Computational Methods in Applied Mathematics
Vol. XX (XX), No. XX, pp. 1–31
© XX Institute of Mathematics, National Academy of Sciences

Efficient Implementation of Adaptive P1-FEM in Matlab

S. Funken · D. Praetorius · P. Wissgott

Abstract — We provide a MATLAB package `p1afem` for an adaptive P1-finite element method (AFEM). This includes functions for the assembly of the data, different error estimators, and an indicator-based adaptive mesh-refining algorithm. Throughout, the focus is on an efficient realization by use of MATLAB built-in functions and vectorization. Numerical experiments underline the efficiency of the code which is observed to be of almost linear complexity with respect to the runtime. Although the scope of this paper is on AFEM, the general ideas can be understood as a guideline for writing efficient MATLAB code.

2010 Mathematical subject classification: 68N15; 65N30; 65M60.

Keywords: MATLAB program; Finite Element Method; Adaptivity; Mesh Refinement; Mesh Coarsening.

1. Introduction

In recent years, MATLAB has become 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 partial differential equations, cf. [22, 23]. In [1], an educational MATLAB code for the P1-Galerkin FEM is proposed which was designed for shortness and clarity. Whereas the given code seems to be of linear complexity with respect to the number of elements, the measurement of the computational time proves quadratic dependence instead. Since this is mainly due to the internal data structure of MATLAB, we show how to modify the existing MATLAB code so that the theoretically predicted complexity can even be measured in computations.

Moreover and in addition to [1], we provide a complete and easy-to-modify package called `p1afem` for adaptive P1-FEM computations, including three different a posteriori error estimators as well as an adaptive mesh-refinement based on a red-green-blue strategy (RGB) or newest vertex bisection (NVB). For the latter, we additionally provide an efficient implementation of the coarsening strategy from CHEN and ZHANG [12, 15]. All parts can

S. Funken
Institute for Numerical Mathematics, University of Ulm, Helmholtzstraße 18, D-89069 Ulm, Germany
E-mail: `Stefan.Funken@uni-ulm.de`.

D. Praetorius
Institute for Analysis and Scientific Computing, Vienna University of Technology, Wiedner Hauptstraße 8-10, A-1040 Wien, Austria
E-mail: `Dirk.Praetorius@tuwien.ac.at` (corresponding author).

P. Wissgott
Institute for Solid State Physics, Vienna University of Technology, Wiedner Hauptstraße 8-10, A-1040 Wien, Austria
E-mail: `wissgott@ifp.tuwien.ac.at`.

easily be combined with MATLAB implementations of other finite elements, cf. e.g. [2, 5, 8, 10, 26]. `p1afem` is implemented in a way, we expect to be optimal in MATLAB as a compromise between clarity, shortness, and use of MATLAB built-in functions. In particular, we use full vectorization in the sense that `for`-loops are eliminated by use of MATLAB vector operations.

The complete MATLAB code of `p1afem` can be downloaded from the web [19], and the technical report [20] provides a detailed documentation of the underlying ideas.

The remaining content of this paper is organized as follows: Section 2 introduces the model problem and the Galerkin scheme. In Section 3, we first recall the data structures of [1] as well as their MATLAB implementation. We discuss the reasons why this code leads to quadratic complexity in practice. Even simple modifications yield an improved code which behaves almost linearly. We show how the occurring `for`-loops can be eliminated by use of MATLAB's vector arithmetics which leads to a further improvement of the runtime. Section 4 gives a short overview on the functionality provided by `p1afem`, and selected functions are further discussed in the remainder of the paper: Section 5 is focused on local mesh-refinement and mesh-coarsening based on NVB. Section 6 provides a realization of a standard adaptive mesh-refining algorithm steered by the residual-based error estimator due to BABUŠKA and MILLER [4]. Section 7 concludes the paper with some numerical experiments and, in particular, comparisons with other MATLAB FEM packages like AFEM [16, 17] or iFEM [13, 14].

2. Model example and P1-Galerkin FEM

2.1. Continuous problem

As model problem, we consider the Laplace equation with mixed Dirichlet-Neumann boundary conditions. Given $f \in L^2(\Omega)$, $u_D \in H^{1/2}(\Gamma_D)$, and $g \in L^2(\Gamma_N)$, we aim to compute an approximation of the solution $u \in H^1(\Omega)$ of

$$\begin{aligned} -\Delta u &= f && \text{in } \Omega, \\ u &= u_D && \text{on } \Gamma_D, \\ \partial_n u &= g && \text{on } \Gamma_N. \end{aligned} \tag{1}$$

Here, Ω is a bounded Lipschitz domain in $\mathbb{R}^2$ whose polygonal boundary $\Gamma := \partial\Omega$ is split into a closed Dirichlet boundary Γ_D with positive length and a Neumann boundary $\Gamma_N := \Gamma\backslash\Gamma_D$. On Γ_N, we prescribe the normal derivative $\partial_n u$ of u, i.e. the flux. For theoretical reasons, we identify $u_D \in H^{1/2}(\Gamma_D)$ with some arbitrary extension $u_D \in H^1(\Omega)$. With

$$u_0 = u - u_D \in H^1_D(\Omega) := \{v \in H^1(\Omega) \,:\, v = 0 \text{ on } \Gamma_D\}, \tag{2}$$

the weak form reads: Find $u_0 \in H^1_D(\Omega)$ such that

$$\int_\Omega \nabla u_0 \cdot \nabla v \, dx = \int_\Omega f v \, dx + \int_{\Gamma_N} g v \, ds - \int_\Omega \nabla u_D \cdot \nabla v \, dx \quad \text{for all } v \in H^1_D(\Omega). \tag{3}$$

Functional analysis provides the unique existence of u_0 in the Hilbert space $H^1_D(\Omega)$, whence the unique existence of a weak solution $u := u_0 + u_D \in H^1(\Omega)$ of (1). Note that u does only depend on $u_D|_{\Gamma_D}$ so that one may consider the easiest possible extension u_D of the Dirichlet trace $u_D|_{\Gamma_D}$ from Γ_D to Ω.

2.2. P1-Galerkin FEM

Let $\mathcal{T}$ be a regular triangulation of Ω into triangles, i.e.

- $\mathcal{T}$ is a finite set of compact triangles $T = \mathrm{conv}\{z_1, z_2, z_3\}$ with positive area $|T| > 0$,
- the union of all triangles in $\mathcal{T}$ covers the closure $\overline{\Omega}$ of Ω,
- the intersection of different triangles is either empty, a common node, or a common edge,
- an edge may not intersect both, Γ_D and Γ_N, such that the intersection has positive length.

In particular, the partition of Γ into Γ_D and Γ_N is resolved by $\mathcal{T}$. Furthermore, hanging nodes are not allowed, cf. Figure 1 for an exemplary regular triangulation $\mathcal{T}$. Let

$$\mathcal{S}^1(\mathcal{T}) := \{V \in C(\Omega) \,:\, \forall T \in \mathcal{T} \quad V|_T \text{ affine}\} \tag{4}$$

denote the space of all globally continuous and $\mathcal{T}$-piecewise affine splines. With $\mathcal{N} = \{z_1, \dots, z_N\}$ the set of nodes of $\mathcal{T}$, we consider the nodal basis $\mathcal{B} = \{V_1, \dots, V_N\}$, where the hat function $V_\ell \in \mathcal{S}^1(\mathcal{T})$ is characterized by $V_\ell(z_k) = \delta_{k\ell}$ with Kronecker's delta. For the Galerkin method, we consider the space

$$\mathcal{S}^1_D(\mathcal{T}) := \mathcal{S}^1(\mathcal{T}) \cap H^1_D(\Omega) = \{V \in \mathcal{S}^1(\mathcal{T}) \,:\, \forall z_\ell \in \mathcal{N} \cap \Gamma_D \quad V(z_\ell) = 0\}. \tag{5}$$

Without loss of generality, there holds $\mathcal{N} \cap \Gamma_D = \{z_{n+1}, \dots, z_N\}$. We assume that the Dirichlet data $u_D \in H^1(\Omega)$ are continuous on Γ_D and replace $u_D|_{\Gamma_D}$ by its nodal interpolant

$$U_D := \sum_{\ell=n+1}^{N} u_D(z_\ell) V_\ell \in \mathcal{S}^1(\mathcal{T}). \tag{6}$$

The discrete variational form

$$\int_\Omega \nabla U_0 \cdot \nabla V \, dx = \int_\Omega f V \, dx + \int_{\Gamma_N} g V \, ds - \int_\Omega \nabla U_D \cdot \nabla V \, dx \quad \text{for all } V \in \mathcal{S}^1_D(\mathcal{T}) \tag{7}$$

then has a unique solution $U_0 \in \mathcal{S}^1_D(\mathcal{T})$ which provides an approximation $U := U_0 + U_D \in \mathcal{S}^1(\mathcal{T})$ of $u \in H^1(\Omega)$. We aim to compute the coefficient vector $\mathbf{x} \in \mathbb{R}^N$ of $U \in \mathcal{S}^1(\mathcal{T})$ with respect to the nodal basis $\mathcal{B}$

$$U_0 = \sum_{j=1}^{n} \mathbf{x}_j V_j, \quad \text{whence} \quad U = \sum_{j=1}^{N} \mathbf{x}_j V_j \quad \text{with } \mathbf{x}_j := u_D(z_j) \text{ for } j = n+1, \dots, N. \tag{8}$$

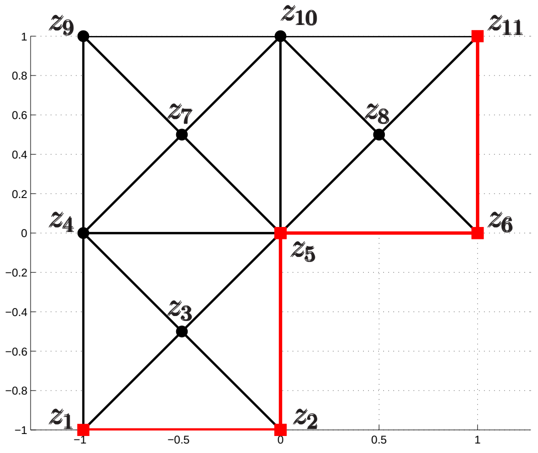
Note that the discrete variational form (7) is equivalent to the linear system

$$\sum_{k=1}^{n} \mathbf{A}_{jk} \mathbf{x}_k = \mathbf{b}_j := \int_\Omega f V_j \, dx + \int_{\Gamma_N} g V_j \, ds - \sum_{k=n+1}^{N} \mathbf{A}_{jk} \mathbf{x}_k \quad \text{for all } j = 1, \dots, n \tag{9}$$

with stiffness matrix entries

$$\mathbf{A}_{jk} = \int_\Omega \nabla V_j \cdot \nabla V_k \, dx = \sum_{T \in \mathcal{T}} \int_T \nabla V_j \cdot \nabla V_k \, dx \quad \text{for all } j, k = 1, \dots, N. \tag{10}$$

For the implementation, we build $\mathbf{A} \in \mathbb{R}^{N \times N}_{sym}$ with respect to all nodes and then solve (9) on the $n \times n$ subsystem corresponding to the free nodes.



	`coordinates`	
1	−1.0	−1.0
2	0.0	−1.0
3	−0.5	−0.5
4	−1.0	0.0
5	0.0	0.0
6	1.0	0.0
7	−0.5	0.5
8	0.5	0.5
9	−1.0	1.0
10	0.0	1.0
11	1.0	1.0

	`elements`		
1	1	2	3
2	2	5	3
3	5	4	3
4	4	1	3
5	4	5	7
6	5	10	7
7	10	9	7
8	9	4	7
9	5	6	8
10	6	11	8
11	11	10	8
12	10	5	8

	`dirichlet`	
1	1	2
2	2	5
3	5	6
4	6	11

	`neumann`	
1	11	10
2	10	9
3	9	4
4	4	1

Figure 1. Exemplary triangulation $\mathcal{T}$ of the L-shaped domain $\Omega = (-1,1)^2\backslash([0,1]\times[-1,0])$ into 12 triangles specified by the arrays `coordinates` and `elements`. The Dirichlet boundary, specified by the array `dirichlet`, consists of 4 edges which are plotted in red. The Neumann boundary is specified by the array `neumann` and consists of the remaining 4 boundary edges. The nodes $\mathcal{N} \cap \Gamma_D = \{z_1, z_2, z_5, z_6, z_{11}\}$ are indicated by red squares, whereas free nodes $\mathcal{N}\backslash\Gamma_D = \{z_3, z_4, z_7, z_8, z_9, z_{10}\}$ are indicated by black bullets.

3. Matlab implementation of P1-Galerkin FEM

In this section, we recall the Matlab implementation of the P1-FEM from [1] and explain, why this code leads to a quadratic growth of the runtime with respect to the number of elements. We then discuss how to write an efficient Matlab code by use of vectorization.

3.1. Data structures and visualization of discrete functions

For the data representation of the set of all nodes $\mathcal{N} = \{z_1, \dots, z_N\}$, the regular triangulation $\mathcal{T} = \{T_1, \dots, T_M\}$, and the boundaries Γ_D and Γ_N, we follow [1]: We refer to Figure 1 for an exemplary triangulation $\mathcal{T}$ and corresponding data arrays, which are formally specified in the following:

The set of all nodes $\mathcal{N}$ is represented by the $N \times 2$ array `coordinates`. The ℓ-th row of `coordinates` stores the coordinates of the ℓ-th node $z_\ell = (x_\ell, y_\ell) \in \mathbb{R}^2$ as

$$\texttt{coordinates(}\ell\texttt{,:)} = [\ x_\ell \quad y_\ell\ ].$$

The triangulation $\mathcal{T}$ is represented by the $M \times 3$ integer array `elements`. The ℓ-th triangle $T_\ell = \mathrm{conv}\{z_i, z_j, z_k\} \in \mathcal{T}$ with vertices $z_i, z_j, z_k \in \mathcal{N}$ is stored as

$$\texttt{elements(}\ell\texttt{,:)} = [\ i \quad j \quad k\ ],$$

where the nodes are given in counterclockwise order, i.e., the parametrization of the boundary ∂T_ℓ is mathematically positive.

The Dirichlet boundary Γ_D is split into K affine boundary pieces, which are edges of triangles $T \in \mathcal{T}$. It is represented by a $K \times 2$ integer array `dirichlet`. The ℓ-th edge $E_\ell = \mathrm{conv}\{z_i, z_j\}$ on the Dirichlet boundary is stored in the form

$$\texttt{dirichlet(}\ell\texttt{,:)}=[\ i \quad j\ ].$$

Listing 1. An educational but inefficient MATLAB implementation

```
1  function [x,energy] = solveLaplace(coordinates,elements,dirichlet,neumann,f,g,uD)
2  nC = size(coordinates,1);
3  x = zeros(nC,1);
4  %*** Assembly of stiffness matrix
5  A = sparse(nC,nC);
6  for i = 1:size(elements,1)
7      nodes = elements(i,:);
8      B = [1 1 1 ; coordinates(nodes,:)'];
9      grad = B \ [0 0 ; 1 0 ; 0 1];
10     A(nodes,nodes) = A(nodes,nodes) + det(B)*grad*grad'/2;
11 end
12 %*** Prescribe values at Dirichlet nodes
13 dirichlet = unique(dirichlet);
14 x(dirichlet) = feval(uD,coordinates(dirichlet,:));
15 %*** Assembly of right-hand side
16 b = -A*x;
17 for i = 1:size(elements,1)
18     nodes = elements(i,:);
19     sT = [1 1 1]*coordinates(nodes,:)/3;
20     b(nodes) = b(nodes) + det([1 1 1 ; coordinates(nodes,:)'])*feval(f,sT)/6;
21 end
22 for i = 1:size(neumann,1)
23     nodes = neumann(i,:);
24     mE = [1 1]*coordinates(nodes,:)/2;
25     b(nodes) = b(nodes) + norm([1 -1]*coordinates(nodes,:))*feval(g,mE)/2;
26 end
27 %*** Computation of P1-FEM approximation
28 freenodes = setdiff(1:nC, dirichlet);
29 x(freenodes) = A(freenodes,freenodes)\b(freenodes);
30 %*** Compute energy || grad(uh) ||^2 of discrete solution
31 energy = x'*A*x;
```

It is assumed that $z_j - z_i$ gives the mathematically positive orientation of Γ, i.e.

$$n_\ell = \frac{1}{|z_j - z_i|}\begin{pmatrix} y_j - y_i \\ x_i - x_j \end{pmatrix},$$

gives the outer normal vector of Ω on E_ℓ, where $z_k = (x_k, y_k) \in \mathbb{R}^2$. Finally, the Neumann boundary Γ_N is stored analogously within an $L \times 2$ integer array `neumann`.

Using this data structure, we may visualize a discrete function $U = \sum_{j=1}^N \mathbf{x}_j V_j \in \mathcal{S}^1(\mathcal{T})$ by

```
trisurf(elements,coordinates(:,1),coordinates(:,2),x,'facecolor','interp')
```

Here, the column vector $\mathbf{x}_j = U(z_j)$ contains the nodal values of U at the j-th node $z_j \in \mathbb{R}^2$ given by `coordinates(j,:)`.

3.2. An educational but inefficient MATLAB implementation (Listing 1)

This section essentially recalls the MATLAB code of [1] for later reference. We emphasize that the implementation of [1] put the focus on shortness and clarity to explain the ideas on how to implement finite elements in MATLAB.

- Line 1: As input, the function `solveLaplace` takes the description of a triangulation $\mathcal{T}$ as well as functions for the volume forces f, the Neumann data g, and the Dirichlet data u_D. According to the MATLAB 7 standard, these functions may be given as function handles or as strings containing the function names. Either function takes n evaluation points $\xi_j \in \mathbb{R}^2$ in form of a matrix $\xi \in \mathbb{R}^{n\times 2}$ and returns a column vector $y \in \mathbb{R}^n$ of the associated function values, i.e., $y_j = f(\xi_j)$. Finally, the function `solveLaplace` returns the coefficient vector $\mathbf{x}_j = U(z_j)$ of the discrete solution $U \in \mathcal{S}^1(\mathcal{T})$, cf. (8), as well as its energy $\|\nabla U\|^2_{L^2(\Omega)} = \sum_{j,k=1}^N \mathbf{x}_j \mathbf{x}_k \int_\Omega \nabla V_j \cdot \nabla V_k \, dx = \mathbf{x} \cdot \mathbf{A}\mathbf{x}$.

- Lines 5–11: The stiffness matrix $\mathbf{A} \in \mathbb{R}^{N\times N}_{sym}$ is built elementwise as indicated in (10). We stress that, for $T_i \in \mathcal{T}$ and piecewise affine basis functions, a summand

$$\int_{T_i} \nabla V_j \cdot \nabla V_k \, dx = |T_i| \, \nabla V_j|_{T_i} \cdot \nabla V_k|_{T_i}$$

vanishes if not both z_j and z_k are nodes of T_i. We thus may assemble $\mathbf{A}$ simultaneously for all $j, k = 1, \dots, N$, where we have a (3×3)-update of $\mathbf{A}$ per element $T_i \in \mathcal{T}$. The matrix $B \in \mathbb{R}^{3\times 3}$ in Line 8 provides $|T_i| = \det(B)/2$. Moreover, `grad(`ℓ`,:)` from Line 9 contains the gradient of the hat function $V_j|_{T_i}$ for j-th node z_j, where `j=elements(i,`ℓ`)`.

- Lines 13–14: The entries of the coefficient vector $\mathbf{x} \in \mathbb{R}^N$ which correspond to Dirichlet nodes, are initialized, cf. (8).

- Lines 16–26: The load vector $\mathbf{b} \in \mathbb{R}^N$ from (9) is built. It is initialized by the contribution of the nodal interpolation of the Dirichlet data (Line 16), cf. (7) resp. (9). Next (Lines 17–21), we elementwise add the volume force

$$\int_\Omega f V_j \, dx = \sum_{T\in\mathcal{T}} \int_T f V_j \, dx \approx \sum_{T\in\mathcal{T}} |T| f(s_T) V_j(s_T).$$

Again, we stress that, for $T \in \mathcal{T}$, a summand $\int_T f V_j \, dx$ vanishes if z_j is not a node of T. Each element T thus enforces an update of three components of $\mathbf{b}$ only. The integral is computed by a 1-point quadrature with respect to the center of mass $s_T \in T$, where $V_j(s_T) = 1/3$. Finally (Lines 22–26), we elementwise add the Neumann contributions

$$\int_{\Gamma_N} g V_j \, ds = \sum_{E\subseteq\Gamma_N} \int_E g V_j \, ds \approx \sum_{E\subseteq\Gamma_N} h_E g(m_E) V_j(m_E).$$

Again, for each edge E on the Neumann boundary, only two components of the load vector $\mathbf{b}$ are effected. The boundary integral is computed by a 1-point quadrature with respect to the edge's midpoint $m_E \in E$, where $V_j(m_E) = 1/2$ and where h_E denotes the edge length.

- Lines 28–29: We first compute the indices of all free nodes $z_j \notin \Gamma_D$ (Line 28). Then, we solve the linear system (9) for the coefficients $\mathbf{x}_j$ which correspond to free nodes $z_j \notin \Gamma_D$ (Line 29). Note that this does not effect the coefficients $\mathbf{x}_k = u_D(z_k)$ corresponding to Dirichlet nodes $z_k \in \Gamma_D$ so that $\mathbf{x} \in \mathbb{R}^N$ finally is, in fact, the coefficient vector of the P1-FEM solution $U \in \mathcal{S}^1(\mathcal{T})$, cf. (8).

On a first glance, one might expect linear runtime of the function `solveLaplace` with respect to the number M of elements — at least up to the solution of the linear system in Line 29. Instead, one observes a quadratic dependence, cf. Figure 5.

Listing 2. Assembly of stiffness matrix in almost linear complexity (intermediate implementation)

```
 1 %*** Assembly of stiffness matrix in linear complexity
 2 nE = size(elements,1);
 3 I = zeros(9*nE,1);
 4 J = zeros(9*nE,1);
 5 A = zeros(9*nE,1);
 6 for i = 1:nE
 7     nodes = elements(i,:);
 8     B = [1 1 1 ; coordinates(nodes,:)'];
 9     grad = B \ [0 0 ; 1 0 ; 0 1];
10     idx = 9*(i-1)+1:9*i;
11     tmp = [1;1;1]*nodes;
12     I(idx) = reshape(tmp',9,1);
13     J(idx) = reshape(tmp,9,1);
14     A(idx) = det(B)/2*reshape(grad*grad',9,1);
15 end
16 A = sparse(I,J,A,nC,nC);
```

3.3. Reasons for MATLAB's inefficiency and some first remedy (Listing 2)

A closer look on the MATLAB code of the function `solveLaplace` in Listing 1 reveals that the quadratic runtime is due to the assembly of $\mathbf{A} \in \mathbb{R}^{N\times N}$ (Lines 5–11): In MATLAB, sparse matrices are internally stored in the compressed column storage format (or: Harwell-Boeing format), cf. [7] for an introduction to storage formats for sparse matrices. Therefore, updating a sparse matrix with new entries, necessarily needs the prolongation and sorting of the storage vectors. For each step i in the update of a sparse matrix, we are thus led to at least $\mathcal{O}(i)$ operations, which results in an overall complexity of $\mathcal{O}(M^2)$ for building the stiffness matrix, where $M = \#\mathcal{T}$.

As has been pointed out by GILBERT, MOLER, and SCHREIBER [21], MATLAB provides some simple remedy for the otherwise inefficient building of sparse matrices: Let $a \in \mathbb{R}^n$ and $I, J \in \mathbb{N}^n$ be the vectors for the coordinate format of some sparse matrix $\mathbf{A} \in \mathbb{R}^{M\times N}$. Then, $\mathbf{A}$ can be declared and initialized by use of the MATLAB command

```
A = sparse(I,J,a,M,N)
```

where, in general, $\mathbf{A}_{ij} = a_\ell$ for $i = I_\ell$ and $j = J_\ell$. If an index pair, $(i,j) = (I_\ell, J_\ell)$ appears twice (or even more), the corresponding entries a_ℓ are added. In particular, the internal realization only needs one sorting of the entries which appears to be of complexity $\mathcal{O}(n \log n)$.

For the assembly of the stiffness matrix, we now replace Lines 5–11 of Listing 1 by Lines 2–16 of Listing 2. We only comment on the differences of Listing 1 and Listing 2 in the following and stress that the remaining `for` loop is finally avoided by vectorization in Listing 3 below.

- Lines 2–5: Note that the elementwise assembly of $\mathbf{A}$ in Listing 1 uses nine updates of the stiffness matrix per element, i.e. the vectors I, J, and a have length $9M$ with $M = \#\mathcal{T}$ the number of elements.

- Lines 10–14: Dense matrices are stored columnwise in MATLAB, i.e., a matrix $V \in \mathbb{R}^{M\times N}$ is stored in a vector $v \in \mathbb{R}^{MN}$ with $V_{jk} = v_{j+(k-1)M}$. For fixed `i` and `idx` in Lines 10–14, there consequently hold

  ```
  I(idx) = elements(i,[1 2 3 1 2 3 1 2 3]);
  J(idx) = elements(i,[1 1 1 2 2 2 3 3 3]);
  ```

Therefore, `I(idx)` and `J(idx)` address the same entries of $\mathbf{A}$ as has been done in Line 10 of Listing 1. Note that we compute the same matrix updates a as in Line 10 of Listing 1.

- Line 16: The sparse matrix $\mathbf{A} \in \mathbb{R}^{N\times N}$ is built from the three coordinate vectors.

A comparison of the assembly times for the stiffness matrix $\mathbf{A}$ by use of the naive code (Lines 5–11 of Listing 1) and the improved code (Lines 2–16 of Listing 2) in Figure 5 below reveals that the new code has almost linear complexity with respect to $M = \#\mathcal{T}$. A further improvement by vectorization is discussed in the following section, and we also refer to [13, 26] for the idea of vectorizing MATLAB codes. Moreover, the work [26] puts emphasis on the unifying vectorization of the matrix assembly for elliptic PDEs and isoparametric elements in 2D and 3D, which is an interesting approach.

Listing 3. A fully vectorized and efficient MATLAB implementation

```matlab
function [x,energy] = solveLaplace(coordinates,elements,dirichlet,neumann,f,g,uD)
nE = size(elements,1);
nC = size(coordinates,1);
x = zeros(nC,1);
%*** First vertex of elements and corresponding edge vectors
c1 = coordinates(elements(:,1),:);
d21 = coordinates(elements(:,2),:) - c1;
d31 = coordinates(elements(:,3),:) - c1;
%*** Vector of element areas 4*|T|
area4 = 2*(d21(:,1).*d31(:,2)-d21(:,2).*d31(:,1));
%*** Assembly of stiffness matrix
I = reshape(elements(:,[1 2 3 1 2 3 1 2 3])',9*nE,1);
J = reshape(elements(:,[1 1 1 2 2 2 3 3 3])',9*nE,1);
a = (sum(d21.*d31,2)./area4)';
b = (sum(d31.*d31,2)./area4)';
c = (sum(d21.*d21,2)./area4)';
A = [-2*a+b+c;a-b;a-c;a-b;b;-a;a-c;-a;c];
A = sparse(I,J,A(:));
%*** Prescribe values at Dirichlet nodes
dirichlet = unique(dirichlet);
x(dirichlet) = feval(uD,coordinates(dirichlet,:));
%*** Assembly of right-hand side
fsT = feval(f,c1+(d21+d31)/3);
b = accumarray(elements(:),repmat(12\area4.*fsT,3,1),[nC 1]) - A*x;
if ~isempty(neumann)
    cn1 = coordinates(neumann(:,1),:);
    cn2 = coordinates(neumann(:,2),:);
    gmE = feval(g,(cn1+cn2)/2);
    b = b + accumarray(neumann(:),...
                     repmat(2\sqrt(sum((cn2-cn1).^2,2)).*gmE,2,1),[nC 1]);
end
%*** Computation of P1-FEM approximation
freenodes = setdiff(1:nC, dirichlet);
x(freenodes) = A(freenodes,freenodes)\b(freenodes);
%*** Compute energy || grad(uh) ||^2 of discrete solution
energy = x'*A*x;
```

3.4. Further improvement by vectorization (Listing 3)

In this section, we further improve the overall runtime of the function `solveLaplace` of Listing 1 and 2. All of the following techniques are based on the empirical observation that vectorized code is always faster than the corresponding implementation using loops. Besides `sparse` discussed above, we shall use the following tools for performance acceleration, provided by MATLAB:

- Dense matrices $A \in \mathbb{R}^{M\times N}$ are stored columnwise in MATLAB, and `A(:)` returns the column vector as used for the internal storage. Besides this, one may use

  ```
  B = reshape(A,m,n)
  ```

  to change the shape into $B \in \mathbb{R}^{m\times n}$ with $MN = mn$, where `B(:)` coincides with `A(:)`.

- The coordinate format of a (sparse or even dense) matrix $A \in \mathbb{R}^{M\times N}$ is returned by

  ```
  [I,J,a] = find(A).
  ```

  With $n \in \mathbb{N}$ the number of nonzero-entries of A, there holds $I, J \in \mathbb{N}^n$, $a \in \mathbb{R}^n$, and $A_{ij} = a_\ell$ with $i = I_\ell$ and $j = J_\ell$. Moreover, the vectors are columnwise ordered with respect to A.

- Fast assembly of dense matrices $A \in \mathbb{R}^{M\times N}$ is done by

  ```
  A = accumarray(I,a,[M N])
  ```

  with $I \in \mathbb{N}^{n\times 2}$ and $a \in \mathbb{R}^n$. The entries of A are then given by $A_{ij} = a_\ell$ with $i = I_{\ell 1}$ and $j = I_{\ell 2}$. As for `sparse`, multiple occurrence of an index pair $(i, j) = (I_{\ell 1}, I_{\ell 2})$ leads to the summation of the associated values a_ℓ.

- For a matrix $A \in \mathbb{R}^{M\times N}$, the rowwise sum $a \in \mathbb{R}^M$ with $a_j = \sum_{k=1}^N A_{jk}$ is obtained by

  ```
  a = sum(A,2).
  ```

  The columnwise sum $b \in \mathbb{R}^N$ is computed by `b = sum(A,1)`.

- Finally, linear arithmetics is done by usual matrix-matrix operations, e.g., `A+B` or `A*B`, whereas nonlinear arithmetics is done by pointwise arithmetics, e.g., `A.*B` or `A.^2`.

To improve the runtime of the function `solveLaplace`, we first note that function calls are generically expensive. We therefore reduce the function calls to the three necessary calls in Line 21, 23, and 28 to evaluate the data functions u_D, f, and g, respectively. Second, a further improvement can be achieved by use of MATLAB's vector arithmetics which allows to replace any `for`-loop.

- Line 10: Let $T = \mathrm{conv}\{z_1, z_2, z_3\}$ denote a non-degenerate triangle in $\mathbb{R}^2$, where the vertices z_1, z_2, z_3 are given in counterclockwise order. With vectors $v = z_2 - z_1$ and $w = z_3 - z_1$, the area of T then reads

  $$2|T| = \det \begin{pmatrix} v_1 & w_1 \\ v_2 & w_2 \end{pmatrix} = v_1 w_2 - v_2 w_1.$$

  Consequently, Line 10 computes the areas of all elements $T \in \mathcal{T}$ simultaneously.

- Line 12–18: We assemble the stiffness matrix $\mathbf{A} \in \mathbb{R}^{N\times N}_{sym}$ as in Listing 2. However, the coordinate vectors I, J, and a are now assembled simultaneously for all elements $T \in \mathcal{T}$ by use of vector arithmetics.

- Lines 23–30: Assembly of the load vector, cf. Lines 16–26 in Listing 1 above: In Line 23, we evaluate the volume forces $f(s_T)$ in the centers s_T of all elements $T \in \mathcal{T}$ simultaneously. We initialize $\mathbf{b}$ with the contribution of the volume forces and of the Dirichlet data (Line 24). If the Neumann boundary is non-trivial, we evaluate the Neumann data $g(m_E)$ in all midpoints m_E of Neumann edges $E \in \mathcal{E}_N$ simultaneously (Line 28). Finally, Line 29 is the vectorized variant of Lines 22–26 in Listing 1.

In Figure 5 below, the fully vectorized implementation of Listing 3 is observed to be 10 times faster than the intermediate implementation of Listing 2. All further implementations of `p1afem` are thus fully vectorized.

4. Overview on functions provided by P1AFEM-package

Our software package `p1afem` provides several modules. To abbreviate the notation for the parameters, we write `C` (`coordinates`), `E` (`elements`), `D` (`dirichlet`), and `N` (`neumann`). For the MATLAB functions which provide the volume data f and the boundary data g and u_D, we assume that either function takes n evaluation points $\xi_j \in \mathbb{R}^2$ in form of a matrix $\xi \in \mathbb{R}^{n\times 2}$ and returns a column vector $y \in \mathbb{R}^n$ of the associated function values, i.e., $y_j = f(\xi_j)$.

Altogether, `p1afem` provides 14 MATLAB functions for the solution of the Laplace equation (`solveLaplace`, `solveLaplace0`, `solveLaplace1`), for local mesh-refinement (`refineNVB`, `refineNVB1`, `refineNVB5`, `refineRGB`, `refineMRGB`), for local mesh-coarsening (`coarsenNVB`), for a posteriori error estimation (`computeEtaR`, `computeEtaH`, `computeEtaZ`). Furthermore, it contains an implementation of a standard adaptive algorithm (`adaptiveAlgorithm`) and an auxiliary function (`provideGeometricData`) which induces a numbering of the edges. For demonstration purposes, the package is complemented by two numerical experiments contained in subdirectories (`example1/`, `example2/`).

4.1. Solving the 2D Laplace equation

The vectorized solver `solveLaplace` is called by

```
[x,energy] = solveLaplace(C,E,D,N,f,g,uD);
```

Details are given in Section 3.4. The functions `solveLaplace0` and `solveLaplace1` are called with the same arguments and realize the preliminary solvers from Section 3.2–3.3.

4.2. Local mesh refinement for triangular meshes

The function for mesh-refinement by newest vertex bisection is called by

```
[C,E,D,N] = refineNVB(C,E,D,N,marked);
```

where `marked` is a vector with the indices of marked elements. The result is a regular triangulation, where all marked elements have been refined by three bisections. Our implementation is flexible in the sense that it only assumes that the boundary Γ is partitioned into finitely many boundaries (instead of precisely into Γ_D and Γ_N). Details are given in Section 5.2.

In addition, we provide implementations of further mesh-refining strategies, which are called with the same arguments: With `refineNVB1`, marked elements are only refined by one bisection. With `refineNVB5`, marked elements are refined by five bisections, and the refined mesh thus has the *interior node property* from [24]. Finally, `refineRGB` provides an implementation of a red-green-blue strategy which is discussed in detail in the technical report [20, Section 5.3], and `refineMRGB` provides a modified red-green-blue refinement from [9] which mathematically guarantees stability of the L^2-projection. As stated above, all these additional mesh-refinement procedures are part of the current `p1afem` library which can be downloaded from the web [19].

4.3. Local mesh coarsening for NVB-refined meshes

For triangulations generated by newest vertex bisection,

```
[C,E,D,N] = coarsenNVB(N0,C,E,D,N,marked);
```

provides the implementation of a coarsening algorithm from [15], where `marked` is a vector with the indices of elements marked for coarsening and where `N0` denotes the number of nodes in the initial mesh $\mathcal{T}_0$. Details are given in Section 5.3.

4.4. A posteriori error estimators

The residual-based error estimator η_R due to BABUŠKA and MILLER [4] is called by

```
etaR = computeEtaR(x,C,E,D,N,f,g);
```

where `x` is the coefficient vector of the P1-FEM solution. The return value `etaR` is a vector containing the local contributions which are used for element marking in an adaptive mesh-refining algorithm. One focus of our implementation is on the efficient computation of the edge-based contributions. Details are found in Section 6.2.

Moreover, `computeEtaH` is called by the same arguments and returns the hierarchical error estimator η_H due to BANK and SMITH [6]. An error estimator η_Z based on the gradient recovery technique proposed by ZIENKIEWICZ and ZHU [31] is returned by

```
etaZ = computeEtaZ(x,C,E);
```

For the implementations of η_H and η_Z, we refer to the technical report [20, Section 6.3–6.4].

4.5. Adaptive mesh-refining algorithm

We include an adaptive algorithm called by

```
[x,C,E,etaR] = adaptiveAlgorithm(C,E,D,N,f,g,uD,nEmax,theta);
```

Here, $\theta \in (0,1)$ is a given adaptivity parameter and `nEmax` is the maximal number of elements. The function returns the adaptively generated mesh, the coefficient vector for the corresponding P1-FEM solution and the associated refinement indicators to provide an upper bound for the (unknown) error. Details are given in Section 6.1.

4.6. Auxiliary function

Our implementation of the error estimators `computeEtaR` and `computeEtaH` and the local mesh-refinement (`refineNVB`, `refineNVB1`, `refineNVB5`, `refineRGB`) needs some numbering of edges. This information is provided by call of the auxiliary function

```
[edge2nodes,element2edges,dirichlet2edges,neumann2edges]
     = provideGeometricData(E,D,N);
```

Details are given in Section 5.1.

Listing 4. Fast computation of geometric data concerning element edges

```
function [edge2nodes,element2edges,varargout] ...
                = provideGeometricData(elements,varargin)
nE = size(elements,1);
nB = nargin-1;
%*** Node vectors of all edges (interior edges appear twice)
I = elements(:);
J = reshape(elements(:,[2,3,1]),3*nE,1);
%*** Symmetrize I and J (so far boundary edges appear only once)
pointer = [1,3*nE,zeros(1,nB)];
for j = 1:nB
    boundary = varargin{j};
    if ~isempty(boundary)
        I = [I;boundary(:,2)];
        J = [J;boundary(:,1)];
    end
    pointer(j+2) = pointer(j+1) + size(boundary,1);
end
%*** Create numbering of edges
idxIJ = find(I < J);
edgeNumber = zeros(length(I),1);
edgeNumber(idxIJ) = 1:length(idxIJ);
idxJI = find(I > J);
number2edges = sparse(I(idxIJ),J(idxIJ),1:length(idxIJ));
[foo{1:2},numberingIJ] = find( number2edges );
[foo{1:2},idxJI2IJ] = find( sparse(J(idxJI),I(idxJI),idxJI) );
edgeNumber(idxJI2IJ) = numberingIJ;
%*** Provide element2edges and edge2nodes
element2edges = reshape(edgeNumber(1:3*nE),nE,3);
edge2nodes = [I(idxIJ),J(idxIJ)];
%*** Provide boundary2edges
for j = 1:nB
    varargout{j} = edgeNumber(pointer(j+1)+1:pointer(j+2));
end
```

4.7. Examples and demo files

The implementation for the numerical experiments from Section 7 are provided in subdirectories `example1/` and `example2/`.

5. Local mesh refinement and coarsening

The accuracy of a discrete solution U of u depends on the triangulation $\mathcal{T}$ in the sense that the data f, g, and u_D as well as possible singularities of u have to be resolved by the triangulation. With the adaptive algorithm of Section 6, this can be done automatically by use of some (local) mesh-refinement to improve $\mathcal{T}$. In this section, we discuss our MATLAB implementation of mesh-refinement based on newest vertex bisection (NVB). Moreover, in parabolic problems the solution u usually becomes smoother if time proceeds. Since the computational complexity depends on the number of elements, one may then want to remove certain elements from $\mathcal{T}$. For NVB, this (local) mesh coarsening can be done efficiently without storing any further data. Although the focus of this paper is not on time dependent problems, we include an efficient implementation of a coarsening algorithm from [15] for the sake of completeness.

5.1. Efficient computation of geometric relations (Listing 4)

For many computations, one needs further geometric data besides the arrays `coordinates`, `elements`, `dirichlet`, and `neumann`. For instance, the mesh-refinement provided below is

edge-based. In particular, we need to generate a numbering of the edges of $\mathcal{T}$. In addition, we need the information which edges belong to a given element and which nodes belong to a given edge.
The necessary data is generated by the function `provideGeometricData` of Listing 4, where we build two additional arrays: For an edge E_ℓ, `edge2nodes(`ℓ`,:)` provides the numbers j, k of the nodes $z_j, z_k \in \mathcal{N}$ such that $E_\ell = \text{conv}\{z_j, z_k\}$. Moreover, `element2edges(i,`$\ell$`)` returns the number of the edge between the nodes `elements(i,`ℓ`)` and `elements(i,`$\ell+1$`)`, where we identify the index $\ell + 1 = 4$ with $\ell = 1$. Finally, we return the numbers of the boundary edges, e.g., `dirichlet2edges(`ℓ`)` is the absolute number of the ℓ-th edge on the Dirichlet boundary.

- Line 1: The function is usually called by

```
[edge2nodes,element2edges,dirichlet2edges,neumann2edges] ...
    = provideGeometricData(elements,dirichlet,neumann)
```

  where the partition of Γ into certain boundary conditions is hidden in the optional arguments `varargin` and `varargout`. This allows to handle any partition of Γ into finitely many boundary conditions (instead of precisely two, namely Γ_D and Γ_N).

- Lines 6–7: We generate node vectors I and J which describe the edges of $\mathcal{T}$: All directed edges $E = \text{conv}\{z_i, z_j\}$ of $\mathcal{T}$ with $z_i, z_j \in \mathcal{N}$ and tangential vector $z_j - z_i$ of $\mathcal{T}$ appear in the form $(i, j) \in \{(I_\ell, J_\ell) : \ell = 1, 2, \dots\} =: G$.

- Lines 9–17: Note that a pair $(i, j) \in G$ is an interior edge of $\mathcal{T}$ if and only if $(j, i) \in G$. We prolongate G by adding the pair (j, i) to G whenever (i, j) is a boundary edge. Then, G is symmetrized in the sense that (i, j) belongs to G if and only if (j, i) belongs to G.

- Lines 19–26: Create a numbering of the edges and an index vector such that the vector `edgeNumber(`ℓ`)` returns the edge number of the edge (I_ℓ, J_ℓ): So far, each edge E of $\mathcal{T}$ appears twice in G as pair (i, j) and (j, i). To create a numbering of the edges, we consider all pairs (i, j) with $i < j$ and fix a numbering (Lines 19–21). Finally, we need to ensure the same edge number for (j, i) as for (i, j). Note that G corresponds to a sparse matrix with symmetric pattern. We provide the coordinate format of the upper triangular part of G, where the entries are the already prescribed edge numbers (Line 23). Next, we provide the coordinate format of the upper triangular part of the transpose G^T, where the entries are the indices with respect to I and J (Line 25). This provides the necessary information to store the correct edge number of all edges (j, i) with $i < j$ (Line 26).

- Lines 28–29: Generate arrays `element2edges` and `edge2nodes`.

- Lines 31–33: Generate, e.g. `dirichlet2edges`, to link boundary edges and numbering of edges.

Remark 5.1. In Lines 24–25 of Listing 4 and the listings below, we use the variable `foo` to store additional return values of MATLAB functions which are not used by our implementation. Since MATLAB Release 2009b, one could also use the symbol $\sim$ which then avoids unnecessary memory allocation and computations to build the corresponding return values.

Listing 5. NVB refinement

```matlab
function [coordinates,newElements,varargout] ...
              = refineNVB(coordinates,elements,varargin)
markedElements = varargin{end};
nE = size(elements,1);
%*** Obtain geometric information on edges
[edge2nodes,element2edges,boundary2edges{1:nargin-3}] ...
    = provideGeometricData(elements,varargin{1:end-1});
%*** Mark edges for refinement
edge2newNode = zeros(max(max(element2edges)),1);
edge2newNode(element2edges(markedElements,:)) = 1;
swap = 1;
while ~isempty(swap)
    markedEdge = edge2newNode(element2edges);
    swap = find( ~markedEdge(:,1) & (markedEdge(:,2) | markedEdge(:,3)) );
    edge2newNode(element2edges(swap,1)) = 1;
end
%*** Generate new nodes
edge2newNode(edge2newNode ~= 0) = size(coordinates,1) + (1:nnz(edge2newNode));
idx = find(edge2newNode);
coordinates(edge2newNode(idx),:) ...
    = (coordinates(edge2nodes(idx,1),:)+coordinates(edge2nodes(idx,2),:))/2;
%*** Refine boundary conditions
for j = 1:nargout-2
    boundary = varargin{j};
    if ~isempty(boundary)
        newNodes = edge2newNode(boundary2edges{j});
        markedEdges = find(newNodes);
        if ~isempty(markedEdges)
            boundary = [boundary(~newNodes,:); ...
                        boundary(markedEdges,1),newNodes(markedEdges); ...
                        newNodes(markedEdges),boundary(markedEdges,2)];
        end
    end
    varargout{j} = boundary;
end
%*** Provide new nodes for refinement of elements
newNodes = edge2newNode(element2edges);
%*** Determine type of refinement for each element
markedEdges = (newNodes ~= 0);
none = ~markedEdges(:,1);
bisec1   = ( markedEdges(:,1) & ~markedEdges(:,2) & ~markedEdges(:,3) );
bisec12  = ( markedEdges(:,1) &  markedEdges(:,2) & ~markedEdges(:,3) );
bisec13  = ( markedEdges(:,1) & ~markedEdges(:,2) &  markedEdges(:,3) );
bisec123 = ( markedEdges(:,1) &  markedEdges(:,2) &  markedEdges(:,3) );
%*** Generate element numbering for refined mesh
idx = ones(nE,1);
idx(bisec1)   = 2; %*** bisec(1): newest vertex bisection of 1st edge
idx(bisec12)  = 3; %*** bisec(2): newest vertex bisection of 1st and 2nd edge
idx(bisec13)  = 3; %*** bisec(2): newest vertex bisection of 1st and 3rd edge
idx(bisec123) = 4; %*** bisec(3): newest vertex bisection of all edges
idx = [1;1+cumsum(idx)];
%*** Generate new elements
newElements = zeros(idx(end)-1,3);
newElements(idx(none),:) = elements(none,:);
newElements([idx(bisec1),1+idx(bisec1)],:) ...
    = [elements(bisec1,3),elements(bisec1,1),newNodes(bisec1,1); ...
       elements(bisec1,2),elements(bisec1,3),newNodes(bisec1,1)];
newElements([idx(bisec12),1+idx(bisec12),2+idx(bisec12)],:) ...
    = [elements(bisec12,3),elements(bisec12,1),newNodes(bisec12,1); ...
       newNodes(bisec12,1),elements(bisec12,2),newNodes(bisec12,2); ...
       elements(bisec12,3),newNodes(bisec12,1),newNodes(bisec12,2)];
newElements([idx(bisec13),1+idx(bisec13),2+idx(bisec13)],:) ...
    = [newNodes(bisec13,1),elements(bisec13,3),newNodes(bisec13,3); ...
       elements(bisec13,1),newNodes(bisec13,1),newNodes(bisec13,3); ...
       elements(bisec13,2),elements(bisec13,3),newNodes(bisec13,1)];
newElements([idx(bisec123),1+idx(bisec123),2+idx(bisec123),3+idx(bisec123)],:) ...
    = [newNodes(bisec123,1),elements(bisec123,3),newNodes(bisec123,3); ...
       elements(bisec123,1),newNodes(bisec123,1),newNodes(bisec123,3); ...
       newNodes(bisec123,1),elements(bisec123,2),newNodes(bisec123,2); ...
       elements(bisec123,3),newNodes(bisec123,1),newNodes(bisec123,2)];
```

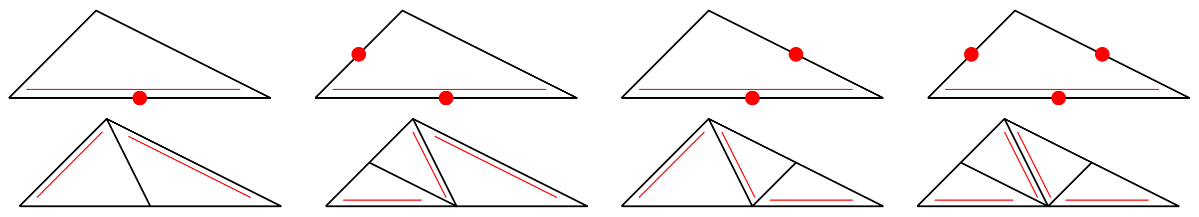


Figure 2. For each triangle $T \in \mathcal{T}$, there is one fixed *reference edge*, indicated by the double line (left, top). Refinement of T is done by bisecting the reference edge, where its midpoint becomes a new node. The reference edges of the son triangles are opposite to this newest vertex (left, bottom). To avoid hanging nodes, one proceeds as follows: We assume that certain edges of T, but at least the reference edge, are marked for refinement (top). Using iterated newest vertex bisection, the element is then split into 2, 3, or 4 son triangles (bottom).

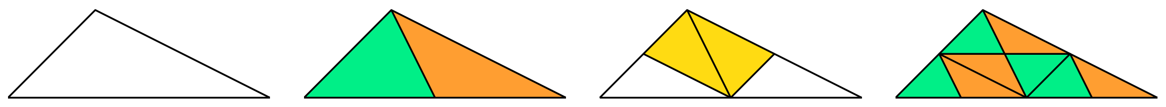


Figure 3. Refinement by newest vertex bisection only leads to finitely many interior angles for the family of all possible triangulations. To see this, we start from a macro element (left), where the bottom edge is the reference edge. Using iterated newest vertex bisection, one observes that only four similarity classes of triangles occur, which are indicated by the coloring. After three levels of bisection (right), no additional similarity class appears.

5.2. Refinement by newest vertex bisection (Listing 5)

Before discussing the implementation, we briefly describe the idea of NVB. To that end, let $\mathcal{T}_0$ be a given initial triangulation. For each triangle $T \in \mathcal{T}_0$ one chooses a so-called *reference edge*, e.g., the longest edge. For NVB, the (inductive) refinement rule reads as follows, where $\mathcal{T}_\ell$ is a regular triangulation already obtained from $\mathcal{T}_0$ by some successive newest vertex bisections:

- To refine an element $T \in \mathcal{T}_\ell$, the midpoint x_T of the reference edge E_T becomes a new node, and T is bisected along x_T and the node opposite to E_T into two son elements T_1 and T_2, cf. Figure 2.
- As is also shown in Figure 2, the edges opposite to the *newest vertex* x_T become the reference edges of the two son triangles T_1 and T_2.
- Having bisected all marked triangles, the resulting partition usually has hanging nodes. Therefore, certain additional bisections finally lead to a regular triangulation $\mathcal{T}_{\ell+1}$.

A moment's reflection shows that the latter closure step, which leads to a regular triangulation, only leads to finitely many additional bisections. An easy explanation might be the following, which is also illustrated in Figure 2:

- Instead of marked elements, one might think of marked edges.
- If any edge of a triangle T is marked for refinement, we ensure that its reference edge is also marked for refinement. This is done recursively in at most $3 \cdot \#\mathcal{T}_\ell$ recursions since then all edges are marked for refinement.

- If an element T is bisected, only the reference edge is halved, whereas the other two edges become the reference edges of the two son triangles. The refinement of T into 2, 3, or 4 sons can then be done in one step.

Listing 5 provides our implementation of the NVB algorithm, where we use the following convention: Let the element T_ℓ be stored by

$$\texttt{elements}(\ell,\texttt{:}) = [\ i\ \ j\ \ k\ ].$$

In this case $z_k \in \mathcal{N}$ is the newest vertex of T_ℓ, and the reference edge is given by $E = \text{conv}\{z_i, z_j\}$.

- Lines 9–10: Create a vector `edge2newNode`, where `edge2newNode(`ℓ`)` is nonzero if and only if the ℓ-th edge is refined by bisection. In Line 10, we mark all edges of the marked elements for refinement. Alternatively, one could only mark the reference edge for all marked elements. This is done by replacing Line 10 by

  ```
  edge2newNode(element2edges(markedElements,1)) = 1;
  ```

- Lines 11–16: Closure of edge marking: For NVB-refinement, we have to ensure that if an edge of $T \in \mathcal{T}$ is marked for refinement, at least the reference edge (opposite to the newest vertex) is also marked for refinement. Clearly, the loop terminates after at most $\#\mathcal{T}$ steps since then all reference edges have been marked for refinement.

- Lines 18–21: For each marked edge, its midpoint becomes a new node of the refined triangulation. The number of new nodes is determined by the nonzero entries of `edge2newNode`.

- Lines 23–35: Update boundary conditions for refined edges: The ℓ-th boundary edge is marked for refinement if and only if `newNodes(`ℓ`)` is nonzero. In this case, it contains the number of the edge's midpoint (Line 26). If at least one edge is marked for refinement, the corresponding boundary condition is updated (Lines 27–34).

- Lines 37–44: Mark elements for certain refinement by (iterated) NVB: Generate array such that `newNodes(i,`ℓ`)` is nonzero if and only if the ℓ-th edge of element T_i is marked for refinement. In this case, the entry returns the number of the edge's midpoint (Line 37). To speed up the code, we use logical indexing and compute a logical array `markedEdges` whose entry `markedEdges(i,`ℓ`)` only indicates whether the ℓ-th edge of element T_i is marked for refinement or not. The sets `none`, `bisec1`, `bisec12`, `bisec13`, and `bisec123` contain the indices of elements according to the respective refinement rule, e.g., `bisec12` contains all elements for which the first and the second edge are marked for refinement. Recall that either none or at least the first edge (reference edge) is marked.

- Lines 46–51: Generate numbering of elements for refined mesh: We aim to conserve the order of the elements in the sense that sons of a refined element have consecutive element numbers with respect to the refined mesh. The elements of `bisec1` are refined into two elements, the elements of `bisec12` and `bisec13` are refined into three elements, the elements of `bisec123` are refined into four elements.

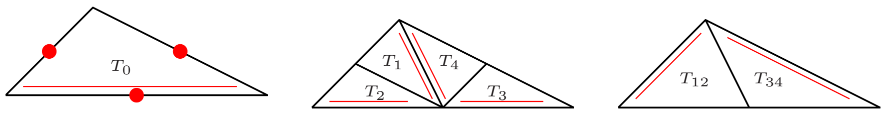


Figure 4. Coarsening is not fully inverse to refinement by newest vertex bisections: Assume that all edges of a triangle are marked for refinement (left). Refinement then leads to 4 son elements (middle). One application of the coarsening algorithm only removes the bisections on the last level (right).

- Lines 53–70: Generate refined mesh according to NVB: For all refinements, we respect a certain order of the sons of a refined element. Namely, if T is refined by NVB into two sons T_ℓ and $T_{\ell+1}$, T_ℓ is the left element with respect to the bisection procedure. This assumption allows the later coarsening of a refined mesh without storing any additional data, cf. [15] and Section 5.3 below.

In numerical analysis, constants usually depend on a lower bound of the smallest interior angle that appears in a sequence $\mathcal{T}_\ell$ of triangulations. It is thus worth noting that newest vertex bisection leads to at most $4 \cdot \#\mathcal{T}_0$ similarity classes of triangles [29] which only depend on $\mathcal{T}_0$, cf. Figure 3. In particular, there is a uniform lower bound for all interior angles in $\mathcal{T}_\ell$.

5.3. Coarsening of refined meshes (Listing 6)

Our Matlab function `coarsenNVB` is a vectorized implementation of an algorithm from [16]. However, our code generalizes the implementation of [16] in the sense that a subset of elements can be chosen for coarsening and redundant memory is set free, e.g., nodes which have been removed by coarsening. Moreover, our code respects the boundary conditions which are also affected by coarsening of $\mathcal{T}$.

We aim to coarsen $\mathcal{T}$ by removing certain nodes added by `refineNVB`: Let $T_1, T_2 \in \mathcal{T}$ be two brothers obtained by newest vertex bisection of a father triangle T_0, cf. Figure 2. Let $z \in \mathcal{N}$ denote the newest vertex of both T_1 and T_2. According to [16], one may coarsen T_1 and T_2 by removing the newest vertex z if and only if z is the newest vertex of all elements $T_3 \in \widetilde{\omega}_z := \{T \in \mathcal{T} \,:\, z \in T\}$ of the patch. Therefore, $z \in \mathcal{N} \backslash \mathcal{N}_0$ may be coarsened if and only if its valence satisfies $\#\widetilde{\omega}_z \in \{2, 4\}$, where $\mathcal{N}_0$ is the set of nodes for the initial mesh $\mathcal{T}_0$ from which the current mesh $\mathcal{T}$ is generated by finitely many (but arbitrary) newest vertex bisections. In case $\#\widetilde{\omega}_z = 2$, there holds $z \in \mathcal{N} \cap \Gamma$, whereas $\#\widetilde{\omega}_z = 4$ implies $z \in \mathcal{N} \cap \Omega$.

We stress that `coarsenNVB` only coarsens marked leaves of the current forest generated by newest vertex bisection, i.e., `coarsenNVB` is *not* inverse to `refineNVB`, cf. Figure 4. However, the benefit of this simple coarsening rule is that no additional data structure as, e.g., a refinement tree has to be built or stored.

- Lines 5–12: Build data structure `element2neighbours` containing geometric information on the neighbour relation: Namely, `k=element2neighbours(j,`ℓ`)` contains the number of the neighbouring element T_k along the ℓ-th edge of T_j, where $k = 0$ for a boundary edge.

- Lines 14–19: We mark nodes which are admissible for coarsening (Lines 17–19). However, we consider only newest vertices added by `refineNVB`, for which the corresponding elements are marked for coarsening (Lines 14–16).

Listing 6. Coarsening of NVB refined meshes

```
 1 function [coordinates,elements,varargout] = coarsenNVB(N0,coordinates,elements,varargin)
 2 nC = size(coordinates,1);
 3 nE = size(elements,1);
 4 %*** Obtain geometric information on neighbouring elements
 5 I = elements(:);
 6 J = reshape(elements(:,[2,3,1]),3*nE,1);
 7 nodes2edge = sparse(I,J,1:3*nE);
 8 mask = nodes2edge>0;
 9 [foo{1:2},idxIJ] = find( nodes2edge );
10 [foo{1:2},neighbourIJ] = find( mask + mask.*sparse(J,I,[1:nE,1:nE,1:nE]') );
11 element2neighbours(idxIJ) = neighbourIJ - 1;
12 element2neighbours = reshape(element2neighbours,nE,3);
13 %*** Determine which nodes (created by refineNVB) are deleted by coarsening
14 marked = zeros(nE,1);
15 marked(varargin{end}) = 1;
16 newestNode = unique(elements((marked & elements(:,3)>N0),3));
17 valence = accumarray(elements(:),1,[nC 1]);
18 markedNodes = zeros(nC,1);
19 markedNodes(newestNode((valence(newestNode) == 2 | valence(newestNode) == 4))) = 1;
20 %*** Collect pairs of brother elements that will be united
21 idx = find(markedNodes(elements(:,3)) & (element2neighbours(:,3) > (1:nE)'))';
22 markedElements = zeros(nE,1);
23 markedElements(idx) = 1;
24 for element = idx
25     if markedElements(element)
26         markedElements(element2neighbours(element,3)) = 0;
27     end
28 end
29 idx = find(markedElements);
30 %*** Coarsen two brother elements
31 brother = element2neighbours(idx,3);
32 elements(idx,[1 3 2]) = [elements(idx,[2 1]) elements(brother,1)];
33 %*** Delete redundant nodes
34 activeNodes = find(~markedNodes);
35 coordinates = coordinates(activeNodes,:);
36 %*** Provide permutation of nodes to correct further data
37 coordinates2newCoordinates = zeros(1,nC);
38 coordinates2newCoordinates(activeNodes) = 1:length(activeNodes);
39 %*** Delete redundant elements + correct elements
40 elements(brother,:) = [];
41 elements = coordinates2newCoordinates(elements);
42 %*** Delete redundant boundaries + correct boundaries
43 for j = 1:nargout-2;
44     boundary = varargin{j};
45     if ~isempty(boundary)
46       node2boundary = zeros(nC,2);
47       node2boundary(boundary(:,1),1) = 1:size(boundary,1);
48       node2boundary(boundary(:,2),2) = 1:size(boundary,1);
49       idx = ( markedNodes & node2boundary(:,2) );
50       boundary(node2boundary(idx,2),2) = boundary(node2boundary(idx,1),2);
51       boundary(node2boundary(idx,1),2) = 0;
52       varargout{j} = coordinates2newCoordinates(boundary(find(boundary(:,2)),:));
53     else
54       varargout{j} = [];
55     end
56 end
```

- Lines 21–29: Decide which brother elements $T_j, T_k \in \mathcal{T}$ are resolved into its father element: We determine which elements may be coarsened (Line 21) and mark them for coarsening (Lines 22–23). According to the refinement rules in `refineNVB`, the former father element T has been bisected into sons $T_j, T_k \in \mathcal{T}$ with $j < k$. By definition, T_j is the left brother with respect to the bisection of T, and the index k satisfies `k=element2neighbours(j,3)`. We aim to overwrite T_j with its father and to remove T_k from the list of elements later on. Therefore, we remove the mark on T_k (Lines 24–28) so that we end up with a list of left sons which are marked for coarsening (Line 29).

- Lines 31–32: We replace the left sons by its father elements.

- Lines 34–38: We remove the nodes that have been coarsened from the list of coordinates (Lines 34–35). This leads to a new numbering of the nodes so that we provide a mapping from the old indices to the new ones (Lines 37–38).

- Lines 40–41: We remove the right sons, which have been coarsened, from the list of elements (Line 40) and respect the new numbering of the nodes (Line 41).

- Lines 43–56: Correct the boundary partition: For each part of the boundary, e.g. the Dirichlet boundary Γ_D, we check whether some nodes have been removed by coarsening (Line 49). For these nodes, we replace the respective two boundary edges by the father edge. More precisely, let $z_j \in \mathcal{N} \cap \Gamma$ be removed by coarsening. We then overwrite the edge with z_j as second node by the father edge (Line 50) and remove the edge, where z_j has been the first node (Lines 51–52).

Listing 7. Adaptive algorithm

```
function [x,coordinates,elements,indicators] ...
    = adaptiveAlgorithm(coordinates,elements,dirichlet,neumann,f,g,uD,nEmax,theta)
while 1
    %*** Compute discrete solution
    x = solveLaplace(coordinates,elements,dirichlet,neumann,f,g,uD);
    %*** Compute refinement indicators
    indicators = computeEtaR(x,coordinates,elements,dirichlet,neumann,f,g);
    %*** Stopping criterion
    if size(elements,1) >= nEmax
        break
    end
    %*** Mark elements for refinement
    [indicators,idx] = sort(indicators,'descend');
    sumeta = cumsum(indicators);
    ell = find(sumeta>=sumeta(end)*theta,1);
    marked = idx(1:ell);
    %*** Refine mesh
    [coordinates,elements,dirichlet,neumann] = ...
       refineNVB(coordinates,elements,dirichlet,neumann,marked);
end
```

6. A posteriori error estimators and adaptive mesh-refinement

In practice, computational time and storage requirements are limiting quantities for numerical simulations. One is thus interested to construct a mesh $\mathcal{T}$ such that the number of elements $M = \#\mathcal{T} \leqslant M_{\max}$ stays below a given bound, whereby the error $\|u - U\|_{H^1(\Omega)}$ of the corresponding Galerkin solution U is (in some sense) minimal.

Such a mesh $\mathcal{T}$ is usually obtained in an iterative manner: For each element $T \in \mathcal{T}$, let $\eta_T \in \mathbb{R}$ be a so-called *refinement indicator* which (at least heuristically) satisfies

$$\eta_T \approx \|u - U\|_{H^1(T)} \quad \text{for all } T \in \mathcal{T}. \tag{11}$$

In particular, the associated *error estimator* $\eta = \big(\sum_{T\in\mathcal{T}} \eta_T^2\big)^{1/2}$ then yields an error estimate $\eta \approx \|u - U\|_{H^1(\Omega)}$. The main point at this stage is that the refinement indicators η_T might be computable, whereas u is unknown and thus the local error $\|u - U\|_{H^1(T)}$ is not.

6.1. Adaptive algorithm (Listing 7)

Given some refinement indicators $\eta_T \approx \|u-U\|_{H^1(T)}$, we mark elements $T \in \mathcal{T}$ for refinement by the Dörfler criterion [18], which seeks to determine the minimal set $\mathcal{M} \subseteq \mathcal{T}$ such that

$$\theta \sum_{T\in\mathcal{T}} \eta_T^2 \leqslant \sum_{T\in\mathcal{M}} \eta_T^2, \tag{12}$$

for some parameter $\theta \in (0,1)$. Then, a new mesh $\mathcal{T}'$ is generated from $\mathcal{T}$ by refinement of (at least) the marked elements $T \in \mathcal{M}$ to decrease the error $\|u-U\|_{H^1(\Omega)}$ efficiently. Note that $\theta \to 1$ corresponds to almost uniform mesh-refinement, i.e. most of the elements are marked for refinement, whereas $\theta \to 0$ leads to highly adapted meshes.

- Line 1–2: The function takes the initial mesh as well as the problem data f, g, and u_D. Moreover, the user provides the maximal number `nEmax` of elements and the adaptivity parameter θ from (12). After termination, the function returns the coefficient vector **x** of the final Galerkin solution $U \in \mathcal{S}^1_D(\mathcal{T})$, cf. (8), the associated final mesh $\mathcal{T}$, and the corresponding vector `indicators` of elementwise error indicators.
- Line 3–20: As long as the number $M = \#\mathcal{T}$ of elements is smaller than the given bound `nEmax`, we proceed as follows: We compute a discrete solution (Line 5) and the vector of refinement indicators (Line 7), whose j-th coefficient stores the value of $\eta_j^2 := \eta_{T_j}^2$. We find a permutation π of the elements such that the sequence of refinement indicators $(\eta^2_{\pi(j)})_{j=1}^M$ is decreasing (Line 13). Then (Line 14), we compute all sums $\sum_{j=1}^{\ell} \eta^2_{\pi(j)}$ and determine the minimal index ℓ such that $\theta \sum_{j=1}^M \eta_j^2 = \theta \sum_{j=1}^M \eta^2_{\pi(j)} \leqslant \sum_{j=1}^{\ell} \eta^2_{\pi(j)}$ (Line 15). Formally, we thus determine the set $\mathcal{M} = \{T_{\pi(j)} \,:\, j = 1,\dots,\ell\}$ of marked elements (Line 16), cf. (12). Finally (Lines 18–19), we refine the marked elements and so generate a new mesh.

In the current state of research, the Dörfler criterion (12) and NVB refinement are used to prove convergence and optimality of AFEM [11], and convergence for general mesh-refining strategies is observed in [3]. In [25], the authors, by others, prove convergence of AFEM for the bulk criterion, which marks elements $T \in \mathcal{T}$ for refinement provided

$$\eta_T \geqslant \theta \max_{T'\in\mathcal{T}} \eta_{T'}. \tag{13}$$

To use it in the adaptive algorithm, one may simply replace Lines 13–16 of Listing 7 by

```
marked = find(indicators>=theta*max(indicators));
```

For error estimation (Line 7) and mesh-refinement (Line 19), also other functions of `p1afem` can be used, cf. Section 4.

6.2. Residual-based error estimator (Listing 8)

We consider the error estimator $\eta_R := \big(\sum_{T\in\mathcal{T}} \eta_T^2\big)^{1/2}$ with refinement indicators

$$\eta_T^2 := h_T^2 \|f\|^2_{L^2(T)} + h_T \|J_h(\partial_n U)\|^2_{L^2(\partial T\cap\Omega)} + h_T \|g - \partial_n U\|^2_{L^2(\partial T\cap\Gamma_N)}. \tag{14}$$

Listing 8. Residual-based error estimator

```matlab
function etaR = computeEtaR(x,coordinates,elements,dirichlet,neumann,f,g)
[edge2nodes,element2edges,dirichlet2edges,neumann2edges] ...
     = provideGeometricData(elements,dirichlet,neumann);
%*** First vertex of elements and corresponding edge vectors
c1  = coordinates(elements(:,1),:);
d21 = coordinates(elements(:,2),:) - c1;
d31 = coordinates(elements(:,3),:) - c1;
%*** Vector of element volumes 2*|T|
area2 = d21(:,1).*d31(:,2)-d21(:,2).*d31(:,1);
%*** Compute curl(uh) = (-duh/dy, duh/dx)
u21 = repmat(x(elements(:,2))-x(elements(:,1)), 1,2);
u31 = repmat(x(elements(:,3))-x(elements(:,1)), 1,2);
curl = (d31.*u21 - d21.*u31)./repmat(area2,1,2);
%*** Compute edge terms hE*(duh/dn) for uh
dudn21 = sum(d21.*curl,2);
dudn13 = -sum(d31.*curl,2);
dudn32 = -(dudn13+dudn21);
etaR = accumarray(element2edges(:),[dudn21;dudn32;dudn13],[size(edge2nodes,1) 1]);
%*** Incorporate Neumann data
if ~isempty(neumann)
  cn1 = coordinates(neumann(:,1),:);
  cn2 = coordinates(neumann(:,2),:);
  gmE = feval(g,(cn1+cn2)/2);
  etaR(neumann2edges) = etaR(neumann2edges) - sqrt(sum((cn2-cn1).^2,2)).*gmE;
end
%*** Incorporate Dirichlet data
etaR(dirichlet2edges) = 0;
%*** Assemble edge contributions of indicators
etaR = sum(etaR(element2edges).^2,2);
%*** Add volume residual to indicators
fsT = feval(f,(c1+(d21+d31)/3));
etaR = etaR + (0.5*area2.*fsT).^2;
```

Here, $J_h(\cdot)$ denotes the jump over an interior edge $E \in \mathcal{E}$ with $E \not\subset \Gamma$. For neighbouring elements $T_\pm \in \mathcal{T}$ with respective outer normal vectors $n_\pm$, the jump of the $\mathcal{T}$-piecewise constant function ∇U over the common edge $E = T_+ \cap T_- \in \mathcal{E}$ is defined by

$$J_h(\partial_n U)|_E := \nabla U|_{T_+} \cdot n_+ + \nabla U|_{T_-} \cdot n_-, \tag{15}$$

which is, in fact, a difference since $n_+ = -n_-$. The residual-based error estimator η_R is known to be reliable and efficient in the sense that

$$C_{\rm rel}^{-1} \|u-U\|_{H^1(\Omega)} \leqslant \eta_R \leqslant C_{\rm eff}\big[\|u-U\|_{H^1(\Omega)} + \|h(f-f_\mathcal{T})\|_{L^2(\Omega)} + \|h^{1/2}(g-g_\mathcal{E})\|_{L^2(\Gamma_N)}\big], \tag{16}$$

where the constants $C_{\rm rel}, C_{\rm eff} > 0$ only depend on the shape of the elements in $\mathcal{T}$ as well as on Ω, see [30, Section 1.2]. Moreover, $f_\mathcal{T}$ and $g_\mathcal{E}$ denote the $\mathcal{T}$-elementwise and $\mathcal{E}$-edgewise integral mean of f and g, respectively. Note that for smooth data, there holds $\|h(f-f_\mathcal{T})\|_{L^2(\Omega)} = \mathcal{O}(h^2)$ as well as $\|h^{1/2}(g-g_\mathcal{E})\|_{L^2(\Gamma_N)} = \mathcal{O}(h^{3/2})$ so that these terms are of higher order when compared with error $\|u-U\|_{H^1(\Omega)}$ and error estimator η_R.

For the implementation, we replace $f|_T \approx f(s_T)$ and $g|_E \approx g(m_E)$ with s_T the center of mass of an element $T \in \mathcal{T}$ and m_E the midpoint an edge of $E \in \mathcal{E}$. We realize

$$\widetilde{\eta}_T^2 := |T|^2 f(s_T)^2 + \sum_{E \in \partial T \cap \Omega} h_E^2 \big(J_h(\partial_n U)|_E\big)^2 + \sum_{E \in \partial T \cap \Gamma_N} h_E^2 \big(g(m_E) - \partial_n U|_E\big)^2 \tag{17}$$

Note that shape regularity of the triangulation $\mathcal{T}$ implies

$$h_E \leqslant h_T \leqslant C\, h_E \quad \text{as well as} \quad 2|T| \leqslant h_T^2 \leqslant C\,|T|, \quad \text{for all } T \in \mathcal{T} \text{ with edge } E \subset \partial T, \tag{18}$$

with some constant $C > 0$, which depends only on a lower bound for the minimal interior angle. Up to some higher-order consistency errors, the estimators $\widetilde{\eta}_R$ and η_R are therefore equivalent.

The implementation from Listing 8 returns the vector of squared refinement indicators $(\widetilde{\eta}^2_{T_1}, \dots, \widetilde{\eta}^2_{T_M})$, where $\mathcal{T} = \{T_1, \dots, T_M\}$. The computation is performed in the following way:

- Lines 5–9 are discussed for Listing 3 above, see Section 3.4.
- Lines 11–13: Compute the $\mathcal{T}$-piecewise constant $(\mathrm{curl} U)|_T = (-\partial U/\partial x_2, \partial U/\partial x_1)|_T \in \mathbb{R}^2$ for all $T \in \mathcal{T}$ simultaneously. To that end, let z_1, z_2, z_3 be the vertices of a triangle $T \in \mathcal{T}$, given in counterclockwise order, and let V_j be the hat function associated with $z_j = (x_j, y_j) \in \mathbb{R}^2$. With $z_4 = z_1$ and $z_5 = z_2$, the gradient of V_j reads
$$\nabla V_j|_T = \frac{1}{2|T|}\,(y_{j+1} - y_{j+2}, x_{j+2} - x_{j+1}). \tag{19}$$
In particular, there holds $2|T|\,\mathrm{curl}V_j|_T = z_{j+1} - z_{j+2}$, where we assume $z_j \in \mathbb{R}^2$ to be a row-vector. With $U|_T = \sum_{j=1}^3 u_j V_j$, Lines 11–13 realize
$$2|T|\,\mathrm{curl}U|_T = 2|T|\sum_{j=1}^{3} u_j\,\mathrm{curl}V_j|_T = (z_3 - z_1)(u_2 - u_1) - (z_2 - z_1)(u_3 - u_1). \tag{20}$$
- Lines 15–18: For all edges $E \in \mathcal{E}$, we compute the jump term $h_E\,J_h(\partial U/\partial n)|_E$ if E is an interior edge, and $h_E\,(\partial U/\partial n)|_E$ if $E \subseteq \Gamma$ is a boundary edge, respectively. To that end, let z_1, z_2, z_3 denote the vertices of a triangle $T \in \mathcal{T}$ in counterclockwise order and identify $z_4 = z_1$ etc. Let n_j denote the outer normal vector of T on its j-th edge E_j of T. Then, $d_j = (z_{j+1} - z_j)/|z_{j+1} - z_j|$ is the tangential unit vector of E_j. By definition, there holds
$$h_{E_j}(\partial U/\partial n_{T,E_j}) = h_{E_j}(\nabla U \cdot n_{T,E_j}) = h_{E_j}(\mathrm{curl}U \cdot d_j) = \mathrm{curl}U \cdot (z_{j+1} - z_j).$$
Therefore, `dudn21` and `dudn13` are the vectors of the respective values for all first edges (between z_2 and z_1) and all third edges (between z_1 and z_3), respectively (Lines 15–16). The values for the second edges (between z_3 and z_2) are obtained from the equality
$$-(z_3 - z_2) = (z_2 - z_1) + (z_1 - z_3)$$
for the tangential directions (Line 17). We now sum the edge-terms of neighbouring elements, i.e. for $E = T_+ \cap T_- \in \mathcal{E}$ (Line 18). The vector `etaR` contains $h_E\,J_h(\partial U/\partial n)|_E$ for all interior edges $E \in \mathcal{E}$ as well as $h_E\,(\partial U/\partial n)|_E$ for boundary edges.
- Lines 20–29: For Neumann edges $E \in \mathcal{E}$, we subtract $h_E g(m_E)$ to the respective entry in `etaR` (Lines 20–25). For Dirichlet edges $E \in \mathcal{E}$, we set the respective entry of `etaR` to zero, since Dirichlet edges do not contribute to $\widetilde{\eta}_T$ (Line 27), cf. (17).
- Line 29: Assembly of edge contributions of $\widetilde{\eta}_T$. We simultaneously compute
$$\sum_{E\in\partial T\cap\Omega} h_E^2\,\big(J_h(\partial_n U)|_E\big)^2 + \sum_{E\in\partial T\cap\Gamma_N} h_E^2\,\big(g(m_E) - \partial_n U|_E\big)^2 \quad \text{for all } T \in \mathcal{T}.$$
- Line 31–32: We add the volume contribution $\big(|T|f(s_T)\big)^2$ and obtain $\widetilde{\eta}^2_T$ for all $T \in \mathcal{T}$.

7. Numerical experiments

To underline the efficiency of the developed MATLAB code, this section briefly comments on some numerical experiments. Throughout, we used the MATLAB version 7.8.0.347 (R2009a) on a common dual-board 64-bit PC with 3 GB of RAM and two AMD Athlon(tm) II X3 445 CPUs with 512 KB cache and 3.1 GHz each running under Linux.

7.1. Stationary model problem

For the first experiment, we consider the Poisson problem

$$-\Delta u = 1 \quad \text{in } \Omega \tag{21}$$

with mixed Dirichlet-Neumann boundary conditions, where the L-shaped domain as well as the boundary conditions are visualized in Figure 1. The exact solution has a singular behaviour at the re-entrant corner. Before the actual computations, the plotted triangulation given in Figure 1 is generally refined into a triangulation $\mathcal{T}_1$ with $M_1 = 3.072$ similar triangles. Throughout, the triangulation $\mathcal{T}_1$ is used as initial triangulation in our numerical computations.

We focus on the following aspects: First, we compare the computational times for six different implementations of the adaptive algorithm described in Section 6.1. Second, we plot the energy error over the total runtime for the adaptive algorithm and a uniform strategy. The latter experiment indicates the superiority of the adaptive strategy compared to uniform mesh refinements.

For all the numerical experiments, the computational time is measured by use of the built-in function `cputime` which returns the CPU time in seconds. Moreover, we take the mean of 11 iterations for the evaluation of these computational times, where the first execution is eliminated to neglect the time used by the MATLAB precompiler for performance acceleration.

In Figure 5, we plot the runtime of one call of the adaptive algorithm from Listing 7 over the number of elements for which the algorithm is executed. In this experiment, we use six different versions of the algorithm: the implementations "slow" (Listing 1), "medium" (Listing 2) and "optimized" (Listing 3) as well as the vectorized assembly from [26, 27] only differ in the method of assembly of the Galerkin data. All of these implementations use the routine `computeEtaR` from Section 6.2 to compute the error indicators and the function `refineNVB` from Section 5.2 for mesh refinement. For the remaining two implementations shown in Figure 3.4, we replace `solveLaplace`, `computeEtaR` and `refineNVB` by the corresponding functions from the AFEM package [16, 17] and the iFEM package [13, 14], respectively.

We compute the total runtime for the adaptive algorithm from $M = 3.072$ up to $M = 2.811.808$ elements. As can be expected from Section 3.2–3.3, the naive assembly (slow) of the Galerkin data from [1] yields quadratic dependence, whereas all remaining codes are empirically proven to be of (almost) linear complexity. The third implementation, which uses all the optimized modules, is approximately 6 times faster than AFEM [16, 17]. At the same time, iFEM [16, 17] and [27] are approximately 40% resp. 20% slower than our implementation. Note that for the maximal number of elements $M = 2.811.808$, one call of the optimized adaptive algorithm takes approximately 17 seconds.

Further numerical experiments show that the MATLAB \ operator yields the highest non-linear contribution to the overall runtime. Consequently, the runtime of the optimized

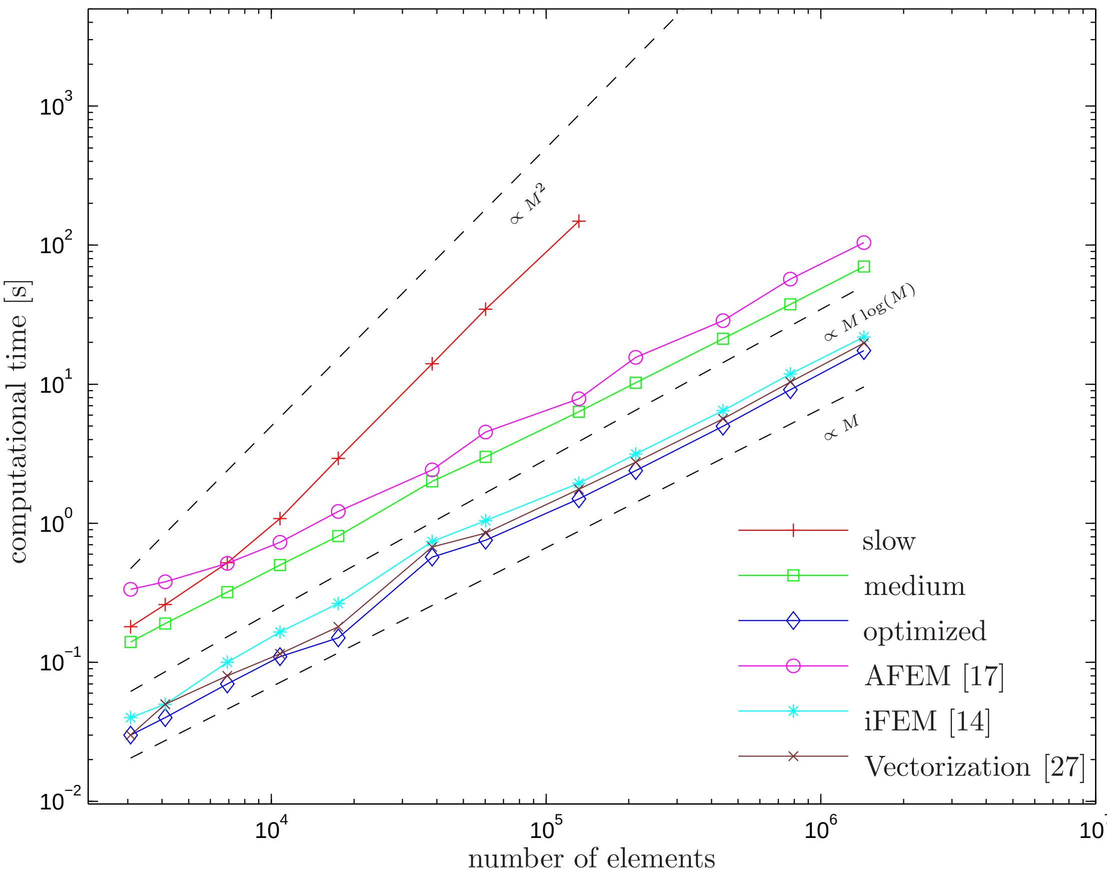


Figure 5. Computational times for various implementations of the adaptive algorithm from Listing 6.1 for Example 7.1 over the number of elements M. We stress that the version which uses the assembly from Listing 1 (slow) has quadratic growth, while all the other implementations only lead to (almost) linear growth of the computational time. In all cases, the algorithm using the optimized routines is the fastest.

adaptive algorithm is dominated by solving the sparse system of equations for more than 200.000 elements. For the final run with $M = 2.811.808$, approximately half of the total runtime is contributed by MATLAB's backslash operator. The assembly approximately takes 20% of the overall time, whereas the contribution of `computeEtaR` and `refineNVB` both account for 15% of the computation. For a detailed discussion of the numerical results, we refer to [20].

In the second numerical experiment for the static model problem, we compare uniform and adaptive mesh-refinement. In Figure 7, we plot the error in the energy norm $\|\nabla(u - U)\|_{L^2(\Omega)}$ and the residual error estimator η_R over the computational time for the adaptive algorithm from Section 6.1 and a uniform strategy. The error is computed with the help of the Galerkin orthogonality which provides

$$\|\nabla(u-U)\|_{L^2(\Omega)} = \big(\|\nabla u\|^2_{L^2(\Omega)} - \|\nabla U\|^2_{L^2(\Omega)}\big)^{1/2}. \tag{22}$$

Let $\mathcal{T}$ be a given triangulation with associated Galerkin solution $U \in \mathcal{S}^1(\mathcal{T})$. If $\mathbf{A}$ denotes the Galerkin matrix and $\mathbf{x}$ denotes the coefficient vector of U, the discrete energy reads

$$\|\nabla U\|^2_{L^2(\Omega)} = \mathbf{x}\cdot\mathbf{A}\mathbf{x}. \tag{23}$$

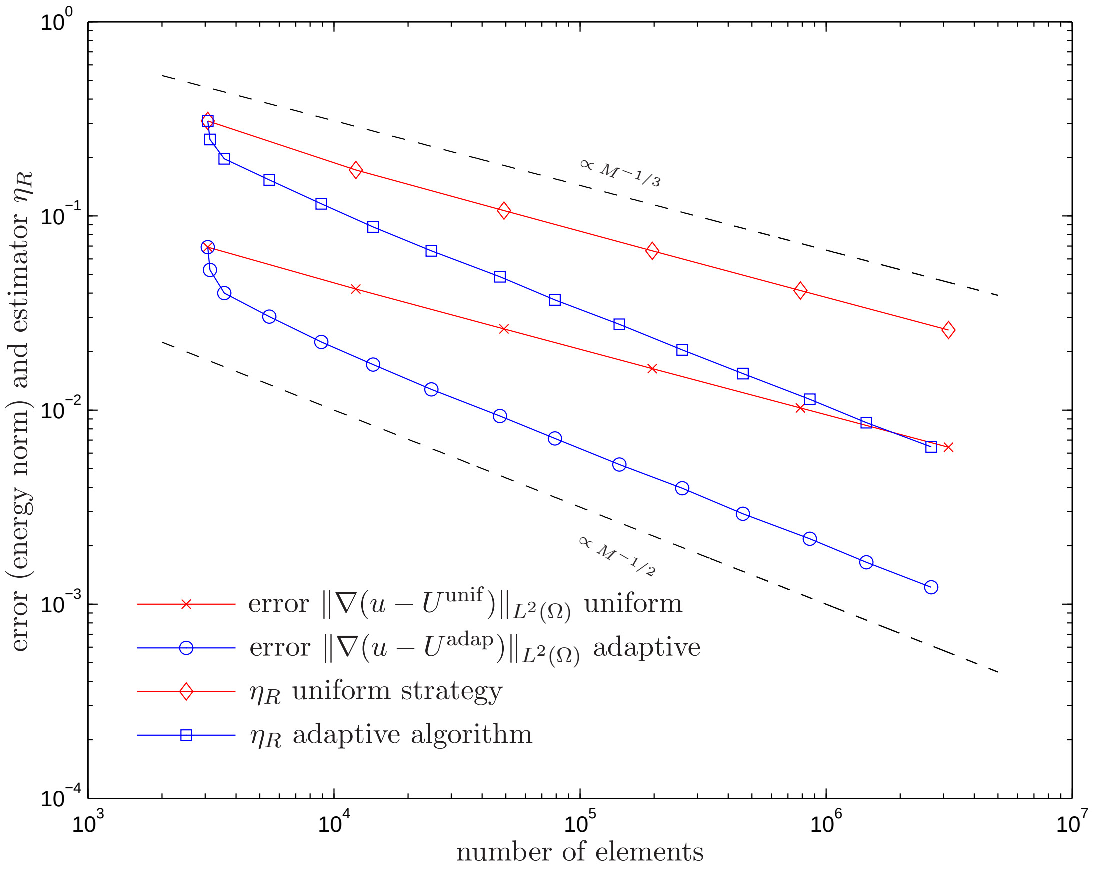


Figure 6. Galerkin error and error estimator η_R in Example 7.1 with respect to the number of elements: We consider uniform mesh-refinement as well as the adaptive algorithm from Listing 7. One observes that adaptive mesh-refinement is clearly superior and that the optimal convergence order is retained.

Since the exact solution $u \in H^1(\Omega)$ is not given analytically, we used Aitkin's Δ^2 method to extrapolate the discrete energies obtained from a sequence of uniformly refined meshes with $M = 3.072$ to $M = 3.145.728$ elements. This led to the extrapolated value

$$\|\nabla u\|^2_{L^2(\Omega)} \approx 1.06422 \tag{24}$$

which is used to compute the error (22) for uniform as well as for adaptive mesh-refinement. In the adaptive process the elements were marked by use of the Dörfler marking (12) with $\theta = 0.5$ and refined by the newest vertex bisection (NVB).

For a fair comparison with the adaptive strategy, the plotted times are computed as follows: For the ℓ-th entry in the plot, the computational time t_ℓ^{unif} corresponding to uniform refinement is the sum of

- the time for $\ell - 1$ successive uniform refinements,
- the time for one assembly and solution of the Galerkin system,

where we always start with the initial mesh $\mathcal{T}_1$ with $M_1 = 3.072$ elements and t_1^{unif} is the time for the assembly and solving for $\mathcal{T}_1$. Contrary to that, the adaptive algorithm from Listing 7 with $\theta = 0.5$ constructs a sequence of successively refined meshes, where $\mathcal{T}_{\ell+1}$ is

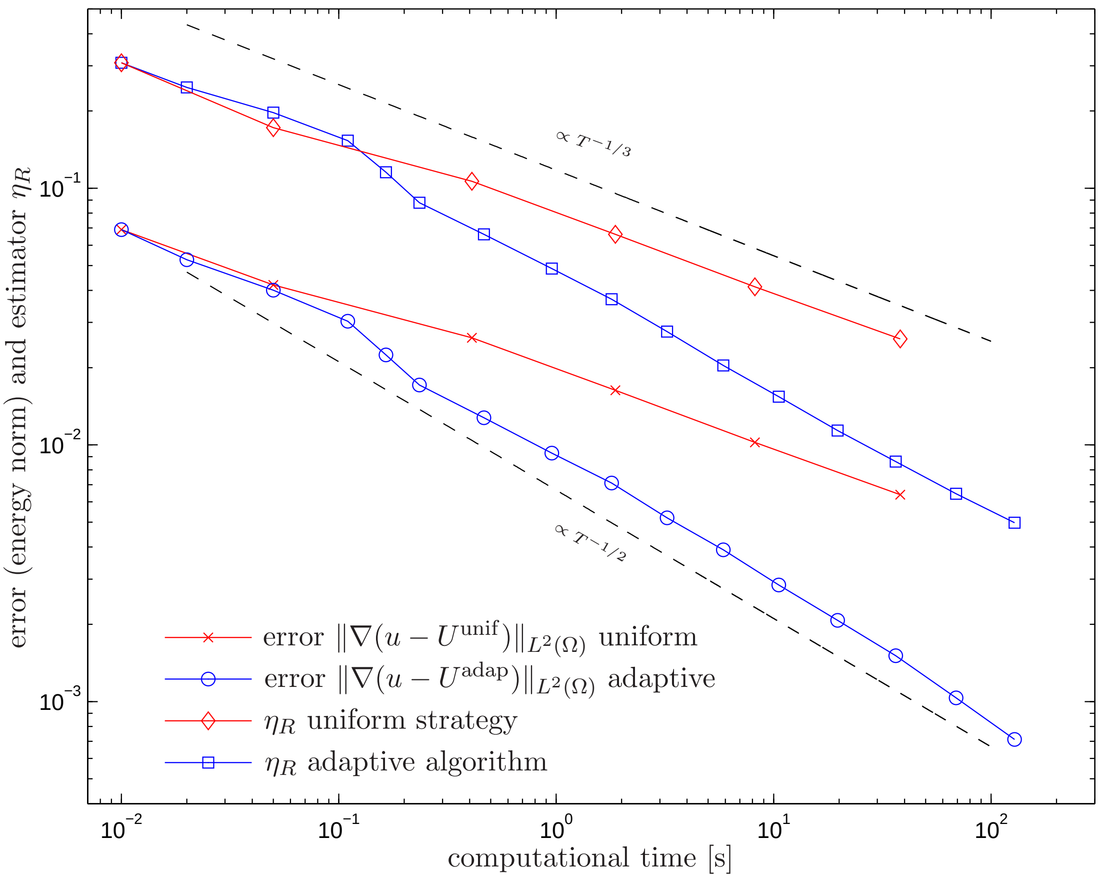


Figure 7. Galerkin error and error estimator η_R in Example 7.1 with respect to computational time: We consider uniform mesh-refinement as well as the adaptive algorithm from Listing 7. For the uniform strategy, we only measure the computational time for ℓ successive uniform mesh-refinements plus one assembly and the solution of the Galerkin system. For the adaptive strategy, we measure the time for the assembly and solution of the Galerkin system, the time for the computation of the residual-based error estimator η_R and the refinement of the marked elements, and we add the time used for the adaptive history. In any case, one observes that the adaptive strategy is much superior to uniform mesh-refinement.

obtained by local refinement of $\mathcal{T}_\ell$ based on the discrete solution U_ℓ. We therefore define the computational time t_ℓ^{adap} for adaptive mesh-refinement in a different way: We again set t_1^{adap} to the time for one call of `solveLaplace` on the initial mesh $\mathcal{T}_1$. The other computational times t_ℓ^{adap} are the sum of

- the time $t_{\ell-1}^{\text{adap}}$ already used in prior steps,
- the time for the computation of the residual-based error estimator η_R,
- the time for the refinement of the marked elements to provide $\mathcal{T}_\ell$,
- the time for the assembly and solution of the Galerkin system for $\mathcal{T}_\ell$.

Within 100 seconds, our MATLAB code computes an approximation with accuracy $\|\nabla(u - U^{\text{adap}})\|_{L^2(\Omega)} \approx 1/1000$, whereas uniform refinement only leads to $\|\nabla(u-U^{\text{unif}})\|_{L^2(\Omega)} \approx 1/100$ within roughly the same time. This shows that not only from a mathematical, but even from a practical point of view, adaptive algorithms are much superior.

```
Set $n := 0$, $t := 0$                                         Initialization
Do while $t \leqslant t_{max}$                                   Time loop
    Set $k := 0$ and $\mathcal{T}_{n,1} := \mathcal{T}_n$                          Initial mesh for refinement loop
    Do                                                          Loop for adaptive mesh-refinement
        Update $k \mapsto k+1$
        Compute discrete solution $U_{n,k}$ on mesh $\mathcal{T}_{n,k}$
        For all $T \in \mathcal{T}_{n,k}$ compute error indicators $\eta_T$      Refinement indicator
          and estimator $\eta_{n,k}^2 := \sum_{T \in \mathcal{T}_{n,k}} \eta_T^2$          Error estimator
        If $\eta_{n,k} > \tau$                                     Adaptive mesh-refinement
            Use Dörfler criterion (12) to mark
              elements for refinement
            Refine marked elements by NVB to
              obtain a 'finer' triangulation $\mathcal{T}_{n,k+1}$
        End If
    While $\eta_{n,k} > \tau$                                      Solution $U_{n,k}$ is accurate enough
    Set $\mathcal{T}_n := \mathcal{T}_{n,k}$ and $U_n := U_{n,k}$
    Set $\mathcal{T}_{n,k}^* := \mathcal{T}_n$                                   Initial mesh for coarsening loop
    Do                                                          Loop for adaptive mesh-coarsening
        Update $k \mapsto k-1$
        Mark elements $T$ for coarsening
          provided $\eta_T^2 \leqslant \sigma\,\tau^2/\#\mathcal{T}_{n,k+1}^*$
        Generate a 'coarser' triangulation $\mathcal{T}_{n,k}^*$
          by coarsening marked elements
        If $\#\mathcal{T}_{n,k}^* < \#\mathcal{T}_{n,k+1}^*$
            Compute discrete solution $U_{n,k}$ on mesh $\mathcal{T}_{n,k}^*$
            For all $T \in \mathcal{T}_{n,k}^*$ compute error indicators $\eta_T$  Refinement (resp. coarsening) indicator
        End if
    While $k \geqslant 1$ and $\#\mathcal{T}_{n,k}^* < \#\mathcal{T}_{n,k+1}^*$             Mesh cannot be coarsened furthermore
    Set $\mathcal{T}_n^* := \mathcal{T}_{n,k}^*$
    Set $\mathcal{T}_{n+1} := \mathcal{T}_n^*$
    Update $n := n+1$, $t := t + \Delta t$                          Go to next time step
End Do
```

Table 1. Adaptive algorithm with refinement and coarsening used for the quasi-stationary Example 7.2.

7.2. Quasi-stationary model problem

In the second example, we consider a homogeneous Dirichlet problem (1) with $\Gamma_D = \partial\Omega$ on the domain $\Omega = (0,3)^2 \setminus [1,2]^2$, cf. Figure 8. The right-hand side $f(x,t) := \exp(-10\,\|x - x_0(t)\|^2)$ is time dependent with $x_0(t) := (1.5 + \cos t, 1.5 + \sin t)$. The initial mesh $\mathcal{T}_0$ consists of 32 elements obtained from refinement of the 8 squares along their diagonals.

In the following, we compute for $n = 0, 1, 2, \dots, 200$ and corresponding time steps $t_n := n\pi/100 \in [0, 2\pi]$ a discrete solution U_n such that the residual-based error estimator $\eta_R = \eta_R(U_n)$ from Section 6.2 satisfies $\eta_R \leqslant \tau$ for a given tolerance $\tau > 0$. Instead of starting always from the coarsest mesh $\mathcal{T}_0$, we use the adaptive algorithm from Table 1 which allows adaptive mesh-refinement as well as mesh-coarsening. For the refinement, we use the Dörfler criterion (12) with parameter $\theta \in (0,1)$. For the coarsening process, we mark those elements $T \in \mathcal{T}$, which satisfy $\eta_T^2 \leqslant \sigma\,\tau^2/\#\mathcal{T}$ for some given parameter $\sigma \in (0,1)$. This criterion is

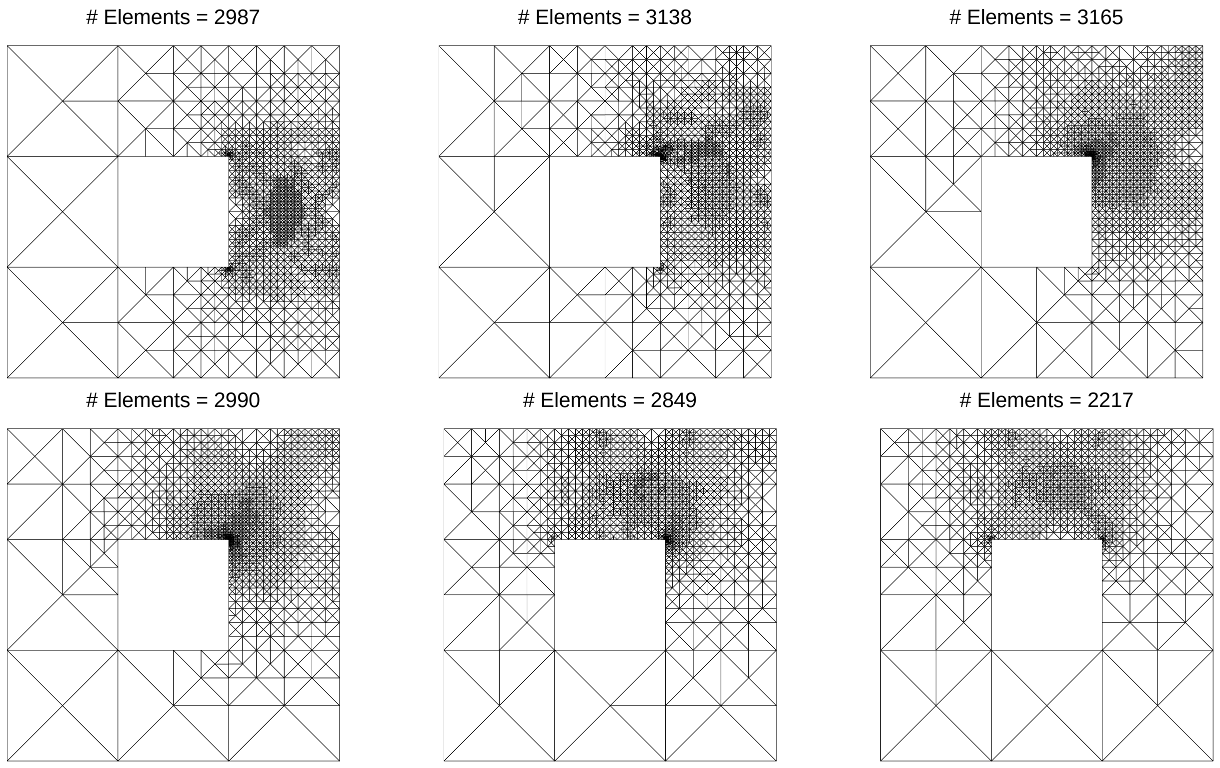


Figure 8. Adaptively generated meshes $\mathcal{T}_n$ at different time steps $n = 1, 11, 21, 31, 41, 51$ for the quasi-stationary Example 7.2, where we used the adaptive algorithm from Table 1 with tolerance $\tau = 0.03$ and parameters $\sigma = 0.25$ and $\theta = 0.25$.

heuristically motivated as follows: For an optimal mesh $\mathcal{T}$ with $\eta_R = \tau$, we expect an equi-distribution of the residual, i.e. $\eta_T = \eta_0$ for all $T \in \mathcal{T}$. Then, $\tau^2 = \eta_R^2 = \sum_{T\in\mathcal{T}} \eta_T^2 = (\#\mathcal{T})\eta_0^2$, whence $\eta_0^2 = \tau^2/\#\mathcal{T}$. Consequently, our criterion marks those elements T for which the contribution η_T seems to be too small with respect to the equi-distribution of the residual. We stop our coarsening process if none of the marked elements can be modified by our procedure described in Section 5.3.

For our numerical experiment, we choose the tolerance $\tau = 0.03$ as well as the parameters $\sigma = 0.25$ for adaptive mesh-coarsening and $\theta = 0.25$ for adaptive mesh-refinement. A sequence of adapted meshes is shown in Figure 8 at times $t_1 = 0$, t_{11}, t_{21}, t_{31}, t_{41}, and t_{51}. We see that the refinement follows mainly the source term f. Moreover, we observe a certain refinement at reentrant corners, and elements 'behind' the source are coarsened.

In Figure 9 we plot the evolution of the number of elements. The upper curve shows the number $\#\mathcal{T}_n$ of elements to satisfy the condition $\eta_R \leqslant \tau$ for each time step, while the lower graph gives the number $\#\mathcal{T}_n^*$ of elements after coarsening of the fine triangulation. Both curves show oscillations. This is in agreement with the theory due to the character of the source term f, since more degrees of freedom are needed for the same accuracy when the source density increases at one of the reentrant corners.

Finally, we compare the performance of the proposed adaptive algorithm with two naive strategies which avoid coarsening, but only use adaptive mesh-refinement. First, we start the adaptive computation in each time step t_n with the initial mesh $\mathcal{T}_0$, i.e., $\mathcal{T}_n := \mathcal{T}_0$ for all $n = 0, \dots, 200$. Second, we always use the preceding mesh to start the adaptive computation, i.e., $\mathcal{T}_{n+1} := \mathcal{T}_n$. Note that in the second case, the number $\#\mathcal{T}_n$ of elements is always increasing with n. However, up to the time step $n = 45$, the latter strategy is

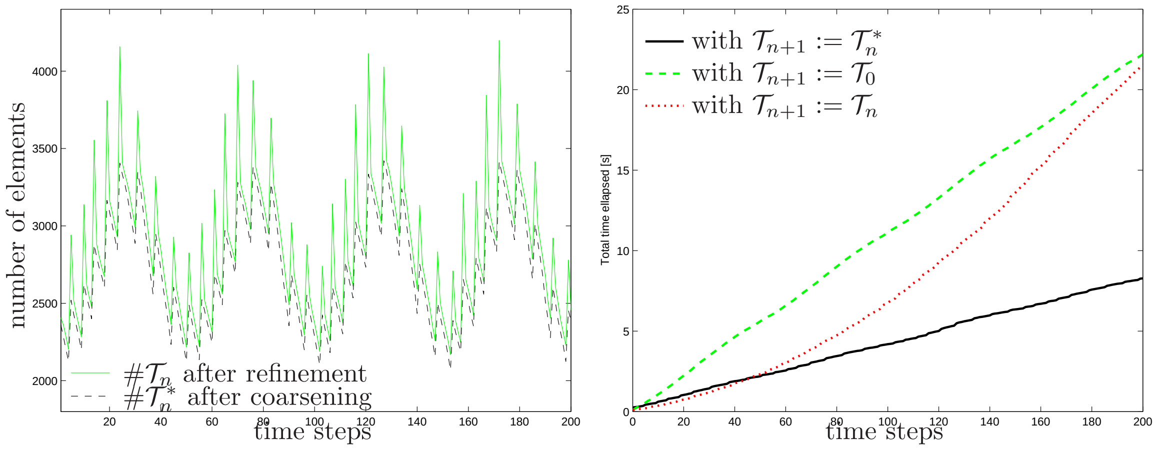


Figure 9. On the left, number $\#\mathcal{T}_n$ of elements in quasi-stationary Example 7.2 after refinement (left) resp. number $\#\mathcal{T}_n^*$ of elements after coarsening (left, dashed) for all time steps $n = 1, 2, 3, \ldots, 200$, where we used the adaptive algorithm from Table 1 with tolerance $\tau = 0.03$ and parameters $\sigma = 0.25$ and $\theta = 0.25$. On the right, we compare different strategies with respect to the overall computational time. First, the proposed algorithm from Table 1, where $\mathcal{T}_{n+1} := \mathcal{T}_n^*$ (right). Second, the same algorithm without coarsening and computation from the scratch, i.e., $\mathcal{T}_{n+1} := \mathcal{T}_0$ (right, dashed). Third, the same algorithm without coarsening and $\mathcal{T}_{n+1} := \mathcal{T}_n$ (right, dotted). For later time steps t_n, we observe that the algorithm from Table 1 is much more efficient than the other two naive strategies.

the most efficient with respect to the overall computational time. For $n \geqslant 45$, the proposed adaptive algorithm from Table 1 becomes the most efficient strategy. Until the final time step $n = 200$, the other two naive strategies become three times slower, and this gap is even increasing with n. Hence, a refinement-coarsening strategy as considered here, is generically much faster than naive approaches.

Acknowledgement. The second author (DP) is funded by the Austrian Science Fund (FWF) under grant P21732. Parts of this paper have been written during a research stay of DP at the Hausdorff Institute for Mathematics in Bonn, which is thankfully acknowledged. The third author (PW) is funded by the Austrian Science Fund (FWF) under grant W004-N13.

References

[1] J. Alberty, C. Carstensen, and S. A. Funken. Remarks around 50 lines of Matlab: short finite element implementation. *Numer. Algorithms*, 20(2-3):117–137, 1999.

[2] J. Alberty, C. Carstensen, S. A. Funken, and R. Klose. Matlab implementation of the finite element method in elasticity. *Computing*, 69(3):239–263, 2002.

[3] M. Aurada, S. Ferraz-Leite, and D. Praetorius. Estimator reduction and convergence of adaptive FEM and BEM. *ASC Report*, 27/2009, Vienna University of Technology, 2009.

[4] I. Babuška and A. Miller. A feedback finite element method with a posteriori error estimation. I. The finite element method and some basic properties of the a posteriori error estimator. *Comput. Methods Appl. Mech. Engrg.*, 61(1):1–40, 1987.

[5] C. Bahriawati and C. Carstensen. Three MATLAB implementations of the lowest-order Raviart-Thomas MFEM with a posteriori error control. *Comput. Methods Appl. Math.*, 5(4):333–361 (electronic), 2005.

[6] R. E. Bank and R. K. Smith. A posteriori error estimates based on hierarchical bases. *SIAM J. Numer. Anal.*, 30(4):921–935, 1993.

[7] R. Barrett, M. Berry, T. F. Chan, and et al. *Templates for the solution of linear systems: building blocks for iterative methods*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1994.

[8] S. Bartels, C. Carstensen, and A. Hecht. P2Q2Iso2D = 2D isoparametric FEM in Matlab. *J. Comput. Appl. Math.*, 192(2):219–250, 2006.

[9] C. Carstensen. An adaptive mesh-refining algorithm allowing for an H^1 stable L^2 projection onto Courant finite element spaces. *Constr. Approx.*, 20(4):549–564, 2004.

[10] C. Carstensen and R. Klose. Elastoviscoplastic finite element analysis in 100 lines of Matlab. *J. Numer. Math.*, 10(3):157–192, 2002.

[11] J. M. Cascon, C. Kreuzer, R. H. Nochetto, and K. G. Siebert. Quasi-optimal convergence rate for an adaptive finite element method. *SIAM J. Numer. Anal.*, 46(5):2524–2550, 2008.

[12] L. Chen. Short bisection implementation in Matlab. *Research Notes*, University of Maryland, 2006.

[13] L. Chen. iFEM: An innovative finite element method package in MATLAB. *Preprint*, University of Maryland, 2008.

[14] L. Chen. ifem: An innovative finite element method package in MATLAB (software download). `http://ifem.wordpress.com`, Version 5 Apr 2009.

[15] L. Chen and C. Zhang. A coarsening algorithm on adaptive grids by newest vertex bisection and its applications. *J. Comput. Math.*, 28(6):767–789, 2010.

[16] L. Chen and C. Zhang. AFEM@Matlab: A Matlab package of adaptive finite element methods. *Research Notes*, University of Maryland, 2006.

[17] L. Chen and C. Zhang. AFEM@Matlab: A Matlab package of adaptive finite element methods. `http://www.mathworks.com/matlabcentral/fileexchange/12679`, Version 17 Nov 2006.

[18] W. Dörfler. A convergent adaptive algorithm for Poisson's equation. *SIAM J. Numer. Anal.*, 33(3):1106–1124, 1996.

[19] S. Funken, D. Praetorius, and P. Wissgott. Efficient implementation of adaptive P1-FEM in Matlab (software download). `http://www.asc.tuwien.ac.at/∼dirk/matlab`.

[20] S. Funken, D. Praetorius, and P. Wissgott. Efficient implementation of adaptive P1-FEM in Matlab (extended preprint). *ASC Report*, 19/2008, Vienna University of Technology, 2008.

[21] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in MATLAB: design and implementation. *SIAM J. Matrix Anal. Appl.*, 13(1):333–356, 1992.

[22] M. S. Gockenbach. *Understanding and implementing the finite element method*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2006.

[23] J. Li and Y.-T. Chen. *Computational partial differential equations using MATLAB®*. Chapman & Hall/CRC Applied Mathematics and Nonlinear Science Series. CRC Press, Boca Raton, FL, 2009. With 1 CD-ROM (Windows, Macintosh and UNIX).

[24] P. Morin, R. H. Nochetto, and K. G. Siebert. Data oscillation and convergence of adaptive FEM. *SIAM J. Numer. Anal.*, 38(2):466–488 (electronic), 2000.

[25] P. Morin, K. G. Siebert, and A. Veeser. A basic convergence result for conforming adaptive finite elements. *Math. Models Methods Appl. Sci.*, 18(5):707–737, 2008.

[26] T. Rahman and J. Valdman. Fast MATLAB assembly of FEM matrices in 2D and 3D: nodal elements. *Technical report*, 11/2011, MPI for Mathematics in the Sciences, Leipzig, 2011.

[27] T. Rahman and J. Valdman. Fast MATLAB assembly of FEM matrices in 2D and 3D: nodal elements. `http://www.mathworks.com/matlabcentral/fileexchange/authors/37756`, Version 02 Mar 2011.

[28] A. Schmidt and K. G. Siebert. *Design of adaptive finite element software*, volume 42 of *Lecture Notes in Computational Science and Engineering*. Springer-Verlag, Berlin, 2005. The finite element toolbox ALBERTA, With 1 CD-ROM (Unix/Linux).

[29] E. Sewell. *Automatic Generation of Triangulations for Piecewise Polynomial Approximations*. PhD thesis, Purdue University, West Lafayette, 1972.

[30] R. Verfürth. *A Review of A Posteriori Error Estimation and Adaptive Mesh-Refinement Techniques*. B.G. Teubner, Stuttgart, 1996.

[31] O. C. Zienkiewicz and J. Z. Zhu. A simple error estimator and adaptive procedure for practical engineering analysis. *Internat. J. Numer. Methods Engrg.*, 24(2):337–357, 1987.