ASC Report No. 19/2008

Efficient Implementation of Adaptive P1-FEM in MATLAB

Stefan Funken, Dirk Praetorius, Philipp Wissgott

Institute for Analysis and Scientific Computing
Vienna University of Technology — TU Wien
www.asc.tuwien.ac.at ISBN 978-3-902627-00-1

Institute for Analysis and Scientific Computing
Vienna University of Technology
Wiedner Hauptstraße 8–10
1040 Wien, Austria

E-Mail: admin@asc.tuwien.ac.at
WWW: http://www.asc.tuwien.ac.at
FAX: +43-1-58801-10196

ISBN 978-3-902627-00-1

EFFICIENT IMPLEMENTATION OF ADAPTIVE P1-FEM IN MATLAB

S. FUNKEN, D. PRAETORIUS, AND P. WISSGOTT

Abstract. We provide a Matlab implementation of 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.

1. Introduction

In recent years, Matlab has become a de facto standard for the development of various kinds of algorithms for numerical simulations.

In [2], a short Matlab code for the P1-Galerkin FEM is proposed. 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.

In this paper, we thus show how to modify the existing Matlab code so that the theoretically predicted complexity can even be measured in computations. Our code is fully vectorized in the sense that `for`-loops are eliminated by use of Matlab vector operations. Moreover and in addition to [2], we provide a complete and easy-to-modify package for adaptive P1-FEM computations, including 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 [10, 12]. All parts of this package [15] are implemented in a way, we expect to be optimal in Matlab as a compromise between clarity, shortness, and use of Matlab built-in functions.

The remaining content is organized as follows: Section 2 introduces the model problem and the Galerkin scheme. In Section 3, we first recall the data structures of [2] 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 code. Section 4 is focused on local mesh-refinement and mesh-coarsening. Section 5 provides a realization of a standard adaptive mesh-refining algorithm. For marking, we provide the Matlab implementations of three types of error estimators: First, the residual-based error estimator introduced by Babuška and Miller [4], second, the hierarchical error estimator due to Bank and Smith [5], third, the gradient recovery technique proposed by Zienkiewicz and Zhu [23]. Section 6 concludes the paper with some numerical experiments.

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 $f \in L^2(\Omega)$, $u_D \in H^1(\Omega)$, and $g \in L^2(\Gamma_N)$, we

Date: August 4, 2008.

aim to compute an approximation of the solution $u \in H^1(\Omega)$ of

$$\begin{aligned} -\Delta u &= f \qquad \text{in } \Omega, \\ u &= u_D \qquad \text{on } \Gamma_D, \\ \partial_n u &= g \qquad \text{on } \Gamma_N. \end{aligned} \tag{2.1}$$

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. With

$$u_0 = u - u_D \in H^1_D(\Omega) := \{v \in H^1(\Omega) \,:\, v = 0 \text{ on } \Gamma_D\}, \tag{2.2}$$

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

$$\int_\Omega \nabla u_0 \cdot \nabla v \, dx = \int_\Omega f v \, dx + \int_{\Gamma_N} g v \, ds - \int_\Omega \nabla u_D \cdot \nabla v \, dx \quad \text{for all } v \in H^1_D(\Omega). \tag{2.3}$$

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 (2.1). 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 $T = \text{conv}\{z_1, z_2, z_3\}$ 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 $\mathcal{T}$. Moreover, hanging nodes are not allowed, cf. Figure 1 for an exemplary regular triangulation $\mathcal{T}$. Let

$$\mathcal{S}^1(\mathcal{T}) := \{V \in C(\Omega) \,:\, \forall T \in \mathcal{T} \quad V|_T \text{ affine}\} \tag{2.4}$$

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

$$\mathcal{S}^1_D(\mathcal{T}) := \mathcal{S}^1(\mathcal{T}) \cap H^1_D(\Omega) = \{V \in \mathcal{S}^1(\mathcal{T}) \,:\, \forall z_\ell \in \mathcal{N} \cap \Gamma_D \quad V(z_\ell) = 0\}. \tag{2.5}$$

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

$$U_D := \sum_{\ell=n+1}^{N} u_D(z_\ell) V_\ell \in \mathcal{S}^1(\mathcal{T}). \tag{2.6}$$

The discrete variational form

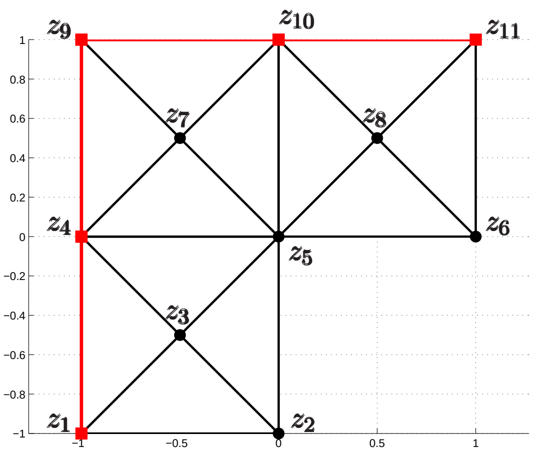
$$\int_\Omega \nabla U_0 \cdot \nabla V \, dx = \int_\Omega f V \, dx + \int_{\Gamma_N} g V \, ds - \int_\Omega \nabla U_D \cdot \nabla V \, dx \quad \text{for all } V \in \mathcal{S}^1_D(\mathcal{T}) \tag{2.7}$$

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 $\mathcal{B}$

$$U_0 = \sum_{j=1}^{n} \mathbf{x}_j V_j, \quad \text{whence} \quad U = \sum_{j=1}^{N} \mathbf{x}_j V_j \quad \text{with } \mathbf{x}_j := u_D(z_j) \text{ for } j = n+1, \dots, N. \tag{2.8}$$

Note that the discrete variational form (2.7) is equivalent to the linear system

$$\sum_{k=1}^{n} \mathbf{A}_{jk} \mathbf{x}_k = \mathbf{b}_j := \int_\Omega f V_j \, dx + \int_{\Gamma_N} g V_j \, ds - \sum_{k=n+1}^{N} \mathbf{A}_{jk} \mathbf{x}_k \quad \text{for all } j = 1, \dots, n \tag{2.9}$$



	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	11	10
2	10	9
3	9	4
4	4	1

	neumann	
1	1	2
2	2	5
3	5	6
4	6	11

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 11×2 array `coordinates` and the 12×3 array `elements`. The Dirichlet boundary, specified in the 4×2 array `dirichlet`, consists of 4 edges which are plotted in red. The nodes $\mathcal{N} \cap \Gamma_D$ are indicated by red squares, whereas free nodes are indicated by black bullets. The Neumann boundary is specified by `neumann` and consists of the remaining 4 boundary edges.

with stiffness matrix entries

$$\mathbf{A}_{jk} = \int_\Omega \nabla V_j \cdot \nabla V_k \, dx = \sum_{T \in \mathcal{T}} \int_T \nabla V_j \cdot \nabla V_k \, dx \quad \text{for all } j,k = 1,\dots,N. \tag{2.10}$$

For the implementation, we build $\mathbf{A} \in \mathbb{R}^{N \times N}_{sym}$ with respect to all nodes and then solve (2.9) on the $n \times n$ subsystem corresponding to the free nodes.

3. MATLAB IMPLEMENTATION OF P1-GALERKIN FEM

In this section, we recall the MATLAB implementation of the P1-FEM from [2] and explain, why this code leads to a quadratic growth of the runtime with respect to the number of elements. We discuss how to write an efficient MATLAB code by use of vectorization. In particular, we collect a number of tricks which are used in our MATLAB implementations lateron. From this point of view, Section 3 is central, and the reader is enforced to read this section carefully. We finally provide a fully vectorized MATLAB implementation of the P1-FEM, which is empirically proven to be optimal in numerical experiments.

3.1. Data Structures. 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 [2]: We refer to Figure 1 for an exemplary triangulation $\mathcal{T}$ and corresponding data arrays `coordinates`, `elements`, `dirichlet`, and `neumann`, which are formally specified in the following:

The set of all nodes $\mathcal{N}$ is represented by the $N \times 2$ array `coordinates`, where $N = \#\mathcal{N}$. 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

`coordinates(`ℓ`,:)` $= [\ x_\ell \ \ y_\ell\ ].$

The choice of the coordinate system and the order of the nodes, i.e. the numbering of $\mathcal{N}$, is arbitrary.

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

`elements(`ℓ`,:)` $= [\ i \ \ j \ \ k\ ],$

where the nodes are given in counterclockwise order, i.e., the parametrization of the boundary ∂T_ℓ is mathematically positive. The order of the triangles, i.e. the numbering of $\mathcal{T}$, is arbitrary.

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 $E_\ell = \operatorname{conv}\{z_i, z_j\}$ on the Dirichlet boundary is stored in the form

`dirichlet(`ℓ`,:)=[` i j `]`.

It is assumed that $z_j - z_i$ gives the mathematically positive orientation of Γ, i.e. the outer normal vector of Ω on E_ℓ reads

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

where $z_k = (x_k, y_k) \in \mathbb{R}^2$. The order of Dirichlet edges is arbitrary. 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,:)`.

Listing 1

```
 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;
```

3.2. A First But Inefficient Matlab Implementation (Listing 1).

This section essentially recalls the Matlab code of [2] for later reference:

- 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 is assumed to take n evaluation points $\xi_j \in \mathbb{R}^2$ in form of a matrix $\xi \in \mathbb{R}^{n\times 2}$ and to return a column vector $y \in \mathbb{R}^n$ of the associated function values, e.g., $y_j = f(\xi_j)$. 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. (2.8), 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 (2.10). 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 $T_i \in \mathcal{T}$. Note that the matrix $B \in \mathbb{R}^{3\times 3}$ defined in Line 8 even provides $|T_i| = \det(B)/2$. Moreover, `grad(`ℓ`,:)` from Line 9 contains the gradient of the hat function $V_j|_{T_i}$ corresponding to the 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. (2.8).
- Lines 16–26: The load vector $\mathbf{b} \in \mathbb{R}^N$ from (2.9) is built. It is initialized by the contribution of the nodal interpolation of the Dirichlet data (Line 16), cf. (2.7) resp. (2.9). Next (Lines 17–21), we elementwise add the volume force
$$\int_\Omega f V_j \, dx = \sum_{T\in\mathcal{T}} \int_T f V_j \, dx.$$
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 $s_T \in T$
$$\int_T f V_j \, dx \approx |T| f(s_T) V_j(s_T) = \frac{|T|}{3} f(s_T) \quad \text{for } z_j \in T \in \mathcal{T}.$$
Finally (Lines 22–26), we elementwise add the Neumann contributions
$$\int_{\Gamma_N} g V_j \, ds = \sum_{E \subseteq \Gamma_N} \int_E g V_j \, ds.$$
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 $m_E \in E$
$$\int_E g V_j \, ds \approx h_E g(m_E) V_j(m_E) = \frac{h_E}{2} g(m_E) \quad \text{for } z_j \in E \in \mathcal{E}, \; E \subseteq \overline{\Gamma}_N,$$
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 (2.9) 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 approximation $U \in \mathcal{S}^1(\mathcal{T})$, cf. (2.8).

On a first glance, one might expect linear runtime of the function `solveLaplace` with respect to the number $\#\mathcal{T}$ of elements — at least up to the solution of the linear system in Line 29. Instead, one observes a quadratic dependence, cf. Table 1–2 and Figure 9 below.

Listing 2

```
%*** Assembly of stiffness matrix in linear complexity
nE = size(elements,1);
I = zeros(9*nE,1);
J = zeros(9*nE,1);
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 Remedy. A closer look on the MATLAB code of the function `solveLaplace` in Listing 1 reveals that the quadratic dependence of the runtime on the number $M = \#\mathcal{T}$ of elements is due to the assembly of the stiffness matrix $\mathbf{A} \in \mathbb{R}^{N\times N}$ (Lines 5–11): In MATLAB, sparse matrices are internally stored in the compressed column storage format (or: Harwell-Boeing format), cf. [6] 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. This theoretically predicted complexity is even observed in Table 2.

As has been pointed out by GILBERT, MOLER, and SCHREIBER [14], 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 $\mathcal{O}(n \log n)$, cf. Table 2 below.

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:

- 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: Storage of the matrix updates in the corresponding entries of the vectors I, J, and a: Note that 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) reveals that the new code has almost linear complexity with respect to $M = \#\mathcal{T}$, cf. Table 2. Moreover, in Figure 9 we even observe that the overall runtime of the modified function `solveLaplace` has logarithmic-linear growth. We remark that, in theory, the assembly of a P1-FEM stiffness matrix in compressed column storage format could be done in linear complexity with respect to the number $M = \#\mathcal{T}$ of elements. However, we observe a logarithmic-linear growth of the runtime for MATLAB's `sparse` function instead. This is probably due to the internal realization of sorting the coordinate vectors.

For the convenience of the reader, the following list collects some MATLAB built-in functions which are used below to optimize the performance of our MATLAB code. 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:

- We have already noted that dense matrices $A \in \mathbb{R}^{M\times N}$ are stored columnwise in MATLAB. The command `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 of $A \in \mathbb{R}^{M\times N}$ into $B \in \mathbb{R}^{m\times n}$ with $MN = mn$, where `B(:)` coincides with `A(:)`.
- For a (sparse or even dense) matrix $A \in \mathbb{R}^{M\times N}$,
  `[I,J,a] = find(A).`
  returns the coordinate format of 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 $A \in \mathbb{R}^{M\times N}$,
  `a = sum(A,2)`
  returns the rowwise sum $a \in \mathbb{R}^M$ of the entries of A, i.e., $a_j = \sum_{k=1}^N A_{jk}$. 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`.

LISTING 3

```
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;
```

```
25 if ~isempty(neumann)
26     cn1 = coordinates(neumann(:,1),:);
27     cn2 = coordinates(neumann(:,2),:);
28     gmE = feval(g,(cn1+cn2)/2);
29     b = b + accumarray(neumann(:),...
30                     repmat(2\sqrt(sum((cn2-cn1).^2,2)).*gmE,2,1),[nC 1]);
31 end
32 %*** Computation of P1-FEM approximation
33 freenodes = setdiff(1:nC, dirichlet);
34 x(freenodes) = A(freenodes,freenodes)\b(freenodes);
35 %*** Compute energy || grad(uh) ||^2 of discrete solution
36 energy = x'*A*x;
```

3.4. An Efficient MATLAB Implementation (Listing 3). In this section, we further improve the overall runtime of the P1-FEM implementation in MATLAB. First, 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 of the function `solveLaplace` can be achieved by use of MATLAB's vector arithmetics which allows to replace any `for`-loop.

- Line 10: Let $T = \text{conv}\{z_1, z_2, z_3\}$ 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. Since the assembly of I and J in Lines 12–13 is along the lines of Listing 2, it only remains to understand that Lines 14–17 compute the same vector a as before. Let $T = \text{conv}\{z_1, z_2, z_3\}$ denote a non-degenerate triangle with vertices $z_j \in \mathbb{R}^2$ $(j = 1, 2, 3)$ given in counterclockwise order and corresponding hat functions V_j. Using an affine mapping between T and a reference element $T_{\text{ref}} = \text{conv}\{(0,0),(0,1),(1,0)\}$ we are able to precalculate the local stiffness matrices up to a linear combination which depends on z_1, z_2, z_3 only, i.e.
$$\begin{aligned}\left(\int_T \nabla V_j \cdot \nabla V_k \, dx\right)_{j,k=1}^3 &= a\begin{pmatrix} -2 & 1 & 1 \\ 1 & 0 & -1 \\ 1 & -1 & 0 \end{pmatrix} + b\begin{pmatrix} 1 & -1 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} + c\begin{pmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} -2a+b+c & a-b & a-c \\ a-b & b & -a \\ a-c & -a & c \end{pmatrix}\end{aligned}$$
with
$$a = \frac{(z_2 - z_1)\cdot(z_3 - z_1)}{4|T|}, \quad b = \frac{|z_3 - z_1|^2}{4|T|}, \quad \text{and} \quad c = \frac{|z_2 - z_1|^2}{4|T|},$$
see [19, Chapter 4].
- 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.

4. Local Mesh Refinement and Coarsening

The accuracy of a discrete approximation 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. Using the adaptive algorithm of Section 5, this can be done automatically, which needs, however, some (local) mesh-refinement to improve $\mathcal{T}$. We provide MATLAB implementations of two popular mesh-refining techniques, namely newest vertex bisection and red-green-blue refinement. 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 newest vertex bisection, 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 [12] for the sake of completeness.

Listing 4

```
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
```

4.1. 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, we shall need a neighbour relation, e.g., for a given interior edge E we aim to find the unique elements $T_+, T_- \in \mathcal{T}$ such that $E = T_+ \cap T_-$. To avoid searching certain data structures, one usually builds appropriate further data. The assembly of this temporary data should be done most efficiently.

The mesh-refinement provided below is edge-based. In particular, we need to generate a numbering of the edges of $\mathcal{T}$. Moreover, 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. To this end, 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 $E_\ell = \mathrm{conv}\{z_j, z_k\}$. 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`)` returns 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 the boundary Γ into certain boundary conditions is hidden in the optional arguments `varargin` and `varargout`. This allows a more flexible treatment and a partition of Γ with respect to 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 $E = \mathrm{conv}\{z_i, z_j\}$ 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 $(i, j) \in \{(I_\ell, J_\ell) : \ell = 1, 2, \dots\} =: G$. From now on, we identify the edge E with the corresponding pair (i, j) in 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 `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.

Listing 5

```
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)];
```

4.2. Refinement by Newest Vertex Bisection (Listing 5). Before discussing the implementation, we briefly describe the idea of newest vertex bisection. 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 newest vertex bisection, 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:

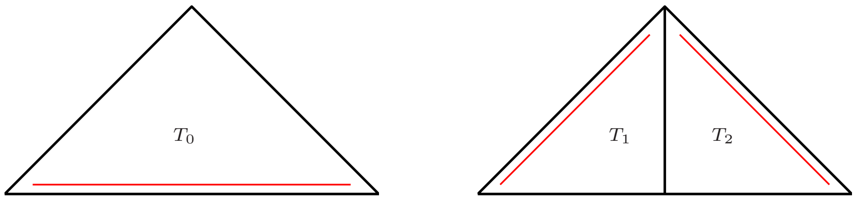


FIGURE 2. For each triangle $T_0 \in \mathcal{T}$, there is one *reference edge*, indicated by the double line (left). Refinement of T_0 is done by bisecting the reference edge, where its midpoint becomes a new node. The reference edges of the son triangles T_1 and T_2 are opposite to this newest vertex (right).

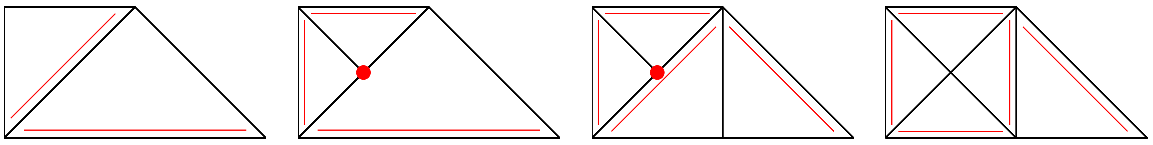


FIGURE 3. Closure of triangulation after refinement of marked elements by newest vertex bisection: Assume that the small element is marked for refinement (first from left). Bisection of the small element leads to a hanging node, indicated by a bullet (second from left). To avoid hanging nodes, the large element as well as one of its sons have to be refined (third from left). This leads to a final configuration, for which no hanging nodes occur (right). Throughout, the respective reference edges are indicated by a double line.

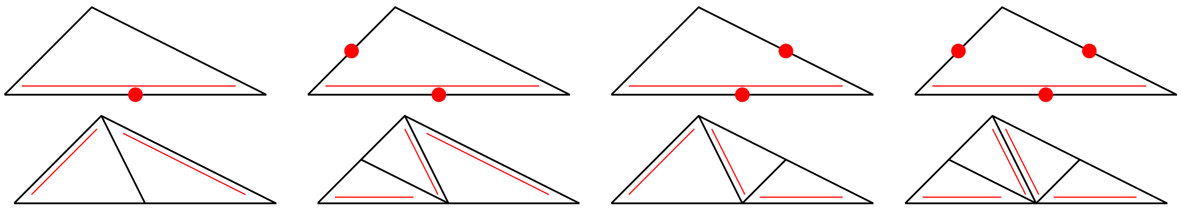


FIGURE 4. Refinement by newest vertex bisections: We consider one triangle T and assume that certain edges, but at least the reference edge, are marked for refinement (top). After refinement, the element is split into 2, 3, or 4 son triangles, respectively (bottom). Throughout, the reference edges are indicated by a double line.

- 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 $\mathcal{T}_\ell$, the resulting partition usually shows hanging nodes. Therefore, one does some additional bisections and finally obtains a regular triangulation $\mathcal{T}_{\ell+1}$, cf. Figure 3.

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 4:

- 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 would be 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.

This will also become clear in the implementation of newest vertex bisection in Listing 5, which is discussed below. For the implementation, we use the following convention: Let the element T_ℓ be stored by

`elements(`ℓ`,:)` $= [\ i\ \ j\ \ k\ ]$.

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

- Line 1: The function is usually called by
  ```
  [coordinates,elements,dirichlet,neumann] ...
        = refineNVB(coordinates,elements,dirichlet,neumann,marked)
  ```
  where `marked` is a vector containing the numbers of the elements which have to be refined.
- 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 mesh-refinement by newest vertex bisection, 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 edge that is marked for refinement, we compute the edge's midpoint as a new node of the refined triangulation. The number of new nodes is determined by the nonzero entries of the vector `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) newest vertex bisection: 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.
- Lines 53–70: Generate refined mesh according to newest vertex bisection: For all refinements, we respect a certain order of the sons of a refined element. Namely, if T is refined by newest vertex bisection 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. [12] and Section 4.4 below.

In numerical analysis, usually constants may depend on the shape of the elements of a triangulation. More precisely, 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 [20] which only depend on $\mathcal{T}_0$,

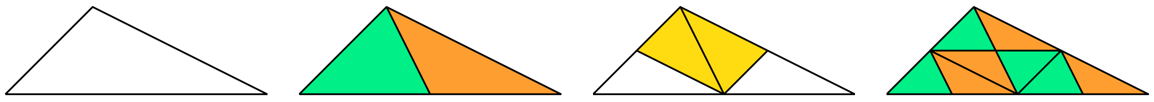


FIGURE 5. Refinement by newest vertex bisection only leads to finitely many interior angles for the family of all possible triangulations obtained by arbitrary newest vertex bisections. To see this, we start from a macro element (left), where the bottom edge is the reference edge. Using iterated newest vertex bisection, one observes that only four similarity classes of triangles occur, which are indicated by the coloring. After three steps of bisections (right), no additional similarity class appears.

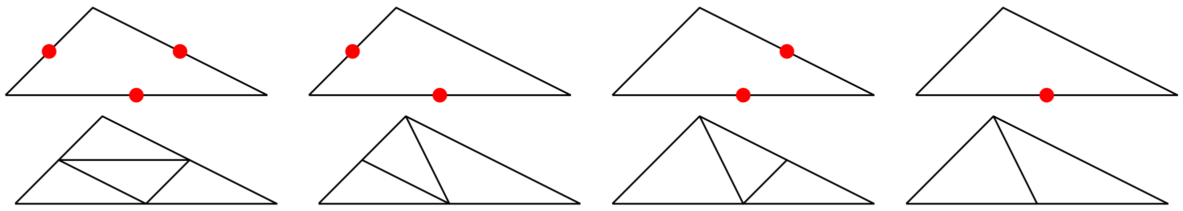


FIGURE 6. RGB refinement: If all edges of a triangle T are marked for refinement, T is split into four similar sons (left). If one or two edges of T are marked for refinement (amongst which, by definition, the longest edge), one uses newest vertex bisection with the longest edge of T as reference edge.

cf. Figure 5. In particular, there is a uniform lower bound for all interior angles in $\mathcal{T}_\ell$ which only depends on $\mathcal{T}_0$.

LISTING 6

```
%*** Sort elements such that first edge is longest
dx = coordinates(elements(:,[2,3,1]),1)-coordinates(elements,1);
dy = coordinates(elements(:,[2,3,1]),2)-coordinates(elements,2);
[hT,idxMax] = max(reshape(dx.^2+dy.^2,nE,3),[],2);
idx = ( idxMax==2 );
elements(idx,:) = elements(idx,[2,3,1]);
idx = ( idxMax==3 );
elements(idx,:) = elements(idx,[3,1,2]);
```

LISTING 7

```
red     = ( markedEdges(:,1) &  markedEdges(:,2) &  markedEdges(:,3) );

idx(red)     = 4; %*** red refinement

newElements([idx(red),1+idx(red),2+idx(red),3+idx(red)],:) ...
    = [elements(red,1),newNodes(red,1),newNodes(red,3); ...
       newNodes(red,1),elements(red,2),newNodes(red,2); ...
       newNodes(red,3),newNodes(red,2),elements(red,3); ...
       newNodes(red,2),newNodes(red,3),newNodes(red,1)];
```

4.3. Red-Green-Blue Refinement (Listing 6 and Listing 7). An alternative to newest vertex bisection is the red-green-blue strategy discussed in [21, Section 4.1]. As before, let $\mathcal{T}_0$ be a given initial mesh and $\mathcal{T}_\ell$ already be obtained by RGB refinement. We assume that certain elements of $\mathcal{T}_\ell$ are marked for further refinement, and we aim to construct $\mathcal{T}_{\ell+1}$. As for newest vertex bisection, we proceed recursively to move the markings to the edges:

- If an element $T \in \mathcal{T}_\ell$ is marked for refinement, we mark all edges of T for refinement.

- If any edge of $T \in \mathcal{T}_\ell$ is marked for refinement, we mark at least the longest edge for refinement.

Having completed the marking of edges, the refinement rules read as follows, cf. Figure 6:

- The midpoint of a marked edge becomes a new node.
- If all edges of an element $T \in \mathcal{T}_\ell$ are marked for refinement, T is red-refined, i.e., split into four similar elements.
- If two edges of an element $T \in \mathcal{T}_\ell$ are marked for refinement, the element is blue-refined: We use the longest edge of T as reference edge and bisect T by two successive newest vertex bisections.
- If only the longest edge of $T \in \mathcal{T}_\ell$ is marked for refinement, one uses green refinement, i.e. newest vertex bisection with respect to the longest edge of T.

The implementation of the newest vertex bisection (Listing 5) can easily be modified to yield a red-green-blue refinement strategy. The essential difference is that one has to guarantee that the reference edge opposite to the newest vertex is the longest edge of each triangle, i.e. we have to extend Listing 5 (before Line 6) by some sorting of the array `elements` in Listing 6. For each element T_ℓ stored by

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

this ensures that $E = \mathrm{conv}\{z_i, z_j\}$ is the longest edge of T_ℓ.

- Lines 2–4: The i-th row of the matrix `reshape(dx.^2+dy.^2,nE,3)` contains the three (squared) edge lengths of the element T_i. Taking, the rowwise maxima of this matrix, we obtain the column vector `hT` whose i-th component contains the maximal (squared) edge length of T_i, i.e., `hT(i)` is the (squared) diameter $h_{T_i}^2 = \mathrm{diam}(T_i)^2$.
- Lines 5–8: The index vector `idxMax` contains the information which edge (first, second, or third) of an element T_i is the longest. Consequently, we have to permute the nodes of an element T_i if (and only if) `idxMax(i)` equals two or three.

Despite of this, the only difference between NVB and RGB refinement is that one now uses a red refinement instead of three bisections if all edges of an element T are marked for refinement. We thus replace Line 44, Line 50, and Lines 66–70 in Listing 5 by the according lines of Listing 7.

As before it is worth noting, that the interior angles of $\mathcal{T}_\ell$ do not degenerate for $\ell \to \infty$. More precisely, if $C_0 > 0$ is a lower bound for the minimal interior angle of $\mathcal{T}_0$, then $C_0/2 > 0$ is a lower bound for the minimal interior angle of $\mathcal{T}_\ell$, where $\mathcal{T}_\ell$ is obtained from $\mathcal{T}_0$ by finitely many but arbitrary steps of RGB refinement [18].

Listing 8

```
function [coordinates,elements,varargout] = coarsenNVB(N0,coordinates,elements,varargin)
nC = size(coordinates,1);
nE = size(elements,1);
%*** Obtain geometric information on neighbouring elements
I = elements(:);
J = reshape(elements(:,[2,3,1]),3*nE,1);
nodes2edge = sparse(I,J,1:3*nE);
mask = nodes2edge>0;
[foo{1:2},idxIJ] = find( nodes2edge );
[foo{1:2},neighbourIJ] = find( mask + mask.*sparse(J,I,[1:nE,1:nE,1:nE]') );
element2neighbours(idxIJ) = neighbourIJ - 1;
element2neighbours = reshape(element2neighbours,nE,3);
%*** Determine which nodes (created by refineNVB) are deleted by coarsening
marked = zeros(nE,1);
marked(varargin{end}) = 1;
newestNode = unique(elements((marked & elements(:,3)>N0),3));
valence = accumarray(elements(:),1,[nC 1]);
markedNodes = zeros(nC,1);
markedNodes(newestNode((valence(newestNode) == 2 | valence(newestNode) == 4))) = 1;
%*** Collect pairs of brother elements that will be united
```

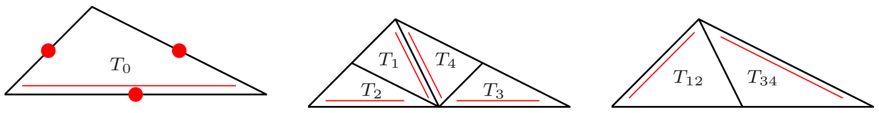


FIGURE 7. Coarsening is not fully inverse to refinement by newest vertex bisections: Assume that all edges of a triangle are marked for refinement (left). Refinement then leads to 4 son elements (middle). One application of the coarsening algorithm only removes the bisections on the last level (right).

```
idx = find(markedNodes(elements(:,3)) & (element2neighbours(:,3) > (1:nE)'))';
markedElements = zeros(nE,1);
markedElements(idx) = 1;
for element = idx
    if markedElements(element)
        markedElements(element2neighbours(element,3)) = 0;
    end
end
idx = find(markedElements);
%*** Coarsen two brother elements
brother = element2neighbours(idx,3);
elements(idx,[1 3 2]) = [elements(idx,[2 1]) elements(brother,1)];
%*** Delete redundant nodes
activeNodes = find(~markedNodes);
coordinates = coordinates(activeNodes,:);
%*** Provide permutation of nodes to correct further data
coordinates2newCoordinates = zeros(1,nC);
coordinates2newCoordinates(activeNodes) = 1:length(activeNodes);
%*** Delete redundant elements + correct elements
elements(brother,:) = [];
elements = coordinates2newCoordinates(elements);
%*** Delete redundant boundaries + correct boundaries
for j = 1:nargout-2;
    boundary = varargin{j};
    if ~isempty(boundary)
      node2boundary = zeros(nC,2);
      node2boundary(boundary(:,1),1) = 1:size(boundary,1);
      node2boundary(boundary(:,2),2) = 1:size(boundary,1);
      idx = ( markedNodes & node2boundary(:,2) );
      boundary(node2boundary(idx,2),2) = boundary(node2boundary(idx,1),2);
      boundary(node2boundary(idx,1),2) = 0;
      varargout{j} = coordinates2newCoordinates(boundary(find(boundary(:,2)),:));
    else
      varargout{j} = [];
    end
end
```

4.4. Coarsening of Refined Meshes (Listing 8).

Our MATLAB function `coarsenNVB` is a vectorized version of a MATLAB implementation by Chen and Zhang [11]. However, our code extends the prior version in the sense that a subset of elements can be chosen for coarsening and redundant memory is set free, e.g., former nodes which have been removed by coarsening. Moreover, our code respects the boundary conditions which are also affected by coarsening of $\mathcal{T}$.

The code aims to coarsen $\mathcal{T}$ by removing certain newest vertices 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. The idea of the algorithm proposed in [11] is that 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. In [11] it is shown that $z \in \mathcal{N} \backslash \mathcal{N}_0$ 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 refer to [11] for the proofs.

We stress that `coarsenNVB` only coarsens marked leaves of the current forest generated by newest vertex bisection. Said differently, `coarsenNVB` is *not* inverse to `refineNVB`: For instance, assume that all edges of a triangle T_0 are marked for refinement and apply newest vertex bisection to T_0, cf. Figure 7 above. In a first (theoretical, but not implementational) step, T_0 is bisected into elements T_{12} and T_{34}. In a second step, T_{12} is bisected into T_1 and T_2, and T_{34} is bisected into T_3 and T_4. The application of `coarsenNVB` (with all elements marked for coarsening) then leads to a triangulation containing T_{12} and T_{34} instead of the coarse element T_0. 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.

- Line 1: The function is usually called by
  ```
  [coordinates,elements,dirichlet,neumann] ...
        = coarsenNVB(N0,coordinates,elements,dirichlet,neumann,marked)
  ```
  where `N0` denotes the number $\#\mathcal{N}_0$ of nodes in the initial mesh and `marked` is a vector containing the numbers of the elements to be coarsened (if possible).
- 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 $k = 0$ if the edge is a boundary edge.
- Lines 14–19: We mark nodes which are admissible for coarsening, where we take into account the coarsening rules of [11] in Lines 17–19. However, we consider only newest vertices added by `refineNVB`, for which the corresponding elements are marked for coarsening (Lines 14–16).
- 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).

5. 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 $M = \#\mathcal{T} \leq M_{\max}$ 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

$$\eta_T \approx \|u - U\|_{H^1(T)} \quad \text{for all } T \in \mathcal{T}. \tag{5.1}$$

In particular, the associated *error estimator* $\eta = \big(\sum_{T\in\mathcal{T}} \eta_T^2\big)^{1/2}$ then yields an error estimate $\eta \approx \|u - U\|_{H^1(\Omega)}$. Some examples for error estimators and associated refinement indicators are given in the subsequent sections. Throughout, the focus is on so-called *a posteriori error estimators*, where η is computed after the computation and with the knowledge of a discrete solution U.

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.

Listing 9

```
 1 function [x,coordinates,elements,indicators] ...
 2     = adaptiveAlgorithm(coordinates,elements,dirichlet,neumann,f,g,uD,nEmax,rho)
 3 while 1
 4     %*** Compute discrete solution
 5     x = solveLaplace(coordinates,elements,dirichlet,neumann,f,g,uD);
 6     %*** Compute refinement indicators
 7     indicators = computeEtaR(x,coordinates,elements,dirichlet,neumann,f,g);
 8     %*** Stopping criterion
 9     if size(elements,1) ≥ nEmax
10         break
11     end
12     %*** Mark elements for refinement
13     [indicators,idx] = sort(indicators,'descend');
14     sumeta = cumsum(indicators);
15     ell = find(sumeta≥sumeta(end)*rho,1);
16     marked = idx(1:ell);
17     %*** Refine mesh
18     [coordinates,elements,dirichlet,neumann] = ...
19        refineNVB(coordinates,elements,dirichlet,neumann,marked);
20 end
```

5.1. Adaptive Algorithm (Listing 9). 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 [13], which seeks to determine the minimal set $\mathcal{M} \subseteq \mathcal{T}$ such that

$$\varrho \sum_{T\in\mathcal{T}} \eta_T^2 \le \sum_{T\in\mathcal{M}} \eta_T^2, \tag{5.2}$$

for some parameter $\varrho \in (0,1)$. 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 $\varrho \to 1$ corresponds to almost uniform mesh-refinement, i.e. most of the elements are marked for refinement, whereas $\varrho \to 0$ leads to highly adapted meshes.

- Line 1–2: The function takes the initial mesh described by `coordinates`, `elements`, `dirichlet`, and `neumann` as well as the problem data f, g, and u_D. Moreover, the user provides the maximal number `nEmax` of elements as well as the adaptivity parameter ϱ from (5.2). After termination, the function returns the coefficient vector `x` of the final Galerkin solution $U \in \mathcal{S}^1_D(\mathcal{T})$, cf. (2.8), the associated final mesh $\mathcal{T}$, and the corresponding vector `indicators` of elementwise error indicators.
- Line 3–20: As long as the number M 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 $\eta_j^2 := \eta_{T_j}^2$. Line 13–16 is the realization of the marking criterion (5.2): We first 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 $\varrho \sum_{j=1}^M \eta_j^2 = \varrho \sum_{j=1}^M \eta^2_{\pi(j)} \leq \sum_{j=1}^{\ell} \eta^2_{\pi(j)}$ (Line 15). Formally, we thus determine the set $\mathcal{M} = \{T_{\pi(j)} : j = 1, \dots, \ell\}$ of marked elements (Line 16). Finally (Lines 18–19), we refine the marked elements and so generate a new mesh.

In the current state of research, the Dörfler criterion (5.2) is used to prove convergence and optimality of AFEM [9]. In practice, more often the bulk criterion is used, which marks elements $T \in \mathcal{T}$ for refinement provided

$$\eta_T \geq \theta \max_{T' \in \mathcal{T}} \eta_{T'}, \tag{5.3}$$

where $\theta \in [0, 1]$ is a given parameter. To use it in the adaptive algorithm, one may simply replace Lines 13–16 of Listing 9 by

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

where `theta` is the parameter θ from (5.3) which also replaces `rho` in the function call. In the current state of research, however the bulk criterion (5.3) is only proven to yield convergence [17], whereas optimal convergence of AFEM is only empirically observed. In the exemplary code of Listing 9, we consider the residual-based error estimator from Section 5.2. This can be replaced by other error estimators, e.g., the hierarchical error estimator or the ZZ-type error estimator from Sections 5.3–5.4 below. Finally, the function call of `refineNVB` in Line 16 can be replaced by `refineRGB` to yield adaptive RGB refinement instead of newest vertex bisection.

Listing 10

```
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;
```

5.2. Residual-Based Error Estimator (Listing 10).

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

$$\eta_T^2 := h_T^2 \|f\|_{L^2(T)}^2 + h_T \|J_h(\partial_n U)\|_{L^2(\partial T\cap\Omega)}^2 + h_T \|g - \partial_n U\|_{L^2(\partial T\cap\Gamma_N)}^2. \tag{5.4}$$

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

$$J_h(\partial_n U)|_E := \nabla U|_{T_+} \cdot n_+ + \nabla U|_{T_-} \cdot n_-, \tag{5.5}$$

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

$$C_{\rm rel}^{-1}\, \|u - U\|_{H^1(\Omega)} \le \eta_R \le C_{\rm eff}\big[\|u - U\|_{H^1(\Omega)} + \|h(f - f_{\mathcal{T}})\|_{L^2(\Omega)} + \|h^{1/2}(g - g_{\mathcal{E}})\|_{L^2(\Gamma_N)}\big] \tag{5.6}$$

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 and the data f, g, see [21, 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 $g|_E \approx g(m_E)$, where s_T again denotes the center of mass of an element $T \in \mathcal{T}$ and where m_E denotes the midpoint of $E \in \mathcal{E}$. We realize

$$\widetilde{\eta}_T^2 := |T|^2\, f(s_T)^2 + \sum_{E\in\partial T\cap\Omega} h_E^2\,\big(J_h(\partial_n U)|_E\big)^2 + \sum_{E\in\partial T\cap\Gamma_N} h_E^2\,\big(g(m_E) - \partial_n U|_E\big)^2 \tag{5.7}$$

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

$$h_E \le h_T \le C\, h_E \quad \text{as well as} \quad 2|T| \le h_T^2 \le C\,|T|, \quad \text{for all } T\in\mathcal{T} \text{ with edge } E\subset\partial T, \tag{5.8}$$

with some constant $C > 0$, which only depends on a lower bound for the minimal interior angle and thus stays bounded for both, newest vertex bisection and red-green-blue refinement. Up to some higher-order consistency errors, the refinement indicators $\widetilde{\eta}_T$ and η_T are therefore equivalent.

The implementation from Listing 10 returns the vector of squared refinement indicators $(\widetilde{\eta}_{T_1}^2, \dots, \widetilde{\eta}_{T_M}^2)$, 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 $({\rm curl}U)|_T = (-\partial U/\partial x_2, \partial U/\partial x_1)|_T \in \mathbb{R}^2$ 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 $z_j = (x_j, y_j) \in \mathbb{R}^2$. Note that the gradient of V_j reads

  $$\nabla V_j|_T = \frac{1}{2|T|}\,(y_{j+1} - y_{j+2}, x_{j+2} - x_{j+1}), \tag{5.9}$$

  where we identify $z_4 = z_1$ and $z_5 = z_2$. In particular, there holds

  $$2|T|\,{\rm curl}V_j|_T = (x_{j+1} - x_{j+2}, y_{j+1} - y_{j+2}) = z_{j+1} - z_{j+2},$$

  where we assume $z_j \in \mathbb{R}^2$ to be a row-vector. With $U|_T = \sum_{j=1}^3 u_j V_j$, we infer

  $$\begin{aligned} 2|T|\,{\rm curl}U|_T = 2|T| \sum_{j=1}^3 u_j\, {\rm curl}V_j|_T &= u_1\,(z_2 - z_3) + u_2\,(z_3 - z_1) + u_3\,(z_1 - z_2) \\ &= (z_3 - z_1)(u_2 - u_1) - (z_2 - z_1)(u_3 - u_1), \end{aligned} \tag{5.10}$$

  which is realized in Lines 11–13.

- 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). This leads to a vector `etaR` which contains $h_E\, J_h(\partial U/\partial n)|_E$ for all interior edges $E \in \mathcal{E}$, whereas `etaR` contains $h_E\,(\partial U/\partial n)|_E$ 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. (5.7).
- Line 29: Assembly of edge contributions of $\widetilde{\eta}_T$. We compute
$$\sum_{E \in \partial T \cap \Omega} h_E^2\,\big(J_h(\partial_n U)|_E\big)^2 + \sum_{E \in \partial T \cap \Gamma_N} h_E^2\,\big(g(m_E) - \partial_n U|_E\big)^2$$
for all $T \in \mathcal{T}$ simultaneously.
- Line 31–32: We finally add the volume contribution $\big(|T| f(s_T)\big)^2$, which yields $\widetilde{\eta}_T^2$ for all $T \in \mathcal{T}$.

Listing 11

```
1  function etaH = computeEtaH(x,coordinates,elements,dirichlet,neumann,f,g)
2  nE = size(elements,1);
3  [edge2nodes,element2edges,dirichlet2edges,neumann2edges] ...
4      = provideGeometricData(elements,dirichlet,neumann);
5  %*** First vertex of elements and corresponding edge vectors
6  c1  = coordinates(elements(:,1),:);
7  d21 = coordinates(elements(:,2),:) - c1;
8  d31 = coordinates(elements(:,3),:) - c1;
9  %*** Vector of element volumes 2*|T|
10 area2 = d21(:,1).*d31(:,2)-d21(:,2).*d31(:,1);
11 %*** Elementwise gradient of uh
12 u21 = repmat( (x(elements(:,2))-x(elements(:,1)))./area2, 1,2);
13 u31 = repmat( (x(elements(:,3))-x(elements(:,1)))./area2, 1,2);
14 grad = (d31.*u21 - d21.*u31)*[0 -1 ; 1 0];
15 %*** Elementwise integrated gradients of hat functions --> 2*int(T,grad Vj)
16 grad1 = [d21(:,2)-d31(:,2) d31(:,1)-d21(:,1)];
17 grad2 = [d31(:,2) -d31(:,1)];
18 grad3 = [-d21(:,2) d21(:,1)];
19 %*** Compute volume contribution of rT (contribution of element bubble)
20 fsT = feval(f,c1+(d21+d31)/3);
21 rT = area2.*fsT/120;
22 %*** Compute volume contributions of rE edgewise (contribution of edge bubbles)
23 rE = repmat(area2.*fsT/24,1,3) - [sum((grad1+grad2).*grad,2) ...
24                                   sum((grad2+grad3).*grad,2) ...
25                                   sum((grad3+grad1).*grad,2)]/6;
26 rE = accumarray(element2edges(:),rE(:),[size(edge2nodes,1) 1]);
27 %*** Incorporate Neumann contributions to rE
28 if ~isempty(neumann)
29     cn1 = coordinates(edge2nodes(neumann2edges,1),:);
```

```
    cn2 = coordinates(edge2nodes(neumann2edges,2),:);
    gmE = feval(g,(cn1+cn2)/2);
    rE(neumann2edges) = rE(neumann2edges) + sqrt(sum((cn2-cn1).^2,2)).*gmE/6;
end
%*** Incorporate Dirichlet data to rE
rE(dirichlet2edges) = 0;
%*** Compute error indicators
etaH = rT.^2 + accumarray(repmat(1:nE,1,3)',rE(element2edges(:)).^2,[nE 1]);
```

5.3. Hierarchical Error Estimator (Listing 11). Next, we consider the hierarchical error estimator $\eta_H := \big(\sum_{T\in\mathcal{T}} \eta_T^2\big)^{1/2}$ introduced by Bank and Smith [5], where our presentation follows [21, Section 1.4]. Here,

$$\eta_T^2 := \Big(\int_\Omega f b_T\,dx\Big)^2 + \sum_{E\subset\partial T\backslash\Gamma_D}\Big(\int_\Omega (f b_E - \nabla U\cdot\nabla b_E)\,dx + \int_{\Gamma_N} g b_E\,ds\Big)^2 \tag{5.11}$$

with the cubic element bubble b_T and the quadratic edge bubble b_E given by

$$b_T = \prod_{z\in\mathcal{N}\cap T} V_z \quad\text{and}\quad b_E = \prod_{z\in\mathcal{N}\cap E} V_z, \tag{5.12}$$

respectively. The error estimator η_H is reliable and efficient in the sense of (5.6), see [21, Proposition 1.15]. The function from Listing (11) computes the vector $(\widetilde{\eta}_{T_1}^2,\dots,\widetilde{\eta}_{T_M}^2)$, where $M = \#\mathcal{T}$ and $\widetilde{\eta}_T \approx \eta_T$ in the sense that the integrals in (5.11) are replaced by appropriate quadrature formulae discussed below.

- Lines 5–10 are discussed for Listing 3 above, see Section 3.4.
- Lines 12–14: Compute the $\mathcal{T}$-elementwise gradient $\nabla U|_T$ simultaneously for all elements $T\in\mathcal{T}$. For details on the computation, see Lines 12–14 of Listing 10, where we compute $\mathrm{curl}U|_T$, cf. Equation (5.10). We only stress that $\mathrm{curl}U|_T$ is the orthogonally rotated gradient $\nabla U|_T$, namely
$$\nabla U|_T = \mathrm{curl}U|_T \begin{pmatrix} 0 & -1\\ 1 & 0\end{pmatrix},$$
where $\mathrm{curl}U|_T$ and $\nabla U|_T$ are understood as row vectors.
- Lines 16–18: With z_1, z_2, z_3 the vertices of an element $T\in\mathcal{T}$ in counterclockwise order, we compute the integrated gradients $2\int_T \nabla V_j\,dx = 2|T|\,\nabla V_j|_T$ of the associated hat functions, cf. Equation (5.9).
- Lines 20–21: The volume term $\int_\Omega f b_T\,ds = \int_T f b_T\,dx$ is computed by quadrature. Up to higher-order terms, it is sufficient to choose a quadrature rule of third order so that the integral is computed exactly for a $\mathcal{T}$-piecewise constant force f. We employ a 10-point Newton-Côtes formula, which evaluates the integrand on the edges and in the barycenter s_T of T. Since b_T vanishes on all edges and outside of T, we obtain
$$\int_\Omega f b_T\,ds \approx \frac{9}{20}\,|T| f(s_T) b_T(s_T) = \frac{|T|}{60}\,f(s_T).$$
- Lines 23–26: We compute, for all edges $E\in\mathcal{E}$, the integrals $\int_\Omega f b_E\,dx - \int_\Omega \nabla U\cdot\nabla b_E\,dx$ by numerical quadrature. First, this is done $\mathcal{T}$-elementwise. Up to higher-order terms, it is sufficient to approximate $\int_T f b_E$ by a quadrature rule of second order. We employ a 4-point formula, which evaluates the integrand in the vertices and in the barycenter s_T of T. Since b_E vanishes in all nodes of T, this yields
$$\int_T f b_E\,ds \approx \frac{3}{4}\,|T|\,f(s_T) b_E(s_T) = \frac{|T|}{12}\,f(s_T),$$
which is independent of the edge E of T. Let z_1, z_2, z_3 denote the vertices of a triangle $T\in\mathcal{T}$ in counterclockwise order with associated hat functions V_j. Let $E_j = \mathrm{conv}\{z_j, z_{j+1}\}$

denote the j-th edge of T with associated edge bubble function $b_j = V_j V_{j+1}$, where we identify $z_4 = z_1$ and $V_4 = V_1$. Then, $\nabla b_j = V_j \nabla V_{j+1} + V_{j+1} \nabla V_j$ leads to

$$\begin{aligned}\int_T \nabla U \cdot \nabla b_j \, dx &= \nabla U|_T \cdot \Big(\int_T V_j \, dx \, \nabla V_{j+1}|_T + \int_T V_{j+1} \, dx \, \nabla V_j|_T \Big) \\ &= \frac{|T|}{3} \nabla U|_T \cdot \big(\nabla V_j|_T + \nabla V_{j+1}|_T \big) \\ &= \frac{1}{6} \nabla U|_T \cdot \Big(2 \int_T \big(\nabla V_j + \nabla V_{j+1} \big) \, dx \Big).\end{aligned}$$

This explains Lines 23–25. Finally (Line 26), we compute, for all edges $E \in \mathcal{E}$,

$$\int_\Omega \big(f b_E - \nabla U \cdot \nabla b_E \big) \, dx = \sum_{\substack{T \in \mathcal{T} \\ E \subseteq \partial T}} \int_T \big(f b_E - \nabla U \cdot \nabla b_E \big) \, dx.$$

- Lines 28–33: For Neumann edges $E \subseteq \Gamma_N$, we have to add $\int_{\Gamma_N} g b_E \, ds = \int_E g b_E \, ds$. The latter integral is approximated by the Simpson rule, which yields

$$\int_{\Gamma_N} g b_E \, ds \approx \frac{2}{3} h_E g(m_E) b_E(m_E) = \frac{h_E}{6} g(m_E).$$

- Line 35: Dirichlet edges do not contribute to (5.11), and we set the corresponding edge contributions to zero.
- Line 37: We sum the contributions of η_T^2 as indicated by (5.11).

Listing 12

```matlab
function etaZ = computeEtaZ(x,coordinates,elements)
nE = size(elements,1);
nC = size(coordinates,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 volumes 2*|T|
area2 = d21(:,1).*d31(:,2)-d21(:,2).*d31(:,1);
%*** Elementwise integrated gradient of uh --> 2*int(T,grad uh)
u21 = x(elements(:,2))-x(elements(:,1));
u31 = x(elements(:,3))-x(elements(:,1));
dudx = d31(:,2).*u21 - d21(:,2).*u31;
dudy = d21(:,1).*u31 - d31(:,1).*u21;
%*** Compute coefficients for Clement interpolant Jh(grad uh)
zArea2 = accumarray(elements(:),[area2;area2;area2],[nC 1]);
qz = [accumarray(elements(:),[dudx;dudx;dudx],[nC 1])./zArea2, ...
      accumarray(elements(:),[dudy;dudy;dudy],[nC 1])./zArea2];
%*** Compute ZZ-refinement indicators
dudx = dudx./area2;
dudy = dudy./area2;
sz = [dudx dudx dudx dudy dudy dudy] - reshape(qz(elements,:), nE,6);
etaZ = (sum(sz.^2,2) + sum(sz.*sz(:,[2 3 1 5 6 4]),2)).*area2/12;
```

5.4. ZZ-Type Error Estimator (Listing 12). Error estimation by local averaging has been proposed in the pioneering work of Zienkiewicz and Zhu [23]. The idea is to smoothen the discrete gradient ∇U appropriately to obtain an improved approximation of ∇u. This smoothing is usually based on some averaging operator

$$\mathcal{A}_h : \mathcal{P}^0(\mathcal{T})^2 \to \mathcal{S}^1(\mathcal{T})^2 \tag{5.13}$$

which maps the $\mathcal{T}$-piecewise constant gradient ∇U onto a $\mathcal{T}$-piecewise affine and globally continuous vector field. Originally, the work [23] uses the usual Clément interpolation operator defined by

$$\mathcal{A}_h q = \sum_{z\in\mathcal{N}} q_z V_z \quad\text{with}\quad q_z := \frac{1}{|\Omega_z|}\int_{\Omega_z} q\,dx \quad\text{and}\quad \Omega_z := \bigcup\{T\in\mathcal{T}\,:\,z\in T\}. \tag{5.14}$$

The ZZ-type error estimator then reads

$$\eta_Z := \|\nabla U - \mathcal{A}_h\nabla U\|_{L^2(\Omega)} = \Big(\sum_{T\in\mathcal{T}} \eta_T^2\Big)^{1/2} \quad\text{with}\quad \eta_T = \|\nabla U - \mathcal{A}_h\nabla U\|_{L^2(T)}. \tag{5.15}$$

For the Dirichlet problem $\Gamma = \Gamma_D$, this error estimator is reliable and efficient up to terms of higher order [8]. For the mixed boundary value problem, the operator $\mathcal{A}_h$ has to be slightly modified, and we refer to [3] for details.

- Lines 5–9 are discussed for Listing 3 above, see Section 3.4.
- Lines 11–14 compute the weighted $\mathcal{T}$-elementwise gradient $2|T|\,\nabla U|_T = 2\int_T \nabla U\,dx$ simultaneously for all elements $T\in\mathcal{T}$. The gradients $\nabla U|_T$ are computed analogously to Lines 12–14 in Listing 11. The vectors `dudx` and `dudy` store the x- resp. y-component of the elementwise quantity $2|T|\,\nabla U|_T$.
- Lines 16–18: In Line 16, the areas $2|\Omega_z|$ from (5.14) are computed simultaneously for all nodes $z\in\mathcal{N}$. Consequently, Lines 17–18 compute the Clément coefficients $q_z\in\mathbb{R}^2$,
$$q_z = \frac{1}{|\Omega_z|}\int_{\Omega_z}\nabla U\,dx = \frac{1}{2|\Omega_z|}\sum_{T\ni z} 2|T|\,\nabla U|_T \quad\text{for all nodes } z\in\mathcal{N}.$$
- Lines 20–23: Simultaneous assembly of the elementwise contributions $\|\nabla U - \mathcal{A}_h\nabla U\|_{L^2(T)}$ of (5.15) for all elements $T\in\mathcal{T}$. Let $z_{\ell,1}, z_{\ell,2}, z_{\ell,3}\in\mathcal{N}$ be the nodes of an element T_ℓ in counterclockwise order. With $V_{\ell,j}$ the corresponding hatfunction and $q_{\ell,j}$ the associated Clément weights, there holds
$$\|\nabla U - \mathcal{A}_h\nabla U\|^2_{L^2(T_\ell)} = \Big\|\nabla U|_T - \sum_{j=1}^3 q_{\ell,j}V_{\ell,j}\Big\|^2_{L^2(T_\ell)} = \Big\|\sum_{j=1}^3 (\nabla U|_T - q_{\ell,j})V_{\ell,j}\Big\|^2_{L^2(T_\ell)}$$
With $M = \#\mathcal{T}$ the number of elements, the coefficients $s_{\ell,j} := \nabla U|_{T_\ell} - q_{\ell,j}$ are stored in an $M\times 6$ array `sz` (Line 22) in the form `sz(`ℓ`,:)` $= [\,s_{\ell,1}\,,\ s_{\ell,2}\,,\ s_{\ell,3}\,]$, where $s_{\ell,j}$ is understood as 1×2 row vector. Note that
$$\int_{T_\ell} V_{\ell,i}V_{\ell,j}\,dx = \begin{cases} |T_\ell|/6 & \text{for } i=j,\\ |T_\ell|/12 & \text{for } i\neq j.\end{cases}$$
This leads to
$$\begin{aligned}\|\nabla U - \mathcal{A}_h\nabla U\|^2_{L^2(T_\ell)} &= \sum_{i,j=1}^3 s_{\ell,i}\cdot s_{\ell,j}\int_{T_\ell} V_{\ell,i}V_{\ell,j}\,dx\\ &= \frac{|T|}{6}\big(|s_{\ell,1}|^2 + |s_{\ell,2}|^2 + |s_{\ell,3}|^2 + s_{\ell,1}\cdot s_{\ell,2} + s_{\ell,1}\cdot s_{\ell,3} + s_{\ell,2}\cdot s_{\ell,3}\big),\end{aligned}$$
so that Line 23 finally computes the vector $\big(\eta_{T_1}^2,\dots,\eta_{T_M}^2\big)$.

6. Numerical Experiments

To underline the efficiency of the developed Matlab code, this section briefly comments on some numerical experiments. Throughout, we used the latest Matlab version 7.6.0.324 (R2008a) on a common dual-board 64-bit PC with 7 GB of RAM and two AMD Opteron(tm) 248 CPUs with 1 MB cache and 2.2 GHz each running under Linux.

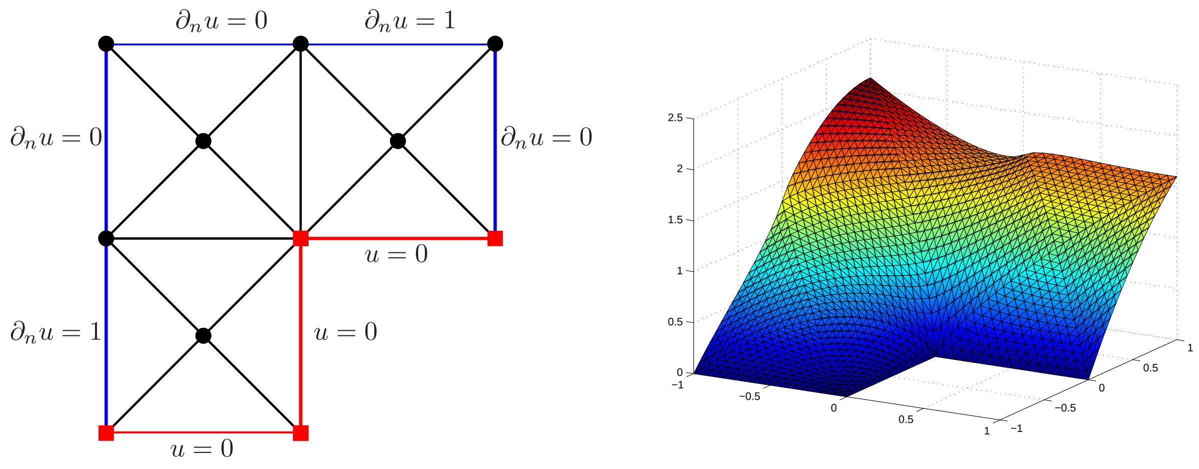


FIGURE 8. Coarse triangulation $\mathcal{T}_0$ of the L-shaped domain $\Omega = (-1,1)^2 \backslash ([0,1] \times [-1,0])$ into 12 triangles used for all numerical experiments (left). In Example 6.1, the Dirichlet boundary Γ_D consists of 3 edges which are plotted in red. The Neumann boundary Γ_N consists of 5 edges which are plotted in blue. The Dirichlet nodes $\mathcal{N} \cap \Gamma_D$ are indicated by red squares, the free nodes $\mathcal{N} \backslash \Gamma_D$ are visualized by black bullets. The exact solution has a singular behaviour (right), and we plot the discrete solution $U \in \mathcal{S}^1_D(\mathcal{T}_1)$ on a uniform triangulation $\mathcal{T}_1$ with $M_1 = 3.072$ triangles.

6.1. Stationary Model Problem. For the first experiment, we consider the Poisson problem

$$-\Delta u = 1 \quad \text{in } \Omega \tag{6.1}$$

with mixed Dirichlet-Neumann boundary conditions, where the L-shaped domain as well as the boundary conditions are visualized in Figure 8. The exact solution then shows a singular behaviour at the re-entrant corner. Before the actual computations, the triangulation shown in Figure 8 is generally refined by four successive uniform RGB refinements, which leads to 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.

The focus of the experiments is on the following aspects: First, we compare the computational times for different implementations of the matrix assembly discussed in Section 3.2–3.4. Second, we consider the runtimes for the modules `computeEtaR`, `computeEtaH`, `computeEtaZ`, and `refineNVB` for various meshes obtained by adaptive or uniform refinement. Finally, we show that these codes favour the use of adaptive mesh-refinement when we plot the Galerkin error over the total runtime.

Throughout, 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 20 iterations for the evaluation of these computational times, where the slowest execution is eliminated.

Table 1 gives the computational times of certain components of our MATLAB code for uniform and adaptive mesh-refinement, namely the assembly of the Galerkin data, the indicator-wise computation of the three error estimators from Section 5.2–5.4 as well as the refinement by newest vertex bisection (NVB). For the assembly, we consider the naive implementation from Listing 1 resp. [2] (slow), the implementation from Listing 2 (medium), and the fully vectorized implementation from Listing 3 (optimized). Finally (optimized2), we give the assembly times by use of Listing 3 and an improved version of `sparse` which is written in C and embedded through MATLAB's `MEX` interface [16]. Since this appears to be twice as fast as the original `sparse` function, we use the improved implementation also for the computation of the error estimators η_R, η_H, and η_Z as well as for the mesh-refinement (NVB).

Besides the absolute computational times, one is always interested in the computational complexity, i.e. in the dependence of the computational time $t(M)$ on the number of elements M.

	assembly time $[s]$				computational time $[s]$			time $[s]$
N	slow	medium	optimized	optimized2	η_R	η_H	η_Z	NVB
3072	4.10_{-01}	2.20_{-01}	1.00_{-02}	1.00_{-02}	1.00_{-02}	1.00_{-02}	1.00_{-02}	1.00_{-02}
12288	1.83_{+00}	8.75_{-01}	4.00_{-02}	3.00_{-02}	2.00_{-02}	4.00_{-02}	3.00_{-02}	6.00_{-02}
49152	3.65_{+01}	3.67_{+00}	3.40_{-01}	2.60_{-01}	2.30_{-01}	2.20_{-01}	1.45_{-01}	1.90_{-01}
196608	6.17_{+02}	1.42_{+01}	1.00_{+00}	5.95_{-01}	5.45_{-01}	7.60_{-01}	3.00_{-01}	7.50_{-01}
786432	1.00_{+04}	5.70_{+01}	3.75_{+00}	2.12_{+00}	1.89_{+00}	2.53_{+00}	1.21_{+00}	3.02_{+00}
3145728	–	2.29_{+02}	1.61_{+01}	8.46_{+00}	7.53_{+00}	1.00_{+01}	4.86_{+00}	1.21_{+01}

	assembly time $[s]$				computational time $[s]$			time $[s]$
N	slow	medium	optimized	optimized2	η_R	η_H	η_Z	NVB
3072	4.10_{-01}	2.20_{-01}	1.00_{-02}	1.00_{-02}	1.00_{-02}	1.00_{-02}	1.00_{-02}	1.00_{-02}
3134	4.40_{-01}	2.20_{-01}	1.00_{-02}	1.00_{-02}	5.00_{-03}	1.00_{-02}	0.00_{+00}	1.00_{-02}
3584	4.80_{-01}	2.50_{-01}	1.00_{-02}	1.00_{-02}	5.00_{-03}	1.00_{-02}	1.00_{-02}	1.00_{-02}
5496	7.20_{-01}	3.90_{-01}	1.00_{-02}	1.00_{-02}	1.00_{-02}	1.50_{-02}	1.00_{-02}	2.00_{-02}
8998	1.31_{+00}	6.40_{-01}	3.00_{-02}	2.00_{-02}	1.50_{-02}	3.00_{-02}	2.00_{-02}	3.00_{-02}
13982	2.42_{+00}	1.00_{+00}	5.00_{-02}	3.00_{-02}	3.00_{-02}	4.00_{-02}	3.00_{-02}	5.00_{-02}
24929	7.19_{+00}	1.93_{+00}	1.90_{-01}	1.25_{-01}	1.20_{-01}	1.45_{-01}	7.00_{-02}	1.30_{-01}
45036	3.55_{+01}	3.36_{+00}	3.30_{-01}	2.50_{-01}	2.20_{-01}	2.40_{-01}	1.40_{-01}	1.50_{-01}
78325	1.16_{+02}	5.71_{+00}	4.60_{-01}	3.00_{-01}	3.50_{-01}	3.30_{-01}	1.70_{-01}	2.20_{-01}
145060	4.17_{+02}	1.04_{+01}	7.25_{-01}	4.75_{-01}	4.80_{-01}	6.10_{-01}	2.30_{-01}	4.00_{-01}
252679	1.19_{+03}	1.80_{+01}	1.19_{+00}	7.05_{-01}	6.85_{-01}	1.00_{+00}	3.90_{-01}	7.35_{-01}
460342	4.10_{+03}	3.30_{+01}	2.08_{+00}	1.18_{+00}	1.13_{+00}	1.56_{+00}	7.10_{-01}	1.27_{+00}
816727	1.21_{+04}	5.86_{+01}	3.76_{+00}	2.08_{+00}	1.97_{+00}	2.64_{+00}	1.24_{+00}	2.31_{+00}
1395226	–	1.00_{+02}	6.54_{+00}	3.53_{+00}	3.36_{+00}	4.42_{+00}	2.11_{+00}	3.83_{+00}
2528443	–	1.82_{+02}	1.23_{+01}	6.39_{+00}	6.08_{+00}	7.95_{+00}	3.84_{+00}	6.79_{+00}
4305409	–	3.11_{+02}	2.22_{+01}	1.11_{+01}	1.04_{+01}	1.34_{+01}	6.60_{+00}	1.23_{+01}

TABLE 1. Absolute computational times for Example 6.1 and uniform (top) resp. adaptive (bottom) mesh-refinement. Here, uniform refinement means that all elements are marked for refinement. We use newest vertex bisection, and marking of an element corresponds to marking all of its edges, i.e. marked elements are refined by three bisections. The last column gives the time for the refinement from M_ℓ to $M_{\ell+1}$ elements, i.e., it takes about 2 minutes to refine the mesh $\mathcal{T}_6^{\rm unif}$ with $M_6^{\rm unif} = 3.145.728$ elements uniformly to obtain a mesh $\mathcal{T}_7^{\rm unif}$ with $M_7^{\rm unif} = 12.582.912$ elements. Note that, for instance, the assembly of the Galerkin data for the uniform mesh $\mathcal{T}_6^{\rm unif}$ can be done in about 8 seconds (optimized2), whereas the naive assembly of [2] from Listing 1 already takes more than 2 hours for a uniform mesh with $M_5^{\rm unif} = 786.432$ elements (slow). For adaptive mesh-refinement we use the algorithm implemented in Listing 9 with the residual-based error estimator η_R from Listing 10 for marking. We stress that in any case, the computation of η_R is faster than building the Galerkin data, whence the computation of the discrete solution $U \in \mathcal{S}^1_D(\mathcal{T}_\ell)$.

Thus, one is interested in the quotients

$$\alpha_1(M) := \frac{t(M)}{M} \quad \text{resp.} \quad \alpha_2(M) := \frac{t(M)}{M^2}, \tag{6.2}$$

where $t(M)$ is the runtime for a computation on a mesh with M elements. For a code of exact linear order the value $\alpha_1(M)$ is constant for all M. Analogously, routines with quadratic order have approximately constant $\alpha_2(M)$.

	relative assembly time $\alpha_2(N)$ resp. $\alpha_1(N)$				relative computational time $\alpha_1(N)$			
N	slow	medium	optimized	optimized2	η_R	η_H	η_Z	NVB
3072	4.34_{-08}	7.16_{-05}	3.26_{-06}	3.26_{-06}	3.26_{-06}	3.26_{-06}	3.26_{-06}	3.26_{-06}
12288	1.21_{-08}	7.12_{-05}	3.26_{-06}	2.44_{-06}	1.63_{-06}	3.26_{-06}	2.44_{-06}	4.88_{-06}
49152	1.51_{-08}	7.48_{-05}	6.92_{-06}	5.29_{-06}	4.68_{-06}	4.48_{-06}	2.95_{-06}	3.87_{-06}
196608	1.60_{-08}	7.21_{-05}	4.86_{-06}	3.03_{-06}	2.77_{-06}	3.87_{-06}	1.53_{-06}	3.81_{-06}
786432	1.56_{-08}	7.25_{-05}	4.77_{-06}	2.70_{-06}	2.40_{-06}	3.22_{-06}	1.54_{-06}	3.84_{-06}
3145728	–	7.28_{-05}	5.12_{-06}	2.69_{-06}	2.39_{-06}	3.10_{-06}	1.54_{-06}	3.83_{-06}

	relative assembly time $\alpha_2(N)$ resp. $\alpha_1(N)$				relative computational time $\alpha_1(N)$			
N	slow	medium	optimized	optimized2	η_R	η_H	η_Z	NVB
3072	4.34_{-08}	7.16_{-05}	3.26_{-06}	3.26_{-06}	3.26_{-06}	3.26_{-06}	3.26_{-06}	3.26_{-06}
3134	4.48_{-08}	7.02_{-05}	3.19_{-06}	3.19_{-06}	1.60_{-06}	3.19_{-06}	0.00_{+00}	3.19_{-06}
3584	3.74_{-08}	6.98_{-05}	2.79_{-06}	2.79_{-06}	1.40_{-06}	2.79_{-06}	2.79_{-06}	2.79_{-06}
5496	2.38_{-08}	7.10_{-05}	1.82_{-06}	1.82_{-06}	1.82_{-06}	2.73_{-06}	1.82_{-06}	3.64_{-06}
8998	1.62_{-08}	7.11_{-05}	3.33_{-06}	2.22_{-06}	1.67_{-06}	3.33_{-06}	2.22_{-06}	3.33_{-06}
13982	1.24_{-08}	7.08_{-05}	3.58_{-06}	2.15_{-06}	2.15_{-06}	2.86_{-06}	2.15_{-06}	3.58_{-06}
24929	1.16_{-08}	7.74_{-05}	7.62_{-06}	5.01_{-06}	4.81_{-06}	5.82_{-06}	2.81_{-06}	5.21_{-06}
45036	1.75_{-08}	7.46_{-05}	7.33_{-06}	5.55_{-06}	4.88_{-06}	5.33_{-06}	3.11_{-06}	3.33_{-06}
78325	1.88_{-08}	7.28_{-05}	5.87_{-06}	3.83_{-06}	4.47_{-06}	4.21_{-06}	2.17_{-06}	2.81_{-06}
145060	1.98_{-08}	7.19_{-05}	5.00_{-06}	3.27_{-06}	3.31_{-06}	4.21_{-06}	1.59_{-06}	2.76_{-06}
252679	1.87_{-08}	7.14_{-05}	4.71_{-06}	2.79_{-06}	2.71_{-06}	3.68_{-06}	1.54_{-06}	2.91_{-06}
460342	1.94_{-08}	7.16_{-05}	4.52_{-06}	2.56_{-06}	2.45_{-06}	3.39_{-06}	1.54_{-06}	2.76_{-06}
816727	1.82_{-08}	7.18_{-05}	4.60_{-06}	2.55_{-06}	2.41_{-06}	3.23_{-06}	1.52_{-06}	2.83_{-06}
1395226	–	7.18_{-05}	4.69_{-06}	2.53_{-06}	2.41_{-06}	3.17_{-06}	1.51_{-06}	2.75_{-06}
2528443	–	7.19_{-05}	4.85_{-06}	2.53_{-06}	2.40_{-06}	3.15_{-06}	1.52_{-06}	2.69_{-06}
4305409	–	7.22_{-05}	5.16_{-06}	2.59_{-06}	2.41_{-06}	3.10_{-06}	1.53_{-06}	2.85_{-06}

TABLE 2. Relative computational times for Example 6.1 and uniform (top) as well as adaptive mesh-refinement (bottom). Note that the second column shows $\alpha_2(M) = t(M)/M^2$ and proves that the assembly of the Galerkin data in Listing 1 is of quadratic complexity. The remaining columns show $\alpha_1(M) = t(M)/M$ and prove that proper use of `sparse` leads to (almost) linear complexity instead.

These relative runtimes are shown in Table 2. As can be expected from Section 3.2–3.3, the naive assembly (slow) of the Galerkin data from [2] yields quadratic dependence, whereas all remaining codes are empirically proven to be of linear complexity.

Note that Tables 1–2 do not include the time for the solution of the Galerkin system by use of MATLAB's backslash operator. In Figure 9, we plot the computational time for one loop of the adaptive algorithm in Listing 9 (Lines 4–19). For a given mesh $\mathcal{T}_\ell$, this includes the computation and solution of the Galerkin system, the computation of the residual-based error estimator η_R, the marking of the elements by use of the Dörfler marking (5.2) with $\varrho = 0.25$, and the refinement of the marked elements by use of newest vertex bisection. For the assembly of the Galerkin data, we again consider the four variants (slow, medium, optimized, optimized2) implemented in Listing 1 (slow), Listing 2 (medium), and Listing 3 (optimized, optimized2). We observe that the assembly of the Galerkin data dominates the computational time and that the direct solution of the sparse Galerkin system appears to be of (almost) linear complexity. Therefore, the improved implementations (medium, optimized, optimized2) provide a code, whose overall computational complexity only grows linearly.

Finally, we compare uniform and adaptive mesh-refinement, where we plot the error in the energy norm $\|\nabla(u-U)\|_{L^2(\Omega)}$ over the computational time. Here, the error is computed with

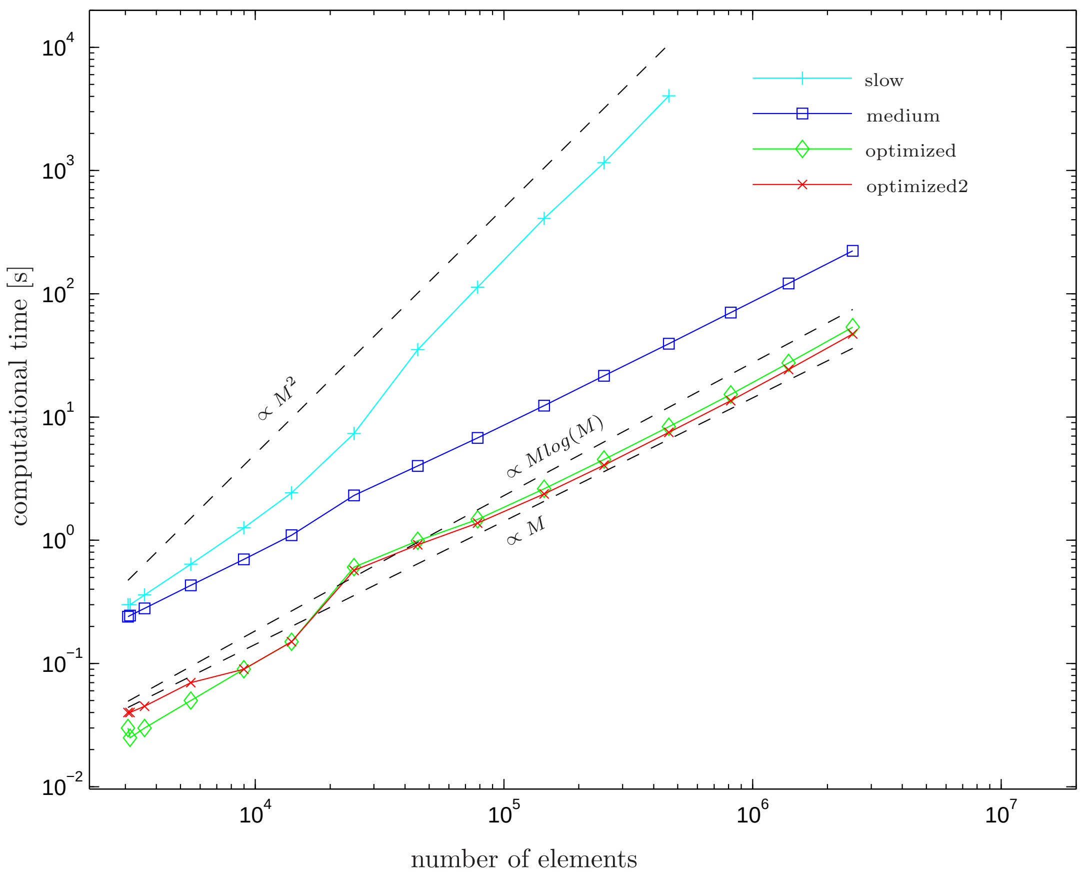


FIGURE 9. Overall computational time in Example 6.1 for one loop of the adaptive algorithm from Listing 9 (Lines 4–19): For the assembly of the Galerkin data, we use the implementations of Listing 1 (slow), Listing 2 (medium), and Listing 3 (optimized, optimized2). We see that the assembly of the Galerkin data dominates the computational time. Moreover, the improved assembly (medium, optimized, optimized2) only lead to (almost) linear growth of the computational time.

the help of the Galerkin orthogonality which provides

$$\|\nabla(u-U)\|_{L^2(\Omega)} = \big(\|\nabla u\|^2_{L^2(\Omega)} - \|\nabla U\|^2_{L^2(\Omega)}\big)^{1/2}. \tag{6.3}$$

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

$$\|\nabla U\|^2_{L^2(\Omega)} = \mathbf{x}\cdot\mathbf{A}\mathbf{x}. \tag{6.4}$$

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

$$\|\nabla u\|^2_{L^2(\Omega)} \approx 6.261224 \tag{6.5}$$

which is used to compute the error (6.3) for uniform as well as for adaptive mesh-refinement.

Figure 10 plots the Galerkin error in dependence of the computational time with respect to different mesh-refinement strategies. First, we consider uniform mesh-refinement based on either a red-green-blue strategy (RGB) or newest vertex bisection (NVB). To allow a fair comparison with the adaptive strategies, the times plotted are computed as follows: For the ℓ-th entry in the plot, the computational time $t^{\rm unif}_\ell$ is the sum of

- the time for $\ell-1$ successive uniform refinements,

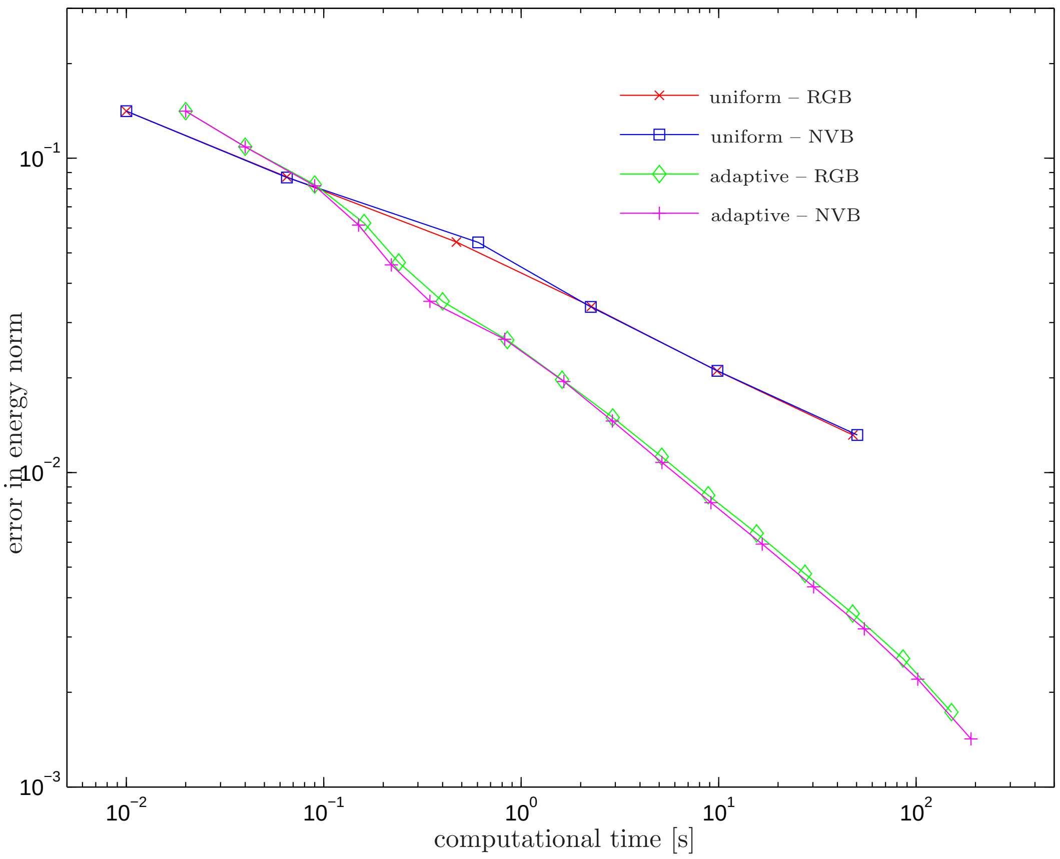


FIGURE 10. Galerkin error in Example 6.1 with respect to computational time for different mesh-refinement strategies: We consider uniform mesh-refinement by use of a red-green-blue strategy (RGB) or newest vertex bisection (NVB) as well as the adaptive algorithm from Listing 9, where marked elements are either refined by RGB or NVB refinement. For the uniform strategies, we only measure the computational time for ℓ uniform mesh-refinements plus one assembly and solution of the Galerkin system. For the adaptive strategies, 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 strategies are much superior to uniform mesh-refinement.

- 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. Contrary to that, the adaptive algorithm from Listing 9 with $\varrho = 0.25$ constructs a sequence of successively refined meshes, where $\mathcal{T}_{\ell+1}$ is obtained by local refinement of $\mathcal{T}_\ell$ based on the discrete solution U_ℓ. We therefore define the computational time $t_\ell^{\rm adap}$ for adaptive mesh-refinement in a different way: We set $t_0^{\rm adap} := 0$. Starting with the initial mesh $\mathcal{T}_1$ with $M_1 = 3.072$ elements, the computational time $t_\ell^{\rm adap}$ is the sum of

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

Independently on the precise refinement, namely RGB or NVB, we observe that the computational overhead in the adaptive computations is negligible even if we aim for an approximation

```
Set $\mathcal{T}_{1,0} = \mathcal{T}_0$, $n = 1$, $t = 0$                         Initialization
Do while $t \le t_{max}$                                                          Time loop
    $k = 0$
    Do                                                                            Loop for adaptive mesh-refinement
        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 $\eta_n^2 := \sum_{T\in\mathcal{T}_{n,k}} \eta_T^2$    Error estimator
        If $\eta_n > \tau$                                                        Adaptive mesh-refinement
            Use Dörfler criterion (5.2) to mark
                elements for refinement
            Refine marked elements by NVB to
                obtain a 'finer' triangulation $\mathcal{T}_{n,k+1}$
            Update $k \mapsto k+1$
        End If
    While $\eta_n > \tau$                                                         Solution $U_{n,k}$ is accurate enough
    Set $U_n := U_{n,k}$, $\mathcal{T}^*_{n,k} = \mathcal{T}_{n,k}$
    Do                                                                            Loop for adaptive mesh-coarsening
        Update $k \mapsto k-1$
        For all $T \in \mathcal{T}^*_{n,k+1}$ compute error indicators $\eta_T$  Refinement (resp. coarsening) indicator
        Mark elements $T$ for coarsening
            provided $\eta_T \le \sigma\,\tau/\#\mathcal{T}^*_{n,k+1}$
        Generate a 'coarser' triangulation $\mathcal{T}^*_{n,k}$
            by coarsening marked elements
        If $\mathcal{T}^*_{n,k} \ne \mathcal{T}^*_{n,k+1}$
            Compute discrete solution $U_{n,k}$ on mesh $\mathcal{T}^*_{n,k}$
        End if
    While $\mathcal{T}^*_{n,k} \ne \mathcal{T}^*_{n,k+1}$                         Mesh cannot be coarsened furthermore
    Set $\mathcal{T}_{n+1,0} = \mathcal{T}^*_{n,k+1}$
    Update $n := n+1$, $t := t + \Delta t$                                        Go to next time step
End Do
```

Table 3. Adaptive algorithm with refinement and coarsening used for the quasi-stationary Example 6.2.

U with low accuracy $\|\nabla(u-U)\|_{L^2(\Omega)} \approx 5/100$. Within 100 seconds, our Matlab code for AFEM computes an approximation with accuracy $\|\nabla(u-U^{\rm adap})\|_{L^2(\Omega)} \approx 1/1000$, whereas uniform mesh-refinement would only lead to $\|\nabla(u-U^{\rm unif})\|_{L^2(\Omega)} \approx 1/100$ 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.

6.2. Quasi-Stationary Model Problem. In the second example, we consider a homogeneous Dirichlet problem (2.1) with $\Gamma_D = \partial\Omega$ on the domain $\Omega = (0,3)^2 \setminus [1,2]^2$, cf. Figure 11. 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 5.2 satisfies $\eta_R \le \tau$ for a given tolerance $\tau > 0$. Instead of starting always from the coarsest mesh $\mathcal{T}_0$, we use the adaptive algorithm from Table 3 which allows adaptive mesh-refinement as well as mesh-coarsening. For the refinement, we use the Dörfler criterion (5.2) with parameter $\varrho \in (0,1)$. For the coarsening process, we mark those elements $T \in \mathcal{T}$, which satisfy $\eta_T \le \sigma\,\eta_R/\#\mathcal{T}$ for some given parameter $\sigma \in (0,1)$. Hence, we try to get some equal distribution of the residual for each triangle $T \in \mathcal{T}$. We stop our coarsening process if none of the marked elements can be modified by our procedure described in Section 4.4.

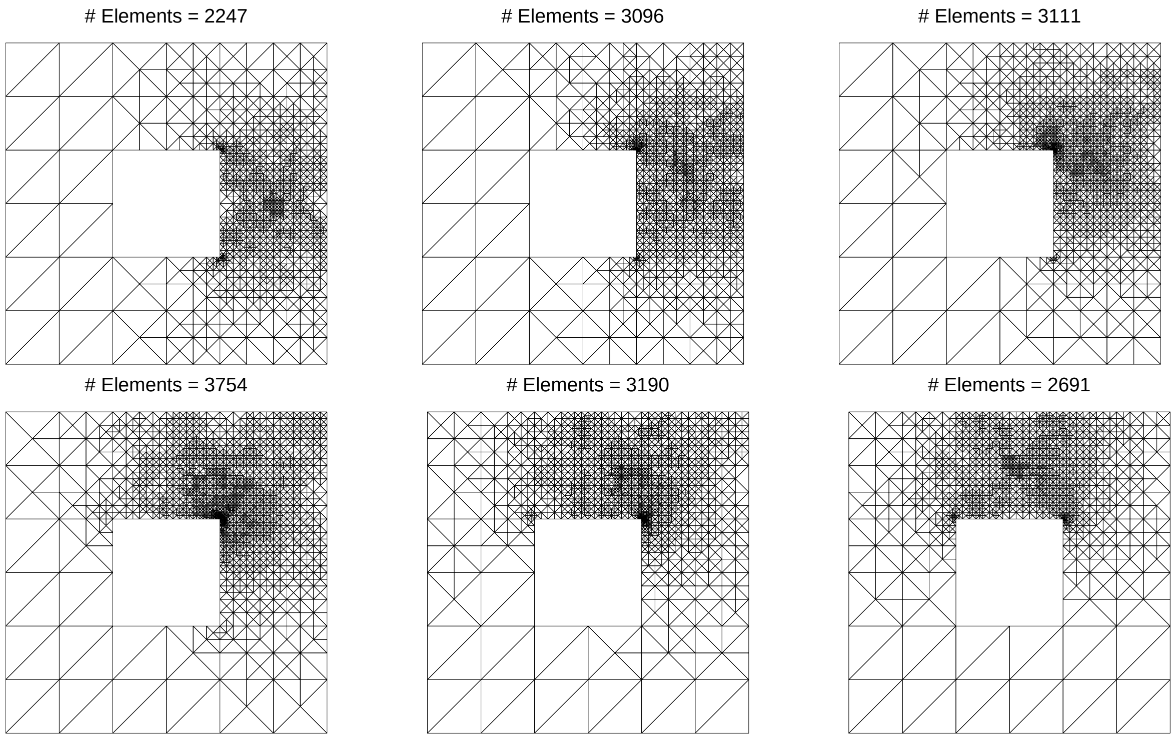


FIGURE 11. Adaptively generated meshes $\mathcal{T}_n$ at different time steps $n = 1, 11, 21, 31, 41, 51$ for the quasi-stationary Example 6.2, where we used the adaptive algorithm from Table 3 with tolerance $\tau = 0.001$ and parameters $\sigma = 0.25$ and $\varrho = 0.25$.

For our numerical experiment, we choose the tolerance $\tau = 0.001$ as well as the parameters $\sigma = 0.25$ for adaptive mesh-coarsening and $\varrho = 0.25$ for adaptive mesh-refinement. A sequence of adapted meshes is shown in Figure 11 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 12 we plot the evolution of the number of elements. The upper curve shows the number of elements to satisfy the condition $\eta_R \leq \tau$ for each time step, while the lower graph gives the number 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. Besides the first mesh, the algorithm needs at most 2 steps of refinement or coarsening for each time step to satisfy certain properties as mentioned above. Hence, a refinement-coarsening strategy as considered here is much faster than computing an adaptive mesh always starting from the coarsest mesh $\mathcal{T}_0$.

References

[1] M. Abramowitz, I. A. Stegun: *Handbook of Mathematical Functions*, Dover, New York, 1972,

[2] J. Alberty, C. Carstensen, S. Funken: *Remarks around 50 Lines of* Matlab*: Short Finite Element Implementation*, Numer. Algorithms 20 (1999), 117–137.

[3] S. Bartels, C. Carstensen: *Each Averaging Technique Yields Reliable Error Control in FEM on Unstructured Grids, Part I: Low Order Conforming, Nonconforming, and Mixed FEM*, Math. Comput. 71 (239), 945–969.

[4] I. Babuška, A. Miller: *A Feedback Finite Element Method with A Posteriori Error Control, Part I*, Comput. Methods Appl. Mech. Engrg 61 (1987), 1–40.

[5] R. Bank, K. Smith: *A Posteriori Error Estimates Based on Hierarchical Bases*, SIAM J. Numer. Anal. 30 (1993), 921–935.

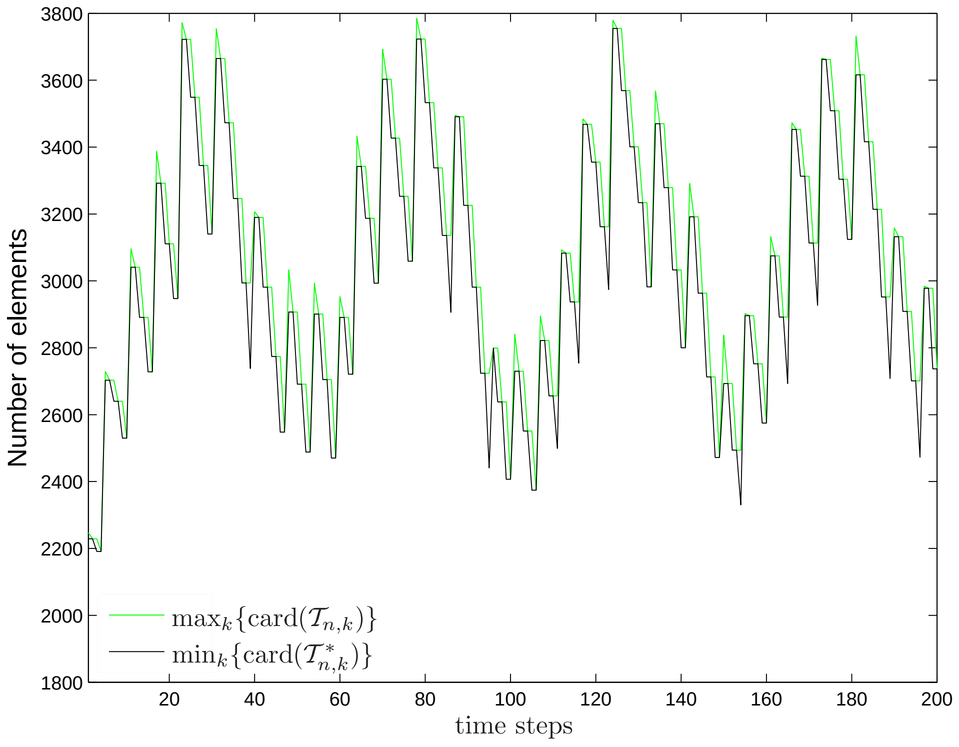


FIGURE 12. Number $\max_k \#\mathcal{T}_{n,k}$ of elements in quasi-stationary Example 6.2 after refinement resp. number $\min_k \#\mathcal{T}^*_{n,k}$ of elements after coarsening for all time steps $n = 1, 2, 3, \ldots, 200$, where we used the adaptive algorithm from Table 3 with tolerance $\tau = 0.001$ and parameters $\sigma = 0.25$ and $\varrho = 0.25$.

[6] R. Barrett, M. Berry, T. Chan, et al.: *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia ^{2}1994.

[7] F. Bornemann, B. Erdmann, R. Kornhuber: *A Posteriori Error Estimates for Elliptic Problems in Two and Three Space Dimensions*, SIAM J. Numer. Anal. 33 (1996), 1188-1204.

[8] C. Carstensen: *All First-Order Averaging Techniques for A Posteriori Finite Element Error Control on Unstructured Grids are Effective and Reliable*, Math. Comput. 73 (2004), 1153–1165.

[9] J. Cascon, C. Kreuzer, R. Nochetto, K. Siebert: *Quasi-Optimal Convergence Rate for an Adaptive Finite Element Method*, Preprint, University of Augsburg 2007.

[10] L. Chen: *Short Bisection Implementation in* Matlab, Research Notes, University of Maryland 2006.

[11] L. Chen, C. Zhang: *AFEM@*Matlab*: A* Matlab *Package of Adaptive Finite Element Methods*, Technical Report, University of Maryland 2006.

[12] L. Chen, C. Zhang: *A Coarsening Algorithm and Multilevel Methods on Adaptive Grids by Newest Vertex Bisection*, Preprint, University of California, Irvine 2007.

[13] W. Dörfler: *A Convergent Adaptive Algorithm for Poisson's Equation*, SIAM J. Numer. Anal. 33 (1996), 1106–1129.

[14] J. Gilbert, C. Moler, R. Schreiber: *Sparse Matrices in* Matlab*: Design and Implementation*, SIAM J. Matrix Anal. Appl. 13 (1992), 333–385.

[15] S. Funken, D. Praetorius, P. Wissgott: *Efficient Matlab Implementation of P1-AFEM*, Software download at `http://www.asc.tuwien.ac.at/∼dirk/matlab`

[16] S. Funken, D. Praetorius, P. Wissgott: *Efficient Implementation of Matlab's Sparse Command for FEM Matrices*, Technical Report, Institute for Numerical Mathematics, University of Ulm, 2008.

[17] P. Morin, K. Siebert, A. Veeser: *A Basic Convergence Result for Conforming Adaptive Finite Elements* M3AS 18 (2008), 707–737.

[18] I. Rosenberg, F. Stenger: *A Lower Bound on the Angles of Triangles Constructed by Bisecting the Longest Edge*, Math. Comput. 29 (1975), 390–395.

[19] H.R. Schwarz: *Finite Element Methods*, Academic Press, London, 1988.

[20] E. Sewell: *Automatic Generation of Triangulations for Piecewise Polynomial Approximations*, Ph.D. thesis, Purdue University, West Lafayette 1972.

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

[22] P. Wissgott: *A Space-Time Adaptive Algorithm for Linear Parabolic Problems*, Diploma thesis, Institute for Analysis and Scientific Computing, Vienna University of Technology, 2007.

[23] O. Zienkiewicz, J, Zhu: *A Simple Error Estimator and Adaptive Procedure for Practical Engineering Analysis*, Internat. J. Numer. Methods Engrg. 24 (1987), 337–357.

Institute for Numerical Mathematics, University of Ulm, Helmholtzstrasse 18, D-89069 Ulm, Germany

E-mail address: `Stefan.Funken@uni-ulm.de`

Institute for Analysis and Scientific Computing, Vienna University of Technology, Wiedner Hauptstrasse 8-10, A-1040 Wien, Austria

E-mail address: `Dirk.Praetorius@tuwien.ac.at` (corresponding author)