Structural and Multidisciplinary Optimization (2021) 63:1579–1607
https://doi.org/10.1007/s00158-020-02742-w

EDUCATIONAL PAPER

Fully and semi-automated shape differentiation in `NGSolve`

Peter Gangl[1] · Kevin Sturm[2] · Michael Neunteufel[2] · Joachim Schöberl[2]

Received: 14 April 2020 / Revised: 10 September 2020 / Accepted: 11 September 2020

Abstract
In this paper, we present a framework for automated shape differentiation in the finite element software `NGSolve`. Our approach combines the mathematical Lagrangian approach for differentiating PDE-constrained shape functions with the automated differentiation capabilities of `NGSolve`. The user can decide which degree of automatisation is required, thus allowing for either a more custom-like or black-box–like behaviour of the software. We discuss the automatic generation of first- and second-order shape derivatives for unconstrained model problems as well as for more realistic problems that are constrained by different types of partial differential equations. We consider linear as well as nonlinear problems and also problems which are posed on surfaces. In numerical experiments, we verify the accuracy of the computed derivatives via a Taylor test. Finally, we present first- and second-order shape optimisation algorithms and illustrate them for several numerical optimisation examples ranging from nonlinear elasticity to Maxwell's equations.

Keywords Shape optimisation · Shape derivative · Automated differentiation · Shape Newton method

1 Introduction

Numerical simulation and shape optimisation tools to solve the problems have become an integral part in the design process of many products. Starting out from an initial design, non-parametric shape optimisation techniques based on first- and second-order shape derivatives can assist in finding shapes of a product which are optimal with respect to a given objective function. Examples include the optimal design of aircrafts (Schmidt et al. 2013; 2011), optimal inductor design (Hömberg and Sokolowski 2003), optimisation of microlenses (Paganini et al. 2015), the optimal design of electric motors (Gangl et al. 2015), applications to mechanical engineering (Allaire et al. 2004; Laurain 2018), multiphysics problems (Feppon et al. 2019), or electrical impedance tomography (EIT) in medical sciences to name only a few (Hintermüller and Laurain 2008).

Shape optimisation algorithms are based on the concept of shape derivatives. Let $\mathcal{P}(\mathbf{R}^d)$ denote the set of all subsets of $\mathbf{R}^d$. Furthermore, let $\mathcal{A} \subset \mathcal{P}(\mathbf{R}^d)$ be a set of admissible shapes and $\mathcal{J} : \mathcal{A} \to \mathbf{R}$ be a shape function. Given an admissible shape $\Omega \in \mathcal{A}$ and a sufficiently smooth vector field V, we define the perturbed domain $\Omega_t := (\mathrm{Id} + tV)(\Omega)$ for a small perturbation parameter $t > 0$. The shape derivative is defined as:

$$D\mathcal{J}(\Omega)(V) := \left(\frac{d}{dt}\mathcal{J}(\Omega_t)\right)\Big|_{t=0} = \lim_{t\to 0} \frac{\mathcal{J}(\Omega_t) - \mathcal{J}(\Omega)}{t}. \quad (1)$$

Remark 1 We remark that a frequently used definition of shape differentiability is to require the mapping $V \mapsto J((\mathrm{Id} + V)(\Omega))$ being Fréchet differentiable in $V = 0$; see (Allaire 2007; Henrot and Pierre 2005; Murat and Simon 1976). This stronger notion of differentiability implies that the limit defined in (1) exists.

In most practically relevant applications, the objective functional depends on the shape of a (sub-)domain via the solution to a partial differential equation (PDE). Thus, one

Responsible Editor: Gregoire Allaire

✉ Peter Gangl
gangl@math.tugraz.at

[1] TU Graz, Steyrergasse 30, 8010 Graz, Austria

[2] TU Wien, Wiedner Hauptstr. 8-10, 1040 Vienna, Austria

is facing a problem of PDE-constrained shape optimisation of the form:

$$\begin{aligned} &\min_{(\Omega,u)\in\mathcal{A}\times Y} J(\Omega,u) \\ &\text{s.t. } (\Omega,u)\in\mathcal{A}\times Y : e(\Omega;u,v)=0 \qquad \text{for all } v\in Y. \end{aligned} \tag{2}$$

Here, the second line represents the constraining boundary value problem posed on a Hilbert space Y, which we assume to be uniquely solvable for all admissible $\Omega \in \mathcal{A}$. Denoting the unique solution for a given $\Omega \in \mathcal{A}$ by u_Ω, we introduce the notation for the reduced functional:

$$\mathcal{J}(\Omega) := J(\Omega, u_\Omega).$$

In order to be able to apply a shape optimisation algorithm to a given problem of this kind, the shape derivative (1) has to be computed; see the standard literature Delfour and Zolésio (2011a) and Sokołowski and Zolésio (1992) or Sturm (2015a) for an overview of different approaches. In the following, we focus on computing the so-called volume form of the shape derivative which in a finite element context is known to give a better approximation compared to the boundary form; see Hiptmair et al. (2015) and Berggren (2010).

The convergence of shape optimisation algorithms can be speeded up by using second-order shape derivatives. Given two sufficiently smooth vector fields V, W, and an admissible shape $\Omega \in \mathcal{A}$, let $\Omega_{s,t} := (\mathrm{Id} + sV + tW)(\Omega)$ be the perturbed domain. Then, the second-order shape derivative is defined as:

$$D^2\mathcal{J}(\Omega)(V)(W) := \left(\frac{d^2}{dsdt}\mathcal{J}(\Omega_{s,t})\right)\Big|_{s,t=0}. \tag{3}$$

Second-order information in Newton-type algorithms has been explored in the articles Novruzi and Roche (2000), Allaire et al. (2016), Paganini and Sturm (2019), Eppler et al. (2007), and Schulz (2014). Since the computation of second-order shape derivatives is more involved and error prone, several authors have employed automatic differentiation (AD) tools; see e.g. Schmidt (2018) and Ham et al. (2019) for two approaches based on the Unified Form Language (UFL) (Alnæs et al. 2014). In Ham et al. (2019), the authors present a fully automated shape differentiation software which uses the transformation properties on the finite element level. In Schmidt (2018) (see also the earlier work (Schmidt 2014)), the automated derivatives are computed using UFL. The strategies of Ham et al. (2019) and Schmidt (2018) differ in that, for the latter, the software computes an unsymmetric shape Hessian since it involves the term $D\mathcal{J}(\Omega)(\partial V W)$. Optionally, the software allows to make the shape Hessian symmetric by requiring $\partial V W = 0$. We will discuss the subtle difference and the relation between the two possible ways of defining shape Hessians in Remark 3 of Section 3.2. Let us also mention Dokken et al. (2020) where automated shape derivatives for transient PDEs in FEniCS and Firedrake are presented.

In this paper, we present an alternative framework for AD of PDE-constrained problems of type (2). There exist several approaches for the rigorous derivation of the shape derivative of PDE-constrained shape functionals; see Sturm (2015b) for an overview. The main idea, however, is always similar. After transforming the perturbed setting back to the original domain, shape differentiation in the direction of a given vector field reduces to the differentiation with respect to the scalar parameter t which now enters via the corresponding transformation and its gradient. It is shown in Sturm (2015a) that the shape derivative for a nonlinear PDE-constrained shape optimisation problem can be computed as the derivative of the Lagrangian with respect to the perturbation parameter. We will illustrate this systematic procedure for a number of different applications and utilise symbolic differentiation provided by the finite element software package `NGSolve` (Schöberl 2014) to obtain the shape derivative for different classes of PDE-constrained optimisation problems. `NGSolve` allows for the fast and efficient numerical solution of a large number of different boundary value problems. The aim of this paper is to extend `NGSolve` by the possibility of semi-automatic and fully automatic shape differentiation and optimisation.

Distinctly from previous approaches, we cover the following two points:

– A fully automated setting requiring as input the weak formulation of the constraint and the cost function,
– A semi-automated setting which offers a highly customisable user interface, but requires mathematical background knowledge.

Structure of the paper In Section 2, we give a brief introduction on how to solve a PDE in `NGSolve` and present its built-in auto-differentiation capabilities. The introduced syntax will also lay the foundation for the following sections. In Section 3, we present a first unconstrained shape optimisation problem and show how to solve it in `NGSolve`. For this purpose, we show how to compute the first- and second-order shape derivative in a semi-automated way. Section 4 extends the preceding section by incorporating a PDE constraint. The strategy is illustrated by means of a simple Poisson equation. We also show how to treat the computation of shape derivatives when the PDE is defined on surfaces. While the semi-automated shape differentiation presented in Sections 3 and 4 requires mathematical background knowledge, in

Section 5 we show how the shape derivatives can be computed in a fully automated fashion. In the last section of the paper, we verify the computed formulas by a Taylor test, discuss optimisation algorithms and present several numerical optimisation examples including nonlinear elasticity, Maxwell's equations and Helmholtz's equation.

2 A brief introduction to NGSolve

In this section, we give a brief overview of the main concepts of the finite element software NGSolve (Schöberl 2014). We first describe the main principles for numerically solving boundary value problems in NGSolve before focusing on its built-in automatic differentiation capabilities. In the subsequent sections of this paper, these ingredients will be combined to implement the shape derivative of unconstrained and PDE-constrained shape optimisation problems in an automated way.

2.1 Solving PDEs with finite elements in NGSolve

In this section, we illustrate the syntax of NGSolve using the python programming language for the Poisson equation with homogeneous Dirichlet conditions as a model problem. We refer the reader to the online documentation

https://ngsolve.org/docu/latest/

for a more detailed description of the many features of this package.

Given a domain $\Omega \subset \mathbf{R}^d$ and a right-hand side f, we consider the model problem to find u satisfying:

$$\begin{aligned} -\Delta u &= f && \text{in } \Omega, \\ u &= 0 && \text{on } \partial\Omega. \end{aligned}$$

The weak form of the model problem reads:

$$\text{Find } u \in H_0^1(\Omega) : \int_\Omega \nabla u \cdot \nabla w \, \mathrm{dx} = \int_\Omega f w \, \mathrm{dx} \quad \forall w \in H_0^1(\Omega). \tag{4}$$

We consider a ball of radius $\frac{1}{2}$ in two space dimensions centred at the point $(0.5, 0.5)^\top$, i.e. $\Omega = B((0.5, 0.5)^\top, 0.5)$, and the right-hand side is defined by $f(x_1, x_2) = 2x_2(1 - x_2) + 2x_1(1 - x_1)$. We will go through the steps for numerically solving this problem by the finite element method.

We begin by importing the necessary functionalities and setting up a finite element mesh.

```
1 from ngsolve import *
2 from netgen.geom2d import SplineGeometry
3
4 geo = SplineGeometry()
5 geo.AddCircle((0.5,0.5),0.5,bc="circle")
6
7 mesh = Mesh(geo.GenerateMesh(maxh=0.2))
8 mesh.Curve(3)
```

The first line imports all modules from the package NGSolve. The second line includes the SplineGeometry function which enables us to define a mesh via a geometric description, in our case a circle centred at $(0.5, 0.5)^\top$ of radius 0.5. Finally, the mesh is generated in line 7, and in line 8 we specify that we want to use a curved finite element mesh for a more accurate approximation of the geometry. For that purpose, a projection-based interpolation procedure is used, see e.g. (Demkowicz 2004).

Next in line 9 we define an H^1 conforming finite element space of polynomial degree 3 and include Dirichlet boundary conditions on the boundary of the domain $\partial\Omega$ (referenced by the string ''circle'' that we assigned in line 5). On this space, we define a trial function u in line 11 and a test function w in line 12. These are purely symbolic objects which are used to define boundary value problems in weak form.

```
9  fes = H1(mesh, order=3, dirichlet="circle")
10
11 u = fes.TrialFunction()
12 w = fes.TestFunction()
```

For a more compact presentation later on, we define a coefficient function X which combines the three spatial components:

```
13 X = CoefficientFunction((x,y,z))
```

Now, the left- and right-hand sides of problem (4) can be conveniently defined as a bilinear or linear form, respectively, on the finite element space fes by the following lines.

```
14 L = LinearForm(fes)
15 f1 = (2*X[1]*(1-X[1])+2*X[0]*(1-X[0]))
16 L += f1 * w * dx
17
18 a = BilinearForm(fes, symmetric=True)
19 a += grad(u)*grad(w)*dx
```

We assemble the system matrix coming from the bilinear form a and the load vector coming from L and solve the corresponding system of linear equations.

```
a.Assemble()
L.Assemble()

gfu = GridFunction(fes)
gfu.vec.data = a.mat.Inverse(fes.FreeDofs(),
    inverse="sparsecholesky") * L.vec

Draw (gfu, mesh, "state")
```

Here, gfu is defined as a GridFunction over the finite element space fes. A GridFunction object is used to save the results by containing the corresponding finite element coefficient vectors. Furthermore, it can evaluate the stored finite element solution at a given mesh point. The Dirichlet conditions are incorporated into the direct solution of the linear system and the numerical solution is drawn in the graphical user interface. The numerical solution is depicted in Fig. 1.

2.2 Automatic differentiation in NGSolve

In NGSolve, symbolic expressions are stored in expression trees; see Fig. 2 for an example. It is possible to differentiate an expression *expr* with respect to a variable *var* appearing in *expr* into a direction *dir* by the command
expr.Diff(var, dir).

Mathematically, this line corresponds to the directional derivative of g:=*expr* at $x := var$ in direction $v := dir$, that is,

$$Dg(x)(v). \tag{5}$$

When calling the Diff command for expr, the expression tree of *expr* is gone through node by node, and for each node the corresponding differentiation rules such as product rule or chain rule are applied. When a node represents the variable with respect to which the differentiation is carried out, it is replaced by the direction *dir* of differentiation.

Figure 2 shows the differentiation of the expression *expr= 2x*x+3y* with respect to *x* into the direction given by *v*:

```
v = Parameter(1)
expr = 2*x*x+3*y
dexpr = expr.Diff(x,v)
print(expr)
print(dexpr)
```

The output of print(expr) reads:

```
coef binary operation '+', real
  coef binary operation '*', real
    coef scale 2, real
      coef coordinate x, real
    coef coordinate x, real
  coef scale 3, real
    coef coordinate y, real
```

which translates to $2x * x + 3y$ and corresponds to the expression tree depicted in Fig. 2a. The output of print(dexpr) reads:

```
coef binary operation '+', real
  coef binary operation '+', real
    coef binary operation '*', real
      coef scale 2, real
        coef N5ngfem28ParameterCoefficient
        FunctionE, real
      coef coordinate x, real
    coef binary operation '*', real
      coef scale 2, real
        coef coordinate x, real
      coef N5ngfem28ParameterCoefficient
      FunctionE, real
  coef scale 3, real
    coef 0, real
```

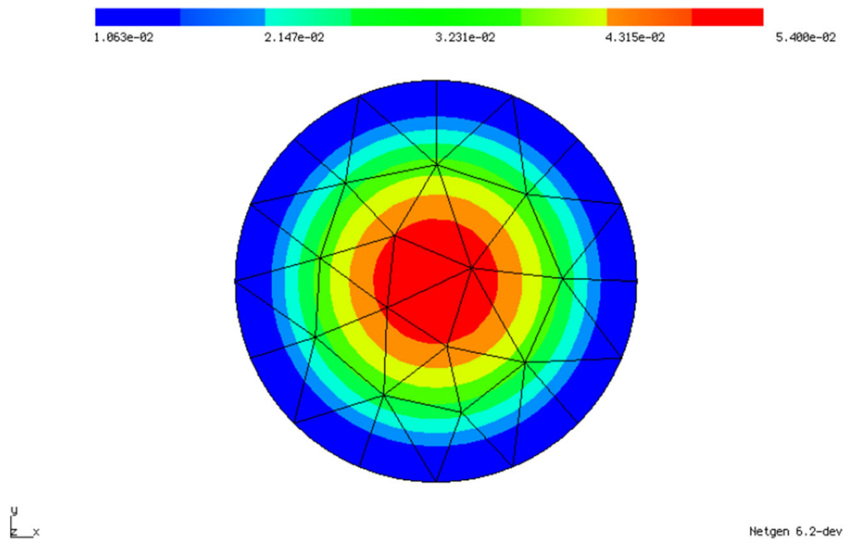


Fig. 1 Solution of problem (4) by code fragments of Section 2.1 with 29 nodes, 40 (curved) triangular elements and polynomial order 3

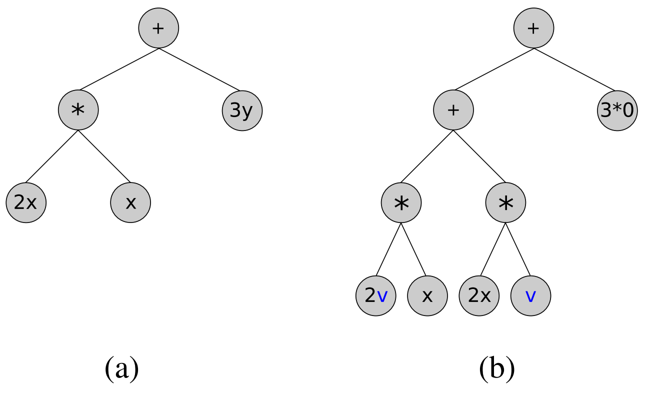


Fig. 2 Illustration of `Diff` command for example *expr= 2x*x+3y*. **a** Expression tree for *expr*. **b** Expression tree for expression obtained by call of `expr.Diff(x, v)`

which translates to $(2v*x+2x*v)+3*0$ and corresponds to the expression tree depicted in Fig. 2b. The coefficient

```
N5ngfem28ParameterCoefficientFunctionE
```

appearing therein is the C++ internal class name of the Python object `Parameter`.

`NGSolve` trial and test functions are purely symbolic objects used for defining bilinear and linear forms. Therefore, they do not depend on the spatial variables x, y, z as can be seen by differentiating them. `NGSolve` `GridFunctions` on the other hand represent functions in the finite element space. However, also for these objects, the space dependency is omitted when performing symbolic differentiation. The code segments

```python
u = fes.TrialFunction()  # symbolic object
w = fes.TestFunction()   # symbolic object
gf = GridFunction(fes)
gf.Set(x*x*y)

print("Diff u w.r.t. x", u.Diff(x))
print("Diff w w.r.t. x", w.Diff(x))
print("Diff gf w.r.t. x", gf.Diff(x))
```

will give the following output:

```
Diff u w.r.t. x:  ConstantCF, val = 0
Diff w w.r.t. x:  ConstantCF, val = 0
Diff gf w.r.t. x:  ConstantCF, val = 0
```

Here, the `GridFunction.Set` method takes a `CoefficientFunction` object and performs a (local) L^2 best approximation into the underlying finite element space with respect to its natural norm and stores the resulting coefficient vector.

3 Semi-automatic shape differentiation without constraints

We will illustrate the steps to be taken in order to obtain the shape derivative of a shape function in a semi-automatic way for a simple shape optimisation problem. For $\Omega \subset \mathbf{R}^d$ bounded and open and a continuously differentiable function $f \in C^1(\mathbf{R}^d)$, we consider the shape differentiation of the shape function:

$$\mathcal{J}(\Omega) = \int_\Omega f(x)\,\mathrm{d}x. \tag{6}$$

Clearly, the minimiser of $\mathcal{J}$ over all measurable sets in $\mathbf{R}^d$ is given by $\Omega^* = \{x \in \mathbf{R}^d : f(x) < 0\}$. We also refer to Schiela and Ortiz (2017) for the computations of first- and second-order variations of functions of type (6) where Ω is a submanifold of $\mathbf{R}^d$.

3.1 First-order shape derivative

Henceforth, we denote by $C^{0,1}(\mathbf{R}^d)^d$ the space of bounded and Lipschitz continuous vector fields $V : \mathbf{R}^d \to \mathbf{R}^d$. In view of Rademachers' theorem (Evans 2010, Thm.6, p. 296), the space $C^{0,1}(\mathbf{R}^d)^d$ corresponds to the Sobolev space $W^{1,\infty}(\mathbf{R}^d)^d$.

Given a vector field $V \in C^{0,1}(\mathbf{R}^d)^d$, we define the transformation:

$$T_t(x) := (\mathrm{Id} + t\,V)(x), \quad x \in \mathbf{R}^d, \quad t \geq 0.$$

Definition 1 The first-order shape derivative of a shape function $\mathcal{J}$ at Ω in direction $V \in C^{0,1}(\mathbf{R}^d)^d$ is defined by:

$$D\mathcal{J}(\Omega)(V) = \lim_{t\to 0} \frac{\mathcal{J}(T_t(\Omega)) - \mathcal{J}(\Omega)}{t}. \tag{7}$$

3.1.1 Shape differentiation of unconstrained volume integrals

Using the transformation $y = T_t(x)$ and the notation $F_t := \partial T_t = I + t\partial V$ for the Jacobian of the transformation T_t, we get for $\mathcal{J}$ as in (6):

$$\mathcal{J}(\Omega_t) = \int_{\Omega_t} f(x')\,\mathrm{d}x' = \int_\Omega (f\circ T_t)(x)\,\det(F_t(x))\,\mathrm{d}x. \tag{8}$$

Now, let us explain how to compute the shape derivative of $\mathcal{J}$. Denoting

$$G(T_t, F_t) := \int_\Omega (f\circ T_t)(x)\,\det(F_t(x))\,\mathrm{d}x, \tag{9}$$

the chain rule gives (formally)

$$\begin{aligned}\frac{d}{dt}\mathcal{J}(\Omega_t)\Big|_{t=0} &= \tfrac{d}{dt}G(T_t, F_t)\big|_{t=0}\\ &= \left(\frac{dG}{dT_t}\frac{dT_t}{dt} + \frac{dG}{dF_t}\frac{dF_t}{dt}\right)\Big|_{t=0}.\end{aligned} \tag{10}$$

Using that $\frac{dT_t}{dt}(x) = V(x)$ and $\frac{dF_t}{dt}(x) = \partial V(x)$, we get for the shape derivative:

$$D\mathcal{J}(\Omega)(V) = \frac{d}{dt}\mathcal{J}(\Omega_t)\bigg|_{t=0} = \left(\frac{dG}{dT_t}V + \frac{dG}{dF_t}\partial V\right)\bigg|_{t=0}.$$

This is the form we use for defining the first-order shape derivative in `NGSolve`. Note that a Lipschitz vector field is differentiable almost everywhere and hence $\partial V(x)$ is defined almost everywhere and bounded.

Given the function $f(x_1, x_2) = (x_1 - 0.5)^2/a^2 + (x_2 - 0.5)^2/b^2 - R^2$ with $a = 1.3$, $b = 1/a$ and $R = 0.5$, we implement the transformed cost function (8) as follows:

```
40 f = ((X[0]-0.5)/1.3)**2+(1.3*(X[1]-0.5))**2 -
       0.5**2
41
42 F = Id(2)                  # symbolic identity
       matrix
43 G_f = f * Det(F) * dx      # F only acts as a
       dummy variable
```

Here, we introduce the symbol `F` and assign to it the value of the identity matrix in line 42. This allows us to differentiate with respect to `F`. Then, we define the function G of (9) in line 43. The shape derivative is a bounded linear functional on a space of vector fields. We introduce a vector-valued finite element space `VEC` and define the object representing the shape derivative `dJOmega_f` as a linear functional on `VEC`. In line 48, we differentiate with respect to the spatial variables in the direction given by `V`. Note that `X` is the coefficient function we introduced in line 13. In line 49, we deal with the differentiation with respect to `F`.

Remark 2 Defining $\xi_t := \det(F_t)$ and using $\frac{d}{dt}\xi_t|_{t=0} = \operatorname{div} V$, it holds:

$$\frac{dG}{dF_t}\frac{dF_t}{dt}\bigg|_{t=0} = \frac{dG}{d\xi_t}\frac{d\xi_t}{dF_t}\frac{dF_t}{dt}\bigg|_{t=0} = \frac{dG}{d\xi_t}\frac{d\xi_t}{dt}\bigg|_{t=0}$$
$$= \frac{dG}{d\xi_t}\operatorname{div} V\bigg|_{t=0} = \int_\Omega f \operatorname{div} V \,\mathrm{d}x.$$

Therefore, we obtain for the first-order shape derivative the well-known formula:

$$D\mathcal{J}(\Omega)(V) = \int_\Omega \nabla f \cdot V + f \operatorname{div} V \,\mathrm{d}x. \tag{11}$$

Finally, if Ω is smooth enough (for instance C^1), it follows by integration by parts in (11) that the shape derivative is given by:

$$D\mathcal{J}(\Omega)(V) = \int_{\partial\Omega} f V \cdot n \,\mathrm{d}s, \tag{12}$$

where n denotes the outward pointing normal along $\partial\Omega$.

```
44 VEC = VectorH1(mesh, order=1, dirichlet="") #
       vectorial FE space of order 1
45 V = VEC.TestFunction()
46
47 dJOmega_f = LinearForm(VEC)
48 dJOmega_f += G_f.Diff(X, V)
49 dJOmega_f += G_f.Diff(F, grad(V))
```

3.1.2 Shape differentiation of unconstrained boundary integrals

For Ω and f as in the previous section, we consider:

$$\mathcal{J}_{bnd}(\Omega) = \int_{\partial\Omega} f(x) \,\mathrm{d}x. \tag{13}$$

Then, we get:

$$\mathcal{J}_{bnd}(\Omega_t) = \int_{\partial\Omega_t} f(x') \,\mathrm{d}s_{x'} \tag{14}$$
$$= \int_{\partial\Omega} (f \circ T_t)(x) \det(F_t(x)) |F_t(x)^{-\top} n(x)| \,\mathrm{d}s_x, \tag{15}$$

see e.g. (Sokołowski and Zolésio 1992, Prop. 2.47), with the outer unit normal vector n and $|\cdot|$ denoting the Euclidean norm. It is shown in (Sokołowski and Zolésio 1992, Prop. 2.50) that the shape derivative of (13) is given by:

$$D\mathcal{J}_{bnd}(\Omega)(V) = \int_{\partial\Omega} \nabla f \cdot V + f(\operatorname{div} V - n^\top \partial V n) ds_x.$$

Again, we can compute the shape derivative in `NGSolve` as the total derivative of expression (15) with respect to the parameter t. In `NGSolve`, the only difference lies in the necessity to use the trace of the gradient of a test vector field `V`.

```
50 G_f_bnd = f * Det(F) * Norm( Inv(F).trans*
       specialcf.normal(2) ) * ds
51
52 dJOmega_f_bnd = LinearForm(VEC)
53 dJOmega_f_bnd+=G_f_bnd.Diff(X, V)
                  # no trace needed
54 dJOmega_f_bnd+=G_f_bnd.Diff(F,grad(V).Trace())
                  # trace needed
```

Note that the trace operator for gradients on the boundary is obligatory in `NGSolve`, whereas for direct evaluation of H^1 trial and test functions itself it is optional.

3.2 Second-order shape derivatives

For second-order shape derivatives, we consider perturbations of the form:

$$T_{s,t}(x) = (\mathrm{Id} + sV + tW)(x), \quad x \in \mathbf{R}^d,$$

for $s, t \geq 0$ and define $\Omega_{s,t} := T_{s,t}(\Omega)$.

Definition 2 The second-order shape derivative of a shape function $\mathcal{J}$ at Ω in direction $(V, W) \in C^{0,1}(\mathbf{R}^d)^d \times C^{0,1}(\mathbf{R}^d)^d$ is defined by:

$$D^2\mathcal{J}(\Omega)(V)(W) = \frac{d^2}{dsdt}\mathcal{J}(\Omega_{s,t})\Big|_{s=t=0}. \tag{16}$$

Remark 3 We remark that if $\mathcal{J}$ is smooth enough, the second-order derivative as defined in (16) is symmetric by definition:

$$D^2\mathcal{J}(\Omega)(V)(W) = D^2\mathcal{J}(\Omega)(W)(V). \tag{17}$$

We stress that this derivative is not the same as the shape derivative obtained by repeated shape differentiation, that is, it does not coincide with (see, e.g., (Delfour and Zolésio 2011b, Chap. 9, Sec. 6)):

$$d^2\mathcal{J}(\Omega)(V)(W) := \lim_{t\to 0}\frac{D\mathcal{J}(T_t^W(\Omega))(V) - D\mathcal{J}(\Omega)(V)}{t} \tag{18}$$

which is in general asymmetric.

The derivative defined in (18) is only symmetric if $\partial V W = 0$ since it holds:

$$d^2\mathcal{J}(\Omega)(V)(W) = D^2\mathcal{J}(\Omega)(V)(W) + D\mathcal{J}(\Omega)(\partial V W), \tag{19}$$

see also the early work of Simon (1989) on this topic. However, in NGSolve, when repeating the shape differentiation procedure introduced in Section 3.1, we compute directly the second-order shape derivative as defined in (16). Here, we exploit the fact that trial functions are independent of the spatial coordinates; see also Section 2.2 and the example below.

Let us now exemplify the computation of the second-order shape derivative for the shape function $\mathcal{J}$ defined in (6). Similarly to the computations of the first derivative, we use the notation $F_{s,t} := \partial T_{s,t} = I + s\partial V + t\partial W$. Then, we get:

$$\begin{aligned}\frac{d^2}{dsdt}\mathcal{J}(\Omega_{s,t})\Big|_{s=t=0} &= \frac{d^2}{dsdt}\int_{\Omega_{s,t}} f(x)\,\mathrm{d}x\Big|_{s=t=0}\\ &= \frac{d^2}{dsdt}\int_{\Omega}(f\circ T_{s,t})(x)\ \det(F_{s,t}(x))\,\mathrm{d}x\Big|_{s=t=0}.\end{aligned}$$

Again, using the notation

$$G(T_{s,t}, F_{s,t}) = \int_{\Omega}(f\circ T_{s,t})(x)\ \det(F_{s,t}(x))\,\mathrm{d}x,$$

we get

$$\begin{aligned}\frac{d^2}{dsdt}\mathcal{J}(\Omega_{s,t})\Big|_{s=t=0} &= \tfrac{d^2}{dsdt}G(T_{s,t}, F_{s,t})\Big|_{s=t=0}\\ &= \tfrac{d}{ds}\left(\tfrac{dG}{dT_{s,t}}\tfrac{dT_{s,t}}{dt} + \tfrac{dG}{dF_{s,t}}\tfrac{dF_{s,t}}{dt}\right)\Big|_{s=t=0}.\end{aligned}$$

Using that $\frac{d^2T_{s,t}}{dsdt} = 0$ and $\frac{d^2F_{s,t}}{dsdt} = 0$, we get further:

$$\begin{aligned}\frac{d^2}{dsdt}\ \mathcal{J}(\Omega_{s,t})\Big|_{s=t=0} &= \tfrac{d}{ds}\left(\tfrac{dG}{dT_{s,t}}\right)\tfrac{dT_{s,t}}{dt} + \tfrac{d}{ds}\left(\tfrac{dG}{dF_{s,t}}\right)\tfrac{dF_{s,t}}{dt}\Big|_{s=t=0}\\ &= \left(\tfrac{d^2G}{dT_{s,t}^2}\tfrac{dT_{s,t}}{ds} + \tfrac{d^2G}{dF_{s,t}dT_{s,t}}\tfrac{dF_{s,t}}{ds}\right)\tfrac{dT_{s,t}}{dt}\\ &\quad + \left(\tfrac{d^2G}{dT_{s,t}dF_{s,t}}\tfrac{dT_{s,t}}{ds} + \tfrac{d^2G}{dF_{s,t}^2}\tfrac{dF_{s,t}}{ds}\right)\tfrac{dF_{s,t}}{dt}\Big|_{s=t=0}.\end{aligned} \tag{20}$$

Formula (20) is used for the automatic derivation of the second-order shape derivative in NGSolve. Using $\frac{dT_{s,t}}{ds}(x) = V(x)$, $\frac{dT_{s,t}}{dt}(x) = W(x)$ and $\frac{dF_{s,t}}{ds}(x) = \partial V(x)$, $\frac{dF_{s,t}}{dt}(x) = \partial W(x)$, we get:

$$\begin{aligned}\frac{d^2}{dsdt}\mathcal{J}(\Omega_{s,t})\Big|_{s=t=0} &= \left(\frac{d^2G}{dT_{s,t}^2}V + \frac{d^2G}{dF_{s,t}dT_{s,t}}\partial V\right)W\\ &\quad + \left(\frac{d^2G}{dT_{s,t}dF_{s,t}}V + \frac{d^2G}{dF_{s,t}^2}\partial V\right)\partial W\Big|_{s=t=0}\end{aligned} \tag{21}$$

Remark 4 We remark that the formula (21) can be evaluated explicitly and reads:

$$\begin{aligned}D^2\mathcal{J}(\Omega)(V, W) = &\int_{\Omega}\nabla^2 fV\cdot W + \nabla f\cdot W\,\mathrm{div}V + \nabla f\cdot V\,\mathrm{div}W\\ &+ f\,\mathrm{div}V\,\mathrm{div}W - f\partial V^{\top} : \partial W\,\mathrm{d}x.\end{aligned}$$

Formula (21) can be implemented in NGSolve as follows:

```
55 d2JOmega_f = BilinearForm(VEC)
56 W = VEC.TrialFunction()
57
58 d2JOmega_f+=(G_f.Diff(X,W)+G_f.Diff(F,grad(W))
       ).Diff(X,V)f
59 d2JOmega_f+=(G_f.Diff(X,W)+G_f.Diff(F,grad(W))
       ).Diff(F,grad(V))
```

Notice that since `W` is a trial function, it is not affected by the differentiation with respect to `X`; see Section 2.2. Therefore, the terms coming from differentiating `W` with respect to the spatial coordinates `X` into the direction of `V` disappear and thus, although code lines 58–59 look like the "derivative of the derivative", we actually compute formula (16) and not (18).

In the same fashion, second-order derivatives of boundary integrals of the form (13) can be computed.

```
60 d2JOmega_f_bnd = BilinearForm(VEC)
61
62 d2JOmega_f_bnd+=(G_f_bnd.Diff(X, W)+G_f_bnd.
       Diff(F,grad(W).Trace())).Diff(X, V)
63 d2JOmega_f_bnd+=(G_f_bnd.Diff(X, W)+G_f_bnd.
       Diff(F,grad(W).Trace())).Diff(F,grad(V).
       Trace())
```

Again note that the trace operator is necessary when dealing with gradients on the boundary.

4 Semi-automatic shape differentiation with PDE constraints

In this section, we describe the automatic computation of the shape derivative for the following type of equality-constrained shape optimisation problems:

$$\min_{(\Omega,u)} J(\Omega, u) \tag{22}$$

subject to $(\Omega, u) \in \mathcal{A} \times Y$ solves

$$e(\Omega, u) = 0, \tag{23}$$

where $e : \mathcal{A} \times Y \to Y^*$ with $e(\Omega, \cdot) : Y(\Omega) \to Y(\Omega)^*$ represents an abstract PDE constraint with $Y = \cup_{\Omega\in\mathcal{A}} Y(\Omega)$ being the union of Banach spaces $Y(\Omega)$ and $\mathcal{A}$ a set of admissible shapes. For any given $\Omega \in \mathcal{A}$, we assume the PDE constraint (23) to admit a unique solution which we denote by u_Ω. Moreover, let $\mathcal{J}(\Omega) := J(\Omega, u_\Omega)$ denote the reduced cost functional. By introducing a Lagrangian function, we can henceforth deal with an unconstrained shape function $\mathcal{L}$ rather than a shape function $\mathcal{J}$ and a PDE constraint. We introduce the Lagrangian:

$$\mathcal{L}(\Omega, u, p) := J(\Omega, u) + \langle e(\Omega, u), p\rangle. \tag{24}$$

Now, an initial shape Ω is perturbed by a family of transformations T_t, resulting in a new shape $\Omega_t := T_t(\Omega)$. Transforming back to the initial shape Ω leads to the Lagrangian:

$$G(t, u, p) := \mathcal{L}(T_t(\Omega), \Phi_t(u), \Phi_t(p)), \quad u, p \in Y(\Omega), \tag{25}$$

where $\Phi_t : Y(\Omega) \to Y(\Omega_t)$ is a bijective mapping. Here, the transformation Φ_t depends on the differential operator involved. For instance:

- If $Y(\Omega) = H^1_0(\Omega)$, then $\Phi_t(u) = u \circ T_t^{-1}$,
- If $Y(\Omega) = H(\mathbf{curl}, \Omega)$, then $\Phi_t(u) = \partial T_t^{-\top}(u \circ T_t^{-1})$,
- If $Y(\Omega) = H(\mathrm{div}, \Omega)$, then $\Phi_t(u) = \frac{1}{\det(\partial T_t)}\partial T_t(u \circ T_t^{-1})$.

Intuitively, the transformations Φ_t are chosen in such a way that the transformed function $\Phi_t(u)$ still belongs to the same space, but on a different domain. For the above three examples, this essentially requires to check how the differential operators ∇, **curl** and div transform under the change of variables T_t, respectively. In fact, one can check that:

$$\begin{aligned}
(\nabla u) \circ T_t &= \partial T_t^{-\top}\nabla(u \circ T_t), \quad u \in H^1_0(\Omega),\\
(\mathbf{curl} u) \circ T_t &= \frac{1}{\xi(t)}\partial T_t \mathbf{curl}\left(\partial T_t^{\top}(u \circ T_t)\right), \quad u \in H(\mathbf{curl}, \Omega),\\
(\mathrm{div} u) \circ T_t &= \frac{1}{\xi(t)}\mathrm{div}\left(\xi(t)\partial T_t^{-1}(u \circ T_t)\right), \quad u \in H(\mathrm{div}, \Omega),
\end{aligned}$$

where $\xi(t) := \det(\partial T_t)$; see also (Monk 2003, Section 3.9). The transformation rules are precisely given by the respective Φ_t. We also note that for smooth functions this can be checked by direct computation.

Now the shape differentiability of (22–23) is reduced to proving that (see Sturm (2015b)):

$$D\mathcal{J}(\Omega)(V) = \frac{d}{dt}G(t, u^t, 0)|_{t=0} = \partial_t G(0, u, p), \tag{26}$$

where $u^t := u_t \circ T_t$ and $u_t \in Y(\Omega_t)$ solves $e(\Omega_t, u_t) = 0$ and p is the solution to the adjoint equation:

$$p \in Y(\Omega), \quad \partial_u G(0, u, p)(\varphi) = 0 \quad \text{for all } \varphi \in Y(\Omega). \tag{27}$$

We stress that the choice of p as the solution of the adjoint equation is important in order for the second equality in (26) to hold. The verification of this equality depends on the specific PDE under consideration and can be accomplished by different methods. We refer the reader to Sturm (2015b) for an overview and remark that (26) holds for a large class of nonlinear PDE-constrained shape optimisation problems; see Sturm (2015a).

The rest of this section is organised as follows: We introduce a model problem, which is the minimisation of a tracking-type cost functional subject to Poisson's equation in Section 4.1. We illustrate how the first- and second-order shape derivative for this PDE-constrained model problem can be obtained in `NGSolve` in Sections 4.2 and 4.3. Finally, we also briefly discuss the extension to partial differentiation equations on surfaces.

4.1 PDE-constrained model problem

We will illustrate the derivation of the first- and second-order shape derivative for the minimisation of a tracking-type cost functional subject to Poisson's equation on the unknown domain Ω. Let $d = 2$ or 3, $f, u_d \in H^1(\mathbf{R}^d)$ and $\mathcal{A} \subset \mathcal{P}(\mathbf{R}^d)$ be a set of admissible shapes. Here, $\mathcal{P}(\mathbf{R}^d)$ denotes the power set of all subsets of $\mathbf{R}^d$. We consider the problem:

$$\min_{(\Omega,u)} J(\Omega, u) = \int_\Omega |u - u_d|^2 \,\mathrm{dx} \tag{28a}$$

subject to $(\Omega, u) \in \mathcal{A} \times H^1_0(\Omega)$ solves

$$\langle e(\Omega, u), \psi\rangle := \int_\Omega \nabla u \cdot \nabla\psi \,\mathrm{dx} - \int_\Omega f\psi \,\mathrm{dx} = 0 \tag{28b}$$

for all $\psi \in H_0^1(\Omega)$. The Lagrangian is given by:

$$\mathcal{L}(\Omega,\varphi,\psi) := \int_\Omega |\varphi - u_d|^2 \,\mathrm{d}x + \int_\Omega \nabla\varphi\cdot\nabla\psi \,\mathrm{d}x - \int_\Omega f\psi \,\mathrm{d}x. \tag{29}$$

Given an admissible shape Ω, a vector field $V \in C^{0,1}(\mathbf{R}^d)^d$ and $t > 0$ small, let $\Omega_t := (\mathrm{Id} + tV)(\Omega)$ be the perturbed domain. Therefore, the parametrised Lagrangian is given by:

$$G(t,\varphi,\psi) := \mathcal{L}(T_t(\Omega), \varphi\circ T_t^{-1}, \psi\circ T_t^{-1}), \quad \varphi,\psi \in H_0^1(\Omega). \tag{30}$$

Changing variables yields:

$$\begin{aligned} G(t,\varphi,\psi) &= \int_\Omega |\varphi - u_d^t|^2 \det(F_t)\mathrm{d}x \\ &\quad + \int_\Omega (F_t^{-\top}\nabla\varphi)\cdot(F_t^{-\top}\nabla\psi)\det(F_t)\,\mathrm{d}x - \int_\Omega f^t\psi \\ &\quad \det(F_t)\,\mathrm{d}x \qquad (31)\\ &=: \tilde{G}(T_t, F_t, \varphi, \psi), \end{aligned}$$

where $u_d^t = u_d \circ T_t$ and $f^t = f \circ T_t$. Here, we also transformed the gradient according to $(\nabla w)\circ T_t = F_t^{-\top}\nabla(w\circ T_t)$ for $w \in H_0^1(\Omega)$. Recall that, for a given $\Omega \in \mathcal{A}$, u_Ω denotes the corresponding unique solution to (28b) and $\mathcal{J}(\Omega)$ the reduced cost functional, $\mathcal{J}(\Omega) := J(\Omega, u_\Omega)$. Let $u^t \in H_0^1(\Omega)$ be the solution of the perturbed state equation brought back to the original domain Ω, that is, $u^t \in H_0^1(\Omega)$ is the unique solution to:

$$\partial_\psi G(t, u^t, 0)(\hat{\psi}) = 0 \quad \text{for all } \hat{\psi} \in H_0^1(\Omega). \tag{32}$$

Note that, for u^t defined by (32), it holds $\mathcal{J}(\Omega_t) = G(t, u^t, \psi)$ for all $\psi \in H_0^1(\Omega)$ and therefore also $D\mathcal{J}(\Omega)(V) = \frac{d}{dt}G(t, u^t, \psi)$ for all $\psi \in H_0^1(\Omega)$.

It can easily be shown that (26) holds and thus the shape derivative in the direction of a vector field $V \in C^{0,1}(\mathbf{R})^d$ is given by:

$$D\mathcal{J}(\Omega)(V) = \partial_t G(0, u, p),$$

where $p \in H_0^1(\Omega)$ denotes the adjoint state and is defined as the unique solution $p \in H_0^1(\Omega)$ to

$$\partial_\varphi G(0, u, p)(\hat{\varphi}) = 0 \quad \text{for all } \hat{\varphi} \in H_0^1(\Omega), \tag{33}$$

or explicitly

$$\int_\Omega \nabla\hat{\varphi}\cdot\nabla p \,\mathrm{d}x = -2\int_\Omega (u - u_d)\hat{\varphi}\,\mathrm{d}x \quad \text{for all } \hat{\varphi} \in H_0^1(\Omega). \tag{34}$$

4.2 First-order shape derivative

By the discussion above, the first-order shape derivative is given by $\partial_t G(0, u, p)$ with G defined in (31) and u and p the unique solutions to the boundary value problems (28b) and (34), respectively.

Writing $\tilde{G}(T_t, F_t) := \tilde{G}(T_t, F_t, u, p) = G(t, u, p)$, we obtain in analogy to the unconstrained problem:

$$D\mathcal{J}(\Omega)(V) = \frac{d}{dt}\mathcal{J}(\Omega_t)\Big|_{t=0} = \left(\frac{d\tilde{G}}{dT_t}V + \frac{d\tilde{G}}{dF_t}\partial V\right)\Big|_{t=0}.$$

We can compute explicitly:

$$\begin{aligned} \frac{d\tilde{G}}{dF_t}|_{t=0}\partial V &= \int_\Omega \mathrm{div}(V)(u-u_d)^2 - (\partial V + \partial V^\top)\nabla u\cdot\nabla p \\ &\quad - \mathrm{div}(V)\nabla u\cdot\nabla p - fp\,\mathrm{div}(V)\,\mathrm{d}x, \qquad (35)\\ \frac{d\tilde{G}}{dT_t}|_{t=0}V &= \int_\Omega -2(u-u_d)\nabla u_d\cdot V - \nabla f\cdot Vp\,\mathrm{d}x. \qquad (36) \end{aligned}$$

Now, we are in a position to compute the first-order shape derivative for the PDE-constrained shape optimisation problem (28) in NGSolve. After solving the state equation as shown in Section 2.1, the adjoint equation can be solved as follows.

```
64 ud = X[0]*(1-X[0])*X[1]*(1-X[1])
65 def Cost(u):
66     return (u-ud)**2 * Det(F) * dx
67
68 #solve adjoint equation
69 gfp = GridFunction(fes)
70 dCostdu = LinearForm(fes)
71 dCostdu += Cost(gfu).Diff(gfu, w)
72 dCostdu.Assemble()
73 gfp.vec.data = -a.mat.Inverse(fes.FreeDofs(),
       inverse="sparsecholesky").T * dCostdu.vec
74
75 Draw(gfp, mesh, "adjoint")
```

We can now define the Lagrangian (31) such that the shape derivative can be obtained by the same procedure as in the unconstrained setting. Note that lines 82–83 coincide with lines 48–49.

```
76 def Equation(u,w):
77     return ((Inv(F).trans * grad(u)) * (Inv(F)
       .trans * grad(w)) - f*w)*Det(F)*dx
78
79 G_pde = Cost(gfu) + Equation(gfu, gfp)
80
81 dJOmega_pde = LinearForm(VEC)
82 dJOmega_pde += G_pde.Diff(X, V)
83 dJOmega_pde += G_pde.Diff(F, grad(V))
```

4.3 Second-order shape derivative

Let us introduce the notation:

$$\begin{aligned} \langle E_{V,W}(s,t)\varphi,\psi\rangle &:= \int_\Omega (F_{s,t}^{-\top}\nabla\varphi)\cdot(F_{s,t}^{-\top}\nabla\psi)\det(F_{s,t})\,\mathrm{d}x \\ &\quad - \int_\Omega f\circ T_{s,t}\psi\det(F_{s,t})\,\mathrm{d}x \qquad (37)\\ J_{V,W}(s,t;\varphi) &:= \int_\Omega |\varphi - u_d\circ T_{s,t}|^2\det(F_{s,t})\mathrm{d}x \qquad (38) \end{aligned}$$

and

$$G_{V,W}(s,t,u,p) := \langle E_{V,W}(s,t)u,p\rangle + J_{V,W}(s,t;u), \tag{39}$$

where $T_{s,t}(x) = x + sV(x) + tW(x)$ and $F_{s,t} := \partial T_{s,t}$. We observe that

$$\mathcal{J}(T_{s,t}(\Omega)) = G_{V,W}(s, t, u^{s,t}, p^{s,t}) \tag{40}$$

with $(u^{s,t}, p^{s,t}) \in H^1_0(\Omega) \times H^1_0(\Omega)$ being the solution to

$$\partial_p G_{V,W}(s, t, u^{s,t}, 0)(\varphi) = 0 \quad \text{for all } \varphi \in H^1_0(\Omega), \tag{41}$$
$$\partial_u G_{V,W}(s, t, u^{s,t}, p^{s,t})(\psi) = 0 \quad \text{for all } \psi \in H^1_0(\Omega) \tag{42}$$

for $s, t \geq 0$. In case, $t = 0$ we write $u^s := u^{s,t}|_{t=0}$ and $p^s := p^{s,t}|_{t=0}$ and similarly for $t = s = 0$ we write $u := u^{s,t}|_{s=t=0}$ and $p := p^{s,t}|_{s=t=0}$. Therefore, consecutive differentiation of (40) first with respect to t at 0 and then with respect to s at 0 yields:

$$\begin{aligned} D^2\mathcal{J}(\Omega)(V)(W) &= \frac{d^2}{dsdt} G_{V,W}(s, t, u^{s,t}, p^{s,t})|_{s=t=0} \\ &= \frac{d}{ds}\partial_t G_{V,W}(s, 0, u^s, p^s)|_{s=0} \\ &= \partial_s\partial_t G_{V,W}(0, 0, u, p) + \partial_u\partial_t G_{V,W}(0, 0, u, p)(\partial_s u^0) \\ &\quad + \partial_p\partial_t G_{V,W}(0, 0, u, p)(\partial_s p^0), \end{aligned} \tag{43}$$

where $\partial_s u^0 \in H^1_0(\Omega)$ solves the material derivative equation:

$$\partial_u\partial_p G_{V,W}(0, 0, u, 0)(\psi)(\partial_s u^0) = -\partial_s\partial_p G_{V,W}(0, 0, u, 0)(\psi) \tag{44}$$

for all $\psi \in H^1_0(\Omega)$ or, equivalently

$$\langle \partial_u E_{V,W}(0, 0)(\partial_s u^0), \psi\rangle = -\langle \partial_s E_{V,W}(0, 0)u, \psi\rangle \tag{45}$$

for all $\psi \in H^1_0(\Omega)$. Note that (45) is obtained by differentiating (41) with respect to s and setting $s = t = 0$. Similarly, the function $\partial_s p^0 \in H^1_0(\Omega)$ solves the material derivative equation obtained by differentiating (42) with respect to s for $s = t = 0$,

$$\begin{aligned} \partial_p\partial_u G_{V,W}(0, 0, u, p)(\psi)(\partial_s p^0) &= -\partial^2_u G_{V,W}(0, 0, u, p)(\psi)(\partial_s u^0) \\ &\quad - \partial_s\partial_u G_{V,W}(0, 0, u, p)(\psi) \end{aligned} \tag{46}$$

for all $\psi \in H^1_0(\Omega)$. The introduction of the adjoint variable p is analogous to the computation of the first-order shape derivative. However, in contrast to the first-order derivative, the evaluation of $D^2\mathcal{J}(\Omega)(V)(W)$ requires the computation of the material derivatives $\partial_s u^0$ and $\partial_s p^0$.

Formally, (44) and (46) can be written as an operator equation with $x = (0, 0, u, p)$:

$$\begin{pmatrix} \partial^2_u G_{V,W}(x) & \partial_p\partial_u G_{V,W}(x) \\ \partial_u\partial_p G_{V,W}(x) & 0 \end{pmatrix} \begin{pmatrix} \partial_s u^0 \\ \partial_s p^0 \end{pmatrix} = -\begin{pmatrix} \partial_s\partial_u G_{V,W}(x) \\ \partial_s\partial_p G_{V,W}(x) \end{pmatrix}. \tag{47}$$

So to evaluate the second derivative (43) in some direction (V, W), we have to solve the system (47).

This is realised in `NGSolve` by setting up a combined finite element space which we denote by `X2`. We define trial and test functions as well as grid functions representing the deformation vector fields V and W, which we initialise with some functions.

```
X2 = FESpace([fes, fes])
dsu, dsp = X2.TrialFunction()
uTest, pTest = X2.TestFunction()
gfV = GridFunction(VEC)
gfW = GridFunction(VEC)
gfV.Set((X[0]*X[0]*X[1]*exp(X[1]),X[1]*X[1]*X
    [0]*exp(X[0])))
gfW.Set((X[1]*X[1]*X[0]*exp(X[0]),X[0]*X[0]*X
    [1]*exp(X[1])))
```

We define a 2×2 block bilinear form as well as a 2×1 block linear form which will represent the left- and right-hand sides of (47), respectively. The operator equation in (47) can be conveniently defined by differentiating the Lagrangian with respect to the corresponding variables.

```
shapeHessLag2 = BilinearForm(X2)
shapeGradLag2 = LinearForm(X2)

shapeHessLag2 += (G_pde.Diff(gfu, uTest)).Diff
    (gfu, dsu)    #block (1,1)
shapeHessLag2 += (G_pde.Diff(gfu, uTest)).Diff
    (gfp, dsp)    #block (1,2)
shapeHessLag2 += (G_pde.Diff(gfp, pTest)).Diff
    (gfu, dsu)    #block (2,1)

#line 1
shapeGradLag2 += (G_pde.Diff(gfu, uTest)).Diff
    (F, grad(gfV))
shapeGradLag2 += (G_pde.Diff(gfu, uTest)).Diff
    (X, gfV)

#line 2
shapeGradLag2 += (G_pde.Diff(gfp,pTest)).Diff(
    F, grad(gfV))
shapeGradLag2 += (G_pde.Diff(gfp,pTest)).Diff(
    X, gfV)
```

We can solve this combined system for $\partial_s u^0$ and $\partial_s p^0$ and access and visualise the two components in the following way:

```
gfCombined2 = GridFunction(X2)
shapeHessLag2.Assemble()
shapeGradLag2.Assemble()
gfCombined2.vec.data = shapeHessLag2.mat.
    Inverse(X2.FreeDofs(), inverse = "umfpack"
    ) * shapeGradLag2.vec

gfdsu = GridFunction(fes)
gfdsp = GridFunction(fes)
gfdsu.vec.data = gfCombined2.components[0].vec
gfdsp.vec.data = gfCombined2.components[1].vec

Draw(gfdsu, mesh, "dsu")
Draw(gfdsp, mesh, "dsp")
```

In order to obtain the second-order shape derivative in the direction given by (V, W), it remains to evaluate the term (43). We define the three terms of (43) as bilinear forms, assemble them, and perform vector-matrix-vector multiplications:

```
w1 = fes.TrialFunction()
q1 = fes.TrialFunction()

shapeHess11 = BilinearForm(VEC)
shapeHess11 += (G_pde.Diff(F,grad(W))+G_pde.
    Diff(X, W)).Diff(F,grad(V))
shapeHess11 += (G_pde.Diff(F,grad(W))+G_pde.
    Diff(X, W)).Diff(X, V)
shapeHess11.Assemble()

shapeHess12 = BilinearForm(trialspace = fes,
    testspace = VEC)
shapeHess12 += (G_pde.Diff(F, grad(V)) + G_pde
    .Diff(X, V)).Diff(gfu,w1)
shapeHess12.Assemble()

shapeHess13 = BilinearForm(trialspace = fes,
    testspace = VEC)
shapeHess13 += (G_pde.Diff(F, grad(V)) + G_pde
    .Diff(X, V)).Diff(gfp,q1)
shapeHess13.Assemble()

av = gfV.vec.CreateVector()
av.data = shapeHess11.mat * gfV.vec

adsu = gfV.vec.CreateVector()
adsu.data = shapeHess12.mat * gfdsu.vec

adsp = gfV.vec.CreateVector()
adsp.data = shapeHess13.mat * gfdsp.vec

d2J = InnerProduct(gfW.vec, av) +
    InnerProduct(gfW.vec, adsu) + InnerProduct
    (gfW.vec, adsp)
```

4.4 PDEs on surfaces

The automated shape differentiation is not restricted to partial differential equations on domains Ω, but is readily extended to surface PDEs. We consider a two-dimensional closed surface $M \subset \mathbf{R}^3$ and denote by n the normal field along M. Let $u_d \in H^1(\mathbf{R}^3)$ be given and define:

$$J(M,u) = \int_M |u-u_d|^2 \, ds, \tag{48}$$

where $u \in H^1(M)$ solves the surface equation

$$\int_M \nabla^M u \cdot \nabla^M \psi + u\psi \, ds = \int_M f\psi \, ds \quad \text{for all } \psi \in H^1(M), \tag{49}$$

where $\nabla^M \psi$ denotes the tangential gradient of ψ; see (Delfour and Zolésio 2011b, p. 493, Def.5.1). We assume that the function $f \in H^1(\mathbf{R}^3)$ is given. The Lagrangian is given by:

$$\mathcal{L}(M,\varphi,\psi) := \int_M |\varphi-u_d|^2 \, ds + \int_M \nabla^M\varphi\cdot\nabla^M\psi + \varphi\psi \, ds - \int_M f\psi \, ds.$$

As in the previous section, we fix an admissible shape M and let $M_t := (\mathrm{Id}+tV)(M)$ be a small perturbation of M by means of a vector field $V \in C^1(\mathbf{R}^3)^3$ for $t > 0$ small. The parametrised Lagrangian is given by:

$$G(t,\varphi,\psi) := \mathcal{L}(T_t(M), \varphi\circ T_t^{-1}, \psi\circ T_t^{-1}), \quad \varphi,\psi \in H^1(M). \tag{50}$$

Define the density $\omega(F_t) := \det(F_t)|F_t^{-\top}n|$. Changing variables and using

$$(\nabla^{M_t}\varphi)\circ T_t = B(F_t)\nabla^M(\varphi\circ T_t),$$
$$B(F_t) = \left(I - \frac{F_t^{-\top}n}{|F_t^{-\top}n|}\otimes\frac{F_t^{-\top}n}{|F_t^{-\top}n|}\right)F_t^{-\top}, \tag{51}$$

yields

$$\begin{aligned} G(t,\varphi,\psi) &= \int_M |\varphi-u_d^t|^2\,\omega(F_t)\,ds \\ &+ \int_M ((B(F_t)\nabla\varphi)\cdot(B(F_t)\nabla\psi)+\varphi\psi)\,\omega(F_t)\,ds \\ &- \int_M f^t\psi\,\omega(F_t)\,ds, \end{aligned} \tag{52}$$

where $u_d^t = u_d\circ T_t$ and $f^t = f\circ T_t$.

Writing $\tilde{G}(T_t,F_t) := G(t,u,p)$, we obtain in analogy to the domain case:

$$D\mathcal{J}(\Omega)(V) = \left(\frac{d\tilde{G}}{dT_t}V + \frac{d\tilde{G}}{dF_t}\partial V\right)\bigg|_{t=0}. \tag{53}$$

We can compute explicitly:

$$\begin{aligned} \frac{d\tilde{G}}{dF_t}|_{t=0}V &= \int_M \mathrm{div}^M(V)(u-u_d)^2 \\ &\quad -(\partial^M V + \partial^M V^\top)\nabla^M u\cdot\nabla^M p \\ &\quad +\mathrm{div}^M(V)(\nabla^M u\cdot\nabla^M p + up) \\ &\quad -fp\,\mathrm{div}^M(V)\,ds, \end{aligned} \tag{54}$$

$$\frac{d\tilde{G}}{dT_t}|_{t=0}\partial V = \int_M -2(u-u_d)\nabla u_d\cdot V - \nabla f\cdot Vp\,ds, \tag{55}$$

where $\partial^M V$ denotes the tangential Jacobian of V defined by $(\partial^M V)_{ij} := (\nabla^M V_i)_j$ for $i,j = 1,\dots,d$, and $\mathrm{div}^M(V) := \partial^M V : I$ the tangential divergence, which is defined as the trace of the tangential Jacobian; see (Delfour and Zolésio 2011b, p. 495).

The implementation is analogous to the previous sections. We will only illustrate first-order derivatives here. We first define the geometry of the unit sphere, create a surface mesh, and define a finite element space on the surface mesh:

```
from netgen.csg import *
from netgen.meshing import *
from ngsolve.internal import visoptions
from ngsolve import *

geo_surf = CSGeometry()
sphere = Sphere(Pnt(0,0,0),1).bc("outer")
geo_surf.Add(sphere)
mesh_surf = Mesh(geo_surf.GenerateMesh(
    perfstepsend=MeshingStep.MESHSURFACE,
    optsteps2d=3,maxh=0.2))
mesh_surf.Curve(3)
fes_surf = H1(mesh_surf,order = 3)
```

Next, we define the transformed cost function and partial differential equation needed for setting up the Lagrangian (52). Here, we again make use of a symbolic object `F` to which we assign the identity matrix. We define the tangential determinant ω and the matrix B defined in (51) as functions of the deformation gradient F_t.

```
154 X = CoefficientFunction((x,y,z))
155 func = CoefficientFunction(X[0]*X[1]*X[2])
156 F = Id(3)
157 tangDet = Det(F) * Norm( Inv(F).trans*
        specialcf.normal(3) )
158 Bmat = (Id(3) - 1/Norm(Inv(F).trans*specialcf.
        normal(3))**2 * OuterProduct(Inv(F).trans*
        specialcf.normal(3), Inv(F).trans*
        specialcf.normal(3)) ) * Inv(F).trans
159 
160 def Equation_surf(u,w):
161     return ( (Bmat*grad(u).Trace()) * (Bmat*
        grad(w).Trace()) + u*w - func * w) *
        tangDet * ds
162 
163 def Cost_surf(u):
164     return u**2 * tangDet * ds
```

Now, we can define the bilinear form and solve the state equation. Here, the right-hand side of the equation is included in the bilinear form and the boundary value problem—although linear—is solved by Newton's method (which terminates after only one iteration) for convenience.

```
165 #set up and solve state equation
166 u_surf, w_surf = fes_surf.TnT()
167 a = BilinearForm(fes_surf)
168 a += Equation_surf(u_surf, w_surf)
169 gfu_surf = GridFunction(fes_surf)
170 solvers.Newton(a, gfu_surf, printing = False)
171 Draw(gfu_surf, mesh_surf, "gfu_surf")
```

Using Newton's method for solving the linear boundary value problem allows us to define both the left- and right-hand sides of the PDE using only one `BilinearForm` `a` (which, strictly speaking, is not bilinear anymore). This way, we can reuse `Equation_surf` as defined in lines 160–161 to define the boundary value problem in line 168.

The adjoint equation is solved as usual:

```
172 #solve adjoint equation
173 lfcost_surf = LinearForm(fes_surf)
174 lfcost_surf += Cost_surf(gfu_surf).Diff(
        gfu_surf, w_surf)
175 lfcost_surf.Assemble()
176 inva = a.mat.Inverse(fes_surf.FreeDofs(),
        inverse="sparsecholesky")
177 gfp_surf = GridFunction(fes_surf)
178 gfp_surf.vec.data = -inva.T * lfcost_surf.vec
179 Draw(gfp_surf, mesh_surf, "gfp_surf")
```

The shape derivative is obtained as in the case of PDEs posed on volumes by the evaluation of (53):

```
180 G_surf = Cost_surf(gfu_surf) + Equation_surf(
        gfu_surf, gfp_surf)
181 
182 VEC3d = VectorH1(mesh_surf, order=1)
183 V3d = VEC3d.TestFunction()
184 dJOmega_surf = LinearForm(VEC3d)
185 dJOmega_surf += G_surf.Diff(X, V3d) + G_surf.
        Diff(F, Grad(V3d).Trace())
```

5 Fully automated shape differentiation

In the previous sections, we used the automatic differentiation capabilities of `NGSolve` to alleviate the shape differentiation procedure. However, so far, we still had to include some knowledge about the problems at hand. So far, it was necessary to define the objective function or Lagrangian G in the correct way, accounting for the correct transformation rules between perturbed and unperturbed domains. In this section, we will show that also this step can be automated since all necessary information are already included in the functional setting. The fully automated shape differentiation is incorporated by the command:

`DiffShape(...)`.

In particular, in the fully automated setting, it is enough to set up the cost function or Lagrangian for the unperturbed setting. For a shape function of the type (6), we can define the shape derivative of the cost function in the following way:

```
186 G_f_0 = f * dx
187 dJOmega_f_0 = LinearForm(VEC)
188 dJOmega_f_0 += G_f_0.DiffShape(V)
```

Note that there is no term of the form `Det(F)` showing up in line 186. Here, the transformation of the domain is taken care of automatically. It can be checked that this really gives the same result as `dJOmega_f` defined in lines 48–49.

```
189 dJOmega_f.Assemble()
190 dJOmega_f_0.Assemble()
191 differenceVec = dJOmega_f.vec.CreateVector()
192 differenceVec.data = dJOmega_f.vec -
        dJOmega_f_0.vec
193 print("|dJOmega_f - dJOmega_f_0| = ", Norm(
        differenceVec) )
```

The above code gives the output:

```
|dJOmega_f - dJOmega_f_0| =
1.5710085738106l9e-17
```

which confirms our claim. The same holds true for second-order shape derivatives. The lines 58–59 can be replaced by a repeated call of `DiffShape(...)`:

```
194 d2JOmega_f_0 = BilinearForm(VEC)
195 d2JOmega_f_0+=G_f_0.DiffShape(V).DiffShape(W)
```

Again, it can be verified that d2JOmega_f_0 coincides with the previously defined quantity d2JOmega_f. Note that slightly different results may occur due to different integration rules used. This can be cured by enforcing an integration rule of higher order for G_f, i.e. by replacing the symbol dx in the definition of G_f with dx(bonus_intorder=2).

In the more general setting of PDE-constrained shape optimisation, the procedure is very similar. Here, the idea exploited in the implementation of the command DiffShape(...) is to just differentiate the general expression (25) with respect to the parameter t. The transformations Φ_t appearing in (25), which depend on the functional setting of the PDE, are identified automatically from the finite element space from which the corresponding functions originate. The shape derivative of lines 82–83 can be obtained by the following code.

```
196 def Cost_0(u):
197     return (u-ud)**2 * dx
198 
199 def Equation_0(u,w):
200     return (grad(u) * grad(w) - f1*w) *dx
201 
202 G_pde_0 = Cost_0(gfu) + Equation_0(gfu, gfp)
203 
204 dJOmega_pde_0 = LinearForm(VEC)
205 dJOmega_pde_0 += G_pde_0.DiffShape(V)
```

Here, gfu and gfp represent the solutions to the state and adjoint equations, respectively, and must have been computed previously. The bilinear form shapeHess11 used in Section 4.3 (see lines 121–122) can be obtained similarly:

```
206 shapeHess11_0 = BilinearForm(VEC)
207 shapeHess11_0 += G_pde_0.DiffShape(W).
        DiffShape(V)
```

The same holds true for boundary integrals

```
208 G_f_bnd_0 = f * ds
209 dJOmega_f_bnd_0 = LinearForm(VEC)
210 dJOmega_f_bnd_0 += G_f_bnd_0.DiffShape(V)
```

and surface PDEs

```
211 def Cost_surf_0(u):
212     return u**2 * ds
213 def Equation_surf_0(u,w):
214     return  (grad(u).Trace()*grad(w).Trace() +
         u*w - func * w) * ds
215 G_surf_0 = Cost_surf_0(gfu_surf) +
        Equation_surf_0(gfu_surf, gfp_surf)
216 dJOmega_surf_0 = LinearForm(VEC3d)
217 dJOmega_surf_0 += G_surf_0.DiffShape(V3d)
```

as well as their respective second-order derivatives.

Remark 5 We remark that the fully automated differentiation using DiffShape(...) should be seen to complement the semi-automated shape differentiation techniques introduced in Sections 3 and 4 rather than to replace them. Using the semi-automated differentiation, the user has the possibility to, on the one hand, keep control over the involved terms, and on the other hand also to adjust the shape differentiation to their custom problems which may be non-standard. As an example where the semi-automated differentiation may be beneficial compared with the fully automated differentiation, we mention the case of time-dependent PDE constraints considered in a space-time setting when a shape deformation is only desired in the spatial coordinates; see Section 7.8. Of course, when one is interested in the shape derivative for a more standard problem, the fully automated way appears to be more convenient and less error prone.

Remark 6 We have seen that the command DiffShape(...) allows computing the shape derivative of unconstrained shape optimisation problems in a fully automated way without specifying any transformation rules; see line 188. For the practically more relevant case of PDE-constrained shape optimisation problems, the state and adjoint equations have to be solved beforehand also in the fully automated context using DiffShape(...). We remark that this can be easily achieved by defining a custom function solvePDE() as it is done for the case of a linear PDE in lines 227–234. Since the purpose of this paper is to illustrate a convenient way of computing shape derivatives and performing shape optimisation rather than to provide a tool for black-box optimisation, this step is left to the user and is not automated, leaving more freedom in the choice of e.g. solvers for the arising linear systems.

6 Optimisation algorithms

In this section, we discuss how to use optimisation algorithms in conjunction with the automated shape differentiation explained in the previous sections. The starting point of our discussion is a fixed initial shape Ω. Then, we consider the mapping:

$$V \mapsto g(V) := \mathcal{J}((\mathrm{Id} + V)(\Omega)) \tag{56}$$

defined on a suitable space of vector fields $\Theta \subset C^{0,1}(\mathsf{D})^d$. Since the mapping g is defined on an open subset Θ of the Banach space $C^{0,1}(\mathsf{D})^d$, we can employ standard algorithms to minimise g over Θ. The only constraint we must impose is that $\mathrm{Id} + V$ remains invertible, which can be difficult in practice. In view of $g(V + tW) = \mathcal{J}((\mathrm{Id} + V + tW)(\Omega)) = \mathcal{J}((\mathrm{Id} + tW \circ (\mathrm{Id} + V)^{-1})((\mathrm{Id} + V)(\Omega)))$ for $V, W \in \Theta$ and t small, we find by differentiating with respect to t at $t = 0$, that:

$$\partial g(V)(W) = D\mathcal{J}((\mathrm{Id} + V)(\Omega))(W \circ (\mathrm{Id} + V)^{-1}) \tag{57}$$

for $V, W \in \Theta$ and $\mathrm{Id} + V$ invertible.

6.1 Gradient computation

The gradient of $\partial g(V)$ in a Hilbert space $H \subset C^{0,1}(\mathsf{D})^d$ is defined by:

$$\partial g(V)(W) = (\nabla^H g(V), W)_H \quad \text{for all } W \in H. \tag{58}$$

Typical choices for H are:

$$H = H_0^1(\mathsf{D})^d,\ (W, V)_H := \int_{\mathsf{D}} \partial W : \partial V + V \cdot W \,\mathrm{dx}, \tag{59}$$

$$H = H_0^1(\mathsf{D})^d,\ (W, V)_H := \int_{\mathsf{D}} \varepsilon(W) : \varepsilon(V) + V \cdot W \,\mathrm{dx}, \tag{60}$$

$$H = H_0^1(\mathsf{D})^d,\ (W, V)_H := \int_{\mathsf{D}} \varepsilon(W) : \varepsilon(V) + V \cdot W + \gamma_{CR} \mathcal{B}V \cdot \mathcal{B}W \,\mathrm{dx}, \tag{61}$$

where $\varepsilon(V) := \frac{1}{2}(\partial V + \partial V^\top)$, $\gamma_{CR} > 0$ and

$$\mathcal{B} := \begin{pmatrix} -\partial_x & \partial_y \\ \partial_y & \partial_x \end{pmatrix}. \tag{62}$$

The last choice, which is restricted to the spatial dimension $d = 2$, corresponds to a penalised Cauchy-Riemann gradient and results in a gradient which is approximately conformal and hence preserves good mesh quality. We refer to Iglesias et al. (2018) for a detailed description. We also refer to de Gournay (2006) and Burger (2002) and Allaire et al. (2021) for the use of different inner products.

6.2 Basic algorithm

Let Ω be an initial shape and let $H \subset C^{0,1}(\mathsf{D})^d$ be a Hilbert space. Then, a basic shape optimisation algorithm reads as follows.

Algorithm 1 gradient algorithm.

```
1:  Input: domain $\Omega_0$, $n = 0$, $N_{max} > 0$, $\epsilon > 0$, $\gamma \geq 0$
2:  Output: optimal shape $\Omega^*$
3:  while $n \leq N_{max}$ and $|\nabla\mathcal{J}(\Omega_n)| > \epsilon$ do
4:      if $\mathcal{J}((\mathrm{Id} - \alpha\nabla\mathcal{J}(\Omega_n))(\Omega_n)) < \mathcal{J}(\Omega_n) - \gamma\alpha|\nabla\mathcal{J}(\Omega_n)|^2$ then
5:          $\Omega_{n+1} \leftarrow (\mathrm{Id} - \alpha\nabla\mathcal{J}(\Omega_n))(\Omega_n)$
6:          $n \leftarrow n + 1$
7:          increase $\alpha$
8:      else
9:          reduce $\alpha$
10:     end if
11: end while
```

We present and explain the numerical realisation of Algorithm 1 in `NGSolve` for the case of a PDE-constrained shape optimisation problem in two space dimensions. The simpler case of an unconstrained shape optimisation problem or the case of three space dimensions can be realised by small modifications of the presented code.

First of all, we mention that we realise shape modifications in `NGSolve` by means of deformation vector fields without actually modifying the coordinates of the underlying finite element grid. Recall the vector-valued finite element space `VEC` over a given mesh as introduced in code line 44. We define a vector-valued `GridFunction` with the name `gfset` which will represent the current shape. We initialise it with some vector-valued coefficient function $V(x_1, x_2) = (x_1^2 x_2, x_2^2 x_1)^\top$ and obtain the deformed shape $(\mathrm{Id} + V)(\Omega)$ by the command `mesh.SetDeformation(gfset)`:

```
218 gfset = GridFunction(VEC)
219 Draw(gfset, mesh, "gfset")
220 SetVisualization (deformation=True)
221 gfset.Set((X[0]*X[0]*X[1],X[1]*X[1]*X[0]))
222 mesh.SetDeformation(gfset)
223 Redraw()
```

Any operation involving the mesh such as integration or assembling of matrices is now carried out for the deformed configuration. To be more precise, a change of variables is performed internally by accounting for the corresponding Jacobi determinant and transforming the derivatives accordingly with the Jacobian of the deformation. Therefore, all resulting coefficient vectors (which are stored in GridFunctions) correspond to the shape functions in reference configuration. The deformation can be unset by the command mesh.UnsetDeformation(). Integrating the constant function over the mesh in the perturbed and unperturbed settings,

```
print(Integrate(1, mesh))
mesh.UnsetDeformation()
print(Integrate(1, mesh))
```

gives the output

```
1.7924529046862627
0.7854072970684544
```

respectively.

In the course of the optimisation algorithm, the state equation as well as the adjoint equation has to be solved for every new shape. We define the following function, which computes the state and adjoint state for a linear PDE constraint:

```
def solvePDE():
    a.Assemble()
    L.Assemble()
    dCostdu.Assemble()

    inva = a.mat.Inverse(fes.FreeDofs(),
    inverse="sparsecholesky")
    gfu.vec.data = inva * L.vec
    gfp.vec.data = -inva.T * dCostdu.vec
```

The shape derivative dJOmega for some problem at hand can be defined as illustrated in Sections 4.1 and Section 5. Finally, we need to define the shape gradient, which is the solution to a boundary value problem of the form (58). We choose the bilinear form defined in (61) with $\gamma_{CR} = 10$:

```
def eps(u):
    return 1/2 * (grad(u)+grad(u).trans)

aX = BilinearForm(VEC)
W, V = VEC.TnT()      # define trial function W
    and test function V

aX += InnerProduct(eps(W), eps(V))*dx +
    InnerProduct(W, V)*dx
aX += 10 * (grad(W)[1,1] - grad(W)[0,0])*(grad
    (V)[1,1] - grad(V)[0,0])*dx
aX += 10 * (grad(W)[1,0] + grad(W)[0,1])*(grad
    (V)[1,0] + grad(V)[0,1])*dx
```

Now, we can run Algorithm 1 for problem (28):

```
alpha = 1
alpha_incr_factor = 1.2
gamma = 1e-4
Nmax = 100
epsilon = 1e-7

isConverged = False
gfset.Set((0,0))
gfX = GridFunction(VEC)
gfsetTemp = GridFunction(VEC)

solvePDE()
Jnew = Integrate(Cost(gfu), mesh)
Jold = Jnew

for k in range(Nmax):
    mesh.SetDeformation(gfset)
    aX.Assemble()
    dJOmega_pde.Assemble()
    invaX = aX.mat.Inverse(VEC.FreeDofs(),
    inverse="sparsecholesky")
    gfX.vec.data = invaX * dJOmega_pde.vec
    currentNormGFX = Norm(gfX.vec)

    while True:
        if currentNormGFX < epsilon:
            isConverged = True
            break

        gfsetTemp.vec.data = gfset.vec - alpha
    * gfX.vec
        mesh.SetDeformation(gfsetTemp)
        solvePDE()
        Jnew = Integrate (Cost(gfu), mesh)
        mesh.UnsetDeformation()
        if Jnew < Jold - gamma * alpha *
    currentNormGFX**2 :
            Jold = Jnew
            gfset.vec.data = gfsetTemp.vec
            alpha *= alpha_incr_factor
            break
        else:
            alpha = alpha / 2
    Redraw(blocking=True)
```

Mesh movement and mesh optimisation As an alternative to realizing the deformations via mesh.SetDeformation(...), where the underlying mesh is not modified, one could also just move every mesh node in the direction of the given descent vector field by changing its coordinates. This can be realised by invoking the following method:

```
def moveNGmesh2D(displ, mesh):
    for p in mesh.ngmesh.Points():
        v = displ(mesh(p[0],p[1]))
        p[0] += v[0]
        p[1] += v[1]
    mesh.ngmesh.Update()
```

Here, the displacement vector field displ, which is of type GridFunction, is evaluated for each mesh node and, subsequently, the mesh nodes are updated. At the end of the procedure, the mesh structure needs to be updated; see line 290. Note that `GridFunctions` can only be evaluated at points inside the mesh (but not necessarily vertices of the mesh). Therefore, in order to evaluate `displ` at the point given by the coordinates `p[0], p[1]`, we need to pass `mesh(p[0],p[1])` in line 287.

One advantage of this strategy is that an ill-shaped mesh can easily be repaired by a call of the method `mesh.ngmesh.OptimizeMesh2d()` followed by `mesh.ngmesh.Update()`. Figure 3 shows an ill-shaped mesh and the result of a call of `mesh.ngmesh.OptimizeMesh2d()`.

6.3 Newton's method for unconstrained problems

The particular choice $H = H_0^1(\mathsf{D})^d$ and

$$(V, W)_H := D^2\mathcal{J}(\Omega)(V)(W), \tag{63}$$

for a given shape function $\mathcal{J}$ leads to Newton's method. We refer to Novruzi and Roche (2000), Allaire et al. (2016), Paganini and Sturm (2019), and Eppler et al. (2007) where shape Newton methods were used previously and to (Hinze et al. 2009, Chapter 2) and (Ito and Kunisch 2008, Chapter 5) for Newton's method in an optimal control setting. This bilinear form is only positive semi-definite on $H_0^1(\mathsf{D})^d$ since $D^2\mathcal{J}(\Omega)(V)(W) = 0$ for V, W with $V = W = 0$ on $\partial\Omega$. Moreover, from the structure theorem for second shape derivatives proved in Novruzi and Pierre (2002), we know that at a stationary point Ω, that is, $D\mathcal{J}(\Omega)(V) = 0$ for all $V \in C^{0,1}(\mathsf{D})^d$, we have

$$D^2\mathcal{J}(\Omega)(V)(W) = \ell_\Omega(V \cdot n, W \cdot n), \tag{64}$$

where $\ell_\Omega : C^0(\partial\Omega) \times C^0(\partial\Omega) \to \mathbf{R}$ is a bilinear function. Hence, we also have $D^2\mathcal{J}(\Omega)(V)(W) = 0$ for all V, W such that $V \cdot n = W \cdot n = 0$. As a result, the gradient

$$(\nabla\mathcal{J}(\Omega), V)_H = D\mathcal{J}(\Omega)(V) \quad \text{for all } V \in H_0^1(\mathsf{D})^d \tag{65}$$

according to (63) is not uniquely determined. To get around this difficulty, the shape Hessian is often regularised by an H^1 term, i.e. (63) is replaced by

$$D^2\mathcal{J}(\Omega)(V)(W) + \delta \int_\Omega \partial V : \partial W + V \cdot W \, \mathrm{d}x, \tag{66}$$

see, e.g. Schmidt (2018), which, however, impairs the convergence speed of Newton's method.

Alternative regularisation strategy Here, we propose the following strategy: We regularise the shape Hessian only on the boundary $\partial\Omega$ and only in tangential direction, i.e. we choose

$$(V, W)_H := D^2\mathcal{J}(\Omega)(V)(W) + \delta \int_{\partial\Omega} (V \cdot \tau)(W \cdot \tau) \tag{67}$$

with a regularisation parameter δ. To exclude the part of the kernel corresponding to interior deformations, we solve the (regularised) Newton (65) only on the boundary $\partial\Omega$. This is realised by setting Dirichlet boundary conditions for all degrees of freedom except those on the boundary.

```
291 VEC2 = VectorH1(mesh, order=1, dirichlet = "
        circle") #auxiliary space for boundary
        conditions
292 aX = BilinearForm(VEC)
293 aX += G_f_0.DiffShape(W).DiffShape(V)
294 aX += 100 * InnerProduct(W, specialcf.
        tangential(2)) * InnerProduct(V, specialcf
        .tangential(2))*ds
295 aX.Assemble()
296 invAX = aX.mat.Inverse(~VEC2.FreeDofs(),
        inverse="umfpack")
297
298 gfX_bnd = GridFunction(VEC)
299 gfX_bnd.vec.data = invAX * dJOmega_f_0.vec
```

As a result, we get a shape gradient $\nabla\tilde{\mathcal{J}}(\Omega)$ which is nonzero only on the boundary. We extend this vector field to the interior by solving an additional boundary value problem (of linearised elasticity type), where we use the deformation given by $\nabla\tilde{\mathcal{J}}(\Omega)$ as Dirichlet boundary conditions.

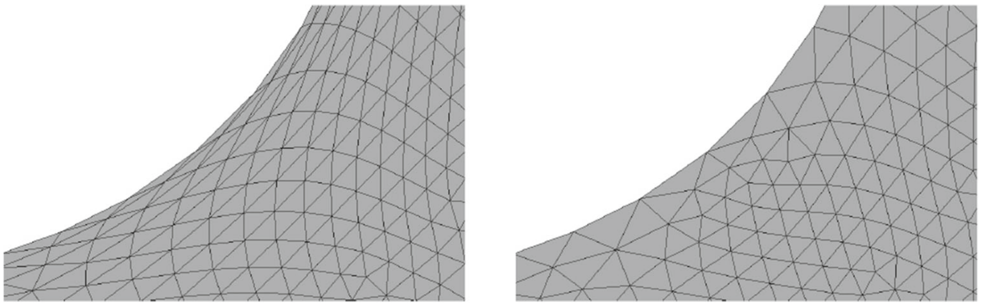


Fig. 3 Before and after mesh optimisation by `mesh.ngmesh.OptimizeMesh2d()`

```
def getExtension(gfX_bnd, freedofs, gfX_ext):
    u,v = VEC.TnT()
    aX_ext = BilinearForm(VEC)
    aX_ext += InnerProduct(grad(u)+grad(u).
    trans, grad(v))*dx+InnerProduct(u,v)*dx

    gfX_ext.Set(gfX_bnd)
    aX_ext.Assemble()

    r = gfX_bnd.vec.CreateVector()
    r.data = (-1) * aX_ext.mat * gfX_ext.vec

    gfX_ext.vec.data += aX_ext.mat.Inverse(
    freedofs=freedofs) * r

getExtension(gfX_bnd, VEC2.FreeDofs(), gfX)
gfset.Set((0,0))
gfset.vec.data = gfset.vec - 1 * gfX.vec
```

The Newton algorithm reads as follows.

Algorithm 2 Newton algorithm.

```
1: Input: domain $\Omega_0$, $n = 0$, $N_{max} > 0$, $\epsilon > 0$
2: Output: optimal shape $\Omega^*$
3: while $n \leq N_{max}$ and $|\nabla\mathcal{J}(\Omega_n)| > \epsilon$ do
4:     solve Eq. 65 to get $\nabla\mathcal{J}(\Omega_n)$
5:     $\Omega_{n+1} \leftarrow (\mathrm{Id} - \nabla\mathcal{J}(\Omega_n))(\Omega_n)$
6:     $n \leftarrow n + 1$
7: end while
```

6.4 Newton's method for PDE-constrained problems

We consider the PDE-constrained model problem of Section 4.1 which is subject to the Poisson equation. The unregularised Newton system reads:

$$D^2\mathcal{J}(\Omega)(V)(W) = -D\mathcal{J}(\Omega)(V) \qquad \text{for all } V \in H_0^1(\Omega). \tag{68}$$

In Section 4.3, we discussed how the second-order shape derivative can be evaluated along a fixed given direction. In this section, we want to assemble the whole shape Hessian and eventually solve a regularised version of (68). Recalling that $D\mathcal{J}(\Omega)(V) = \partial_s G_{V,0}(x)$ with $x = (0, 0, u, p)$, we see that (47) and (43) lead to:

$$\mathcal{H}\begin{pmatrix}\tilde{V}\\ \partial_s u^0\\ \partial_s p^0\end{pmatrix} = -\begin{pmatrix}\partial_t G_{V,W}(x)\\ 0\\ 0\end{pmatrix}. \tag{69}$$

with

$$\mathcal{H}(x) = \begin{pmatrix}\partial_s\partial_t G_{V,W}(x) & \partial_u\partial_t G_{V,W}(x) & \partial_p\partial_t G_{V,W}(x)\\ \partial_s\partial_u G_{V,W}(x) & \partial_u^2 G_{V,W}(x) & \partial_p\partial_u G_{V,W}(x)\\ \partial_s\partial_p G_{V,W}(x) & \partial_u\partial_p G_{V,W}(x) & 0\end{pmatrix}. \tag{70}$$

The component $\tilde{V}$ then represents the direction which we use for the shape Newton optimisation step. The matrix in (69) can be realised in NGSolve by using a combined finite element space X3 consisting of three components as follows:

```
X3 = FESpace([VEC, fes, fes])
PHI, u1, p1= X3.TrialFunction()
PSI, uTest1, pTest1 = X3.TestFunction()

shapeHessLag3 = BilinearForm(X3)
shapeHessLag3 += G_pde_0.DiffShape(PHI).
    DiffShape(PSI)          #block (1,1)
shapeHessLag3 += G_pde_0.DiffShape(PSI).Diff(
    gfu,u1)          #block (1,2)
shapeHessLag3 += G_pde_0.DiffShape(PSI).Diff(
    gfp,p1)          #block (1,3)
shapeHessLag3 += G_pde_0.Diff(gfu, uTest1).
    DiffShape(PHI)     #block (2,1)
shapeHessLag3 += (G_pde_0.Diff(gfu, uTest1)).
    Diff(gfu, u1)    #block (2,2)
shapeHessLag3 += (G_pde_0.Diff(gfu, uTest1)).
    Diff(gfp, p1)    #block (2,3)
shapeHessLag3 += G_pde_0.Diff(gfp,pTest1).
    DiffShape(PHI)      #block (3,1)
shapeHessLag3 += (G_pde_0.Diff(gfp,pTest1)).
    Diff(gfu, u1)     #block (3,2)
```

The right-hand side of (69) can be defined as follows:

```
shapeGradLag3 = LinearForm(X3)
shapeGradLag3 += (-1) * G_pde_0.DiffShape(PSI)
```

Recall that the system (65) has a nontrivial kernel as discussed in Section 6.3. This problem can be circumvented by proceeding like in the unconstrained case. We add a regularisation only on the boundary,

```
delta = 1
shapeHessLag3 += delta * InnerProduct(PHI,
    specialcf.tangential(2)) * InnerProduct(
    PSI, specialcf.tangential(2))*ds
```

and exclude the interior degrees of freedom in the first row and column of the 3×3 block system. This can be realised by setting Dirichlet boundary conditions for the interior degrees of freedom, i.e. by dealing with the free degrees of freedom,

```
# copy of VEC with Dirichlet boundary
    conditions on whole boundary:
VEC2 = VectorH1(mesh, order = 1, dirichlet = "
    .*")
freeDofsCombined = BitArray(VEC2.ndof + 2*fes.
    ndof)
for i in range(VEC2.ndof):
    freeDofsCombined[i] = not VEC2.FreeDofs()[
    i]
for i in range(fes.ndof):
    freeDofsCombined[VEC2.ndof+i] = fes.
    FreeDofs()[i]
    freeDofsCombined[VEC2.ndof+fes.ndof+i] =
    fes.FreeDofs()[i]
```

and solving the regularised system using these free dofs:

```
gfCombined3 = GridFunction(X3)
shapeHessLag3.Assemble()
shapeGradLag3.Assemble()
gfCombined3.vec.data = shapeHessLag3.mat.
    Inverse(freedofs=freeDofsCombined, inverse
    ="umfpack") * shapeGradLag3.vec
```

The Newton direction is then given as the first of the three components of the obtained solution.

```
Vtilde_bnd = GridFunction(VEC)
Vtilde = GridFunction(VEC)
Vtilde_bnd.vec.data = gfCombined3.components
    [0].vec
getExtension(Vtilde_bnd, VEC2.FreeDofs(),
    Vtilde)

gfset.vec.data = gfset.vec + 1 * Vtilde.vec
```

7 Numerical experiments

In this section, we first verify the computed shape derivatives by performing a Taylor test, and then apply the automated shape differentiation and the numerical algorithms introduced in the preceding sections in numerical examples.

7.1 Code verification

We verify the expressions that we obtained in a semi-automatic or fully automatic way for the first- and second-order shape derivatives by looking at the Taylor expansions of the perturbed shape functionals. We illustrate our findings in two examples in $\mathbf{R}^2$. On the one hand, we consider a shape function as introduced in (6) with an additional boundary integral as in (13), henceforth denoted by $\mathcal{J}_1$; on the other hand, we consider the PDE-constrained shape optimisation problem defined by (28), the reduced form of which will be denoted by $\mathcal{J}_2(\Omega)$. More precisely, we consider:

$$\mathcal{J}_1(\Omega) = \int_\Omega f(x)\,\mathrm{d}x + \int_{\partial\Omega} f(x)\,\mathrm{d}s, \tag{71}$$

$$\mathcal{J}_2(\Omega) = \int_\Omega |u_\Omega - u_d|^2\,\mathrm{d}x \;\text{ where } u_\Omega \text{ solves (28b)}. \tag{72}$$

In the case of $\mathcal{J}_1$, we used the function:

$$f(x_1,x_2) = \left(0.5 + \sqrt{x_1^2+x_2^2}\right)^2 \left(0.5 - \sqrt{x_1^2+x_2^2}\right)^2$$

and for $\mathcal{J}_2$, we used $u_d(x_1,x_2) = x_1(1-x_1)x_2(1-x_2)$ and $f(x_1,x_2) = 2x_2(1-x_2) + 2x_1(1-x_1)$ for the function f in the PDE constraint (28b).

For the test of the first-order shape derivatives $D\mathcal{J}_i(\Omega)(V)$, we choose a fixed shape Ω and a vector field $V \in C^{0,1}(\mathbf{R}^2)^2$ and observe the quantity:

$$\delta_1(\mathcal{J}_i,t) := |\mathcal{J}_i((\mathrm{Id}+tV)(\Omega)) - \mathcal{J}_i(\Omega) - t\,D\mathcal{J}_i(\Omega)(V)|, \tag{73}$$

for $t \searrow 0$. Likewise, for the second-order shape derivative, we consider the remainder:

$$\delta_2(\mathcal{J}_i,t) := \Big| \mathcal{J}_i((\mathrm{Id}+tV)(\Omega)) - \mathcal{J}_i(\Omega) - t\,D\mathcal{J}_i(\Omega)(V) - \tfrac{1}{2}t^2 D^2\mathcal{J}_i(\Omega)(V)(V) \Big|$$

as $t \searrow 0$. By the definition of first- and second-order shape derivatives, it must hold that

$$\delta_1(\mathcal{J}_i,t) = \mathcal{O}(t^2) \quad \text{and} \quad \delta_2(\mathcal{J}_i,t) = \mathcal{O}(t^3) \quad \text{as } t \searrow 0. \tag{74}$$

This behavior can be observed in Fig. 4a for $\mathcal{J}_1$ and in Fig. 4b for $\mathcal{J}_2$, where we used $V(x_1,x_2) = (x_1^2 x_2 e^{x_2}, x_2^2 x_1 e^{x_1})$ in both cases.

The experiments for shape function $\mathcal{J}_1$ were conducted on a mesh consisting of 13,662 vertices, 26,946 elements, and with polynomial order 2 (resulting in 54,269 degrees of freedom), and the experiment for $\mathcal{J}_2$ with 95,556 vertices and 190,062 elements and polynomial degree 1 (95,556 degrees of freedom). We conducted these experiments for a number of different problems with different vector fields V, in particular with different PDE constraints and boundary conditions, and obtained similar results in all instances provided a sufficiently fine mesh was used.

7.2 A first shape optimisation problem

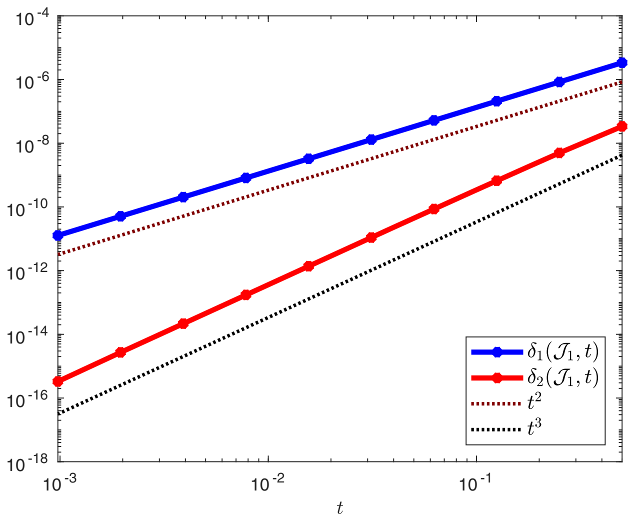
In this section, we revisit problem (6) introduced in Section 3, i.e. the problem of finding a shape Ω such that the cost function $\mathcal{J}(\Omega) = \int_\Omega f(x)\,\mathrm{d}x$ is minimised.

7.2.1 First-order methods

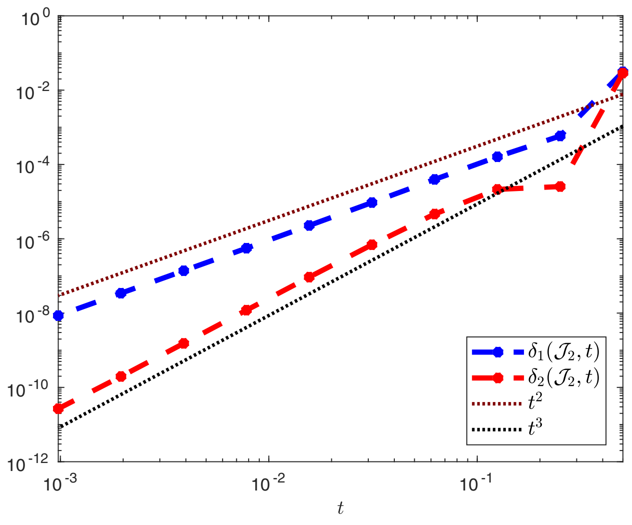
We illustrate our first-order methods in a problem which was also considered in Iglesias et al. (2018) and reproduce the results obtained there. We choose the function:

$$f(x_1,\ x_2) = \left(\sqrt{(x_1-a)^2+bx_2^2}-1\right)\left(\sqrt{(x_1+a)^2+bx_2^2}-1\right) \cdot \left(\sqrt{bx_1^2+(x_2-a)^2}-1\right)\left(\sqrt{bx_1^2+(x_2+a)^2}-1\right) - \epsilon \tag{75}$$

with $a = \frac{4}{5}$, $b = 2$ and $\epsilon = 0.001$. Recall that the optimal shape is given by $\{(x_1,x_2) \in \mathbf{R}^2 : f(x_1,x_2) < 0\}$ which is depicted in Fig. 5 (right). We start our optimisation



(a)



(b)

Fig. 4 Taylor test for functions $\mathcal{J}_1$ and $\mathcal{J}_2$

algorithm with the unit disk, $\Omega^0 = B_1(0)$ as an initial design. Note that the optimal design cannot be reached by means of shape optimisation using boundary perturbations. However, we expect the outer curve of the optimal shape to be reached very closely.

We apply Algorithm 1 with the shape gradient $\nabla\mathcal{J}$ associated to the H^1 inner product (59), to the bilinear form of linearised elasticity (60) and including the additional Cauchy-Riemann term (61). We chose the algorithmic parameters $\gamma = 1e-4$, $\epsilon = 1e-7$, a mesh consisting of 2522 vertices and 4886 elements and a globally continuous vector-valued finite element space `VEC` of order 3. The results can be seen in Figs. 6, 7 and 8, respectively.

7.2.2 Second-order method

Since Newton's method converges quadratically only in a neighborhood of the optimal solution, we choose a simpler optimal design here. We choose:

$$f(x_1,x_2) = \frac{x^2}{a^2} + \frac{y^2}{b^2} - 1 \tag{76}$$

which yields an ellipse with the lengths of the two semi-axes a and b. We choose $a = 1.3$ and $b = 1/a$ and again start the optimisation with the unit disk as an initial shape. Figure 9 shows the initial and optimised design after only six iterations of Algorithm 2 with $(\cdot,\cdot)_H$ chosen as in (67) with $\delta = 100$. A comparison of the convergence histories between the choice (67) with $\delta = 100$ and (66) with $\delta = 0.5$ is shown in the right picture of Fig. 9. In both cases, we tested a range of different values for δ and compared the convergence histories for the values which yielded the fastest convergence. The experiments were conducted on a finite element mesh consisting of 2522 nodes and 4886

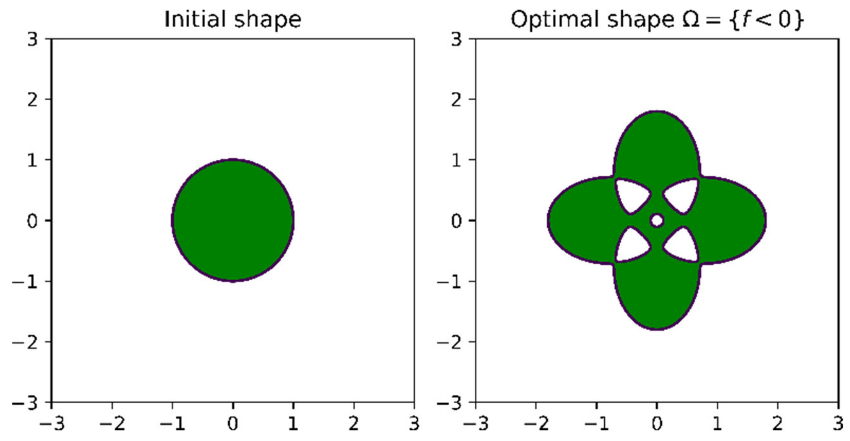


Fig. 5 Initial domain Ω_0 and optimal domain Ω^* for problem (6) with f chosen according to (75)

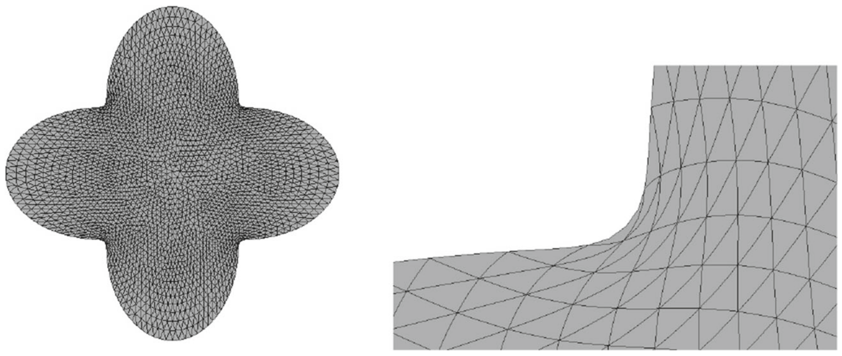


Fig. 6 Results of problem (6) with f as in (75) and the shape gradient associated to the H^1 inner product (59)

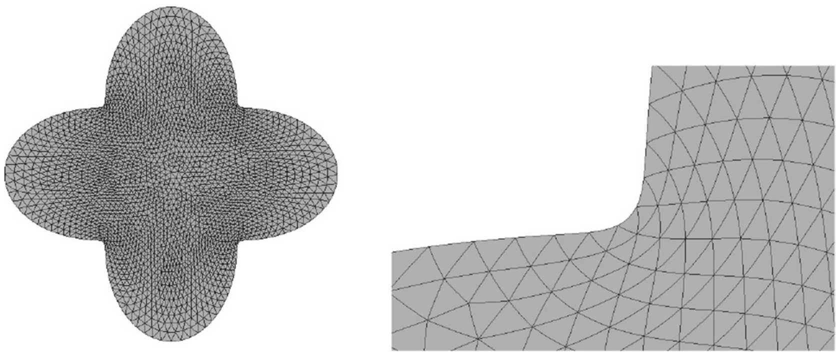


Fig. 7 Results of problem (6) with f as in (75) and the shape gradient associated to the elasticity bilinear form (60)

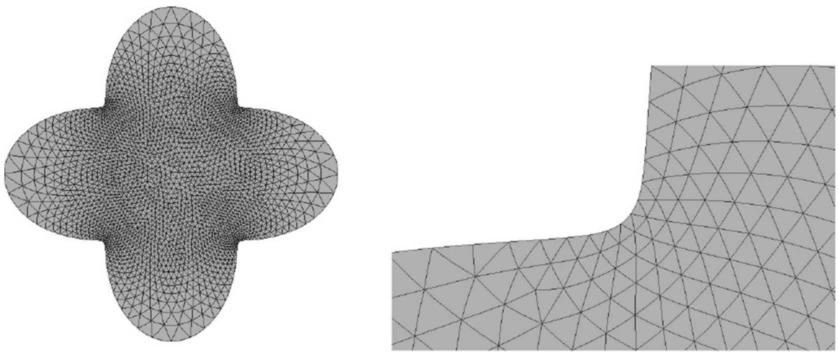


Fig. 8 Results of problem (6) with f as in (75) and the shape gradient associated to the elasticity bilinear form with Cauchy-Riemann term (61)

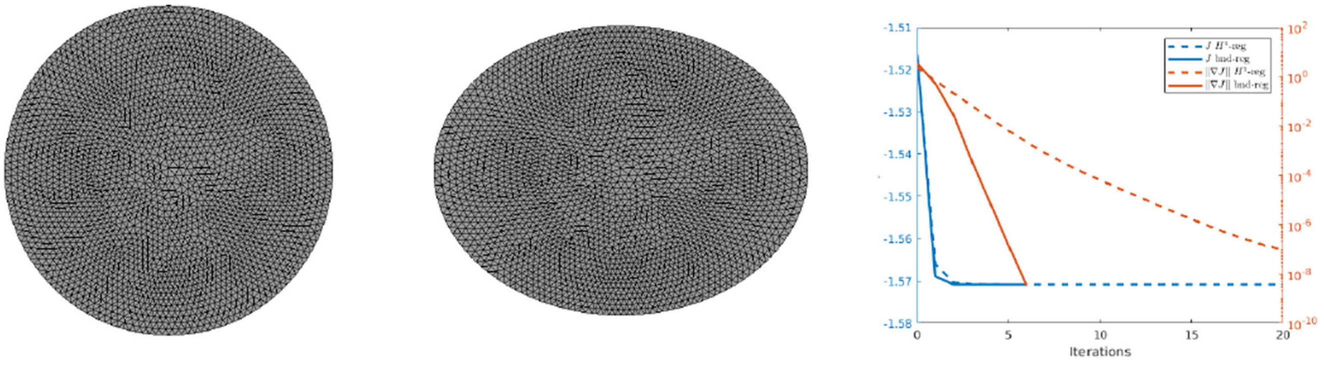


Fig. 9 Numerical results for problem (67) with f as in (76) using second-order method. Left: Initial design. Centre: Optimised design after six iterations using (65)/(67). Right: Objective value $\mathcal{J}$ and norm of shape gradient $\|\nabla \mathcal{J}(\Omega)\|$ in the course of second-order optimisation using (66) with $\delta = 0.5$ and (67) with $\delta = 100$

triangular elements with a finite element space VEC of order 3, with the algorithmic parameter $\epsilon = 10^{-7}$.

7.3 Shape optimisation subject to the Poisson equation

In this section, we revisit the model problem introduced in Section 4.1 with $f(x_1, x_2) = 2x_2(1 - x_2) + 2x_1(1 - x_1)$ and $u_d(x_1, x_2) = x_1(1 - x_1)x_2(1 - x_2)$. Note that the data is chosen in such a way that, for $\Omega^* = (0, 1)^2$ it holds $\mathcal{J}(\Omega^*) = 0$ and thus Ω^* is a global minimiser of $\mathcal{J}$. We show results obtained by first- and second-order shape optimisation methods exploiting automated differentiation.

We ran the optimisation algorithm in three versions. On the one hand, we applied a first-order method with constant step size $\alpha = 1$. On the other hand, we applied two second-order methods with the two different regularisation strategies for the shape Hessian in (65) introduced in (66) and (67). We chose the regularisation parameters δ empirically such that the method performs as well as possible. In the case of (66), we chose $\delta = 0.001$ and in the case of (67) $\delta = 1$. The experiments were conducted on a finite element mesh consisting of 4886 elements with 2522 vertices and polynomial degree 1. In Fig. 10, we can observe the decrease of the objective function as well as of the norm of the shape gradient over 200 iterations for these three algorithmic settings.

Figure 11 shows the initial design as well as the design after 200 iterations of the second-order method with regularisation strategy (67). Note that the improved design is very close to $\Omega^* = (0, 1)^2$, which is a global solution. The initial design was chosen as the disk of radius $\frac{1}{2}$ centred at the point $\left(\frac{1}{2}, \frac{1}{2}\right)^\top$. The objective value was reduced from $5.297 \cdot 10^{-5}$ to $1.0317 \cdot 10^{-9}$.

7.4 Nonlinear elasticity

Here, we illustrate the applicability of the automated shape differentiation and optimisation in the more realistic and more complicated setting of nonlinear elasticity in two space dimensions using a Saint Venant–Kirchhoff material with Young's modulus $E = 1000$ and Poisson ratio $\nu = 0.3$. We consider a two-dimensional cantilever which is clamped on the upper and lower left parts of the boundary, $\Gamma_l^1 = \{0\} \times (0.88, 1)$ and $\Gamma_l^2 = \{0\} \times (0, 0.12)$, respectively, and is subject to a surface force $g_N = (0, -100)^\top$ on $\Gamma_r = \{1\} \times (0.45, 0.55)$. The initial geometry with 3 holes is depicted in Fig. 12a. Let $\Gamma_l := \Gamma_l^1 \cup \Gamma_l^2$ and $H^1_{\Gamma_l}(\Omega)^2$ the subspace of $H^1(\Omega)^2$ with vanishing trace on Γ_l. The displacement $u \in H^1_{\Gamma_l}(\Omega)^2$ under the surface force g_N is given as the solution to the boundary value problem:

$$\int_\Omega S(u) : \nabla v \, \mathrm{d}x = \int_{\Gamma_r} g_N \cdot v \, \mathrm{d}s \tag{77}$$

for all $v \in H^1_{\Gamma_l}(\Omega)^2$. Here, $S(u)$ denotes the Saint Venant–Kirchhoff stress tensor:

$$S(u) = (I_2 + \nabla u)\left[\lambda \mathrm{Tr}\left(\frac{1}{2}(C(u) - I_2)\right) I_2 + \mu (C(u) - I_2)\right], \tag{78}$$

where $C(u) = (I_2 + \nabla u)^\top (I_2 + \nabla u)$ and I_2 is the identity matrix (see also (Allaire et al. 2004, Sec. 8)), and λ and μ

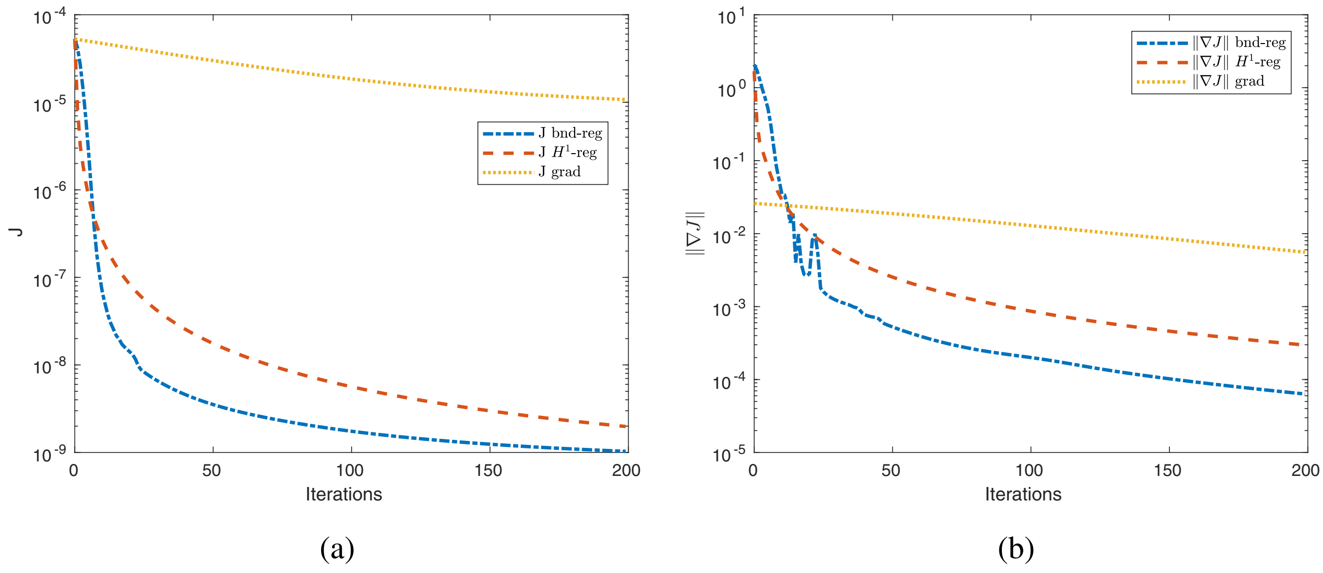


Fig. 10 Convergence behaviour for shape optimisation problem (28) with proposed regularisation strategies (67) and (66) as well as first-order method with constant step size $\alpha = 1$. **a** Behaviour of objective function $\mathcal{J}$. **b** Behaviour of norm of shape gradient $\|\nabla \mathcal{J}(\Omega)\|$

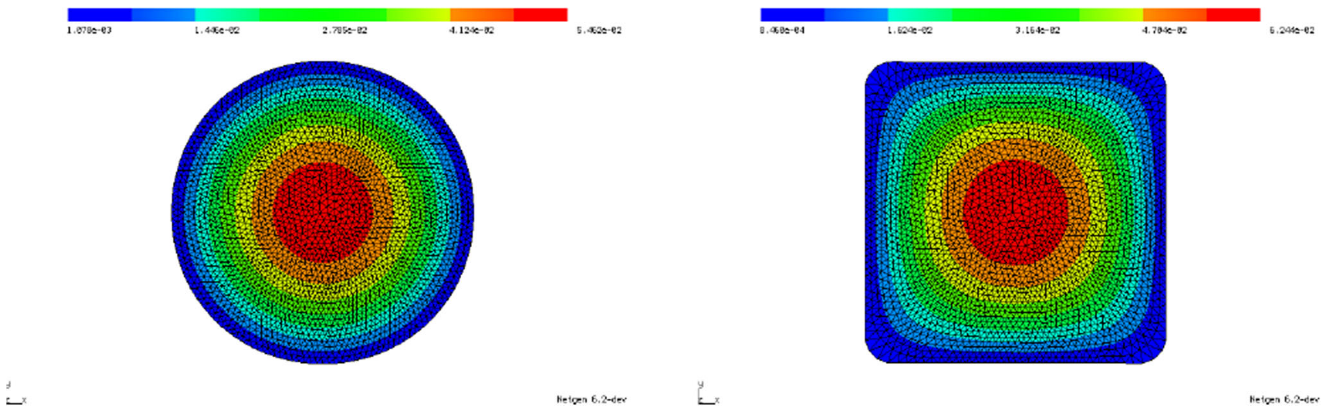


Fig. 11 Shape optimisation for problem (28). Left: Initial design. Right: Improved design after 200 iterations of second-order algorithm with regularisation as proposed in (67). Objective value was reduced from $5.297 \cdot 10^{-5}$ to $1.0317 \cdot 10^{-9}$. Colour shows solution of constraining PDE (28b)

denote the Lamé constants:

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}, \qquad \mu = \frac{E}{2(1+\nu)}. \tag{79}$$

We minimise the functional

$$J(\Omega, u) = \int_\Omega S(u) : \nabla u \,\mathrm{dx} + \alpha \int_\Omega 1 \,\mathrm{dx} \tag{80}$$

with $\alpha = 2.5$ subject to (77) which amounts to maximising the structure's stiffness while bounding the allowed amount of material used.

We remark that the well-posedness of (77) is not clear (see also the discussion in (Allaire et al. 2004, Sec. 8)). Nevertheless, application of the automated shape differentiation and optimisation yields a significant improvement of the initial design. The highly nonlinear PDE constraint (77) is solved by Newton's method. In order to have good starting values, a load stepping strategy is employed, i.e. the load on the right-hand side is gradually increased, the PDE is solved and the solution is used as an initial guess for the next load step. This is repeated until the full load is applied. With these ingredients at hand, Algorithm 1 (i.e. code lines 244–284) can be run. We chose the algorithmic parameters `alpha = 0.1` (as an initial value), `alpha_incr_factor = 1` (i.e. no increase), `gamma = 1e-4` and `epsilon = 1e-7`. Moreover, we used (59) with an additional Cauchy-Riemann term as in (61) with weight $\gamma_{CR} = 10$. The objective value was reduced from 3.125 to 2.635 (volume term from 1.290 to 1.096) in 15 iterations of Algorithm 1. The results were obtained on a mesh consisting of 10,614 elements and 5540 vertices using piecewise linear, globally continuous finite elements.

7.5 Helmholtz equation

In this section, we consider the problem of finding the optimal shape of a scattering object. More precisely, we consider the minimisation of the functional:

$$\int_{\Gamma_r} u\bar{u} \,\mathrm{ds} \tag{81}$$

subject to the Helmholtz equation with impedance boundary conditions on the outer boundary: Find $u \in H^1(\Omega, \mathbf{C})$ such that

$$\int_\Omega [\nabla u \cdot \nabla \bar{w} - \omega^2 u\bar{w}] \,\mathrm{dx} - i\,\omega \int_\Gamma u\bar{w} \,\mathrm{ds} = \int_\Omega f\bar{w} \tag{82}$$

for all $w \in H^1(\Omega, \mathbf{C})$. Here, $\bar{w}$ denotes the complex conjugate of a complex-valued function w, ω denotes the

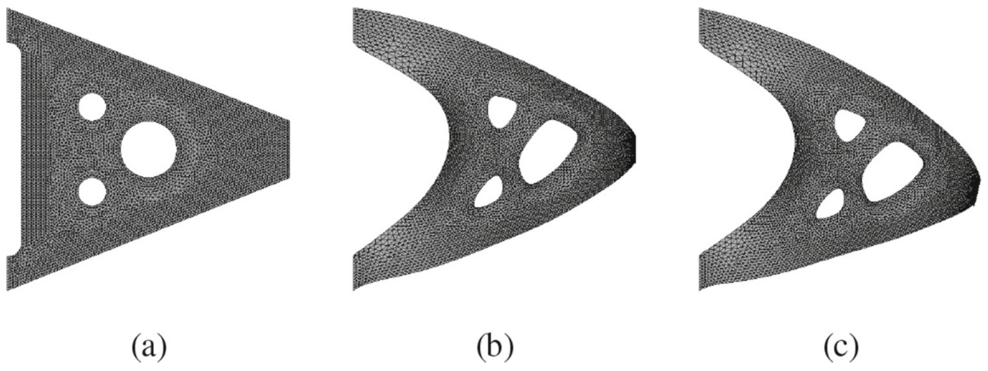


Fig. 12 Initial and optimised geometry of cantilever under vertical force on right-hand side using St. Venant–Kirchhoff model in nonlinear elasticity. **a** Initial geometry. **b** Optimised geometry (reference configuration). **c** Optimised geometry (deformed configuration)

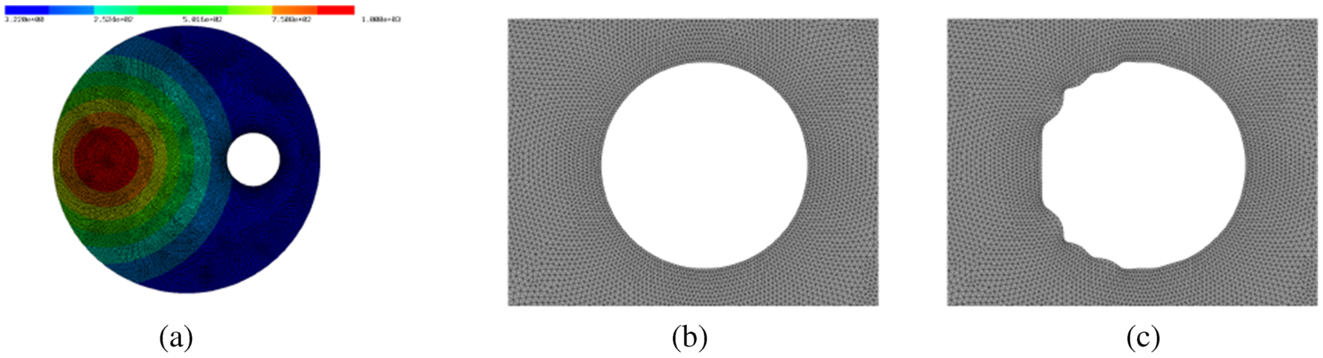


Fig. 13 **a** Geometry with the right-hand side f. **b** Initial shape of scatterer (zoom of geometry in **a**). **c** Optimised shape of scatterer (zoom)

wave number, i denotes the complex unit and the function f on the right-hand side is chosen as

$$f(x_1, x_2) = 10^3 \cdot e^{-9((x_1-0.2)^2+(x_2-0.5)^2)} \tag{83}$$

(see Fig. 13a). Furthermore,

$$\Omega = B((0.5, 0.5)^\top, 1) \setminus B((0.75, 0.5)^\top, 0.15)$$

denotes the domain of interest, $\Gamma = \{(x_1, x_2) : x_1^2+x_2^2 = 1\}$ the outer boundary and $\Gamma_r = \{(x_1, x_2) : x_1^2 + x_2^2 = 1, x_1 \geq 0\}$ the right half of the outer boundary. Here, only the inner boundary $\partial\Omega \setminus \Gamma$ is subject to the shape optimisation. Thus, the aim of this model problem is to find a shape of the scattering object such that the waves are reflected away from Γ_r.

Figure 13b and c show the initial and final shape of the scattering object, respectively. Figure 14 shows the norm of the state for the initial configuration (circular shape of scattering object) and for the optimised configuration. The objective value was reduced from $3.44 \cdot 10^{-3}$ to $3.31 \cdot 10^{-3}$. The forward simulations were performed using piecewise linear finite elements on a triangular grid consisting of 34,803 degrees of freedom. The optimisation stopped after 12 iterations.

7.6 Application to an electric machine

In this section, we consider the setting of three-dimensional nonlinear magnetostatics in $H(\textbf{curl}, \mathsf{D})$ as it appears in the simulation of electric machines. Let $\mathsf{D} \subset \mathbf{R}^3$ denote the computational domain, which consists of ferromagnetic material, air regions and permanent magnets (see Fig. 15). Our aim is to minimise the functional

$$\int_{\Omega_g} |\textbf{curl}u \cdot n - B_d^n|^2 \, \mathrm{dx}, \tag{84}$$

where Ω_g denotes the air gap region of the machine, n denotes an extension of the normal vector to the interior of Ω_g, $B_d^n : \Omega_g \to \mathbf{R}^3$ is a given smooth function and $u \in$

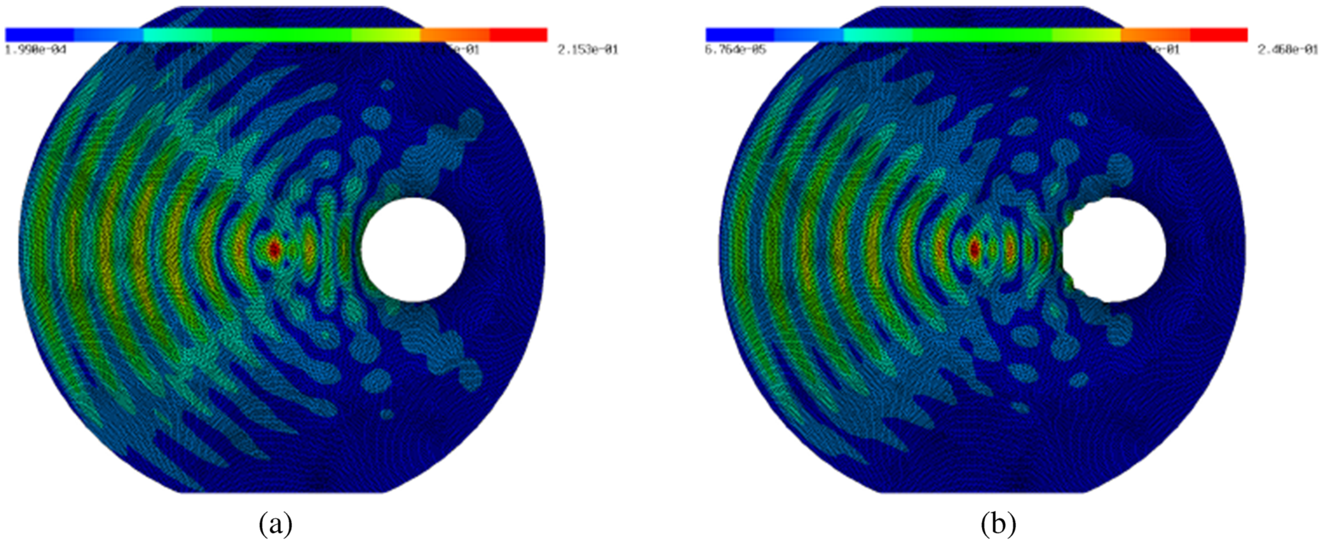


Fig. 14 **a** Absolute value of state u for initial configuration. **b** Absolute value of state u for optimised configuration

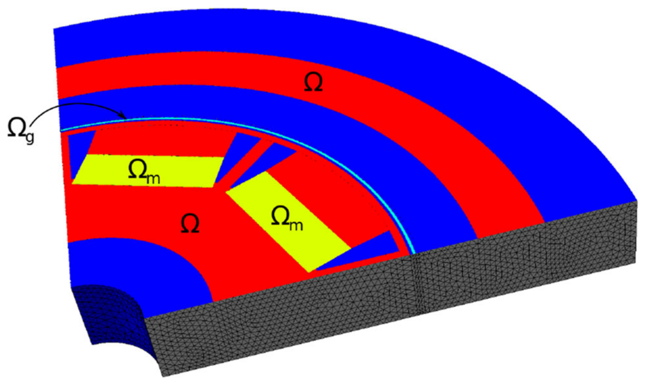


Fig. 15 Geometry of electric motor with subdomains in 2D cross section. The ferromagnetic subdomains Ω are depicted in red, Ω_m corresponds to the permanent magnets. The rest of the computational domain represents air. Furthermore, Ω_g represents the air gap region that is relevant for the cost function (84)

$H_0(\mathbf{curl}, \mathrm{D})$ is the solution to the boundary value problem

$$\int_D \nu_\Omega(|\mathbf{curl} u|)\mathbf{curl} u\cdot\mathbf{curl} w+\delta u\cdot w \text{ dx} = \int_{\Omega_m} M\cdot\mathbf{curl} w \text{ dx} \tag{85}$$

for all $w \in H_0(\mathbf{curl}, \mathrm{D})$. Here, $\Omega \subset \mathrm{D}$ denotes the union of the ferromagnetic parts of the electric machine, Ω_m denotes the permanent magnets subdomain and

$$\nu_\Omega = \chi_\Omega(x)\hat{\nu}(|\mathbf{curl} u|) + \chi_{\mathrm{D}\setminus\Omega}(x)\nu_0 \tag{86}$$

denotes the magnetic reluctivity, which is a nonlinear function $\hat{\nu}$ inside the ferromagnetic regions and equal to a constant ν_0 elsewhere. Furthermore, $\delta > 0$ is a small regularisation parameter and $M : \mathrm{D} \to \mathbf{R}^3$ denotes the magnetisation in the permanent magnets. The nonlinear function $\hat{\nu}$ satisfies a Lipschitz condition and a strong monotonicity condition such that problem (85) is well-posed. The goal of minimising the cost function (84) is to obtain a design which exhibits a smooth rotation pattern. Note that in this particular example we do not consider rotation of the machine, but rather a fixed rotor position, and there are no electric currents present. We refer the reader to (Gangl and Sturm 2019, Sec. 6) for a more detailed description of the problem and to (Gangl et al. 2015) for a 2D version of the same problem.

As mentioned in Section 4, the transformation Φ_t used in (25) depends on the differential operator. For the **curl**-operator, we have

$$\begin{aligned} \Phi_t(u) &= \partial T_t^{-\top}(u \circ T_t^{-1}) \qquad \text{and} \\ (\mathbf{curl}(\Phi_t(u))) \circ T_t &= \tfrac{1}{\det(\partial T_t)}\partial T_t \mathbf{curl}(u), \end{aligned}$$

see e.g. (Monk 2003, Section 3.9). Thus, the variational (85) can be defined as follows.

```
from math import pi
nu0 = 1e7 / (4*pi)
delta = 0.1

F = Id(3)
c1 = 1/Det(F) * F
c2 = Inv(F).trans

def EquationIron(u,w):
    return ( nuIron(Norm(c1*curl(u))) * (c1*
    curl(u))*(c1*curl(w)) + delta*(c2*u)*(c2*w
    ) )*Det(F)*dx("iron")

def EquationAir(u,w):
    return (nu0*(c1*curl(u))*(c1*curl(w)) +
    delta*(c2*u)*(c2*w))*Det(F)*dx("air|airgap
    ")

def EquationMagnets(u,w):
    return (nu0*(c1*curl(u))*(c1*curl(w)) +
    delta*(c2*u)*(c2*w)-magn*(c1*curl(w))) *
    Det(F)*dx("magnets")

def Equation(u,w):
   return EquationIron(u,w) + EquationAir(u,w)
    + EquationMagnets(u,w)
```

Here, the computational domain consists of a subdomain representing the ferromagnetic part of the machine (``iron'') and a subdomain comprising the permanent magnets (``magnets''); the union of all air subdomains, including the air gap between rotating and fixed part of the machine, is given by ``air|airgap'' (see Fig. 15).

Moreover, nuIron denotes the nonlinear reluctivity function $\hat{\nu}$ and magn contains the magnetisation direction of the permanent magnets. Likewise, the objective function can be defined as follows:

```
def Cost(u):
    return (InnerProduct(c1*curl(u),n2D) - Bd)
    *Det(F)*dx("airgap")
```

where n2D and Bd represent the extension of the normal vector to the interior of the air gap and the desired curve, respectively. For the definition of all quantities, we refer the reader to Online Resource 1. The shape differentiation as well as the optimisation loop now works in the same way as in the previous examples. Figure 16 shows the initial design of the motor as well as the optimised design obtained after 11 iterations of Algorithm 1 with $\gamma = 0$. The experiment was conducted using a tetrahedral finite element mesh consisting of 13,440 vertices, 57,806 elements and Nédélec elements of order 2 (resulting in a total of 323,808 degrees of freedom). The objective value was reduced from $2.5944 \cdot 10^{-8}$ to $4.565 \cdot 10^{-10}$ in the course of the first order optimisation algorithm after 11 iterations. It can be seen from Fig. 17 that the difference between the quantity $\mathbf{curl}(u) \cdot n$ and the desired curve B_d^n inside the air gap decreases significantly.

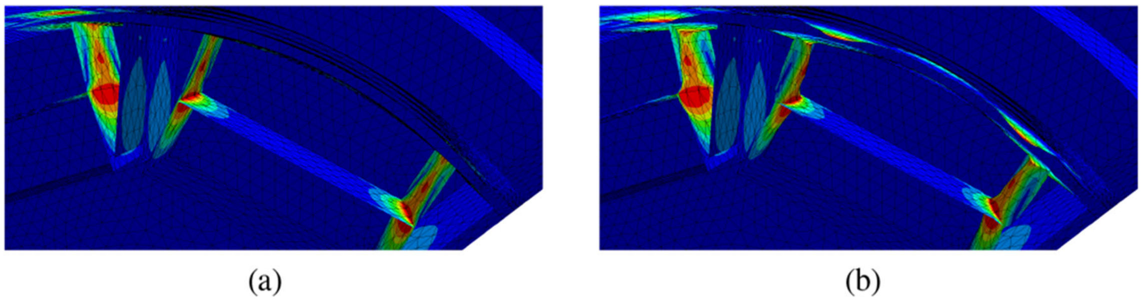


Fig. 16 **a** Initial design of electric machine. **b** Optimised design

7.7 Surface PDEs

Next, we also show the application of a shape optimisation algorithm to a problem constrained by a surface PDE. We solve problem (48–49) with $u_d = 0$, $f(x_1, x_2, x_3) = x_1x_2x_3$ and initial shape $M = S^2$ the unit sphere in $\mathbf{R}^3$. We applied a first-order algorithm with a line search. Figure 18 shows the initial geometry as well as the decrease of the objective function and of the norm of the shape gradient. The objective value was reduced from $7.08 \cdot 10^{-4}$ to $9.88 \cdot 10^{-9}$. Figure 19 shows the final design which was obtained after 575 iterations from two different perspectives. The experiment was conducted using a surface mesh with 332 vertices and 660 faces and polynomial degree 3 (resulting in 2972 degrees of freedom).

7.8 Time-dependent PDE using space-time method

In this section, we illustrate a non-standard situation where the fully automated shape differentiation using the command `.DiffShape()` fails, but the semi-automated way can be used to compute the shape derivative.

The situation is that of a parabolic PDE constraint in a space-time setting where the time variable is considered as just another space variable. Let $T > 0$ and $\Omega \subset \mathbf{R}^d$ be given and define the space-time cylinder $Q := \Omega \times (0, T) \subset \mathbf{R}^{d+1}$. For given smooth functions u_d and f defined on Q, we consider the problem:

$$\min_{\Omega} \quad \int_Q |u - u_d|^2 \, \mathrm{d}(x, \tau) \tag{87}$$

$$\text{s.t.} \int_Q \partial_\tau u v + \nabla_x u \cdot \nabla_x v \, \mathrm{d}(x, \tau) = \int_Q f v \, \mathrm{d}(x, \tau) \tag{88}$$

for all v in the Bochner space $L^2(0, T; H_0^1(\Omega))$ where the state u is to be sought in the Bochner space $L^2(0, T; H_0^1(\Omega)) \cap H^1(0, T; H^{-1}(\Omega))$ with the initial condition $u(x, 0) = 0$. Here, $\nabla_x = (\partial_{x_1}, \dots, \partial_{x_d})^\top$ denotes the spatial gradient and ∂_τ the time derivative. Note that we denote the time variable by τ in order not to interfere with the shape parameter t. We refer the interested reader to (Steinbach 2015) for details on this space-time formulation of the PDE constraint. As it is outlined there, the PDE can be solved numerically by choosing the same ansatz and test space consisting of piecewise linear and globally continuous finite element functions on Q.

For simplicity, we restrict ourselves to the case where $d = 1$, i.e. to the case where the spatial domain Ω is an

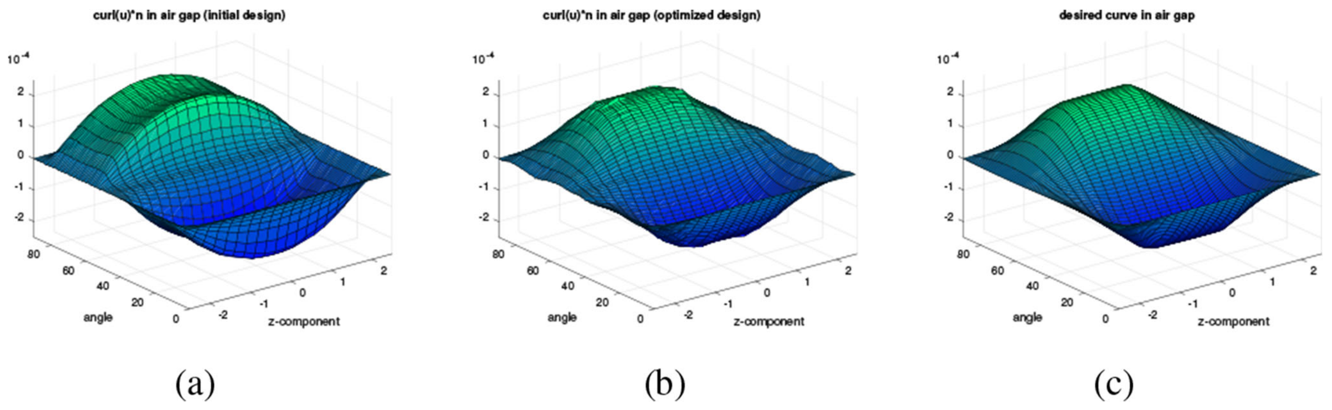
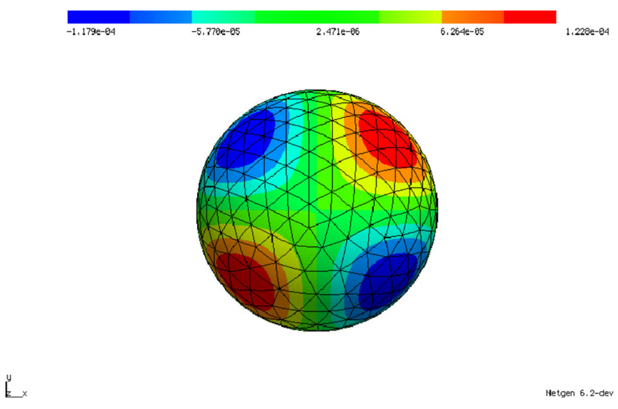
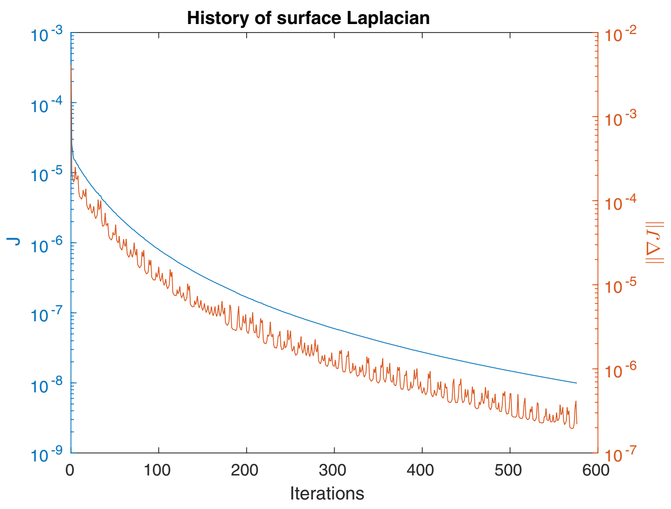


Fig. 17 Improvement of $\mathbf{curl}u \cdot n$ as a function of z and the angle φ for a fixed radius r inside Ω_g compared to desired curve B_d^n. **a** $\mathbf{curl}u \cdot n$ for initial configuration. **b** $\mathbf{curl}u \cdot n$ for optimised configuration after 10 iterations. **c** Desired curve B_d^n in polar coordinates as function of z and the angle φ for a fixed radius r



(a)



(b)

Fig. 18 **a** Initial geometry for shape optimisation with respect to surface PDE (48–49). **b** History of objective value and norm of shape gradient using a first-order algorithm with line search

interval. We are interested in the shape derivative of problem (87) with respect to spatial perturbations, i.e., with respect to transformations of the form:

$$T_t(x,\tau) = \begin{pmatrix} x + tV(x,\tau) \\ \tau \end{pmatrix}$$

where $V \in C^{0,1}(Q, \mathbf{R}^d)$ and $t \geq 0$. We recall the notation $F_t(x,\tau) = \partial T_t(x,\tau)$. By this choice of transformation T_t we exclude an unwanted deformation of the space-time cylinder into the time direction as the time horizon $T > 0$ is assumed to be fixed.

Following the lines of the previous examples, we can define the cost function, the PDE and the Lagrangian:

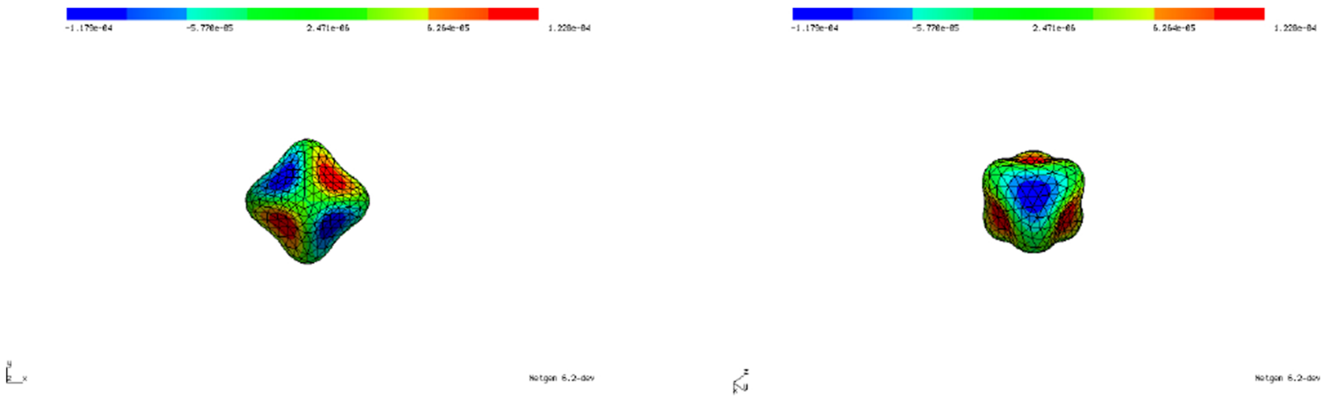
```
F11 = Parameter(1)
F12 = Parameter(0)
F21 = Parameter(0)
F22 = Parameter(1)
F = CoefficientFunction((F11, F12, F21, F22),
    dims=(2,2))

def Cost_vol(u):
    return (u-ud)*(u-ud)*Det(F)*dx

def Equation_vol(u,v):
   return ( (Inv(F.trans)*grad(u))[1]*v + (Inv
    (F.trans)*grad(u))[0]*(Inv(F.trans)*grad(v
    ))[0]-f*v)*Det(F)*dx

G_vol = Cost_vol(gfu) + Equation_vol(gfu,gfp)
```

Here, gfu denotes the solution to the state (88) and gfp the solution to the adjoint equation, which is posed



(a) (b)

Fig. 19 **a** Final design after 575 iterations. **b** Different view of **a**

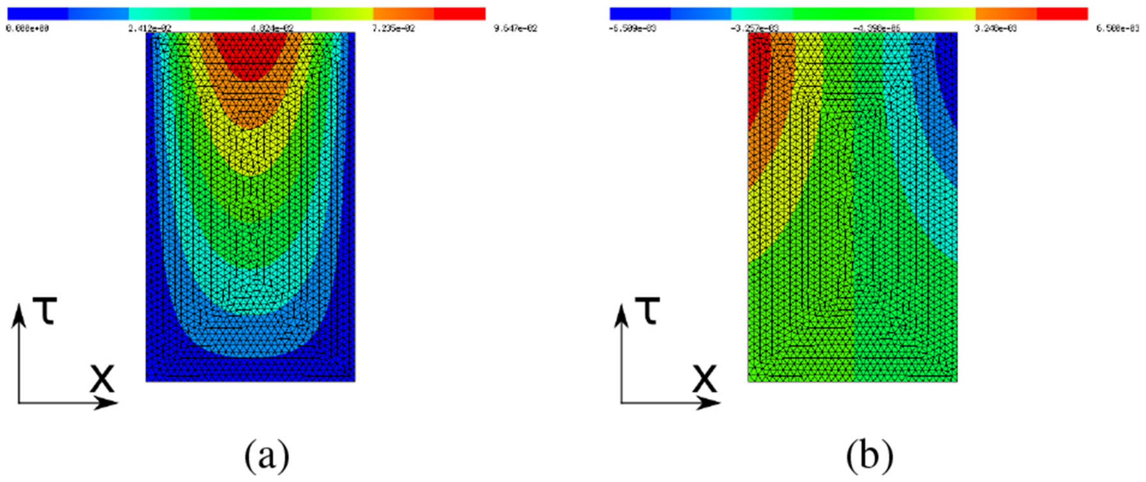


Fig. 20 Space-time cylinder in initial configuration. **a** Solution to state (88). **b** Time-dependent descent vector field $\tilde{W}$ obtained by solving (91)

backward in time and reads in its strong form:

$$\begin{aligned} -\partial_\tau p - \Delta p &= -2(u-u_d)(x,\tau) \in Q, \\ p(x,\tau) &= 0 (x,\tau) \in \partial\Omega \times (0,T), \\ p(x,T) &= 0 x \in \Omega. \end{aligned}$$

We can compute the shape derivative similarly to the previous examples by means of formula (10), i.e.,

$$\frac{d}{dt}\mathcal{J}(\Omega_t)\bigg|_{t=0} = \left(\frac{dG}{dT_t}\frac{dT_t}{dt} + \frac{dG}{dF_t}\frac{dF_t}{dt}\right)\bigg|_{t=0}. \qquad (89)$$

However, it must be noted that in this special situation we have

$$\frac{dT_t}{dt} = \begin{pmatrix} V \\ 0 \end{pmatrix} \text{ and } \frac{dF_t}{dt} = \begin{pmatrix} \partial_x V & \partial_\tau V \\ 0 & 0 \end{pmatrix}. \qquad (90)$$

The shape derivative can now be obtained as follows: Given a mesh of the space-time cylinder Q, we define an $\mathbf{R}^d$-valued H^1-space to represent the vector field V (here we assumed $d = 1$, thus we are facing a scalar-valued space). The shape derivative is a linear functional on this space and is obtained by plugging in (90) into (89):

```
VEC1 = H1(mesh, order=1)
V = VEC1.TestFunction()

dJOmega_para = LinearForm(VEC1);
dJOmega_para += G_vol.Diff(x, V)
dJOmega_para += G_vol.Diff(F11, grad(V)[0])
dJOmega_para += G_vol.Diff(F12, grad(V)[1])
```

Remark 7 The fully automated shape differentiation command .DiffShape(V) cannot be used here because the vector field V has fewer components than the dimension of the mesh. On the other hand, if V was chosen as a vector field with $d + 1$ components, the command

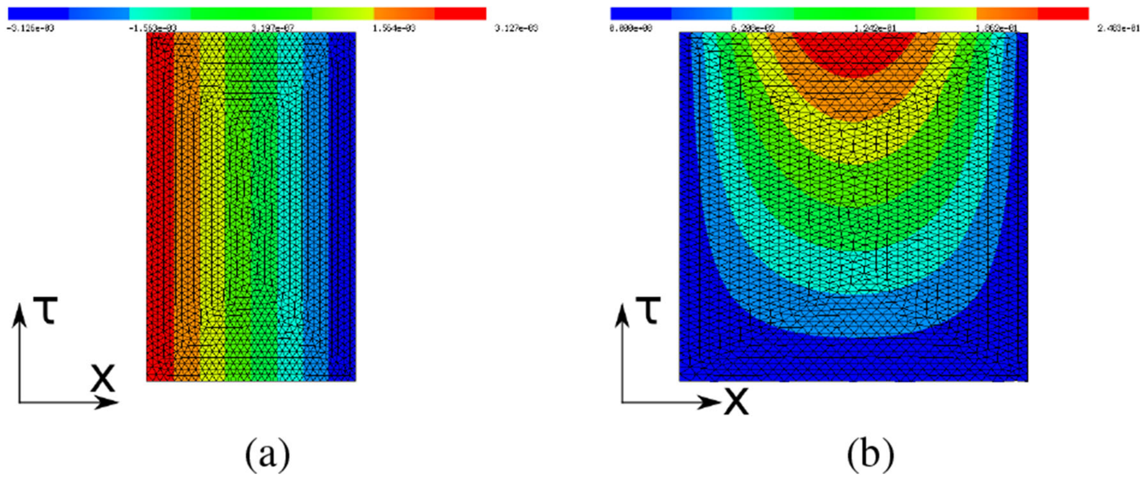


Fig. 21 Space-time cylinder in initial and final configurations. **a** Time-independent descent vector field W obtained by averaging $\tilde{W}$. **b** Solution to state (88) on final design

`.DiffShape(V)` would evaluate formula (89), but would assume $\frac{dT_t}{dt} = V = (V_x, V_\tau)^\top$ and

$$\frac{dF_t}{dt} = \partial V = \begin{pmatrix} \partial_x V_x & \partial_\tau V_x \\ \partial_x V_\tau & \partial_\tau V_\tau \end{pmatrix}$$

and could not take into account the special situation at hand as shown in (90). This example is meant to illustrate the greater flexibility of the semi-automated compared with the fully automated shape differentiation.

Code lines 388–391 show the computation of the shape derivative in the direction of an $\mathbf{R}^d$-valued function $V = V(x, \tau)$ (recall $d = 1$ here). However, using τ-dependent vector fields would result in time-dependent optimal shapes, which is often not desired. Rather, one is interested in vector fields of the form $V = V(x) \neq V(x, \tau)$ which still yield a descent, i.e. $D\mathcal{J}(\Omega)(V) < 0$. This can be achieved as follows:

1. Compute a time-dependent shape gradient $\tilde{W}$ by solving

$$\int_Q \partial \tilde{W} : \partial \tilde{V} + \tilde{W} \cdot \tilde{V} = D\mathcal{J}(\Omega)(\tilde{V}) \quad \text{for all } \tilde{V}, \quad (91)$$

2. Set $W(x, \tau) = \frac{1}{T} \int_0^T \tilde{W}(x, s)\mathrm{d}s$.

Note that $W(x, \tau)$ is constant in τ. Then we see by plugging in $\tilde{V} = -W$ in (91) that

$$D\mathcal{J}(\Omega)(-W) < 0,$$

thus $-W$ is a descent direction.

We used this strategy to solve problem (87) for $d = 1$ with the data $u_d(x, \tau) = x(1-x)\tau$, $f(x, \tau) = x(1-x) + 2\tau$ numerically starting out from the initial domain $\Omega_{init} = (0.2, 0.8)$ and the fixed time interval $(0, T) = (0, 1)$. Note that the data is chosen such that the domain $\Omega^\star = (0, 1)$ is a global solution to problem (87).

Figure 20 shows the initial design together with the solution to the state equation and the time-dependent descent vector field $\tilde{W}$ obtained as solution of (91). Figure 21a shows the averaged vector field W which is independent of τ and yields a uniform deformation of the space-time cylinder. The final design after 293 iterations can be seen in Fig. 21b. The cost function was reduced from $4.65 \cdot 10^{-3}$ to $9.95 \cdot 10^{-9}$.

For more details on the implementation of this example, we refer to Online Resource 1 and for more details on shape optimisation in a space-time setting, we refer the interested reader to Köthe (2020).

8 Conclusion

We showed how to obtain first- and second-order shape derivatives for unconstrained and PDE-constrained shape optimisation problems in a semi-automatic and fully automatic way in the finite element software package `NGSolve`. We verified the proposed method numerically by Taylor tests and by showing its successful application to several shape optimisation problems. We believe that this intuitive approach can help research scientists working in the field of shape optimisation to further improve numerical methods on the one hand, and product engineers working with `NGSolve` to design devices in an optimal fashion on the other hand.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/s00158-020-02742-w.

Acknowledgements Open access funding provided by Graz University of Technology. We would like to thank Christian Köthe for his contribution to Section 7.8.

Funding Michael Neunteufel has been funded by the Austrian Science Fund (FWF) project W1245.

Compliance with ethical standards

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

Replication of results The python scripts which were used for the results presented in this paper are available in Online Resource 1. All computations were performed using `NGSolve` version V6.2.2004.

References

Allaire G (2007) Conception optimale des structures. Springer, New York

Allaire G, Cancès E, Vié JL (2016) Second-order shape derivatives along normal trajectories, governed by Hamilton-Jacobi equations. Struct Multidiscip Optim 54(5):1245–1266. https://doi.org/10.1007/s00158-016-1514-2

Allaire G, Dapogny C, Jouve F (2021) Shape and topology optimization. to appear in Handbook of Numerical Analysis, 22. https://www.elsevier.com/books/geometric-partial-differential-equations-part-2/nochetto/978-0-444-64305-6

Allaire G, Jouve FJ, Toader A-M (2004) Structural optimization using sensitivity analysis and a level-set method. J Comput Phys 194(1):363–393. https://doi.org/10.1016/j.jcp.2003.09.032. http://www.sciencedirect.com/science/article/pii/S002199910300487X

Alnæs MS, Logg A, Ølgaard KB, Rognes ME, Wells GN (2014) Unified form language. ACM Transactions on Mathematical Software 40(2):1–37. https://doi.org/10.1145/2566630

Berggren M (2010) A unified discrete-continuous sensitivity analysis method for shape optimization. In: Applied and numerical partial differential equations, 15 of Comput. Methods Appl. Sci., pp 25–39, Springer, New York

Burger M (2002) A framework for the construction of level set methods for shape optimization and reconstruction. Interfaces and Free Boundaries 5:301–329

de Gournay F (2006) Velocity extension for the level-set method and multiple eigenvalues in shape optimization. SIAM J Control Optim 45(1):343–367. https://doi.org/10.1137/050624108

Delfour MC, Zolésio J-P (2011) Shapes and geometries, volume 22 of Advances in Design and Control, 2nd edn. Society for Industrial and Applied Mathematics (SIAM), Philadelphia

Delfour MC, Zolésio JP (2011) Shapes and geometries. Society for Industrial and Applied Mathematics

Demkowicz L (2004) Projection-based interpolation. ICES Report 4(3):1–22

Dokken JS, Mitusch SK, Funke SW (2020) Automatic shape derivatives for transient PDEs in FEniCS and Firedrake. arXiv e-prints, arXiv:2001.10058

Eppler K, Harbrecht H, Schneider R (2007) On convergence in elliptic shape optimization. SIAM Journal on Control and Optimization 46(1):61–83. https://doi.org/10.1137/05062679x

Evans L (2010) Partial differential equations. American Mathematical Society, Providence. With the collaboration of Marc Schoenauer (INRIA) in the writing of Chapter 8

Feppon F, Allaire G, Bordeu F, Cortial J, Dapogny C (2019) Shape optimization of a coupled thermal fluid-structure problem in a level set mesh evolution framework. SeMA 76(3):413–458. https://doi.org/10.1007/s40324-018-00185-4

Gangl P, Langer U, Laurain A, Meftahi H, Sturm K (2015) Shape optimization of an electric motor subject to nonlinear magnetostatics. SIAM J Sci Comput 37(6):B1002–B1025. https://doi.org/10.1137/15100477X

Gangl P, Sturm K (2019) Asymptotic analysis and topological derivative for 3D quasi-linear magnetostatics. arXiv:1908.10775

Ham DA, Mitchell L, Paganini A, Wechsung F (2019) Automated shape differentiation in the unified form language. Struct Multidiscip Optim 60(5):1813–1820. https://doi.org/10.1007/s00158-019-02281-z

Henrot A, Pierre M (2005) Variation et optimisation de formes : une analyse géométrique. Springer, Berlin

Hintermüller M, Laurain A (2008) Electrical impedance tomography: from topology to shape. Control Cybern 37(4):913–933

Hinze M, Pinnau R, Ulbrich M, Ulbrich S (2009) Optimization with pde constraints. 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

Hömberg D, Sokolowski J (2003) Optimal shape design of inductor coils for surface hardening. SIAM J Control Optim 42(3):1087–1117. https://doi.org/10.1137/s0363012900375822

Iglesias JA, Sturm K, Wechsung F (2018) Two-dimensional shape optimization with nearly conformal transformations. SIAM Journal on Scientific Computing 40(6):A3807–A3830. https://doi.org/10.1137/17m1152711

Ito K, Kunisch K (2008) Lagrange multiplier approach to variational problems and applications. Society for Industrial and Applied Mathematics, Philadelphia

Köthe C (2020) PDE-constrained shape optimization for coupled problems using space-time finite elements. Master's Thesis, Graz University of Technology

Laurain A (2018) A level set-based structural optimization code using FEniCS. Struct Multidiscip Optim 58(3):1311–1334. https://doi.org/10.1007/s00158-018-1950-2

Monk P (2003) Finite element methods for maxwell's equations. Numerical Mathematics and Scientific Computation. Clarendon Press

Murat F, Simon J (1976) Sur le contrôle par un domaine géométrique. Rapport 76015, Université Pierre et Marie Curie, Paris

Novruzi A, Pierre M (2002) Structure of shape derivatives. J Evol Equ 2(3):365–382. https://doi.org/10.1007/s00028-002-8093-y

Novruzi A, Roche JR (2000) Newton's method in shape optimisation: A three-dimensional case. Bit Numerical Mathematics 40(1):102–120. https://doi.org/10.1023/a:1022370419231

Paganini A, Sargheini S, Hiptmair R, Hafner C (2015) Shape optimization of microlenses. Opt Express 23(10):13099. https://doi.org/10.1364/oe.23.013099

Paganini A, Sturm K (2019) Weakly normal basis vector fields in RKHS with an application to shape Newton methods. SIAM J Numer Anal 57(1):1–26. https://doi.org/10.1137/17m1131623

Schiela A, Ortiz J (2017) Second order directional shape derivatives. https://epub.uni-bayreuth.de/3251/

Schmidt S (2014) A two stage CVT / eikonal convection mesh deformation approach for large nodal deformations. arXiv e-prints, arXiv:1411.7663

Schmidt S (2018) Weak and strong form shape Hessians and their automatic generation. SIAM Journal on Scientific Computing 40(2):C210–C233. https://doi.org/10.1137/16m1099972

Schmidt S, Ilic C, Schulz V, Gauger N (2013) Three-dimensional large-scale aerodynamic shape optimization based on shape calculus. AIAA J 51(11):2615–2627. https://doi.org/10.2514/1.J052245

Schmidt S, Ilic C, Schulz V, Gauger NR (2011) Airfoil design for compressible inviscid flow based on shape calculus. Optim Eng 12(3):349–369. https://doi.org/10.1007/s11081-011-9145-3

Schöberl J (2014) C++11 implementation of finite elements in NGSolve. Institute for Analysis and Scientific Computing, Vienna University of Technology, 30

Schulz VH (2014) A riemannian view on shape optimization. Found Comput Math 14(3):483–501. https://doi.org/10.1007/s10208-014-9200-5

Simon J (1989) Second variations for domain optimization problems. Control theory of distributed parameter systems and applications 91:361–378

Sokołowski J, Zolésio J-P (1992) Introduction to shape optimization, volume 16 of Springer Series in Computational Mathematics. Springer-Verlag, Berlin. Shape sensitivity analysis

Steinbach O (2015) Space-Time Finite Element Methods for Parabolic Problems. Computational Methods in Applied Mathematics 15(4):551–566. https://doi.org/10.1515/cmam-2015-0026

Sturm K (2015) Minimax Lagrangian approach to the differentiability of nonlinear PDE constrained shape functions without saddle point assumption. SIAM Journal on Control and Optimization 53(4):2017–2039. https://doi.org/10.1137/130930807

Sturm K (2015) Shape differentiability under non-linear PDE constraints. In: New trends in shape optimization, 166 of Internat. Ser. Numer. Math., pp 271–300, Birkhäuser/Springer, Cham