

Ulm University
Department of Mathematics and Economics



ulm university universität
uulm

Adaptive Mesh Refinement in 2D - An Efficient Implementation in MATLAB for Triangular and Quadrilateral Meshes

Anja Schmidt

A thesis presented for the degree of
Master of Science in Mathematics
at Ulm University

Revised Version from September 13th, 2018

Supervisor

Prof. Dr. Stefan A. Funken

Examiner

Prof. Dr. Stefan A. Funken
Prof. Dr. Karsten Urban

It is not knowledge, but the act of
learning, not possession but the act
of getting there, which grants the
greatest enjoyment.

CARL FRIEDRICH GAUSS,
(1777–1855)

Acknowledgements.

I would like to express my gratitude to my supervisor Prof. Dr. Stefan A. Funken for his continuous support throughout the last three years. His door was always open to me and he took the time to discuss ideas and guide me with his vast knowledge. I am looking forward to a continuing cooperation as a PhD student under his supervision.

Furthermore, I want to thank Prof. Dr. Karsten Urban for the contribution to this work through further comments and kindly serving on my thesis committee.

I want to say thanks to my colleagues from the Institute of Numerical Mathematics at Ulm University for discussing ideas and making research more enjoyable.

Apart from this, my thanks also goes to the Talanx Foundation for a financial support during my Master's studies.

Last but not least, I am more than grateful for the continuous support from my mother, boyfriend, sister, grandmother and all of my friends. Without you in my life I would not have been able to achieve what I have already achieved. Thank you for your love!

Contents

1	Introduction	1
1.1	Adaptive Finite Element Method	1
1.2	Outline	2
2	General Framework	3
2.1	Model Problem	3
2.2	Galerkin Discretization of the Problem	4
2.3	Data Structures	5
2.4	SOLVE	7
2.5	ESTIMATE	8
2.6	MARK	9
2.7	REFINE	9
3	Mesh Refinement Strategies	12
3.1	Preliminaries for Triangular Meshes	12
3.2	TrefineR	14
3.3	TrefineRG	18
3.4	TrefineNVB	23
3.5	TrefineRGB	30
3.6	Preliminaries for Quadrilateral Meshes	34
3.7	QrefineR	39
3.8	QrefineRGtri	42
3.9	QrefineRB	47
3.10	QrefineR2	54
3.11	QrefineRG2	56
4	Implementation	61
4.1	Data Structures	61
4.2	Implementation with Hash Maps	65
4.3	Implementation with Virtual Elements	85

4.4	Implementation in Three Steps	100
5	Numerical Experiments	118
5.1	Solving Poisson Problems with AFEM	118
5.2	Mesh Refinements within a Different Context	123
6	Conclusion	129

1 Introduction

In many applications such as rendering in computer graphics or solving partial differential equations (PDEs) with the finite element method (FEM), an appropriate generation of meshes is indispensable. However, 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 of other ones and by the use of an external mesh refinement software, it is difficult to integrate this step with 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 [35]. Thus, in this work we provide an accessible mesh refinement implementation in MATLAB that can be used in a very flexible way. Nine different mesh refinement strategies for triangular and quadrilateral elements are investigated in the course of this work. Since the main focus of this work is the setting of adaptive finite element methods (AFEM), we give a short introduction of the adaptive finite element method so that the reader can put this work into its proper context and assess it correctly. We also hope to serve the educational purpose of how to implement mesh refinement strategies in an easy but efficient way.

1.1 Adaptive Finite Element Method

The adaptive finite element method aims to approximate the solution of a partial differential equation. To this end, the underlying domain of the problem is split into elements and on each element basis functions are chosen. On this mesh, the solution of the PDE is approximated. We call this step **SOLVE**. The AFEM consists of four components, namely:

SOLVE - ESTIMATE - MARK - REFINES

After solving the PDE, one needs to **ESTIMATE** the error on each element. If the error is too big, those elements will be **MARK**ed for further refinement. De-

pending on the marked elements, the mesh of elements will be **REFINED** or not. This process loops until the error estimator is small enough for all elements (or the maximal allowed number of elements is reached) and then stops. As a result of the adaptivity, we receive a highly adapted mesh which is quite useful for problems with singularities or re-entrant corners. This is just a short introduction, the framework will be explained in more detail in this work. However, we mainly want to focus on the step **REFINE** and allow different types of refinement strategies.

A short finite element method for P_1 - Q_1 -finite elements with triangles and quadrilaterals is already implemented in MATLAB from J. Alpert, C. Carstensen, and S. A. Funken in 1999 [2]. A couple of years later an adaptive P_1 -finite element method is provided as Matlab package `plafem` [22]. Our contribution can be seen as an extension to these works in the sense that we investigate the properties of different mesh refinement strategies and implement those, whereas we not only focus on P_1 -elements as in [22] but also extend them to Q_1 -finite elements. The aim of this work is to provide efficient implementations of different adaptive mesh refinement techniques that can, among other contexts, be used in the step **REFINE** in the AFEM. This work shall also contribute to the long-term objective of creating an AFEM-package that offers more flexibility in the **REFINE** step. However, a full description of the package goes beyond the scope of this work and will be examined in a future work.

1.2 Outline

This work is organized as follows: We first present our model problem and the general framework of adaptive finite element methods. This includes basic definitions and some explanations to the four steps in the AFEM. In Chapter 3, we focus on the part **REFINE** and present different refinement strategies for triangles and quadrilaterals and their properties. In the following chapter, an efficient implementation of the introduced refinement strategies in MATLAB is described. Examples in Chapter 5 demonstrate that a quasi-optimal convergence rate can be obtained with adaptive mesh refinement strategies which shows the superiority over a uniform mesh refinement. Furthermore, a closer look on the efficiency of the implementations is made. In the last chapter, we summarize the results, conclude the work and give an insight into future work.

2 General Framework

In this chapter, we present our model problem and give an overview of the components of AFEM. Furthermore, we give some important definitions that will be used hereinafter. The general framework as described in this chapter is based on [2, 22]. We keep the notation and formulation close to its original.

2.1 Model Problem

We consider the Poisson problem with mixed boundary conditions. Let therefore $\Omega \subset \mathbb{R}^2$ be a bounded Lipschitz domain with polygonal boundary $\Gamma = \partial\Omega$. Given $f \in L^2(\Omega)$, $u_D \in H^1(\Omega)$ and $g \in L^2(\Gamma_N)$, we seek $u \in H^1(\Omega)$ with

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

where Γ_D is a closed subset with positive measure of Γ and $\Gamma_N := \Gamma \setminus \Gamma_D$.

As a well-known result, the unique existence of the solution of this model problem (2.1) is given by the Lax-Milgram Lemma.

For simplification, we rewrite the inhomogeneous Dirichlet boundary conditions $u = u_D$ into a homogeneous Dirichlet boundary condition by setting $\tilde{u} := u - u_D$ such that $\tilde{u} = 0$ on Γ_D , i.e.

$$\tilde{u} \in H_D^1(\Omega) := \left\{ v \in H^1(\Omega) \mid v = 0 \text{ on } \Gamma_D \right\}.$$

With this incorporation the weak formulation reads:

Find $\tilde{u} \in H_D^1(\Omega)$ such that

$$\int_{\Omega} \nabla \tilde{u} \cdot \nabla v \, dx = \int_{\Omega} f v \, dx + \int_{\Gamma_N} g v \, ds - \int_{\Omega} \nabla u_D \cdot \nabla v \, dx, \quad \forall v \in H_D^1(\Omega). \tag{2.2}$$

The unique solution of model problem (2.1) is then given by $u = \tilde{u} + u_D \in H^1(\Omega)$.

2.2 Galerkin Discretization of the Problem

For a numerical treatment, the spaces $H^1(\Omega)$ and $H_D^1(\Omega)$ are not suitable since they are infinite dimensional spaces. To deal with this problem we choose, as in [2], finite dimensional subspaces $S \subset H^1(\Omega)$ and $S_D = S \cap H_D^1(\Omega)$. We want to approximate the functions $u_D \in H^1(\Omega)$ and $v \in H_D^1(\Omega)$ with functions in finite dimensional spaces. For $U_D \in S$ the problem (2.2) then reads:

Find $\tilde{U} \in S_D$ such that

$$\int_{\Omega} \nabla \tilde{U} \nabla V \, dx = \int_{\Omega} f V \, dx + \int_{\Gamma_N} g V \, ds - \int_{\Omega} \nabla U_D \nabla V \, dx, \quad \forall V \in S_D. \quad (2.3)$$

Since S and S_D are finite dimensional spaces, we can find a basis (η_1, \dots, η_N) of S and the dimension M of S_D is given by $M \leq N - 2$. This is valid since we assume Dirichlet boundary Γ_D with positive length and therefore at least one edge, i.e. two nodes, are constrained through the boundary condition. Thus, we can choose a set $I = \{i_1, \dots, i_M\} \subset \{1, \dots, N\}$ such that $(\eta_{i_1}, \dots, \eta_{i_M})$ is a basis of S_D . Then, (2.3) is equivalent to

$$\int_{\Omega} \nabla \tilde{U} \nabla \eta_j \, dx = \int_{\Omega} f \eta_j \, dx + \int_{\Gamma_N} g \eta_j \, ds - \int_{\Omega} \nabla U_D \nabla \eta_j \, dx, \quad \forall j \in I.$$

Using the following basis representation and the nodal interpolant

$$\tilde{U} = \sum_{k \in I} x_k \eta_k \in S_D \text{ and } U_D = \sum_{k=1}^N U_k \eta_k \in S$$

we finally obtain a linear system of equations

$$\underbrace{\left(\int_{\Omega} \nabla \eta_k \nabla \eta_j \, dx \right)_{j,k \in I}}_{:= A} \underbrace{\left(x_k \right)_{k \in I}}_{:= x} = \underbrace{\left(\int_{\Omega} f \eta_j \, dx + \int_{\Gamma_N} g \eta_j \, ds - \sum_{k=1}^N U_k \int_{\Omega} \nabla \eta_k \nabla \eta_j \, dx \right)_{j \in I}}_{:= b}. \quad (2.4)$$

The matrix A is symmetric and positive definite, thus (2.4) has a unique solution $x \in \mathbb{R}^M$ and the discretized solution U to problem (2.1) is given by

$$U = U_D + \tilde{U} = \sum_{k=1}^N U_k \eta_k + \sum_{k \in I} x_k \eta_k \in S.$$

2.3 Data Structures

The question that arises is how to construct the spaces S and S_D . We need a concrete basis for those spaces and therefore consider a decomposition of the polygonal domain Ω . On this decomposition, we can define piecewise functions in a way that they form a basis of the spaces S and S_D .

Firstly, let us define some basic terms that are of importance throughout this work.

Definition 2.3.1 (Triangulation [15]) *Let Ω a polygonal Lipschitz domain in \mathbb{R}^2 . A triangulation \mathcal{T} of Ω is a finite set of polygonal Lipschitz domains for which holds*

$$\overline{\Omega} = \bigcup_{T \in \mathcal{T}} T$$

and for two elements $T_1, T_2 \in \mathcal{T}$ with $T_1 \neq T_2$ it holds

$$\overset{\circ}{T}_1 \cap \overset{\circ}{T}_2 = \emptyset.$$

Here we denote $\overset{\circ}{T}$ the interior of T .

Remark 2.3.2 *The literature frequently defines elements $T \in \mathcal{T}$ as open elements. We want to point out that we use a definition where the elements $T \in \mathcal{T}$ are considered to be closed.*

The given definition is too general for our purposes and therefore, we provide a more concrete definition.

Definition 2.3.3 (Regular triangulation [15]) *Let \mathcal{T} a triangulation of Ω and let $\Gamma := \partial\Omega$ be divided into a closed segment Γ_D and $\Gamma_N := \Gamma \setminus \Gamma_D$. We call \mathcal{T} a regular triangulation of Ω if*

- i) for all $T \in \mathcal{T}$ holds $T \cap \Gamma_D$ is empty or an edge,*
- ii) for all $T_1, T_2 \in \mathcal{T}$, with $T_1 \neq T_2$ holds $T_1 \cap T_2$ is empty, a common node or a common edge,*

iii) for all $T \in \mathcal{T}$ holds $\overset{\circ}{T} \neq \emptyset$.

Remark 2.3.4 If $x \in T_1 \cap T_2$ is a vertex of T_1 but not of T_2 , we call x a hanging node. A regular triangulation does not contain any hanging nodes. We call a triangulation with hanging nodes an irregular triangulation or k -irregular triangulation where k is the maximal number of hanging nodes per edge.

For simplification, we also introduce the following notations.

Definition 2.3.5 We denote the set of all vertices of a triangulation \mathcal{T} with \mathcal{N} , and the set of all edges with \mathcal{E} .

Note, that with this definition we can define the following sets:

$$\begin{aligned} \mathcal{N}(E) &:= \{z \in \mathcal{N} \mid z \in E\} \text{ denotes the nodes of edge } E \in \mathcal{E}, \\ \mathcal{N}(T) &:= \{z \in \mathcal{N} \mid z \in T\} \text{ denotes the nodes of element } T \in \mathcal{T}, \\ \mathcal{E}(z) &:= \{E \in \mathcal{E} \mid z \in E\} \text{ denotes the edges with node } z \in \mathcal{N}, \\ \mathcal{E}(T) &:= \{E \in \mathcal{E} \mid E \subset \partial T\} \text{ denotes the edges of element } T \in \mathcal{T}, \\ \mathcal{T}(E) &:= \{T \in \mathcal{T} \mid E \subset \partial T\} \text{ denotes the elements with edge } E \in \mathcal{E}, \\ \mathcal{T}(z) &:= \{T \in \mathcal{T} \mid z \in T\} \text{ denotes the elements with node } z \in \mathcal{N}. \end{aligned}$$

With this preliminaries, we consider a triangulation of a set Ω into triangles or quadrilaterals. Starting with a triangulation \mathcal{T} into triangles T , we introduce the spaces

$$P_r(T) = \{p : T \rightarrow \mathbb{R} \mid p \text{ is a polynomial of degree at most } r \text{ on } T\}$$

with $r \in \mathbb{N}$. For $r = 1$, this is the space of linear functions $P_1(T)$ of the form

$$p(x) = a + bx_1 + cx_2, \quad x = (x_1, x_2) \in \mathbb{R}^2, \quad p \in P_1(T), \quad a, b, c \in \mathbb{R}$$

and thus the dimension of $P_1(T)$ is three. Let t_1, t_2, t_3 be the vertices of a triangle $T \in \mathcal{T}$. For all $i \in \{1, 2, 3\}$, there exists exactly one $v_i \in P_1(T)$ such that $v_i(t_j) = \delta_{i,j}$, $i, j \in \{1, 2, 3\}$, i.e. the values at the vertices suffice to determine the coefficients of the linear function uniquely. The set $\{v_1, v_2, v_3\}$ forms a nodal basis of $P_1(T)$. Thus, coming back to the triangulation of Ω , we define the finite dimensional space

S as the space of continuous piecewise linear functions, i.e.

$$S = \left\{ p : \Omega \rightarrow \mathbb{R} \mid p \in C(\Omega) \text{ and } p|_T \in P_1(T), T \in \mathcal{T} \right\} \subset H^1(\Omega).$$

Analogously, for a triangulation \mathcal{T} into quadrilaterals T , we introduce the spaces

$$Q_r(T) = \left\{ q : T \rightarrow \mathbb{R} \mid q(x) = \sum_{j=1}^r \sum_{i=1}^r a_{i,j} x_1^i x_2^j, \ x = (x_1, x_2) \in T, \ a_{i,j} \in \mathbb{R} \right\}$$

with $r \in \mathbb{N}$. For $r = 1$, this is the space of bilinear functions, i.e. functions of the form

$$q(x) = a + bx_1 + cx_2 + dx_1x_2, \quad x = (x_1, x_2) \in \mathbb{R}^2, \quad q \in Q_1(T), \quad a, b, c, d \in \mathbb{R}$$

and $\dim(Q_1(T)) = 4$. As in the case of triangles, we pick the basis $\{v_1, v_2, v_3, v_4\}$ of $Q_1(T)$ as a nodal basis $v_i(t_j) = \delta_{i,j}$ with vertices t_1, t_2, t_3, t_4 of the quadrilateral $T \in \mathcal{T}$. It can be shown that an appropriate choice of the space S is also given by

$$S = \left\{ q : \Omega \rightarrow \mathbb{R} \mid q \in C(\Omega) \text{ and } q|_T \in Q_1(T), T \in \mathcal{T} \right\} \subset H^1(\Omega),$$

the space of continuous piecewise bilinear functions. To receive the space $S_D \subset S$, we just replace $C(\Omega)$ in the definitions by $C_D(\Omega) = \left\{ p \in C(\overline{\Omega}) \mid p|_{\Gamma_D} = 0 \right\}$ and thus receive the finite dimensional space $S_D \subset H_D^1(\Omega)$. Further information can be found in any standard literature such as [14].

2.4 SOLVE

Using the spaces S and S_D , the stiffness matrix of the linear system of equations in (2.4) can be rewritten to

$$A_{jk} = \sum_{T \in \mathcal{T}} \int_T \nabla \eta_j \cdot \nabla \eta_k \, dx. \quad (2.5)$$

Furthermore, the right hand side reads

$$b_j = \sum_{T \in \mathcal{T}} \int_T f \eta_j \, dx + \sum_{E \in \Gamma_N} \int_E g \eta_j \, ds - \sum_{k=1}^N U_k \sum_{T \in \mathcal{T}} \int_T \nabla \eta_j \cdot \nabla \eta_k \, dx.$$

A detailed description on the assembly of the stiffness matrix and the right hand side as well as the incorporation of the boundary data is provided in [2]. The system of linear equations can then be solved by use of the backslash operator in MATLAB.

2.5 ESTIMATE

In this section, we focus on the part **ESTIMATE** as it is described in [22]. With an adaptive algorithm, we want to reduce computational time and storage. Therefore, we try to keep the number of elements as small as possible, whereas we also want to achieve a certain accuracy of the computed solution. This means, we need to make sure that the error

$$\|u - U\|_{H^1(\Omega)}$$

is minimal. Since u is unknown, this error cannot be computed and we need to find some error estimators that approximate the error in a good manner. With the help of an error estimator, a highly adapted mesh is then generated iteratively. For each element $T \in \mathcal{T}$, let $\eta_T \in \mathbb{R}$ be a *refinement indicator* that satisfies

$$\eta_T \approx \|u - U\|_{H^1(T)} \quad \forall T \in \mathcal{T}.$$

The associated *error estimator* $\eta = (\sum_{T \in \mathcal{T}} \eta_T^2)^{\frac{1}{2}}$ then reveals an error estimate for $\|u - U\|_{H^1(\Omega)}$. A classification and overview of a posteriori error estimates is given in [32]. We want to mention some sources in this context and only state the estimator that will be used in our work. The estimates can be categorized in residual estimates [5, 4, 32], solution of local problems as error estimation [5, 10], hierarchical basis error estimates [9, 20, 32], averaging techniques for error estimation [36] and duality techniques [11].

In this work, we only consider the residual-based error estimator $\eta_R := (\sum_{T \in \mathcal{T}} \eta_T^2)^{\frac{1}{2}}$ as discussed in [22] with refinement indicator

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

Here, h_T denotes the diameter of T , i.e. $\text{diam}(T) = \sup \{|x - y| : x, y \in T\}$. The term J_h describes the jump over an interior edge E and is defined by

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

with $T_+, T_- \in \mathcal{T}$ neighboring elements with outer unit normal vectors n_+, n_- . For triangles $\nabla U|_T$ is a constant function and thus the implementation in [22] is adequate. However, for quadrilaterals $\nabla U|_T$ is not constant anymore. Hence, a further implementation is needed which is provided by Stefan A. Funken for quadrilaterals and further specific cases such as mixed or irregular meshes.

2.6 MARK

The step **MARK** selects elements based on the refinement indicator calculated in the step **ESTIMATE**. Well-known techniques are the Dörfler marking [21] and the maximum criterion introduced by Babuška and Vogelius [6]. The maximum criterion marks elements $T \in \mathcal{T}$ for refinement if

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

for an arbitrary $\theta \in (0, 1)$. The implementation can easily be done, see [22]. In the case of Dörfler marking, we find the minimal set $\mathcal{M} \subset \mathcal{T}$ such that

$$\theta \sum_{T \in \mathcal{T}} \eta_T^2 \leq \sum_{T \in \mathcal{M}} \eta_T^2$$

for an arbitrary $\theta \in (0, 1)$ only elements $T \in \mathcal{M}$ are marked. If we choose $\theta = 1$, we get a uniform mesh refinement and for $\theta \rightarrow 0$ the resulting meshes are highly adapted. An efficient implementation of the Dörfler criterion can also be found in [22].

2.7 REFINE

The step **REFINE** is of main interest in this work. We want to highlight crucial properties of a triangulation and why they matter in the context of the finite element method.

The *shape regularity* of a triangulation plays an important role in the analysis of the finite element method. This condition paves the way for deriving optimal interpolation order, proving convergence of the finite element method, and deriving a posteriori error estimates as we have seen in Section 2.5. It is also an important property in frameworks outside of the finite element method. For example, the nu-

merical computation of the surface area of a cylinder fails for degenerated triangles that do not hold the shape regularity [33]. We will focus on this property for each refinement strategy. But first, we give some formal definitions and discuss their meaning. For triangles we define the shape regularity with the so-called *inscribed ball condition* proposed by Ciarlet in 1978 [18].

Definition 2.7.1 (Shape regularity for triangles) *A family $\{\mathcal{T}_h\}$ of triangulations with $h = \max_{T \in \mathcal{T}} h_T > 0$ (into triangles) is called shape regular if there exists $c \geq 1$ with*

$$\frac{h_T}{\rho_T} \leq c \quad \forall T \in \mathcal{T}_h, \quad \forall h > 0,$$

where h_T denotes the diameter of $T \in \mathcal{T}_h$ and ρ_T is the diameter of the largest ball inscribed in T .

This property guarantees that the elements do not degenerate. We can also state an equivalent angle condition introduced by Zlámal in 1968 [37], which is known as the *minimal angle condition*.

Definition 2.7.2 (Minimal angle condition for triangles) *For a family $\{\mathcal{T}_h\}$ as above the minimal angle of all triangles $T \in \mathcal{T}_h$ is bounded away from zero, i.e. there exists $\alpha_0 > 0$ such that the minimal angle α_T of triangle T satisfies:*

$$\alpha_T \geq \alpha_0 > 0 \quad \forall T \in \mathcal{T}_h.$$

Zlámal derived optimal interpolation error bounds and thus by Céa's Lemma also some rate of the discretization error by using the minimal angle condition. In 1976, Babuška and Aziz proposed a weaker condition, called the *maximum angle condition* [3].

Definition 2.7.3 (Maximal angle condition for triangles) *For a family $\{\mathcal{T}_h\}$ as above, the maximal angle of all triangles $T \in \mathcal{T}_h$ is bounded from above by π , i.e. there exists $\gamma_0 < \pi$ such that the maximal angle γ_T of triangle T satisfies:*

$$\gamma_T \leq \gamma_0 < \pi \quad \forall T \in \mathcal{T}_h.$$

Under this condition an optimal interpolation error bound can also be proven [3]. However, even the maximum angle condition is not necessary for convergence as stated in [23]. It is obvious that the minimal angle condition implies the maximum

angle condition since $\gamma_T \leq \pi - 2\alpha_T \leq \pi - 2\alpha_0 = \gamma_0$, but the converse is not true. The maximal angle condition allows triangles to become very thin, whereas the minimum angle condition prevents this case.

The question that arises is whether a similar definition makes sense for quadrilaterals. Using the same terms as in Definition 2.7.1 for a quadrilateral, two unintentional things can happen:

- the maximum angle tends to π ,
- the shortest edge length approaches zero,

and thus a quadrilateral can degenerate to a triangle. To avoid this, we extend this definition with a further condition which is proposed by Ciarlet and Raviart in [19].

Definition 2.7.4 (Shape regularity for quadrilaterals) *A family $\{\mathcal{T}_h\}$ of triangulations with $h = \max_{T \in \mathcal{T}} h_T > 0$ into quadrilaterals is called shape regular if there exists $c_1, c_2 > 0$ such that*

$$\frac{\bar{h}_T}{\underline{h}_T} \leq c_1 \quad \forall T \in \mathcal{T}_h$$

with \bar{h}_T and \underline{h}_T the longest and shortest edge of the quadrilateral T respectively and

$$|\cos(\theta)| \leq c_2 < 1 \quad \forall \text{ inner angles } \theta \text{ of } T.$$

There exist less restrictive conditions on shape regularity as discussed in [1] but for our purposes this definition suffices.

3 Mesh Refinement Strategies

As already mentioned in the introduction, we investigate nine different mesh refinement strategies for triangular and quadrilateral elements in the course of this work. Let us first give an overview of these refinement strategies as illustrated in Figure 3.1 for triangular meshes and in Figure 3.2 for quadrilateral meshes.

Our type of naming convention should be understood in the following way:

- the first character describes the mesh type: `T` for triangular meshes and `Q` for quadrilateral meshes;
- `refine` for the step **REFINE** in the adaptive finite element method;
- the suffix `R` for red refinement, `RB` for red-blue refinement, `RG` for red-green refinement, `RGB` for red-green-blue refinement, `NVB` for newest vertex bisection;
- a further suffix `tri` characterizes that for the green refinement triangles are allowed, if not obvious;
- a further suffix `2` indicates that the edges are trisected, i.e. two new nodes on one edge are introduced. By default, edges are bisected such that only one new node is formed.

3.1 Preliminaries for Triangular Meshes

The main idea of adaptive finite element methods is to refine the mesh only locally. To ascertain in which regions a refinement makes sense the method uses refinement indicators as explained in Chapter 2. As input of the **REFINE** step, we have a triangulation \mathcal{T}_ℓ and a set of marked elements \mathcal{M}_ℓ ($\ell = 0, 1, 2, \dots$). For all triangular meshes, we want to understand the element marking in the following way. If a triangle T is an element in \mathcal{M}_ℓ , we mark all three edges of this triangle for bisection. Thus, also edges from neighboring elements $T \notin \mathcal{M}_\ell$ may be affected by this approach.

For triangular meshes, there exist three well-known refinement rules, namely the red-, green-, and blue-refinement rule.

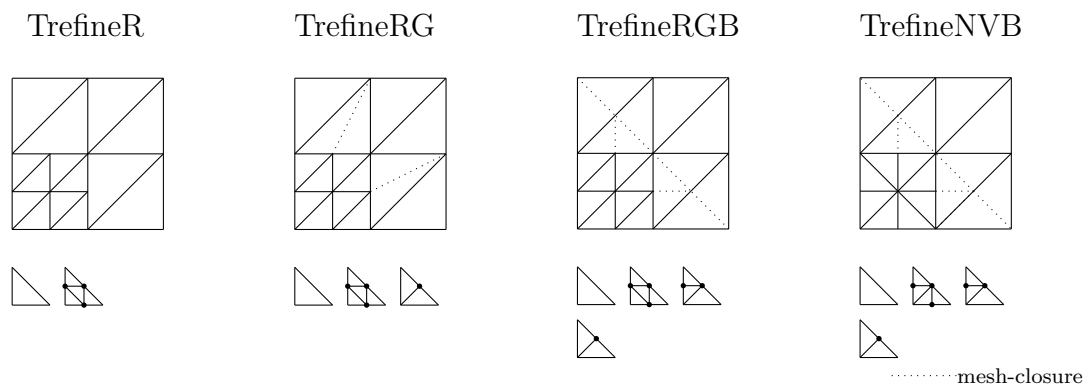


Figure 3.1: Refinement strategies for triangular meshes. An exemplary mesh is refined in the lower left corner. Depending on the allowed refinement patterns shown below the mesh, introduced hanging nodes can be eliminated. This process is called *mesh closure* and is signalized through dotted lines.

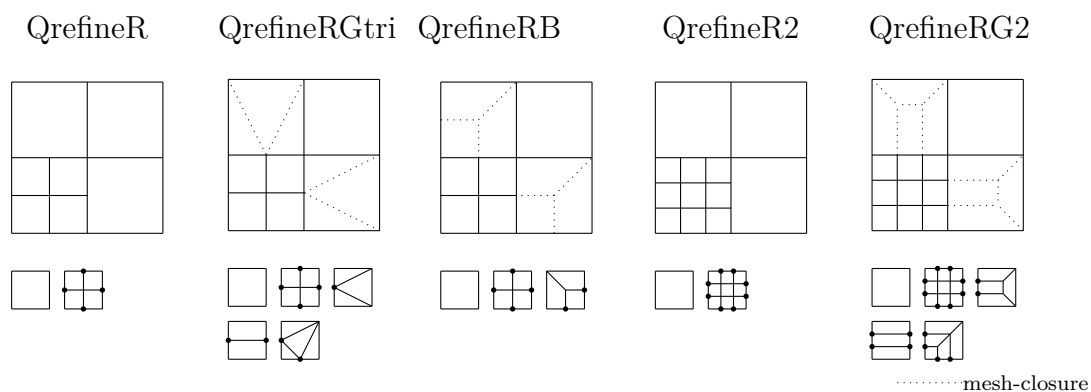


Figure 3.2: Refinement strategies for quadrilateral meshes. An exemplary mesh is refined in the lower left corner. Depending on the allowed refinement patterns shown below the mesh, introduced hanging nodes can be eliminated. This process is called *mesh closure* illustrated through dotted lines.

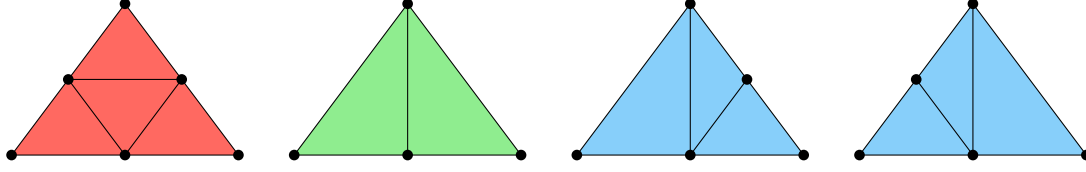


Figure 3.3: Red-, green-, blue_ℓ- and blue_r-refinement of a triangle (from left to right).

Definition 3.1.1 (Red refinement of a triangle) *A triangle T is split into four similar subtriangles by connecting the midpoints of all edges of T with each other. This is called a red refinement.*

Definition 3.1.2 (Green refinement of a triangle) *A triangle T is divided into two subtriangles by bisecting one edge of the triangle and connecting this midpoint to the vertex opposite to this edge. This is called a green refinement.*

Definition 3.1.3 (Blue refinement of a triangle) *A triangle T is subdivided into three triangles by a green refinement with subsequent bisection of one of the other edges. This midpoint is then connected with the midpoint introduced in the green refinement. We call this blue refinement.*

The red-, green-, and blue-refinements are illustrated in Figure 3.3.

In the following, we mix these refinement rules in different ways and investigate the properties for each strategy. Since the color of these refinement types plays an important role, we have the convention that all triangles T in the initial triangulation \mathcal{T}_0 are painted in red. For all strategies, we wish to maintain the regularity of the grid, but in cases in which this is not possible, we follow the 1-Irregular Rule proposed by Bank, Sherman, and Weiser in [8] which says: *Refine any unrefined element that has more than one hanging node on an edge.*

3.2 TrefineR

In TrefineR we only allow one refinement pattern, namely a red refinement. Thus, for a triangle we have two options depicted in Figure 3.4. Obviously, neighboring elements may be affected by a red refinement, i.e. hanging nodes can arise.

If we apply these two patterns straightforward, the 1-irregularity of the grid cannot be ensured, see Figure 3.5. Thus, we deploy the 1-Irregular Rule to resolve this issue, see Figure 3.6.

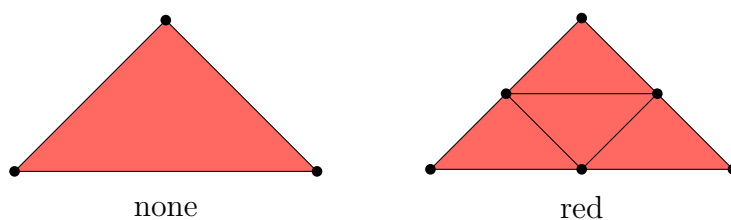


Figure 3.4: Refinement patterns for TrefineR.

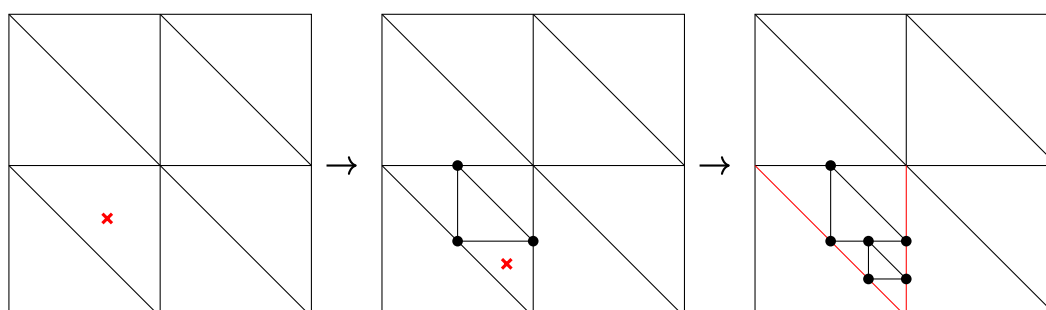


Figure 3.5: A straightforward application of the two refinement patterns in TrefineR. An element is marked for refinement as indicated through a red cross. By applying the two refinement patterns, we receive a grid in the second refinement that is not 1-irregular any more. The edges colored in red have two hanging nodes. Thus a straightforward application of the two refinement patterns does not suffice.

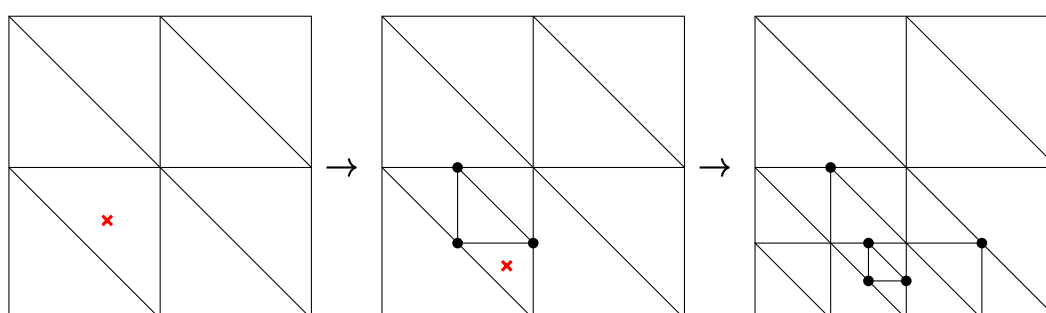


Figure 3.6: Resolving the issues of a straightforward application of the two refinement patterns in TrefineR. We deploy the 1-Irregular Rule which says that any unrefined element that has more than one hanging node is refined. Thus, the 1-irregularity of the grid is ensured.

To distinguish between red elements and elements with hanging nodes, we paint elements with hanging nodes in gray for further considerations. The procedure of TrefineR is basically given by two ingredients. We always use the refinement patterns as shown in Figure 3.4 and maintain the 1-Irregular Rule whenever needed. An explanatory description in the upcoming section shall give a better insight on how to understand the strategy.

3.2.1 Description of TrefineR

In the beginning, all elements of the initial triangulation \mathcal{T}_0 are painted in red. Starting with a red triangle, three different cases can occur:

- Case 1) no edge is marked,
- Case 2) one or two edges are marked,
- Case 3) all three edges are marked either because all three neighboring edges are marked or because the element is marked.

In Case 1, this element remains unaffected. In Case 2, the element also remains unchanged except for an introduction of hanging nodes on the marked edges. In Case 3, the triangle is red refined. This results into four similar triangles. For an exemplary illustration, refer to Figure 3.7.

If a triangular element already has some hanging nodes, indicated by gray, further four situations can arise:

- Case 1) no edge is marked,
- Case 2) only further regular edges are marked such that the total number of hanging nodes does not exceed two,
- Case 3) this element is marked for refinement or further regular edges are marked until all three edges have hanging nodes,
- Case 4) at least one irregular edge is marked for further refinement through neighboring elements.

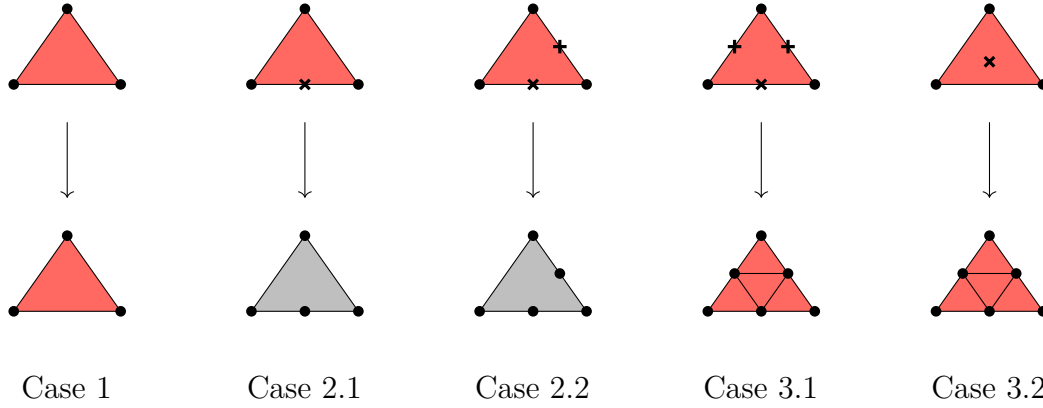


Figure 3.7: Exemplary illustration of TrefineR: refinement of red elements. In Case 1, the triangle remains unaffected. In Case 2, hanging nodes are introduced, in Case 3 all three edges are marked through neighboring elements (Case 3.1) or through element marking (Case 3.2). This results into a red refinement, i.e. four smaller triangles are formed. For a better understanding, we colored red elements in red and red elements with hanging nodes in gray.

In Case 1, the element remains unchanged. In Case 2, remaining regular edges are bisected. If in the end all edges are bisected (Case 3), the element becomes red refined. This is also true for marked elements. In Case 4, irregular edges are marked for refinement. Since we do not allow more than one hanging node per edge, we first refine this element and introduce a new hanging node on the marked irregular edges. See Figure 3.8 for an illustration.

3.2.2 Properties

We observe the following important properties of TrefineR.

Remark 3.2.1 *The meshes generated by TrefineR are nested, 1-irregular and hold the shape regularity.*

This is true because we notice that all triangles defined through a red refinement are geometrically similar to their father element. Thus, all refined meshes given by TrefineR hold the shape regularity. The grids are also nested, since in each step only nodes and edges are added and never removed. However, the resulting mesh is 1-irregular because at most one hanging node per edge is allowed. Thus, to ensure the continuity of the solution one needs to impose further constraints at the hanging

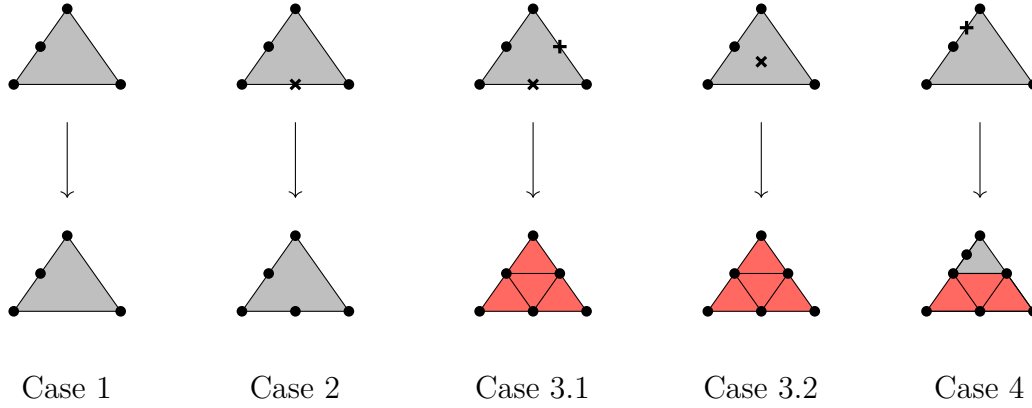


Figure 3.8: Exemplary illustration of TrefineR: refinement of elements with irregular edges. In Case 1, the element remains unchanged. In the second case, regular edges are marked for refinement and result into more hanging nodes (Case 2) or a red refinement (Case 3.1). In Case 3, the element is marked for refinement, i.e. only regular edges are bisected. In the last case, an irregular edge is marked for further refinement. Before bisecting an irregular edge, a red refinement of the father element is done. Again, the gray color indicates red elements with hanging nodes and the red color is used for red elements without hanging nodes.

nodes. To avoid hanging nodes, we investigate the next refinement strategy.

3.3 TrefineRG

TrefineRG can be seen as an extension of TrefineR and was first introduced by Bank and Sherman [7]. In TrefineR, we had to deal with hanging nodes. To avoid hanging nodes, we allow a green refinement additionally to the red refinement. A green refinement eliminates hanging nodes by connecting those with the vertex opposite to the hanging node of this triangle. This option is only available for a triangle with one hanging node. Whenever two edges are marked for bisection, we also mark the third edge; thus, we get a red refinement. Figure 3.9 shows the allowed patterns for red elements and Figure 3.10 for green elements in TrefineRG. A straightforward application of the three refinement patterns leads to degenerated triangles; see Figure 3.11. In order to prevent the triangles from becoming too thin, a green element is coarsened to its father element before it is bisected further as shown in Figure 3.12. The TrefineRG process is given in Algorithm 3.1.

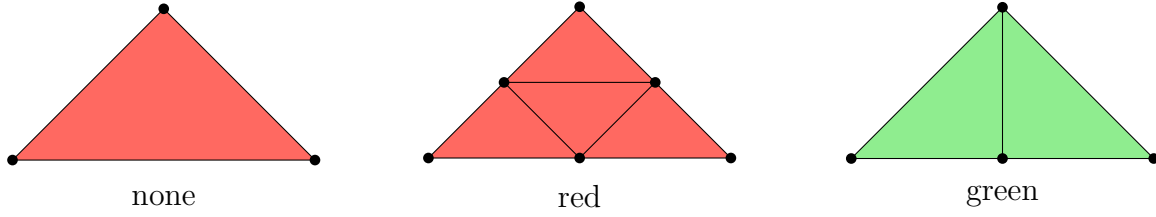


Figure 3.9: Refinement patterns for red elements in TrefineRG. From left to right: A red element and its refinement patterns. Note, that a green refinement can also split a red triangle along the other medians of this element. Thus, there exists three options of green-refining a triangle.

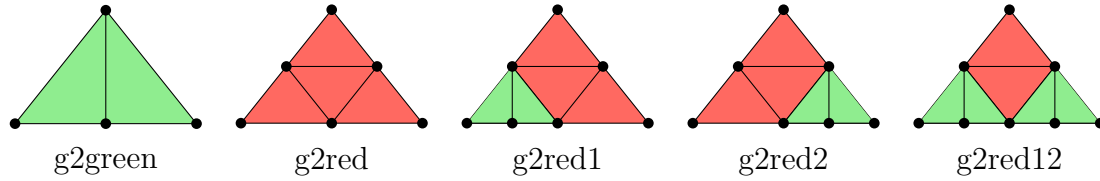


Figure 3.10: Refinement patterns for green elements in TrefineRG. From left to right: A green element and its refinement patterns. The labeling of each refinement pattern is used in the implementation and therefore given for an easier understanding.

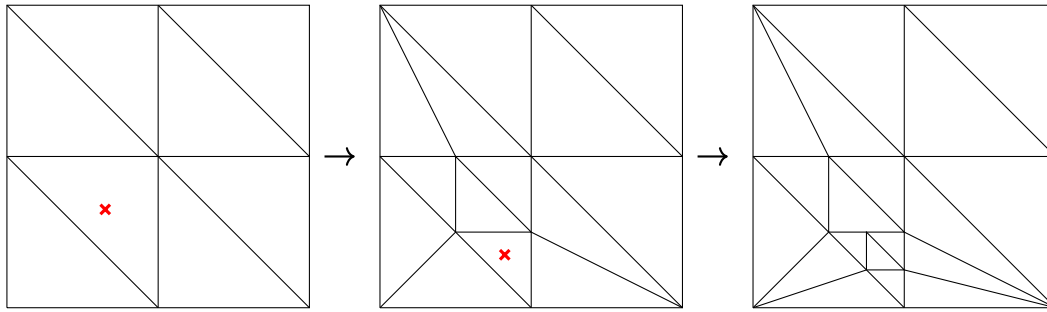


Figure 3.11: A straightforward application of the three refinement patterns in TrefineRG. An element is marked for refinement as indicated through a red cross. By applying the three refinement patterns, we receive from step to step thinner triangles. Thus a straightforward application of the three refinement patterns does not guarantee a shape regular grid.

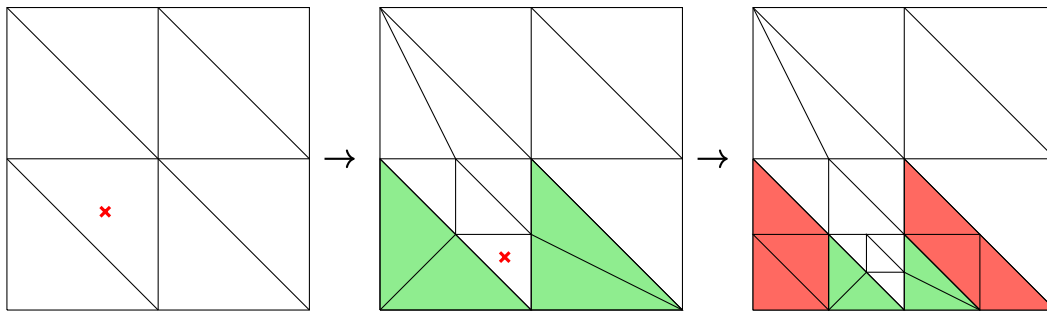


Figure 3.12: Resolving the issues of a straightforward application of the three refinement patterns in TrefineRG. We coarsen the green refinement to its father element and refine it red before bisecting further. Thus the triangles do not degenerate.

Algorithm 3.1: TrefineRG

INPUT: A triangulation $(\mathcal{T}_\ell^{\text{red}}, \mathcal{T}_\ell^{\text{green}}, (K_T)_{T \in \mathcal{T}_\ell^{\text{green}}})$ and a set of marked elements $\mathcal{M}_\ell \subseteq \mathcal{T}_\ell = \mathcal{T}_\ell^{\text{red}} \cup \mathcal{T}_\ell^{\text{green}}$. The triangulation consists of red and green elements and for each green element the father element K_T is specified.

Set $\mathcal{E}_{\mathcal{M}_\ell}^{(0)} = \emptyset$, $k := 1$ and define the set of marked edges

$$\mathcal{E}_{\mathcal{M}_\ell}^{(k)} := \left\{ \mathcal{E}(T) : T \in \mathcal{M}_\ell \cap \mathcal{T}_\ell^{\text{red}} \right\} \cup \left\{ \mathcal{E}(K_T) : K_T \text{ is father element of } T, T \in \mathcal{M}_\ell \cap \mathcal{T}_\ell^{\text{green}} \right\}$$

While $\mathcal{E}_{\mathcal{M}_\ell}^{(k)} \neq \mathcal{E}_{\mathcal{M}_\ell}^{(k-1)}$:

- set $k = k + 1$ and $\mathcal{E}_{\mathcal{M}_\ell}^{(k)} = \mathcal{E}_{\mathcal{M}_\ell}^{(k-1)}$
- $\forall T \in \mathcal{T}_\ell^{\text{red}}$: find pattern in Figure 3.9 that introduces the least new marked edges and extend $\mathcal{E}_{\mathcal{M}_\ell}^{(k)}$ by the non-marked edges of T that need to be further refined due to the pattern
- $\forall T \in \mathcal{T}_\ell^{\text{green}}$: find pattern for the father element K_T in Figure 3.10 that introduces the least new marked edges and extend $\mathcal{E}_{\mathcal{M}_\ell}^{(k)}$ by the non-marked edges of K that need to be further refined due to the pattern

Define new triangulation $\mathcal{T}_{\ell+1}$ by matching patterns from Figure 3.9 for all $T \in \mathcal{T}_\ell^{\text{red}}$ and from Figure 3.10 for all K_T with $T \in \mathcal{T}_\ell^{\text{green}}$. Paint elements in the corresponding color and specify the father elements for all $T \in \mathcal{T}_{\ell+1}^{\text{green}}$.

OUTPUT: Triangulation $(\mathcal{T}_{\ell+1}^{\text{red}}, \mathcal{T}_{\ell+1}^{\text{green}}, (K_T)_{T \in \mathcal{T}_{\ell+1}^{\text{green}}})$ into red and green triangles with father elements K_T for all $T \in \mathcal{T}_{\ell+1}^{\text{green}}$.

In the following, we describe Algorithm 3.1 exemplary for a better understanding.

3.3.1 Description of TrefineRG

Starting with a red element of a triangular mesh, three different cases can arise:

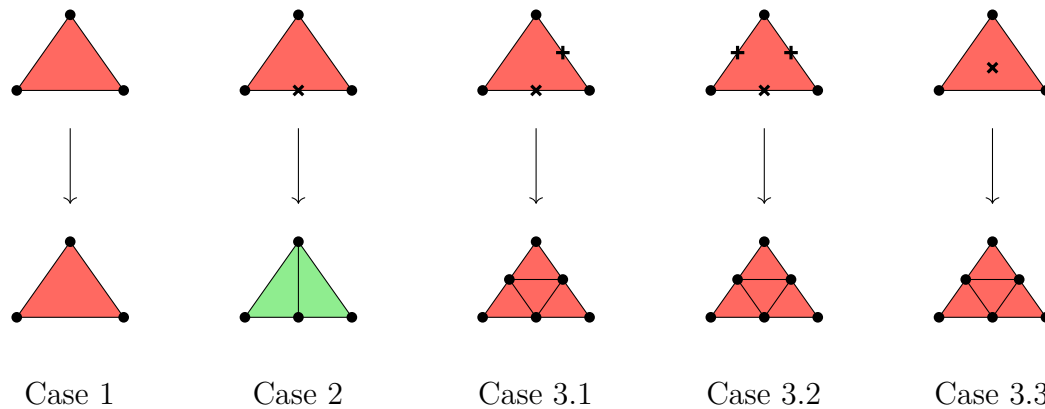


Figure 3.13: Exemplary illustration of TrefineRG: refinement of red elements. In Case 1, the element remains unchanged. In the second case, the marked edge is bisected and the new node is connected with the vertex opposite to this new node, i.e. a green refinement. In the third case, the element is red refined. For a better understanding the elements are painted in the corresponding color.

Case 1) no edge is marked,

Case 2) one edge is marked,

Case 3) two or more edges are marked either because neighboring edges are marked or because the element is marked.

In Case 1, the element remains untouched. In Case 2, the element is green refined by connecting the bisected edge with the vertex opposite to this node. A red refinement of an element is caused when two or more edges of a triangle are marked for bisection or if the element itself is marked for refinement (Case 3). For an exemplary illustration, please refer to Figure 3.13.

A green element can now be affected by further marking:

Case 1) no edge is marked,

Case 2) a 'non-bisected' edge is marked,

Case 3) one or both green elements are marked,

Case 4) an edge which is already bisected through a green refinement is marked.

In Case 1, the element remains unchanged. In Case 2, one or both edges are marked that haven't been influenced by the last green refinement step, i.e. non-bisected

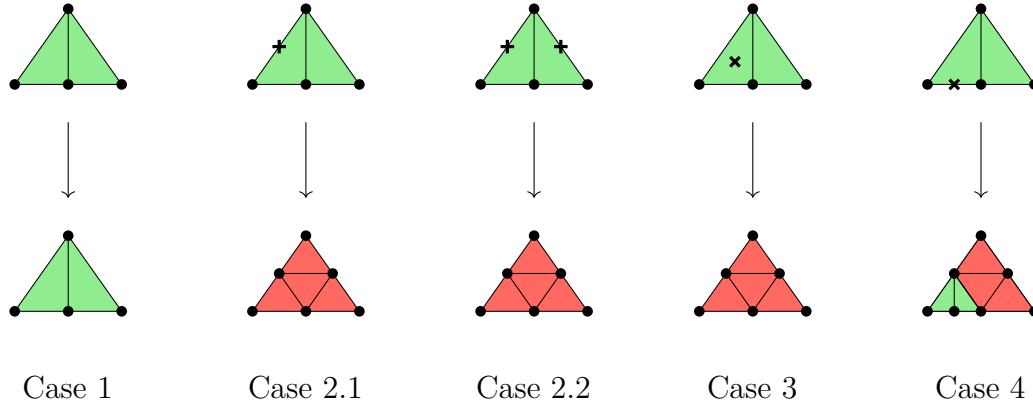


Figure 3.14: Exemplary illustration of TrefineRG: refinement of green elements. In Case 1, the element remains unaffected. In the second case, non-bisected edges are marked for refinement and thus the green refinement is undone and the element is red refined. The same happens in Case 3 when one or both green triangles are marked for further refinement. If an already bisected edge is marked, we first undo the green refinement, refine red and for the marked edges refine green; see Case 4.

edges of the red father element. In this case the green refinement is undone and the father element is red refined. This also happens in Case 3, when one or both green elements are marked for further refinement, the green element is coarsened and a red refinement of the red father element is done. In Case 4, a bisected edge is marked for further bisection. To ensure the 1-irregularity of the grid, we first undo the green refinement, refine red and then refine green in the smaller triangle where one edge is marked for bisection. Analogously, if both edges of an already bisected edge are marked for refinement. An exemplary illustration is given in Figure 3.14

3.3.2 Properties

Hanging nodes are eliminated by applying the green rule. Thus, for TrefineRG the following properties hold:

Remark 3.3.1 *The meshes generated by TrefineRG are non-nested, regular and hold the shape regularity.*

As in the TrefineR case, a red refinement ensures the shape regularity of the grid. However, applying a green refinement rule, the created elements can be less shape regular than the red elements. Thus, to avoid the loss of shape regularity, we undo the green refinement before we bisect further. Thereby, TrefineRG can be seen as the regularization of TrefineR. In each refinement step hanging nodes are removed

with a green refinement.

As a matter of fact, those grids are not nested any more. In each step, we only add more nodes, but it is possible to remove an edge. Difficulties arise with this, since the Galerkin-orthogonality does not hold for this case. However, convergence for the adaptive finite element method with a red-green refinement can be proven by the use of quasi-orthogonality under some conditions [34]. The assumption in the proof is that the initial grid \mathcal{T}_0 only contains elements T where each angle of T is larger than $\frac{\pi}{4}$ or T is an isosceles triangle with vertex angle $\frac{\pi}{6} < \theta < \frac{\pi}{2}$ [34, Remark 2].

3.4 TrefineNVB

Another approach to refine a triangular mesh is proposed by Sewell in [29] which is now known as the newest vertex bisection method (NVB). As the name suggests, in this approach we always bisect the newest vertex. Thus, we need to provide the position of the newest vertex or a reference edge which is opposite to the newest vertex. This uniquely defines the new reference edge during the refinement process. However, it is not clear which edges to pick as reference edges in the initial triangulation \mathcal{T}_0 . For now, we assume that we are given a triangulation with reference edges. Later on, we provide different algorithms on how to choose the reference edges appropriately. With the reference edge of a triangle, the refinement patterns shown in Figure 3.15 are possible by subsequently dividing the newest vertex. Obviously, the resulting refinements are a green- and blue-refinement as defined in the beginning of this elaboration. However, we do not receive the red refinement that we have already defined in Definition 3.1.1. Thus, we provide another definition for the red_{NVB} refinement.

Definition 3.4.1 (red_{NVB} refinement of a triangle) *A triangle is refined into four new triangles by bisecting the reference edge and connecting the midpoint with the opposite vertex and with the midpoints of the other two edges. We call this a red_{NVB} refinement.*

We denote $\text{ref}_E(T)$ as the reference edge E of triangle T . In contrast to TrefineRG, a green and blue element respectively are not coarsened before refined further. Thus, after the refinement process, all triangles are painted in red, no matter which

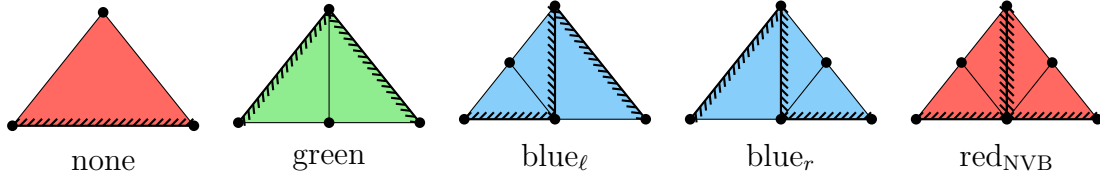


Figure 3.15: Refinement patterns for TrefineNVB. Reference edges are highlighted by hatched lines.

refinement rule was taken. The algorithm is described in Algorithm 3.2.

Algorithm 3.2: TrefineNVB

INPUT: A triangulation with reference edges $(\mathcal{T}_\ell, (\text{ref}_E(T))_{T \in \mathcal{T}_\ell})$ and a non-empty set of marked elements $\mathcal{M}_\ell \subseteq \mathcal{T}_\ell$.

Set $\mathcal{E}_{\mathcal{M}_\ell}^{(0)} = \emptyset, k := 1$ and define the set of marked edges $\mathcal{E}_{\mathcal{M}_\ell}^{(k)} := \{\mathcal{E}(T) : T \in \mathcal{M}_\ell\}$.

While $\mathcal{E}_{\mathcal{M}_\ell}^{(k)} \neq \mathcal{E}_{\mathcal{M}_\ell}^{(k-1)}$:

- define $\mathcal{E}_{\mathcal{M}_\ell}^{(k+1)} := \mathcal{E}_{\mathcal{M}_\ell}^{(k)} \cup \{\text{ref}_E(T) : \forall T \in \mathcal{T}_\ell \text{ with } \mathcal{E}(T) \cap \mathcal{E}_{\mathcal{M}_\ell}^{(k)} \neq \emptyset\}$
- set $k := k + 1$

Refine each triangle $T \in \mathcal{T}_\ell$

- with $|\mathcal{E}_{\mathcal{M}_\ell}^{(k)}(T)| = 3$: red_{NVB}
- with $|\mathcal{E}_{\mathcal{M}_\ell}^{(k)}(T)| = 2$: blue
- with $|\mathcal{E}_{\mathcal{M}_\ell}^{(k)}(T)| = 1$: green

For each refinement specify the new reference edges.

OUTPUT: The triangulation with reference edges $(\mathcal{T}_{\ell+1}, (\text{ref}_E(T))_{T \in \mathcal{T}_{\ell+1}})$.

A detailed description on this procedure is given in the following section.

3.4.1 Description of TrefineNVB

Starting with a triangular mesh, we choose a reference edge for each element, e.g. the longest edge and consider this as the edge opposite to the newest vertex. As in the presented refinement strategies above, we mark elements by marking each edge

for bisection. Thus, edges can either be marked by its element or by neighboring elements. In this case, contrary to the above strategies, the edges in an element have a specific role, i.e. we cannot just consider what happens if one edge is marked. We also need to distinguish between a marked edge and a marked reference edge. This is illustrated in Figure 3.16. The reference edge of an element is emphasized with a hatched line. When an edge is marked which is not the reference edge, we need to bisect the reference edge as long as the marked edge is the new reference edge. That is in the case of a triangle, if any edge of a triangle is marked, we need to ensure that the reference edge is marked, too. We then get the following distinction of cases:

- Case 1) no edge is marked,
- Case 2) only the reference edge is marked,
- Case 3) one edge is marked, which is not the reference edge (and the reference edge might be marked),
- Case 4) two edges are marked, which are not the reference edge (and the reference edge might be marked) or the element is marked.

In Case 1, the element remains untouched. In Case 2, this results into two new triangles and complies with the green refinement and is therefore painted in green. Contrary to the definition of a green refinement, we need to store the new reference edges which are the edges opposite to the newest node. In Case 3, we need to ensure that the reference edge is marked and then get a blue refinement. Again, the new reference edges are the edges opposite to the newest vertex and shown in Figure 3.16 as hatched lines. If two edges are marked, which are not the reference edge, all edges are marked or the element itself is marked, we get a red_{NVB} refinement. Here, not the midpoints of the bisected edges are connected with each other, but in an iterative process the newest vertex is bisected as long as all hanging nodes are removed and thus the result is Case 4.

The main difference to TrefineRG is that we do not remove green or blue refinements anymore. Hence, all elements in the refined mesh are painted in red. We want to present the advantages of this procedure in the next section.

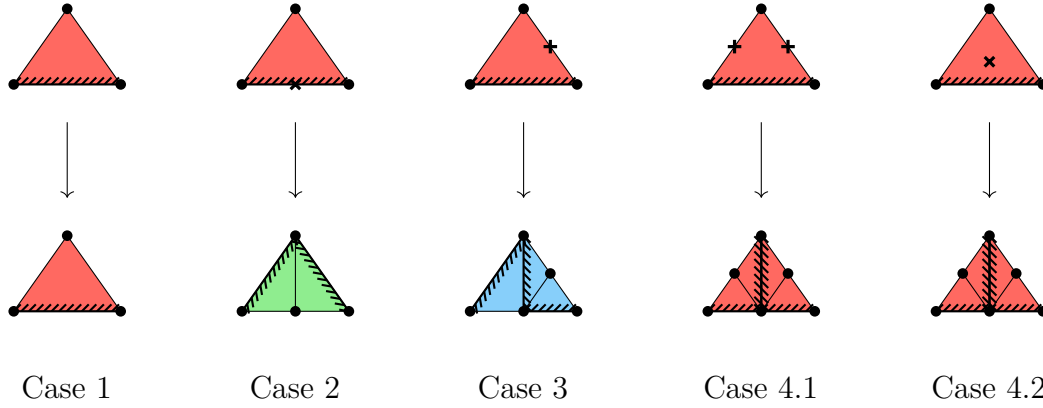


Figure 3.16: Exemplary illustration of TrefineNVB: refinement. The reference edge is emphasized with a hatched line. In Case 1, the triangle remains unaffected. In Case 2, the element is green refined, new reference edges are determined. In Case 3, the reference edge might be marked or not, but we need to ensure that the reference edge is marked and thus the element is blue refined. In the last case we get a red_{NVB} refinement if two edges which are not the reference edge are marked, all edges are marked or the element itself is marked. Contrary to the typical red refinement, the midpoints are not connected with each other, but we bisect the newest node as long as all hanging nodes are removed through this process.

3.4.2 Properties

In TrefineRG, we always removed the green refinement before bisecting further and thus obtained a non-nested grid. To overcome this issue, in TrefineNVB we do not remove green and blue refinements. However, showing the shape regularity is not as obvious as it was in the last strategy. Sewell showed in [29] that with the newest vertex bisection the descendants of a triangle can be grouped into four similarity classes as illustrated in Figure 3.17. Hence, NVB leads to at most $4 \cdot \#$ of triangles in \mathcal{T}_0' and therefore a uniform lower bound for all interior angles in \mathcal{T}_ℓ can be found. Thus, the minimum and maximum angle conditions are satisfied for all descendants. Obviously, the resulting mesh is a regular triangulation. We remark:

Remark 3.4.2 *The meshes generated by TrefineNVB are nested, regular and hold the shape regularity.*

Note, that instead of bisecting the newest node, one could also imagine to bisect the longest edge. This version has been firstly investigate by Rosenberg and Stenger in [28]. They prove that it holds $\theta \geq \frac{\alpha_0}{2}$ for all interior angles θ of a triangulation \mathcal{T} generated by the longest edge bisection process. Here, α_0 denotes the smallest angle in the initial grid \mathcal{T}_0 . Further results are provided in [27].

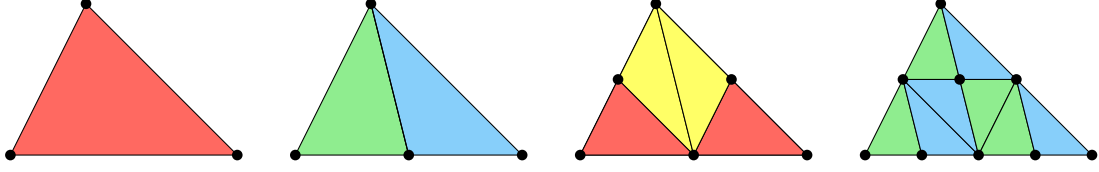


Figure 3.17: Four similarity classes in TrefineNVB. During the refinement process in the newest vertex bisection the triangles fall into four similarity classes, pictured in different colors.

3.4.3 On the Choice of the Reference Edge in the Initial Mesh

In this section, we discuss possible choices of reference edges and their influence in the completion process. The goal of adaptive finite element methods is to refine only locally where it is necessary. However, removing hanging nodes during the refinement process can inflate the number of refined elements. Therefore, a proper choice of the reference edges can prevent from refining too strongly.

3.4.3.1 Longest Edge

A very intuitive choice is the longest edge of a triangle. The triangle is always split at the vertex opposite to the reference edge and in the case of the longest edge, the largest angle is subdivided. This is favorable to maintain a shape regular mesh throughout the whole refinement process. However, the choice of the longest edge can cause a chain of further refinements; see Figure 3.18. Assume that the triangle with the red cross is marked for refinement. We would then successively refine *all* triangles due to the location of the reference edges. Thus, the completion process may inflate the number of refined elements if the initial mesh is unfortunate. Also, there's no rule on how to choose the reference edge in the case of multiple edges with maximum length. Hence, in this event the reference edge is randomly picked out from the set of longest edges.

3.4.3.2 Reference Edges à la Carstensen

Another choice of reference edges is proposed by Carstensen in [16]. His concept is based on the term *isolated elements*.

Definition 3.4.3 (Isolated element) *An element T in \mathcal{T}_0 is called isolated if the reference edge $\text{ref}_E(T)$ of triangle T is shared with another triangle $K \in \mathcal{T}_0$, i.e.*

$T \cap K = \text{ref}_E(T)$, but for the reference edge $\text{ref}_E(K)$ of triangle K holds $\text{ref}_E(K) \neq \text{ref}_E(T)$.

Carstensen proposes an algorithm that generates the reference edges such that two distinct isolated elements do not share an edge. This is done in the following way; see Algorithm 3.3 ([16, Algorithm 2.1]).

Algorithm 3.3: Reference Edges à la Carstensen [16]

INPUT: A triangulation \mathcal{T}_0 .

Set $\mathcal{T}^{(0)} := \mathcal{T}_0$ and $j := 0$.

While $\mathcal{T}^{(j)} \neq \emptyset$.

- choose an element $T \in \mathcal{T}^{(j)}$ and find another element $K \in \mathcal{T}^{(j)}$ such that $T \cap K$ is a common edge
- if such $K \in \mathcal{T}^{(j)}$ exists choose one of them and set their common edge $T \cap K = \text{ref}_E(T) = \text{ref}_E(K)$ as the reference edge of both elements. Set $\mathcal{T}^{(j+1)} := \mathcal{T}^{(j)} \setminus \{T, K\}$ and $j := j + 1$
- if no such K exists, choose some edge of T as reference edge and define $\mathcal{T}^{(j+1)} := \mathcal{T}^{(j)} \setminus \{T\}$, $j := j + 1$

OUTPUT: A set of reference edges $(\text{ref}_E(T))_{T \in \mathcal{T}_0}$.

Possible choices of reference edges can be seen in Figure 3.19. The proposed algorithm does not return a unique allocation of reference edges. The choice is dependent of the sequence of chosen elements $T \in \mathcal{T}^{(j)}$ in the algorithm. However, it ensures that two distinct isolated elements do not share an edge. We see that this choice of reference edges is more favorable as the longest edge since it prevents from refining a chain of triangles, see Figure 3.19.

3.4.3.3 Reference Edges à la Stevenson

In [13], Binev, Dahmen, and DeVore address the question whether the overall number of triangles created through the completion process stays proportional to the number of elements marked throughout the refinement process. The reference edges in \mathcal{T}_0 are chosen in the following way: Let T and K be two different elements of \mathcal{T}_0 . If $T \cap K$ is an edge, it holds that this edge is either the reference edge of both triangles T and

K or of none. Binev et al. proved that such a distribution of reference edges exists in \mathbb{R}^2 but the proof is not constructive and therefore they do not provide an algorithm that creates a mesh fulfilling this property. However, under this assumption on the choice of reference edges they showed that the refinement process using newest vertex bisection fulfills the following property.

Theorem 3.4.4 (Quasi-optimality of mesh-closure [13, Theorem 2.4])

Let \mathcal{T}_0 be an initial partition with reference edges located as described above. Let $\mathcal{T}_1, \dots, \mathcal{T}_n$ be a sequence of partitions generated through the newest vertex bisection. Then, there exists a constant c depending on the shape regularity constant of the initial mesh \mathcal{T}_0 such that

$$\#\mathcal{T}_\ell - \#\mathcal{T}_0 \leq c \sum_{j=0}^{\ell-1} \#\mathcal{M}_j \quad \forall \ell \in \mathbb{N}.$$

Here, $\mathcal{M}_j \subseteq \mathcal{T}_j$ denotes an arbitrary set of marked elements in the refinement step j .

Consequently, the number of elements in \mathcal{T}_ℓ is controlled.

In [31], Stevenson proposes an approach that leads to a mesh that fulfills the described property. To this end, we first subdivide each triangle of the initial mesh \mathcal{T}_0 into three subtriangles by connecting each vertex with the centroid of this triangle. We define the outer edges of the father triangle as the new reference edges of the subtriangles. Thus, we ensure the property, that a common edge is either the reference edge of both triangles or of none. The approach is outlined in Algorithm 3.4.

Algorithm 3.4: Reference Edges à la Stevenson

INPUT: A triangulation \mathcal{T}_0 .

Set $\mathcal{T}_0^{new} := \emptyset$.

For each element $T = \triangle z_1 z_2 z_3$ in \mathcal{T}_0 :

- find the centroid c of triangle T
- refine triangle T into three new triangles $T_1 = \triangle z_1 z_2 c$, $T_2 = \triangle z_2 z_3 c$ and $T_3 = \triangle z_3 z_1 c$
- set $\mathcal{T}_0^{new} := \mathcal{T}_0^{new} \cup T_1 \cup T_2 \cup T_3$

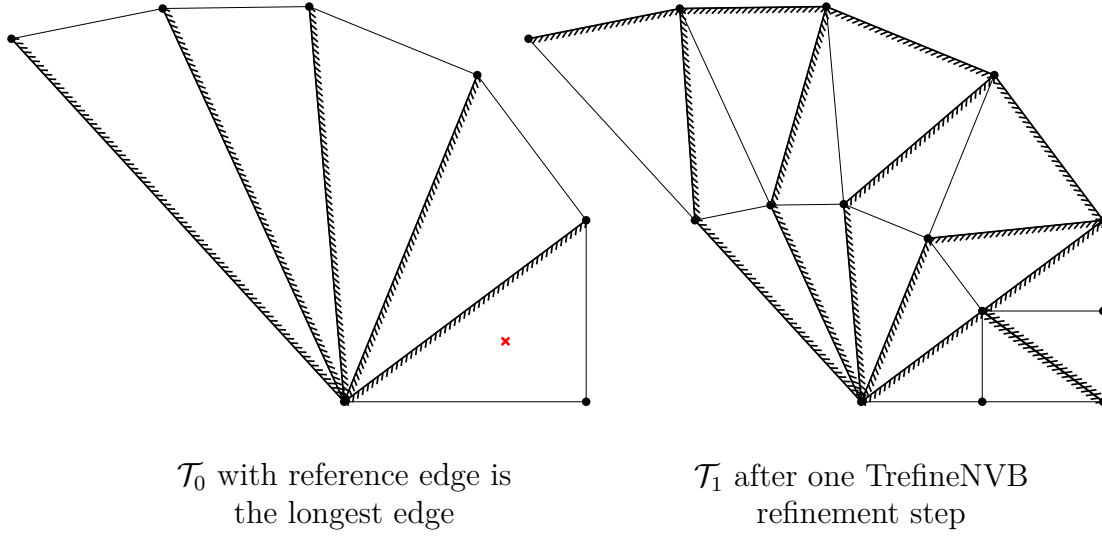


Figure 3.18: Reference edge is longest edge (illustrated as hatched lines). By marking one element, indicated by the red cross, a chain of further refinement is entailed.

- set $\text{ref}_E(T_1) := \overline{z_1 z_2}$, $\text{ref}_E(T_2) := \overline{z_2 z_3}$ and $\text{ref}_E(T_3) := \overline{z_3 z_1}$

OUTPUT: Triangulation with reference edges $(T_0^{\text{new}}, (\text{ref}_E(T))_{T \in \mathcal{T}_0^{\text{new}}})$.

Although the mesh proposed à la Stevenson is no longer as coarse as the initial mesh, using reference edges à la Stevenson prevents a chain of further refinements in the completion process; see Figure 3.20 for illustration.

3.5 TrefineRGB

Another refinement strategy is constructed by using the newest vertex bisection but replacing the red_{NVB} refinement with the red refinement from Definition 3.1.1. This idea has been mentioned by Carstensen in [16]. In this method, the use of reference edges and generation of nested meshes originate from TrefineNVB. Since the resulting patterns are the green, blue and red refinements defined in the beginning of this section, this method is called TrefineRGB. We refer to Figure 3.21 for an illustration of allowed patterns and corresponding reference edges. The discussion on the choice of reference edges carries over from Section 3.4.3. Algorithm 3.5 is slightly modified to Algorithm 3.2.

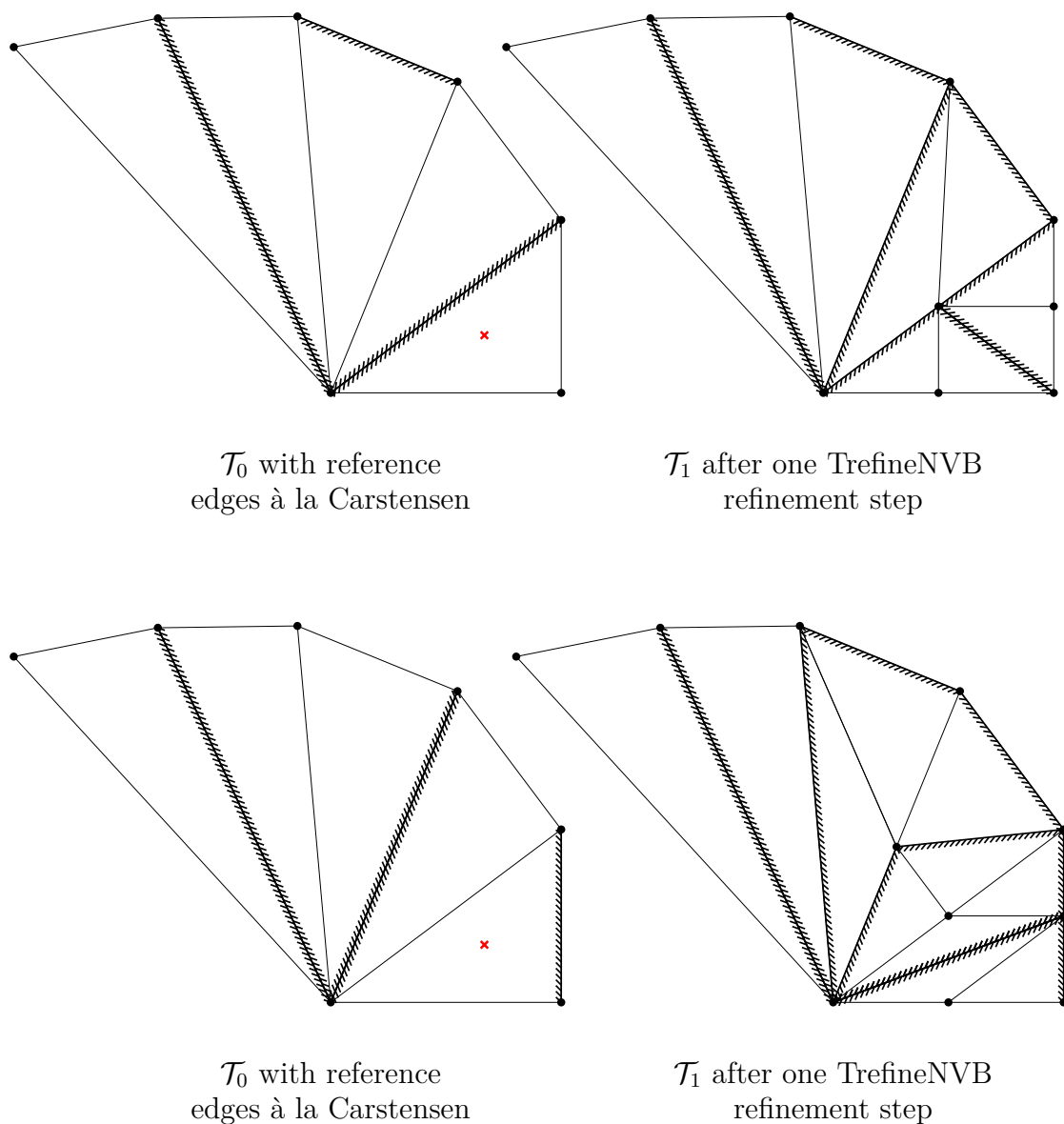


Figure 3.19: Two conceivable choices of reference edges à la Carstensen (indicated by hatched lines). Algorithm 3.3 does not provide a unique choice of the reference edges. Thus, more combinations are possible as the ones depicted. One can see that these choices are more favorable because the chain of further refinements starting in one marked element stops in each direction after at most 2 further refined triangles. This is true because the algorithm provides two isolated distinct elements that do not share an edge.

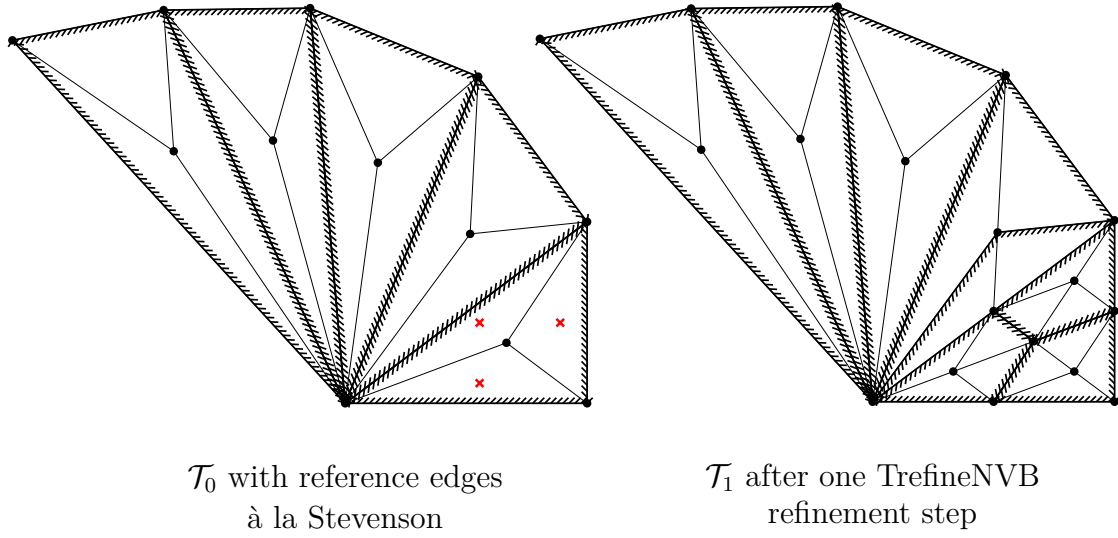


Figure 3.20: Reference edges à la Stevenson (depicted as hatched lines). If the reference edges are chosen à la Stevenson, the elements will first all be refined by connecting the vertices of the triangle with its centroid. The outer edges are then the new reference edges. In this case, all three triangles of the marked father element are marked and it is obvious, that the chain of further refinements also stops with this choice.

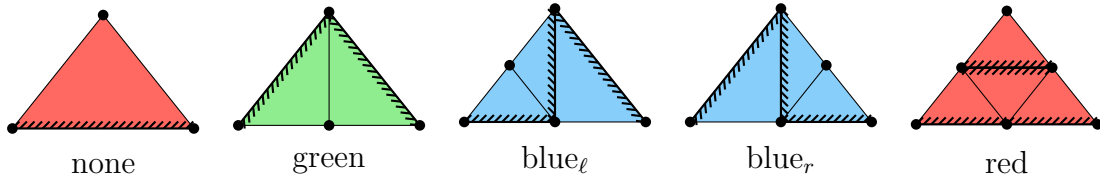


Figure 3.21: Refinement patterns for TrefineRGB. Reference edges are highlighted by hatched lines.

Algorithm 3.5: TrefineRGB

INPUT: A triangulation with reference edges $(\mathcal{T}_\ell, (\text{ref}_E(T))_{T \in \mathcal{T}_\ell})$ and a non-empty set of marked elements $\mathcal{M}_\ell \subseteq \mathcal{T}_\ell$.

Set $\mathcal{E}_{\mathcal{M}_\ell}^{(0)} = \emptyset, k := 1$ and define the set of marked edges $\mathcal{E}_{\mathcal{M}_\ell}^{(k)} := \{\mathcal{E}(T) : T \in \mathcal{M}_\ell\}$.

While $\mathcal{E}_{\mathcal{M}_\ell}^{(k)} \neq \mathcal{E}_{\mathcal{M}_\ell}^{(k-1)}$:

- define $\mathcal{E}_{\mathcal{M}_\ell}^{(k+1)} := \mathcal{E}_{\mathcal{M}_\ell}^{(k)} \cup \{\text{ref}_E(T) : \forall T \in \mathcal{T}_\ell \text{ with } \mathcal{E}(T) \cap \mathcal{E}_{\mathcal{M}_\ell}^{(k)} \neq \emptyset\}$
- set $k := k + 1$

Refine each triangle $T \in \mathcal{T}_\ell$

- with $|\mathcal{E}_{\mathcal{M}_\ell}^{(k)}(T)| = 3$: red
- with $|\mathcal{E}_{\mathcal{M}_\ell}^{(k)}(T)| = 2$: blue
- with $|\mathcal{E}_{\mathcal{M}_\ell}^{(k)}(T)| = 1$: green

For each refinement specify the new reference edges.

OUTPUT: The triangulation with reference edges $(\mathcal{T}_{\ell+1}, (\text{ref}_E(T))_{T \in \mathcal{T}_{\ell+1}})$.

Henceforth, we illustrate the procedure of TrefineRGB.

3.5.1 Description of TrefineRGB

In the same manner as in TrefineNVB, we choose a reference edge for each element, e.g. the longest edge and consider this as the edge opposite to the newest vertex. From then on, we mark elements by marking each edge for bisection. And thus the same cases as in TrefineNVB arise:

Case 1) no edge is marked,

Case 2) only the reference edge is marked ,

Case 3) one edge is marked, which is not the reference edge (and the reference edge might be marked),

Case 4) two edges are marked, which are not the reference edge (and the reference edge might be marked) or the element is marked.

As the procedure is the same except for Case 4, we only focus on this refinement. We have already mentioned that this is replaced by the regular red refinement. However, we still need to determine the new reference edge of each triangle. Since this can not be done by choosing the edge opposite to the newest vertex, we indicate the choice of the reference edge by hatched lines, as shown in Figure 3.22. Note, that all elements of the refined mesh are painted in red because the algorithm does not differentiate between red, green oder blue elements.

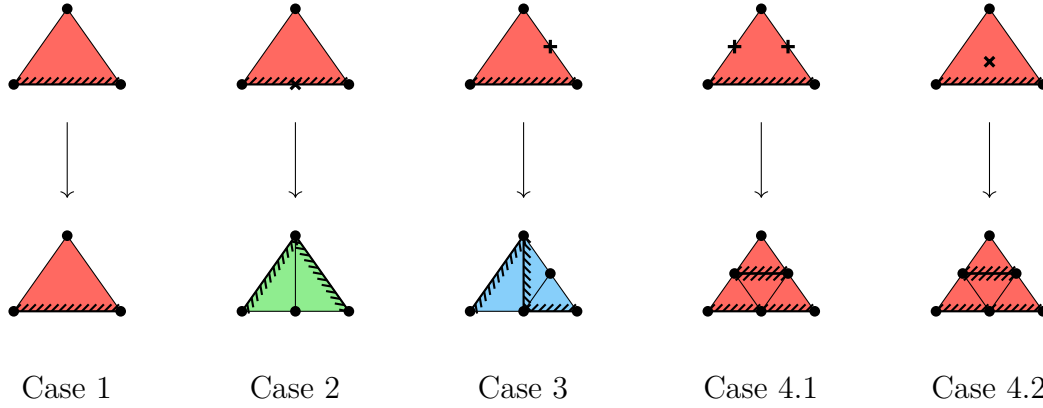


Figure 3.22: Exemplary illustration of TrefineRGB: refinement. The procedure is the same as in TrefineNVB except for Case 4, compare Figure 3.16. In this case, we refine red and set the new reference edges as indicated by the hatched lines.

3.5.2 Properties

Remark 3.5.1 *The meshes generated by TrefineRGB are nested, regular and hold the shape regularity.*

The first two properties are obvious. The shape regularity of this refinement strategy can be proven with the help of similarity classes. We have already mentioned that a red refinement results into four new triangles that are similar to its father element. Also, by bisecting the edges and connecting them to the opposite vertex, at most four similarity classes arise. By going through all possible refinements of each similarity class, we see that all descendant triangles fall into the same four similarity classes, as depicted in Figure 3.23. Thus, the red–green–blue refinement leads to at most $4 \cdot \#$ 'of triangles in \mathcal{T}_0 '. Hence, a uniform lower bound for all interior angles in \mathcal{T}_ℓ can be found. The minimum and maximum angle conditions are satisfied for all descendants.

3.6 Preliminaries for Quadrilateral Meshes

From now on, we consider meshes with quadrilateral elements. In the **REFINE** step, we have a triangulation \mathcal{T}_ℓ into quadrilaterals and a set of marked elements \mathcal{M}_ℓ ($\ell = 0, 1, 2, \dots$). There are two possible understandings of element marking. We first discuss possibility number one: A marked element is quadrisected into four new quadrilaterals. Thus, for this approach all four edges of the quadrilateral are bisected. There exist different refinement rules, proposed by Bank, Sherman, and

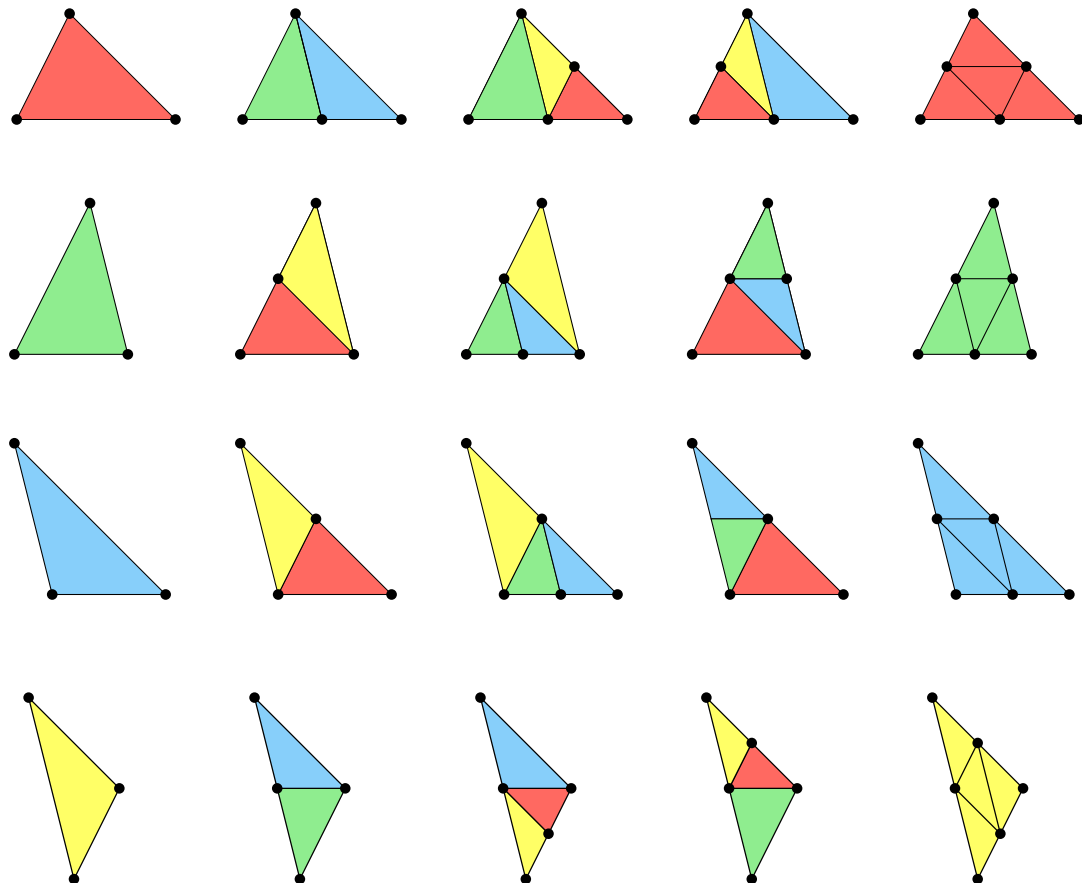


Figure 3.23: Four similarity classes in TrefineRGB. For each similarity class, we illustrate the possible refinements. Here, each row illustrates the possible refinements of the first triangle in this row. The colors display the corresponding similarity classes.

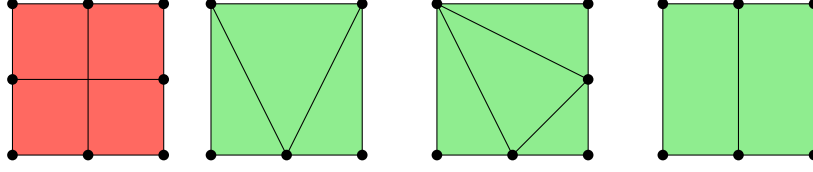


Figure 3.24: Red, green of type 1, green of type 2, and green of type 3 refinement rules for a quadrilateral.

Weiser [8].

Definition 3.6.1 (Red refinement of a quadrilateral) A quadrilateral T is subdivided into four quadrilaterals by connecting the midpoints of opposite edges with each other. This is called a red refinement.

Definition 3.6.2 (Green refinement of type 1 for quadrilaterals) A quadrilateral T is subdivided into three triangles by connecting the midpoint of one edge with the vertices of the opposite edge. We call this a green refinement of type 1.

Definition 3.6.3 (Green refinement of type 2 for quadrilaterals) A quadrilateral T is subdivided into four triangles by connecting the midpoints of two adjacent edges and the common vertex of the other two edges with each other. We call this a green refinement of type 2.

Definition 3.6.4 (Green refinement of type 3 for quadrilaterals) A quadrilateral is subdivided into two quadrilaterals by connecting the midpoints of two opposite edges. We call this a green refinement of type 3.

We refer to Figure 3.24 for an illustration. A disadvantage of these refinement types is that the resulting mesh is a mixture of triangles and quadrilaterals and therefore the data structure is complicated. To overcome this issue Mao, Zhao, and Shi proposed a different understanding of element marking in [35]. They split a marked quadrilateral by enneasection, i.e. the quadrilateral is subdivided into nine quadrilaterals. Thus, the second possibility is to trisect all edges of a marked quadrilateral. An advantage of this procedure is that now green refinement patterns with quadrilaterals only can be defined. Therefore, we get the following refinement rules.

Definition 3.6.5 (Red2 refinement of a quadrilateral) A quadrilateral is subdivided into 9 quadrilaterals by trisecting each edge and connecting opposite lying nodes with each other. This is called a red2 refinement.

Definition 3.6.6 (Green2 refinement of type 1 of a quadrilateral)

One edge of a quadrilateral $T = \square P_1 P_2 P_3 P_4$ is trisected, e.g. $\overline{P_1 P_2}$ with resulting new nodes $N_1 = \frac{1}{3}(2P_1 + P_2)$ and $N_2 = \frac{1}{3}(P_1 + 2P_2)$. Furthermore, new midpoints $M_1 = (2P_1 + P_2 + 2P_3 + 4P_4)/9$ and $M_2 = (P_1 + 2P_2 + 4P_3 + 2P_4)/9$ are introduced. The quadrilateral is split into four child-quadrilaterals T_1, T_2, T_3 , and T_4 with $T_1 = \square P_1 N_1 M_1 P_4$, $T_2 = \square N_1 N_2 M_2 M_1$, $T_3 = \square P_2 P_3 M_2 N_2$, and $T_4 = \square P_3 P_4 M_1 M_2$. We call this refinement a green2 refinement of type 1.

Definition 3.6.7 (Green2 refinement of type 2 of a quadrilateral)

Two adjacent edges of a quadrilateral $T = \square P_1 P_2 P_3 P_4$ are trisected, e.g. $\overline{P_1 P_2}$ with new nodes $N_1 = \frac{1}{3}(2P_1 + P_2)$, $N_2 = \frac{1}{3}(P_1 + 2P_2)$ and $\overline{P_2 P_3}$ with new nodes $N_3 = \frac{1}{3}(2P_2 + P_3)$, $N_4 = \frac{1}{3}(P_2 + 2P_3)$. Furthermore, new midpoints $M_1 = (2P_1 + P_2 + 2P_3 + 4P_4)/9$ and $M_2 = (2P_1 + 4P_2 + 2P_3 + P_4)/9$ are introduced. The quadrilateral is split into five child-quadrilaterals T_1, T_2, T_3, T_4 , and T_5 with $T_1 = \square P_1 N_1 M_1 P_4$, $T_2 = \square N_1 N_2 M_2 M_1$, $T_3 = \square P_2 N_3 M_2 N_4$, $T_4 = \square N_4 M_1 M_2 N_3$, and $T_5 = \square P_3 P_4 M_1 N_4$. We call this refinement a green2 refinement of type 2.

Definition 3.6.8 (Green2 refinement of type 3 of a quadrilateral)

A quadrilateral T is subdivided into three quadrilaterals by trisecting two opposite lying edges and connecting the opposite lying midpoints of these edges with each other. We call this a green refinement of type 3.

The refinement patterns are visualized in Figure 3.25. In the following, we mix these refinement rules in different ways and investigate their properties. Since the color of the elements plays an important role, we have the convention that all elements in the initial triangulation \mathcal{T}_0 are painted in red. For all strategies, we wish to maintain the regularity of the grid. In cases in which this is not possible we follow the 1-Irregular Rule for the red refinement and the 2-Irregular Rule for the red2 refinement. These rules say: *Refine any unrefined element that has more than one (two) hanging nodes on an edge.* This rule is known from [8] and we also want to use their proposed 3-Neighbor Rule: *Refine any element with three neighbors that have been red/red2 refined;* see Figure 3.26 for an illustration. These two rules ensure that there are at most four basis functions having support in one element and thus the stiffness matrix is sparse; compare [8, Q1-Q7].

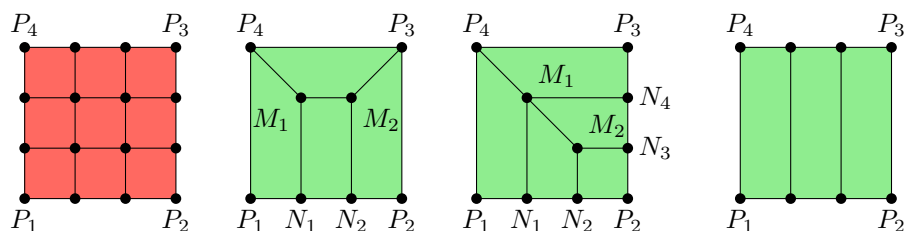


Figure 3.25: Red2, green2 of type 1, green2 of type 2 and green2 of type 3 refinement rules for a quadrilateral.

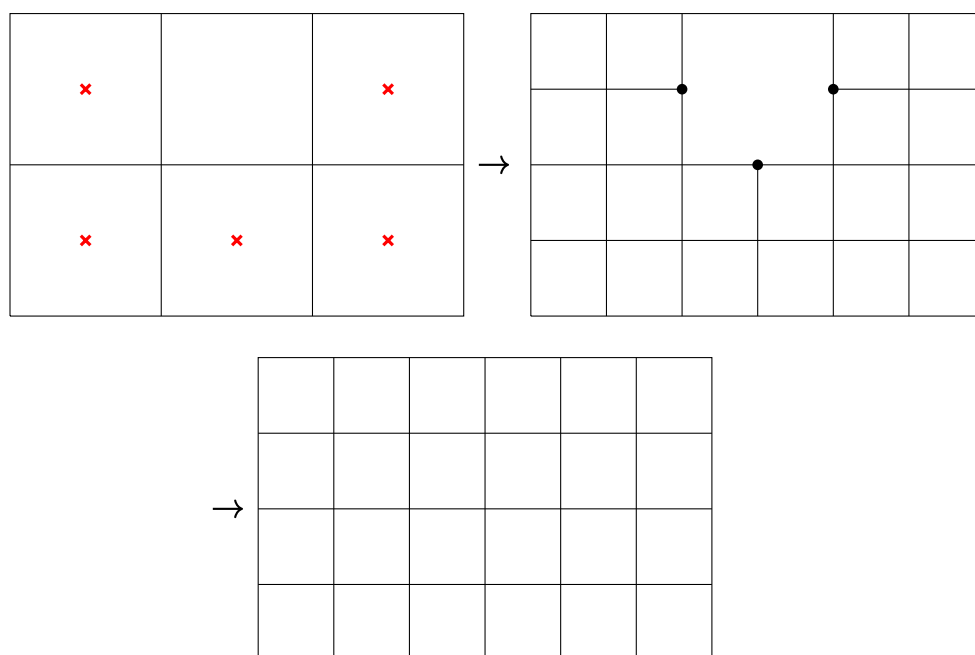


Figure 3.26: The 3-Neighbor Rule. The red crosses indicate that the elements are marked for refinement. The non-marked element has three neighbors that are marked for refinement. The 3-Neighbor Rule tells to also refine this element, i.e. in this example the mesh is uniformly refined even though not all elements were marked for refinement.

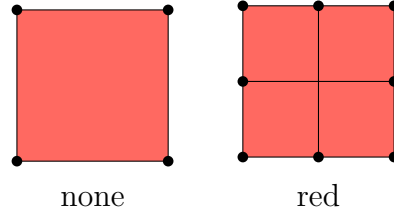


Figure 3.27: Refinement patterns for QrefineR.

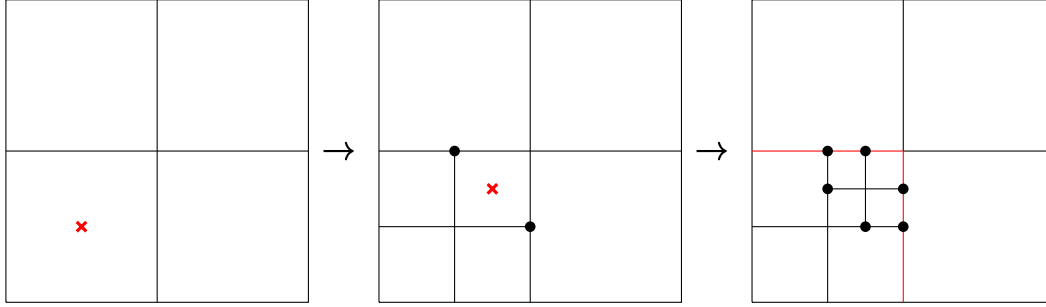


Figure 3.28: A straightforward application of the two refinement patterns in QrefineR. An element is marked for refinement as indicated through a red cross. By applying the two refinement patterns, we receive a grid in the second refinement that is not 1-irregular any more. The edges colored in red have two hanging nodes. Thus a straightforward application of the two refinement patterns does not suffice.

3.7 QrefineR

In QrefineR only a red refinement is allowed. According to this, the acknowledged patterns are shown in Figure 3.27. Obviously, applying the red refinement rule without having any completion rules like the green refinement leads to an irregular grid. A naive way to apply these rules is shown in Figure 3.28. However, we see that this method introduces more than one hanging node per edge. Hence, in this case the 1-Irregular Rule is deployed to resolve this issue as depicted in Figure 3.29. To this end, it is important to distinguish between red elements with hanging nodes and without hanging nodes. For clarification, we paint red elements with hanging nodes in gray. The formal procedure of QrefineR is described through three components: the refinement patterns shown in Figure 3.27, the 1-Irregular and the 3-Neighbor Rule. Furthermore, a detailed description of QrefineR is given in the following section.

3.7.1 Description of QrefineR

It is already known that a distinction between red and gray elements is necessary. A red element is affected by the following cases:

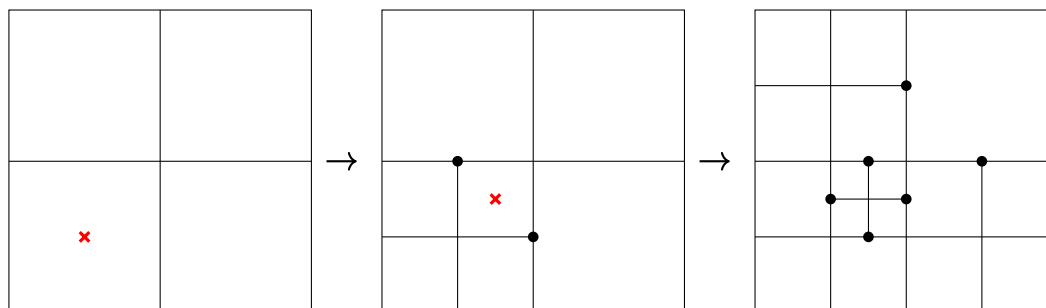


Figure 3.29: Resolving the issues of a straightforward application of the two refinement patterns in QrefineR. We deploy the 1-Irregular Rule which says that any unrefined element that has more than one hanging node is refined. Thus, the 1-irregularity of the grid is ensured.

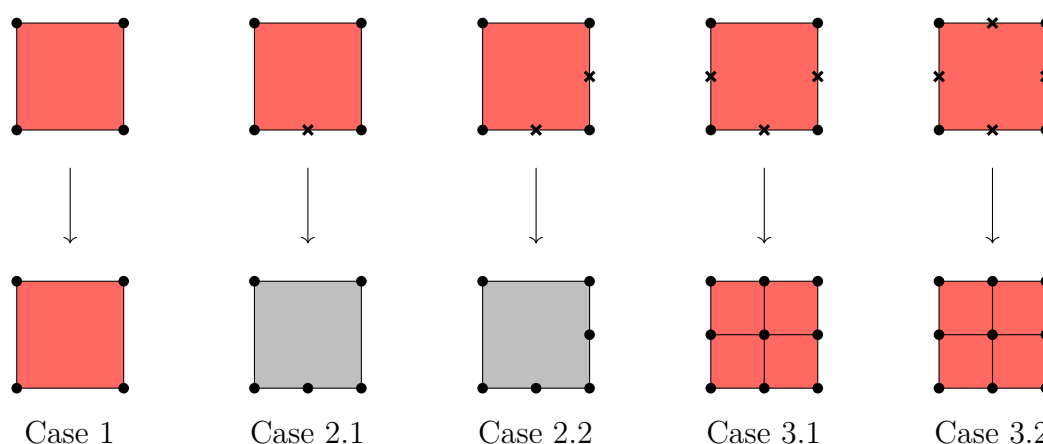


Figure 3.30: Exemplary illustration of QrefineR: refinement of red elements. In Case 1, the quadrilateral remains unaffected. In Case 2, hanging nodes are introduced. In Case 3, the element is red refined.

Case 1) no edge is marked,

Case 2) one edge or two edges are marked through neighboring elements,

Case 3) three or more edges are marked through neighboring elements or the element itself is marked.

In Case 1, the element remains unchanged. In Case 2, hanging nodes are created as the midpoints of the marked edges. If three or more edges are marked through neighboring elements or element marking (Case 3), this element is red refined. This procedure is depicted in Figure 3.30. Red elements are painted in red, whereas elements with hanging nodes are painted in gray.

Gray elements can also be affected by further markings. The following cases arise:

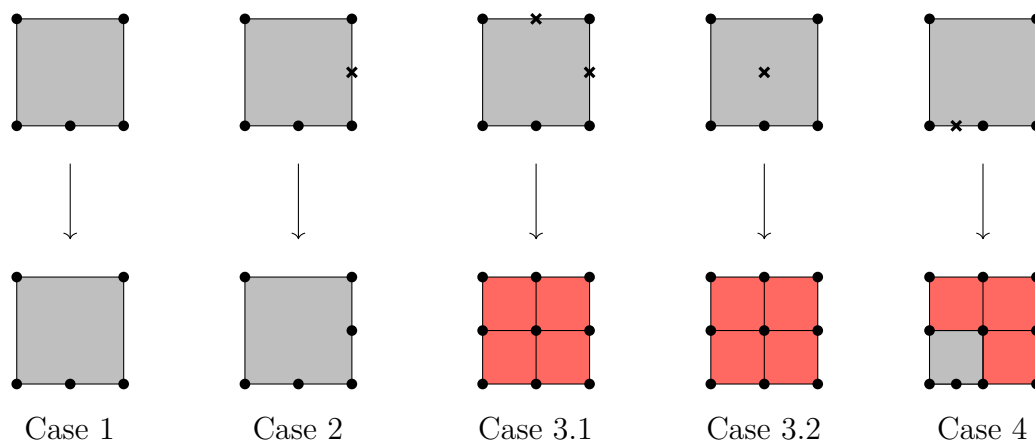


Figure 3.31: Exemplary illustration of QrefineR: refinement of irregular elements. In Case 1, the element remains unaffected. In Case 2, a further node is introduced.

In Case 3, the element is red refined if a refinement would lead to at least three hanging nodes. In Case 4, an irregular edge is marked and thus the element is first red refined before a new hanging node is introduced.

Case 1) no edge is marked,

Case 2) one regular edge is marked,

Case 3) two or more regular edges are marked by neighboring elements or the element itself is marked,

Case 4) at least one irregular edge is marked.

In Case 1, the element stays the same. In Case 2, a further hanging node is generated. If at least two regular edges are marked or the element itself is marked (Case 3), we refine red. In Case 4, at least one irregular edge is marked and thus the 1-Irregular Rule is deployed. Hence, the 1-irregularity of the grid is ensured by first refining red to remove hanging nodes and afterwards introducing a new hanging node on the irregular edge. These situations are depicted in Figure 3.31.

3.7.2 Properties

The triangulations \mathcal{T}_ℓ outputted from QrefineR have the following properties.

Remark 3.7.1 *The meshes generated by QrefineR are nested, 1-irregular and hold the shape regularity.*

The resulting mesh in QrefineR is obviously a 1-irregular mesh because at most one

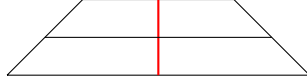


Figure 3.32: Main idea of proving the shape regularity for the red refinement of quadrilaterals. One can see that the red line is shorter than any other outer edge of the quadrilateral.

hanging node per edge is allowed. Also, the mesh is nested because only further nodes are added but never removed. The shape regularity can be proven in the same manner as done in [35] from Zhao, Mao, and Shi for a similar problem. The proof is carried out in the work of Beuter [12]. Let us mention the main ideas of the proof. Let therefore T be a quadrilateral with smallest edge \underline{h}_T and T_i the child-quadrilaterals after a red refinement for $i = 1, 2, 3, 4$ with smallest edges \underline{h}_{T_i} . The inequality $\underline{h}_{T_i} \geq \frac{1}{2}\underline{h}_T$ does not always hold; as illustrated in Figure 3.32. The red line is shorter than all other edges of quadrilateral T . Hence, for the smallest edge after refinement the inequality $\underline{h}_{T_i} \geq \frac{1}{2}\underline{h}_T$ does *not* hold $\forall T \in \mathcal{T}_h$. By defining a continuous function on a compact interval, the extreme value theorem provides a lower bound for \underline{h}_{T_i} in dependence of the initial triangulation \mathcal{T}_0 . Thus, the relation $\frac{\bar{h}_T}{\underline{h}_T}$ can be bounded from above $\forall T \in \mathcal{T}_h$. The angle condition can easily be shown by means of trigonometric identities. Hence, the shape regularity for quadrilaterals is ensured.

3.8 QrefineRGtri

We wish to find patterns that complete the refinement of quadrilaterals and remove hanging nodes. We still follow the 3-Neighbor Rule, i.e. only patterns for one marked edge, two opposite lying marked edges, and two adjacent marked edges are needed. In all other cases, the element is red refined. An intuitive choice is proposed by Bank, Sherman, and Weiser in [8] and corresponds to our definitions of a green refinement of type 1, 2, and 3. This leads to a mixed mesh of triangles and quadrilaterals; see Figure 3.33. Thus, the data structure is complicated and the crossover from a triangle to a quadrilateral needs to be considered in practical implementations. To ensure the shape regularity of the grid, all green refinements are coarsened before bisected further. The method is given in Algorithm 3.6.

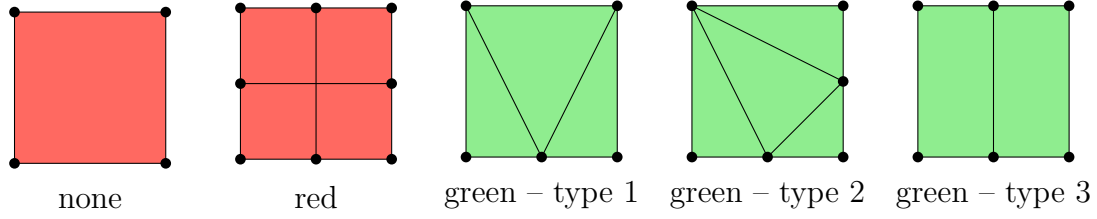


Figure 3.33: Refinement patterns for red elements in QrefineRGtri. From left to right: A red element and its refinement patterns. Green refinements are proposed by Bank, Sherman, and Weiser [8]. All green refinements can also be applied for rotated locations of hanging nodes. Thus, there exist four different green – type 1 refinements, four different green – type 2 refinements and two different green – type 3 refinements.

Algorithm 3.6: QrefineRGtri

INPUT: A triangulation $(\mathcal{T}_\ell^{\text{red}}, \mathcal{T}_\ell^{\text{green}}, (K_T)_{T \in \mathcal{T}_\ell^{\text{green}}})$ and a set of marked elements $\mathcal{M}_\ell \subseteq \mathcal{T}_\ell = \mathcal{T}_\ell^{\text{red}} \cup \mathcal{T}_\ell^{\text{green}}$ with $\mathcal{T}_\ell^{\text{green}} = \mathcal{T}_\ell^{\text{green t1}} \cup \mathcal{T}_\ell^{\text{green t2}} \cup \mathcal{T}_\ell^{\text{green t3}}$. The triangulation consists of red and green elements and for each green element the father element K_T is specified. There exists three different types of green refinements.

Set $\mathcal{E}_{\mathcal{M}_\ell}^{(0)} = \emptyset$, $k := 1$ and define the set of marked edges

$$\mathcal{E}_{\mathcal{M}_\ell}^{(k)} := \left\{ \mathcal{E}(T) : T \in \mathcal{M}_\ell \cap \mathcal{T}_\ell^{\text{red}} \right\} \cup \left\{ \mathcal{E}(K) : K \text{ is father element of } T, T \in \mathcal{M}_\ell \cap \mathcal{T}_\ell^{\text{green}} \right\}.$$

While $\mathcal{E}_{\mathcal{M}_\ell}^{(k)} \neq \mathcal{E}_{\mathcal{M}_\ell}^{(k-1)}$:

- set $k = k + 1$ and $\mathcal{E}_{\mathcal{M}_\ell}^{(k)} = \mathcal{E}_{\mathcal{M}_\ell}^{(k-1)}$
- $\forall T \in \mathcal{T}_\ell^{\text{red}}$: find pattern in Figure 3.33 that introduces the least new marked edges and extend $\mathcal{E}_{\mathcal{M}_\ell}^{(k)}$ by the non-marked edges of T that need to be further refined due to the pattern
- $\forall T \in \mathcal{T}_\ell^{\text{green t1}}$: find pattern for the father element K_T in Figure 3.35 that introduces the least new marked edges and extend $\mathcal{E}_{\mathcal{M}_\ell}^{(k)}$ by the non-marked edges of K that need to be further refined due to the pattern
- $\forall T \in \mathcal{T}_\ell^{\text{green t2}}$: find pattern for the father element K_T in Figure 3.36 that introduces the least new marked edges and extend $\mathcal{E}_{\mathcal{M}_\ell}^{(k)}$ by the non-marked edges of K that need to be further refined due to the pattern

- $\forall T \in \mathcal{T}_\ell^{\text{green t3}}$: find pattern for the father element K_T in Figure 3.37 that introduces the least new marked edges and extend $\mathcal{E}_{\mathcal{M}_\ell}^{(k)}$ by the non-marked edges of K that need to be further refined due to the pattern

Define new triangulation $\mathcal{T}_{\ell+1}$ by matching patterns from Figure 3.33 for all $T \in \mathcal{T}_\ell^{\text{red}}$, from Figure 3.35 for all K_T with $T \in \mathcal{T}_\ell^{\text{green t1}}$, from Figure 3.36 for all K_T with $T \in \mathcal{T}_\ell^{\text{green t2}}$ and from Figure 3.37 for all K_T with $T \in \mathcal{T}_\ell^{\text{green t3}}$. Partition elements in the corresponding types and specify the father elements for all $T \in \mathcal{T}_{\ell+1}^{\text{green}}$.

OUTPUT: Triangulation into red and green elements $(\mathcal{T}_{\ell+1}^{\text{red}}, \mathcal{T}_{\ell+1}^{\text{green}} = \mathcal{T}_{\ell+1}^{\text{green t1}} \cup \mathcal{T}_{\ell+1}^{\text{green t2}} \cup \mathcal{T}_{\ell+1}^{\text{green t3}}, (K_T)_{T \in \mathcal{T}_{\ell+1}^{\text{green}}})$ with father elements K_T for all $T \in \mathcal{T}_{\ell+1}^{\text{green}}$.

A detailed description follows in the upcoming section.

3.8.1 Description of QrefineRGtri

For a red element the following cases are conceivable:

- Case 1) no edge is marked,
- Case 2) one edge is marked,
- Case 3) only two adjacent edges are marked,
- Case 4) only two opposite edges are marked,
- Case 5) three or more edges are marked through neighboring elements or the element itself is marked.

In Case 1, the element remains unchanged. In Case 2 - 4, we match one of the green refinements defined in the beginning of this elaboration; see Figure 3.33. If three or more edges are marked, the element is red refined (Case 5). An exemplary illustration of the cases is shown in 3.34.

For each type of green elements, there exist further possible refinement patterns. We do not provide a detailed version on all cases. However, all possible refinement patterns for each type are shown in Figures 3.35, 3.36, 3.37. The reader can easily comprehend the strategy through the declaration of all refinement patterns. We

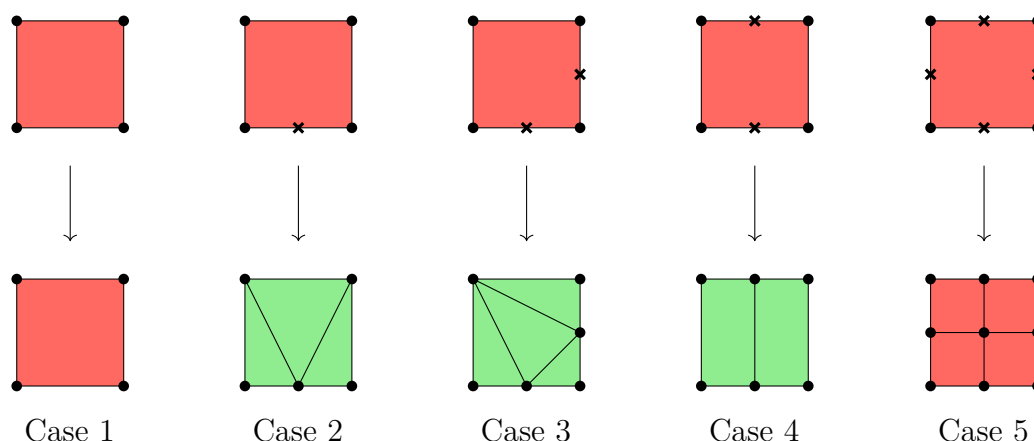


Figure 3.34: Exemplary illustration of QrefineR: refinement of red elements. In Case 1, the quadrilateral remains unaffected. In Case 2 –4, hanging nodes are eliminated by matching the appropriate refinement pattern. In Case 5, if at least three edges are marked the element is red refined.

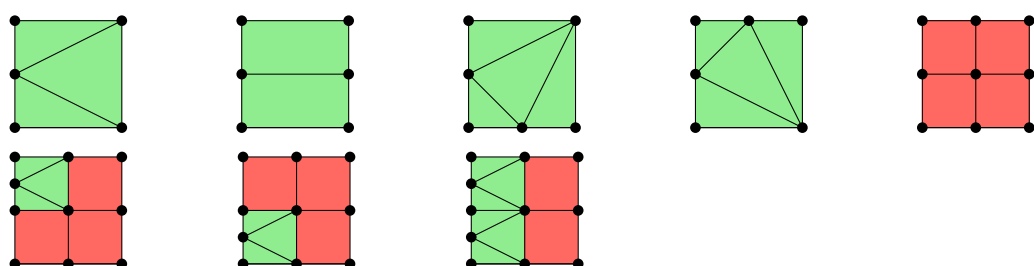


Figure 3.35: Refinement patterns for green elements of type 1 in QrefineRGtri. From top left to bottom right: A green element of type 1 and all possible refinement patterns.

notice that a green element of type 1 has eight refinement patterns whereas the number of cases reaches seventeen for green elements of type 2 and 3.

3.8.2 Properties

Remark 3.8.1 *The meshes generated by QrefineRGtri are non-nested and regular. They consist of a mixture of triangles and quadrilaterals and they all hold the shape regularity.*

During the refinement process all hanging nodes are removed and thus the mesh is regular. Due to the coarsening of green elements before bisecting further, edges are removed and the grid is not nested any more. The shape regularity for QrefineRGtri carries over from QrefineR. For this method, the shape regularity for red elements is

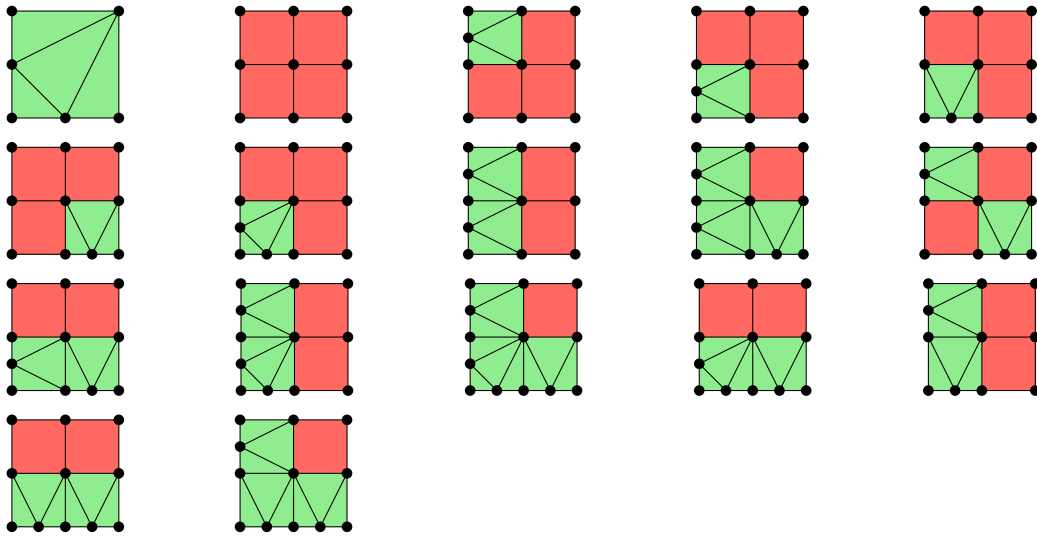


Figure 3.36: Refinement patterns for green elements of type 2 in QrefineRGtri.
 From top left to bottom right: A green element of type 2 and all possible refinement patterns.

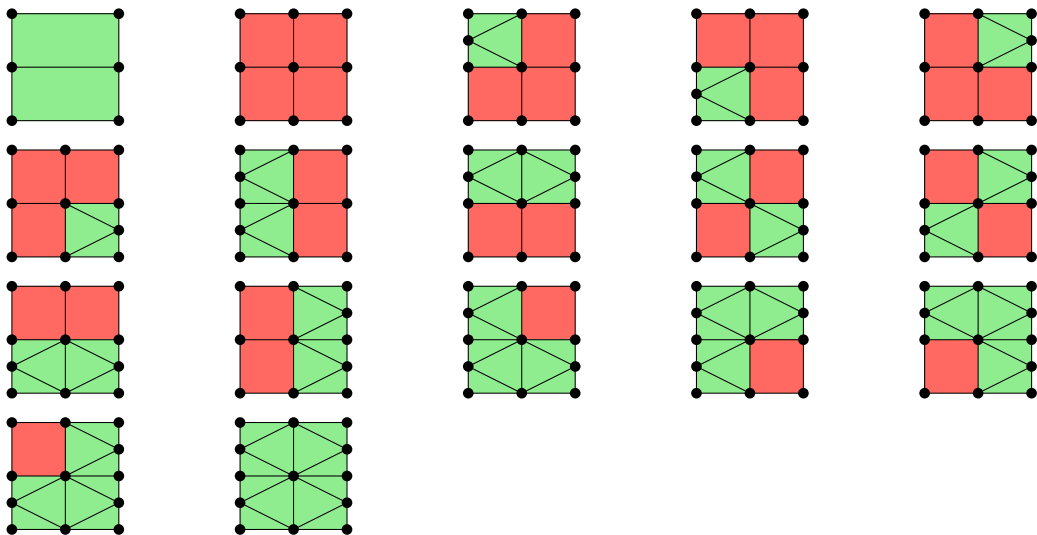


Figure 3.37: Refinement patterns for green elements of type 3 in QrefineRGtri.
 From top left to bottom right: A green element of type 3 and all possible refinement patterns.

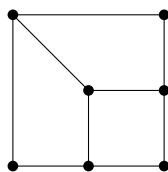


Figure 3.38: Proposed pattern for QrefineRB.

verified. The shape regularity of green elements only depends on the shape regularity of the father element because a green element is coarsened to its father element before it is bisected further. This can be proven easily. The property that the resulting mesh is a mixture of triangles and quadrilaterals is clear from the definitions of the allowed green refinements.

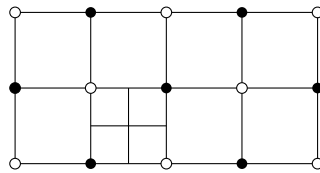
3.9 QrefineRB

Before proceeding with the red2 refinement introduced in the preliminaries, we want to address the question whether it is possible to complete the refinement process to remove hanging nodes with quadrilaterals only. We have already seen that green2 refinements only consist of quadrilaterals but are just available for red2 refinements. However, finding a matching pattern with quadrilaterals only is not trivial for the red refinement.

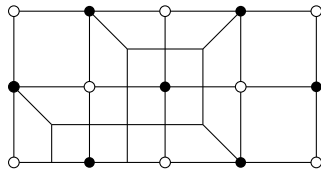
To this end, we want to make use of a pattern that is created by bisecting two adjacent edges and connecting those nodes with the midpoint of the quadrilateral. This midpoint is also connected with the common vertex of the other two edges of the quadrilateral and thus resembles a Y-formation; see Figure 3.38.

We allow the introduced Y-formation and the red rule to refine elements. However, without further assumptions this does not lead to a unique refinement. To see this let us paint the nodes of an exemplary mesh in black and white. Originating from a red refinement, we can now place the small quadrilaterals around black nodes or around white nodes or we can mix both; see Figure 3.39. The questions that arise are whether the non-uniqueness of the refinement strategy can somehow be fixed and if so, does this method guarantee the closure of the mesh? Due to the 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. But if we utilize this pattern to surround a node until another

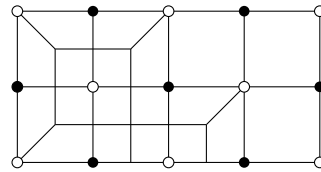
originating from a red refinement



i) we can place the small quadrilaterals around black nodes:



ii) we can place the small quadrilaterals around white nodes:



iii) or we can mix both:

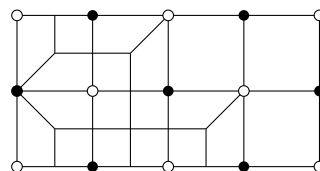


Figure 3.39: Non-uniqueness of refinement patterns in QrefineRB. Originating from a red refinement there are the possibilities of placing the small quadrilaterals around black nodes, around white nodes or a mixture of both. Thus, without further assumptions this refinement pattern does not lead to a unique refinement.

hanging node is reached, hanging nodes will be eliminated. Kobbelt showed in [24] that using this refinement pattern eliminates all hanging nodes.

Theorem 3.9.1 (Kobbelt, [24, Section 6]) *On closed nets (every edge is part of exactly two elements) we have to require the number of edges with hanging nodes to be even. On open nets (boundary edges occur which belong to one element only), any boundary edge can stop a chain of blue elements.*

The proof is based on the following considerations. A red refinement produces four hanging nodes. Every succeeding red refinement changes the number of hanging nodes by an even number $n \in \mathbb{Z}$, $-4 \leq n \leq 4$. Consider the case where the amount of hanging nodes is even and the elements form a ring with an odd number of hanging nodes 'inside' and 'outside'. This cannot happen because every edge with hanging node is shared by two such quadrilaterals and reduces the number by two. Thus, the number of boundary edges of this inner region is again even. Here, we assume that the refinement level of a neighboring element differs at most by one. This does not cause any issues since the following algorithm ensures this property. On open nets, the statement is clear. Hence, this method can be used to eliminate hanging nodes.

To realize this method, a convention of uniquely placing the Y-formation is still needed. An easy way to do this is by painting the nodes alternately in black and white and determine to place the small quadrilateral around black nodes. However, in practice coloring the nodes alternately can be of great effort and supplementary there exist meshes for which an alternating coloring cannot be achieved; see Figure 3.40. Thus, we follow Kobbelt's proposal to first refine the mesh uniformly. The nodes can then be assigned to two generations of meshes as shown in Figure 3.41. Consequently, we determine the reference node $\text{ref}_z(T)$ for all $T \in \mathcal{T}_0$ as the oldest node after one uniform refinement. Note, that in the alternate coloring the oldest node corresponds to the white nodes. Hence, the allowed patterns for QrefineRB are then as depicted in Figure 3.42. For now, we differentiate between the left and right blue element. However, for the implementation we do not need to distinguish between those two elements because of the following reasons. Consider the mesh where nodes are painted alternately in two colors. The goal of this refinement strategy is to place the small quadrilaterals around black nodes. Thus, we just need to find the diagonal of two white nodes to determine the location of the Y-formation.

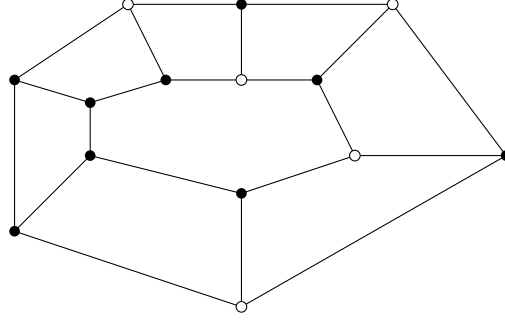


Figure 3.40: A counterexample for coloring the nodes alternately. In the shown mesh, the nodes cannot be painted alternately in two different colors due to the odd number of nodes forming a ring.

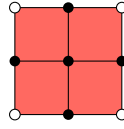


Figure 3.41: An element that is uniformly refined has nodes from two generations. The oldest nodes are depicted as white bullets. The new nodes are shown in black.

However, this can uniquely be determined by setting one of the white nodes as the reference node $\text{ref}_z(T)$. An easy way to ascertain this is by taking the oldest node as the reference node. However, the choice of the reference node can be interchanged along the diagonal without changing the method.

We want to define this refinement rule as a blue refinement for quadrilaterals.

Definition 3.9.2 (Blue refinement of a quadrilateral) *Given a quadrilateral with a reference node and two adjacent marked edges that are either both on the left hand side or on the right hand side of the diagonal starting in the reference node to the opposite lying node. A quadrilateral is blue refined by connecting the midpoint of the quadrilateral with the midpoints of those two adjacent edges and with the common vertex of the other two edges.*

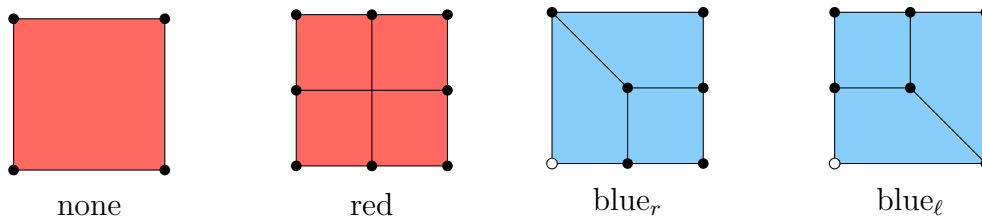


Figure 3.42: Refinement patterns for red elements in QrefineRB. From left to right: A red element and all possible refinement patterns. Initially, we distinguish a blue_r and a blue_ℓ element but in the further course they are treated identically.

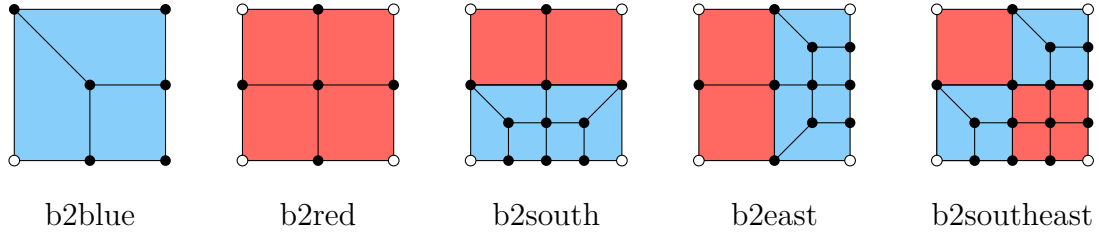


Figure 3.43: Refinement patterns for blue elements in QrefineRB. From left to right: A blue element and all possible refinement patterns. The labeling of each refinement pattern is used in the implementation and therefore given for an easier understanding.

Algorithm 3.7: QrefineRB

INPUT: A triangulation $(\mathcal{T}_\ell^{\text{red}}, (\text{ref}_z(T))_{T \in \mathcal{T}_\ell^{\text{red}}}, \mathcal{T}_\ell^{\text{blue}}, (K_T)_{T \in \mathcal{T}_\ell^{\text{blue}}})$ and a set of marked elements $\mathcal{M}_\ell \subseteq \mathcal{T}_\ell = \mathcal{T}_\ell^{\text{red}} \cup \mathcal{T}_\ell^{\text{blue}}$. The triangulation consists of red elements with reference node $\text{ref}_z(T)$ for all $T \in \mathcal{T}_\ell^{\text{red}}$ and blue elements with specification of the father element K_T for all $T \in \mathcal{T}_\ell^{\text{blue}}$.

Set $\mathcal{E}_{\mathcal{M}_\ell}^{(0)} = \emptyset$, $k := 1$ and define the set of marked edges

$$\mathcal{E}_{\mathcal{M}_\ell}^{(k)} := \left\{ \mathcal{E}(T) : T \in \mathcal{M}_\ell \cap \mathcal{T}_\ell^{\text{red}} \right\} \cup \left\{ \mathcal{E}(K) : K \text{ is father element of } T, T \in \mathcal{M}_\ell \cap \mathcal{T}_\ell^{\text{blue}} \right\}.$$

While $\mathcal{E}_{\mathcal{M}_\ell}^{(k)} \neq \mathcal{E}_{\mathcal{M}_\ell}^{(k-1)}$:

- set $k = k + 1$ and $\mathcal{E}_{\mathcal{M}_\ell}^{(k)} = \mathcal{E}_{\mathcal{M}_\ell}^{(k-1)}$
- $\forall T \in \mathcal{T}_\ell^{\text{red}}$: find pattern in Figure 3.42 that introduces the least new marked edges and extend $\mathcal{E}_{\mathcal{M}_\ell}^{(k)}$ by the non-marked edges of T that need to be further refined due to the pattern
- $\forall T \in \mathcal{T}_\ell^{\text{blue}}$: find pattern for the father element K_T in Figure 3.43 that introduces the least new marked edges and extend $\mathcal{E}_{\mathcal{M}_\ell}^{(k)}$ by the non-marked edges of K that need to be further refined due to the pattern

Define new triangulation $\mathcal{T}_{\ell+1}$ by matching patterns from Figure 3.42 for all $T \in \mathcal{T}_{\ell}^{\text{red}}$ and from Figure 3.43 for all K_T with $T \in \mathcal{T}_{\ell}^{\text{blue}}$. Paint elements in the corresponding color and specify the father elements K_T for all $T \in \mathcal{T}_{\ell+1}^{\text{blue}}$ and new reference nodes $\text{ref}_z(T)$ as the oldest node $\forall T \in \mathcal{T}_{\ell+1}^{\text{red}}$.

OUTPUT: Triangulation $(\mathcal{T}_{\ell}^{\text{red}}, (\text{ref}_z(T))_{T \in \mathcal{T}_{\ell}^{\text{red}}}, \mathcal{T}_{\ell}^{\text{blue}}, (K_T)_{T \in \mathcal{T}_{\ell}^{\text{blue}}})$ into red and blue elements with reference nodes $\text{ref}_z(T)$ for all $T \in \mathcal{T}_{\ell+1}^{\text{red}}$ and father elements K_T for all $T \in \mathcal{T}_{\ell+1}^{\text{blue}}$.

QrefineRB is presented in Algorithm 3.7.

3.9.1 Description of QrefineRB

Let us now give a more explanatory description of QrefineRB. To avoid degeneracies of quadrilaterals, we always remove blue refinements before bisecting further, thus we need to distinguish between red and blue elements.

A red element can be affected by the following markings:

- Case 1) no edge is marked,
- Case 2) one or two edges are marked on the right hand side of the diagonal starting in the reference node,
- Case 3) one or two edges are marked on the right hand side of the diagonal starting in the reference node,
- Case 4) two opposite edges are marked,
- Case 5) three or more edges are marked through neighboring elements or the element itself is marked.

In Case 1, the element remains unchanged. In Case 2, the element is blue_r and in Case 3 it is blue_l refined. If two opposite edges are marked as in Case 3, the element is red refined because there does not exist appropriate blue refinements. Analogously to the other refinement strategies, if three or more edges are marked or the element itself is marked, the quadrilateral is red refined. A depiction of these cases is given in Figure 3.44.

Consider a blue_r element. Note that a blue_l element is analog since the reference

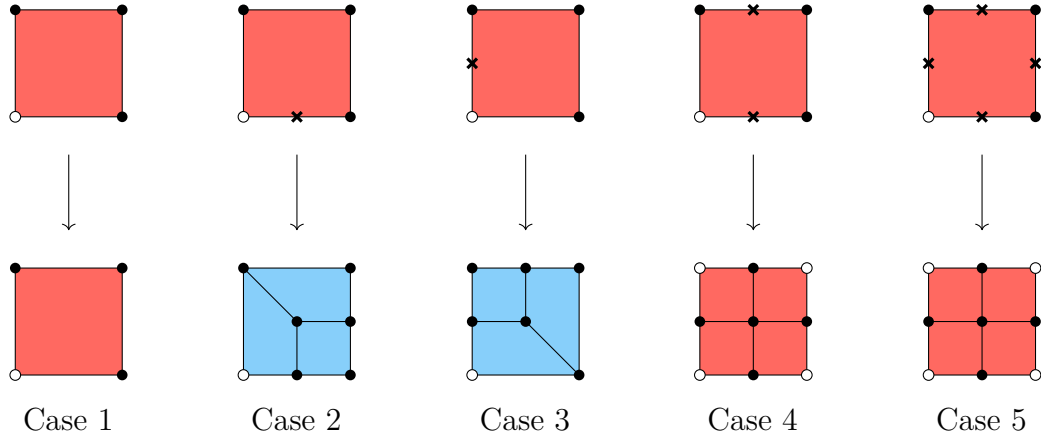


Figure 3.44: Exemplary illustration of QrefineRB: refinement of red elements. In Case 1, the element remains unaffected. In Case 2, a blue_r and in Case 3 a blue_l refinement is done. In Case 4 - 5, the element is red refined. Note, that the reference node $\text{ref}_z(T)$ of an element T is highlighted with a white bullet.

nodes can be interchanged along the diagonal without varying the strategy. The following cases arise:

- Case 1) no edge is marked,
- Case 2) one or two 'non-bisected' edges are marked,
- Case 3) the bisected edge with the reference node as vertex is further marked,
- Case 4) the bisected edge without reference node as vertex is further marked,
- Case 5) both bisected edges are further marked.

In Case 1, the element stays the same. In Case 2, the blue refinement is undone and the father element is red refined. In Case 3, if one of the bisected edges are further bisected, the blue refinement is coarsened and the element is red refined with subsequent blue refinements to eliminate hanging nodes. The same happens in Case 4, where only the location of new hanging nodes and therefore the position of succeeding blue refinements differs. In Case 5, the blue element is coarsened and the element is red refined with subsequent red and blue refinements to eliminate hanging nodes. For an exemplary illustration we refer to Figure 3.45.

3.9.2 Properties

This QrefineRB strategy provides the following properties.

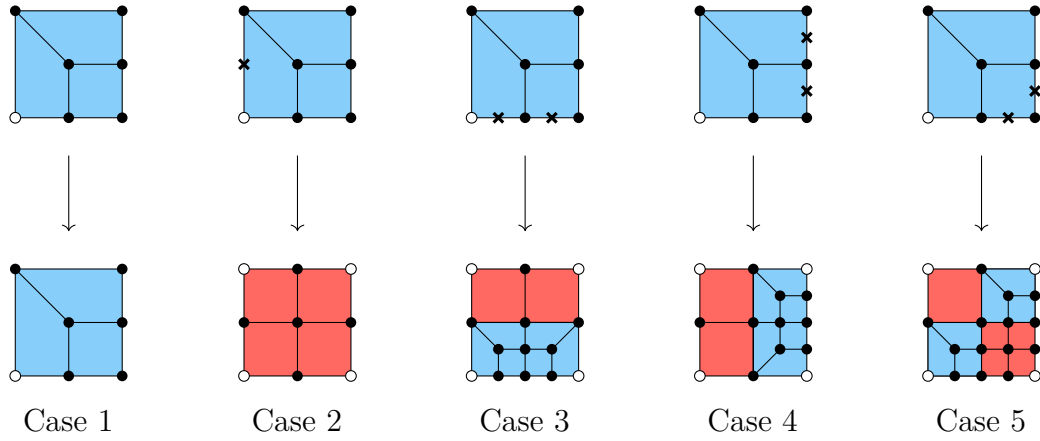


Figure 3.45: Exemplary illustration of QrefineRB: refinement of blue elements. In Case 1, the element stays the same. In Case 2, the blue element is coarsened and then red refined. In Case 3 - 4, the blue element is coarsened and red refined with subsequent blue refinements. In Case 4, the element is coarsened and red refined. Then further blue and red refinements are fitted. Reference nodes are depicted as white bullets.

Remark 3.9.3 *The meshes generated by QrefineRB are non-nested, regular and hold the shape regularity.*

Due to the coarsening of a blue refinement, the meshes are non-nested. Hanging nodes are eliminated by the blue refinement as shown by Kobbelt in [24] and discussed in the above section. The shape regularity of the red refinement is already proven. Since the blue refinement is coarsened before bisected further, the angles of the blue element depend on the shape regularity of red elements only. Therefore, the idea of Zhao, Mao, and Shi in [35] carries over for blue refinements and is presented in detail in the work of Beuter [12].

3.10 QrefineR2

As we have already introduced in the preliminaries for quadrilaterals, there is also the possibility to refine an element by enneasection. Firstly, we introduce a version where we only allow a red2 refinement as defined above. Thus, the allowed patterns are shown in 3.46. We deploy the 2-Irregular Rule which says: *Refine any unrefined element that has more than two hanging nodes on an edge.* To this end, the method is basically described by the following components: the allowed refinement patterns shown in Figure 3.46, the 2-Irregular and 3-Neighbor Rule.

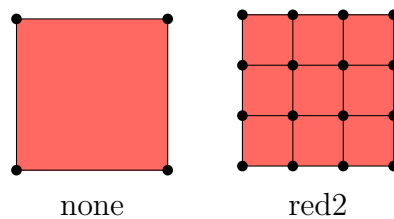


Figure 3.46: Refinement patterns for QrefineR2.

3.10.1 Description of QrefineR2

An exemplary illustration of QrefineR2 is presented in this section. As before, we distinguish between red2 elements and red2 elements with hanging nodes colored in gray. For red elements the upcoming possibilities are conceivable:

- Case 1) no edge is marked,
- Case 2) one or two edges is marked,
- Case 3) three or more edges are marked or the element itself is marked.

In Case 1, the element remains untouched. In Case 2, two hanging nodes per marked edge are introduced and thus the element is painted in gray. In Case 3, a red2 refinement is done if more than three edges are marked through neighboring elements or the element itself is marked. A depiction is given in Figure 3.47.

For gray elements, i.e. elements with hanging nodes, the following cases arise:

- Case 1) no further edge is marked,
- Case 2) one further regular edge is marked,
- Case 3) two or more further regular edges are marked or the element itself is marked,
- Case 4) at least one irregular edge is further marked.

In Case 1, the element stays the same. In Case 2, further two hanging nodes are introduced on the marked edge. If in total at least three edges have hanging nodes, the element is red2 refined (Case 3). In Case 4, the element is red2 refined before

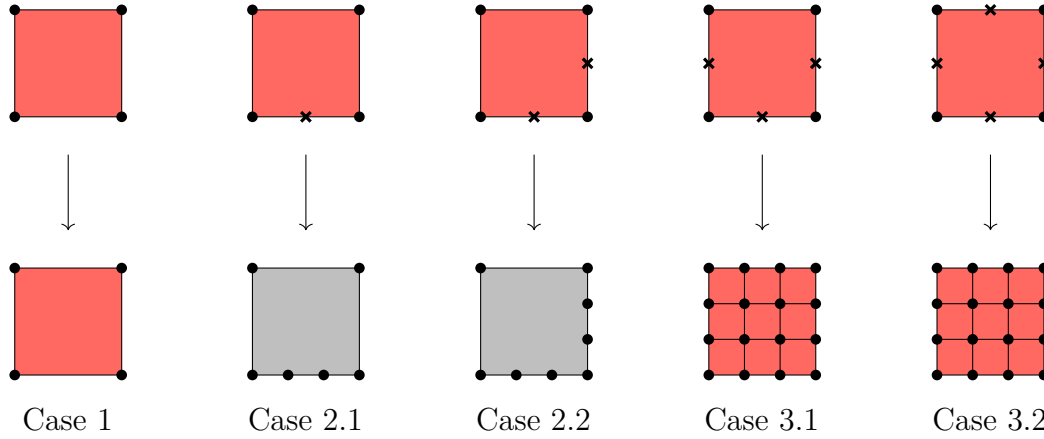


Figure 3.47: Exemplary illustration of QrefineR2: refinement of red elements. In Case 1, the quadrilateral remains unaffected. In Case 2, two hanging nodes per edge are introduced. In Case 3, the element is red2 refined.

in a further step new hanging nodes are introduced; refer to Figure 3.48 for an illustration.

3.10.2 Properties

Remark 3.10.1 *The meshes generated by QrefineR2 are nested, 2-irregular and hold the shape regularity.*

In each refinement step only new nodes are introduced but never removed, thus the mesh is nested. The 2-irregularity of the grid is obvious by deploying the 2-Irregular Rule. The shape regularity is proven by Zhao, Mao, and Shi in [35].

3.11 QrefineRG2

The 2-irregularity of QrefineR2 can be circumvented by using green2 refinements. Henceforth, for QrefineRG2, we allow the patterns illustrated in Figure 3.49. The method is presented in Algorithm 3.8.

Algorithm 3.8: QrefineRG2

INPUT: A triangulation $(\mathcal{T}_\ell^{\text{red}}, \mathcal{T}_\ell^{\text{green2}}, (K_T)_{T \in \mathcal{T}_\ell^{\text{green2}}})$ and a set of marked elements $\mathcal{M}_\ell \subseteq \mathcal{T}_\ell = \mathcal{T}_\ell^{\text{red}} \cup \mathcal{T}_\ell^{\text{green2}}$ with $\mathcal{T}_\ell^{\text{green2}} = \mathcal{T}_\ell^{\text{green2 t1}} \cup \mathcal{T}_\ell^{\text{green2 t2}} \cup \mathcal{T}_\ell^{\text{green2 t3}}$. The triangulation consists of red and green elements and for each green element the father element K_T is specified. There exists three different

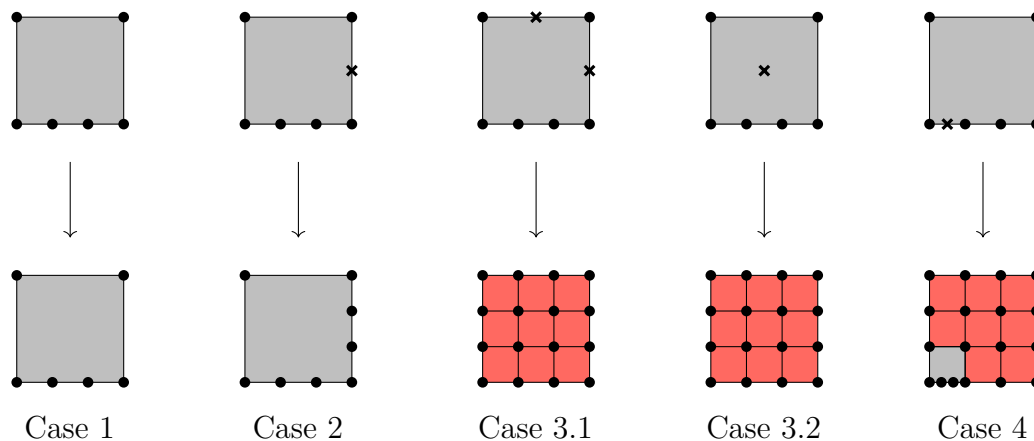


Figure 3.48: Exemplary illustration of QrefineR2: refinement of irregular elements. In Case 1, the element remains unaffected. In Case 2, further hanging nodes are introduced. In Case 3, the element is red2 refined if a refinement would lead to at least three irregular edges. In Case 4, an irregular edge is marked and thus the element is first red refined before new hanging nodes are introduced.

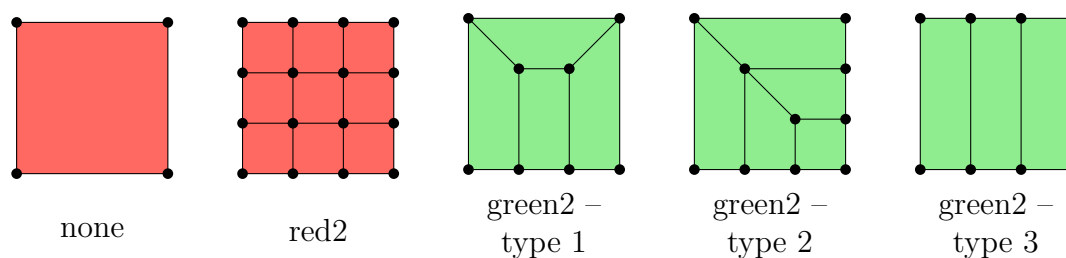


Figure 3.49: Refinement patterns for red elements in QrefineRG2. From left to right: a red element and all its possible refinement patterns.

types of green refinements.

Set $\mathcal{E}_{\mathcal{M}_\ell}^{(0)} = \emptyset$, $k := 1$ and define the set of marked edges

$$\mathcal{E}_{\mathcal{M}_\ell}^{(k)} := \left\{ \mathcal{E}(T) : T \in \mathcal{M}_\ell \cap \mathcal{T}_\ell^{\text{red}} \right\} \cup \left\{ \mathcal{E}(K) : K \text{ is father element of } T, T \in \mathcal{M}_\ell \cap \mathcal{T}_\ell^{\text{green2}} \right\}.$$

While $\mathcal{E}_{\mathcal{M}_\ell}^{(k)} \neq \mathcal{E}_{\mathcal{M}_\ell}^{(k-1)}$:

- set $k = k + 1$ and $\mathcal{E}_{\mathcal{M}_\ell}^{(k)} = \mathcal{E}_{\mathcal{M}_\ell}^{(k-1)}$
- $\forall T \in \mathcal{T}_\ell^{\text{red}}$: find pattern in Figure 3.49 that introduces the least new marked edges and extend $\mathcal{E}_{\mathcal{M}_\ell}^{(k)}$ by the non-marked edges of T that need to be further refined due to the pattern
- $\forall T \in \mathcal{T}_\ell^{\text{green2 t1}}$: find pattern for the father element K_T analog to Figure 3.35 that introduces the least new marked edges and extend $\mathcal{E}_{\mathcal{M}_\ell}^{(k)}$ by the non-marked edges of K that need to be further refined due to the pattern
- $\forall T \in \mathcal{T}_\ell^{\text{green2 t2}}$: find pattern for the father element K_T analog to Figure 3.36 that introduces the least new marked edges and extend $\mathcal{E}_{\mathcal{M}_\ell}^{(k)}$ by the non-marked edges of K that need to be further refined due to the pattern
- $\forall T \in \mathcal{T}_\ell^{\text{green2 t3}}$: find pattern for the father element K_T analog to Figure 3.37 that introduces the least new marked edges and extend $\mathcal{E}_{\mathcal{M}_\ell}^{(k)}$ by the non-marked edges of K that need to be further refined due to the pattern

Define new triangulation $\mathcal{T}_{\ell+1}$ by matching patterns from Figure 3.49 for all $T \in \mathcal{T}_\ell^{\text{red}}$, from an analog to Figure 3.35 for all K_T with $T \in \mathcal{T}_\ell^{\text{green2 t1}}$, from an analog to Figure 3.36 for all K_T with $T \in \mathcal{T}_\ell^{\text{green2 t2}}$ and from an analog to Figure 3.37 for all K_T with $T \in \mathcal{T}_\ell^{\text{green2 t3}}$. Partition elements in the corresponding types and specify the father elements for all $T \in \mathcal{T}_{\ell+1}^{\text{green2}}$.

OUTPUT: Triangulation into red and green elements $(\mathcal{T}_{\ell+1}^{\text{red}}, \mathcal{T}_{\ell+1}^{\text{green2}} = \mathcal{T}_{\ell+1}^{\text{green2 t1}} \cup \mathcal{T}_{\ell+1}^{\text{green2 t2}} \cup \mathcal{T}_{\ell+1}^{\text{green2 t3}}, (K_T)_{\mathcal{T}_{\ell+1}^{\text{green2}}})$ with father elements K_T for all $T \in \mathcal{T}_{\ell+1}^{\text{green2}}$.

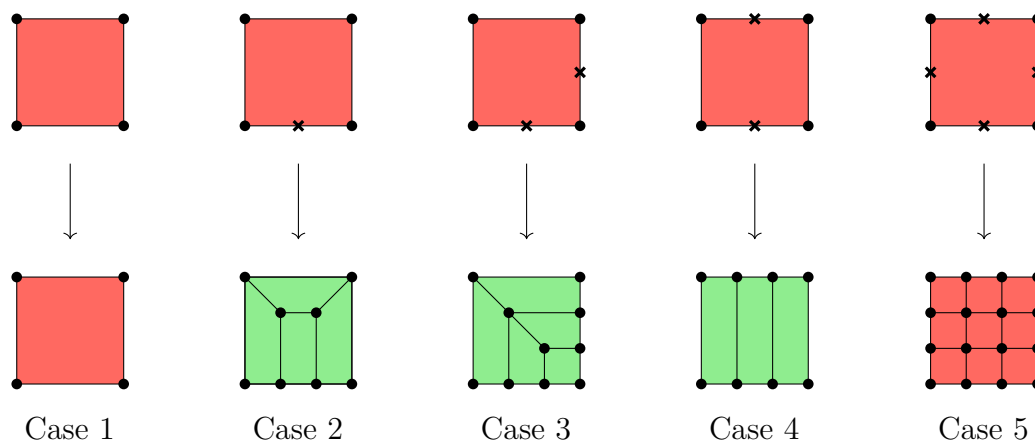


Figure 3.50: Exemplary illustration of QrefineRG2: refinement of red elements. In Case 1, the element remains unaffected. In Case 2 – 4, hanging nodes are eliminated by matching the appropriate green2 refinement pattern. In Case 5, if at least three edges are marked, the element is red2 refined.

3.11.1 Description of QrefineRG2

This method is very similar to QrefineRGtri, where triangles are used for the closure of the mesh. We again, need to distinguish between red2 and green2 elements. The following cases arise in the case of red2 elements:

- Case 1) no edge is marked,
- Case 2) one edge is marked,
- Case 3) only two adjacent edges are marked,
- Case 4) only two opposite edges are marked,
- Case 5) three or more edges are marked through neighboring elements or the element itself is marked.

In Case 1, the element is untouched. In Case 2 – 4 an appropriate green2 pattern is fitted. In Case 5, the element is red refined. An exemplary illustration of these cases is given in Figure 3.50.

A distinction of cases for each green2 element is very extensive. We therefore refer to QrefineRGtri, where we listed all possible refinement patterns. The same patterns carry over from QrefineRGtri by using the green2 refinements instead of the green refinements.

3.11.2 Properties

Remark 3.11.1 *The meshes generated by QrefineRG2 are non-nested, regular and hold the shape regularity.*

Mao, Zhao, and Shi showed the shape regularity in [35]. The remaining statements are trivial.

4 Implementation

We present an implementation of the refinement strategies introduced in the previous chapter. We pursue implementation approaches that will not unnecessarily complicate the code in terms of readability or distinction of cases. Therefore, a categorization in three different groups is useful. For implementing the mesh refinement strategies, we need to consider how many cases can occur and how to gather those cases and summarize them in a useful matter. Table 4.1 shows a classification into three reasonable implementation approaches. These will be described in the further course of this work.

hash maps	virtual elements	3 steps
TrefineNVB TrefineRGB TrefineRG QrefineRB	TrefineR QrefineR QrefineR2	QrefineRG2 QrefineRGtri

Table 4.1: Classification of refinement strategies into reasonable implementation approaches.

Before we go into detail, we first take a look at the underlying data structures in all implementations.

4.1 Data Structures

In Chapter 2, we have already introduced the terms triangulation $\mathcal{T} = \{T_1, \dots, T_M\}$, set of nodes $\mathcal{N} = \{z_1, \dots, z_N\}$, and boundaries Γ_D and Γ_N . To represent the data structure of a mesh, we follow [22] and specify those terms in the following way:

The set of nodes \mathcal{N} is specified by the $N \times 2$ array `coordinates`. In the ℓ -th row of `coordinates` the x – and y – coordinates of the ℓ -th node $z_\ell = (x_\ell, y_\ell) \in \mathbb{R}^2$ are stored as

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

The triangulation \mathcal{T} is described through a $M \times 3$ integer array `elements3` for triangles and through a $M \times 4$ integer array `elements4` for quadrilaterals. 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

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

Analogously, for the ℓ -th quadrilateral $T_\ell = \text{conv} \{z_i, z_j, z_k, z_l\} \in \mathcal{T}$ with vertices $z_i, z_j, z_k, z_l \in \mathcal{N}$ holds

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

The nodes are ordered in a mathematical positive sense. The boundary data is represented by the boundary edges that are determined by their corresponding nodes, i.e. the ℓ -th edge $E_\ell = \text{conv} \{z_i, z_j\}$ is stored as

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

depending on the boundary conditions. Thus, the Dirichlet boundary is stored within an $K \times 2$ array `dirichlet`, where K is the number of Dirichlet boundary edges. Analogously, the Neumann data is stored within an $L \times 2$ array `neumann` with the number of Neumann boundary edges L . Also here, the boundary edges are given in a mathematical positive sense to ensure that

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

is the outer normal vector of Ω on E_ℓ , where $z_k = (x_k, y_k) \in \mathbb{R}^2$. With these notations, we can represent a triangulation into triangles as shown in Figure 4.1 and similarly, a triangulation into quadrilaterals as depicted in Figure 4.2.

We investigate the function `provideGeometricData` in Listing 4.1. This function supplies further geometric data that is needed besides the problem describing variables `coordinates`, `elements`, `dirichlet` and `neumann` and thus is used in all implementations. Since the element marking is edge-based, we wish to have further data from which we can read out the important information instead of searching costly. Hence, we enumerate the edges of a triangulation and generate an array `edge2nodes` which provides the same information as $\mathcal{N}(E)$ for all $E \in \mathcal{E}$, i.e. the corresponding nodes of all edges. Furthermore, we give information about edges that belong to an element, more precisely $\mathcal{E}(T)$ for all $T \in \mathcal{T}$,

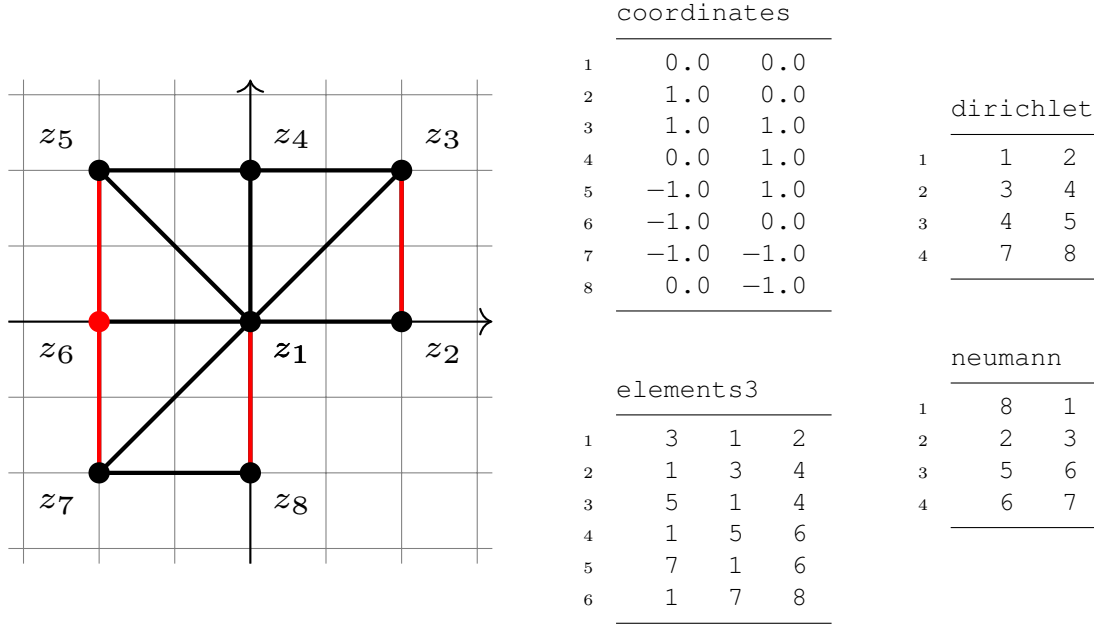


Figure 4.1: Triangulation \mathcal{T} of the L -shaped domain $\Omega = (-1, 1)^2 \setminus ([0, 1] \times [-1, 0])$ into 6 triangles characterized by the arrays `coordinates` and `elements3`. The boundary edges of the L -shape are partitioned into Dirichlet and Neumann boundary. Here, the array `dirichlet` consists of the 4 boundary edges plotted in black. The array `neumann` consists of the remaining 5 boundary edges, indicated in red. The nodes $\mathcal{N} \cap \Gamma_D = \{z_1, z_2, z_3, z_4, z_5, z_7, z_8\}$ are indicated by black bullets, whereas free nodes $\mathcal{N} \setminus \Gamma_D = \{z_6\}$ are depicted as red bullets.

in the variables `element3edges` for triangular elements and `element4edges` for quadrilateral elements. We also provide optional information about the edges at the Dirichlet or Neumann boundary through `varargin` and `varargout`, i.e. $\mathcal{N}(E)$ for all $E \in \{E \in \mathcal{E} : E \text{ is a boundary edge with Dirichlet data}\}$ and $\mathcal{N}(E)$ for all $E \in \{E \in \mathcal{E} : E \text{ is a boundary edge with Neumann data}\}$. A detailed description of Listing 4.1 is listed below.

- Lines 1–2: The function is usually called by `[edge2nodes, element3edges, element4edges, dirichlet2edges, neumann2edges] = provideGeometricData(elements3, elements4, dirichlet, neumann)`, where the boundary conditions `dirichlet` and `neumann` are hidden in the optional arguments `varargin` and `varargout`. If the mesh only consists of triangular elements, we hand over `elements4 = zeros(0, 4)`, similarly if the mesh only consists of quadrilateral elements, we hand over `elements3 = zeros(0, 3)`.
- Lines 5–12: For a triangle with vertices z_1, z_2 and z_3 , an edge is described with

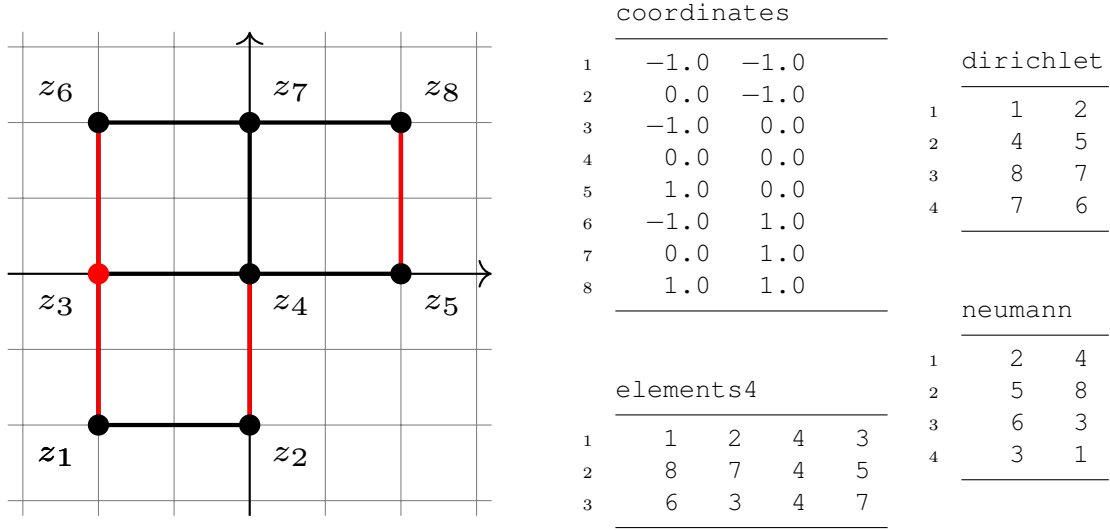


Figure 4.2: Triangulation \mathcal{T} of the L -shaped domain $\Omega = (-1, 1)^2 \setminus ([0, 1] \times [-1, 0])$ into 3 quadrilaterals characterized by the arrays `coordinates` and `elements4`. The boundary edges of the L -shape are partitioned into Dirichlet and Neumann boundary. Here, the array `dirichlet` consists of the 4 boundary edges plotted in black. The array `neumann` consists of the remaining 5 boundary edges, indicated in red. The nodes $\mathcal{N} \cap \Gamma_D = \{z_1, z_2, z_4, z_5, z_6, z_7, z_8\}$ are indicated by black bullets, whereas free nodes $\mathcal{N} \setminus \Gamma_D = \{z_3\}$ are depicted as red bullets.

$E = \text{conv} \{z_i, z_j\}$ where $i, j \in \{1, 2, 3\}, i \neq j$, analogously for quadrilaterals. A collection of all edges is then stored in `edges`. Additionally, we also incorporate the boundary edges, if given. The pointer `ptr` counts the number of all existing edges for triangles, quadrilaterals and boundary edges. Duplications due to neighboring elements are included.

- Line 14: The edges are sorted such that the smallest number of a node is in column one. This is beneficial to find out which edges are listed twice and by the `unique` command it is ensured that each edge is stored once, i.e. no duplications are allowed. An edge E_ℓ is then the output `edge2nodes(l, :)` which provides the numbers i, j of the nodes z_i, z_j such that $E_\ell = \text{conv} \{z_i, z_j\}$.
- Lines 15–16: `element3edges(i, l)` provides the number of the edge between the nodes `elements(i, l)` and `elements(i, l+1)` where we identify $\ell + 1 = 4$ with $\ell = 1$. Similarly, `element4edges(i, l)` returns the number of the edge between the nodes as above where we identify $\ell + 1 = 5$ with $\ell = 1$.
- Line 18–20: If boundary conditions are handed over, the number of edges that correspond to the Dirichlet or Neumann data is also provided.

Listing 4.1: provideGeometricData

```

1 function [edge2nodes,element3edges,element4edges,varargout] ...
2     = provideGeometricData(elements3,elements4,varargin)
3 %*** Obtain geometric information on edges
4 %*** Collect all edges
5 edges = [reshape(elements3(:, [1:3,2:3,1]), [], 2); ...
6     reshape(elements4(:, [1:4,2:4,1]), [], 2)] ;
7 ptr = [3*size(elements3,1),4*size(elements4,1),zeros(1,nargin-2)];
8 for j = 1:nargin-2
9     ptr(j+2) = size(varargin{j},1);
10    edges = [edges;varargin{j}];
11 end
12 ptr = cumsum(ptr);
13 %*** Create numbering of edges
14 [edge2nodes,~,ie] = unique(sort(edges,2), 'rows');
15 element3edges = reshape(ie(1:ptr(1)), [], 3);
16 element4edges = reshape(ie(ptr(1)+1:ptr(2)), [], 4);
17 %*** Provide boundary2edges
18 for j = 1:nargin-2
19     varargout{j} = ie(ptr(j+1)+1:ptr(j+2));
20 end

```

4.2 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 marked edge has value one and a non-marked edge has value zero. A triangle has three edges and thus the number of different marking possibilities is $2^3 = 8$. These are depicted in Table 4.2. For each combination, a binary and its corresponding decimal number are assigned. Note, that we use the binary number in a reverse order. The same applies for quadrilaterals. However, a quadrilateral has four edges and thus the number of possibilities is $2^4 = 16$.

Following the descriptions for each refinement strategy, we translate the allowed refinement patterns for each strategy into binary numbers and store those in hash. The function `hash2map` determines which further edges need to be marked to get one of the allowed patterns. Here, only further edges can be marked but a marking can not be undone. In Table 4.3, we illustrate the approach for TrefineNVB. As visible, we only have five different refinement patterns but eight combinations of markings.

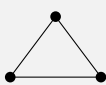
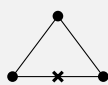
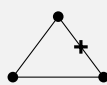
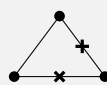
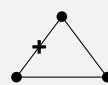
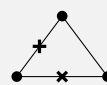
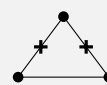
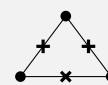
								
bin	000	100	010	110	001	101	011	111
dec	0	1	2	3	4	5	6	7

Table 4.2: Marking a triangle. There are $2^3 = 8$ possibilities to mark the edges of a triangle. Thus, for the implementation we assign a binary and decimal value for each possible marking. Note, that the binary value is used in a reverse order. The first edge is considered to be the bottom edge and the other edges are counted in a counterclockwise order.

Thus, the pattern is chosen such that the least amount of new marked edges is introduced. If we establish this data for all combinations, the information can easily be read out during the iterative process of refinement. The implementation is shown in Listing 4.2 and described in the following.

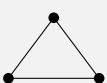
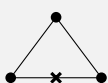
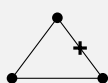
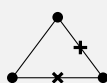
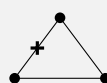

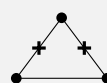
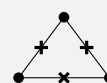
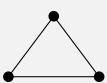
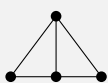
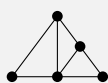
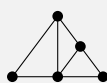
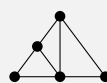
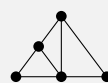
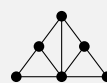
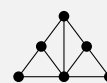
mark								
bin	000	100	010	110	001	101	011	111
dec	0	1	2	3	4	5	6	7
	↓	↓	↓	↓	↓	↓	↓	↓
hash								
bin	000	100	110	110	101	101	111	111
val	0	1	2	2	3	3	4	4
type	none	green	blue _r	blue _r	blue _ℓ	blue _ℓ	red _{NVB}	red _{NVB}

Table 4.3: Mapping of eight possible markings to the five patterns allowed in TrefineNVB. For each hash a binary number is given and a type name plus value is assigned.

Listing 4.2: hash2map

```

1 function [map, val] = hash2map(dec, hash)
2 n = size(hash, 2);
3 bin = rem(floor(dec * pow2(1 - n : 0)), 2);
4 bin = fliplr(bin);
5 map = zeros(size(bin));

```

```

6 val = zeros(1,size(bin,1));
7 [idx,jdx] = find(bin);
8 for i = 1:size(bin,1)
9     if dec(i)
10         kdx = idx==i;
11         % already marked edges stay marked edges
12         [pos,~] = find(hash(:,jdx(kdx))));
13         % find corresponding hash
14         [~,mdx] = min(sum(abs(hash(pos,:))...
15                             -bin(i*ones(length(pos),1),:)),2));
16         map(i,:) = hash(pos(mdx),:);
17         val(i) = pos(mdx);
18     end
19 end
20 end

```

- Lines 3–4: For a given decimal `dec`, the associated binary value `bin` is computed by finding the binary digits b_i with the formula $b_i = \lfloor \text{dec} \cdot 2^{i-1} \rfloor \bmod 2$ for all $i \in \{1, \dots, \# \text{of digits}\}$ where $\lfloor \cdot \rfloor$ is the floor function which outputs the greatest integer that is less than or equal to the input. Since we use the binary representation in a reverse order, the binary number is flipped from left to right with the MATLAB function `fliplr`.
- Lines 7–12: In Line 7, indices are found that correspond to a value of 1 in the variable `bin`. With this information we find out, provided that the decimal is non-zero, in which column at least one value is non-zero. A marking is never taken back and through additional markings it is ensured that a mapping to the allowed configurations in the hash is feasible. Therefore, in Line 12, by finding the columns in the hash where at least the already marked edges are marked, the possible configurations for this case are determined.
- Lines 13–17: To decide which entry in the table is the scheme to use, we determine the combination of marked edges which requests for the least amount of further marking. This can be done by comparing the hash entries chosen in Line 12 with the already marked edges. Through taking the absolute value of the difference and subsequently summing up, the index of the minimal value tells the correct mapping. We return `map` as a binary value, telling which

edges have to be marked and `val` which attaches a number to this specific refinement type.

The general procedure for an implementation with hash maps can be described in a few steps. First of all, further geometric data is generated by calling the function `provideGeometricData`. The elements are marked by marking all edges for bisection. Then, the refinement patterns are listed in a hash map and with the function `hash2map`, the variable `map` is created in which it is clearly prescribed what to do in each case. With this help, we determine which further edges need to be marked to maintain one of the patterns and do this until no further edges need to be marked. This is basically the completion process to ensure the regularity of the grid. For each marked edge, the `coordinates`-array is extended by the midpoints of these edges. If boundary data is given, those edges are also bisected. A direct assignment of new nodes to a refined edge is provided in the variable `newNodes`. Next, an element numbering is created which ensures that elements having one father element are consecutively listed. With the help of old `elements` and the created variable `newNodes` it is possible to define the new elements.

In the following, an implementation for each refinement strategy is described in detail. The focus is on special features that only occur in a specific type of refinement.

4.2.1 TrefineNVB

An efficient implementation of TrefineNVB is already given in [22]. We want to maintain the notation and parts of the structure. But instead of finding which further edges need to be marked in each step, we read out the information from the output map of the function in Listing 4.2. The implementation is given in Listing 4.3.

Listing 4.3: TrefineNVB

```

1 function [coordinates,newElements,varargout] ...
2     = TrefineNVB(coordinates,elements,varargin)
3 markedElements = varargin{end};
4 nE = size(elements,1);
5 %*** Obtain geometric information on edges
6 [edge2nodes,element2edges,~,boundary2edges{1:nargin-3}] ...
7     = provideGeometricData(elements,zeros(0,4),varargin{1:end-1});
8 %*** Mark edges for refinement

```

```

9 edge2newNode = zeros(1,max(max(element2edges)));
10 edge2newNode(element2edges(markedElements,:)) = 1;
11 hash = [1,0,0;1,1,0;1,0,1;1,1,1];
12 [map,value] = hash2map((0:7)',hash);
13 %*** Change flags for elements
14 swap = 1;
15 while ~isempty(swap)
16     markedEdge = edge2newNode(element2edges);
17     dec = sum(markedEdge(1:nE,:) .* (ones(nE,1)*2.^(0:2)),2);
18     val = value(dec+1);
19     [idx,jdx] = find(~markedEdge(1:nE,:) & map(dec+1,:));
20     swap = idx + (jdx-1)*nE;
21     edge2newNode(element2edges(swap,1)) = 1;
22 end
23 %*** Generate new nodes
24 edge2newNode(edge2newNode~=0)...
25     = size(coordinates,1) + (1:nnz(edge2newNode));
26 idx = find(edge2newNode);
27 coordinates(edge2newNode(idx,:),:) = (coordinates(edge2nodes(idx,1),:))...
28     +coordinates(edge2nodes(idx,2),:))/2;
29 %*** Refine boundary conditions
30 varargout = cell(nargout-2,1);
31 for j = 1:nargout-2
32     boundary = varargin{j};
33     if ~isempty(boundary)
34         newNodes = edge2newNode(boundary2edges{j})';
35         markedEdges = find(newNodes);
36         if ~isempty(markedEdges)
37             boundary = [boundary(~newNodes,:); ...
38                 boundary(markedEdges,1),newNodes(markedEdges); ...
39                 newNodes(markedEdges),boundary(markedEdges,2)];
40         end
41     end
42     varargout{j} = boundary;
43 end
44 %*** Provide new nodes for refinement of elements
45 newNodes = edge2newNode(element2edges);
46 %*** Determine type of refinement for each element
47 none = find(val==0) ;
48 green = find(val==1);
49 bluer = find(val==2);

```

```

50 blue1 = find(val==3);
51 red = find(val==4);
52 %*** Generate element numbering for refined mesh
53 idx = ones(nE,1);
54 idx(none) = 1;
55 idx(green) = 2;
56 idx([bluer, blue1]) = 3;
57 idx(red) = 4;
58 idx = [1;1+cumsum(idx)];
59 %*** Generate new elements
60 newElements = zeros(idx(end)-1,3);
61 newElements(idx(none),:) = elements(none,:);
62 tmp = [elements, newNodes];
63 newElements([idx(green),1+idx(green)],:) ...
64     = [tmp(green,[3,1,4]); tmp(green,[2,3,4])];
65 newElements([idx(bluer),1+idx(bluer),2+idx(bluer)],:) ...
66     = [tmp(bluer,[3,1,4]);tmp(bluer,[4,2,5]);tmp(bluer,[3,4,5])];
67 newElements([idx(blue1),1+idx(blue1),2+idx(blue1)],:) ...
68     = [tmp(blue1,[4,3,6]);tmp(blue1,[1,4,6]);tmp(blue1,[2,3,4])];
69 newElements([idx(red),1+idx(red),2+idx(red),...
70     3+idx(red)],:) = [tmp(red,[4,3,6]);tmp(red,...
71     [1,4,6]);tmp(red,[4,2,5]); tmp(red,[3,4,5])];

```

- Lines 1–3: The function is usually called by

```
[coordinates,elements3,dirichlet,neumann] =
```

```
TrefineNVB(coordinates,elements3,dirichlet,neumann,marked3)
```

where we output all information about the refined mesh. Optional arguments are `dirichlet`, `neumann` and `marked3`. We anticipate that the last entry in `varargin` corresponds to the marked elements and is always handed over.

- Lines 5–7: Further information about the relation between elements, edges and nodes is determined by the function described in Listing 4.1.
- Lines 8–10: The array `edge2newNode` is created. We want `edge2newnode(ℓ)` to be non-zero if the ℓ -th edge is marked for bisection and to be zero if the ℓ -th edge is not marked. In Line 10, we set all edges of marked elements to one. This makes sense because we mark elements by marking each edge for bisection.
- Lines 11–12: In hash all possible configurations for NVB are stored. If an edge

is marked, we need to ensure that at least the reference edge is also marked. Instead of storing the reference edge in a separate array, the new reference edge is always placed as the first edge of an element. The reference edge is then implicitly given through the numbering of edges in an element. With this convention the patterns 000, 100, 110, 101, and 111 are allowed. With the function `hash2map` all markings are mapped to a pattern in `hash`, as shown in Table 4.3. In the further course, the information can be read out of `map`.

- Lines 13–22: `markedEdge(ℓ , :)` gives binary values for the first, second and third edge of the ℓ -th element which state whether an edge is marked for bisection or not. By comparing those values with the appropriate pattern determined and stored in `map`, missing edges are marked. By adding marked edges, further changes can be triggered and thus, we iterate through this process until no further changes are made, i.e. the variable `swap` is empty.
- Lines 23–28: The number of non-zero entries in `edge2newNode` determines the quantity of new nodes that need to be generated. Henceforth, we expand `coordinates` by the coordinates of the new nodes that are the midpoints of each edge. To compute these values, we need the two nodes of this edge that are stored in `edge2nodes`. The index of the new node of edge number ℓ is saved in `edge2newNode(ℓ)`.
- Lines 30–43: In Line 34–35, indices are found and stored in `markedEdges` that correspond to boundary data that needs to be refined. The new boundary data is defined as the non-refined boundary data plus refined edges due to new nodes on these edges; see Lines 37–39. This part of the code is only reached if we call the function with additional output arguments. If no associated input arguments are provided, the output is an empty array.
- Line 45: `newNodes(i , ℓ)` is non-zero if the ℓ -th edge of element T_i is marked. Then, the entry at this location is the index of the edge's midpoint.
- Lines 46–58: The elements are sorted by their refinement pattern. This can easily be done by using the numeric value `val` that is assigned to each pattern. Thus, the types `none`, `green`, `bluer`, `blue1`, and `red` give the corresponding indices. With this help, a new element numbering can be created by defining how many triangles are formed in each case. `none` defines one tri-

angle, green two triangles, blue, blue1 three and red four triangles. With this information, we generate an element numbering for the refined mesh, ensuring that triangles originating from one father element are listed next to each other.

- Lines 59–71: In this part, the elements are generated. For an easier implementation, all values are stored in a temporary data structure. Here,

$$\text{tmp}(i, :) = [z_1, z_2, z_3, m_{z_1, z_2}, m_{z_2, z_3}, m_{z_3, z_1}]$$

where z_j are the vertices of triangle T_i and m_{z_j, z_k} are the midpoints between the vertices z_j and z_k . If there does not exist a midpoint due to the refinement strategy, the entry at this location is zero. The elements are formed keeping in mind that the first edge of an element is always the new reference edge and thus the edge that is opposite to the newest vertex.

4.2.2 TrefineRGB

It is known that the refinement strategy TrefineRGB only differs from TrefineNVB in the case when all edges are marked for refinement, i.e. instead of three bisections TrefineRGB uses a red refinement; see Chapter 3. Thus, we can modify Listing 4.3 and replace Lines 69–71 by the ones given in Listing 4.4. This part defines the new triangles for the case in which all edges are marked for bisection and uses the red refinement. The first edge of each triangle is the new reference edge as shown in Figure 3.22.

Listing 4.4: TrefineRGB: Lines 69–71

```

69 newElements([idx(red), 1+idx(red), 2+idx(red), ...
70     3+idx(red)], :) = [tmp(red, [1, 4, 6]); tmp(red, ...
71     [4, 2, 5]); tmp(red, [6, 5, 3]); tmp(red, [5, 6, 4])];

```

4.2.3 TrefineRG

We follow the same approach for TrefineRG. The main difference is that we need to distinguish whether an element is a red or a green element. In the case of red

elements, we refine those elements further. In the case of green elements, we first undo this refinement and proceed from this point. We now explain Listing 4.5.

- Lines 1–5: The function is usually called by

```
[coordinates,elements3,dirichlet,neumann] =  
TrefineRG(coordinates,elements3,dirichlet,neumann,marked3).
```

The variable `elements3` consists of red and green elements and they are stored block by block, starting with red elements. Again, the last entry of `varargin` is considered to be the marked elements. Here, a persistent variable `nG` is defined which tells the number of green elements. A persistent variable is local to the function in which it is declared. Between function calls, the value of the variable `nG` is kept in memory, i.e. the number of green elements after each refinement step is updated to access the correct value in the upcoming refinement step (Line 126).

- Lines 6–13: The function `provideGeometricData` is called to generate further geometric data. In the first run, the variable `nG` is not equipped with a value. Thus, in the case that `nG` is empty, it is set to zero. Since we only separate into green and red triangles, the number of red elements `nR` can then be calculated as the difference of the number of all elements `nE` and `nG`.
- Lines 15–19: We differentiate between red elements (Line 16–17) and green elements (Line 18–19). A red element is marked by marking each edge for bisection. Marking a green element entails the retraction of the green refinement and refines the father element red. In this case, we need to find the non-marked edges of this father element and mark them for bisection, i.e. set their values to one. The father element consists of two green elements T_1 with vertices v_i and T_2 with vertices w_i , $i \in \{1, 2, 3\}$, i.e. the corresponding non-marked edges are the first edge $\text{conv}\{v_1, v_2\}$ of element T_1 and the second edge $\text{conv}\{w_2, w_3\}$ of element T_2 ; see Figure 4.3. This is found in MATLAB by `element2edges(nR+[2*marked-1,nE+2*marked])`. The variable `edge2newNode` tells which edges are marked for bisection using logical values.
- Lines 20–23: The possible refinement configurations of red elements are listed in `hashR`, analog for green elements in `hashG`. Red elements can either stay (000), be refined red (111), or green where there exist three different positions of a green refinement. More precisely the triangle can be divided at the first

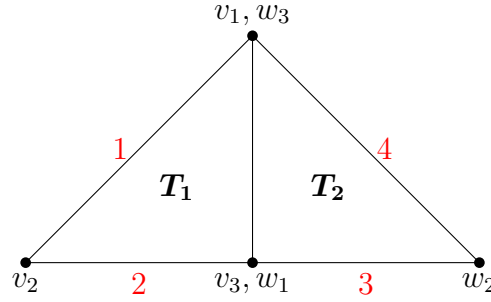
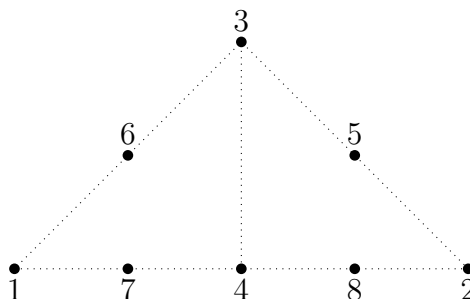


Figure 4.3: Numbering of outer edges for green elements in TrefineRG.

(100), second (010) or third edge (001). A green element is never considered independently. Responsible for further refinement are only the outer edges of the pair of green elements that correspond to one father element. Hence, for each pair of green elements there exist four outer edges and $2^4 = 16$ possible markings; see Figure 4.3. With this enumeration of edges, we only allow the refinement configurations listed in `hashG` and they are deduced from Figure 3.10. With the help of the function `hash2map`, we then describe for each possible combination of marked edges what will be further marked due to the allowed refinement configurations.

- Lines 24–33: In Line 15–19, we already managed to mark edges for bisection due to element markings. There are also options to mark edges through neighboring elements which also needs to be dealt with in the algorithm. Lines 24–33 take care of red elements and ensure the completion of the refinement. For this purpose, the decimal value of marked edges of each element is determined and the unique refinement configuration is read out from `mapR`. A comparison of marked edges and the refinement configuration in Line 31 supplies the indices where a non-marked edge needs to be set to one.
- Lines 34–43: Analogously, the same occurs for green elements. Thus, the outer edges of the father element are declared in `bit` and the corresponding decimal value for the binary value of the four outer edges informs about the correct refinement configuration listed in `mapG`. In Line 39, a comparison of marked edges and the corresponding configuration provides indices for further markings in the completion process. Lines 40–43 mark the outer edges shown in Figure 4.3, if the variable `flags` is non-empty. This completion process for red and green triangles (Lines 25–44) is repeated until no further edge needs to be marked and hence `swap` and `flags` are empty.

Figure 4.4: Numbering of the nodes in variable `tmp` in TrefineRG.

- Lines 45–125: The ideas of the remaining lines are basically the same as Lines 23–71 in Listing 4.3. In Lines 45–50 new nodes are generated and in Lines 51–65 boundary edges are refined. In Lines 68–79 the type of refinement for red and green elements is determined and with this information a new element numbering for the refined mesh is generated (Lines 80–90). Observe, that `newElements` contains a block of red elements with indices `rdx` of Line 86, following another block of green elements with indices `gdx`; see Line 90. For easily accessing nodes to define new elements, we define the variable `tmp` which entries are the nodes at the corresponding location; see Figure 4.4. Not all nodes are defined in each element, thus in those cases the node value is set to zero. New red elements are formed in Lines 91–109 and new green elements are generated in Lines 110–125.

Listing 4.5: TrefineRG

```

1 function [coordinates,newElements,varargout] ...
2     = TrefineRG(coordinates,elements,varargin)
3 persistent nG
4 nE = size(elements,1);
5 markedElements = varargin{end};
6 %*** Obtain geometric information on edges
7 [edge2nodes,element2edges,~,boundary2edges{1:nargin-3}] ...
8     = provideGeometricData(elements,zeros(0,4),varargin{1:end-1});
9 %*** Count number of green sibling elements;
10 if isempty(nG)
11     nG = 0;
12 end
13 nR = nE-nG;
14 %*** Mark edges for refinement

```



```

15 edge2newNode = zeros(1,size(edge2nodes,1));
16 marked = markedElements(markedElements<=nR);
17 edge2newNode(element2edges(marked,:)) = 1;
18 marked = ceil((markedElements(markedElements>nR)-nR)/2);
19 edge2newNode(element2edges(nR+[2*marked-1,nE+2*marked])) = 1;
20 hashR = logical([1,1,1;1,0,0;0,1,0;0,0,1]);
21 [mapR,valueR] = hash2map((0:7)',hashR);
22 hashG = logical([1,0,0,1;1,1,0,1;1,0,1,1;1,1,1,1]);
23 [mapG,valueG] = hash2map((0:15)',hashG);
24 swap = 1;
25 while ~isempty(swap) || any(flags(:))
26     markedEdge = edge2newNode(element2edges);
27     %*** Change flags for red elements
28     bit = markedEdge(1:nR,:);
29     dec = sum(bit.*(ones(nR,1)*2.^(0:2)),2);
30     valR = valueR(dec+1);
31     [idx,jdx] = find(~bit & mapR(dec+1,:));
32     swap = idx + (jdx-1)*nE;
33     edge2newNode(element2edges(swap)) = 1;
34     %*** Change flags for green elements
35     bit = [markedEdge(nR+1:2:end,1:2), markedEdge(nR+2:2:end,1:2)];
36     dec = sum(bit.*(ones(nG/2,1)*2.^(0:3)),2);
37     valG = valueG(dec+1);
38     gdx = find(valG)';
39     flags = ~bit & mapG(dec+1,:);
40     edge2newNode(element2edges(nR+2*gdx(flags(gdx,1))-1,1)) = 1;
41     edge2newNode(element2edges(nR+2*gdx(flags(gdx,2))-1,2)) = 1;
42     edge2newNode(element2edges(nR+2*gdx(flags(gdx,3)),1)) = 1;
43     edge2newNode(element2edges(nR+2*gdx(flags(gdx,4)),2)) = 1;
44 end
45 %*** Generate new nodes
46 edge2newNode(edge2newNode~=0) = size(coordinates,1) ...
47     + (1:nnz(edge2newNode));
48 idx = find(edge2newNode);
49 coordinates(edge2newNode(idx),:) = (coordinates(edge2nodes(idx,1),:))...
50     +coordinates(edge2nodes(idx,2),:))/2;
51 %*** Refine boundary conditions
52 varargout = cell(nargout-2,1);
53 for j = 1:nargout-2
54     boundary = varargin{j};
55     if ~isempty(boundary)

```

```

56     newNodes = edge2newNode(boundary2edges{j})';
57     markedEdges = find(newNodes);
58     if ~isempty(markedEdges)
59         boundary = [boundary(~newNodes,:); ...
60                     boundary(markedEdges,1),newNodes(markedEdges); ...
61                     newNodes(markedEdges),boundary(markedEdges,2)];
62     end
63 end
64 varargout{j} = boundary;
65 end
66 %*** Provide new nodes for refinement of elements
67 newNodes = edge2newNode(element2edges);
68 %*** Determine type of refinement for each red element
69 none = find(valR == 0);
70 r2red = find(valR == 1);
71 r2green1 = find(valR == 2);
72 r2green2 = find(valR == 3);
73 r2green3 = find(valR == 4);
74 %*** Determine type of refinement for each green element
75 g2green = nR + find(valG == 0);
76 g2red = nR + find(valG == 1);
77 g2red1 = nR + find(valG == 2);
78 g2red2 = nR + find(valG == 3);
79 g2red12 = nR + find(valG == 4);
80 %*** Generate element numbering for refined mesh
81 rdx = zeros(nR+nG/2,1);
82 rdx(none) = 1;
83 rdx([r2red,g2red]) = 4;
84 rdx([g2red1,g2red2]) = 3;
85 rdx(g2red12) = 2;
86 rdx = [1;1+cumsum(rdx)];
87 gdx = zeros(size(rdx));
88 gdx([r2green1, r2green2, r2green3, g2green, g2red1, g2red2]) = 2;
89 gdx(g2red12) = 4;
90 gdx = rdx(end)+[0;0+cumsum(gdx)];
91 %*** Generate new red elements
92 newElements = 1+zeros(gdx(end)-1,3);
93 newElements(rdx(none),:) = elements(none,:);
94 tmp = [elements(1:nR,:),newNodes(1:nR,:),zeros(nR,2);...
95        elements(nR+1:2:end,2),elements(nR+2:2:end,[2,3,1])...
96        newNodes(nR+2:2:end,2), newNodes(nR+1:2:end,1:2),...

```

```

97     newNodes (nR+2:2:end,1) ];
98 newElements ([rdx(r2red),1+rdx(r2red),2+rdx(r2red),3+rdx(r2red)], :) ...
99     = [tmp(r2red,[4,5,6]);tmp(r2red,[1,4,6]);tmp(r2red,[2,5,4]);...
100     tmp(r2red,[3,6,5])];
101 newElements ([rdx(g2red),1+rdx(g2red),2+rdx(g2red),3+rdx(g2red)], :) ...
102     = [tmp(g2red,[4,5,6]);tmp(g2red,[1,4,6]);...
103     tmp(g2red,[4,2,5]);tmp(g2red,[3,6,5])];
104 newElements ([rdx(g2red1),1+rdx(g2red1),2+rdx(g2red1)], :) ...
105     = [tmp(g2red1,[4,5,6]);tmp(g2red1,[4,2,5]);tmp(g2red1,[3,6,5])];
106 newElements ([rdx(g2red2),1+rdx(g2red2),2+rdx(g2red2)], :) ...
107     = [tmp(g2red2,[4,5,6]);tmp(g2red2,[1,4,6]);tmp(g2red2,[3,6,5])];
108 newElements ([rdx(g2red12),1+rdx(g2red12)], :) ...
109     = [tmp(g2red12,[4,5,6]);tmp(g2red12,[3,6,5])];
110 %*** New green elements
111 newElements ([gdx(r2green1),1+gdx(r2green1)], :) ...
112     = [tmp(r2green1,[3,1,4]);tmp(r2green1,[4,2,3])];
113 newElements ([gdx(r2green2),1+gdx(r2green2)], :) ...
114     = [tmp(r2green2,[1,2,5]);tmp(r2green2,[5,3,1])];
115 newElements ([gdx(r2green3),1+gdx(r2green3)], :) ...
116     = [tmp(r2green3,[2,3,6]);tmp(r2green3,[6,1,2])];
117 newElements ([gdx(g2green),1+gdx(g2green)], :) ...
118     = [tmp(g2green,[3,1,4]);tmp(g2green,[4,2,3])];];
119 newElements ([gdx(g2red1),1+gdx(g2red1)], :) ...
120     = [tmp(g2red1,[6,1,7]);tmp(g2red1,[7,4,6])];
121 newElements ([gdx(g2red2),1+gdx(g2red2)], :) ...
122     = [tmp(g2red2,[5,4,8]);tmp(g2red2,[8,2,5])];
123 newElements ([gdx(g2red12),1+gdx(g2red12),2+gdx(g2red12),...
124     3+gdx(g2red12)], :) = [tmp(g2red12,[6,1,7]);tmp(g2red12,[7,4,6]);...
125     tmp(g2red12,[5,4,8]);tmp(g2red12,[8,2,5])];
126 nG = size(newElements,1)-rdx(end)+1;

```

4.2.4 QrefineRB

Similarly to TrefineRG, a distinction into the color of elements is needed in QrefineRB. Here, we differentiate between red and blue elements. A specification of reference nodes $\text{ref}_z(T)$ for each quadrilateral T is also needed. For this, the reference node is always placed as the first vertex of the element. Thus, the reference node is implicitly given in the data structure. Let us now examine the code in Listing 4.6:

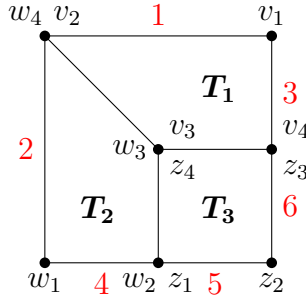


Figure 4.5: Numbering of outer edges for blue elements in QrefineRB.

- Lines 1–5: The function is usually called by

```
[coordinates,elements4,dirichlet,neumann] =
QrefineRB(coordinates,elements4,dirichlet,neumann,
marked4).
```

The variable `elements4` consists of red and blue elements and they are stored block by block, starting with red elements. The last entry of `varargin` is considered to be the marked elements. Again, a persistent variable `nB` is defined which gives the number of blue elements and is updated in the last step (Line 169).
- Lines 6–13: Further geometric data is generated by calling the function `provideGeometricData`. The number of red elements `nR` can easily be determined from the number of blue elements `nB` and the overall number of elements `nE`.
- Lines 14–19: Red elements are marked by marking all edges for refinement (Lines 15–17). For blue elements, only non-marked edges of the father element are marked. Thus, they are determined with `element2edges(nR+[3*marked-2,3*marked+3*nE-1])` and correspond to edge 1 and 2 of the outer edges of a blue element as shown in Figure 4.5.
- Lines 20–23: Refinement patterns for red and blue elements are translated into a binary number and saved in `hashR` and `hashB`. Hence for red elements, `red` (1111), `bluer` (1100), and `bluel` (0011) come into question. With the numbering of edges depicted in Figure 4.5, 110000 is `b2red`, 111001 corresponds to `b2south`, 110110 is `b2east`, and 111111 leads to `b2southeast`; compare Figure 3.45. With 6 outer edges, there are $2^6 = 64$ possibilities of different markings. For all potential markings, the correct refinement pattern is determined and stored in `mapR` and `mapB`.

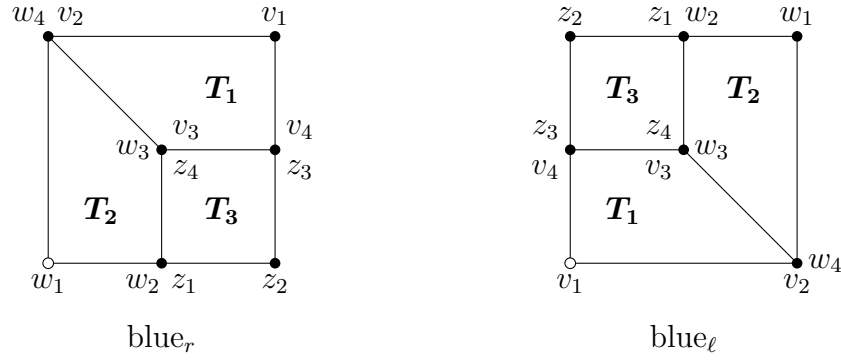
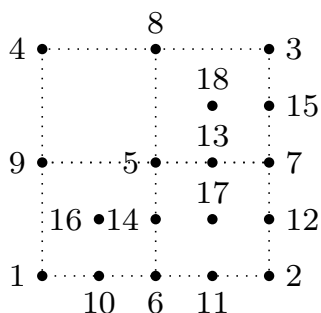


Figure 4.6: Storing blue_r and blue_ℓ elements. It becomes clear that after storing the blue elements are not distinguishable any more.

- Lines 24–33: To ensure the regularity of the grid, further markings may be needed to match appropriate patterns and eliminate all hanging nodes. Thus, this is done in an iterative process until no further changes are made.
- Lines 34–50: The same is done for blue elements. However, for this only the outer edges are concerned. Those are specified in `bit` and ordered in the same way as shown in Figure 4.5. A particularity is that for a marked third or fourth outer edge an additional node needs to be created. Lines 49–50 ensure that the edge $\text{conv}\{v_3v_4\}$ of element T_1 is marked for refinement if the third outer edge is marked and the edge $\text{conv}\{w_2w_3\}$ of element T_2 is marked for refinement if the fourth outer edge is marked; compare Figure 4.5.
- Lines 51–168: The rest of the code is a straightforward adaption of the already discussed implementations. However, some ins and outs exist. A main difference is, that besides the new nodes that are created on edges also a midpoint needs to be generated due to the patterns. This is done in Lines 79–87 for red elements that are refined and in Lines 112–127 for blue elements of type `b2south`, `b2southeast`, and `b2east`. A further particularity is that even though the reference node $\text{ref}_z(T)$ of an element T has an important role and it is distinguished between a blue_r and a blue_ℓ element on the first glance, we will not make a difference between blue elements in the further course. We already stated that reference nodes can be interchanged along the diagonal and that is why no further distinction is needed. Hence, blue elements are stored in the same manner as shown in Figure 4.6. Apart from that, creating the new elements is done with a `tmp` structure which enumeration is shown in Figure 4.7. For the enumeration, we always keep in mind that the new reference node

Figure 4.7: Numbering of the nodes in variable `tmp` in `QrefineRB`.

is stored at location one of the element. The new reference node is always the node from the oldest generation.

Listing 4.6: `QrefineRB`

```

1 function [coordinates,newElements,varargout] ...
2     = QrefineRB(coordinates,elements,varargin)
3 persistent nB;
4 nE = size(elements,1);
5 markedElements = varargin{end};
6 %*** Obtain geometric information on edges
7 [edge2nodes,~,element2edges,boundary2edges{1:nargin-3}] ...
8     = provideGeometricData(zeros(0,3),elements,varargin{1:end-1});
9 %*** Count number of blue sibling elements;
10 if isempty(nB)
11     nB=0;
12 end
13 nR = nE-nB;
14 %*** Mark edges for refinement
15 edge2newNode = zeros(1,size(edge2nodes,1));
16 marked = markedElements(markedElements<=nR);
17 edge2newNode(element2edges(marked,:)) = 1;
18 marked = ceil((markedElements(markedElements>nR)-nR)/3);
19 edge2newNode(element2edges(nR+[3*marked-2,3*marked+3*nE-1])) = 1;
20 hashR = [1,1,1,1;1,1,0,0;0,0,1,1];
21 [mapR,valueR] = hash2map((0:15)',hashR);
22 hashB = logical([1,1,0,0,0,0;1,1,1,0,0,1;1,1,0,1,1,0;1,1,1,1,1,1]);
23 [mapB,valueB] = hash2map((0:63)',hashB);
24 swap = 1;
25 while ~isempty(swap) || any(flags(:))

```

```

26     markedEdge = edge2newNode(element2edges);
27     %*** Change flags for red elements
28     bit = markedEdge(1:nR,:);
29     dec = sum(bit.*(ones(nR,1)*2.^(0:3)),2);
30     valR = valueR(dec+1);
31     [idx,jdx] = find(~bit & mapR(dec+1,:));
32     swap = idx+(jdx-1)*nE;
33     edge2newNode(element2edges(swap)) = 1;
34     %*** Change flags for blue elements
35     bit = [markedEdge(nR+1:3:end,1),markedEdge(nR+2:3:end,4), ...
36           markedEdge(nR+1:3:end,4),markedEdge(nR+2:3:end,1), ...
37           markedEdge(nR+3:3:end,1:2)];
38     dec = sum(bit.*(ones(nB/3,1)*2.^(0:5)),2);
39     valB = valueB(dec+1);
40     bdx = find(valB)';
41     flags = ~bit(bdx,:) & mapB(dec(dec>0)+1,:);
42     edge2newNode(element2edges(nR+3*bdx(flags(:,1))-2,1))=1;
43     edge2newNode(element2edges(nR+3*bdx(flags(:,2))-1,4))=1;
44     edge2newNode(element2edges(nR+3*bdx(flags(:,3))-2,4))=1;
45     edge2newNode(element2edges(nR+3*bdx(flags(:,4))-1,1))=1;
46     edge2newNode(element2edges(nR+3*bdx(flags(:,5)),1))=1;
47     edge2newNode(element2edges(nR+3*bdx(flags(:,6)),2))=1;
48 end
49 edge2newNode(element2edges(nR+3*bdx(hashB(valB(bdx),3))-2,3))=1;
50 edge2newNode(element2edges(nR+3*bdx(hashB(valB(bdx),4))-1,2))=1;
51 %*** Generate new nodes on edges
52 edge2newNode(edge2newNode~=0) = size(coordinates,1)...
53     + (1:nnz(edge2newNode));
54 idx = find(edge2newNode);
55 coordinates(edge2newNode(idx),:) = (coordinates(edge2nodes(idx,1),:))...
56     + coordinates(edge2nodes(idx,2),:))/2;
57 %*** Refine boundary conditions
58 varargout = cell(nargout-2,1);
59 for j = 1:nargout-2
60     boundary = varargin{j};
61     if ~isempty(boundary)
62         newNodes = edge2newNode(boundary2edges{j})';
63         markedEdges = find(newNodes);
64         if ~isempty(markedEdges)
65             boundary = [boundary(~newNodes,:); ...
66                 boundary(markedEdges,1),newNodes(markedEdges); ...

```

```

67         newNodes (markedEdges), boundary (markedEdges, 2) ];
68     end
69 end
70 varargout{j} = boundary;
71 end
72 %*** Provide new nodes for refinement of elements
73 newNodes = edge2newNode(element2edges);
74 %*** Determine type of refinement for each red element
75 none    = find(valR == 0);
76 red     = find(valR == 1);
77 bluer   = find(valR == 2);
78 blue1   = find(valR == 3);
79 %*** Generate new interior nodes if red elements are refined
80 idx = [red, bluer, blue1];
81 midNodes = zeros(nE, 1);
82 midNodes(idx) = size(coordinates, 1) + (1:length(idx));
83 coordinates = [coordinates; ...
84     ( coordinates(elements(idx, 1), :) ...
85     + coordinates(elements(idx, 2), :) ...
86     + coordinates(elements(idx, 3), :) ...
87     + coordinates(elements(idx, 4), :) )/4];
88 %*** Determine type of refinement for each blue element
89 b2blue    = nR + find(valB==0);
90 b2red     = nR + find(valB==1);
91 b2east    = nR + find(valB==2);
92 b2south   = nR + find(valB==3);
93 b2southeast = nR + find(valB==4);
94 %*** Generate element numbering for refined mesh
95 rdx = zeros(nR+nB/3, 1);
96 rdx(none) = 1;
97 rdx([red, b2red]) = 4;
98 rdx([b2south, b2east]) = 2;
99 rdx(b2southeast) = 5;
100 rdx = [1; 1+cumsum(rdx)];
101 bdx = zeros(size(rdx));
102 bdx([bluer, blue1, b2blue]) = 3;
103 bdx([b2south, b2east, b2southeast]) = 6;
104 bdx = rdx(end) + [0; 0+cumsum(bdx)];
105 %*** Generate new red elements
106 tmp = [elements(1:nR, :), midNodes(1:nR, :), newNodes(1:nR, :), ...
107     zeros(nR, 6); elements(nR+2:3:end, 1), elements(nR+3:3:end, 2), ...

```



```

108     elements(nR+1:3:end,1:2),elements(nR+3:3:end,4),...
109     elements(nR+2:3:end,2),elements(nR+1:3:end,4),...
110     newNodes(nR+1:3:end,1),newNodes(nR+2:3:end,4),newNodes(nR...
111     +2:3:end,1),newNodes(nR+3:3:end,:),newNodes(nR+1:3:end,4)];
112 %*** Generate new interior nodes if blue elements are refined
113 dummy = unique([b2south(:);b2southeast(:)]);
114 tmp(dummy,16) = size(coordinates,1)+(1:length(dummy));
115 coordinates = [coordinates; ...
116     (9*coordinates(tmp(dummy,1),:)+3*coordinates(tmp(dummy,2),:)...
117     +1*coordinates(tmp(dummy,3),:)+3*coordinates(tmp(dummy,4),:))/16];
118 dummy = unique([b2south(:);b2southeast(:);b2east(:)]);
119 tmp(dummy,17) = size(coordinates,1)+(1:length(dummy));
120 coordinates = [coordinates; ...
121     (3*coordinates(tmp(dummy,1),:)+9*coordinates(tmp(dummy,2),:)...
122     +3*coordinates(tmp(dummy,3),:)+1*coordinates(tmp(dummy,4),:))/16];
123 dummy = unique([b2east(:);b2southeast(:)]);
124 tmp(dummy,18) = size(coordinates,1)+(1:length(dummy));
125 coordinates = [coordinates; ...
126     (1*coordinates(tmp(dummy,1),:)+3*coordinates(tmp(dummy,2),:)...
127     +9*coordinates(tmp(dummy,3),:)+3*coordinates(tmp(dummy,4),:))/16];
128 %*** Generate new red elements first
129 newElements = 1+zeros(bdx(end)-1,4);
130 newElements(rdx(none),:) = elements(none,:);
131 newElements([rdx(red),1+rdx(red),2+rdx(red),3+rdx(red)],:) ...
132     = [tmp(red,[1,6,5,9]);tmp(red,[2,7,5,6]);...
133     tmp(red,[3,8,5,7]);tmp(red,[4,9,5,8]);];
134 newElements([rdx(b2red),1+rdx(b2red),2+rdx(b2red),3+rdx(b2red)],:) ...
135     = [tmp(b2red,[1,6,5,9]);tmp(b2red,[2,7,5,6]); ...
136     tmp(b2red,[3,8,5,7]);tmp(b2red,[4,9,5,8]);];
137 newElements([rdx(b2east),1+rdx(b2east)],:) ...
138     = [tmp(b2east,[1,6,5,9]);tmp(b2east,[4,9,5,8]);];
139 newElements([rdx(b2south),1+rdx(b2south)],:) ...
140     = [tmp(b2south,[3,8,5,7]);tmp(b2south,[4,9,5,8]);];
141 newElements([rdx(b2southeast),1+rdx(b2southeast), ...
142     2+rdx(b2southeast),3+rdx(b2southeast),4+rdx(b2southeast)],:) ...
143     = [tmp(b2southeast,[6,11,17,14]);tmp(b2southeast,[2,12,17,11]);...
144     tmp(b2southeast,[7,13,17,12]);tmp(b2southeast,[5,14,17,13]); ...
145     tmp(b2southeast,[4,9,5,8]);];
146 %*** New blue elements
147 newElements([bdx(bluer),1+bdx(bluer),2+bdx(bluer)],:) ...
148     = [tmp(bluer,[3,4,5,7]);tmp(bluer,[1,6,5,4]);tmp(bluer,[6,2,7,5]);];

```

```

149 newElements([bdx(blue1),1+bdx(blue1),2+bdx(blue1)],:) ...
150     = [tmp(blue1,[1,2,5,9]);tmp(blue1,[3,8,5,2]);tmp(blue1,[8,4,9,5])];
151 newElements([bdx(b2blue),1+bdx(b2blue),2+bdx(b2blue)],:)= [tmp(...
152     b2blue,[3,4,5,7]);tmp(b2blue,[1,6,5,4]);tmp(b2blue,[6,2,7,5])];
153 newElements([bdx(b2south),1+bdx(b2south),2+bdx(b2south), ...
154     3+bdx(b2south),4+bdx(b2south),5+bdx(b2south)],:) ...
155     = [tmp(b2south,[5,9,16,14]);tmp(b2south,[1,10,16,9]); ...
156     tmp(b2south,[10,6,14,16]);tmp(b2south,[2,7,17,11]); ...
157     tmp(b2south,[5,14,17,7]);tmp(b2south,[14,6,11,17])];
158 newElements([bdx(b2east),1+bdx(b2east),2+bdx(b2east), ...
159     3+bdx(b2east),4+bdx(b2east),5+bdx(b2east)],:) ...
160     = [tmp(b2east,[5,6,17,13]);tmp(b2east,[2,12,17,6]); ...
161     tmp(b2east,[12,7,13,17]);tmp(b2east,[3,8,18,15]); ...
162     tmp(b2east,[5,13,18,8]);tmp(b2east,[13,7,15,18])];
163 newElements([bdx(b2southeast),1+bdx(b2southeast),2+...
164     bdx(b2southeast),3+bdx(b2southeast),4+bdx(b2southeast),...
165     5+bdx(b2southeast)],:) = [tmp(b2southeast,[5,9,16,14]);...
166     tmp(b2southeast,[1,10,16,9]);tmp(b2southeast,[10,6,14,16]);...
167     tmp(b2southeast,[3,8,18,15]);tmp(b2southeast,[5,13,18,8]);...
168     tmp(b2southeast,[13,7,15,18])];
169 nB = size(newElements,1)-rdx(end)+1;

```

Overall, it can be seen that all presented implementations follow the same concept. For some refinement strategies there is more to be considered and thus the code is more extensive. TrefineNVB and TrefineRGB are done in 71 lines of MATLAB. This is the easiest refinement method because elements are not separated into different colors and thus the completion process and the generation of new elements is manageable. In TrefineRG almost as twice as much lines (126) are needed to implement this method. This is because of the separation of red and green elements that always need to be considered differently. Furthermore, QrefineRB has 169 lines of code. The extension is due to the distinction between blue and red elements as well as the generation of midpoints that are needed to create the allowed patterns.

4.3 Implementation with Virtual Elements

Next, we take a closer look at refinement strategies that lead to irregular grids. In this case, irregular edges and their corresponding hanging nodes need to be stored in the existing data structure or new variables need to be defined to differentiate ir-

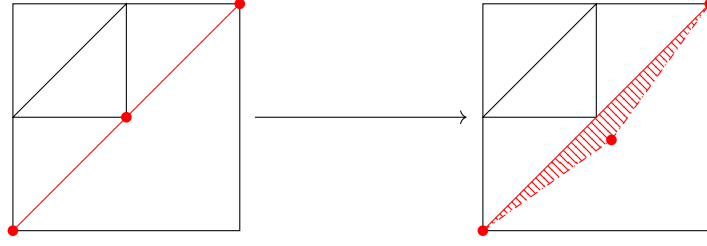


Figure 4.8: Irregularity data as virtual elements. An irregular edge with a hanging node (left) can be interpreted as a virtual triangle (right). For two hanging nodes per edge the virtual element is a quadrilateral.

regular edges from regular edges. This is important because in each refinement step the 1-Irregular Rule might have to be deployed or not. To this end, an additional variable named `irregular` is used that stores the nodes of an irregular edge and its hanging node. We realize this by interpreting the irregular edge as a virtual element; see Figure 4.8. We always set the hanging node on location three of this variable, i.e. `irregular(ℓ , 3)` gives the hanging node of the ℓ -th irregular edge. The edges of this virtual element can then be found by the function `provideGeometricData`. This convention is handy to ensure the 1-irregularity of the grid. If one of the two halves of the irregular edge is marked, then also the unrefined neighbor element is refined.

The procedure for an implementation with virtual elements can be summarized in a few steps. First of all, further geometric data is generated by the function `provideGeometricData`. Here, also the new variable `irregular` is incorporated to receive a numbering for irregular edges. Then, all edges of marked elements are marked for refinement. Furthermore, existing hanging nodes are incorporated in this data. To ensure the 1-irregularity of the grid further edges need to be marked. In particular, if an unrefined element has more than one (two) hanging nodes on its edge the element is refined. This process is iterated until no further changes are made, that is no further edges are marked. From then on, the procedure resembles the approach of hash maps. Firstly, new nodes are generated and boundary data is refined. The location of new nodes is provided. The type of refinement is determined and with this help an element numbering can be created. New elements are generated and additionally all irregular edges are given back as virtual elements such that they can easily be retrieved in the next refinement step. A detailed investigation with focus on some characteristics is given in the upcoming sections.

4.3.1 TrefineR

In TrefineR only two patterns are allowed, namely none or red, see Figure 3.4. This is implemented in Listing 4.7 and explained in the following.

- Lines 1–4: The function is usually called by


```
[coordinates,elements3,irregular,dirichlet,neumann] =
TrefineR(coordinates,elements3,irregular,dirichlet,
neumann,marked3)
```

 where we output all information about the refined mesh. Optional arguments are `dirichlet`, `neumann` and `marked3`. The last entry in `varargin` is anticipated to be the marked elements.
- Lines 5–10: Triangular elements and irregular virtual elements are handed over to the function `provideGeometricData` to receive edge-based information on the mesh.
- Lines 11–14: The variable `edge2newNodes` is created which is set to one for all edges of marked elements. Furthermore, edges with hanging nodes are considered in this marking process, too (Line 14).
- Lines 15–27: To ensure the 1-irregularity of the grid, it is always checked whether further markings entail an introduction of more hanging nodes on one edge. In this case, the 1-Irregular Rule needs to be deployed, that is neighboring elements need to be refined. To get this information, in Lines 22–26 the index in `markedEdge` is set to `-1` for the first edge of the virtual element if the second or third edge is marked. Note, that the first edge of the virtual element always corresponds to the longest edge. Thus, in the completion process elements with a `markedEdge` entry of `-1` are also red refined even though not all edges are marked for refinement (Lines 18–21).
- Lines 28–33: New nodes are generated. Since existing hanging nodes from the last refinement step were considered in the variable `edge2newNode` and those nodes already exist, the related indices are removed. That means only new nodes are generated as the midpoints of marked edges.
- Lines 34–48: Boundary edges are also refined if boundary data is handed over in `varargin`. This is done analogously to all other implementations.

- Lines 49–51: We allot indices of new nodes to the edges where this node is introduced. Not only newly generated nodes but also already existing hanging nodes are considered (Line 50).
- Lines 52–55: Following a binary representation of marked edges, only if all edges are marked, i.e. the corresponding decimal number is 7, the element is red refined. In all other cases (< 7) no refinement is needed.
- Lines 56–60: An element numbering is created. A red refinement results into 4 triangles whereas no refinement only defines one triangle. With this element numbering it is ensured that elements originating from the same father element are listed next to each other.
- Lines 61–68: Elements that are not refined are just adopted from the old mesh. Red refined elements are split into four new similar triangles.
- Lines 69–78: For a complete representation of the refined mesh, new irregularity data needs to be created. This is done for all elements where at least one edge is marked but that are not refined red ($0 < \text{reftyp} < 7$). Thus, edges of these elements are found that were marked for bisection. The new irregularity data is then generated as follows. The first two entries are the nodes that define this irregular edge and the third entry is the corresponding hanging node (Line 73). However, also the former irregularity data needs to be updated. Hence, in Lines 74–75 it is checked whether a former irregular edge receives a new node or not. If so, `newIrregular` is expanded by the nodes that describe the new irregular edge and its corresponding hanging node. Henceforth, `newIrregular` is a $M \times 3$ array where M is the number of irregular edges.

Listing 4.7: TrefineR

```

1 function [coordinates,newElements,newIrregular,varargout] ...
2     = TrefineR(coordinates,elements,irregular,varargin)
3 nE = size(elements,1);
4 markedElements = varargin{end};
5 %*** Obtain geometric information on edges
6 [edge2nodes,element2edges,~,boundary2edges{1:nargin-4}] ...
7     = provideGeometricData([elements;irregular],zeros(0,4),...
8     varargin{1:end-1});

```

```

9  irregular2edges = element2edges(nE+1:end,:);
10 element2edges = element2edges(1:nE,:);
11 %*** Mark edges for refinement and existing hanging nodes
12 edge2newNode = zeros(1,size(edge2nodes,1));
13 edge2newNode(element2edges(markedElements,:)) = 1;
14 edge2newNode(irregular2edges(:,1)) = 1;
15 kdx = 1;
16 while ~isempty(kdx) || ~isempty(swap)
17     markedEdge = edge2newNode(element2edges);
18     %*** Change flags for elements
19     kdx = find(sum(abs(markedEdge),2)<3 & min(markedEdge,[],2)<0);
20     [idx,jdx] = find(~markedEdge(kdx,:));
21     edge2newNode(element2edges(kdx(idx)+(jdx-1)*nE)) = 1;
22     %*** Change flags for irregular marker elements
23     markedEdge = edge2newNode(irregular2edges);
24     flag = irregular2edges(any(markedEdge(:,2:end),2),1);
25     swap = find(edge2newNode(flag) ~= -1);
26     edge2newNode(flag(swap)) = -1;
27 end
28 %*** Generate new nodes on edges
29 edge2newNode(irregular2edges(:,1)) = -1;
30 idx = edge2newNode>0;
31 edge2newNode(idx) = size(coordinates,1) + (1:nnz(idx));
32 coordinates(edge2newNode(idx),:)=(coordinates(edge2nodes(idx,1),:)...
33                                     +coordinates(edge2nodes(idx,2),:))/2;
34 %*** Refine boundary conditions
35 varargout = cell(nargout-3,1);
36 for j = 1:nargout-3
37     boundary = varargin{j};
38     if ~isempty(boundary)
39         newNodes = edge2newNode(boundary2edges{j})';
40         markedEdges = find(newNodes);
41         if ~isempty(markedEdges)
42             boundary = [boundary(~newNodes,:); ...
43                         boundary(markedEdges,1),newNodes(markedEdges); ...
44                         newNodes(markedEdges),boundary(markedEdges,2)];
45         end
46     end
47     varargout{j} = boundary;
48 end
49 %*** Provide new nodes for refinement of elements

```

```

50 edge2newNode(irregular2edges(:,1)) = irregular(:,3);
51 newNodes = reshape(edge2newNode(element2edges), [], 3);
52 %*** Determine type of refinement for each element
53 reftyp = (newNodes ~= 0)*2.^(0:2)';
54 none    = reftyp < 7;
55 red     = reftyp == 7;
56 %*** Generate element numbering for refined mesh
57 idx = zeros(nE,1);
58 idx(none) = 1;
59 idx(red) = 4;
60 idx = [1;1+cumsum(idx)];
61 %*** Generate new elements
62 newElements = zeros(idx(end)-1,3);
63 newElements(idx(none), :) = elements(none, :);
64 newElements([idx(red), 1+idx(red), 2+idx(red), 3+idx(red)], :) ...
65     = [elements(red,1), newNodes(red,1), newNodes(red,3); ...
66     newNodes(red,1), elements(red,2), newNodes(red,2); ...
67     newNodes(red,3), newNodes(red,2), elements(red,3); ...
68     newNodes(red,2), newNodes(red,3), newNodes(red,1)];
69 %*** Generate new irregularity data
70 kdx = find(reftyp > 0 & reftyp < 7);
71 [idx, jdx, val] = find( newNodes(kdx, :));
72 edx = element2edges(kdx(idx)+(jdx-1)*nE);
73 newIrregular = [edge2nodes(edx(:), :), val(:)];
74 newNodes = reshape(edge2newNode(irregular2edges(:,2:3)), [], 2);
75 kdx = find(sum(newNodes, 2) ~= 0);
76 [idx, jdx, val] = find( newNodes(kdx, :));
77 edx = irregular2edges(kdx(idx)+(jdx-1+1)*size(irregular2edges,1));
78 newIrregular = [newIrregular; [edge2nodes(edx(:), :), val(:)]];

```

4.3.2 QrefineR

QrefineR in Listing 4.8 differs slightly from the implementation of TrefineR. Therefore, only the differences are focussed in the following consideration.

- Lines 1–4: The function is usually called by `[coordinates, elements4, irregular, dirichlet, neumann] = QrefineR(coordinates, elements4, irregular, dirichlet, neumann, marked4)`. Optional arguments `dirichlet`, `neumann` and `marked4` are hidden in the `varargin`. The last entry in `varargin` represents the marked elements.

- Lines 5–8: Edge-based information on the mesh is generated with `provideGeometricData`. Here, the irregularity data is handed over as virtual triangles and the elements themselves as quadrilaterals.
- Lines 9–12: The variable `edge2newNode` indicates for all edges whether an edge is marked or not. Thus, for marked elements all edges are set to one. Furthermore, already existing hanging nodes are incorporated in this array.
- Lines 13–26: In those lines, the completion process is done which ensures that the 1-Irregular Rule and the 3-Neighbor Rule are maintained. To this end, in Lines 21–25 the entry in `markedEdges` is set to -1 for the first edge if at least one of the other edges of the virtual elements in `irregular` is marked. With this information, the 1-Irregular Rule is maintained in Lines 16–20. If three edges are marked (3-Neighbor Rule) or there exists an edge with a corresponding value of -1 in `markedEdge` (1-Irregular Rule), the element is red refined.
- Lines 27–32: New nodes are generated that were introduced in this refinement step. Since already existing hanging nodes are also stated in `edge2newNode`, those need to be removed beforehand. This is done by finding all irregular edges and set their corresponding index in `edge2newNode` to -1 . By only choosing the positive entries, it can be ensured that only non-existing nodes are created.
- Lines 33–47: Boundary data is refined as in all other implementations.
- Lines 48–50: The indices of new and existing hanging nodes are stored at the location of edges on which they lie.
- Lines 51–54: Using the binary representation of marking an element, for a quadrilateral $2^4 = 16$ possibilities arise. Thus, for a binary number of 15 all edges are marked for refinement and this leads to a red refinement. In all other cases (< 15) the quadrilateral remains the same.
- Lines 55–63: In contrast to triangles, red refined quadrilaterals also need a further midpoint of the element. The coordinates of these midpoints are created by summing up a forth of all vertices of the element.

- Lines 64–68: An element numbering for the refined mesh is created, where a red refinement leads to four new quadrilaterals. No refinement defines again one element.
- Lines 69–76: New elements are generated.
- Lines 77–86: The new regularity data is created analogously to TrefineR. Regular edges that are refined need to be considered (Lines 78–81) as well as irregular edges that have been further refined (Lines 82–86).

Listing 4.8: QrefineR

```

1 function [coordinates,newElements,newIrregular,varargout] ...
2     = QrefineR(coordinates,elements,irregular,varargin)
3 nE = size(elements,1);
4 markedElements = varargin{end};
5 %*** Obtain geometric information on edges
6 [edge2nodes,irregular2edges,element2edges,...
7     boundary2edges{1:nargin-4}] ...
8     = provideGeometricData(irregular,elements,varargin{1:end-1});
9 %*** Mark edges for refinement and existing hanging nodes
10 edge2newNode = zeros(1,size(edge2nodes,1));
11 edge2newNode(element2edges(markedElements,:)) = 1;
12 edge2newNode(irregular2edges(:,1)) = 1;
13 kdx = 1;
14 while ~isempty(kdx) || ~isempty(swap)
15     markedEdge = edge2newNode(element2edges);
16     %*** Change flags for elements
17     kdx = find(sum(abs(markedEdge),2)<4 & ...
18         (sum(abs(markedEdge),2)>2 | min(markedEdge,[],2)<0));
19     [idx,jdx] = find(~markedEdge(kdx,:));
20     edge2newNode(element2edges(kdx(idx)+(jdx-1)*nE)) = 1;
21     %*** Change flags for irregular marker elements
22     markedEdge = edge2newNode(irregular2edges);
23     flag = irregular2edges(any(markedEdge(:,2:end),2),1);
24     swap = find(edge2newNode(flag) ~= -1);
25     edge2newNode(flag(swap)) = -1;
26 end
27 %*** Generate new nodes on edges
28 edge2newNode(irregular2edges(:,1)) = -1;

```

```

29 idx = edge2newNode>0;
30 edge2newNode(idx) = size(coordinates,1) + (1:nz(idx));
31 coordinates(edge2newNode(idx),:)= (coordinates(edge2nodes(idx,1),:)) ...
32                               +coordinates(edge2nodes(idx,2),:))/2;
33 %*** Refine boundary conditions
34 varargout = cell(nargout-3,1);
35 for j = 1:nargout-3
36     boundary = varargin{j};
37     if ~isempty(boundary)
38         newNodes = edge2newNode(boundary2edges{j})';
39         markedEdges = find(newNodes);
40         if ~isempty(markedEdges)
41             boundary = [boundary(~newNodes,:); ...
42                         boundary(markedEdges,1),newNodes(markedEdges); ...
43                         newNodes(markedEdges),boundary(markedEdges,2)];
44         end
45     end
46     varargout{j} = boundary;
47 end
48 %*** Provide new nodes for refinement of elements
49 edge2newNode(irregular2edges(:,1)) = irregular(:,3);
50 newNodes = reshape(edge2newNode(element2edges),[],4);
51 %*** Determine type of refinement for each element
52 reftyp = (newNodes~=0)*2.^(0:3)';
53 none    = reftyp < 15;
54 red     = reftyp == 15;
55 %*** Generate new interior nodes if red elements are refined
56 idx = find(red);
57 midNodes = zeros(nE,1);
58 midNodes(idx) = size(coordinates,1)+(1:length(idx));
59 coordinates = [coordinates;...
60               ( coordinates(elements(idx,1),:) ...
61               + coordinates(elements(idx,2),:) ...
62               + coordinates(elements(idx,3),:) ...
63               + coordinates(elements(idx,4),:) )/4];
64 %*** Generate element numbering for refined mesh
65 idx = zeros(nE,1);
66 idx(none) = 1;
67 idx(red) = 4;
68 idx = [1;1+cumsum(idx)];
69 %*** Generate new elements

```

```

70 newElements = zeros(idx(end)-1,4);
71 newElements(idx(none),:) = elements(none,:);
72 newElements([idx(red),1+idx(red),2+idx(red),3+idx(red)],:)=...
73     [elements(red,1),newNodes(red,1),midNodes(red),newNodes(red,4);...
74     elements(red,2),newNodes(red,2),midNodes(red),newNodes(red,1);...
75     elements(red,3),newNodes(red,3),midNodes(red),newNodes(red,2);...
76     elements(red,4),newNodes(red,4),midNodes(red),newNodes(red,3)];
77 %*** Generate new irregularity data
78 kdx = find(reftyp >0 & reftyp < 15);
79 [idx,jdx,val] = find(newNodes(kdx,:));
80 edx = element2edges(kdx(idx)+(jdx-1)*nE);
81 newIrregular = [edge2nodes(edx,:),val(:)];
82 newNodes = reshape(edge2newNode(irregular2edges(:,2:3)),[],2);
83 kdx = find(sum(newNodes,2) ~= 0);
84 [idx,jdx,val] = find(newNodes(kdx,:));
85 edx = irregular2edges(kdx(idx)+(jdx-1+1)*size(irregular2edges,1));
86 newIrregular = [newIrregular;[edge2nodes(edx(:,:),val(:)]];

```

4.3.3 QrefineR2

QrefineR2 follows the same implementation structure. However, there are some changes due to the fact that an edge is trisected instead of bisected and thus the number of new introduced nodes per edge is two and not one anymore. We investigate the differences of Listing 4.9 to the previous two implementations.

- Lines 1–4: The function is usually called by `[coordinates,elements4,irregular,dirichlet,neumann] = QrefineR2(coordinates,elements4,irregular,dirichlet,neumann,marked4)`. Optional arguments `dirichlet`, `neumann` and `marked4` are hidden in `varargin`. The last entry in `varargin` is always interpreted as marked elements.
- Lines 5–11: Geometric information on edges is provided by calling the function `provideGeometricData`. Here, no triangular elements exist. However, irregular virtual elements are quadrilaterals, see Figure 4.9, and therefore incorporated in the quadrilateral data. Afterwards, the data is again separated into `irregular2edges` and `element2edges`. An additional variable `orientation` is introduced. As the name suggests, this variable tells about the orientation of the nodes and will be used in a later step.

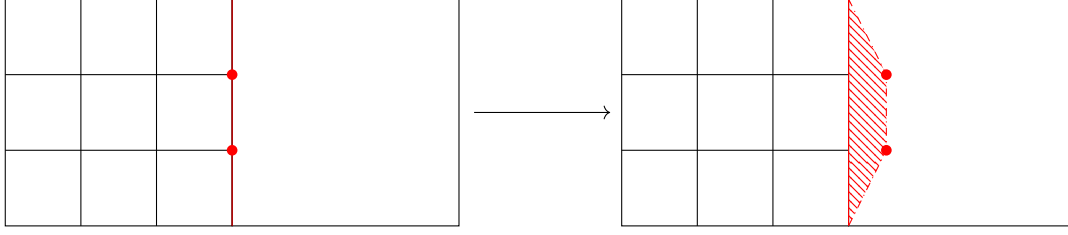


Figure 4.9: Irregularity data for 2-irregular grids. For a maximum of two hanging nodes per edge, the irregular edge can be interpreted as a virtual quadrilateral, depicted as the hashed area. The edges of this virtual quadrilateral are one long edge and three smaller edges that are basically just parts of the long edge.

- Lines 12–15: `edge2newNode` is a binary array of dimension $1 \times \#$ 'of edges' where `edge2newNode(1, ℓ)` is set to one if the ℓ -th edge is marked and zero otherwise. We mark elements by marking all edges of this element for refinement (Line 14). Furthermore, also irregular edges are set to one (Line 15).
- Lines 16–29: These lines describe the completion process which ensures the 2-irregularity of the grid and the 3-Neighbor Rule. For this, if one of the three small edges of an irregular quadrilateral is marked, see Figure 4.9, then the long edge is set to -1 (Lines 26–28). If more than two edges of an element are marked or one edges entry is set to -1 , the element is `red2` refined (Lines 20–24). This further marking is done until no further changes are made, i.e. `kdx` and `swap` are both empty arrays.
- Lines 30–40: New nodes on marked edges shall be generated. For `QrefineR2`, a marked edge is trisected. That means two new nodes per edges are introduced. That is the reason for extending the variable `edge2newNode` by itself in Line 31. Since in `edge2newNode` already existing hanging nodes are taken into account, those need to be removed before generating the nodes (Lines 32–33). Then, two new nodes per marked edge are created by trisecting the edge.
- Lines 41–59: Boundary conditions are refined as in all other implementations apart from one change. The boundary edges are refined by trisecting the edges instead of bisecting. Thus, a refined boundary edge defines three new edges instead of two.
- Lines 60–62: Indices for the newly generated nodes are stored at the location of the edges where they are introduced.

- Lines 63–66: For each element the refinement pattern is determined from the possible ones, i.e. none and red2. A red2 refinement is done if all edges of the quadrilateral is marked. This corresponds to a decimal number of 15.
- Lines 67–82: New irregularity data on the refined mesh is generated. For all non-refined elements with at least one marked edge the variable `newIrregular` is defined with nodes of the irregular edge and its corresponding two hanging nodes as entries (Lines 68–74). Furthermore, former irregular edges that are refined further are updated (Lines 78–82).
- Lines 83–86: To define red2 refined elements, the indices of four new midpoints are created.
- Lines 87–91: An element numbering is generated where no refinement leads to one quadrilateral and a red2 refinement defines 9 new triangles.
- Lines 92–96: To ensure that the edges of an element are stored in a mathematical positive sense, the order of hanging nodes on some edges needs to be changed. An illustration is given in Figure 4.10.
- Lines 97–113: New elements are generated in four steps, always defining a corner element and its neighboring element in mathematical positive sense. That is T_1 and T_2 are generated for $j = 1$, T_3 and T_4 for $j = 2$ and so on. In each step, the coordinates for the new midpoints are also generated. In a final step the middle element T_9 is defined (Line 113). The situation is depicted in Figure 4.11.

Listing 4.9: QrefineR2

```

1 function [coordinates,newElements,newIrregular,varargout] ...
2     = QrefineR2(coordinates,elements,irregular,varargin)
3 nE = size(elements,1);
4 markedElements = varargin{end};
5 %*** Obtain geometric information on edges
6 [edge2nodes,~,element2edges,boundary2edges{1:nargin-4}] ...
7     = provideGeometricData(zeros(0,3),[elements;irregular],...
8     varargin{1:end-1});
9 irregular2edges = element2edges(nE+1:end,:);

```

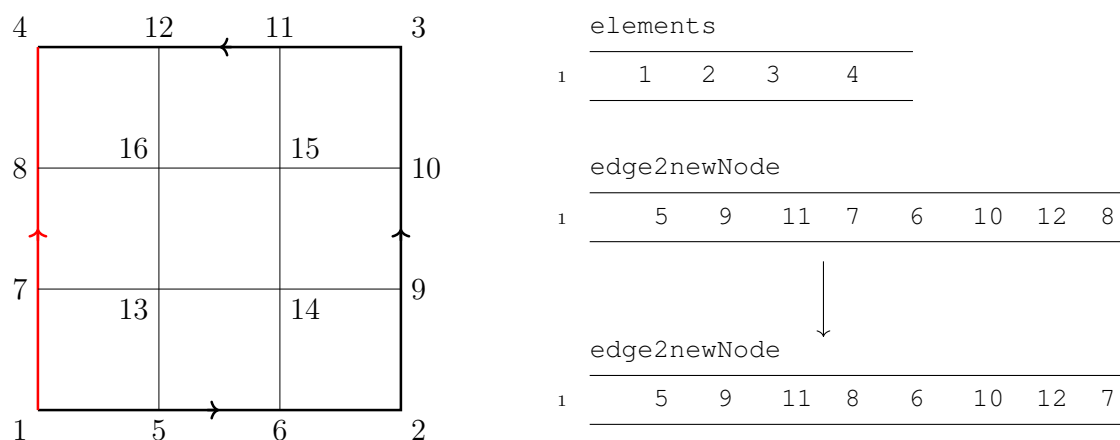


Figure 4.10: Orientation of edges in QrefineR2. For an element defined as in `elements`, `elements(:, [2, 3, 4, 1]) < elements = [0 0 0 1]` and thus for the last edge the order of hanging nodes needs to be interchanged. This has the following reasons: The variable `edge2nodes` defines the two nodes of an edge where the first entry is always set to the smallest index. Hence, `edge2nodes` defines implicitly a direction indicated by arrows. Therefore, the first introduced hanging node lies closer to the node with the smallest index than the second generated hanging node on this edge. `edge2newNode(1:4, :)` gives the first introduced hanging nodes on each edge and `edge2newNode(5:8, :)` the second generated hanging nodes on the nodes. If now opposite lying nodes need to be connected to get a red2 refinement, a correct assignment of nodes is necessary. Thus, on the edge with nodes 1 and 4 the nodes 7 and 8 are interchanged to fulfill the mathematical positive sense for all edges of the quadrilateral.

T_7	T_6	T_5
T_8	T_9	T_4
T_1	T_2	T_3

Figure 4.11: Order of newly generated elements after a red2 refinement. The elements are created in four steps. For $j = 1$ the corner element T_1 and its neighboring element T_2 is generated. For $j = 2$ the corner element T_3 and its neighbor T_4 is created, and so on. In the last step the middle element T_9 is defined.

```

10 element2edges = element2edges(1:nE,:);
11 orientation = elements(:, [2,3,4,1]) < elements;
12 %*** Mark edges for refinement and existing hanging nodes
13 edge2newNode = zeros(1, size(edge2nodes,1));
14 edge2newNode(element2edges(markedElements,:)) = 1;
15 edge2newNode(irregular2edges(:,1)) = 1;
16 kdx = 1;
17 while ~isempty(kdx) || ~isempty(swap)
18     markedEdge = edge2newNode(element2edges);
19     %*** Change flags for elements
20     kdx = find(sum(abs(markedEdge),2) < 4 & ...
21         (sum(abs(markedEdge),2) > 2 | min(markedEdge,[],2) < 0));
22     [idx, jdx] = find(~markedEdge(kdx,:));
23     edge2newNode(element2edges(kdx(idx) + (jdx-1)*nE)) = 1;
24     %*** Change flags for irregular marker elements
25     markedEdge = edge2newNode(irregular2edges);
26     flag = irregular2edges(any(markedEdge(:,2:end),2),1);
27     swap = find(edge2newNode(flag) ~ -1);
28     edge2newNode(flag(swap)) = -1;
29 end
30 %*** Generate new nodes on edges
31 edge2newNode = edge2newNode([1,1],:);
32 edge2newNode(irregular2edges(:,1),:) = -1;
33 idx = edge2newNode(:,1) > 0;
34 edge2newNode(idx,:) = size(coordinates,1) + reshape(1:2*nnz(idx),2,[],:);
35 coordinates(edge2newNode(idx,1),:) ...
36     = (2*coordinates(edge2nodes(idx,1),:) + ...
37     coordinates(edge2nodes(idx,2),:))/3;
38 coordinates(edge2newNode(idx,2),:) ...
39     = (coordinates(edge2nodes(idx,1),:) + ...
40     2*coordinates(edge2nodes(idx,2),:))/3;
41 %*** Refine boundary conditions
42 varargout = cell(nargout-3,1);
43 for j = 1:nargout-3
44     boundary = varargin{j};
45     if ~isempty(boundary)
46         newNodes = edge2newNode(boundary2edges{j},:);
47         markedEdges = find(newNodes(:,1));
48         if ~isempty(markedEdges)
49             ind = boundary(markedEdges,1) > boundary(markedEdges,2);
50             newNodes(markedEdges(ind),:) ...

```

```

51         = newNodees (markedEdges (ind), [2,1]);
52         boundary = [boundary (~newNodes (:,1),:); ...
53             boundary (markedEdges,1), newNodees (markedEdges,1); ...
54             newNodees (markedEdges,1), newNodees (markedEdges,2); ...
55             newNodees (markedEdges,2), boundary (markedEdges,2)];
56     end
57 end
58     varargout{j} = boundary;
59 end
60 %*** Provide new nodes for refinement of elements
61 edge2newNode (irregular2edges (:,1),:) = irregular (:,[4,3]);
62 newNodes = reshape (edge2newNode (element2edges,:), [],8);
63 %*** Determine type of refinement for each element
64 reftyp = (newNodes (:,1:4) ~= 0)*2.^(0:3)';
65 none    = reftyp < 15;
66 red2    = reftyp == 15;
67 %*** Generate new irregularity data
68 kdx = find (reftyp > 0 & reftyp < 15);
69 if ~isempty (kdx)
70     [idx,jdx] = find ( newNodes (kdx,1:4));
71     edx = element2edges (kdx (idx)+(jdx-1)*nE);
72     newIrregular = [edge2nodes (edx (:),:), ...
73         newNodees (kdx (idx (:))+(jdx (:)+3)*nE), ...
74         newNodees (kdx (idx (:))+(jdx (:)-1)*nE)];
75 else
76     newIrregular = zeros (0,4);
77 end
78 newEdgeNodes = reshape (edge2newNode (irregular2edges (:,2:4),1), [],3);
79 kdx = find (sum (newEdgeNodes,2) ~= 0);
80 [idx,jdx,val] = find ( newEdgeNodes (kdx,:));
81 edx = irregular2edges (kdx (idx)+(jdx-1+1)*size (irregular2edges,1));
82 newIrregular = [newIrregular; [edge2nodes (edx (:),:), val (:)+1, val (:)]];
83 %*** Generate new interior nodes if red elements are refined
84 sdx = find (red2);
85 midNodes = zeros (nE,4);
86 midNodes (sdx,:) = size (coordinates,1)+reshape (1:4*length (sdx), [],4);
87 %*** Generate element numbering for refined mesh
88 idx = zeros (nE,1);
89 idx (none)    = 1;
90 idx (red2)    = 9;
91 idx = [1;1+cumsum (idx)];

```



```

92 %*** Incorporate orientation of edges
93 [idx0,jdx0] = find(orientation);
94 tmp = newNodes(idx0 + (jdx0-1)*nE);
95 newNodes(idx0 + (jdx0-1)*nE) = newNodes(idx0 + (jdx0+3)*nE);
96 newNodes(idx0 + (jdx0+3)*nE) = tmp;
97 %*** Generate new elements
98 newElements = zeros(idx(end)-1,4);
99 newElements(idx(none), :) = elements(none, :);
100 s = [2,3,4,1]; p=[8,5,6,7];
101 c = [4,2,1,2;2,4,2,1;1,2,4,2;2,1,2,4];
102 for j=1:4
103     newElements([2*(j-1)+idx(red2),2*j-1+idx(red2)], :) = ...
104         [elements(red2,j),newNodes(red2,j),midNodes(red2,j),...
105         newNodes(red2,p(j));newNodes(red2,j),newNodes(red2,j+4),...
106         midNodes(red2,s(j)),midNodes(red2,j)];
107     coordinates = [coordinates;...
108         (c(1,j)*coordinates(elements(sdx,1), :) ...
109         + c(2,j)*coordinates(elements(sdx,2), :) ...
110         + c(3,j)*coordinates(elements(sdx,3), :) ...
111         + c(4,j)*coordinates(elements(sdx,4), :))/9];
112 end
113 newElements(8+idx(red2), :) = midNodes(red2, :);

```

To sum up, irregular refinement methods are also easy to implement. TrefineR is done in 78 lines of MATLAB, QrefineR is a little more extensive with 86 lines. This is because of the midpoints that need to be created additionally. QrefineR2 is the most complicated one because besides the fact that always two new nodes per edge and four midpoints are created, also the orientation of edges need to be considered. The implementation is realized in 113 lines.

4.4 Implementation in Three Steps

Only QrefineRGtri and QrefineRG2 are not covered by the implementation with hash maps or virtual elements. One could imagine to implement these refinement strategies with hash maps, too. However, this would lead to an excessive code. Let us illustrate why that is the case for QrefineRGtri. A red element can be marked in $2^4 = 16$ different ways and is mapped to 12 different refinement patterns. These 12 refinement patterns can then be categorized into four types, namely red, green

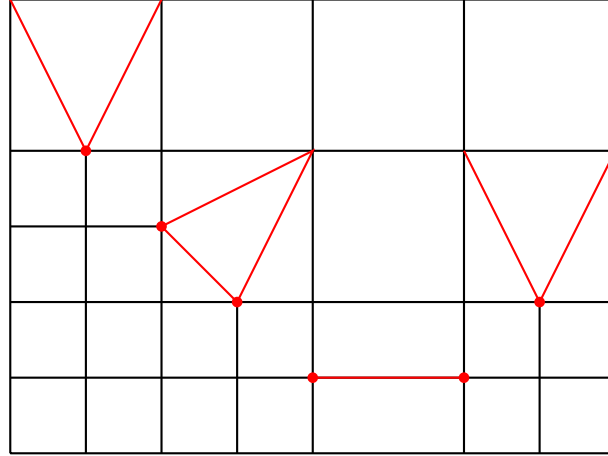


Figure 4.12: Regularizing an irregular grid and vice versa. The output of `QrefineR` is an irregular mesh depicted in black. To eliminate hanging nodes, the mesh is regularized by matching the appropriate refinement patterns green type 1, green type 2 and green type 3 (illustrated in red). In the next refinement step, the regularizing patterns need to be removed to input an irregular mesh to `QrefineR`. The approach is subdivided into three step: `recoarseedges` – `QrefineR` – `regularizeedges`. The same approach works for `QrefineRG2`.

type 1, green type 2 and green type 3. There exist $2^5 = 32$ possibilities to mark a green element of type 1 and this is mapped to eight different refinement patterns. For green elements of type 2 and type 3, there are $2^6 = 64$ possible markings and they are mapped to 17 different refinement patterns. Again, these can be grouped into the same categories. For `QrefineRG2`, the number of possibilities even reaches $2^8 = 256$ for a green2 element of type 2 or 3. In other words, mapping all marking combinations to the allowed refinement patterns becomes quite confusing with hash maps. Therefore, for these two refinement methods we follow a different approach that is based on three different steps. We want to make use of the implementation for irregular meshes. The basic idea is to regularize a refined irregular mesh by matching allowed green patterns to eliminate hanging nodes; see Figure 4.12. In the upcoming refinement step, those regularity edges need to be removed to input an irregular mesh to the function `QrefineR` or `QrefineR2`. In summary, the process *recoarse regularity edges - refine irregular grid - regularize edges* is iterated until no further refinement is requested.

4.4.1 QrefineRGtri

It is favorable to first explain the function `regularizeedges_tri` in Listing 4.10 because some data structures that are created in this function are exploited in

`recoarseedges_tri.`

- Lines 1–2: The function is usually called by `[coordinates,elements4,elements3] = regularizeedges_tri(coordinates,elements4,irregular)` where the inputs arguments are given by `coordinates`, `elements4`, and `irregular`. We wish to output a regularized mesh into quadrilaterals and triangles and therefore output `elements4` and `elements3`.
- Lines 4–11: Edge-based information on the mesh is provided by the function `provideGeometricData`. This time, also additional geometric information `edge2element` is constructed. `edge2element(ℓ ,1)` gives the corresponding element for the ℓ -th edge of the mesh and `edge2element(ℓ ,2)` gives the position of the edge in that element, i.e. 1,2,3 or 4. Note that an edge is not uniquely assigned to one element as usually two elements share one edge. In this case, the corresponding element T is chosen randomly. However, for irregular edges the assignment is unambiguous.
- Lines 12–17: With the help of the just introduced variable `edge2element`, elements with irregular edges and their location within the elements are found. The corresponding hanging node is stored in `IrregularEdges`.
- Lines 18–25: As a next step, from `IrregularEdges` it can be read out how many hanging nodes exist and where they are located. Using this information it can be clearly stated which refinement patterns need to be used. For one hanging node per element (`marks==1`), a `green1` pattern is needed. For two adjacent edges with hanging nodes, a `green2` pattern is used (Line 24) and for two opposite lying edges with hanging nodes, a `green3` pattern can be matched (Line 25).
- Lines 26–30: Since within the refinement type also different orientations are possible, those need to be determined, too. For a `green2` element, this is done in the following way. We enumerate the edges of a quadrilateral and set them to one if the edge has a hanging node and to zero otherwise. We consider an array that is of order `[4,1,2,3]`. If the first and fourth edge is marked `orientation_green2=1`. For the first and second edge, it is 2. For the third and fourth edge the result is 4. An exception is given if the the second and third edge is marked. In that case `orientation_green2` is 5 and thus all

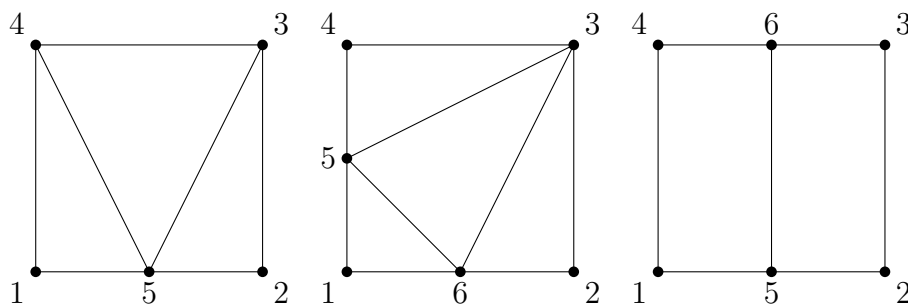


Figure 4.13: Numbering of nodes in `tmp` for green elements in `regularizeedges_tri`.

entries with 5 are corrected to 3 in Line 28. Hence, for a `green2` element four different orientations exist. For `green3` elements only two different orientations exist. Therefore, only the first two edges are considered. For an orientation of `green1` only the position of the edge with hanging node within the element is given as `orientation_green1`.

- Lines 31–37: To be able to differentiate between a red element and a green element, the following unification is done. The minimal index of node is set at location one for all red elements. Furthermore, the new irregularity data is preallocated in `elements4` where red elements are stored as unified elements in a block.
- Lines 38–54: In these lines, the elements are rotated until all green elements are located such that the orientation is one. Thus, per green type only one case is left over.
- Lines 55–58: New `tmp` structures for all refinement types are defined as shown in Figure 4.13.
- Lines 59–75: New elements are generated using the `tmp` structure. For the green type 3 refinement it is ensured that the first entry of the element is not the minimum. Thus, newly generated nodes 5 and 6 are the leading nodes for these elements. With this data structure, green elements of type 3 can be distinguished from red elements.

The output of `regularizeedges_tri` is a regular mesh into quadrilaterals and triangles.

Listing 4.10: QrefineRGtri - regularizededges_tri

```

1 function [coordinates,newElements4, newElements3] ...
2     = regularizededges_tri(coordinates,elements,irregular)
3 nE= size(elements,1);
4 %*** Provide geometric data
5 [~,irregular2edges,element2edges] ...
6     = provideGeometricData(irregular,elements);
7 edge2element = zeros(nE,2);
8 for k = 1:4
9     edge2element(element2edges(:,k),1) = 1:nE;
10    edge2element(element2edges(:,k),2) = k;
11 end
12 %*** Provide irregularity data
13 IrregularEdges=zeros(nE,4);
14 for i = 1:length(irregular(:,1))
15     id=edge2element(irregular2edges(i,:),:);
16     IrregularEdges(id(:,1),id(:,2))=irregular(i,3);
17 end
18 %*** Determine type of refinement
19 markedEdges=IrregularEdges(:,:);
20 markedEdges(markedEdges~=0)=1;
21 marks = sum(markedEdges,2);
22 none = marks==0;
23 green1 = marks==1;
24 green2 = min((markedEdges(:,1:2)+markedEdges(:,3:4)),[],2)==1;
25 green3 = max((markedEdges(:,1:2)+markedEdges(:,3:4)),[],2)==2;
26 % Determine orientation of refinement patterns
27 orientation_green2=markedEdges(green2,[4,1,2,3])*[0,1,1,4]';
28 orientation_green2(orientation_green2==5)=3;
29 [orientation_green3,~,~]=find(markedEdges(green3,1:2)'==1);
30 [orientation_green1,~,~]=find(markedEdges(green1(:,1),:)'==1);
31 % Standardizing elements such that z1<min{z2,z3,z4} for regular
    elements
32 elements_tmp=elements(none,:);
33 ind=elements_tmp(:,2)==min(elements_tmp,[],2);
34 elements_tmp(ind,:)=elements_tmp(ind,[2,3,4,1]);
35 newElements4 = [elements_tmp; zeros(length(orientation_green3)*2,4)];
36 newElements3 = zeros(length(orientation_green1)*3 ...
37     +length(orientation_green2)*4,3);
38 %*** Rotate data until orientation of refinement pattern is at 1
39 elements_gr1=elements(green1,:);

```

```

40 elements_gr2=elements (green2,:);
41 elements_gr3=elements (green3,:);
42 edge2newNodegr2=IrregularEdges (green2(:,1),[4,1,2,3]);
43 edge2newNodegr3=IrregularEdges (green3(:,1),1:4);
44 A = [1,2,3,4;4,1,2,3;3,4,1,2;2,3,4,1];
45 for i =2:4
46     idx=orientation_green1==i;
47     jdx=orientation_green2==i;
48     elements_gr1 (idx,A(i,:))=elements_gr1 (idx,:);
49     elements_gr2 (jdx,A(i,:))=elements_gr2 (jdx,:);
50     edge2newNodegr2 (jdx,A(i,:))=edge2newNodegr2 (jdx,:);
51 end
52 jdx=orientation_green3==2;
53 elements_gr3 (jdx,A(2,:))=elements_gr3 (jdx,:);
54 edge2newNodegr3 (jdx,A(2,:))=edge2newNodegr3 (jdx,:);
55 %*** Generate new nodes and elements
56 tmp_green1=[elements_gr1,sum (IrregularEdges (green1(:,1),:),2)];
57 tmp_green2=[elements_gr2,edge2newNodegr2(:,1:2)];
58 tmp_green3=[elements_gr3,edge2newNodegr3(:,[1,3])];
59 if ~isempty (tmp_green1)
60     gdx1=1:3:length (orientation_green1)*3;
61     newElements3 ([gdx1,1+gdx1,2+gdx1],:)=...
62         [tmp_green1(:,[1,5,4]); tmp_green1(:,[3,4,5]);...
63         tmp_green1(:,[2,3,5])];
64 end
65 if ~isempty (tmp_green2)
66     gdx2 = length (orientation_green1)*3+...
67         (1:4:length (orientation_green2)*4);
68     newElements3 ([gdx2,1+gdx2,2+gdx2,3+gdx2],:)=...
69         [tmp_green2(:,[4,5,3]); tmp_green2(:,[5,6,3]);...
70         tmp_green2(:,[6,2,3]); tmp_green2(:,[5,1,6])];
71 end
72 if ~isempty (tmp_green3)
73     gdx3=nnz (newElements4(:,1))+1:2:length (newElements4);
74     newElements4 ([gdx3,1+gdx3],:)= [tmp_green3(:,[5,2,3,6]);...
75     tmp_green3(:,[6,4,1,5])];
76 end

```

For a further refinement, regularizing edges need to be eliminated before the grid can be refined with the help of `QrefineR`. This is done with `recoarseedges_tri` in Listing 4.11.

- Lines 1–2: The function is usually called by `[coordinates,elements4,marked,irregular] = recoarseedges_tri(coordinates,elements3,elements4,marked3,marked4)` with the standard input arguments `coordinates`, `elements` and `marked`. Since the mesh in `QrefineRGtri` is a mixture of triangles and quadrilaterals, a distinction between triangles '3' and quadrilaterals '4' is made. Boundary conditions are not further considered in this function.
- Lines 3 and 85–88: In the case where there are no green quadrilaterals or triangles the mesh consists of quadrilaterals only and there are no irregular edges. Thus, we output the same mesh with irregularity data `zeros(0,3)`.
- Lines 3–34: If there are green elements that need to be coarsened, it is helpful to know which type of green element it is. Thus, for quadrilaterals the constructed data structure is exploited. Hence, the first index where the first entry of `elements4` is not the minimal entry is assigned to green elements of type 3 (Lines 5–8). For green elements of type 2, it is checked for which indices three consecutive triangles have a some corner (Lines 12–23), corresponding to node 3 in the middle picture of Figure 4.13. All other triangles are green elements of type 1 (Lines 24–30). Furthermore, if no green elements of type 3 are found all elements in `elements4` are considered to be red elements (Lines 31–34).
- Lines 35–37: Red elements remain unchanged and are stored in `newElements`.
- Lines 38–59: As input two different marked arrays are given, one for triangles and one for quadrilaterals. If green elements are marked their corresponding father element is considered to be marked. To this end, for all types of green elements, the father element is marked if at least one of the triangles/quadrilaterals are marked for refinement. `marked4` gives the new data that will be handed back as output.
- Lines 60–77: All green elements are coarsened to its father element and thus the data structure that was built in `regularizeedges` is again exploited. We state the indices of nodes in `tmp` as depicted in Figure 4.13. Thus, `tmp_green1 = [1,5,4,3,4,5,2,3,5]` and the father element `[1,2,3,4]` is created by choosing the indices `[1,7,4,3]` in Line 63 and for the irregular edge it is

searched for the indices $[1, 2, 5]$ which are found at locations $[1, 7, 2]$ in Line 64. The same is done for the other two refinement types. `tmp_green2 = [4, 5, 3, 5, 6, 3, 6, 2, 3, 5, 1, 6]` and `tmp_green3 = [5, 2, 3, 6, 6, 4, 1, 5]`. Out of this data structure father elements and irregularity data is built.

- Lines 78–86: As a last step, the numbering of each element is standardized such that the minimal index is in the first column of `newElements`.

Listing 4.11: QrefineRGtri - recoarseedges_tri

```

1 function [coordinates,newElements,marked4,irregular]...
2     =recoarseedges_tri(coordinates,elements3,elements4,marked3,marked4)
3 if nnz(find(elements4(:,1))...
4     ~= min(elements4,[],2))) || ~isempty(elements3)
5     %*** Determine type of refinement for quadrilaterals
6     nE=length(elements4(:,1));
7     rdx=find(elements4(:,1) ~= min(elements4,[],2));
8     gdx3=rdx:2:nE;
9     %*** Determine type of refinement for triangles
10    nG = length(elements3);
11    gdx2 = nG-3;
12    while gdx2>0
13        if elements3(gdx2,3) == elements3(gdx2+1,3) && ...
14            elements3(gdx2+1,3) == elements3(gdx2+2,3)
15            gdx2 = gdx2 -4;
16            if gdx2 ==1
17                break;
18            end
19        else
20            gdx2 = gdx2+4;
21            break;
22        end
23    end
24    if gdx2 ==0
25        gdx2 = 4:4:nG; gdx1 = 1;
26    elseif gdx2== -3
27        gdx2 = []; gdx1 = [];
28    else
29        gdx1 = 1:3:gdx2-1; gdx2 = gdx2:4:nG;
30    end

```



```

31  %*** Initial triangulation is not sorted
32  if isempty(rdx)
33      rdx = nE+1;
34  end
35  %*** Generate red elements
36  newElements=elements4(1:rdx(1)-1,:);
37  irregular=[];
38  % Mark father element for marked green elements
39  markedelements3=zeros(length(elements3(:,1)),1);
40  markedelements4=zeros(length(elements4(:,1)),1);
41  markedelements3(marked3)=1;
42  markedelements4(marked4)=1;
43  newmarks=markedelements4(1:rdx(1)-1);
44  if ~isempty(gdx1)
45      marks1 = ...
46      max(reshape(markedelements3(gdx1(1):gdx1(end)+2)',3,[]))';
47      newmarks=[newmarks;marks1];
48  end
49  if ~isempty(gdx2)
50      marks2 = ...
51      max(reshape(markedelements3(gdx2(1):gdx2(end)+3)',4,[]))';
52      newmarks=[newmarks;marks2];
53  end
54  if ~isempty(gdx3)
55      marks3 = ...
56      max(reshape(markedelements4(gdx3(1):gdx3(end)+1)',2,[]))';
57      newmarks=[newmarks;marks3];
58  end
59  marked4=find(newmarks==1);
60  %*** Recoarse green elements and generate irregularity data
61  if ~isempty(gdx1)
62      tmp_green1=reshape(elements3(gdx1(1):gdx1(end)+2,:),9,[]);
63      newElements = [newElements;tmp_green1(:, [1,7,4,3])];
64      irregular = [irregular;tmp_green1(:, [1,7,2])];
65  end
66  if ~isempty(gdx2)
67      tmp_green2 =reshape(elements3(gdx2(1):gdx2(end)+3,:),12,[]);
68      newElements = [newElements;tmp_green2(:, [11,8,3,1])];
69      irregular = [irregular;tmp_green2(:, [11,8,5]);...
70                  tmp_green2(:, [1,11,2])];
71  end

```

```

72     if ~isempty(gdx3)
73         tmpgreen3=reshape(elements4(gdx3(1):gdx3(end)+1,:)',8,[]);
74         newElements = [newElements;tmpgreen3(:, [7,2,3,6])];
75         irregular = [irregular;tmpgreen3(:, [7,2,1]);...
76                     tmpgreen3(:, [3,6,4])];
77     end
78     %*** Ensure z1 < z2 for irregularity data
79     change_ind=min(irregular,[],2)==(irregular(:,2));
80     irregular(change_ind,:)=irregular(change_ind,[2,1,3]);
81     %*** Ensure z1 < min(z2,z3,z3) for all elements
82     A=[1,2,3,4;4,1,2,3;3,4,1,2;2,3,4,1];
83     for i = 1:4
84         change_ind=min(newElements,[],2)==newElements(:,i);
85         newElements(change_ind,A(i,:))=newElements(change_ind,:);
86     end
87 else
88     newElements=elements4;
89     irregular=zeros(0,3);
90 end
91 end

```

4.4.2 QrefineRG2

As above, we also start to explain the function `regularizeedges` in Listing 4.12 because some of the built data structure is exploited in the function `recoarseedge`. The structure of both codes are very similar to the ones in `QrefineRGtri`, however, there are additional characteristics to consider.

- Lines 1–2: The function is usually called by `[coordinates, newElements, marked, irregular] =recoarseedges (coordinates, elements4, marked4)` with input arguments `coordinates`, `elements4` and `marked4`. Boundary conditions are not considered within this function.
- Lines 6–15: Edge-based information of the mesh is provided by the function `provideGeometricData`. Additional geometric information `edge2element` is created as in `regularizeedges_tri`.
- Lines 16–21: With the help of the introduced geometric data, irregular edges can be assigned to a specific edge in an element. At those locations the two

hanging nodes are stored in `IrregularEdges`. Note, that `IrregularEdges` is a three dimensional variable with the `IrregularEdges(:, :, 1)` telling the first hanging node on an edge and `IrregularEdges(:, :, 2)` the second hanging node.

- Lines 22–27: Analogously to Figure 4.10, it is ensured that the edges are numbered in a mathematical positive sense.
- Lines 28–35: By determining the number of marked edges within an element and the position of marked edges, the refinement pattern can be assigned to each element. Thus, for no marked edges (`marks==0`), the element stays the same. For one marked edge per element (`marks==1`), the element is green2 type 1 refined. And for two adjacent marked edges (Line 34) and two opposite lying marked edges (Line 35) the elements are green2 type 2 refined and green2 type 3 refined, respectively.
- Lines 36–40: There exist four different orientations of refinement patterns for green2 type 1 and type 2 elements and further two orientations for green2 type 3 elements. The position of each green element is determined by following the same strategy as in `regularizededges_tri`.
- Lines 41–46: `newElements` are created with a block of red2 elements corresponding to none first. For all red2 elements it is ensured that the first entry of an element is the minimal index within this element.
- Lines 47–50: The patterns in `QrefineRG2` require to generate interior nodes for green2 elements of type 1 and type 2. Here, `midnodes1` and `midnodes2` give the indices of coordinates of the new midnodes that are created at a later point.
- Lines 51–72: For each type of green2 elements, the data is rotated until for each type only one orientation is left over. This simplifies the code drastically.
- Lines 73–118: New elements and new midnodes are created. For each green2 refinement type a `tmp` structure is created that is visualized in Figure 4.14. With this numbering new elements are created such that the inner edges are stated first and a mathematical positive order is guaranteed. This helps to

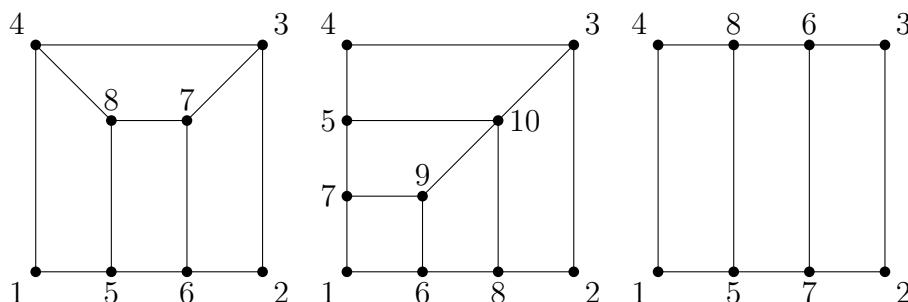


Figure 4.14: Numbering of nodes in tmp for green elements in regularizededges.

still identify red2 elements by their minimal entry at location one. This data structure will be exploited in the function `recoarseedges`.

Listing 4.12: QrefineRG2 - regularizededges

```

1 function [coordinates,newElements] ...
2     = regularizededges(coordinates,elements,irregular)
3 nE= size(elements,1);
4 nC = size(coordinates,1);
5 nI = size(irregular,1);
6 %*** provide geometric data
7 [~,~,element2edges] ...
8     = provideGeometricData(zeros(0,3),[elements;irregular]);
9 irregular2edges = element2edges(end-(nI)+1:end,1);
10 element2edges = element2edges(1:end-nI,:);
11 edge2element = zeros(nE,2);
12 for k = 1:4
13     edge2element(element2edges(:,k),1) = 1:nE;
14     edge2element(element2edges(:,k),2) = k;
15 end
16 %*** Provide irregularity data
17 IrregularEdges = zeros(nE,4,2);
18 for i = 1:length(irregular(:,1))
19     id=edge2element(irregular2edges(i),:);
20     IrregularEdges(id(:,1),id(:,2),:)=irregular(i,3:4);
21 end
22 %*** Check orientation of hanging nodes
23 orientation_irr=elements(:, [2,3,4,1])>elements;
24 changeOr = [logical(zeros(size(orientation_irr))),orientation_irr];
25 tmp = IrregularEdges;
26 IrregularEdges(orientation_irr)=IrregularEdges(changeOr);

```

```

27 IrregularEdges(changeOr)=tmp(orientation_irr);
28 %*** Determine type of refinement
29 markedEdges=IrregularEdges(:, :, 1);
30 markedEdges(markedEdges ~= 0)=1;
31 marks = sum(markedEdges, 2);
32 none = marks==0;
33 green1 = marks==1;
34 green2 = min((markedEdges(:, 1:2)+markedEdges(:, 3:4)), [], 2)==1;
35 green3=max((markedEdges(:, 1:2)+markedEdges(:, 3:4)), [], 2)==2;
36 %*** Determine orientation of refinement patterns
37 orientation_green2=markedEdges(green2, [4, 1, 2, 3])*[0, 1, 1, 4]';
38 orientation_green2(orientation_green2==5)=3;
39 [orientation_green3,~,~]=find(markedEdges(green3, 1:2)')==1;
40 [orientation_green1,~,~]=find(markedEdges(green1(:, 1), :)')==1;
41 % standardize elements such that z1 < min{z2,z3,z4} for regular
    elements
42 elements_tmp=elements(none, :);
43 ind=elements_tmp(:, 2)==min(elements_tmp, [], 2);
44 elements_tmp(ind, :)=elements_tmp(ind, [2, 3, 4, 1]);
45 newElements = [elements_tmp; zeros(length(orientation_green1)*4 ...
46     +length(orientation_green2)*5+length(orientation_green3)*3, 4)];
47 %*** Generate new interior nodes for green1 and green2 elements
48 midnodes1=nC+reshape(1:2*length(orientation_green1), [], 2);
49 midnodes2=nC+2*max(size(midnodes1(:, 1), 1))+...
50     reshape(1:2*length(orientation_green2), [], 2);
51 %*** Rotate data until orientation of refinement patterns is at 1
52 elements_gr1=elements(green1, :);
53 elements_gr2=elements(green2, :);
54 elements_gr3=elements(green3, :);
55 edge2newNodegr2_1=IrregularEdges(green2(:, 1), [4, 1, 2, 3], 1);
56 edge2newNodegr2_2=IrregularEdges(green2(:, 1), [4, 1, 2, 3], 2);
57 edge2newNodegr3_1=IrregularEdges(green3(:, 1), 1:4, 1);
58 edge2newNodegr3_2=IrregularEdges(green3(:, 1), 1:4, 2);
59 A = [1, 2, 3, 4; 4, 1, 2, 3; 3, 4, 1, 2; 2, 3, 4, 1];
60 c = [4, 2, 1, 2; 2, 4, 2, 1; 1, 2, 4, 2; 2, 1, 2, 4];
61 for i =2:4
62     idx=orientation_green1==i;
63     jdx=orientation_green2==i;
64     elements_gr1(idx, A(i, :))=elements_gr1(idx, :);
65     elements_gr2(jdx, A(i, :))=elements_gr2(jdx, :);
66     edge2newNodegr2_1(jdx, A(i, :))=edge2newNodegr2_1(jdx, :);

```

```

67     edge2newNodegr2_2(jdx,A(i,:))=edge2newNodegr2_2(jdx,:);
68 end
69 jdx=orientation_green3==2;
70 elements_gr3(jdx,A(2,:))=elements_gr3(jdx,:);
71 edge2newNodegr3_1(jdx,A(2,:))=edge2newNodegr3_1(jdx,:);
72 edge2newNodegr3_2(jdx,A(2,:))=edge2newNodegr3_2(jdx,:);
73 %*** Generate new nodes and elements
74 tmp_green1=[elements_gr1,sum(IrregularEdges(green1(:,1),:,1),2),...
75     sum(IrregularEdges(green1(:,1),:,2),2),midnodes1];
76 tmp_green2=[elements_gr2,edge2newNodegr2_1(:,1:2),...
77     edge2newNodegr2_2(:,1:2),midnodes2];
78 tmp_green3=[elements_gr3,edge2newNodegr3_1(:,[1,3]),...
79     edge2newNodegr3_2(:,[1,3])];
80 if ~isempty(tmp_green1)
81     coordinates=[coordinates;...
82         (c(1,3)*coordinates((tmp_green1(:,1)),:)) ...
83         + c(2,3)*coordinates((tmp_green1(:,2)),:) ...
84         + c(3,3)*coordinates((tmp_green1(:,3)),:) ...
85         + c(4,3)*coordinates((tmp_green1(:,4)),:))/9];
86     coordinates=[coordinates;...
87         (c(1,4)*coordinates((tmp_green1(:,1)),:)) ...
88         + c(2,4)*coordinates((tmp_green1(:,2)),:) ...
89         + c(3,4)*coordinates((tmp_green1(:,3)),:) ...
90         + c(4,4)*coordinates((tmp_green1(:,4)),:))/9];
91     gdx1=nnz(newElements(:,1))+(1:4:(length(orientation_green1)*4));
92     newElements([gdx1,gdx1+1,gdx1+2,gdx1+3],:)=...
93         [tmp_green1(:,[7,8,5,6]);tmp_green1(:,[7,6,2,3]);...
94         tmp_green1(:,[8,7,3,4]);tmp_green1(:,[5,8,4,1])];
95 end
96 if ~isempty(tmp_green2)
97     coordinates=[coordinates;...
98         (c(1,1)*coordinates((tmp_green2(:,1)),:)) ...
99         + c(2,1)*coordinates((tmp_green2(:,2)),:) ...
100        + c(3,1)*coordinates((tmp_green2(:,3)),:) ...
101        + c(4,1)*coordinates((tmp_green2(:,4)),:))/9];
102     coordinates=[coordinates;...
103         (c(1,3)*coordinates((tmp_green2(:,1)),:)) ...
104         + c(2,3)*coordinates((tmp_green2(:,2)),:) ...
105         + c(3,3)*coordinates((tmp_green2(:,3)),:) ...
106         + c(4,3)*coordinates((tmp_green2(:,4)),:))/9];
107     gdx2=nnz(newElements(:,1))+(1:5:(length(orientation_green2)*5));

```

```

108     newElements([gdx2,gdx2+1,gdx2+2,gdx2+3,gdx2+4],:)=...
109         [tmp_green2(:, [10,9,6,8]);tmp_green2(:, [10,8,2,3]);...
110         tmp_green2(:, [5,10,3,4]);tmp_green2(:, [10,5,7,9]);...
111         tmp_green2(:, [1,6,9,7])];
112 end
113 if ~isempty(tmp_green3)
114     gdx3=nnz(newElements(:,1))+(1:3:(length(orientation_green3)*3));
115     newElements([gdx3,gdx3+1,gdx3+2],:)=...
116         [tmp_green3(:, [6,7,2,3]);tmp_green3(:, [5,8,4,1]);...
117         tmp_green3(:, [6,8,5,7])];
118 end

```

Before the regular mesh can be refined further, the regularity edges need to be coarsened before it can be inputted to the function `QrefineR2`. This is done in `recoarseedges` in Listing 4.13 that is investigated in the following.

- Lines 1–2 : The function is usually called by
`[coordinates, newElements, marked, irregular]`
`=recoarseedges (coordinates, elements4, marked4)` with input arguments `coordinates`, `elements4` and `marked4`. Boundary conditions do not need to be adapted and are therefore not considered in this function.
- Lines 3 and 77–80: If for all elements the minimal entry is the first entry of `elements`, then there are no green2 elements and the regular mesh is just the inputted mesh with irregularity data `zeros(0,4)`.
- 4–23: If this is not the case, green2 elements exist. For a further treatment, green2 elements are classified into their type of refinements. We exploit the data structure defined in `regularizeedges`. Red2 elements are stored as a block followed by elements of green2 type 1. For green2 elements of type 1, it is ensured that the minimal entry is never set at position one and thus the first element where the minimal entry is not the first entry of `elements` determines the red2 and green2 elements (Line 5). Green2 elements of type 3 are stored as a block at the end of `elements`. Thus, by comparing when the second index is the same for two consecutively stored elements, this can be assigned to a green2 element of type 3. This is because the last two elements of a green2 type 3 refinement are stored in the order `[5,8,4,1]` and `[6,8,5,7]` (Lines 7–9). Hence, the second entry is the same. Green2 elements of type 2 can

be determined by searching for their last set element $[1, 6, 7, 9]$. Here, the minimal value is at position one (Line 11). A green2 element of type 1 can then be determined as the remaining green2 elements (Lines 12–18).

- Lines 24–26: With this classification, red2 elements are adopted without modifications.
- Lines 27–46: Green2 elements are coarsened to its father element. Thus, if a green2 quadrilateral is marked for refinement, its father element needs to be marked, too.
- Lines 47–64: Green2 elements are coarsened to its father element by exploiting the data structure.
 $\text{tmp_green1} = [7, 8, 5, 6, 7, 6, 2, 3, 8, 7, 3, 4, 5, 8, 4, 1]$, $\text{tmp_green2} = [10, 9, 6, 8, 10, 8, 2, 3, 5, 10, 3, 4, 9, 10, 5, 7, 1, 6, 9, 7]$, and $\text{tmp_green3} = [6, 7, 2, 3, 5, 8, 4, 1, 6, 8, 5, 7]$ contain the indices shown in Figure 4.14. The father element can then be achieved from this in Lines 50, 55, and 61. Irregularity data is given by the two nodes of the irregular edge and their two corresponding hanging nodes (Lines 51, 56–57, 62–63).
- Lines 65–67: By coarsening green2 elements to their corresponding father elements, also the midpoints are not needed any more. Those nodes were introduced at the end and thus only the last entries of coordinates need to be removed.
- Lines 68–76: `irregular` and `elements` are standardized such that the minimal entry in both arrays is at position one.

Listing 4.13: QrefineRG2 - recoarseedges

```

1 function [coordinates,newElements,marked,irregular]...
2     =recoarseedges(coordinates,elements,marked)
3 if nnz(elements(:,1) ~= min(elements,[],2))>0
4     *** Determine type of refinement
5     rdx=find(elements(:,1) ~= min(elements,[],2));
6     green_elements=elements(rdx(1):end,:);
7     tmp=flipud(green_elements(:,2));
8     green3=tmp==[tmp(2:end,:);0];

```



```

9      green3=find(green3((1:length(green3))) ~= 0);
10     green_elements=green_elements(1:end-length(green3)*3,:);
11     green2=find(green_elements(:,1)==min(green_elements,[],2))-5;
12     if ~isempty(green2)
13         green1=(1:4:green2(1)-1)-1;
14     elseif ~isempty(green3)
15         green1=(1:4:green3(1)-1-rdx(1))-1;
16     else
17         green1=(1:4:length(green_elements(:,1))-1)-1;
18     end
19     gdx1=(green1+rdx(1))';
20     nG1=length(green1);
21     gdx2=green2+rdx(1);
22     nG2=length(green2);
23     gdx3=length(elements)-flipud(green3)-1;
24     %*** Generate red elements
25     newElements=elements(1:rdx(1)-1,:);
26     irregular=[];
27     %*** Mark father element for marked green elements
28     markedelements=zeros(length(elements(:,1)),1);
29     markedelements(marked)=1;
30     newmarks=markedelements(1:rdx(1)-1);
31     if ~isempty(gdx1)
32         marks1 = ...
33             max(reshape(markedelements(gdx1(1):gdx1(end)+3)',4,[]))';
34         newmarks=[newmarks;marks1];
35     end
36     if ~isempty(gdx2)
37         marks2 = ...
38             max(reshape(markedelements(gdx2(1):gdx2(end)+4)',5,[]))';
39         newmarks=[newmarks;marks2];
40     end
41     if ~isempty(gdx3)
42         marks3 = ...
43             max(reshape(markedelements(gdx3(1):gdx3(end)+2)',3,[]))';
44         newmarks=[newmarks;marks3];
45     end
46     marked=find(newmarks==1);
47     %*** Recoarse green elements and generate irregularity data
48     if ~isempty(gdx1)
49         tmp_green1=reshape(elements(gdx1(1):gdx1(end)+3,:)',16,[]);

```

```

50     newElements=[newElements;tmp_green1(:, [7,8,12,16])];
51     irregular=[irregular;tmp_green1(:, [7,16,3,4])];
52 end
53 if ~isempty(gdx2)
54     tmp_green2=reshape(elements(gdx2(1):gdx2(end)+4,:)',20,[]);
55     newElements=[newElements;tmp_green2(:, [17,7,8,12])];
56     irregular=[irregular;tmp_green2(:, [7,17,3,4]);...
57         tmp_green2(:, [12,17,15,14])];
58 end
59 if ~isempty(gdx3)
60     tmp_green3=reshape(elements(gdx3(1):gdx3(end)+2,:)',12,[]);
61     newElements=[newElements;tmp_green3(:, [3,4,7,8])];
62     irregular=[irregular;tmp_green3(:, [3,8,5,2]);...
63         tmp_green3(:, [4,7,10,9])];
64 end
65 %*** Delete midpoints of green elements
66 nMidpoints=nG1*2+nG2*2;
67 coordinates=coordinates(1:end-nMidpoints,:);
68 %*** Ensure z1 < z2 for irregularity data
69 change_ind=min(irregular,[],2)==(irregular(:,2));
70 irregular(change_ind,:)=irregular(change_ind,[2,1,4,3]);
71 %*** Ensure z1 < min(z2,z3,z3) for all elements
72 A=[1,2,3,4;4,1,2,3;3,4,1,2;2,3,4,1];
73 for i = 1:4
74     change_ind=min(newElements,[],2)==newElements(:,i);
75     newElements(change_ind,A(i,:))=newElements(change_ind,:);
76 end
77 else
78     newElements=elements;
79     irregular=zeros(0,4);
80 end
81 end

```

All in all, these two refinement strategies can also be realized in a total of $86 + 76 + 91 = 253$ lines for QrefineRGtri and $113 + 118 + 81 = 312$ lines for QrefineRG2 even though we saw in the beginning that there are many different cases to consider.

5 Numerical Experiments

In this chapter, we test our implementation exemplary. Firstly, we apply the algorithms in the framework of the adaptive finite element method. Secondly, it is presented that the implementation can also be applied in a different context. All tests are performed on an Apple MacBook Air with the following specifications:

Kernel :	1,6GHz Intel Core i5
RAM :	8 GB 1600 MHz DDR3
OS :	Mac OS High Sierra, Version 10.13.2
MATLAB :	Version 9.2.0 (R2017a)

5.1 Solving Poisson Problems with AFEM

The main purpose of this work is to design an efficient implementation of adaptive mesh refinement for use in the context of adaptive finite element methods. Since the adaptive finite element method is based on four steps, namely **SOLVE - ESTIMATE - MARK -REFINE**, further implementations of the other modules are needed. For triangular meshes without hanging nodes this is already implemented in the work of Funken et al. in [2]. Implementations for quadrilaterals and elements with hanging nodes used for our examples are specifically implemented for this work and originate from Stefan A. Funken. Within this framework, we show examples with Dirichlet boundary data as well as one with mixed boundary data. Let us consider a L-shaped domain $\Omega = (-1, 1)^2 \setminus ([0, 1] \times [-1, 0])$ for our examples. The L-shaped domain is non-convex with a re-entrant corner at $(0, 0)$ and thus the solution u has less regularity. To approximate the solution accurately in a local region of the re-entrant corner a finer mesh is needed than in other parts of the domain. Thus, an application of the standard finite element with uniform mesh refinements fails to provide an accurate solution while maintaining computational tractability. However, adaptive finite element methods are superior since they can provide meshes that are finer near the re-entrant corner; see for example Figure 5.4. We regard the adaptive finite element method with a residual-based error estimator and Dörfler marking

with parameter $\theta = 0.5$ as presented in the general framework in Chapter 2. Within this scope, the implementations of the introduced mesh refinement strategies in this work are used.

5.1.1 Example 1

As a first example, we consider the Poisson problem with Dirichlet boundary data given through

$$\begin{aligned} -\Delta u &= 1 && \text{in } \Omega, \\ u(x, y) &= 0 && \text{on } \Gamma = \partial\Omega. \end{aligned} \quad (5.1)$$

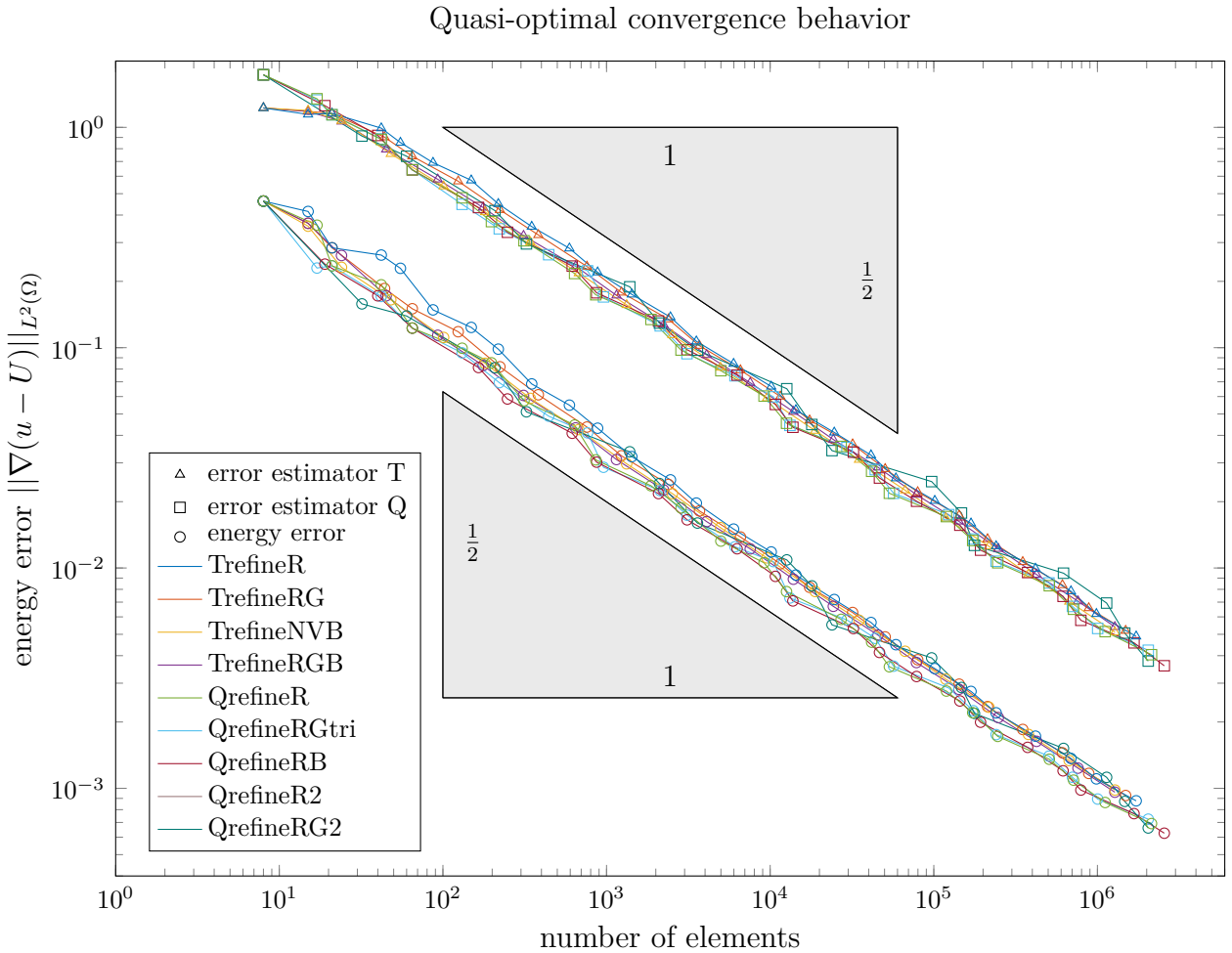


Figure 5.1: Error estimator η and energy error $\|\nabla(u - U)\|_{L^2(\Omega)}$ with respect to the number of elements for Example (5.1). We consider the adaptive finite element method with a residual-based error estimator, Dörfler marking and all presented mesh refinement strategies. All refinement strategies have the same quasi-optimal convergence behavior.

The estimator η and energy error $\|\nabla(u - U)\|_{L^2(\Omega)}$ are depicted in Figure 5.1. For the computation of the energy error we make use of the Galerkin orthogonality, i.e.

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

The discrete energy $\|\nabla U\|_{L^2(\Omega)}^2$ can be computed with the help of the stiffness matrix A and the coefficient vector x of the Galerkin solution U through

$$\|\nabla U\|_{L^2(\Omega)}^2 = x^T A x.$$

Here, the exact solution u is not given analytically and thus we used, as in [22], Aitken's method to extrapolate the value $\|\nabla u\|_{L^2(\Omega)}^2 = 0.21407587$. If the exact solution is known, it is straightforward to compute $\|\nabla u\|_{L^2(\Omega)}^2$. One observes that for this example *all of our* mesh refinement strategies have the same quasi-optimal convergence behavior $N^{-1/2}$ where N is the number of elements. In general, the quasi-optimal energy error decay is proportional to $N^{-1/d}$, with d being the dimension of $\Omega \in \mathbb{R}^d$ [26]. The result shown in Figure 5.1 is promising as the theory of convergence of adaptive finite element methods to the best of our knowledge still lacks to explain the optimal convergence rates for some refinement methods. Proofs of convergence are stated in [21, 25] for TrefineNVB and in [34] for TrefineRG. However, these proofs do not show any superiority of adaptive finite element methods over standard finite element methods since they do not provide an analysis of the rate of convergence. The only known works that provide convergence rates of adaptive finite element methods using newest vertex bisection are the followings: Binev, Dahmen, and DeVore proved convergence rates for the Laplace operator based on the utilization of a coarsening strategy in [13] and a separate marking strategy through oscillation terms. Later on, Stevenson showed in [30] that coarsening is not needed. In 2008, Cascon, Kreuzer, Nochetto, and Siebert extended those results for general second order linear, symmetric elliptic operators and avoided separate marking due to oscillation terms; see [17]. To the best of our knowledge, there are no works on convergence rates for other refinement strategies apart from the newest vertex bisection so far.

5.1.2 Example 2

As a second example, we consider the Laplace problem with Dirichlet boundary data given in polar coordinates (r, α) through

$$\begin{aligned} -\Delta u &= 0 && \text{in } \Omega, \\ u(r, \alpha) &= r^{\frac{2}{3}} \sin \frac{2}{3} \alpha && \text{on } \Gamma = \partial\Omega. \end{aligned} \quad (5.2)$$

Here, $u(r, \alpha) = r^{\frac{2}{3}} \sin \frac{2}{3} \alpha$ is the exact solution of the problem. As in Example (5.1), we determine the same quasi-optimal rate of convergence in the H^1 -norm $\|u - U\|_{H^1(\Omega)}$ as shown in Figure 5.2.

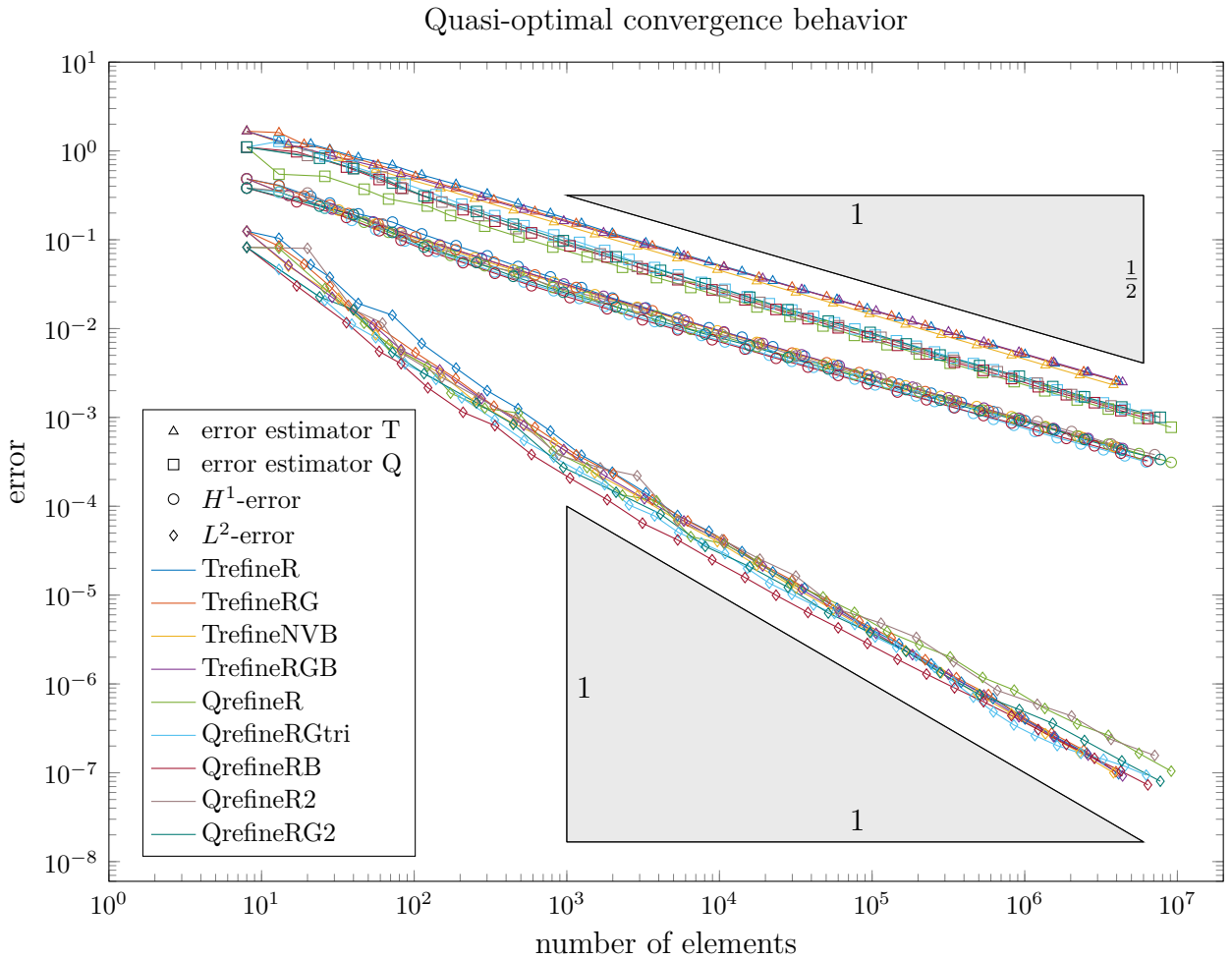


Figure 5.2: Error estimator η , H^1 - error $\|u - U\|_{H^1(\Omega)}$ and L^2 - error $\|u - U\|_{L^2(\Omega)}$ with respect to the number of elements for Example (5.2). We consider the adaptive finite element method with a residual-based error estimator, Dörfler marking and all presented mesh refinement strategies. All refinement strategies have the same quasi-optimal convergence behavior.

Since all mesh refinement strategies lead to the same convergence rate, it is of interest

to compare computational times. To this end, we measure the time for assembly and solution of the system, the time for computation of the residual-based error estimator η , marking, and the refinement of marked elements. This is shown in Figure 5.3. We observe that all implementations lead to (almost) linear growth. However, we also determine a gap between triangular meshes and quadrilateral meshes. The computational time for quadrilateral meshes exceeds the one for triangular meshes. This is explained by the assembly of the stiffness matrix. To compute the entries in the stiffness matrix, integrals have to be evaluated; see (2.5). In the case of triangular elements, the basis functions are linear and their gradient is constant. Thus, the integral of a constant term is easy to determine; compare [22]. In the case of quadrilateral elements, the basis functions are bilinear, which leads to a non-constant gradient and as a consequence Gaussian quadrature is needed for evaluation. For this, a bilinear mapping of each quadrilateral to the reference quadrilateral $[-1, 1]$ is done. We use 4 Gaussian points in each direction for an evaluation. With this indispensable fact in mind, we see that the implementation is still quite efficient for quadrilateral elements.

For further presentation of different refinement strategies, different states of refinements are shown in Figure 5.4, Figure 5.5, and Figure 5.6.

To depict the superiority of the adaptive method for this example, we compare standard and adaptive finite element methods for the mesh refinement strategy TrefineRGB. The standard finite element method uses uniform mesh refinements which is achieved by setting the parameter $\theta = 1$ in the Dörfler marking. We see the superiority of the adaptive method ($\theta = 0.5$) and uniform method ($\theta = 1$) for this example in Figure 5.7.

5.1.3 Example 3

As a third example, we consider the Poisson problem with mixed boundary conditions. For this, the Dirichlet boundary Γ_D is as shown in Figure 4.1 and Figure 4.2 and $\Gamma_N = \partial\Omega \setminus \Gamma_D$. The problem then reads

$$\begin{aligned} -\Delta u &= -1 && \text{in } \Omega, \\ u(x, y) &= \frac{1}{2}x^2 && \text{on } \Gamma_D, \\ \partial_n u &= |x| && \text{on } \Gamma_N, \end{aligned} \tag{5.3}$$

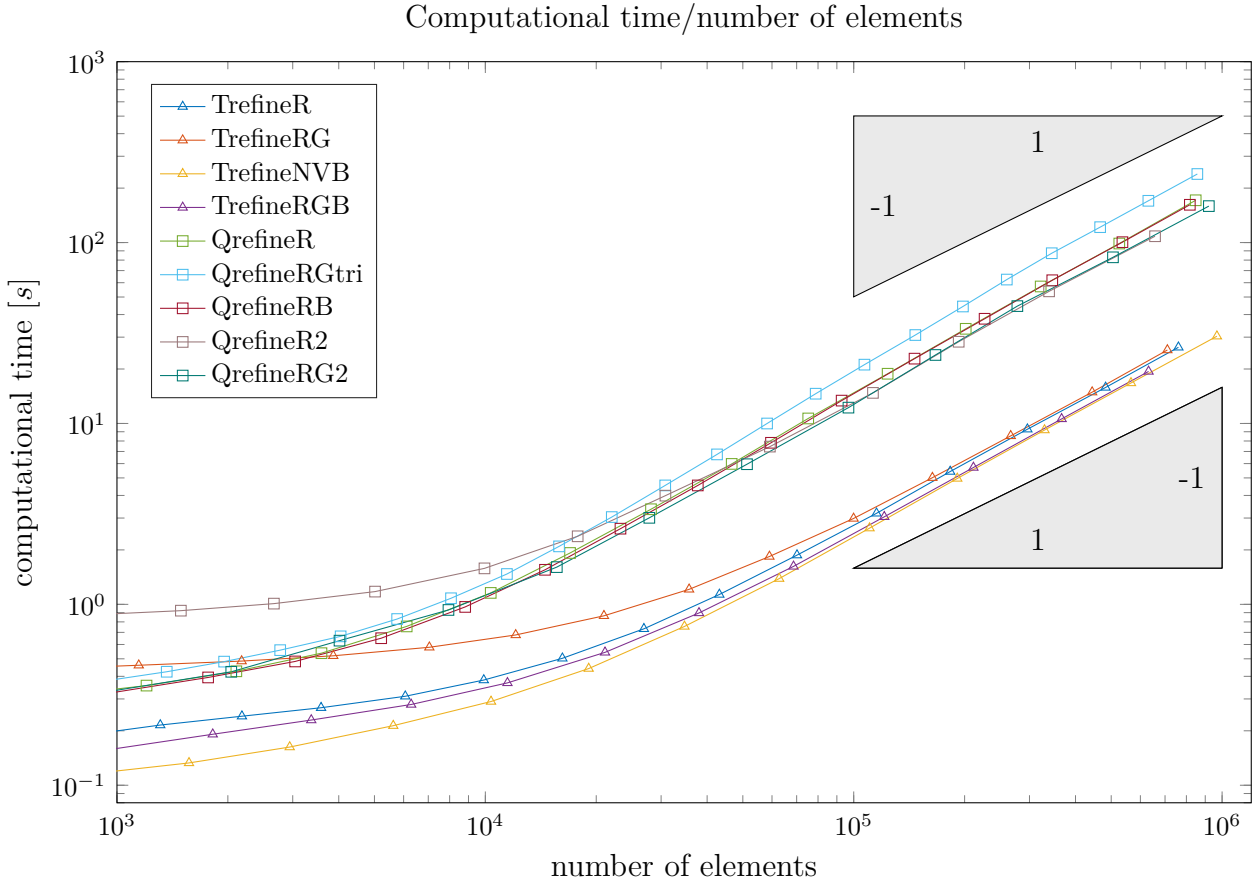


Figure 5.3: Computational times for different refinement strategies for Example (5.2) with respect to the number of elements. We measure the time for the assembly and solution of the system, the time for computation of the residual-based error estimator and the refinement of marked elements. For each refinement step, we cumulate the time from the adaptive history and the newest refinement step. One observes that all implementations lead to (almost) linear growth. The adaptive finite element method is faster for triangular meshes. In particular, the well-known newest vertex bisection is the fastest in our example.

and has the exact solution $u(x, y) = \frac{1}{2}x^2$. Again, we receive quasi-optimal convergence rates for all mesh refinement strategies as illustrated in Figure 5.8.

5.2 Mesh Refinements within a Different Context

In this section, we show that our mesh refinement strategies with their implementations are universally usable. Since the Dirichlet and Neumann boundary data are optional arguments in the implementation, it is easy to use our algorithms in other contexts where mesh refinement is needed, too. For example, we can refine elements along a circle as depicted in Figure 5.9. But also other contexts are conceivable.

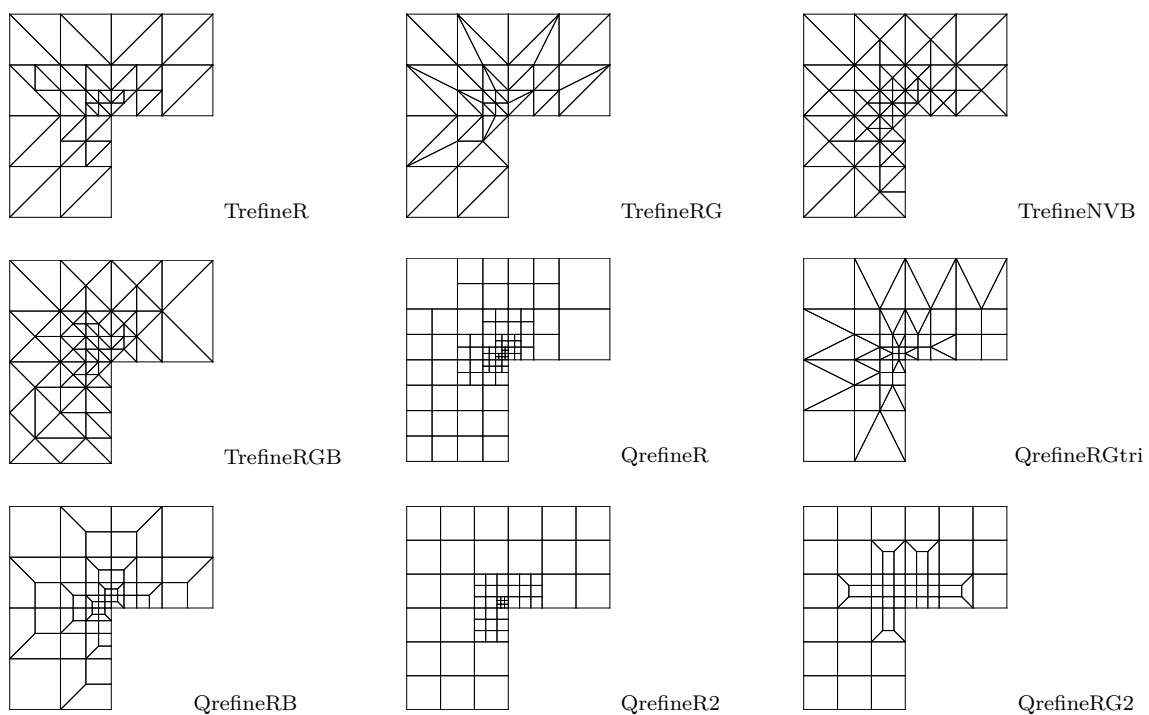


Figure 5.4: Generated meshes through AFEM for Example (5.1). The process is stopped as the number of elements exceeds the maximum number of elements allowed $n_{E_{\max}}=50$.

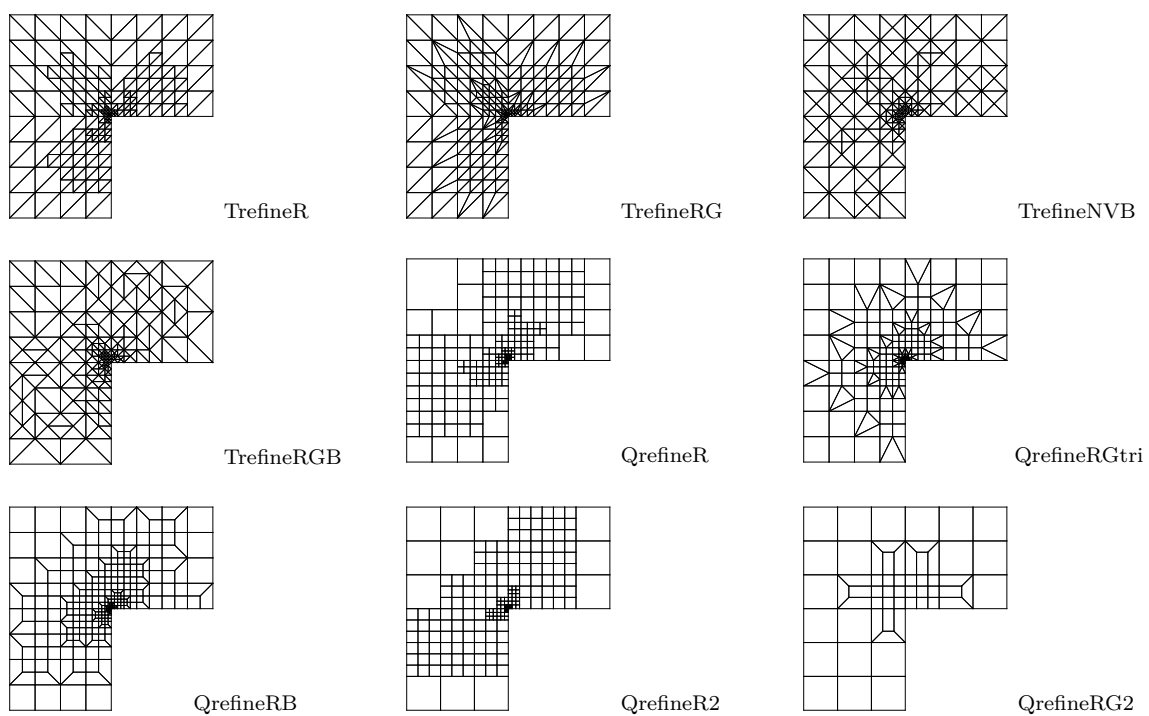


Figure 5.5: Generated meshes through AFEM for Example (5.1). The process is stopped as the number of elements exceeds the maximum number of elements allowed $n_{E_{\max}}=200$.

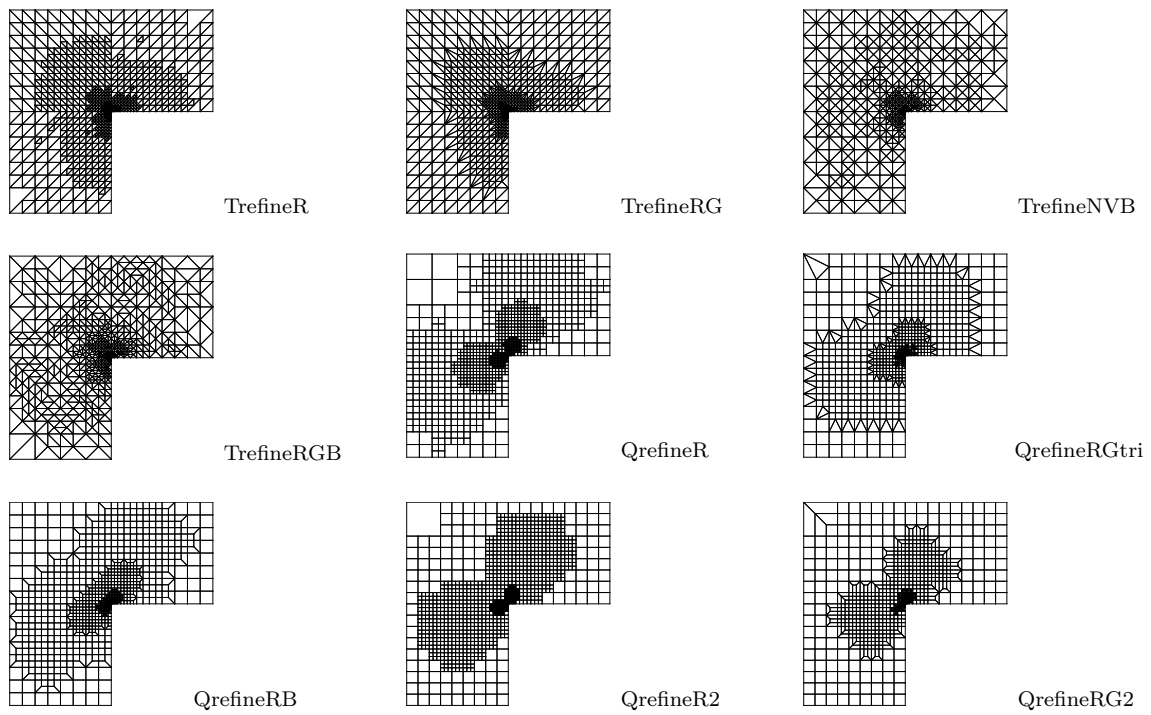


Figure 5.6: Generated meshes through AFEM for Example (5.1). The process is stopped as the number of elements exceeds the maximum number of elements allowed $n_{\text{Emax}}=800$.

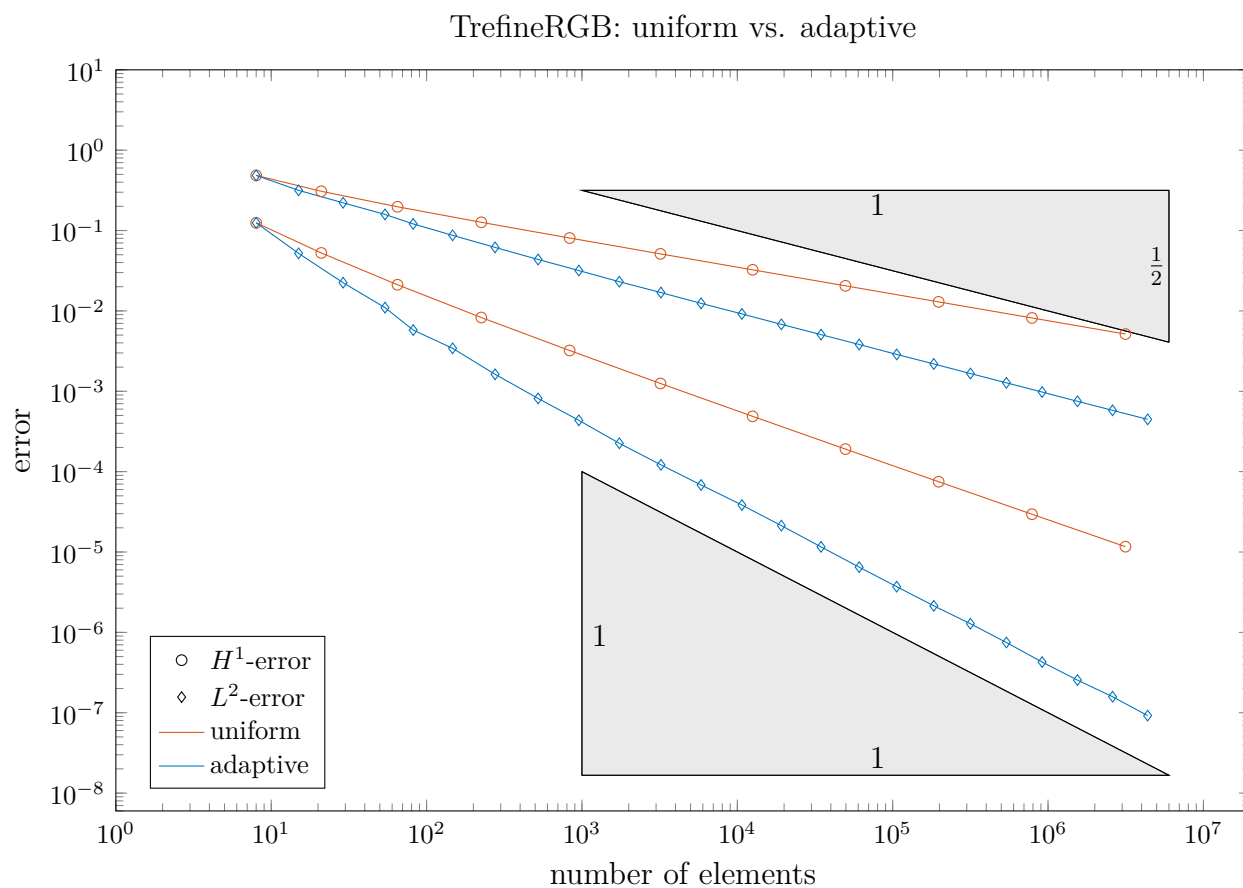


Figure 5.7: H^1 - error and L^2 - error with respect to the number of elements for uniform and adaptive mesh refinement in Example (5.2). As a mesh refinement strategy TrefineRGB is used. The plot shows the superiority of adaptive methods over the standard finite element method since the slopes for uniform mesh refinements are smaller and thus the convergence rate is worse.

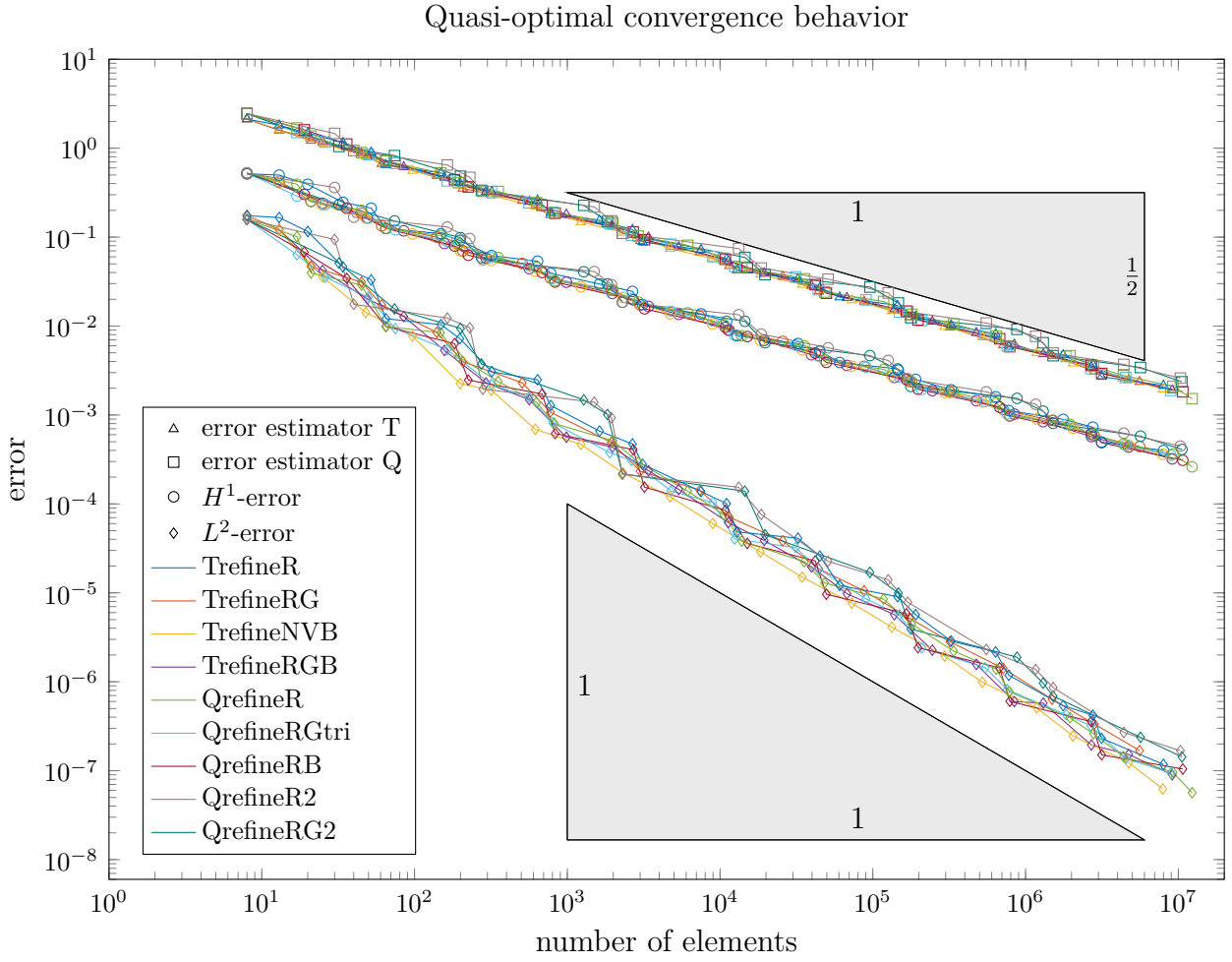


Figure 5.8: Error estimator η , H^1 - error $\|u - U\|_{H^1(\Omega)}$ and L^2 - error $\|u - U\|_{L^2(\Omega)}$ with respect to the number of elements for Example (5.3). We consider the adaptive finite element method with a residual-based error estimator, Dörfler marking and all presented mesh refinement strategies. In comparison to the other examples, the course of the errors are not as straight. However, the quasi-optimal convergence behavior is also obtained for this problem.

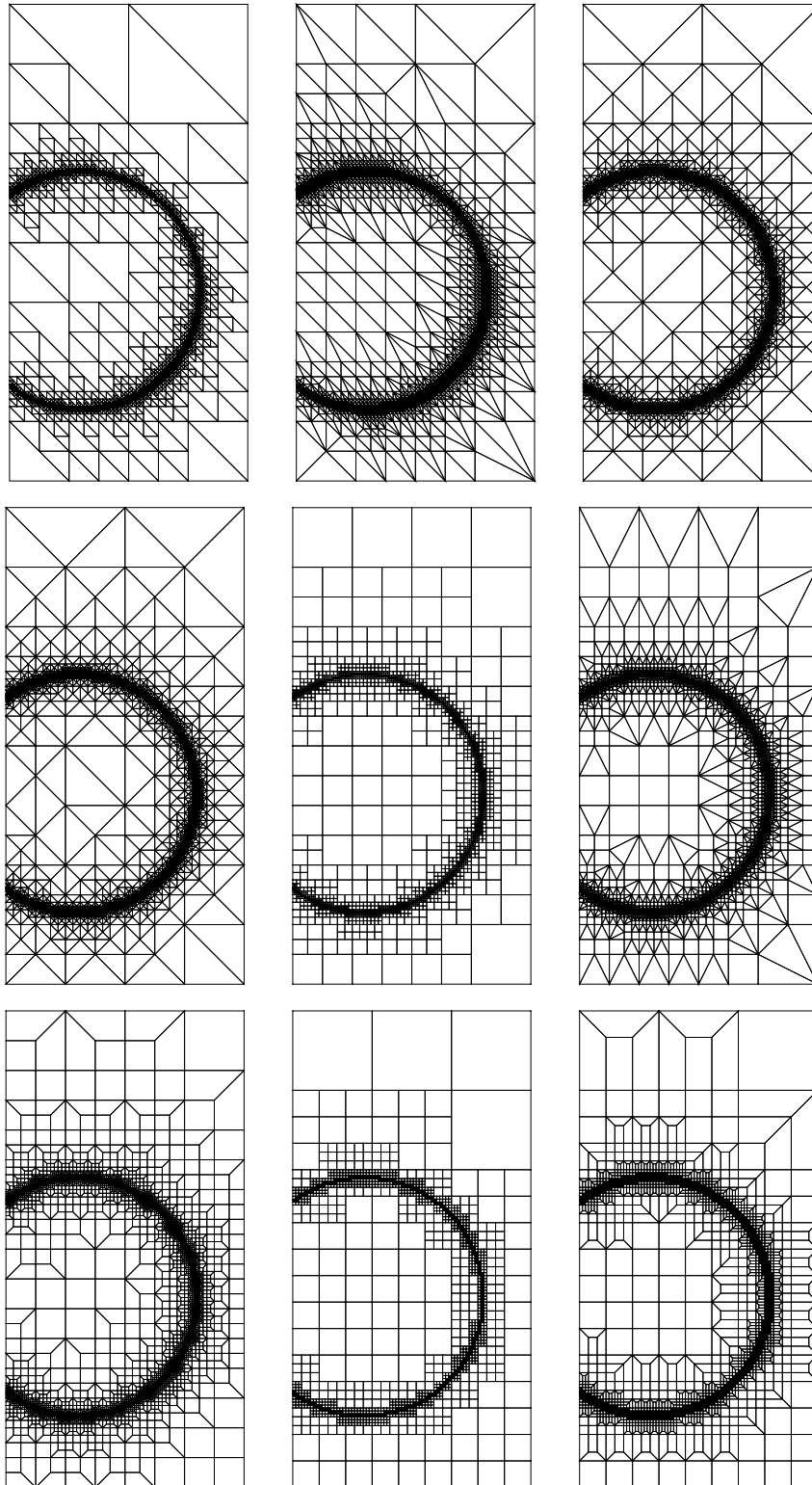


Figure 5.9: Refinement along a circle. The presented refinement strategies can also be used to refine elements along a circle. Elements are marked for refinement if they intersect with a given circle. From top left to bottom right the following refinements are used: TrefineR, TrefineRG, TrefineNVB, TrefineRGB, QrefineR, QrefineRGtri, QrefineRB, QrefineR2, QrefineRG2.

6 Conclusion

In this work, we gave an overview of nine different refinement strategies including triangular, quadrilateral and mixed meshes, and also irregular grids. Furthermore, we managed to provide implementations of these strategies that are easy-to-understand but do not lack efficiency – our examples are run for 10^7 elements in about 2 minutes. To this end, we fulfilled the aim to provide an efficient implementation of mesh refinements, that include triangular and quadrilateral meshes, that is easy to use and adaptable to many purposes and serves the educational purpose of how to implement refinement strategies in an easy but efficient way. We hope that this work may find its way into teaching of Numerical PDEs at Ulm University as it sets a more practical focus on the topic that might be very valuable for our students. For each strategy, theoretical properties are investigated that are mainly important in the context of adaptive finite element methods but may also play a role in other contexts where mesh refinement methods are needed. Furthermore, we put the grid refinement into its proper context in the AFEM such that the reader can understand its role in the whole framework. As discussed, proofs of convergence and rates of convergence for AFEM are, to the best of our knowledge, based on triangular meshes, in particular on the newest vertex bisection. However, our numerical experiments encourage to give more attention to quadrilateral meshes and try to develop the theory of convergence of the adaptive finite element method with quadrilateral meshes.

As future work, we would like to extend the presented refinement implementations into a complete AFEM-package including all four components of the AFEM and a documentation. Further considerable future work is to extend these refinement strategies to three dimensions.

Bibliography

- [1] Gabriel Acosta and Gabriel Monzón. The minimal angle condition for quadrilateral finite elements of arbitrary degree. *Journal of Computational and Applied Mathematics*, 317:218–234, 2017.
- [2] Jochen Albrety, Carsten Carstensen, and Stefan A. Funken. Remarks around 50 lines of Matlab: short finite element implementation. *Numerical Algorithms*, 20(2-3):117–137, 1999.
- [3] Ivo Babuška and A. Kadir Aziz. On the angle condition in the finite element method. *SIAM Journal on Numerical Analysis*, 13(2):214–226, 1976.
- [4] Ivo Babuška and Werner C. Rheinboldt. A-posteriori error estimates for the finite element method. *International Journal for Numerical Methods in Engineering*, 12(10):1597–1615, 1978.
- [5] Ivo Babuška and Werner C. Rheinboldt. Error estimates for adaptive finite element computations. *SIAM Journal on Numerical Analysis*, 15(4):736–754, 1978.
- [6] Ivo Babuška and Michael Vogelius. Feedback and Adaptive Finite Element Solution of One-Dimensional Boundary Value Problems. *Numerische Mathematik*, 44:75–102, 1984.
- [7] Randolph E. Bank and Andrew H. Sherman. An adaptive, multi-level method for elliptic boundary value problems. *Computing*, 26(2):91–105, 1981.
- [8] Randolph E. Bank, Andrew H. Sherman, and Alan Weiser. Some refinement algorithms and data structures for regular local mesh refinement. *Scientific Computing, Applications of Mathematics and Computing to the Physical Sciences*, 1:3–17, 1983.
- [9] Randolph E. Bank and R. Kent Smith. A posteriori error estimates based on hierarchical bases. *SIAM Journal on Numerical Analysis*, 30(4):921–935, 1993.

- [10] Randolph E. Bank and Alan Weiser. Some a posteriori error estimators for elliptic partial differential equations. *Mathematics of computation*, 44(170):283–301, 1985.
- [11] Roland Becker and Rolf Rannacher. An optimal control approach to a posteriori error estimation in finite element methods. *Acta numerica*, 10:1–102, 2001.
- [12] Stefanie Beuter. Regular mesh refinement of quadrilateral meshes. Bachelor thesis, Universität Ulm, 11 2016.
- [13] Peter Binev, Wolfgang Dahmen, and Ron DeVore. Adaptive finite element methods with convergence rates. *Numerische Mathematik*, 97(2):219–268, 2004.
- [14] Dietrich Braess. *Finite Elemente: Theorie, schnelle Löser und Anwendungen in der Elastizitätstheorie*. Springer Spektrum, 5. Überarb. Aufl. edition, 2013.
- [15] Carsten Carstensen. Finite Elemente. *Wissenschaftliches Rechnen*, 11 1999.
- [16] Carsten Carstensen. An adaptive mesh-refining algorithm allowing for an h 1 stable l_2 projection onto courant finite element spaces. *Constructive Approximation*, 20(4):549–564, 2004.
- [17] J. Manuel Cascon, Christian Kreuzer, Ricardo H. Nochetto, and Kunibert G. Siebert. Quasi-optimal convergence rate for an adaptive finite element method. *SIAM Journal on Numerical Analysis*, 46(5):2524–2550, 2008.
- [18] Philippe G. Ciarlet. *The finite element method for elliptic problems*. SIAM, 2002.
- [19] Philippe G. Ciarlet and P.-A. Raviart. Interpolation theory over curved elements, with applications to finite element methods. *Computer Methods in Applied Mechanics and Engineering*, 1(2):217–249, 1972.
- [20] Peter Deuffhard, Peter Leinen, and Harry Yserentant. Concepts of an adaptive hierarchical finite element code. *IMPACT of Computing in Science and Engineering*, 1(1):3–35, 1989.
- [21] Willy Dörfler. A convergent adaptive algorithm for poisson’s equation. *SIAM Journal on Numerical Analysis*, 33(3):1106–1124, 1996.

- [22] Stefan A. Funken, Dirk Praetorius, and Philipp Wissgott. Efficient implementation of adaptive P1-FEM in Matlab. *Computational Methods in Applied Mathematics Comput. Methods Appl. Math.*, 11(4):460–490, 2011.
- [23] Antti Hannukainen, Sergey Korotov, and Michal Křížek. The maximum angle condition is not necessary for convergence of the finite element method. *Numerische Mathematik*, 120(1):79–88, 2012.
- [24] Leif Kobbelt. Interpolatory subdivision on open quadrilateral nets with arbitrary topology. In *Computer Graphics Forum*, volume 15, pages 409–420. Wiley Online Library, 1996.
- [25] Pedro Morin, Ricardo H. Nochetto, and Kunibert G. Siebert. Data oscillation and convergence of adaptive fem. *SIAM Journal on Numerical Analysis*, 38(2):466–488, 2000.
- [26] Pedro Morin, Ricardo H. Nochetto, and Kunibert G. Siebert. Convergence of adaptive finite element methods. *SIAM review*, 44(4):631–658, 2002.
- [27] Francisco Perdomo and Ángel Plaza. Properties of triangulations obtained by the longest-edge bisection. *Central European Journal of Mathematics*, 12(12):1796–1810, 2014.
- [28] Ivo G. Rosenberg and Frank Stenger. A lower bound on the angles of triangles constructed by bisecting the longest side. *Mathematics of Computation*, 29(130):390–395, 1975.
- [29] E.G. Sewell. *Automatic Generation of Triangulations for Piecewise Polynomial Approximation*. Purdue University, 1972.
- [30] Rob Stevenson. Optimality of a standard adaptive finite element method. *Foundations of Computational Mathematics*, 7(2):245–269, 2007.
- [31] Rob Stevenson. The completion of locally refined simplicial partitions created by bisection. *Mathematics of computation*, 77(261):227–241, 2008.
- [32] Rüdiger Verfürth. *A review of a posteriori error estimation and adaptive mesh-refinement techniques*. John Wiley & Sons Inc, 1996.
- [33] Frieda Zames. Surface area and the cylinder area paradox. *The Two-Year College Mathematics Journal*, 8(4):207–211, 1977.

- [34] XuYing Zhao, Jun Hu, and ZhongCi Shi. Convergence analysis of the adaptive finite element method with the red-green refinement. *Science China Mathematics*, 53(2):499–512, 2010.
- [35] Xuying Zhao, Shipeng Mao, and Zhongci Shi. Adaptive finite element methods on quadrilateral meshes without hanging nodes. *SIAM Journal on Scientific Computing*, 32(4):2099–2120, 2010.
- [36] Olgierd C. Zienkiewicz and Jian Z. Zhu. A simple error estimator and adaptive procedure for practical engineerng analysis. *International journal for numerical methods in engineering*, 24(2):337–357, 1987.
- [37] Miloš Zlámal. On the finite element method. *Numerische Mathematik*, 12(5):394–409, 1968.

List of Figures

3.1	Refinement strategies for triangular meshes.	13
3.2	Refinement strategies for quadrilateral meshes.	13
3.3	Red-, green-, blue _ℓ - and blue _r -refinement of a triangle.	14
3.4	Refinement patterns for TrefineR.	15
3.5	A straightforward application of the two refinement patterns in TrefineR.	15
3.6	Resolving the issues of a straightforward application of the two refinement patterns in TrefineR.	15
3.7	Exemplary illustration of TrefineR: refinement of red elements.	17
3.8	Exemplary illustration of TrefineR: refinement of elements with irregular edges.	18
3.9	Refinement patterns for red elements in TrefineRG.	19
3.10	Refinement patterns for green elements in TrefineRG.	19
3.11	A straightforward application of the three refinement patterns in TrefineRG.	19
3.12	Resolving the issues of a straightforward application of the three refinement patterns in TrefineRG.	19
3.13	Exemplary illustration of TrefineRG: refinement of red elements.	21
3.14	Exemplary illustration of TrefineRG: refinement of green elements.	22
3.15	Refinement patterns for TrefineNVB.	24
3.16	Exemplary illustration of TrefineNVB: refinement.	26
3.17	Four similarity classes in TrefineNVB.	27
3.18	Reference edge is longest edge.	30
3.19	Two conceivable choices of reference edges à la Carstensen.	31
3.20	Reference edges à la Stevenson.	32
3.21	Refinement patterns for TrefineRGB.	32
3.22	Exemplary illustration of TrefineRGB: refinement.	34
3.23	Four similarity classes in TrefineRGB.	35

3.24	Red, green of type 1, green of type 2, and green of type 3 refinement rules for a quadrilateral.	36
3.25	Red2, green2 of type 1, green2 of type 2 and green2 of type 3 refinement rules for a quadrilateral.	38
3.26	The 3-Neighbor Rule.	38
3.27	Refinement patterns for QrefineR.	39
3.28	A straightforward application of the two refinement patterns in QrefineR.	39
3.29	Resolving the issues of a straightforward application of the two refinement patterns in QrefineR.	40
3.30	Exemplary illustration of QrefineR: refinement of red elements.	40
3.31	Exemplary illustration of QrefineR: refinement of irregular elements.	41
3.32	Main idea of proving the shape regularity for the red refinement of quadrilaterals.	42
3.33	Refinement patterns for red elements in QrefineRGtri.	43
3.34	Exemplary illustration of QrefineR: refinement of red elements.	45
3.35	Refinement patterns for green elements of type 1 in QrefineRGtri.	45
3.36	Refinement patterns for green elements of type 2 in QrefineRGtri.	46
3.37	Refinement patterns for green elements of type 3 in QrefineRGtri.	46
3.38	Proposed pattern for QrefineRB.	47
3.39	Non-uniqueness of refinement patterns in QrefineRB.	48
3.40	A counterexample for coloring the nodes alternately.	50
3.41	An element that is uniformly refined has nodes from two generations.	50
3.42	Refinement patterns for red elements in QrefineRB.	50
3.43	Refinement patterns for blue elements in QrefineRB.	51
3.44	Exemplary illustration of QrefineRB: refinement of red elements.	53
3.45	Exemplary illustration of QrefineRB: refinement of blue elements.	54
3.46	Refinement patterns for QrefineR2.	55
3.47	Exemplary illustration of QrefineR2: refinement of red elements.	56
3.48	Exemplary illustration of QrefineR2: refinement of irregular elements.	57
3.49	Refinement patterns for red elements in QrefineRG2.	57
3.50	Exemplary illustration of QrefineRG2: refinement of red elements.	59
4.1	Triangulation \mathcal{T} of the L -shaped domain into 6 triangles.	63
4.2	Triangulation \mathcal{T} of the L -shaped domain into 3 quadrilaterals.	64

4.3	Numbering of outer edges for green elements in TrefineRG.	74
4.4	Numbering of the nodes in variable <code>tmp</code> in TrefineRG.	75
4.5	Numbering of outer edges for blue elements in QrefineRB.	79
4.6	Storing <code>blue_r</code> and <code>blue_l</code> elements.	80
4.7	Numbering of the nodes in variable <code>tmp</code> in QrefineRB.	81
4.8	Irregularity data as virtual elements.	86
4.9	Irregularity data for 2-irregular grids.	95
4.10	Orientation of edges in QrefineR2.	97
4.11	Order of newly generated elements after a red2 refinement.	97
4.12	Regularizing an irregular grid and vice versa.	101
4.13	Numbering of nodes in <code>tmp</code> for green elements in <code>regularizeedges_tri</code> .	103
4.14	Numbering of nodes in <code>tmp</code> for green elements in <code>regularizeedges</code> .	111
5.1	Error estimator η and energy error $\ \nabla(u - U)\ _{L^2(\Omega)}$ with respect to the number of elements for Example (5.1).	119
5.2	Error estimator η , H^1 - error $\ u - U\ _{H^1(\Omega)}$ and L^2 - error $\ u - U\ _{L^2(\Omega)}$ with respect to the number of elements for Example (5.2).	121
5.3	Computational times for different refinement strategies for Example (5.2) with respect to the number of elements.	123
5.4	Generated meshes through AFEM for Example (5.2).	124
5.5	Generated meshes through AFEM for Example (5.2).	124
5.6	Generated meshes through AFEM for Example (5.2).	125
5.7	H^1 - error $\ u - U\ _{H^1(\Omega)}$ and L^2 - error $\ u - U\ _{L^2(\Omega)}$ with respect to the number of elements for uniform and adaptive mesh refinement in Example (5.2).	126
5.8	Error estimator η , H^1 - error $\ u - U\ _{H^1(\Omega)}$ and L^2 - error $\ u - U\ _{L^2(\Omega)}$ with respect to the number of elements for Example (5.3).	127
5.9	Refinement along a circle.	128

List of Tables

4.1	Classification of refinement strategies into reasonable implementation approaches.	61
4.2	Marking a triangle.	66
4.3	Mapping of eight possible markings to the five patterns allowed in TrefineNVB.	66

Listings

4.1	provideGeometricData	65
4.2	hash2map	66
4.3	TrefineNVB	68
4.4	TrefineRGB: Lines 69-71	72
4.5	TrefineRG	75
4.6	QrefineRB	81
4.7	TrefineR	88
4.8	QrefineR	92
4.9	QrefineR2	96
4.10	QrefineRGtri - regularizedges_tri	103
4.11	QrefineRGtri - recoarseedges_tri	107
4.12	QrefineRG2 - regularizeedges	111
4.13	QrefineRG2 - recoarseedges	115

Declaration of Originality

I hereby declare that this thesis was performed and written on my own and that references and resources used within this work have been explicitly indicated.

I am aware that making a false declaration may have serious consequences.

Ulm, April 26th, 2018

Anja Schmidt