Applied Mathematics and Computation 442 (2023) 127731

Contents lists available at ScienceDirect

Applied Mathematics and Computation

journal homepage: www.elsevier.com/locate/amc

MooAFEM: An object oriented Matlab code for higher-order adaptive FEM for (nonlinear) elliptic PDEs

Michael Innerberger[1,*], Dirk Praetorius

TU Wien, Institute of Analysis and Scientific Computing, Wiedner Hauptstr. 8–10/E101/4, Wien 1040, Austria

ARTICLE INFO

Article history:
Received 15 June 2022
Revised 4 November 2022
Accepted 15 November 2022
Available online 19 December 2022

2010 MSC:
35-04
65N30
65N50
68N19

Keywords:
Finite element software
Adaptivity
Object oriented design
Higher-order FEM
Iterative linearization

ABSTRACT

We present an easily accessible, object oriented code (written exclusively in MATLAB) for adaptive finite element simulations in 2D. It features various refinement routines for triangular meshes as well as fully vectorized FEM ansatz spaces of arbitrary polynomial order and allows for problems with very general coefficients. In particular, our code can handle problems typically arising from iterative linearization methods used to solve nonlinear PDEs. Due to the object oriented programming paradigm, the code can be used easily and is readily extensible. We explain the basic principles of our code and give numerical experiments that underline its flexibility as well as its efficiency.

© 2022 The Author(s). Published by Elsevier Inc.
This is an open access article under the CC BY license
(http://creativecommons.org/licenses/by/4.0/)

1. Introduction

In today's ever-changing landscape of mathematical software, MATLAB has successfully reinforced its role as a *de facto* standard for the development and prototyping of various kinds of algorithms for numerical simulations. In particular, it has proven to be an excellent tool for academic education, e.g., in the field of the partial differential equations (PDEs), which can be solved by the finite element method (FEM); see, e.g., [1]. While the impact of vectorization for efficient FEM codes in MATLAB is well-known in the literature [2–6], we show, on the one hand, how FEM codes in MATLAB can massively benefit from object oriented programming (OOP) and, on the other hand, how OOP even simplifies the efficient implementation of FEM for nonlinear PDEs.

In this work, we present our own freely available code MooAFEM (MATLAB object oriented adaptive FEM) [7]. It is specifically tailored to (higher-order) adaptive FEM simulations involving linear second-order elliptic PDEs with very general coefficients: Let $\Omega \subset \mathbb{R}^2$ be a bounded domain with polygonal boundary $\partial\Omega$ that is split into *Robin*, *Neumann*, and *Dirichlet*

* Corresponding author.
E-mail addresses: Michael.Innerberger@asc.tuwien.ac.at (M. Innerberger), Dirk.Praetorius@asc.tuwien.ac.at (D. Praetorius).

[1] The authors acknowledge support through the Austrian Science Fund (FWF) through the doctoral school *Dissipation and dispersion in nonlinear PDEs* (grant W1245), the special research program *Taming complexity in PDE systems* (grant SFB F65), and the standalone project *Computational nonlinear PDEs* (grant P33216).

https://doi.org/10.1016/j.amc.2022.127731
0096-3003/© 2022 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY license
(http://creativecommons.org/licenses/by/4.0/)

boundary, i.e., $\partial\Omega = \overline{\Gamma}_R \cup \overline{\Gamma}_N \cup \overline{\Gamma}_D$ and Γ_R, Γ_N, Γ_D are pairwise disjoint. The model problem then reads

$$-\mathrm{div}\mathbf{A}\nabla u + \boldsymbol{b}\cdot\nabla u + cu = f - \mathrm{div}\boldsymbol{f} \quad \text{in } \Omega, \tag{1a}$$
$$\alpha u + \mathbf{A}\nabla u\cdot\boldsymbol{n} = \gamma \quad \text{on } \Gamma_R, \tag{1b}$$
$$\mathbf{A}\nabla u\cdot\boldsymbol{n} = \phi \quad \text{on } \Gamma_N, \tag{1c}$$
$$u = 0 \quad \text{on } \Gamma_D. \tag{1d}$$

Note that also inhomogeneous Dirichlet data $g \in H^{1/2}(\Gamma_D)$ can be handled by the usual superposition ansatz, i.e., $u = u_D + \widehat{g}$, where $\widehat{g} \in H^1(\Omega)$ is a lifting of the inhomogeneous boundary data (e.g., by $L^2(\Omega)$-projection or by nodal interpolation in the discrete case) and $u_D \in H^1(\Omega)$ with $u_D = 0$ on Γ_D solves an appropriate variant of (1); see, e.g., [8–11]. Due to the modular design of MooAFEM, also other methods for incorporating inhomogeneous Dirichlet data (such as penalty methods [12] or Nitsche's method [13]) can be implemented analogously to Robin-type boundary conditions.

MooAFEM is able to discretize problem (1) with conforming FEM spaces of arbitrary polynomial order and, by use of OOP, allows that the coefficients $\mathbf{A}$, $\boldsymbol{b}$, c, and α, as well as the data f, $\boldsymbol{f}$, γ, and ϕ can be any function that depends on a spatial variable; in particular, FEM functions are also valid as coefficients. In contrast to existing object oriented MATLAB FEM codes like [14], this allows to treat also non-linear PDEs (as well as nonlinear boundary conditions), since these must be solved by iterative linearization techniques in practice, which lead to problems of the form (1), where the coefficients as well as the data depend on previous iterates (see Section 5.3 below). The implementation of such (nonlinear) problems with our code is fairly easy and extensively documented below. Furthermore, all computations (e.g., function access, quadrature, and assembly) are vectorized efficiently such that the measured computation time scales optimally with the number of degrees of freedom.

Through proper referencing mechanics enabled by the use of OOP in MATLAB, all vectorizations are hidden behind an application layer. Keeping class design lean and hierarchies flat, together with consistent, descriptive naming and an extensive documentation, this application layer is highly readable while still granting the user full control over the underlying algorithms (e.g., quadrature rules and interpolation methods). Furthermore, separation into several loosely coupled modules, an extensive unit-test suite covering the critical parts of the code, and well-designed interfaces for mathematical operations allow for facile extension of our code. Thus, MooAFEM combines the flexibility and reusability of object oriented software with the accessibility of the high-level scripting language MATLAB, providing a much needed comprehensive, flexible, and efficient library for AFEM research with only about one thousand executable lines.

Software requirements

The MooAFEM package requires an installation of MATLAB version R2020b (v9.9) or later. We stress that, besides the core MATLAB libraries, no other toolboxes are required. For this reason, it may also be ported to Octave with reasonable effort (although this was never attempted).

Outline

We first comment on the precise scope of our software package and reinforce the arguments in favor of object oriented design in Section 2. In particular, we give a schematic algorithm for AFEM that showcases the natural syntax of the MooAFEM package. The general structure of the code is outlined in Section 3: an exhaustive overview over available features is given and the most important ones are explained in more depth. In Section 4, we explain the underlying memory layout of the data structures used in MooAFEM and other implementational aspects. Finally, we extend the basic AFEM algorithm from Section 2 to more interesting examples in Section 5: higher-order FEM, goal-oriented FEM, and iterative linearization of nonlinear equations.

2. Adaptive algorithm and importance of OOP

2.1. Adaptive FEM

To solve equation (1), we employ FEM with underlying triangulation $\mathcal{T}_H$ of Ω. To this triangulation, we associate the FEM space $\mathcal{S}^p(\mathcal{T}_H) := \mathcal{P}^p(\mathcal{T}_H) \cap H^1(\Omega)$ with

$$\mathcal{P}^p(\mathcal{T}_H) := \{v \in L^2(\Omega) \,|\, v|_T \text{ is a polynomial of degree p for all } T \in \mathcal{T}_H\}.$$

With $H^1_D(\Omega) := \{v \in H^1(\Omega) \,|\, v|_{\Gamma_D} = 0\}$, we set $\mathcal{S}^p_D(\mathcal{T}_H) := \mathcal{S}^p(\mathcal{T}_H) \cap H^1_D(\Omega)$. The discrete weak form of (1) then reads: Find $u \in \mathcal{S}^p_D(\mathcal{T}_H)$ such that

$$\begin{aligned} a(u_H, v_H) &:= \int_\Omega \mathbf{A}\nabla u_H\cdot\nabla v_H + \boldsymbol{b}\cdot\nabla u_H v_H + c\,u_H v_H \,\mathrm{d}x + \int_{\Gamma_R} \alpha\, u_H v_H \,\mathrm{d}s \\ &= \int_\Omega f v_H + \boldsymbol{f}\cdot\nabla v_H \,\mathrm{d}x + \int_{\Gamma_N} \phi\, v_H \,\mathrm{d}s + \int_{\Gamma_R} \gamma\, v_H \,\mathrm{d}s =: F(v_H) \quad \text{for all } v_H \in \mathcal{S}^p_D(\mathcal{T}_H). \end{aligned} \tag{2}$$

```
1  mesh = Mesh.loadFromGeometry('unitsquare');
2  fes = FeSpace(mesh, LowestOrderH1Fe);
3  u = FeFunction(fes);
4  blf = BilinearForm(fes);
5  blf.a = Constant(mesh, 1);
6  lf = LinearForm(fes);
7  lf.f = Constant(mesh, 1);
8
9  while mesh.nElements < 1e6
10   A = assemble(blf);
11   F = assemble(lf);
12   freeDofs = getFreeDofs(fes);
13   u.setFreeData(A(freeDofs,freeDofs) \ F(freeDofs));
14
15   hT = sqrt(getAffineTransformation(mesh).area);
16   qrEdge = QuadratureRule.ofOrder(1, '1D');
17   qrTri = QuadratureRule.ofOrder(1, '2D');
18   volumeRes = integrateElement(CompositeFunction(@(x) x.^2, lf.f), qrTri);
19   edgeRes = integrateNormalJump(Gradient(u), qrEdge, @(j) j.^2, {}, ':');
20   edgeRes(mesh.boundaries{:}) = 0;
21   eta2 = hT.^2.*volumeRes + hT.*sum(edgeRes(mesh.element2edges), Dim.Vector);
22
23   marked = markDoerflerSorting(eta2, 0.5);
24   mesh.refineLocally(marked, 'NVB');
25 end
```

Listing 1. Adaptive P1-FEM for Poisson problem $-\Delta u = 1$ on $\Omega = (0,1)^2$ subject to $u = 0$ on $\partial\Omega$.

Our software is intended to solve equation (2) by adaptive finite element methods (AFEM), an abstract form of which is presented in Algorithm 1 [15,16].

A realization of the abstract adaptive Algorithm 1 is shown in the code snippet in Listing 1 below. It computes the lowest-order FEM solution of the Poisson equation $-\Delta u = 1$ on the unit square $\Omega := (0,1)^2$ with homogeneous Dirichlet data $u = 0$ on the boundary $\Gamma_D := \partial\Omega$.

Algorithm 1 Abstract adaptive finite element method (AFEM)

Input: Initial triangulation $\mathcal{T}_0$ of Ω
Loop: For $\ell = 0, 1, \dots$ do

(i) Solve equation (2) to obtain u_ℓ
(ii) Estimate the error by computing refinement indicators $\eta_\ell(T)$ for all $T \in \mathcal{T}_\ell$
(iii) Mark elements $\mathcal{M}_\ell \subseteq \mathcal{T}_\ell$ based on η_ℓ
(iv) Refine marked elements to obtain $\mathcal{T}_{\ell+1} := \texttt{refine}(\mathcal{T}_\ell, \mathcal{M}_\ell)$

Output: Sequence of solutions u_ℓ

In Listing 1, lines 1–7 are setup code that initializes all necessary data structures. Lines 10–13 solve equation (2), i.e., they realize Algorithm 1(i). For Algorithm 1(ii), the error is then estimated in lines 15-21 by local contributions of the residual *a posteriori* error estimator [17]

$$\eta_\ell(T)^2 := |T|\,\|f\|^2_{L^2(T)} + |T|^{1/2}\|[\![\nabla u \cdot \boldsymbol{n}]\!]\|^2_{L^2(\partial T \cap \Omega)} \quad \text{for all } T \in \mathcal{T}_\ell.$$

In line 23, Algorithm 1(iii) is executed by the so-called *Dörfler* marking criterion [18]

$$\theta \sum_{T \in \mathcal{T}_\ell} \eta_\ell(T)^2 \le \sum_{T \in \mathcal{M}_\ell} \eta_\ell(T)^2 \quad \text{with } \theta = 0.5.$$

Finally, line 24 corresponds to Algorithm 1(iv) and uses *newest vertex bisection* (NVB) [19] to refine (at least) the marked elements.

In the following, we assume that there is a fixed initial triangulation $\mathcal{T}_0$ of Ω. All further meshes are supposed to be obtained by a finite number of successive refinement steps (with possibly multiple refinement strategies) from $\mathcal{T}_0$.

Remark 1. We note that all other coefficients from (1) can be set just as easily as in line 6–7 of Listing 1. The corresponding members of `blf` and `lf` are the following:

- `blf.a`, `blf.b`, `blf.c`, `lf.f`, and `lf.fvec` for the data of (1a);
- `blf.robin` and `lf.robin` for the data of (1b);
- `lf.neumann` for the data of (1c).

The types of functions that can be used are described in Section 3 below. There, also quadrature rules are presented, which can be assigned to the corresponding members `qra`, `qrb`, `qrc`, and `qrRobin` for `blf` as well as `qrf`, `qrfvec`, `qrRobin`, and `qrNeumann` for `lf`.

2.2. Necessity of OOP in MATLAB FEM

The use of OOP is not mandatory in MATLAB, but it facilitates code that is powerful yet concise and flexible. In particular, the code from Listing 1 relies heavily on OOP due to some peculiarities of the MATLAB programming language. Most notably, the referencing mechanics of MATLAB differ greatly from languages like C/C++, Java, or Python, where referencing variables is either done by default, or explicitly. We proceed by outlining the most important issues that we aim to address by OOP.

2.2.1. Re-using data

MATLAB has a lazy copy policy for function arguments: Arguments are generally passed by reference, but copied if they are modified within the function. However, no local copy is made of variables that are assigned to themselves by returning data:

```
1 function z = foo(x, y, z)
2  y(1) = x;
3  z = y + z;
4 end
5 z = foo(x, y, z)
```

In this example, `x` is passed by reference, `y` is copied because it is modified in line 2 (which is equivalent to passing the argument by value), and `z` is modified but not copied since the output value is again assigned to `z` in line 5.

FEM computations often re-use data throughout many sub-tasks; e.g., element areas are used in assembly of FEM systems, *a posteriori* error estimation, and even interpolation, via integration routines. With the passing mechanics outlined above, there are two options to approach this issue: First, data can be recomputed in each of the sub-tasks, yielding a clear public API, but causing superfluous operations. Second, the data can be precomputed explicitly and held in memory. In this case, the data management has to be done on the highest level of code by the user, or computations have to be grouped to respect data availability; both lead to a public API that is error-prone and inflexible. It is therefore highly desirable to have proper call by reference mechanics, which, in MATLAB, are only available through OOP.

2.2.2. Encapsulating data efficiently

The default for objects in MATLAB are *value classes*, which cannot change their state. The reason for this is the argument passing mechanism described above. In fact, except for pathological cases, the method invocation `obj.method(...)` and the function call `method(obj,...)` are equivalent. Hence, the copy-on-modify mechanics for function arguments also apply to instances of (value) classes. However, classes that are derived from the abstract `handle` class can overcome the limitations of value classes in the sense that they allow for modifications of state through methods:

```
1 classdef MyClass
2  %...
3  function obj = modify(obj, value)
4   obj.field = value;
5  end
6 end
7 obj = MyClass();
8 obj = modify(obj, 1);
```

```
1 classdef MyClass <  handle
2  %...
3  function modify(obj, value)
4   obj.field = value
5  end
6 end
7 obj = MyClass()
8 obj.modify(1)
```

Both code snippets result in `obj.field` being equal to one.

Also, handle classes can inherently be referenced: assigning an instance of a handle class to a variable does not copy the underlying data, but only assigns a reference. Finally, handle classes have native support for the *observer pattern*, which is one of the central design elements of our code; see Section 3.1.2 for details.

In order to communicate clearly, where methods can possibly alter the state of an object, we adhere to a coding best-practice called *command query separation* throughout documentation and examples: Commands, i.e., methods that alter the internal state of the calling object, are called with dot-notation `obj.method(...)` and never return data; queries, i.e., methods that do not alter the state of the calling object but may return data, are called with function call-notation `method(obj,...)`.

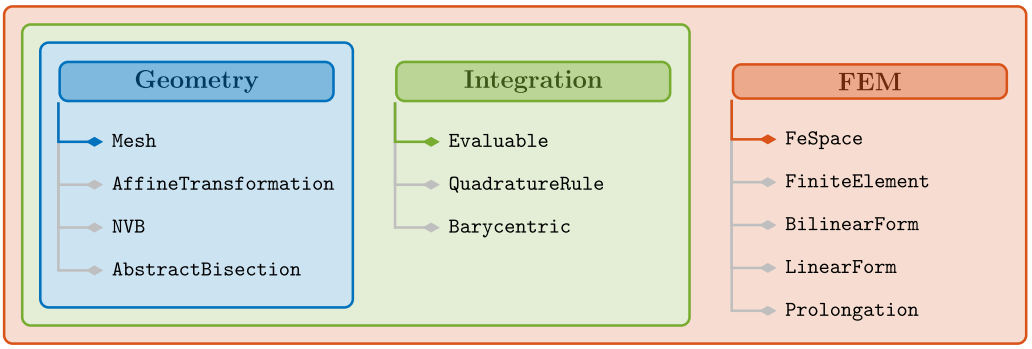


Fig. 1. Overall structure of the presented software package. Shown are all classes of the software package, subdivided into three modules. The most important class of each submodule is at the top of the respective list.

2.2.3. Code safety and error checks

One of the disadvantages of dynamically typed languages like MATLAB is the lack of automatic type checks and function overloading. By using classes to represent (even trivial) data, this behavior can be emulated to a certain degree.

Type checks can be automated by function argument validation, which was introduced recently in MATLAB. This is achieved by an optional `arguments`-block after the function head that performs some checks on all input arguments of that function. In particular, it can check class, dimension, and values of the input. This greatly improves usability and error mitigation.

Since method invocation `obj.method(...)` and function call `method(obj,...)` are virtually equivalent, function dispatch in MATLAB goes by the first argument of a function. This emulates function overloading at least in the first argument; e.g., in Listing 1 (solving the Poisson problem), one can readily use `plot(mesh)` and `plot(u)` to plot the mesh and the FEM solution. While this might be seen as syntactic sugar, it also greatly aids debugging.

2.2.4. Vectorization

One of the key features of MATLAB are efficiently vectorized built-in linear algebra operations. The usual local FEM formulation (i.e., on single elements and edges), however, does not allow for performance improvements through vectorization and parallelization, which are most pronounced if used with sufficiently large arrays to compensate for possible overhead. It is therefore desirable to defer actual computation as long as possible to make optimal use of MATLAB built-in routines.

Our code provides several well-defined interfaces very close to the natural (local) formulation of FEM which can be used to extend the functionality; see Section 3.3. The referencing mechanics of handle classes then allow the internal generation of global data from this local code by pre-existing routines and passing it to efficient built-in routines.

3. Code structure

Most multi-purpose FEM packages have a huge code-base (often combining several languages) and, necessarily, a cleverly designed class hierarchy that may require significant effort to understand and get it running [20–24]. Since one of our aims is that our code is easy to modify and extend, we strive for a relatively small code-base that nevertheless covers the most widespread demands of academic FEM software. Thus, the core of our software package is made up of only twelve classes and interfaces that can be roughly divided into three modules: Geometry, integration, and FEM. This partition is shown in Fig. 1.

What follows is a description of the separate core classes as well as their inter relationship to one another. Following the partition of the code, our presentation is divided into three parts.

3.1. Module geometry

The geometry module can be used entirely on its own. It handles mesh generation as well as local mesh refinement.

3.1.1. Mesh representation

As the underlying mesh $\mathcal{T}_H$ is the cornerstone in any adaptive FEM algorithm, the `Mesh` class is the central building block of MooAFEM. It consists of all data that is important for the digital representation of a 2D triangulation: coordinates, edges, elements, connectivity of edges and elements, edge orientation, and boundary information. The precise data structures for storing this information are described in Section 4 below.

Here, we focus on the role of the class in the code compound. Most other classes need a `Mesh` to function properly and, hence, store a reference to a suitable instance. Validity of data is strongly coupled to the underlying mesh: as soon as the mesh changes, derived data (e.g., geometric information as well as the data used in FEM computations) may be invalid. It is

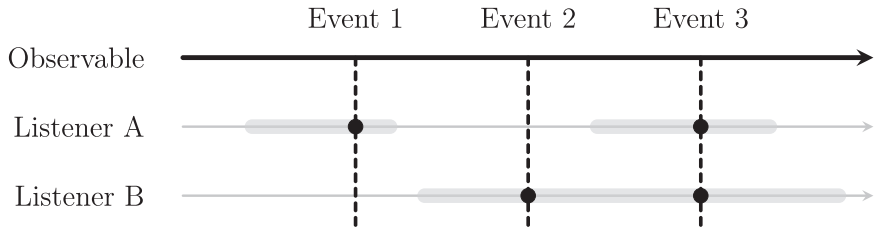


Fig. 2. Schematic functionality of the observer pattern. The timelines represent the lifetimes of the observable object and the listeners. Two listeners are temporarily registered to receive events (highlighted in gray). If the listeners are registered during the broadcast of an event (dashed lines), some internal reaction is triggered (bold dots). For each additional observable or event type of the same observable, a separate graph is needed.

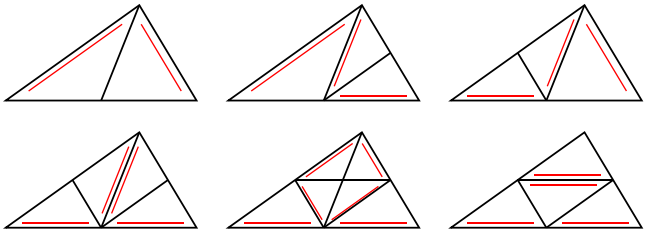


Fig. 3. Implemented bisection methods (top left to bottom right): `Bisec1`, `Bisec12`, `Bisec13`, `Bisec123`, `Bisec5`, and `BisecRed`. The refinement edge of the parent triangle is the bottom line, those of the children are highlighted by parallel lines.

therefore of vital importance that changes in the mesh are made public to every object that stores a reference to it, contrary to the usual flow of information in object oriented code. Also, objects that do not explicitly depend on a mesh may need to act if the mesh changes. A well-known remedy for this issue is presented in the next section.

3.1.2. Observer pattern

The observer pattern [25] is used to broadcast *events* from a central object (the *observable*) to other objects (the observers, termed *listeners* in MATLAB), which can react to the event in a predefined way and which can, at runtime, register and de-register to receive such events; see Fig. 2. The events signal, e.g., a change of state of the observable. All classes that derive from `handle` in MATLAB automatically implement the interfaces necessary to be the source of events. In particular, our `Mesh` class broadcasts events that signal a change in the mesh (e.g., by the call `refine`$(\mathcal{T}_\ell, \mathcal{M}_\ell)$ in Algorithm 1) as well as completed computation of bisection data to signal imminent refinement; this is covered in the next two sections.

3.1.3. Refinement

Mesh refinement is implementationally divided into bisection of single elements $T \in \mathcal{T}_H$ and coordination of bisections on the whole mesh $\mathcal{T}_H$ to obtain $\mathcal{T}_h = \texttt{refine}(\mathcal{T}_H, \mathcal{M}_H)$. In the class `AbstractBisection`, possible bisections of a single triangle $T \in \mathcal{T}_H$ are encoded. Subclasses of this class manage the generation of all `Mesh` data structures (see Section 4 below) in the passage from $T \in \mathcal{T}_H$ to its children $\{T' \in \mathcal{T}_h \,|\, T' \subseteq T\}$ in $\mathcal{T}_h$. The subclasses that are currently implemented are shown in Fig. 3 and build on the implementation of [3].

Local bisections are combined in mesh refinement routines derived from newest vertex bisection (NVB) [19,26], in the course of which all elements $T \in \mathcal{T}_H$ are assigned a subclass of `AbstractBisection`. We follow the iterative algorithm given in [26] for NVB, which terminates regardless of the mesh $\mathcal{T}_H$ under consideration. Denoting the set of edges in $\mathcal{T}_H$ by $\mathcal{E}_H$ and given a subset $\mathcal{M}_H \subseteq \mathcal{T}_H$ of marked elements, the abstract scheme is the following:

(i) Determine all edges in $\mathcal{E}_H$ that have to be bisected in order to bisect all elements in $\mathcal{M}_H$ by a given bisection rule.
(ii) Compute the *mesh closure*, i.e., determine all edges that *additionally* have to be refined to eliminate hanging vertices.
(iii) For every element in $\mathcal{T}_H$ determine which bisection method to employ, based on the marked edges.
(iv) Execute bisection of all elements according to their assigned bisection methods.

The first three steps are executed by subclasses of `NVB`, the fourth step is carried out by the mesh itself. The rationale behind this splitting is that, before the fourth step, all necessary information to carry out mesh refinement is already known; thus, this provides a natural hook for other classes to harness this information, e.g., prolongation operators or multi-grid solvers.

Several realizations of the abstract scheme presented above are implemented in our software package: `NVB1`, `NVB3` (= `NVB`), `NVB5`, and `RGB`, which are outlined in [3], as well as `NVBEdge`, which is an edge-driven refinement strategy used in [27,28].

3.1.4. Additional geometric information

For triangular meshes, all elements can be obtained by affine transformations of the so-called reference triangle $T_{\text{ref}} := \text{conv}\{(0,0),(1,0),(0,1)\}$, i.e., for all $T \in \mathcal{T}_H$, there exists an affine diffeomorphism $F_T : T_{\text{ref}} \to T$. Many computations

in FEM need additional geometric information based on this diffeomorphism. In particular, the transposed inverse $DF_T^{-\intercal}$ and the determinant $\det DF_T = 2|T|$ of its derivative are of utmost importance in integration and assembly routines. Together with the length $ds = |E|$ of each edge $E \in \mathcal{E}_H$ and its unit-normal vector $\boldsymbol{n}_E$, these data are stored in instances of `AffineTransformation`, which take a `Mesh` to construct.

For performance reasons, instances of `AffineTransformation` are requested from the mesh and are cached, i.e., computed at the first request, then stored as a reference within the mesh object and returned if further requests occur.

3.2. Module integration

Integration is a crucial part of FEM assembly and postprocessing (e.g., *a posteriori* error estimation). The module defines two classes that encapsulate data for numerical integration (also termed *quadrature*) routines and one to encapsulate function evaluation with a unified interface. This module can be used only in conjunction with the geometry module.

3.2.1. Barycentric coordinates

We denote all evaluation points and quadrature nodes in barycentric coordinates. On a triangle $T = \operatorname{conv}\{z^{(1)}, z^{(2)}, z^{(3)}\}$ and a point $x \in T$, we denote by $\lambda \in [0,1]^3$ the barycentric coordinates of x, determined by the equations

$$\sum_{i=1}^{3} \lambda_i = 1 \quad \text{and} \quad \sum_{i=1}^{3} \lambda_i z^{(i)} = x.$$

If the triangle T is non-degenerate, λ is unique. In MooAFEM, 2D barycentric coordinates are implemented in the class `Barycentric2D`, which is derived from the abstract class `Barycentric`. For convenience, this class stores a collection of barycentric coordinates.

The concept of barycentric coordinates can be generalized to d-simplices with $d \geq 1$ such that $\lambda \in [0,1]^{d+1}$. In particular, in the case $d = 1$ we have $\lambda \in [0,1]^2$, which is used for evaluation points and quadrature rules on edges and implemented in the class `Barycentric1D`. For any point $x \in T_{\text{ref}} = \operatorname{conv}\{(0,0),(1,0),(0,1)\}$ in the reference triangle, the barycentric coordinates are $\lambda = (1 - x_1 - x_2, x_1, x_2)$; for any point $x \in E_{\text{ref}} = [0,1]$ on the reference edge, they are $\lambda = (x, 1-x)$.

Denoting all function evaluation points in barycentric coordinates allows for triangle independent representation. This is reflected by `Evaluable.eval` below, which is the core of our vectorization efforts and paramount for the efficiency of MooAFEM.

3.2.2. Quadrature data

To approximate the integral of a (possibly vector-valued) function $f : T \to \mathbb{R}^n$ for some $n \in \mathbb{N}$ on a triangle $T \in \mathcal{T}_H$, we employ numerical quadrature:

$$\int_T f(x)\,dx \approx |T| \sum_{k=1}^{N} \omega_k f\big(x(\lambda^{(k)}, T)\big), \tag{3}$$

where $(\lambda^{(k)})_{k=1}^N$ is a collection of barycentric coordinates, $x(\lambda^{(k)}, T)$ is the Cartesian coordinate corresponding to the k-th barycentric coordinate on T, and $(\omega_k)_{k=1}^N$ are weights with $\sum_{k=1}^N \omega_k = 1$.

Barycentric coordinates and weights make up a `QuadratureRule` object. Quadrature rules can either be constructed explicitly by giving barycentric coordinates and weights, or by the static method

```
qr = QuadratureRule.ofOrder(order, [dim]);
```

The optional string argument `dim` is used to distinguish between `'1D'` and `'2D'` quadrature rules, the default being `'2D'`. For 1D, suitable Gauss-rules are implemented. For 2D, symmetric quadrature rules up to order 5 are implemented [29]. Higher order quadrature rules on triangles use tensorized Gauss-rules on $[0,1]^2$ and the Duffy transform $\Phi : [0,1]^2 \to T_{\text{ref}}$, $\Phi(s,t) = (s, t(1-s))$; [30].

3.2.3. Function evaluation

The core of the integration module is a wrapper for functions $f : \Omega \to \mathbb{R}^n$ for some $n \in \mathbb{N}$, the abstract `Evaluable` class:

```
classdef Evaluable < handle
properties (Abstract, GetAccess='public', SetAccess='protected')
  mesh
end
methods (Abstract, Access='public')
  eval(obj, bary, idx)
end
end
```

The abstract method `eval` is intended to evaluate the function at all points $x(\lambda^{(k)}, T)$ for all barycentric coordinates $\lambda^{(k)}$ in `bary` and all elements given by the index set `idx`.

By programming all routines (e.g., integration, plotting, finite element assembly) only to this interface, one can wrap virtually anything in a suitable subclass of `Evaluable` and readily use the predefined routines. The most important classes that implement this interface are:

- `Constant`: Efficiently wraps constant functions in the `Evaluable` interface.
- `MeshFunction`: General functions $f : \Omega \to \mathbb{R}^n$ for some $n \in \mathbb{N}$.
- `FeFunction`: FEM functions, e.g., $u \in \mathcal{S}^p(\mathcal{T}_H)$.
- `Gradient`, `Hessian`: Element-wise gradient ∇u and Hessian $\nabla^2 u$ for FEM functions.
- `CompositeFunction`: Arbitrarily combine any of the above; see the following explanation.

In particular, all of the above can be used as coefficients in (1).

Especially powerful is the subclass `CompositeFunction`, which uses the composite pattern [25]:

```
f = CompositeFunction(funcHandle, funcArgument1,..., funcArgumentN)
```

This class takes a function handle and one `Evaluable` for every argument of the function handle. E.g., xu^2 can be implemented by

```
f = CompositeFunction(@(x,u) x.*u.^2,...
  MeshFunction(mesh, @(x) x(1,:,:)), FeFunction(fes));
```

If `f` is evaluated, first all arguments are evaluated, then the resulting data is processed by the function handle. By polymorphism, the `Evaluable` arguments can be of any subclass. This allows to define complex functions that still can be evaluated efficiently, since evaluation of the function handle is deferred until the data for all requested elements and quadrature nodes is available. In this way, the vectorization capabilities of MATLAB are used to full extent.

3.2.4. Quadrature routines

There are several routines for quadrature implemented in our MooAFEM package:

- `integrateElement`: Integration on elements; cf. (3).
- `integrateEdge`: An analogue over edges. This can only be used for subclasses of `Evaluable` that implement the method `evalEdge`. Edge evaluation is not well-defined for some functions that are not continuous across edges (e.g., element-wise polynomials).
- `integrateJump`, `integrateNormalJump`: Integrate $[\![\cdot]\!]$ and $[\![(\cdot)\cdot \boldsymbol{n}]\!]$ over edges, respectively.

All quadrature routines take an `Evaluable`, a `QuadratureRule`, and an optional set of indices that corresponds to a subset of elements or edges on which the quadrature should be evaluated. The routines handling (normal) jumps take additional arguments: a function handle, a cell array of `Evaluable`, and edge indices.

```
int = integrateJump(f, qr, funcHandle, funcArg, idx)
```

This is used for post-processing the jump with the first argument of the function handle being reserved for the jump; i.e., the routine works roughly as follows: First, compute the jump by $\texttt{jump} = [\![\texttt{f}]\!]$ (or $\texttt{jump} = [\![\texttt{f}\cdot\boldsymbol{n}]\!]$). Then, all additional `Evaluables` are evaluated on the edges indicated by `idx`, where also the value of the jump is updated by the function handle:

```
val = {evalEdge(funcArg{1}, idx),..., evalEdge(funcArg{N}, idx)};
jump(idx) = functionHandle(jump(idx), val{1},..., val{N});
```

Multiple such triplets `funcHandle`, `funcArg`, `idx` for post-processing are allowed and sequentially applied as in the above listing, one after another.

3.3. Module FEM

This last module uses the classes presented in the last sections to conveniently represent FEM functions and efficiently assemble FEM data.

3.3.1. Finite element spaces

Finite elements are usually defined on the reference triangle T_{ref}, everything else follows from using the affine transformation F_T for every $T \in \mathcal{T}_H$. This viewpoint is represented accordingly in our code. The abstract class `FiniteElement` asks to implement evaluation of all basis functions (as well as their gradient and their Hessian) on T_{ref}. Furthermore, evaluation on a reference edge (if applicable) and the connectivity of the degrees of freedom (DOFs), i.e., how the finite element couples across edges and vertices, have to be specified.

The class `FeSpace` combines the local information of finite elements with the global geometry of the mesh. It takes a `Mesh` and a `FiniteElement` to assemble lists of global DOFs per element and edge, as well as free DOFs, i.e., DOFs that do not lie on Γ_D.

So far, Lagrange finite elements of arbitrary order are implemented; both $H^1(\Omega)$-conforming (i.e., $\mathcal{S}^p(\mathcal{T}_H) = \mathcal{P}^p(\mathcal{T}_H) \cap H^1(\Omega)$) and $L^2(\Omega)$-conforming (i.e., $\mathcal{P}^p(\mathcal{T}_H)$). The underlying implementation uses Bernstein–Bézier polynomials [31]. For lowest-order finite elements (both continuous and discontinuous), additional optimized implementations are available.

3.3.2. FEM system assembly

Let $(\varphi_k)_{k=1}^N$ be a basis of $\mathcal{S}^p(\mathcal{T}_H)$. Responsible for the assembly of the data

$$A_{ij} := \int_\Omega \mathbf{A}\nabla\varphi_j \cdot \nabla\varphi_i + \boldsymbol{b} \cdot \nabla\varphi_j\varphi_i + c\,\varphi_j\varphi_i \,\mathrm{d}x + \int_{\Gamma_R} \alpha\,\varphi_j\varphi_i \,\mathrm{d}s, \tag{4a}$$

$$F_i := \int_\Omega f\varphi_i + \boldsymbol{f} \cdot \nabla\varphi_i \,\mathrm{d}x + \int_{\Gamma_N} \phi\,\varphi_i \,\mathrm{d}s + \int_{\Gamma_R} \gamma\,\varphi_i \,\mathrm{d}s \tag{4b}$$

are the classes `BilinearForm` and `LinearForm`, respectively. Both classes have fields for their respective coefficients and quadrature rules for each of the terms in (4); see Remark 1. The data in (4) are then obtained by calling the `assemble` methods of both classes, using a generalization of the efficiently vectorized method outlined in [3].

Note that `Evaluable` is a handle class. Thus, no data must be copied to set (bi-)linear form coefficients. Furthermore, in situations where the coefficients change frequently, e.g., in the presence of iterative solvers for nonlinear PDEs, the coefficients of the (bi-)linear form can change between two consecutive calls of `assemble`. This results in much cleaner code since one does not need to re-set the coefficients.

3.3.3. Prolongation

It is often necessary to prolongate FEM functions $u_H \in \mathcal{S}^p(\mathcal{T}_H)$ on a mesh $\mathcal{T}_H$ to the richer space $\mathcal{S}^p(\mathcal{T}_h)$ with a refined mesh $\mathcal{T}_h$. This is handled by subclasses of the abstract class `Prolongation`. The implemented prolongation methods are `LoFeProlongation` and `FeProlongation` for lowest-order and general (in particular, higher-order) FEM functions, respectively. Note that the latter is not tailored to a specific finite element and, hence, its computational effort is slightly higher than that of the former. The syntax of prolongation of a function `u = FeFunction(fes)` on a finite element space `fes` from a coarse to a refined mesh is as follows:

```
1 P = FeProlongation(fes);
2 mesh.refineLocally(marked);
3 u.setData(P.prolongate(u));
```

The call to `P.prolongate` performs a matrix-vector multiplication with the (sparse) prolongation matrix `P.matrix`, which is set automatically by element-wise Lagrange interpolation whenever the mesh is refined, due to the events sent by the mesh; see Section 3.1.2 and the examples in Section 5.

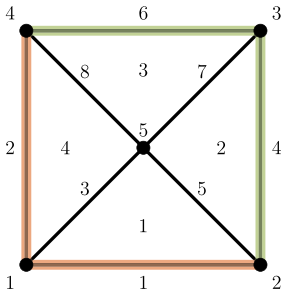
4. Data structures

4.1. Mesh

The `Mesh` class stores information about coordinates, edges, and elements. In the following, let $n_\mathcal{V}, n_\mathcal{E}, n_\mathcal{T} \in \mathbb{N}$ denote the number of vertices, edges, and elements, respectively. The class has the following fields:

- `coordinates` ($2 \times n_\mathcal{V}$): Coordinates of mesh vertices. The entries `coordinates(1,i)` and `coordinates(2,i)` store the x- and y-coordinates of the i-th vertex, respectively. The order of the coordinates is provided by the user.
- `edges` ($2 \times n_\mathcal{E}$): Indices of vertices of all edges in the mesh. The i-th edge of the mesh starts at vertex `edges(1,i)` and ends at vertex `edges(2,i)`. The order is determined automatically from information provided in `elements`. Boundary edges are oriented such that the domain lies on its left; inner edges cannot be assigned a meaningful orientation and, therefore, it is chosen randomly.
- `elements` ($3 \times n_\mathcal{T}$): Indices of vertices of which elements are comprised. The i-th element is spanned by the vertices with indices `elements(:,i)`, where the order is counter-clockwise. The order of elements is provided by the user.
- `element2edges` ($3 \times n_\mathcal{T}$): Indices of edges which are contained in elements. The i-th element contains edges with indices `element2edges(:,i)`. The j-th edge `element2edges(j,i)` of the i-th element is the one between the vertices with indices `elements(j,i)` and `elements(mod(j+1,3)+1, i)` (but not necessarily in that order). This information is determined automatically.
- `boundaries` (cell array): Indices of all edges that form a specific part of the boundary. The cell `boundariesi` is a vector of edge indices that form the i-th boundary (if present). The boundary parts are provided by the user (see below), but the association with edge indices is done automatically.

The orientation of the normal vector from `AffineTransformation` follows the orientation of the edges: it points to the right of the edge. In particular, this means that the normal vectors on boundary edges point out of the domain.



n	1	2	3	4	5	6	7	8
coordinates	0.0	1.0	1.0	0.0	0.5			
	0.0	0.0	1.0	1.0	0.5			
edges	1	4	1	2	2	3	3	4
	2	1	5	3	5	4	5	5
elements	1	2	3	4				
	2	3	4	1				
	5	5	5	5				
element2edges	1	4	6	2				
	5	7	8	3				
	3	5	7	8				

n	1	2
bndEdges{1}	1	4
	2	1
bndEdges{2}	2	3
	3	4
boundaries{1}	1	2
boundaries{2}	4	6

Fig. 4. Example mesh on the unit square $(0,1)^2 \subset \mathbb{R}^2$ as well as corresponding data structures. Boundary part 1 (e.g., Γ_D) is marked in red, boundary part 2 (e.g., Γ_N) is marked in green.

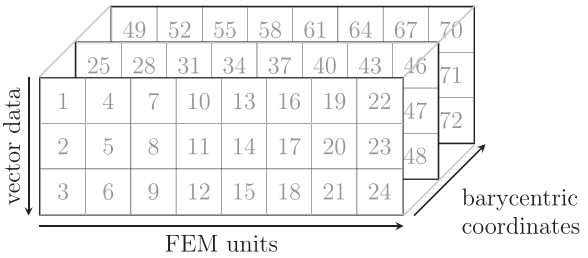


Fig. 5. Illustration of the memory layout chosen in our implementation. The numbers indicate the order in which the items are stored in memory.

4.2. Mesh construction

An instance of the `Mesh` class can be constructed by

```
1 mesh = Mesh(coordinates, elements, bndEdges);
```

Here, `coordinates` and `elements` have to be given as they are stored in the class (see above). The cell array `bndEdges` describes the boundary parts, where the i-th edge of the k-th boundary part lies between the vertices with indices `bndEdgesk(1,i)` and `bndEdgesk(2,i)`. This is necessary because the user does not know the internal edge numbering before construction. Special attention has to be payed to the correct orientation of the elements (counter-clockwise) and the edges on the boundary (domain on their left), because this is not checked by the constructor.

The arrays required by the constructor can be assembled and passed from a MATLAB script, or loaded from comma separated value files by the static method

```
1 mesh = Mesh.loadFromGeometry('<name>');
```

These files must be placed in a subdirectory `lib/mesh/@Mesh/geometries/<name>` and be named `coordinates.dat`, `elements.dat`, and `boundary<n>.dat`, where boundary parts are enumerated by $n \in \mathbb{N}$.

Mesh construction and data structures are showcased in Fig. 4. Note that the orientation of the user-provided edges in `bndEdges` is preserved by the automatically generated field `edges`.

4.3. Array layout

The array layout is chosen such that the first three dimensions of arrays correspond to modeling concepts of finite elements; see Fig. 5:

- **1. dimension (columns)**: This corresponds to the components of vector- or matrix-valued data. Matrices are stored column-wise.
- **2. dimension (rows)**: This corresponds to the different units of the mesh, i.e., elements, edges, or vertices.
- **3. dimension (pages)**: This corresponds to different barycentric coordinates, e.g., for numerical quadrature.

The rationale behind this order is twofold. First, objects on the same element often need to be multiplied together. Hence, it is of advantage if the data representing these objects are continuous in memory. This is achieved by arranging them along the columns of the array, since MATLAB stores arrays in column-major order. Second, arrays that extend into the third dimension are somewhat clumsy to work with and hard to debug for programmers. Since evaluations on multiple barycentric coordinates occur mostly internally (e.g., for quadrature rule or plotting), arranging different barycentric coordinates along

the third dimension minimizes exposure of the user to more-than-two dimensional arrays. Within MooAFEM, one can use the enumeration class `Dim` to access these dimensions by `Dim.Vector`, `Dim.Elements`, and `Dim.QuadratureNodes`, respectively.

As an illustrative example, consider the call

```
1 f = MeshFunction(mesh, @(x) x);
2 val = eval(f, bary);
```

which evaluates $f : \Omega \to \mathbb{R}^2 : x \mapsto x$ on a collection `bary` of barycentric coordinates and a given mesh element-wise. The value stored in `val(i,j,k)` corresponds to $x_i(\lambda^{(k)}, T_j)$, i.e., the i-th component of `f` evaluated at the k-th barycentric coordinate on the j-th element. The matrix valued function

$$f : \Omega \to \mathbb{R}^{2\times 2} : x \mapsto \begin{pmatrix} 1x_1 & 3x_1 \\ 2x_1 & 4x_1 \end{pmatrix}$$

can be implemented by `f = MeshFunction(mesh, @(x) [1;2;3;4].*x(1,:,:))`. The corresponding evaluation `val(i,j,k)` is equal to $i \cdot x_1(\lambda^{(k)}, T_j)$, since matrices are stored column-wise. See also Fig. 5 for a sketch of this memory layout.

4.4. Efficient linear algebra

According to the last section, 3D arrays are essentially interpreted as collections of matrices stored column-wise. To efficiently execute matrix operations within this memory layout, the function

```
1 C = vectorProduct(A, B, sizeA, sizeB);
```

is used. It computes the product of two 3D arrays `A` and `B`, where the first dimension is interpreted as matrix with given size `sizeA` and `sizeB`, respectively. In particular, for all admissible $i, j \in \mathbb{N}$, the output of above call satisfies

```
1 C(:,i,j) = reshape(A(:,i,j), sizeA) * reshape(B(:,i,j), sizeB);
```

If either `sizeA` or `sizeB` is a column-vector, the corresponding factor in the above listing is transposed. For an example, consider the vector $\mathtt{v} := (1, 2, 3, 4, 5, 6)^\intercal$, which, in our memory model, can be interpreted as

$$\texttt{[2,3]}: \quad A := \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix} \quad \text{or} \quad \texttt{[3,2]}: \quad B := \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}.$$

Clearly there holds $A^\intercal \neq B$. Therefore, transposing the size is necessary to indicate transposition of the corresponding factor in the matrix product:

$$\begin{aligned} AB &= \texttt{vectorProduct(v, v, [2,3], [3,2])}, \\ \mathtt{A}\mathtt{A}^\intercal &= \texttt{vectorProduct(v, v, [2,3], [2,3]')}, \\ B^\intercal B &= \texttt{vectorProduct(v, v, [3,2]', [3,2])}. \end{aligned}$$

The function `vectorProduct` thus enables all possible matrix operations within our memory layout in a convenient, yet very efficient way. In fact, this routine can also deal with arrays of arbitrary dimension, where extension of singleton dimensions is done automatically, as is common in MATLAB. Per default, the sizes are chosen such that the dot product

```
1 C(:,i,j) = A(:,i,j)' * B(:,i,j);
```

is computed, which is the most common application; this is also reflected in the name.

5. Examples

In this section, we discuss several extensions of the basic AFEM algorithm that is implemented in Listing 1. We do not claim that MooAFEM can deal with all FEM applications out of the box, but are convinced that our code structure makes extensions and modifications relatively easy.

5.1. Higher order AFEM with known solution

As a first example we consider the L-shape $\Omega := (-1, 1)^2 \setminus ([0, 1] \times [-1, 0])$ with boundary parts $\Gamma_R = \emptyset$, $\Gamma_N := ([0, 1] \times \{0\}) \cup (\{0\} \times [-1, 0])$, and $\Gamma_D := \partial\Omega \setminus \Gamma_N$. With $(r(x), \varphi(x))$ being the polar coordinates of $x \in \mathbb{R}^2$, we prescribe the exact solution

$$u(x) := r(x)^{2/3} \sin(2\pi/3)$$

and note that it solves

$$-\Delta u = 0 \ \text{ in } \Omega, \quad \nabla u \cdot \boldsymbol{n} =: \phi \ \text{ on } \Gamma_N, \quad u = 0 \ \text{ on } \Gamma_D. \tag{5}$$

To solve (5) numerically with MooAFEM, only minor adjustments of the code from Listing 1 are necessary. First, obviously, the correct mesh must be loaded via

```
1 mesh = Mesh.loadFromGeometry('Lshape');
```

This geometry has two predefined boundary parts: The first (boundary index 1) is at the re-entrant corner (Γ_D), the second (boundary index 2) is everything else (Γ_N). Next, the finite element space has to be chosen accordingly with some $p \in \mathbb{N}$:

```
1 fes = FeSpace(mesh, HigherOrderH1Fe(p), 'dirichlet', 1);
```

This loads an implementation of $\mathcal{S}^p_D(\mathcal{T}_0)$. No further adjustments regarding the implementation of higher-order finite elements are necessary.

The next changes concern the coefficients of the linear form `lf`. Our problem (5) includes only the Neumann part of the right-hand side from (2), which can be implemented by the following listing.

```
 1 lf.neumann = MeshFunction(mesh, @exactSolutionNeumannData);
 2 lf.bndNeumann = 2;
 3 function y = exactSolutionNeumannData(x)
 4  x1 = x(1,:,:);
 5  x2 = x(2,:,:);
 6  % determine boundary parts
 7  right = (x1 >  0) & (abs(x1) >  abs(x2));
 8  left = (x1 <  0) & (abs(x1) > abs(x2));
 9  top = (x2 >  0) & (abs(x1) <  abs(x2));
10  bottom = (x2 <  0) & (abs(x1) <  abs(x2));
11  % compute d/dn u
12  [phi, r] = cart2pol(x(1,:,:), x(2,:,:));
13  Cr = 2/3 * r.^(-4/3);
14  Cphi = 2/3 * (phi + 2*pi*(phi <  0));
15  dudx = Cr.* (x1.*sin(Cphi) - x2.*cos(Cphi));
16  dudy = Cr.* (x2.*sin(Cphi) + x1.*cos(Cphi));
17  y = zeros(size(x1));
18  y(right) = dudx(right);
19  y(left) =-dudx(left);
20  y(top) = dudy(top);
21  y(bottom) =-dudy(bottom);
22 end
```

Here, the main part is the implementation of the Neumann derivative $\nabla u \cdot \boldsymbol{n}$, rather than making the function available to the assembly routines. Finally, we need to set quadrature rules of sufficiently high order for the corresponding terms of the (bi-)linear form:

```
1 blf.qra = QuadratureRule.ofOrder(max(2*p-2, 1));
2 lf.qrNeumann = QuadratureRule.ofOrder(2*p, '1D');
```

With these preparatory steps, the FEM system is solved by lines 10–13 of Listing 1.

Finally, the *a posteriori* indicators have to be adjusted to the current setting:

$$\eta_H(T)^2 = h_T^2 \|\Delta u_H\|^2_{L^2(T)} + h_T \big[\|[\![\nabla u_H \cdot \boldsymbol{n}]\!]\|^2_{L^2(\partial T \cap \Omega)} + \|(\nabla u_H - \nabla u) \cdot \boldsymbol{n}\|^2_{L^2(\partial T \cap \Gamma_N)} \big]. \tag{6}$$

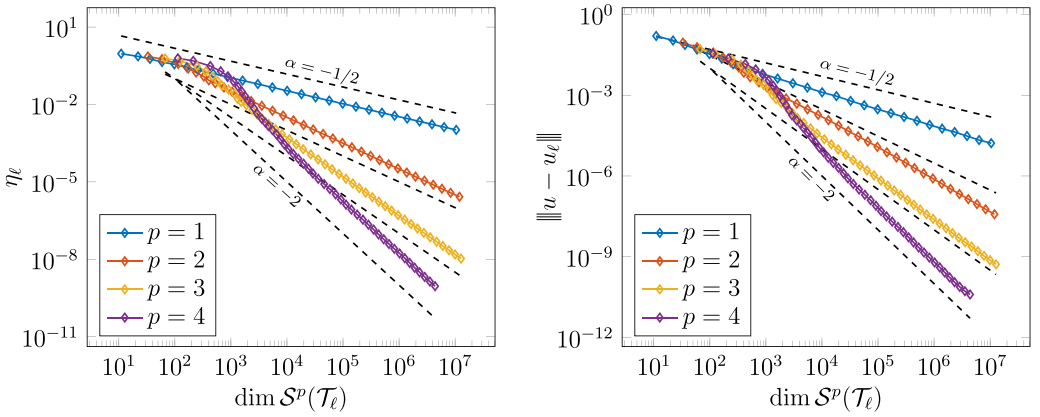


Fig. 6. Error estimator η_ℓ (left) and energy error $|||u - u_\ell|||$ (right) over number of DOFs for problem (5) with different polynomial orders p.

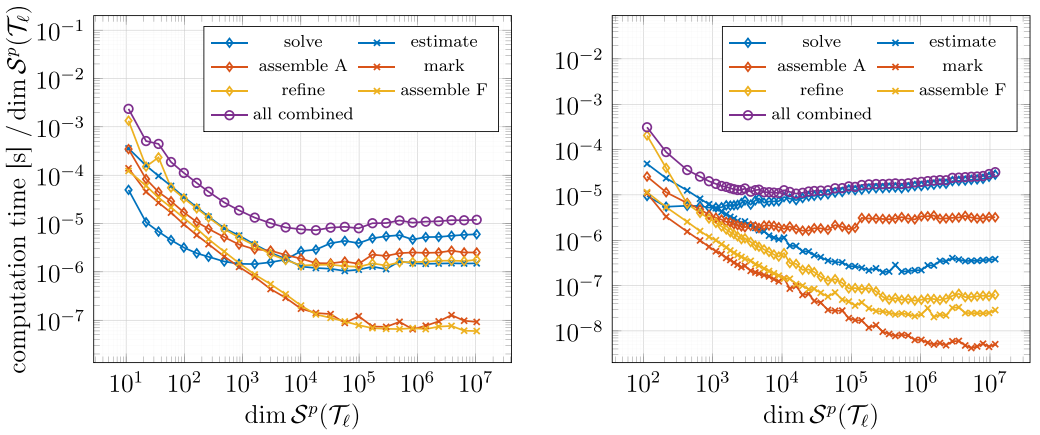


Fig. 7. Computation time per DOF over number of DOFs for the different parts of the AFEM algorithm for problem (5) with polynomial degree $p = 1$ (left) and $p = 4$ (right).

Recall from Section 4.3 that matrices are stored column-wise in the first dimension of 3D arrays. Thus, the L^2-norm of the volume term can be computed by

```
f = CompositeFunction(@(D2u) (D2u(1,:,:) + D2u(4,:,:)).^2, Hessian(u));
qr = QuadratureRule.ofOrder(max(2*(p-2), 1));
volumeRes = integrateElement(f, qr);
```

The edgewise L^2-norms are a bit more involved. This is handled by

```
qr = QuadratureRule.ofOrder(p, '1D');
edgeRes = integrateNormalJump(Gradient(u), qr,...
 @(j) zeros(size(j)), {}, mesh.boundaries{1},...
 @(j,phi) j-phi, {lf.neumann}, mesh.boundaries{2},...
 @(j) j.^2, {}, ':');
```

The syntax of `integrateNormalJump` is explained in Section 3.2. First, the jump $[\![\nabla u_H \cdot \boldsymbol{n}]\!]$ is computed on every edge. Then, since the edges on Γ_D (boundary index 1) do not contribute to the error estimator, the corresponding contributions are set to zero. Furthermore, the term $\nabla u \cdot \boldsymbol{n}$, which is stored in `lf.neumann`, is subtracted on Γ_N (boundary index 2). Finally, every edge contribution is squared to obtain the edgewise L^2-norms of (6).

The remainder of the code is analogous to the one presented in Listing 1. Note that virtually all changes are due to the different model problem and not for implementational reasons. Fig. 6 shows the results obtained for $p = 1, 2, 3, 4$. Note that, from an implementational point of view, the polynomial degree $p \in \mathbb{N}$ can be chosen arbitrarily high. Computation times for the different parts of the adaptive algorithm are shown in Fig. 7. In both the lowest and the higher order case, most time is spent for solution of the linear system. In the higher order case, one clearly sees that solving with the MATLAB backslash operator has more than linear complexity.

As a final note, the exact error of the finite element solution u_H can be easily computed by the following code snippet. Recall that `A` is the finite element matrix of the Laplacian.

```
uex = FeFunction(fes);
uex.setData(nodalInterpolation(MeshFunction(mesh, @exactSolution), fes));
deltaU = u.data - uex.data;
H1Error = sqrt(deltaU * A * deltaU');

function y = exactSolution(x)
 [phi, r] = cart2pol(x(1,:,:), x(2,:,:));
 phi = phi + 2*pi*(phi <  0);
 y = r.^(2/3).* sin(2/3 * phi);
end
```

5.2. Goal-oriented AFEM with discontinuous data

With $\Omega := (0,1)^2$ and $\Gamma_D := \partial\Omega$, we consider an example from [32]:

$$-\Delta u = -\operatorname{div}\boldsymbol{f} \ \text{ in } \Omega, \quad u = 0 \ \text{ on } \Gamma_D, \quad \text{where } \boldsymbol{f}(x) := \begin{cases} (1,0) & \text{if } x_1 + x_2 < 1/2, \\ (0,0) & \text{else.} \end{cases} \tag{7}$$

For most FEM software, discontinuous coefficients or data demand some caution: for quadrature nodes that lie on the discontinuity, evaluation is not well-defined. A first solution is to make the initial triangulation $\mathcal{T}_0$ of Ω resolve the regions of discontinuity. In our case, this can be achieved by uniform refinement using the RGB-strategy:

```
mesh = Mesh.loadFromGeometry('unitsquare');
mesh.refineUniform(1, 'RGB');
```

This is also needed for residual error estimators, since they are comprised of element-wise L^2-norms of $\operatorname{div}\boldsymbol{f}$ (which vanishes if the discontinuity is resolved by the mesh and is not defined otherwise).

A second problem is the jump term $[\![\cdot]\!]$ in the error estimators, since this is evaluated on edges, where the discontinuity now lies. This can be solved by interpolating the data to a non-continuous FEM space. To obtain vector-valued data, we first interpolate the non-continuous first component and then compose this with the vanishing second component, according to the memory layout presented in Section 4.3:

```
ncFes = FeSpace(mesh, LowestOrderL2Fe);
w = FeFunction(ncFes);
chiT = MeshFunction(mesh, @(x) sum(x, Dim.Vector) <  1/2);
w.setData(nodalInterpolation(chiT, ncFes));
lfF = LinearForm(fes);
lfF.fvec = CompositeFunction(@(w) [w;zeros(size(w))], w);
```

The nodal interpolation in the listing above only sets the data for `w` on the initial mesh $\mathcal{T}_0$. To have `w` available on refined meshes, we can repeat this interpolation process after every mesh refinement. A more efficient method is to use the prolongation class `P = LoFeProlongation(fes)` that is tailored specifically to lowest order L^2- and H^1-elements; see Section 3.3. Data for prolongation is computed automatically whenever the mesh is refined; see Section 3.1. After updating `w` by `w.setData(P.prolongate(w))`, the next call of `assemble(lfF)` already yields the updated right-hand side, since the coefficient `lfF.fvec` stores a reference to `w`.

In *goal-oriented* adaptive FEM (GOAFEM), we are interested in the *goal value* $G(u)$ for a linear functional

$$G : H^1_D(\Omega) \to \mathbb{R}, \quad G(v) = \int_\Omega \boldsymbol{g} \cdot \nabla v \, dx \quad \text{with} \quad \boldsymbol{g}(x) := \begin{cases} (-1,0) & \text{if } x_1 + x_2 > 3/2, \\ (0,0) & \text{else.} \end{cases}$$

Approximating the goal value $G(u)$ is often more interesting in applications than approximating the solution u as a whole. To efficiently approximate the goal value in the spirit of Algorithm 1, one introduces the so-called *dual* problem

$$-\Delta z = -\operatorname{div}\boldsymbol{g} \ \text{ in } \Omega, \quad z = 0 \ \text{ on } \Gamma_D, \tag{8}$$

which can be implemented and solved analogously to (7). Since solving is often the most time consuming part of AFEM, we can do this in parallel for (7) and (8):

```
rhs = [assemble(lfF), assemble(lfG)];
uz = A(freeDofs,freeDofs) \ rhs(freeDofs,:);
u.setFreeData(uz(:,1));
z.setFreeData(uz(:,2));
```

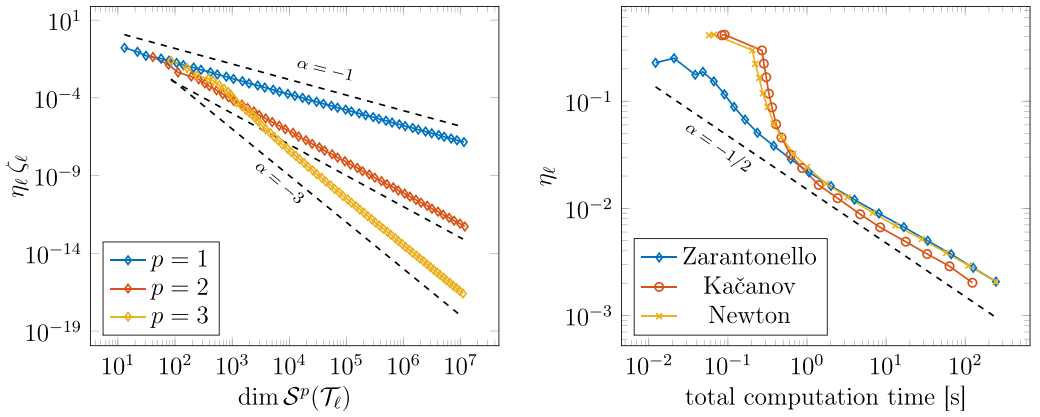


Fig. 8. Left: Estimator for the goal error (9) over number of DOFs for problem (7) from Section 5.2 with different polynomial orders p. Right: Error estimator over total computation time for the linearization methods from Section 5.3.

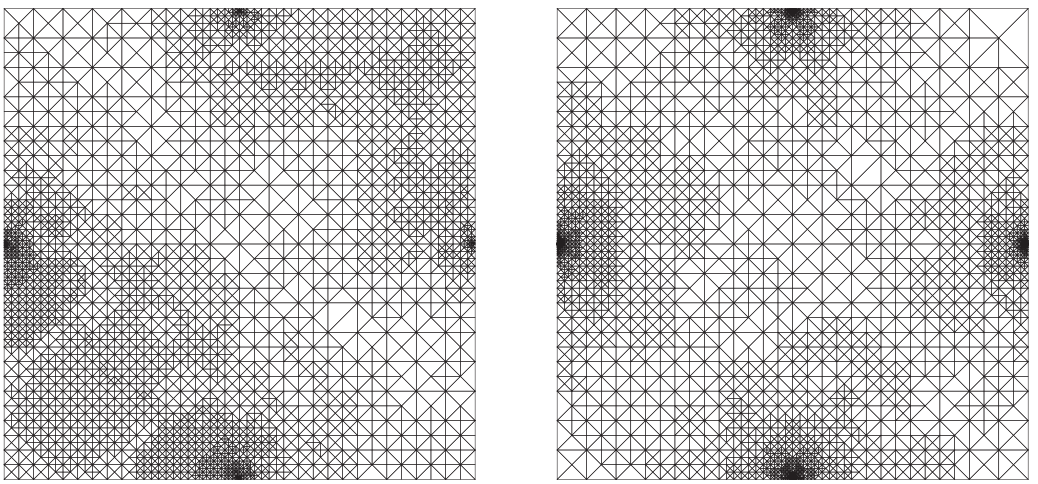


Fig. 9. Meshes generated from the GOAFEM algorithm from Section 5.2 with polynomial orders $p = 1$ (left) and $p = 3$ (right).

After solving (7) and (8) by FEM on a triangulation $\mathcal{T}_H$ to obtain the discrete solutions u_H and z_H, respectively, one can compute the *a posteriori* residual error estimators

$$\eta_H(T)^2 = h_T^2 \|\Delta u_H\|_{L^2(T)}^2 + h_T \|[\![(\nabla u_H - \boldsymbol{f}) \cdot \boldsymbol{n}]\!]\|_{L^2(\partial T \cap \Omega)}^2,$$
$$\zeta_H(T)^2 = h_T^2 \|\Delta z_H\|_{L^2(T)}^2 + h_T \|[\![(\nabla z_H - \boldsymbol{g}) \cdot \boldsymbol{n}]\!]\|_{L^2(\partial T \cap \Omega)}^2$$

analogously to (6). The error in the goal functional is controlled by the estimator product

$$|G(u) - G(u_H)| \lesssim \left[\sum_{T \in \mathcal{T}_H} \eta_H(T)^2\right]^{1/2} \left[\sum_{T \in \mathcal{T}_H} \zeta_H(T)^2\right]^{1/2}, \tag{9}$$

for which different marking criteria have been analyzed [33]. Thus, the remaining implementation comprises only minor modifications of Listing 1. The upper bounds of this last equation for different polynomial orders p can be seen in Fig. 8 and the resulting meshes for $p = 1, 3$ are shown in Fig. 9.

5.3. Iterative solution of nonlinear equations

In this last example, we consider the L-shape $\Omega := (-1, 1)^2 \setminus ([0, 1] \times [-1, 0])$ with Dirichlet boundary $\Gamma_D := \partial\Omega$. On this domain, we consider the quasi-linear problem

$$-\operatorname{div}\big(\mu\big(|\nabla u|^2\big)\nabla u\big) = 1 \quad \text{in } \Omega, \quad u = 0 \quad \text{on } \Gamma_D, \quad \text{with} \quad \mu(t) = 1 + \exp(-t). \tag{10}$$

This is a variation of an example given in [34], where also adaptive iterative linearization techniques (AILFEM) for this class of problems are presented. With a given initial guess $u^0 \in H^1_D(\Omega)$, we consider the following linearizations:

(i) **Zarantonello iteration**: Let $\delta > 0$ be sufficiently small. Given $u^n \in H^1_D(\Omega)$, the next iterate $u^{n+1} \in H^1_D(\Omega)$ reads $u^{n+1} := u^n + \delta v$, where $v \in H^1_D(\Omega)$ solves

$$-\Delta v = \operatorname{div}\big(\mu\big(|\nabla u^n|^2\big)\nabla u^n\big) + 1.$$

(ii) **Kačanov iteration**: Given $u^n \in H^1_D(\Omega)$, the next iterate $u^{n+1} \in H^1_D(\Omega)$ solves

$$-\operatorname{div}\big(\mu\big(|\nabla u^n|^2\big)\nabla u^{n+1}\big) = 1.$$

(iii) **Newton iteration**: Given $u^n \in H^1_D(\Omega)$, the next iterate $u^{n+1} \in H^1_D(\Omega)$ reads $u^{n+1} := u^n + v$, where $v \in H^1_D(\Omega)$ solves

$$-\operatorname{div}\big(\mu\big(|\nabla u^n|^2\big)\nabla v + 2\mu'\big(|\nabla u^n|^2\big)(\nabla u^n \otimes \nabla u^n)\nabla v\big) = \operatorname{div}\big(\mu\big(|\nabla u^n|^2\big)\nabla u^n\big) + 1.$$

All iterations (i)–(iii) feature coefficients that depend in a nonlinear fashion on the previous iterate u^n. However, their implementation is relatively simple, owing to the uniform evaluation mechanics of the `Evaluable` interface, from which also `FeFunction` is derived. Assuming that, for some triangulation $\mathcal{T}_H$ of Ω, the previous iterates u^n_H correspond to the `FeFunction` instance `u`, the following code snippet acts as template for all three iterations with $u^0_H = 0$:

```
% set coefficients of blf & lf
u = FeFunction(fes);
u.setData(0);
v = FeFunction(fes);
freeDofs = getFreeDofs(fes);
while true
 A = assemble(blf);
 F = assemble(lf);
 % solve linear systems and update data of u
end
```

The two steps in this template that are merely outlined in a comment differ for each method. They are described in the following listing, separated by comments:

```
% --- Zarantonello : setup
blf.a = Constant(mesh, 1);
lf.f = Constant(mesh, 1);
lf.fvec = CompositeFunction(@(p) -mu(vectorProduct(p, p)).* p, Gradient(u));
% --- Zarantonello : update
v.setFreeData(A(freeDofs,freeDofs) \ F(freeDofs));
u.setData(u.data + delta*v.data);
% --- Kacanov : setup
blf.a = CompositeFunction(@(p) mu(vectorProduct(p, p)), Gradient(u));
lf.f = Constant(mesh, 1);
% --- Kacanov : update
u.setFreeData(A(freeDofs,freeDofs) \ F(freeDofs));
% --- Newton : setup
blf.a = CompositeFunction(@(p) mu(vectorProduct(p, p)).* [1;0;0;1]...
 + 2*muPrime(vectorProduct(p, p)).*vectorProduct(p, p, [2,1], [2,1]'),...
 Gradient(u));
lf.f = Constant(mesh, 1);
lf.fvec = CompositeFunction(@(p) -mu(vectorProduct(p, p)).* p, Gradient(u));
% --- Newton : update
v.setFreeData(A(freeDofs,freeDofs) \ F(freeDofs));
u.setData(u.data + v.data);
% --- Additional functions
mu = @(t) 1 + exp(-t);
muPrime = @(t) -exp(-t);
```

As explained in Section 4.4, with `p = Gradient(u)`, the two calls of `vectorProduct` in the Newton bilinear form represent

$$|\nabla u^n_H|^2 = \texttt{vectorProduct(p, p)},$$
$$\nabla u^n_H \otimes \nabla u^n_H = \texttt{vectorProduct(p, p, [2,1], [2,1]')}.$$

To get an adaptive algorithm in the spirit of Algorithm 1 for lowest order FEM, i.e., $p = 1$, error estimation is done by

$$\eta_H(T)^2 := h_T^2 \|1\|^2_{L^2(T)} + h_T \|[\![\mu(|\nabla u^n_H|^2)\nabla u^n_H \cdot \boldsymbol{n}]\!]\|^2_{L^2(\partial T \cap \Omega)},$$

which is analogous to (6). Finally, we remark that [34] suggests to use $u^0_0 = 0 \in \mathcal{S}^1_D(\mathcal{T}_0)$ only on the coarsest level and then to proceed by nested iteration

$$u^0_{\ell+1} := u^{n(\ell)}_\ell \in \mathcal{S}^1_D(\mathcal{T}_{\ell+1}) \quad \text{for all } \ell \in \mathbb{N},$$

where $\underline{n}(\ell)$ is the last iteration on the previous level $\mathcal{T}_\ell$, i.e., $u_\ell^{\underline{n}(\ell)}$ is the final iterate on $\mathcal{T}_\ell$. For lowest-order $H^1_D(\Omega)$-conforming FEM, this can be done by the prolongation class `LoFeProlongation`; see Section 3.3.

A numerical comparison of the three presented iterative linearization methods can be seen in Fig. 8.

Data Availability

The code is publicly available through the provided link in the references.

References

[1] J. Li, Y.-T. Chen, Computational Partial Differential Equations Using MATLAB®, CRC Press, 2019, doi:10.1201/9780429266027.
[2] L. Chen, iFEM: an integrated finite element methods package in MATLAB, Technical Report, 2009. https://github.com/lyc102/ifem.
[3] S. Funken, D. Praetorius, P. Wissgott, Efficient implementation of adaptive P1-FEM in Matlab, Comput. Methods Appl. Math. 11 (4) (2011) 460–490, doi:10.2478/cmam-2011-0026.
[4] M. Čermák, S. Sysala, J. Valdman, Efficient and flexible MATLAB implementation of 2d and 3d elastoplastic problems, Appl. Math. Comput. 355 (2019) 595–614, doi:10.1016/j.amc.2019.02.054.
[5] A. Moskovka, J. Valdman, Fast MATLAB evaluation of nonlinear energies using FEM in 2d and 3d: Nodal elements, Appl. Math. Comput. 424 (2022) 127048, doi:10.1016/j.amc.2022.127048.
[6] Y. Yu, varfem: variational formulation based programming for finite element methods in matlab (2022). 2206.06918
[7] MooAFEM: An object oriented Matlab library for adaptive FEM, (https://www.asc.tuwien.ac.at/~praetorius/?id=mooafem) Accessed on 2022-05-23.
[8] A. Ern, J.-L. Guermond, Theory and practice of finite elements, Applied Mathematical Sciences, volume 159, Springer-Verlag, New York, 2004, doi:10.1007/978-1-4757-4355-5.
[9] S. Bartels, C. Carstensen, G. Dolzmann, Inhomogeneous dirichlet conditions in a priori and a posteriori finite element error analysis, Numer. Math. 99 (1) (2004) 1–24, doi:10.1007/s00211-004-0548-3.
[10] C. Carstensen, M. Feischl, M. Page, D. Praetorius, Axioms of adaptivity, Comput. Math. Appl. 67 (6) (2014) 1195–1253, doi:10.1016/j.camwa.2013.12.003.
[11] M. Feischl, M. Page, D. Praetorius, Convergence and quasi-optimality of adaptive FEM with inhomogeneous Dirichlet data, J. Comput. Appl. Math. 255 (2014) 481–501, doi:10.1016/j.cam.2013.06.009.
[12] I. Babuška, The finite element method with penalty, Math. Comput. 27 (122) (1973) 221–228, doi:10.2307/2005611.
[13] J. Nitsche, Über ein Variationsprinzip zur Lösung von Dirichlet-Problemen bei Verwendung von Teilräumen, die keinen Randbedingungen unterworfen sind, Abh. Math. Semin. Univ. Hambg. 36 (1971) 9–15, doi:10.1007/BF02995904.
[14] M. Dudzinski, M. Rozgić, M. Stiemer, oFEM: An object oriented finite element package for matlab, Appl. Math. Comput. 334 (2018) 117–140, doi:10.1016/j.amc.2017.11.042.
[15] W. Bangerth, R. Rannacher, Adaptive Finite Element Methods for Differential Equations, Springer Basel AG, 2003, doi:10.1007/978-3-0348-7605-6.
[16] R. Stevenson, Optimality of a standard adaptive finite element method, Found. Comput. Math. 7 (2) (2007) 245–269, doi:10.1007/s10208-005-0183-0.
[17] R. Verfürth, A posteriori error estimation techniques for finite element methods, Numerical Mathematics and Scientific Computation, Oxford University Press, Oxford, 2013, doi:10.1093/acprof:oso/9780199679423.001.0001.
[18] W. Dörfler, A convergent adaptive algorithm for Poisson's equation, SIAM J. Numer. Anal. 33 (3) (1996) 1106–1124, doi:10.1137/0733054.
[19] R. Stevenson, The completion of locally refined simplicial partitions created by bisection, Math. Comput. 77 (261) (2008) 227–241, doi:10.1090/S0025-5718-07-01959-X.
[20] J. Schöberl, C++11 implementation of finite elements in NGSolve, 2014. ASC Report No. 30/2014.
[21] M. Alns, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M.E. Rognes, G.N. Wells, The fenics project version 1.5, Arch. Numer. Software Vol 3 (2015), doi:10.11588/ans.2015.100.20553.
[22] D. Arndt, W. Bangerth, D. Davydov, T. Heister, L. Heltai, M. Kronbichler, M. Maier, J.-P. Pelteret, B. Turcksin, D. Wells, The deal.II finite element library: Design, features, and insights, Comput. Math. Appl. 81 (2021) 407–422, doi:10.1016/j.camwa.2020.02.022.
[23] P. Bastian, M. Blatt, A. Dedner, N.-A. Dreier, C. Engwer, R. Fritze, C. Grser, C. Grninger, D. Kempf, R. Klfkorn, M. Ohlberger, O. Sander, The dune framework: Basic concepts and recent developments, Comput. Math. Appl. 81 (2021) 75–112, doi:10.1016/j.camwa.2020.06.007.
[24] R. Bulle, J.S. Hale, A. Lozinski, S.P.A. Bordas, F. Chouly, Hierarchical a posteriori error estimation of bank-weiser type in the fenics project (2021). 2102.04360
[25] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of reusable object-oriented software, volume 99, Addison-Wesley Reading, Massachusetts, 1995.
[26] M. Karkulik, D. Pavlicek, D. Praetorius, On 2D newest vertex bisection: optimality of mesh-closure and H^1-stability of L_2-projection, Constr. Approx. 38 (2) (2013) 213–234, doi:10.1007/s00365-013-9192-4.
[27] L. Diening, C. Kreuzer, R. Stevenson, Instance optimality of the adaptive maximum strategy, Found. Comput. Math. 16 (1) (2016) 33–68, doi:10.1007/s10208-014-9236-6.
[28] M. Innerberger, D. Praetorius, Instance-optimal goal-oriented adaptivity, Comput. Methods Appl. Math. 21 (1) (2021) 109–126, doi:10.1515/cmam-2019-0115.
[29] L. Zhang, T. Cui, H. Liu, A set of symmetric quadrature rules on triangles and tetrahedra, J. Comput. Math. 27(1) (2009) 89–96.
[30] M.G. Duffy, Quadrature over a pyramid or cube of integrands with a singularity at a vertex, SIAM J. Numer. Anal. 19 (6) (1982) 1260–1262, doi:10.1137/0719090.
[31] M. Ainsworth, G. Andriamaro, O. Davydov, Bernstein–Bézier finite elements of arbitrary order and optimal assembly procedures, SIAM J. Sci. Comput. 33 (6) (2011) 3087–3109, doi:10.1137/11082539X.
[32] M.S. Mommer, R. Stevenson, A goal-oriented adaptive finite element method with convergence rates, SIAM J. Numer. Anal. 47 (2) (2009) 861–886, doi:10.1137/060675666.
[33] M. Feischl, D. Praetorius, K.G. van der Zee, An abstract analysis of optimal goal-oriented adaptivity, SIAM J. Numer. Anal. 54 (3) (2016) 1423–1448, doi:10.1137/15M1021982.
[34] P. Heid, D. Praetorius, T.P. Wihler, Energy contraction and optimal convergence of adaptive iterative linearized finite element methods, Comput. Methods Appl. Math. 21 (2) (2021) 407–422, doi:10.1515/cmam-2021-0025.