma\,, \\ p\mathbf{n} - \nu\nabla u \cdot \mathbf{n} &= 0 && \text{on } \Gamma\,. \end{aligned}$$

Here, $\mathbf{g}$ is given by a Poiseuille flow at the inlet and is zero on the walls of the pipe

The goal is to modify the central region of the pipe so that the shape functional

$$\mathrm{J}(\Omega) = \int_\Omega \nu\nabla\mathbf{u} : \nabla\mathbf{u}\,\mathrm{d}\boldsymbol{x}$$

is minimized. To solve this shape optimization problem, we parametrize the initial design with a polygonal mesh and update the node coordinates using a descent direction optimization algorithm with a fixed step size. As descent directions, we use Riesz representatives of the shape gradient with respect to the inner product induced by the Laplacian, i.e., at each step the deformation is given by the solution to

$$\begin{aligned} -\Delta\mathbf{V} &= -\mathrm{dJ}(\Omega) && \text{in } \Omega \\ \mathbf{V} &= 0 && \text{on fixed boundaries.} \end{aligned} \tag{18}$$

This approach is also known as Laplace smoothing. To avoid degenerate results, we penalize changes of the pipe volume. The whole algorithm, comprising of state and adjoint equations and shape derivatives, is contained in Listing 4 and described in detail in the following paragraph. The optimized shape is displayed in Fig. 3 (bottom), the convergence history is in Fig. 4. These results are compatible with those in Schmidt (2010, Sect. 6.2.3) and clearly indicate the success of the shape optimization algorithm.

Description of Listing 4 In lines 2–4, we load the finite element mesh `pipe.msh` and extract the vertex coordinates. This mesh is generated with Gmsh (Geuzaine and Remacle 2009) and is available as part of "Software used in 'Automated shape differentiation in the Unified Form Language' (2019)". Lines 5–8 define the Gramian matrix of the inner product employed to compute descent directions. In lines 9–14, we define the space of P2-P1 Taylor-Hood finite elements, which we use to discretize the weak formulation of the Navier-Stokes equations, and set up the functions containing the solutions to the state and adjoint equation as well as the test functions for the weak form. In lines 15–22, we

```
1  from firedrake import *

2  mesh = Mesh("pipe.msh")
3  coords = mesh.coordinates.vector()
4  X = SpatialCoordinate(mesh)

5  W = mesh.coordinates.function_space()
6  gradJ = Function(W)
7  phi, psi = TrialFunction(W), TestFunction(W)
8  A_riesz = assemble(inner(grad(phi), grad(psi)) * dx)

9  Z = VectorFunctionSpace(mesh, "CG", 2) \
10     * FunctionSpace(mesh, "CG", 1)
11 z, z_adjoint = Function(Z), Function(Z)
12 u, p = split(z)
13 test = TestFunction(Z)
14 v, q = split(test)

15 nu = 1./400.
16 e = nu*inner(grad(u), grad(v))*dx - p*div(v)*dx \
17     + inner(dot(grad(u), u), v)*dx + div(u)*q*dx
18 uin = 6 * as_vector([(1-X[1])*X[1], 0])
19 bcs = [DirichletBC(Z.sub(0), 0., [3, 4]),
20        DirichletBC(Z.sub(0), uin, 1)]
21 sp = {"mat_type": "aij", "pc_type": "lu",
22       "pc_factor_mat_solver_type": "mumps"}

23 J = nu * inner(grad(u), grad(u)) * dx
24 volume = Constant(1.) * dx(domain=mesh)
25 target_volume = assemble(volume)
26 dvol = derivative(volume, X)
27 L = replace(e, {test: z_adjoint}) + J
28 dL = derivative(L, X)
29 c = 0.1

30 out = File("u.pvd")
31 def solve_state_and_adjoint():
32     solve(e==0, z, bcs=bcs, solver_parameters=sp)
33     solve(derivative(L, z)==0, z_adjoint,
34           bcs=homogenize(bcs), solver_parameters=sp)
35     out.write(z.split()[0])

36 solve_state_and_adjoint()
37 for i in range(100):
38     dJ = assemble(dL).vector() \
39         + assemble(dvol).vector() * c * 2\
40         * (assemble(volume)-target_volume)
41     solve(A_riesz, gradJ, dJ,
42           bcs=DirichletBC(W, 0, [1, 2, 3]))
43     print("i = %3i; J = %.6f; ||dJ|| = %.6f"
44           % (i, assemble(J), norm(grad(gradJ))))
45     coords -= 0.5 * gradJ.vector()
46     solve_state_and_adjoint()
```

Listing 4 Firedrake code to optimize the shape of a pipe and minimize the dissipation of kinetic energy into heat. Lines 26 and 28 use the newly developed automatic shape differentiation

define the weak formulation of the Navier-Stokes equations and certain parameters to prescribe the use of the MUMPS direct solver (Amestoy et al. 2000) to solve the linearized equations. In lines 23–29, we define the shape functional J, the functional describing the volume of the shape, as well as the Lagrangian and its derivative. In particular, note that the shape derivative of the Lagrangian can be computed

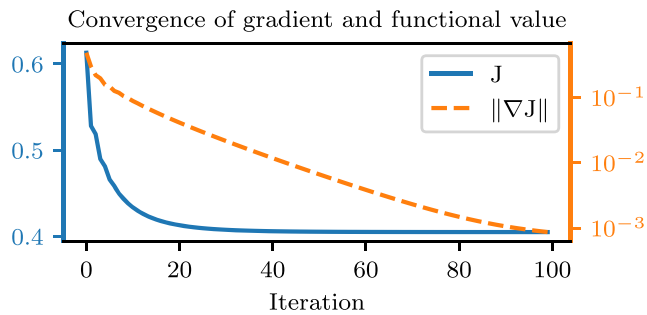

Fig. 4 The value of the objective (plotted in linear scale) is reduced from approximately 0.61298 to 0.40506. The H^1-norm of the gradient (plotted in logarithmic scale) is reduced from 0.487274 to 0.000870

with the simple command `dL = derivative(L, X)` in line 28. Without the new automatic shape differentiation capability in UFL, line 28 would have to be replaced with the following formula

```
dL = -inner(nu*grad(u)*grad(W), grad(v))*dx
     -inner(nu*grad(u), grad(v)*grad(W))*dx
     -inner(v, grad(u)*grad(W)*u)*dx
     +tr(grad(v)*grad(W))*p*dx
     -tr(grad(u)*grad(W))*q*dx
     +div(W)*inner(nu*grad(u), grad(v))*dx
     -div(W)*inner(div(v), p)*dx
     +div(W)*inner(div(u), q)*dx
     +div(W)*inner(v, grad(u)*u)*dx
     +nu*inner(grad(u), grad(u))*div(W)*dx
     -2*nu*inner(grad(u)*grad(W), grad(u))
     *dx
```

In lines 30–35, we set up a function that updates the solution to the state and the adjoint equations. We emphasize that this shape optimization problem is not self-adjoint and that UFL derives the adjoint equation automatically. Note that, whenever the function `solve_state_and_adjoint` is called, the new values of the velocity **u** are stored in the file `u.pvd` (which can visualized using Paraview (Ahrens et al. 2005)). Finally, lines 36–46 contain the optimization algorithm: for 100 iterations, we compute the shape derivative and penalize volume changes (lines 38–40), compute the descent direction (lines 41–42), update the domain (line 45), and update the state and adjoint solutions (line 46).

Remark 5 The optimization algorithm of Listing 4 is based on a simple optimization strategy and can be improved in several ways, at the mere cost of adding lines of code. For instance, instead of using a fixed step size and a fixed number of iterations, one could implement an adaptive step-size selection and stopping criteria. Additionally, one could experiment with different inner products to define descent directions (https://epubs.siam.org/doi/abs/10.1137/17M1152711), as well as compute second-order derivatives of J and implement (Quasi-)Newton methods (Schmidt 2018). Despite the room for improvement, we would like to

stress that Listing 4 can be readily used for a 3D problem by simply passing a 3D mesh and changing the inflow boundary condition in line 18.

6 Discussion

We have presented a new and equivalent formulation of shape derivatives in the context of finite elements as Gâteaux derivatives on the reference element. While the formulation applies to finite elements in general, we have implemented this new approach in UFL due to its extensive support for symbolic calculations. This new UFL capability allows computing shape derivatives of functionals that are defined as volume or boundary integrals, and that are constrained to linear and nonlinear PDEs. During shape differentiation, our code treats finite element functions and global functions differently. This behavior is correct and necessary to handle PDE-constraints properly. In combination with a finite element software package, such as FEniCS or Firedrake, that takes as input UFL, this enables the entirely automated shape differentiation of functionals subject to boundary value problems. This notably simplifies tackling PDE-constrained shape optimization problems.

Compared to the existing shape differentiation toolbox FEMorph, our code does not compute shape derivatives in strong form because it neither relies on shape calculus differentiation rules nor performs integration by parts. However, in practice, we do not consider this a limitation as it has been shown in Hiptmair et al. (2015), Berggren (2010), and Zhu (2018) that the weak form is superior when the state and the adjoint equations are discretized by finite elements.

7 Replication of results

The code for the numerical experiments is available at Software used in 'Automated shape differentiation in the Unified Form Language' (2019).

Author contributions This work originated in a discussion between the four authors at the FEniCS 18 conference, where AP and DAH suggested to calculate shape derivates as in Section 2. FW implemented this idea in UFL with help from LM. The manuscript was written by AP and FW, with feedback from LM and DAH.

Funding information DAH is supported by the Natural Environment Research Council (grant no. NE/K008951/1). LM is supported by the Engineering and Physical Sciences Research Council (grant no. EP/L000407/1). FW is supported by the EPSRC Centre For Doctoral Training in Industrially Focused Mathematical Modelling (grant no. EP/L015803/1).

Compliance with ethical standards

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

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

Ahrens J, Geveci B, Law C (2005) Paraview: an end-user tool for large data visualization. The visualization handbook, 717

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):9:1–9,37. https://doi.org/10.1145/2566630

Amestoy PR, Duff IS, L'Excellent JY, Koster J (2000) Mumps: a general purpose distributed memory sparse solver. In: International workshop on applied parallel computing. Springer, pp 121–130

Berggren M (2010) A unified discrete-continuous sensitivity analysis method for shape optimization. In: Applied and numerical partial differential equations, Comput. Methods Appl. Sci., vol 15. Springer, New York, pp 25–39. https://doi.org/10.1007/978-90-481-3239-3_4

Delfour MC, Zolésio JP (2011) Shapes and geometries. Metrics, analysis, differential calculus, and optimization advances in design and control, vol 22, 2nd edn. Society for Industrial and Applied Mathematics (SIAM), Philadelphia. https://doi.org/10.1137/1.9780898719826

Geuzaine C, Remacle JF (2009) Gmsh: a 3-D finite element mesh generator with built-in pre-and post-processing facilities. Int J Numer Methods Eng 79(11):1309–1331

Hintermüller M, Ring W (2004) A second order shape optimization approach for image segmentation. SIAM J Appl Math 64(2):442–467. https://doi.org/10.1137/S0036139902403901

Hinze M, Pinnau R, Ulbrich M, Ulbrich S (2009) Optimization with PDE constraints mathematical modelling: theory and applications, vol 23. Springer, New York

Hiptmair R, Paganini A, Sargheini S (2015) Comparison of approximate shape gradients. BIT 55(2):459–485. https://doi.org/10.1007/s10543-014-0515-z

Logg A, Wells GN (2010) DOLFIN: automated finite element computing. ACM Trans Math Softw 37(2):20:1–20,28. https://doi.org/10.1145/1731022.1731030

Logg A, Mardal KA, Wells GN (eds) (2012) Automated solution of differential equations by the finite element method: the FEniCS book. Springer

Rathgeber F, Ham DA, Mitchell L, Lange M, Luporini F, McRae ATT, Bercea GT, Markall GR, Kelly PHJ (2016) Firedrake: automating the finite element method by composing abstractions. ACM Trans Math Softw 43(3):24:1–24,27. https://doi.org/10.1145/2998441

Schmidt S (2010) Efficient large scale aerodynamic design based on shape calculus. Ph.D. thesis, Universität Trier

Schmidt S (2018) Weak and strong form shape Hessians and their automatic generation. SIAM J Sci Comput 40(2):C210–C233.

https://doi.org/10.1137/16M1099972. Software freely available at https://bitbucket.org/Epoxid/femorph

Schmidt S, Schütte M, Walther A (2018) Efficient numerical solution of geometric inverse problems involving Maxwell's equations using shape derivatives and automatic code generation. SIAM J Sci Comput 40(2):B405–B428. https://doi.org/10.1137/16M110602X

Simon J (1980) Differentiation with respect to the domain in boundary value problems. Numer Funct Anal Optim 2(7-8):649–687. https://doi.org/10.1080/01630563.1980.10120631

Software used in 'Automated shape differentiation in the Unified Form Language' (2019). https://doi.org/10.5281/zenodo.2621254

Zhu S (2018) Effective shape optimization of laplace eigenvalue problems using domain expressions of Eulerian derivatives. J Optim Theory Appl 176(1):17–34. https://doi.org/10.1007/s10957-017-1198-9

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.