Ameshref: A Matlab-Toolbox for Adaptive Mesh Refinement in Two Dimensions

Stefan A. Funken and Anja Schmidt

Abstract This paper presents a MATLAB-Toolbox named `ameshref` that provides an efficient implementation of various adaptive mesh refinement strategies allowing triangular and quadrilateral grids with and without hanging nodes. For selected methods, we give an insight into the strategy itself and the core ideas for an efficient realization. This is achieved by utilization of reasonable data structure, use of MATLAB built-in functions and vectorization. To serve educational purposes on how to implement a method efficiently, the code is kept accessible but short. Numerical experiments underline the efficiency of the code and show the flexible deployment in different contexts where adaptive mesh refinement is in use. Our implementation is accessible and easy-to-understand and thus considered to be a valuable tool in research and education.

1 Introduction and Outline

Keeping computational cost low while maintaining a good accuracy in solving partial differential equations (PDEs) can be achieved by adaptively refining the underlying domain. For this and more general applications in mind, we present an efficient implementation of nine adaptive mesh refinement strategies in two dimensions. Different grid refinement methods have been proposed and investigated over the last few decades and various scientists have provided public code for one or another method; see [13] for an overview and list of public code and, in particular, [5, 6, 9] for MATLAB implementations. Nonetheless, existing mesh refinement tools are often inaccessible. Commercial software packages act as a "black box" and open source codes are mostly too complex to be understood by a wider audience. Furthermore, the mesh refinement is often just one step in a series and by the use of an external mesh refinement software, it is cumbersome to integrate this step with

S. A. Funken · A. Schmidt (✉)
Institut für Numerische Mathematik, Universität Ulm, Ulm, Germany
e-mail: stefan.funken@uni-ulm.de; anja.schmidt@uni-ulm.de

V. A. Garanzha et al. (eds.), *Numerical Geometry, Grid Generation and Scientific Computing*, Lecture Notes in Computational Science and Engineering 131,
https://doi.org/10.1007/978-3-030-23436-2_20

other implementations. In addition, most of these tools only provide a triangulation of the region into triangles. However, for some applications like the evaluation of stress fields or in computational fluid dynamics, it is beneficial to provide the geometric data as a grid of quadrilaterals [16]. Thus, in this work we provide an accessible mesh refinement implementation in MATLAB that offers flexibility in the refinement step. The data structure is kept simple by only requiring the element-connectivity and coordinates of the vertices as input data. In contrast to other implementations, we do not follow a recursive approach and prefer the realization in terms of vectorization. Unlike using a vertex-based marking strategy as, e.g., in [12], we understand a marking as an edge-based marking.

The following mesh refinement strategies for triangular and quadrilateral elements are realized in our MATLAB `ameshref`-package: For triangular meshes, the *red-green (TrefineRG)* refinement proposed by Bank and Sherman in [2], the *newest vertex bisection (TrefineNVB)* first mentioned by Sewell in [14], and the *red-green-blue (TrefineRGB)* refinement method using reference edges as discussed by Carstensen in [4]. A further *red (TrefineR)* refinement strategy emerges naturally from the *red-green* refinement if hanging nodes are allowed. For quadrilateral meshes, there also exists a *red (QrefineR)* refinement strategy, precisely a refinement by quadrisection, touched upon from Verfürth in [15]. Furthermore, Bank et al. proposed a *red-green (QrefineRG)* algorithm to eliminate hanging nodes arising by quadrisection [3]. A further *red-blue (QrefineRB)* strategy is investigated which was inspired by a work of Kobbelt [10]. Mao et al. presented a refinement method based on enneasection (division of a quadrilateral in nine smaller quadrilaterals) [16] which we call *red2-green2 (QrefineRG2)* refinement strategy in this work. Similarly, for this method the *red2 (QrefineR2)* refinement strategy naturally arises if irregular grids are allowed. An overview of the just mentioned methods is illustrated in Fig. 1.

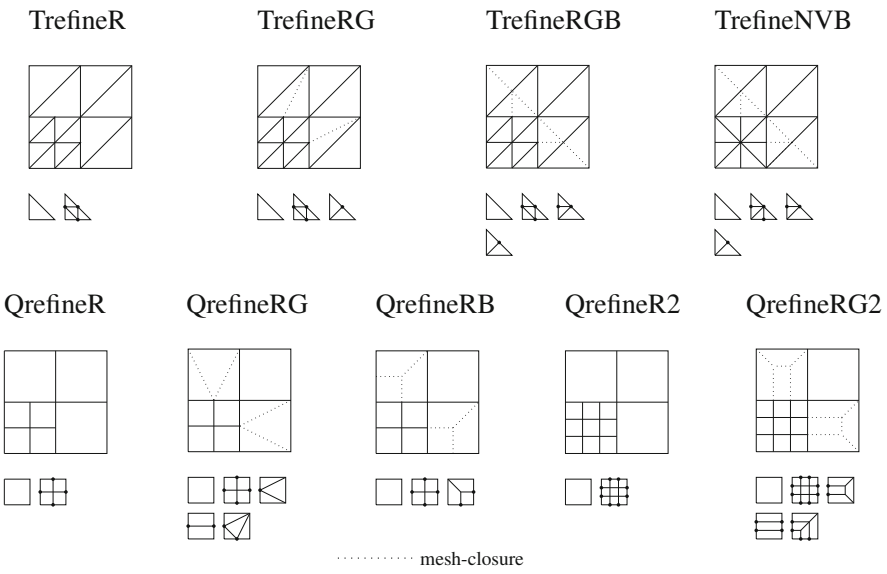


Fig. 1 Overview of refinement strategies implemented in our MATLAB-toolbox

Our realization of mesh refinement strategies follows the general procedure

MARK – CLOSURE – CREATE NODES – CREATE ELEMENTS.

We understand the first step **MARK** in the following way: elements are marked by flagging each edge of the element for bisection or trisection, respectively. Hence, also non-marked elements can be affected by neighboring elements. The proceeding step **CLOSURE** makes sure that either hanging nodes are removed or the level of two neighboring elements differs at most by one as is needed for irregular mesh refinement strategies. We provide a detailed explanation of how to realize the first two steps in the course of this work. The last two steps are self-explanatory and are not addressed in this work.

The complete MATLAB code of `ameshref` can be downloaded from the web [8], and the technical report provides a detailed documentation of the underlying ideas [11]. We restrict our discussions to two representative methods *QrefineRB* and *TrefineR* to explain the main ideas; for a short but complete version on all nine refinement methods we recommend [7].

The rest of this paper is organized as follows. First of all, we introduce the methods QrefineRB and TrefineR. Subsequently in Sect. 3, we present the data structure used for a grid and state the main ideas for an efficient realization. This work closes with some numerical experiments discussed in Sect. 4.

2 Mesh Refinement Strategies

We consider a polygonal Lipschitz domain Ω in $\mathbb{R}^2$. Let therefore $\mathscr{T}$ be a *triangulation*, i.e., $\mathscr{T}$ is a finite set of compact elements T with positive area $|T| > 0$, the union of all elements in $\mathscr{T}$ covers the closure $\overline{\Omega}$ and for two elements $T_1, T_2 \in \mathscr{T}$ with $T_1 \neq T_2$ holds $\mathring{T}_1 \cap \mathring{T}_2 = \emptyset$, where $\mathring{T}$ denotes the interior of T. Here, we restrict the elements T to be of triangular or quadrilateral shape. In addition, we call $\mathscr{T}$ a *regular triangulation* of Ω if for all $T_1, T_2 \in \mathscr{T}$ with $T_1 \neq T_2$ holds that $T_1 \cap T_2$ is empty, a common node or a common edge. This definition prevents a triangulation from having *hanging nodes*. If $z \in T_1 \cap T_2$ is a vertex of T_1 but not of T_2, we call z a hanging node.

As a first method, we present *TrefineR*, a refinement strategy for triangles with one red refinement pattern. This pattern is formed by connecting the midpoints of each edge with each other; see Fig. 2. The four emerging triangles are geometrically similar to their ancestor [15]. If this pattern is used adaptively, hanging nodes can not be eliminated without refining the mesh uniformly. For this reason, we allow hanging nodes in this method but restrict the hanging nodes per edge to be one by the 1-Irregular Rule proposed in [3]: *Refine any unrefined element that has more than one hanging node on an edge.*

Fig. 2 Refinement patterns in TrefineR

Fig. 3 Red and blue patterns

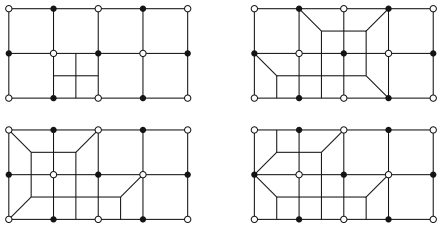


Fig. 4 Non-uniqueness of QrefineRB without any further assumptions. Top left to bottom right: red refinement and possible eliminations of hanging nodes by using blue patterns

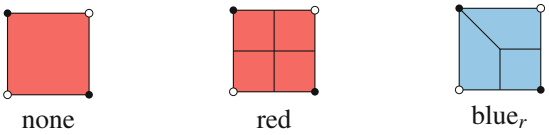


Fig. 5 Convention of placing the patterns in relation to black and white nodes

The second refinement method to showcase is *QrefineRB* with a red and blue refinement pattern shown in Fig. 3 where the blue pattern resembles a Y-formation. We seek a blue pattern that regularizes a mesh by eliminating hanging nodes.

However, due to the blue pattern, by repairing one hanging node, another hanging node is introduced, i.e., hanging nodes are shifted to an adjacent edge. Only connecting two hanging nodes leads to a regular mesh. To resolve this issue we utilize this pattern to surround a node until another is reached and thus hanging nodes are eliminated [10]. There still remains the question about the realization of this method, i.e., how to uniquely place the blue pattern.

Without any further assumptions on how to place the blue pattern, this refinement strategy does not lead to a unique refinement. For an illustration of the non-uniqueness, the nodes of an exemplary mesh are painted alternately in black and white. Originating from a red refinement shown in the top left of Fig. 4, the subsequent meshes are some possible closures, namely placing the Y-formation around black nodes, around white nodes or a mixture of both. To guarantee uniqueness, we stick to the convention of placing the Y-formation around black nodes, i.e., we only allow the patterns in relation to black and white nodes depicted in Fig. 5.

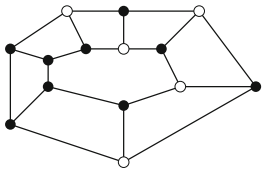


Fig. 6 Counterexample for coloring the nodes alternately. Nodes cannot be painted alternately in two different colors due to the odd number of nodes forming a ring

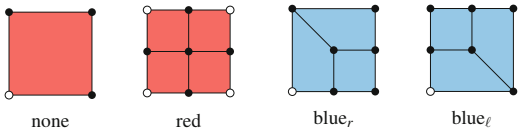


Fig. 7 A red element and its refinement patterns in QrefineRB

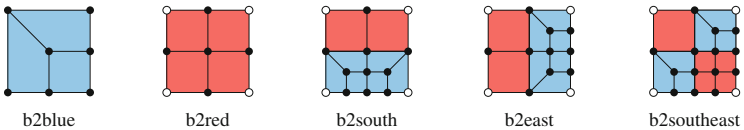


Fig. 8 A blue element and its refinement patterns in QrefineRB

To realize this method, painting the nodes alternately in black and white and subsequently matching the patterns would be an obvious approach. However, in practice coloring the nodes alternately can be of great effort and supplementary there exist meshes for which an alternating coloring can not be achieved; see Fig. 6.

Thus, we follow Kobbelt's proposal to first refine the mesh uniformly [10], i.e., by red-refining each element. Instead of coloring the nodes alternately, we make use of the property that after one refinement the nodes of an element can be assigned to two generations. We store the oldest node of each element as reference node and consider this reference node as one of the white nodes and therefore, we allow the refinement patterns in relation to the reference node shown in Fig. 7. Since we only deal with one reference node, we have to take into account a blue$_r$ and blue$_\ell$ refinement pattern. However, a distinction is only needed to cover all cases for the refinement of a red element. Afterwards, a blue element is refined as depicted in Fig. 8—no distinction into blue$_r$ and blue$_\ell$ is needed.

To avoid degenerated quadrilaterals, a blue element is coarsened before further refined; see Fig. 8 for a blue element and its refinement patterns.

3 Implementation of Adaptive Mesh Refinement

We first present the used data structure of a triangulation $\mathscr{T}$. Subsequently, the steps **MARK** and **CLOSURE** are explained in more detail. For the latter, we make use of two different implementation approaches, namely a *hash map* concept for QrefineRB and an implementation by means of *virtual elements* for TrefineR. The concepts translate with some minor changes for the other refinement strategies not investigated in this work. For the interested reader an extensive presentation is provided in [11].

3.1 Data Structure of a Triangulation

To represent the data, we need to specify the triangulation $\mathscr{T} = \{T_1, \ldots, T_M\}$, the corresponding set of nodes $\mathscr{N} = \{z_1, \ldots, z_n\}$ and potentially boundary data. We follow [1, 9] and define `coordinates` as a $N \times 2$ array with the ℓ-th row

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

representing the x- and y-coordinates of the ℓ-th node $z_\ell = (x_\ell, y_\ell) \in \mathbb{R}^2$. Furthermore, the triangulation $\mathscr{T}$ is stored in a $M \times 3$ array `elements3` for triangles, i.e., the ℓ-th element $T_\ell = \operatorname{conv}\{z_i, z_j, z_k\}$ is represented by

$$\texttt{elements3}(\ell,:) = [i \quad j \quad k].$$

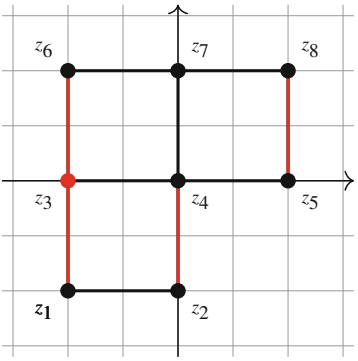
Analogously, a $M \times 4$ array `elements4` for quadrilaterals stores the ℓ-th quadrilateral element $T_\ell = \operatorname{conv}\{z_i, z_j, z_k, z_l\}$ as

$$\texttt{elements4}(\ell,:) = [i \quad j \quad k \quad l].$$

If boundary edges are given, the ℓ-th edge $E_\ell = \operatorname{conv}\{z_i, z_j\}$ corresponds to

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

Depending on the problem at hand, the naming and usage of the boundary conditions can be adjusted and are not limited to Dirichlet and Neumann boundaries. See an exemplary illustration of the introduced terms in Fig. 9. It is ensured that the order of the nodes in a triangulation is given in a mathematical positive sense.



coordinates

1	-1.0	-1.0
2	0.0	-1.0
3	-1.0	0.0
4	0.0	0.0
5	1.0	0.0
6	-1.0	1.0
7	0.0	1.0
8	1.0	1.0

dirichlet

1	1	2
2	4	5
3	8	7
4	7	6

elements4

1	1	2	4	3
2	8	7	4	5
3	6	3	4	7

neumann

1	2	4
2	5	8
3	6	3
4	3	1

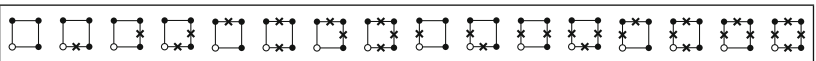
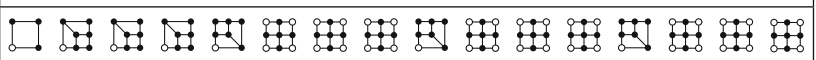
Fig. 9 A triangulation $\mathscr{T}$ of the L-shaped domain $\Omega = (-1,1)^2 \backslash ([0,1] \times [-1,0])$ into 3 quadrilaterals characterized by the arrays `coordinates` and `elements4`. The boundary edges of the L-shape are partitioned into Dirichlet (in black) and Neumann boundary (in red). This illustration is based on the data representation in [1, 9]

Using this data structure, we may visualize a grid by MATLAB's built-in function `patch('Faces', elements4, 'Vertices', coordinates, 'Facecolor', 'none')`. This works analogously for triangular meshes by replacing `elements4` by `elements3`.

3.2 MARK

As already stated, we understand a marked element as an element where each edge is marked for refinement. To this end, it is favorable to generate a numbering of the edges instead of searching data structure costly. This is accomplished by the function `provideGeometricData` which is an enhanced version of the one provided in [9]. This function returns the corresponding nodes in `edge2nodes(`ℓ`,:)`$= [i \quad j]$ for the ℓ-th edge $E_\ell = \operatorname{conv}\{z_i, z_j\}$ and the edges of the triangulation $\mathscr{T}$ in `element3edges` and `element4edges` respectively. `element3edges(i,`ℓ`)` provides the number of the edge between the nodes `elements3(i,`$\ell \mod 3+1$`)` and `elements3(i,`$(\ell+1) \mod 3+1$`)`. Analogously for `elements4edges` by replacing $\ell \mod 3$ by $\ell \mod 4$. Additionally, optional information about boundary edges can be queried. Remember, that for this refinement strategy a distinction into red and blue elements is necessary. To this end, by utilization of this function, all edges of a marked red element are flagged and furthermore, for a marked blue element, all edges of the father are flagged. A distinction between red and blue elements is easily made by storing them block by block in `elements4`.

Table 1 In- and output of the function `hashmap` for QrefineRB

mark															
bin	0000	1000	0100	1100	0010	1010	0110	1110	0001	1001	0101	1101	0011	1011	0111	1111
dec	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
hash																
bin	0000	1100	1100	1100	0011	1111	1111	1111	0011	1111	1111	1111	0011	1111	1111	1111
type	none	blue_r	blue_r	blue_r	blue_ℓ	red	red	red	blue_ℓ	red	red	red	blue_ℓ	red	red	red

3.3 CLOSURE

Once the edges of marked elements are flagged, further edges might need to be marked as well to properly map the marking to an admissible grid. Note, that further edges can always be marked whereas removing a marking is not considered.

3.3.1 Implementation with Hash Maps

Our first approach is based on hash maps. For this procedure we understand each edge as a binary, i.e., a flagged edge has value one and a non-flagged edge has value zero. We illustrate the procedure in Table 1 for QrefineRB. For a quadrilateral, there exist $2^4 = 16$ possibilities to mark edges. Note, that the corresponding binary number is given in a reverse order. For each marking it is clear by definition of the refinement strategy which pattern encoded in *hash* is the one to choose. This mapping is done automatically by the auxiliary function `hash2map` with input parameters `dec`, i.e., we specify the decimal number corresponding to the marking and the possible refinement patterns given by the refinement strategy in `hash`. This is done in a loop until no further markings are necessary. Remember that the assignment of a reference node is needed for this strategy. We have the convention to store the reference node of the ℓ-th element in `elements4(`ℓ`,1)`.

3.3.2 Implementation with Virtual Elements

Unlike for regular refinements, in the irregular case hanging nodes are not removed. However, to follow the 1-Irregular Rule for TrefineR, a distinction into regular and irregular edges is indispensable. To this end, we make use of virtual elements `irregular` as depicted in Fig. 10. Then, `irregular(`ℓ`,3)` gives the hanging node of the ℓ-th irregular edge. This convention is handy because if one of the two halves

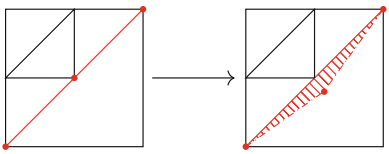


Fig. 10 Irregularity data as virtual elements. An irregular edge with a hanging node (left) can be interpreted as a virtual triangle (right)

of the irregular edge is marked, the unrefined neighbor element can be flagged for refinement, too.

4 Numerical Experiments

To demonstrate the efficiency of the developed MATLAB code, we provide some numerical experiments performed on an Apple MacBook Air with a 1.6 GHz Intel Core i5, a RAM of 8 GB 1600 MHz DDR3 on macOS 10.13.2 (High Sierra). Throughout, MATLAB version 9.2.0 (R2017a) is used.

For the first experiment, we consider a refinement along a circle that can be found as test example of the `ameshref`-package as `example1/`. We measure 15 times the computational time by use of MATLAB's built-in function `cputime` and plot the mean of the measured times for each strategy in Fig. 11. This plot shows an almost linear behavior between the number of elements and computational time. The non-linear behavior at the beginning of the refinement process is an overlay of MATLAB's precompiling process. A refined mesh with 10^7 elements is generated within 20–30 s on a standard laptop. The refined meshes for this example are depicted in Fig. 12.

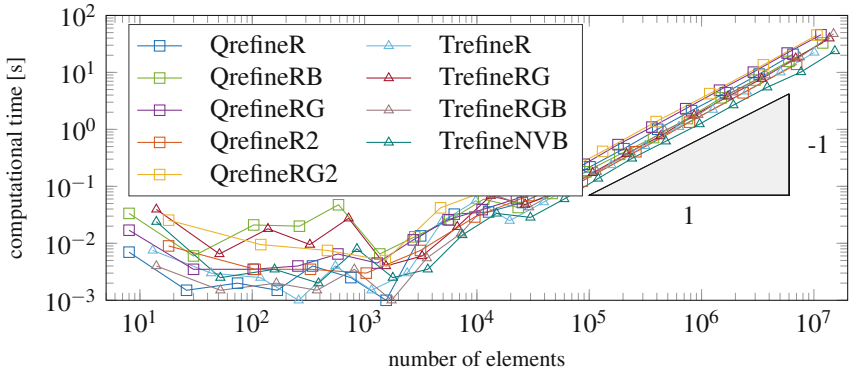


Fig. 11 Computational times for the adaptive mesh refinements over the number of elements. A nearly linear behavior between the number of elements and computational time evens out for all of our implementations

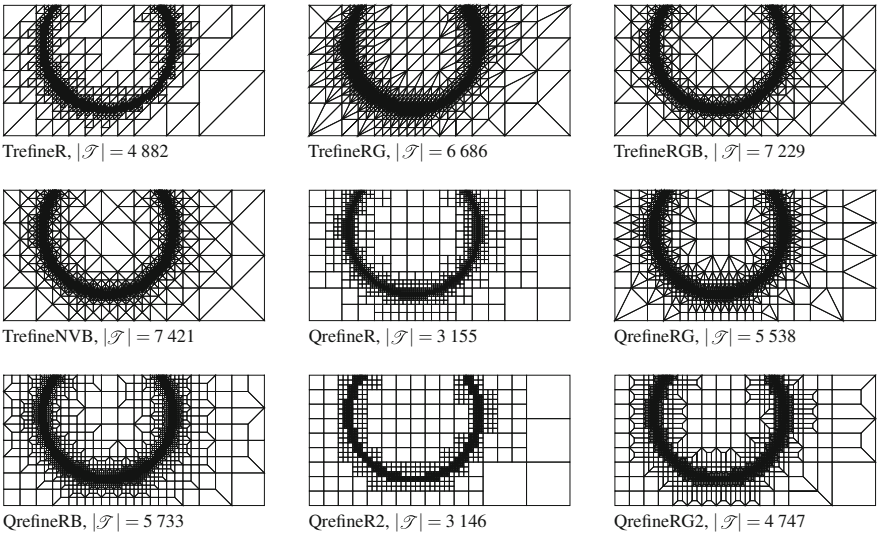


Fig. 12 Refinement along a circle using different strategies. Elements are marked for refinement if they intersect with a given circle. Below the triangulations, the used refinement method and the number of elements in the mesh are specified

References

1. Alberty, J., Carstensen, C., Funken, S.A.: Remarks around 50 lines of Matlab: short finite element implementation. Numer. Algorithm. **20**(2–3), 117–137 (1999)
2. Bank, R.E., Sherman, A.H.: An adaptive, multi-level method for elliptic boundary value problems. Computing **26**(2), 91–105 (1981)
3. Bank, R.E., Sherman, A.H., Weiser, A.: Some refinement algorithms and data structures for regular local mesh refinement. In: Stepleman, R., et al. (eds.) Scientific Computing, vol. 1, pp. 3–17. IMACS, North-Holland (1983)
4. Carstensen, C.: An adaptive mesh-refining algorithm allowing for an H^1-stable L^2-projection onto Courant finite element spaces. Constr. Approx. **20**(4), 549–564 (2004)
5. Chen, L.: iFEM: an innovative finite element methods package in MATLAB. Preprint, University of Maryland (2008)
6. Chen, L.: Short implementation of bisection in MATLAB. In: Adv. Comput. Math.: Selected Papers from the International Workshop on Computational Sciences and Its Education, pp. 318–332. World Scientific, Singapore (2008)
7. Funken, S.A., Schmidt, A.: Adaptive mesh refinement in 2D — an efficient implementation in MATLAB. Submitted to Comput. Methods Appl. Math. (2018)
8. Funken, S.A., Schmidt, A.: Ameshref – Efficient Implementation of Adaptive Mesh Refinement in 2D. Software download at https://github.com/aschmidtuulm/ameshref
9. Funken, S.A., Praetorius, D., Wissgott, P.: Efficient implementation of adaptive P1-FEM in Matlab. Comput. Methods Appl. Math. **11**(4), 460–490 (2011)
10. Kobbelt, L.: Interpolatory subdivision on open quadrilateral nets with arbitrary topology. In: Comput. Graph. Forum, vol. 15, pp. 409–420. Wiley, Edinburgh (1996)
11. Schmidt, A.: Adaptive mesh refinement in 2D — an efficient implementation in Matlab for triangular and quadrilateral meshes. Master's thesis, Universität Ulm (2018)

12. Schneiders, R.: Algorithms for quadrilateral and hexahedral mesh generation. Proc. VKI-LS Comput. Fluid Dyn. **4**, 31–43 (2000)
13. Schneiders, R.: Mesh generation and grid generation on the web. http://www.robertschneiders.de/meshgeneration/meshgeneration.html. Accessed 21 Aug 2018
14. Sewell, E.: Automatic Generation of Triangulations for Piecewise Polynomial Approximation. Purdue University, West Lafayette (1972)
15. Verfürth, R.: A Review of a Posteriori Error Estimation and Adaptive Mesh-Refinement Techniques. Wiley, Chichester (1996)
16. Zhao, X., Mao, S., Shi, Z.: Adaptive finite element methods on quadrilateral meshes without hanging nodes. SIAM J. Sci. Comput. **32**(4), 2099–2120 (2010)