The Standard Finite Element Method and the Weak Galerkin Method for Elliptic Boundary Value Problems

Senior Thesis

written by

Mark Taylor

supervised by

Prof. Qilong Zhai

Department of Mathematics

Jilin University

June 4, 2021

Abstract

This thesis introduces the bare-bones of finite element methods (FEM) for second-order elliptic boundary value problems and presents a straightforward implementation in MATLAB for the standard FEM. Then We give some conclusions about error estimates and the rate of convergence. In the end, we provide a cutting-edge technique—the weak Galerkin method as a comparison to the standard (Galerkin) FEM. The accuracy of approximation by these two methods will be analyzed experimentally through a couple of numerical experiments.

Acknowledgment

First, the author would like to thank Prof. Ralf Hiptmair who wrote (and is actively updating) this MARVELOUS FEM (NPDE actually) book. It is of extremely high quality, comprising a tremendous number of beautiful figures which makes the book incredibly accessible and interesting.

Then the author want to thank Prof. Qilong Zhai who has provided great suggestions in the writing and for supervising the author with great patience and responsibility. This thesis couldn't be done without his help.

In addition, the author is extremely grateful to Miss White for discussing the quadratic finite elements for the two-point boundary value problems with him last term, which underlay an important understanding of higher order finite elements to the author. Also, the author is very appreciative of the practical help from Shuaishuai Lu in implementing FEM.

Lastly, the author want to give thanks to my awesome roomies who have been there for me sharing joys and sorrows with me in these years, to all the adorable classmates and cute teachers who have made our college life this lovely, and most of all huge thanks to my beloved parents who have always been so supportive.

Contents

List of Algorithms

0 Introduction

This thesis consists of two parts: Section 1–5 is the first part introducing the standard Galerkin FEM, and Section 6 is the second part introducing the weak Galerkin FEM.

In the first part, we extract and summarize the essentials (weak formulation, discretization, and implementation details) of finite element methods in the online book *Numerical Methods for Partial Differential Equations* [Hip21] from ETH Lecture 401-0674-00L by Prof. Ralf Hiptmair. We will also add some our own understanding in it, of course. We aim to make it as accessible as possible. So We present this part in a way such that one can learn by analogy. This is made easy by first examining the one-dimensional two-point boundary value problems. Then by analogy one can smoothly proceed to the two-dimensional elliptic boundary value problem. It allows one to learn the basic concepts of FEM with relative ease. By learning this five sections one can implement a small framework for numerically solving some elliptic boundary values problems in the standard Galerkin FEM. We also provide a minimum implementation in MATLAB, the code of which can be found at `https://github.com/How-u-doing/Numerical_Analysis/tree/master/Chapter11_FEM2d`. Besides, the 1D counterpart involved in Section 5.3 is available at `https://github.com/How-u-doing/Numerical_Analysis/tree/master/Chapter10_BVPforODEs`.

We start our journey in Section 1 by converting boundary value problems into their equivalent weak formulations. In the second part, Section 2, we provide the discretization of boundary value problems, which is only one, but a critical step in numerical treatment of boundary value problems. This step assembles and builds the Galerkin matrix and right-hand side vector, which are the two final components in finite element methods that will then be used to solve the linear systems of equations in order to obtain the numerical solution. Also, the assembly of mass matrix and the imposition of natural and essential boundary conditions will be discussed. Subsequently, by convention, we "analyze" the accuracy and convergence of this numerical method (FEM) in Section 3. The fourth segment in Section 4 contains some implementation details in the context of MATLAB, including mesh generation, index mapping, local computations of element matrices and element vectors, incorporation of boundary conditions, etc. To the end of the first part, we present two numerical experiments of 2nd-order elliptic boundary value problems in Section 5. We will solve those two BVPs using the algorithms mentioned in Section 2 and Section 4. Additionally, examples of two-point BVPs that employ both linear finite elements and quadratic finite elements are also provided for the sake of comparison and completeness.

In the end, in the second part as a further reading, we provide and discuss a newly developed novel technique, the weak Galerkin finite element method. The introduction of the weak gradients makes it powerful enough to handle non-continuous piecewise polynomial across cells. We will see how it differs from the standard Galerkin FEM in terms of approximation quality in Section 6.4.

1 Weak Formulation

A finite element method is characterized by a variational formulation, a discretization strategy, one or more solution algorithms, and post-processing procedures. In this section, we will transform our mathematical models (boundary value problems) into their corresponding weak formulations. Let us first look at two model problems that will form the basis for this thesis.

P1: A one-dimensional two-point boundary value problem with homogeneous Dirichlet boundary conditions

$$-\frac{\mathrm{d}}{\mathrm{d}x}\left(p(x)\frac{\mathrm{d}u}{\mathrm{d}x}\right)+q(x)u=f(x),\quad x\in(a,b), \tag{1}$$

$$u(a)=0,\ u(b)=0, \tag{2}$$

where $p(x)\in C^1(\bar{I}),\ p(x)\geq p_{min}>0,\ q(x)\in C^1(\bar{I}),\ q(x)\geq 0,\ f(x)\in L^2(I),\ I=]a,b[$.[1]

P2: A second-order elliptic homogeneous Neumann problem

$$-\nabla\cdot(\boldsymbol{\alpha}(\boldsymbol{x})\nabla u)+\gamma(\boldsymbol{x})u=f,\quad \text{in } \Omega, \tag{3}$$

$$\boldsymbol{\alpha}(\boldsymbol{x})\nabla u\cdot\boldsymbol{n}=0,\quad \text{on } \partial\Omega, \tag{4}$$

where Ω is an open bounded domain in $\mathbb{R}^2$, ∇u denotes the gradient of the function u, $\boldsymbol{\alpha}$ is a symmetric uniformly positive definite[2] 2×2 matrix-valued function, γ is a scalar-valued function, and $\boldsymbol{n}$ is the outward unit vector normal to the boundary $\partial\Omega$.

For differential equation (1) in P1, multiplying a test function $v(x)$ on both sides and applying integration by parts we have

$$\begin{aligned}\int_a^b\left[-\frac{\mathrm{d}}{\mathrm{d}x}\left(p(x)\frac{\mathrm{d}u}{\mathrm{d}x}\right)v+q(x)uv\right]\mathrm{d}x&=\int_a^b\left(p(x)\frac{\mathrm{d}u}{\mathrm{d}x}\frac{\mathrm{d}v}{\mathrm{d}x}+q(x)uv\right)\mathrm{d}x-p(x)\frac{\mathrm{d}u}{\mathrm{d}x}v(x)|_a^b\\&=\int_a^b fv\,\mathrm{d}x.\end{aligned} \tag{5}$$

Now we choose the test space $V=H_0^1:=\{u\in H^1(I),[3]\ u=0 \text{ at } x=a \text{ and } x=b\}$ and $v(x)\in V$, then the second term on the right-hand side of (5) vanishes since $v(a)=v(b)=0$. Hence,

$$\int_a^b\left(p(x)\frac{\mathrm{d}u}{\mathrm{d}x}\frac{\mathrm{d}v}{\mathrm{d}x}+q(x)uv\right)\mathrm{d}x=\int_a^b fv\,\mathrm{d}x\quad\forall v\in V. \tag{6}$$

Using space of trial functions $U=V$ we can rephrase problem (6) in the following *weak* form:

$$\begin{cases}\text{Find } u\in U \text{ such that}\\ a(u,v)=\ell(v)\quad\forall v\in V.\end{cases} \tag{7}$$

Here $a:U\times V\mapsto\mathbb{R}$ is a continuous bilinear form and $\ell:V\mapsto\mathbb{R}$ is a continuous linear form, with

$$a(u,v)=\int_a^b\left(p(x)\frac{\mathrm{d}u}{\mathrm{d}x}\frac{\mathrm{d}v}{\mathrm{d}x}+q(x)uv\right)\mathrm{d}x,\quad \ell(v)=\int_a^b fv\,\mathrm{d}x. \tag{8}$$

[1] The notation $]a,b[$ is used for an open interval, more commonly written as (a,b). But the latter can be confused with a vector or an element of $\mathbb{R}^2$.

[2] A matrix-valued function $\mathbf{A}:\Omega\mapsto\mathbb{R}^{n,n}, n\in\mathbb{N}$, is called uniformly positive definite, if

$$\exists\epsilon^->0:\quad \boldsymbol{z}^\top\mathbf{A}(\boldsymbol{x})\boldsymbol{z}\geq\epsilon^-\|\boldsymbol{z}\|\quad\forall\boldsymbol{z}\in\mathbb{R}^n$$

for *almost all* $\boldsymbol{x}\in\Omega$, that is, only with the exception of a set of volume zero.

[3] $H^1(\Omega)$ is a the Sobolev space of functions on Ω with generalized derivatives $L^2(\Omega)$, i.e.

$$H^1(\Omega):=\{v:\Omega\mapsto\mathbb{R}\text{ integrable}:\ \int_\Omega|\nabla v(\boldsymbol{x})|^2\mathrm{d}\boldsymbol{x}<\infty\}$$

Problem (7) is also referred to as a *variational problem* and as the *variational form* of (1, 2). The reasons for using the terms "variational" and "weak" to describe (7) are explained in [Goc06].

The standard finite element approximation of (6) amounts to the construction of Galerkin or weighted residual approximations of (1, 2) over *finite dimensional* subspaces $U_h \subset U$ and $V_h \subset V$.[4] That is,

$$\begin{cases} \text{Find } u_h \in U_h \text{ such that} \\ a(u_h, v_h) = \ell(v) \quad \forall v_h \in V_h, \end{cases} \tag{9}$$

where

$$a(u_h, v_h) = \int_a^b \left(p(x) \frac{\mathrm{d}u_h}{\mathrm{d}x} \frac{\mathrm{d}v_h}{\mathrm{d}x} + q(x) u_h v_h \right) \mathrm{d}x, \quad \ell(v) = \int_a^b f v_h \, \mathrm{d}x. \tag{10}$$

Similarly, for partial differential equation (3) in P2 we integrate by parts via Green's theorem noticing the boundary condition (4):

$$\int_\Omega \left(-\nabla \cdot (\boldsymbol{\alpha}(\boldsymbol{x}) \nabla u)\, v + \gamma(\boldsymbol{x}) u v\right) \mathrm{d}\boldsymbol{x} = \int_\Omega \left(\boldsymbol{\alpha}(\boldsymbol{x}) \nabla u \cdot \nabla v + \gamma(\boldsymbol{x}) u v\right) \mathrm{d}\boldsymbol{x} - \int_{\partial\Omega} \boldsymbol{\alpha}(\boldsymbol{x}) \nabla u \cdot \boldsymbol{n}\, v \, \mathrm{d}s = \int_\Omega f v \, \mathrm{d}\boldsymbol{x},$$

$$\Downarrow$$

$$\int_\Omega \left(\boldsymbol{\alpha}(\boldsymbol{x}) \nabla u \cdot \nabla v + \gamma(\boldsymbol{x}) u v\right) \mathrm{d}\boldsymbol{x} = \int_\Omega f v \, \mathrm{d}\boldsymbol{x} \quad \forall v \in V, \tag{11}$$

where $V = H^1(\Omega)$.

Likewise, using $U = V = H^1(\Omega)$, $U_h \subset U$, $V_h \subset V$ we can obtain exactly the same *discrete weak form* as that of (9), but with $a(u_h, v_h)$ and $\ell(v_h)$ being

$$a(u_h, v_h) = \int_\Omega \left(\boldsymbol{\alpha}(\boldsymbol{x}) \nabla u_h \cdot \nabla v_h + \gamma(\boldsymbol{x}) u_h v_h\right) \mathrm{d}\boldsymbol{x}, \quad \ell(v_h) = \int_\Omega f v_h \, \mathrm{d}\boldsymbol{x}. \tag{12}$$

(9, 10) and (9, 12) are the *discrete weak formulations* based on which we will develop our numerical solutions in later sections.

2 Discretization

In previous section, we have derived the continuous and discrete weak formulations equivalent to the boundary value problems P1, P2. Reducing the continuous infinite dimensional problems to a finite dimensional vector subspace allows us to numerically compute u_h as a finite linear combination of the basis vectors in U_h. In this section we will dive into the details of Galerkin discretization, which consist of three parts: choice of trial/test space, choice of basis, computing Galerkin matrices and right-hand side vectors.

2.1 Choices of Trial/Test Space and Basis

We adopt Galerkin method throughout this thesis so we have $U_h = V_h$ (in general, $U_h \neq V_h$, though). Let $\mathfrak{B}_h = \{b_h^1, b_h^2, ..., b_h^N\}$ be an ordered basis of U_h, then u_h, v_h can be written as follows

$$u_h = \mu_1 b_h^1 + \mu_2 b_h^2 + ... + \mu_N b_h^N, \quad \mu_i \in \mathbb{R}, \tag{13}$$

$$v_h = \nu_1 b_h^1 + \nu_2 b_h^2 + ... + \nu_N b_h^N, \quad \nu_i \in \mathbb{R}. \tag{14}$$

The number $N \in \mathbb{R}$ above is the dimension of the discrete trial and test space U_h, i.e. $N := \dim U_h$.

Inserting (13) and (14) into $a(u_h, v_h)$ and $\ell(v_h)$ and considering the linearity of $a(\cdot, \cdot)$ and $\ell(\cdot)$ give the following chain of equivalent statements:

$$u_h \in U_h : \quad a(u_h, v_h) = \ell(v_h) \quad \forall v_h \in U_h. \tag{15}$$

[4] The subscript h is a discretization parameter generally chosen as a measure of the mesh size.

$$\Leftrightarrow \quad \sum_{k=1}^{N}\sum_{j=1}^{N}\mu_k\nu_j a(b_h^k, b_h^j) = \sum_{j=1}^{N}\nu_j \ell(b_h^j) \quad \forall \nu_1, ..., \nu_N \in \mathbb{R}, \tag{16}$$

$$\Leftrightarrow \quad \sum_{j=1}^{N}\nu_j \left(\sum_{k=1}^{N}\mu_k a(b_h^k, b_h^j) - \ell(b_h^j)\right) = 0 \quad \forall \nu_1, ..., \nu_N \in \mathbb{R}, \tag{17}$$

$$\Leftrightarrow \quad \sum_{k=1}^{N}\mu_k a(b_h^k, b_h^j) = \ell(b_h^j) \quad \text{for } j = 1, ..., N. \tag{18}$$

$$\Leftrightarrow \quad \mathbf{A}\vec{\boldsymbol{\mu}} = \vec{\boldsymbol{\varphi}}, \tag{19}$$

$$\text{where} \quad \mathbf{A} = \left[a(b_h^k, b_h^j)\right]_{j,k=1}^{N} \in \mathbb{R}^{N,N}, \ \vec{\boldsymbol{\mu}} = (\mu_1, ..., \mu_N)^\top \in \mathbb{R}^N, \ \vec{\boldsymbol{\varphi}} = \left[\ell(b_h^j)\right]_{j=1}^{N}. \tag{20}$$

We can see that problem (15) is finally converted to a linear system of equations $\mathbf{A}\vec{\boldsymbol{\mu}} = \vec{\boldsymbol{\varphi}}$ which then can be solved by a computer by applying one or more solution algorithms that have been very well established.

We call $\mathbf{A} = \left[a(b_h^k, b_h^j)\right]_{j,k=1}^{N} \in \mathbb{R}^{N,N}$, $\vec{\boldsymbol{\varphi}} = \left[\ell(b_h^j)\right]_{j=1}^{N}$, $\vec{\boldsymbol{\mu}} = (\mu_1, ..., \mu_N)^\top \in \mathbb{R}^N$, and $u_h = \sum_{k=1}^{N}\mu_k b_h^k$ the Galerkin matrix, right-hand side vector, coefficient vector, and recovery of solution, respectively. For legacy reasons, Galerkin matrix, right-hand side vector, and Galerkin matrix for $(u, v) \mapsto \int_\Omega uv \, \mathrm{d}\boldsymbol{x}$ are also referred to as stiffness matrix, load vector, and mass matrix, respectively. These came from the field of solid mechanics (linear elasticity) to which finite element methods were primarily applied in the early stage (late 60s and early 70s). Besides, the term *degree of freedom* (d.o.f./DOF) is also frequently used in finite element methods. It has double meanings: ❶ either a single component of the basis expansion vector $\boldsymbol{\mu}$ for the Galerkin solution, ❷ or a basis function $b_h^i \in \mathfrak{B}_h$.

For the sake of function approximation the finite element method chooses polynomials. In specific, the finite element method is based on approximation by *continuous, piecewise polynomials*, where "piecewise" is to be understood with respect to a *partition* of the computational domain Ω. Here $\Omega =]a, b[$ for the 1D problem P1 and $\Omega =$ "an open bounded domain $\subset \mathbb{R}^2$" for the 2D problem P2.

Now let us take a look at the *elements* that constitute the domain Ω in 1D and 2D respectively. By the way, the name "finite element" out the "finite element method" comes from dividing the problem domain into a finite number of geometry shapes, in particular, intervals in 1D, triangles or quadrilaterals in 2D, tetrahedrons or hexahedra in 3D, etc. See Figure 1 below.

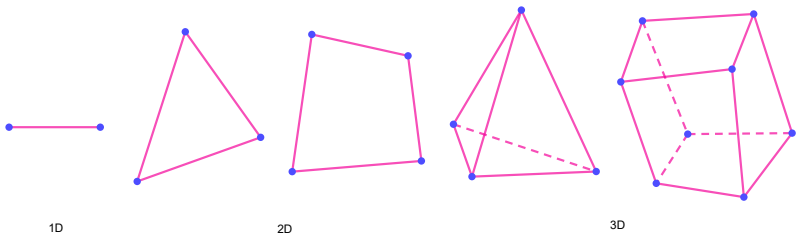


Figure 1: Types of finite elements

2.1.1 Meshes (Grids) in 1D: Intervals

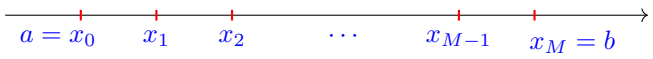


Figure 2: A partition on interval $[a, b]$

When talking about piecewise polynomials one has to fix a partitioning of the domain $]a, b[$ first. Therefore we equip $\Omega = [a, b]$ with $M + 1$ nodes ($M \in \mathbb{N}$) forming the set (see Figure 2)

$$\mathcal{V}(\mathcal{M}) := \{a = x_0 < x_1 < ... < x_{M-1} < x_M = b\}.$$

The nodes define small intervals that constitute a mesh/grid

$$\mathcal{M} := \{]x_{j-1}, x_j[:\ 1 \le j \le M\}.$$

The intervals $[x_{j-1}, x_j]$, $j = 1, ..., M$ are the *cells* of the mesh $\mathcal{M}$, which is often identified with the set of its cells. A special case is an equidistant mesh with uniformly spaced nodes:

$$x_j = a + jh, \quad h = \frac{b-a}{M}.$$

The local and global resolution of a mesh/grid is measured through two quantities, the

$$\begin{array}{ll} \text{(local) cell size} & h_j := |x_j - x_{j-1}|,\ j = 1, ..., M \\ \text{(global) meshwidth} & h_{\mathcal{M}} := \max_j |x_j - x_{j-1}|. \end{array}$$

2.1.2 Meshes in 2D: Triangulations

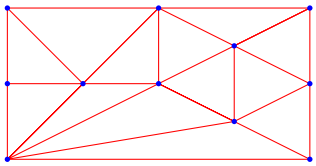


Figure 3: Triangular mesh in 2D

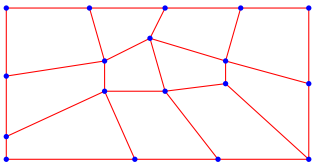


Figure 4: Quadrilateral mesh in 2D

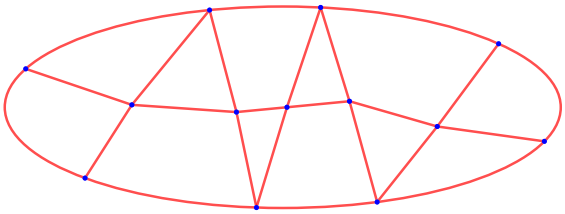


Figure 5: A 2D *hybrid* mesh comprising triangles, quadrilaterals, and curvilinear cells (at $\partial\Omega$)

While in 1D splitting the interval into disjoint sub-intervals is about the only meaningful option to define a partition, we have many more possibilities in higher dimensions, for example, triangular mesh (Figure 3), quadrilateral mesh (Figure 4), or hybrid mesh (Figure 5), etc. Here we opt for triangulations, which are the most common meshes in two dimensions. The definition of finite element triangulation and some common parlance we use are given below:

Definition 2.1 (Triangulation)**.** A triangulation $\mathcal{M}$ of Ω satisfies

(i) $\mathcal{M} = \{K_i\}_{i=1}^M$, $M \in \mathbb{R}$, $K_i :=$ open triangle

(ii) disjoint interiors: $i \neq j \Rightarrow K_i \cap K_j = \varnothing$

(iii) tiling/partition property: $\bigcup_{i=1}^{M} \overline{K}_i = \overline{\Omega}$

(iv) intersection $\overline{K}_i \cap \overline{K}_j$, $i \neq j$ is
- either $\varnothing$,
- or an edge of both triangles,
- or a vertex of both triangles.

Common parlance:	vertices of triangles	=	nodes of mesh	=	set $\mathcal{V}(\mathcal{M})$
	triangles of the mesh	=	cells or elements of mesh	=	set $\mathcal{M}$

2.1.3 Space and Basis in 1D

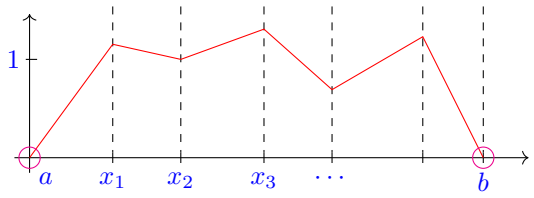


Figure 6: $\Uparrow$ a function $\in \mathcal{S}^0_{1,0}(\mathcal{M})$

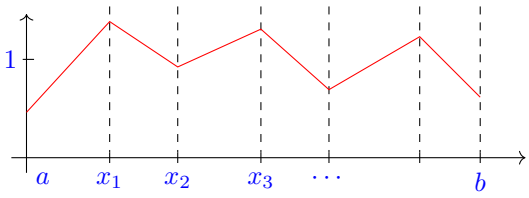


Figure 7: $\Uparrow$ a function $\in \mathcal{S}^0_1(\mathcal{M})$

For the 1D problem P1, we consider the simplest space of continuous, M-piecewise polynomial functions in $H^1_0(]a,b[)$:

$$U_h = \mathcal{S}^0_{1,0}(\mathcal{M}) := \left\{ \begin{array}{l} v \in C^0[a,b] : v_{|[x_{i-1},x_i]} \text{ linear}, \\ i = 1,...,M,\ v(a) = v(b) = 0 \end{array} \right\}, \tag{21}$$

$$N := \dim \mathcal{S}^0_{1,0}(\mathcal{M}) = M - 1.$$

In above notation, the symbol $\mathcal{S}$ in $\mathcal{S}^0_{1,0}$ comes form the fact that the space is comprised of scalar functions, the superscript 0 in $\mathcal{S}^0_{1,0}$ is owing to $C^0[a,b]$ (continuous functions), the subscript 1 in $\mathcal{S}^0_{1,0}$ is due to the linearity (locally 1st degree polynomials), and the subscript 0 in $\mathcal{S}^0_{1,0}$ is because of being zero at the (homogeneous) boundary (see Figure 6). For a more general two-point Dirichlet problem, we can define (see Figure 7)

$$\mathcal{S}^0_1(\mathcal{M}) := \left\{ v \in C^0[a,b] : v_{|[x_{i-1},x_i]} \text{ linear}, \quad \forall i = 1,...,M \right\}. \tag{22}$$

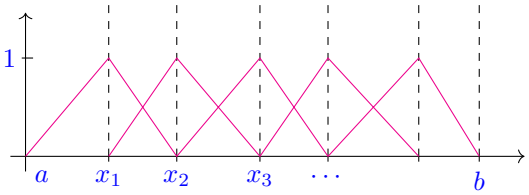


Figure 8: 1D tent functions in $\mathcal{S}^0_{1,0}(\mathcal{M})$

Our choice of the ordered basis $\mathfrak{B}_h = \{b^1_h, ..., b^{M-1}_h\}$ of U_h is the 1D tent functions (see Figure 8):

$$b^j_h(x) := \begin{cases} (x - x_{j-1})/h_j, & \text{if } x_{j-1} \le x \le x_j, \\ (x_{j+1} - x)/h_{j+1}, & \text{if } x_j \le x \le x_{j+1}, \\ 0, & \text{elsewhere.} \end{cases} \tag{23}$$

$$b^j_h(x_i) = \delta(x) := \begin{cases} 1, & \text{if } i = j, \\ 0, & \text{if } i \ne j. \end{cases} \tag{24}$$

In (13), letting x be the interior nodes of the mesh ($x = x_j$, $j = 1, ..., M-1$) and noting (24) we have:

$$u_h \in \mathcal{S}^0_{1,0}(\mathcal{M}) \quad \Leftrightarrow \quad u_h = \sum_{i=1}^{M-1} u_h(x_i) b^i_h. \tag{25}$$

2.1.4 Space and Basis in 2D

Linear Finite Element Space

Now we examine the two-dimensional linear finite element space as well as the basis in 2D. Our first objective is to we generalize the space $\mathcal{S}_1^0(\mathcal{M})$ as defined in (22) to 2D. To do so we first extend he concept of (affine) linear scalar-valued functions. Table 1 below exhibits the natural correspondence of concepts in 1D and 2D. Then we define $\mathcal{S}_1^0(\mathcal{M})$ over a triangular mesh in 2D in the same fashion as we defined it in 1D over a partition of an interval.

	$d=1$	$d=2$
Grid/mesh cells:	intervals $]x_{i-1}, x_i[,\ i=1,...,M$	triangles $K_i,\ i=1,...,M$
Linear functions:	$x \in \mathbb{R} \mapsto \alpha + \beta \cdot x,\ \alpha, \beta \in \mathbb{R}$	$\boldsymbol{x} \in \mathbb{R}^2 \mapsto \alpha + \boldsymbol{\beta} \cdot \boldsymbol{x},\ \alpha \in \mathbb{R}, \boldsymbol{\beta} \in \mathbb{R}^2$

Table 1: Affine linear functions in 1D and 2D

This suggests that we try a definition analogous to the 1D case (22) (see Figure 9):

$$U_h = \mathcal{S}_1^0(\mathcal{M}) := \left\{ v \in C^0(\overline{\Omega}) : \forall K \in \mathcal{M} : \begin{array}{l} v_{|K}(\boldsymbol{x}) = \alpha_K + \boldsymbol{\beta}_K \cdot \boldsymbol{x}, \\ \alpha_K \in \mathbb{R}, \boldsymbol{\beta}_K \in \mathbb{R}^2, \boldsymbol{x} \in K \end{array} \right\} \subset H^1(\Omega). \tag{26}$$

The proof of the subset relationship above between $\mathcal{S}_1^0(\mathcal{M})$ and $H^1(\Omega)$ involves the notion of *weak derivatives* and thus will not be discussed here but an intuitive theorem alongside a graph is given in the book [Hip21] in 1.3.4.22. The theorem implies that a function that is piecewise (with regard to a "nice" partition of Ω) smooth and bounded belongs to $H^1(\Omega)$ if and only if it is continuous on the entire domain Ω, which accounts for the requirement of $C^0(\overline{\Omega})$ in the above definition.

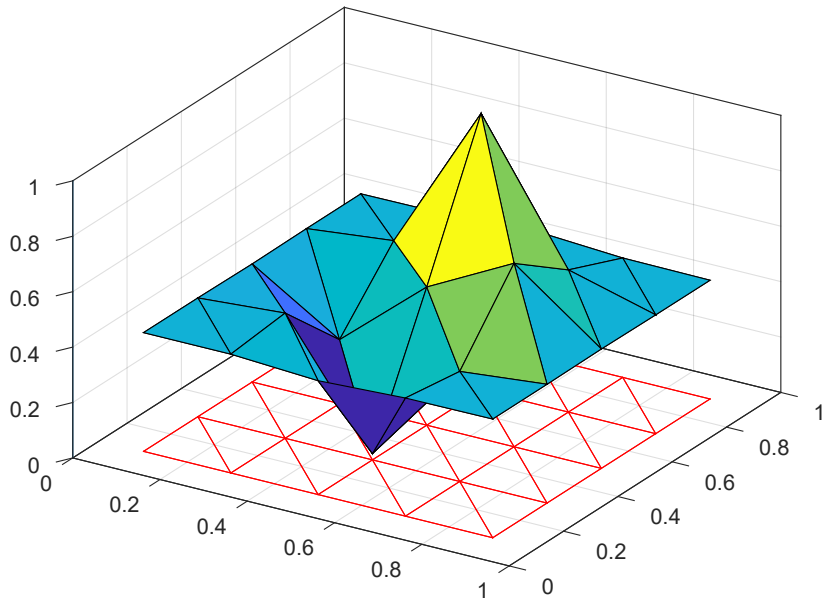


Figure 9: ⇑ a continuous piecewise affine linear function $\in \mathcal{S}_1^0(\mathcal{M})$ on a triangular mesh $\mathcal{M}$

It can be seen that for $u_h \in \mathcal{S}_1^0(\mathcal{M})$ the gradient $\mathbf{grad}\, u_h$ can be computed on each triangle as piecewise constant function, i.e.

$$\text{on } K \in \mathcal{M} : \ \mathbf{grad}(\alpha_K + \boldsymbol{\beta}_K \cdot \boldsymbol{x}) = \boldsymbol{\beta}_K. \tag{27}$$

Nodal Basis Functions

Our next goal is the generalization of "tent functions", see (23) and Figure 8. In 1D, adding two more "half-tent" functions (see the two green parts in Figure 10), cardinal basis functions belonging to the end points x_0 and x_M, we obtain a basis of $\mathcal{S}_1^0(\mathcal{M})$:

$$\mathfrak{B} = \{b_h^0, ..., b_h^M\}, \tag{28}$$

$$b_h^j(x_i) = \delta(x) := \begin{cases} 1, & \text{if } i = j, \\ 0, & \text{if } i \neq j. \end{cases} \tag{29}$$

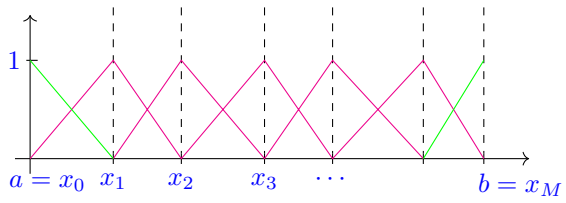


Figure 10: 1D tent functions in $\mathcal{S}_1^0(\mathcal{M})$

Actually, the "nodal (value) property" condition (29) already defines a tent function in the space $\mathcal{S}_1^0(\mathcal{M})$. This approach is directly carried over to 2D: for any node $\boldsymbol{x} \in \mathcal{V}(\mathcal{M})$ we let it have height 1, then together with all its adjacent nodes they form a tent shape function (partial tent shape on the boundary $\partial\Omega$), see Figure 11 for an illustration.

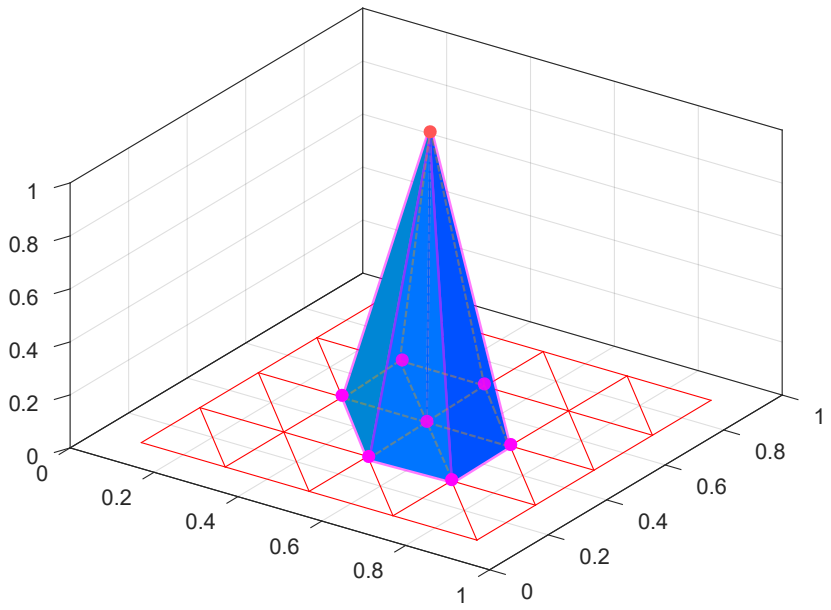


Figure 11: A (global) piecewise linear nodal basis function on a triangular mesh $\mathcal{M}$

From the above picture we can see that the basis function b_h^i can be viewed as the intersection of the xOy plane and several slanted planes of these local tetrahedrons on the adjacent triangles to vertex x_i. Figure 12 shown on the next page is one of the six local nodal basis functions that form the global piecewise linear nodal basis function in Figure 11. Further more, from a triangle's perspective, if we look at each triangle on the mesh $\mathcal{M}$, each of the 3 vertices can serve as the *pivot* so that the slanted plane of this local tetrahedron constitutes one part of the global tent function that is based on the *pivot* (vertex).

Thus, we can define the 2D counterpart of the 1D tent function basis by "nodal conditions" in the following fashion:

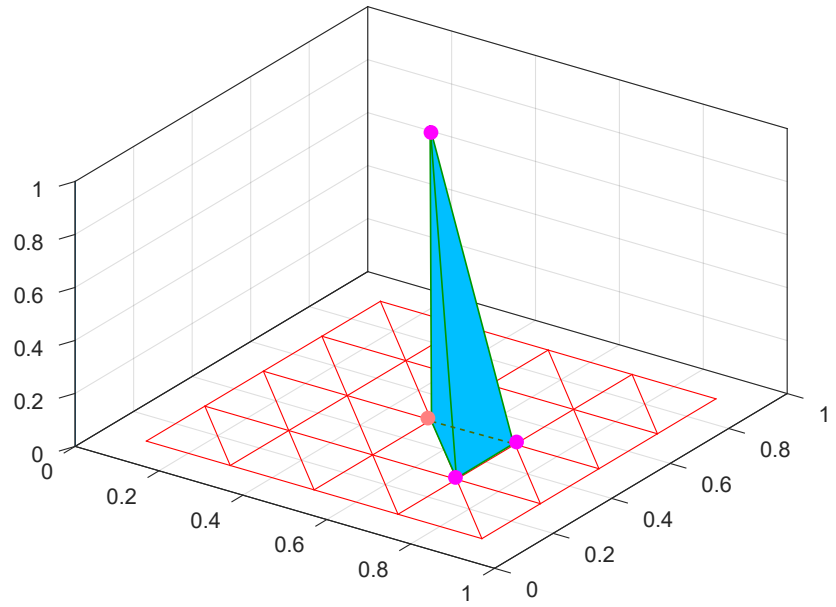
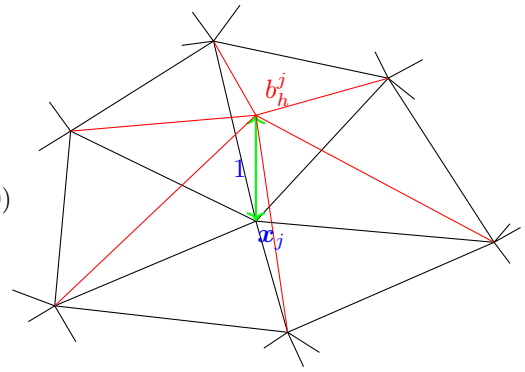


Figure 12: A local nodal basis function over a triangle with the pink node being the *pivot*

Writing $\mathcal{V}(\mathcal{M}) = \{\boldsymbol{x}_1, ..., \boldsymbol{x}_N\}$, an ordering of the nodes is implied, the nodal basis $\mathfrak{B}_h = \{b_h^1, ..., b_h^N\}$ of $\mathcal{S}_1^0(\mathcal{M})$ satisfies the conditions

$$
\begin{aligned}
&b_h^j \in \mathcal{S}_1^0(\mathcal{M}),\\
&b_h^j(\boldsymbol{x}_i) = \begin{cases} 1, & \text{if } \boldsymbol{x}_i = \boldsymbol{x}_j,\\ 0, & \text{if } \boldsymbol{x}_i \in \mathcal{V}(\mathcal{M}) \setminus \{\boldsymbol{x}_j\}, \end{cases}\\
&i, j = \{1, ..., N\}.
\end{aligned} \tag{30}
$$



Given the cardinal basis property of $\mathfrak{B}_h$ with respect to the node set $\mathcal{V}(\mathcal{M}) = \{\boldsymbol{x}_1, ..., \boldsymbol{x}_N\}$ in (30), the coefficients of the nodal basis expansion of a $u_h \in \mathcal{S}_1^0(\mathcal{M})$ coincide with the nodal values $u_h(\boldsymbol{x}_i)$:

$$
u_h \in \mathcal{S}_1^0(\mathcal{M}): \quad u_h = \sum_{i=1}^{N} \mu_i b_h^i \quad \Leftrightarrow \quad \mu_i = u_h(\boldsymbol{x}_i) \quad \forall i = 1, ..., N. \tag{31}
$$

2.2 Computing Galerkin Matrices and R.H.S. Vectors

2.2.1 In One-Dimension

In Section 2.1.3 we have obtained the 1D tent basis functions

$$
b_h^j(x) := \begin{cases} (x - x_{j-1})/h_j, & \text{if } x_{j-1} \le x \le x_j,\\ (x_{j+1} - x)/h_{j+1}, & \text{if } x_j \le x \le x_{j+1},\\ 0, & \text{elsewhere.} \end{cases}
$$

Then we can conclude that

$$
|i - j| \ge 2 \quad \Rightarrow \quad \frac{\mathrm{d}b_h^j}{\mathrm{d}x}(x)\frac{\mathrm{d}b_h^i}{\mathrm{d}x}(x) = 0, \quad b_h^j(x) b_h^i(x) = 0 \quad \forall x \in [a, b], \tag{32}
$$

which follows from the fact that there is no overlap of supports of the two basis functions.

Since by (10) and (20)

$$(\mathbf{A})_{ij} = a(b_h^j, b_h^i) = \int_a^b \left(p \frac{\mathrm{d}b_h^j}{\mathrm{d}x} \frac{\mathrm{d}b_h^i}{\mathrm{d}x} + q b_h^j b_h^i \right) \mathrm{d}x, \tag{33}$$

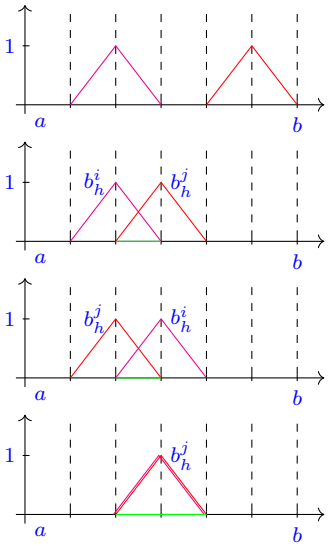
(32) suggests that the Galerkin matrix for problem P1 is *tridiagonal* (also *symmetric* by (33)).

For simplicity, we first look at a simplest case where $p(x) = 1,\ q(x) = 0$. Noting that the gradients (derivatives) of the tent functions are piecewise constant:

$$\frac{\mathrm{d}b_h^j}{\mathrm{d}x} = \begin{cases} h_j, & \text{if } x_{j-1} \le x \le x_j, \\ -h_{j+1}, & \text{if } x_j \le x \le x_{j+1}, \\ 0, & \text{elsewhere}, \end{cases} \tag{34}$$

then it immediately follows that

$$\int_a^b \frac{\mathrm{d}b_h^j}{\mathrm{d}x}(x) \frac{\mathrm{d}b_h^i}{\mathrm{d}x}(x)\,\mathrm{d}x = \begin{cases} 0, & \text{if } |i-j| > 2, \\ -\dfrac{1}{h_{i+1}}, & \text{if } j = i+1, \\ -\dfrac{1}{h_i}, & \text{if } j = i-1, \\ \dfrac{1}{h_i} + \dfrac{1}{h_{i+1}}, & \text{if } 1 \le i = j \le M-1. \end{cases}$$



So we can obtain the following Galerkin matrix which is *symmetric*, *positive definite*, and *tridiagonal*:

$$\mathbf{A} = \begin{bmatrix} \frac{1}{h_1} + \frac{1}{h_2} & -\frac{1}{h_2} & 0 & & & 0 \\ -\frac{1}{h_2} & \frac{1}{h_2} + \frac{1}{h_3} & -\frac{1}{h_3} & & & \\ 0 & \ddots & \ddots & \ddots & & \\ & & & & \ddots & 0 \\ & & & \ddots & \ddots & -\frac{1}{h_{M-1}} \\ 0 & & & 0 & -\frac{1}{h_{M-1}} & \frac{1}{h_{M-1}} + \frac{1}{h_M} \end{bmatrix} \in \mathbb{R}^{N,N}, \quad N := M-1. \tag{35}$$

Specially, for an equidistant mesh $\mathcal{M}$ with uniform meshwidth $h > 0$ the finite element linear system of equations (19) becomes

$$\frac{1}{h} \begin{bmatrix} 2 & -1 & 0 & & & 0 \\ -1 & 2 & -1 & & & \\ 0 & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & 0 \\ & & & -1 & 2 & -1 \\ 0 & & & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} \mu_1 \\ \vdots \\ \mu_N \end{bmatrix} = h \begin{bmatrix} f(x_1) \\ \vdots \\ f(x_N) \end{bmatrix}. \tag{36}$$

In fact, by combining the composite trapezoidal rule

$$\int_a^b \psi(t)\,\mathrm{d}t \approx \sum_{j=1}^{M} \frac{1}{2} h_j(\psi(x_{j-1}) + \psi(x_j)) \tag{37}$$

and the cardinal basis property (24) we have

$$\varphi_k := (\vec{\boldsymbol{\varphi}})_k = \int_a^b f(x) b_h^k(x)\,\mathrm{d}x \approx \frac{1}{2}(h_k + h_{k+1}) f(x_k), \quad 1 \le k \le N, \tag{38}$$

which explains the right-hand side vector in (36).

More generally, the Galerkin matrix and the right-hand side vector can be computed by the following formulas:

$$\begin{aligned} a(b_h^j, b_h^{j-1}) = a(b_h^{j-1}, b_h^j) &= \int_{x_{j-1}}^{x_j} \left[p \frac{\mathrm{d}b_h^j}{\mathrm{d}x} \frac{\mathrm{d}b_h^{j-1}}{\mathrm{d}x} + q b_h^j b_h^{j-1} \right] \mathrm{d}x \\ &= \int_{x_{j-1}}^{x_j} \left[-p(x) h_j^{-2} + q(x) b_h^j(x) b_h^{j-1}(x) \right] \mathrm{d}x \\ &= \int_0^1 \left[-h_j^{-1} p(x_{j-1} + h_j\xi) + h_j q(x_{j-1} + h_j\xi)(1-\xi)\xi \right] \mathrm{d}\xi, \end{aligned} \tag{39}$$

$$\begin{aligned} a(b_h^j, b_h^j) &= \int_{x_{j-1}}^{x_j} \left[p(\frac{\mathrm{d}b_h^j}{\mathrm{d}x})^2 + q(b_h^j)^2 \right] \mathrm{d}x + \int_{x_j}^{x_{j+1}} \left[p(\frac{\mathrm{d}b_h^j}{\mathrm{d}x})^2 + q(b_h^j)^2 \right] \mathrm{d}x \\ &= \int_{x_{j-1}}^{x_j} \left[p(x) h_j^{-2} + q(x)(b_h^j(x))^2 \right] \mathrm{d}x + \int_{x_j}^{x_{j+1}} \left[p(x) h_{j+1}^{-2} + q(x)(b_h^j(x))^2 \right] \mathrm{d}x \\ &= \int_0^1 \left[h_j^{-1} p(x_{j-1} + h_j\xi) + h_j q(x_{j-1} + h_j\xi)\xi^2 \right] \mathrm{d}\xi + \\ &\quad \int_0^1 \left[h_{j+1}^{-1} p(x_j + h_{j+1}\xi) + h_{j+1} q(x_j + h_{j+1}\xi)(1-\xi)^2 \right] \mathrm{d}\xi, \end{aligned} \tag{40}$$

$$\begin{aligned} (\vec{\boldsymbol{\varphi}})_j &= \int_{x_{j-1}}^{x_j} f(x) b_h^j(x)\,\mathrm{d}x + \int_{x_j}^{x_{j+1}} f(x) b_h^j(x)\,\mathrm{d}x \\ &= h_j \int_0^1 f(x_{j-1} + h_j\xi)\xi\,\mathrm{d}\xi + h_{j+1} \int_0^1 f(x_j + h_{j+1}\xi)(1-\xi)\,\mathrm{d}\xi. \end{aligned} \tag{41}$$

Here $j = 2, ..., N$ for (39), and $j = 1, ..., N$ for (40) and (41).

Next, we focus on the 2D computations of Galerkin matrix and RHS vector. Before the start of our computations we shall first look at the structures of Galerkin matrices.

2.2.2 Sparsity of Galerkin Matrix

We have learned from previous sub-subsection that the Galerkin matrix is *tridiagonal* under the linear finite element Galerkin discretization in one dimension. In the 2D counterpart, we can also prove that it is always sparse, that is, most of its elements are zero.

Lemma 2.1 (Sparsity of Galerkin matrix [Hip21, Lemma 2.4.4.2])**.** *There is a constant $C > 0$ depending only on the topology of Ω, that is, the number of "holes" in it, such that for any triangular mesh $\mathcal{M}$ of Ω ($N := \sharp\mathcal{V}(\mathcal{M})$ = number of vertices)*

$$\sharp\{(i,j) \in \{1, ..., N\}^2 : (\mathbf{A})_{ij} \neq 0\} \le 7 \cdot N + C,$$

where $\mathbf{A}$ is any Galerkin matrix arising from a discretization of a 2nd-order linear scalar elliptic variational problem with linear finite elements.

Proof. We rely on Euler's formula for triangulations.

$$\sharp\mathcal{M} - \sharp\mathcal{E}(\mathcal{M}) + \sharp\mathcal{V}(\mathcal{M}) = \chi_\Omega, \quad \chi_\Omega = \text{Euler characteristic of } \Omega.$$

Note that χ_Ω is a topological invariant (alternating sum of Betti numbers).
By combinatorial considerations (traverse edges and count triangles):

$$2 \cdot \sharp\mathcal{E}_I(\mathcal{M}) + \sharp\mathcal{E}_B(\mathcal{M}) = 3 \cdot \sharp\mathcal{M},$$

where $\mathcal{E}_I(\mathcal{M})$, $\mathcal{E}_B(\mathcal{M})$ stands for the sets of interior and boundary edges of $\mathcal{M}$, respectively.
Combining the above two equations yields

$$\sharp\mathcal{E}_I(\mathcal{M}) + 2 \cdot \sharp\mathcal{E}_B(\mathcal{M}) = 3(\sharp\mathcal{V}(\mathcal{M}) - \chi_\Omega).$$

Then use

$$N = \sharp\mathcal{V}(\mathcal{M}), \quad \operatorname{nnz}(\mathbf{A}) \le N + 2 \cdot \sharp\mathcal{E}(\mathcal{M}) \le 7 \cdot \sharp\mathcal{V}(\mathcal{M}) - 6\chi_\Omega,$$

which implies the assertion for any triangulation. This completes the proof. □

2.2.3 Computation of Galerkin Matrix

Now we investigate an efficient algorithm for computing the non-zero entries of the sparse finite element Galerkin matrix. For the sake of simplicity, we shall, for the moment, drop the second term $\gamma(\boldsymbol{x})u$ and let $\boldsymbol{\alpha}(\boldsymbol{x})$ be a 2×2 identity matrix in (3) so that the bilinear form in (12) becomes

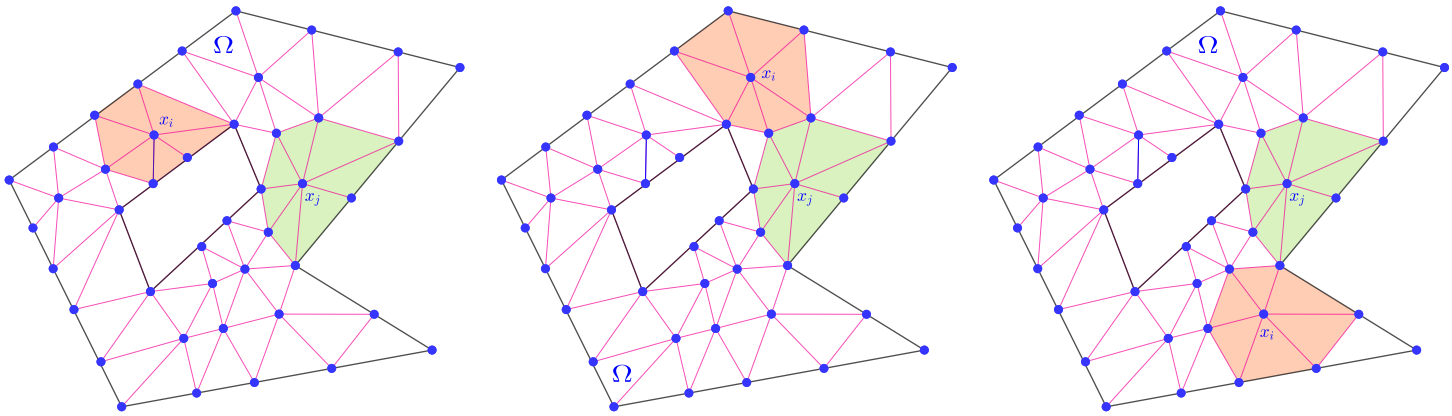
$$a(u_h, v_h) := \int_\Omega \mathbf{grad}\, u_h \cdot \mathbf{grad}\, v_h \, \mathrm{d}\boldsymbol{x}, \quad u_h, v_h \in H^1(\Omega),$$

thus leading the entries of the Galerkin matrix $\mathbf{A}$ to

$$(\mathbf{A})_{ij} = a(b_h^j, b_h^i) = \int_\Omega \mathbf{grad}\, b_h^j \cdot \mathbf{grad}\, b_h^i \, \mathrm{d}\boldsymbol{x}.$$

It is not hard to see that

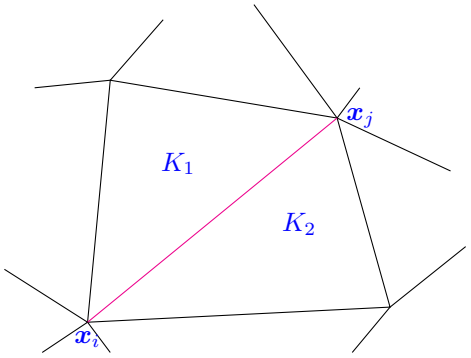
$$\left\{ \begin{array}{c} \text{Differing nodes } \boldsymbol{x}_i, \boldsymbol{x}_j \in \mathcal{V}(\mathcal{M}) \\ \text{that are not connected by an edge} \end{array} \right. \quad \Leftrightarrow \quad \left. \operatorname{Vol}(\operatorname{supp}(b_h^i) \cap \operatorname{supp}(b_h^j)) = 0 \right\} \quad \Rightarrow \quad (\mathbf{A})_{ij} = 0.$$



Therefore, in order to compute $(\mathbf{A})_{ij}$ we only need to deal with the situations where the two nodes $\boldsymbol{x}_i, \boldsymbol{x}_j \in \mathcal{V}(\mathcal{M})$ are either *connected by an edge* of the triangulation or *coincide*. We shall first elaborate the former case.

When two nodes are connected by an edge, the edge can be either an interior edge or a boundary edge. If it is an interior edge then it must be shared by two triangles whereas a boundary edge enjoys the full ownership by a particular boundary triangle. For the first case we can think of $(\mathbf{A})_{ij}$ as the result of summing up the two triangles:

$$(\mathbf{A})_{ij} = \int_{K_1} \mathbf{grad}\, b^j_{h|K_1} \cdot \mathbf{grad}\, b^i_{h|K_1}\, \mathrm{d}\boldsymbol{x} + \int_{K_2} \mathbf{grad}\, b^j_{h|K_2} \cdot \mathbf{grad}\, b^i_{h|K_2}\, \mathrm{d}\boldsymbol{x}$$



While the boundary edge case can be understood as

$$(\mathbf{A})_{ij} = \int_{K_\mathrm{B}} \mathbf{grad}\, b^j_{h|K_\mathrm{B}} \cdot \mathbf{grad}\, b^i_{h|K_\mathrm{B}}\, \mathrm{d}\boldsymbol{x},$$

where K_B is a triangle on the boundary.

In any cases, we can view them as *summing* ***cell contributions***, that is, if an edge is shared by two cells (triangles) then add them up, if it is only owned by a single cell (triangle) then add this single one. This idea is termed as ***assembly***, which is the *key* to implementing the finite element method.

Local Computations

Motivated by the formulas above we now fix our attention on a single triangle $K \in \mathcal{M}$, restrict the bilinear form to it, and examine the cell contribution

$$a_K(b^j_h, b^i_h) = \int_K \mathbf{grad}\, b^j_{h|K} \cdot \mathbf{grad}\, b^i_{h|K}\, \mathrm{d}\boldsymbol{x}, \quad \boldsymbol{x}_i, \boldsymbol{x}_j \text{ nodes} \in \text{vertices of } K. \tag{42}$$

Thus it is desirable if we could find out the analytic formulas for the restrictions $b^i_{h|K}$. Let $\boldsymbol{a}^1_K, \boldsymbol{a}^2_K, \boldsymbol{a}^3_K$ be the vertices of the triangle K with coordinates $\boldsymbol{a}^1_K = \begin{bmatrix} a^1_1 \\ a^1_2 \end{bmatrix}$, $\boldsymbol{a}^2_K = \begin{bmatrix} a^2_1 \\ a^2_2 \end{bmatrix}$, and $\boldsymbol{a}^3_K = \begin{bmatrix} a^3_1 \\ a^3_2 \end{bmatrix}$, we write

$$\lambda_i := b^j_{h|K} \quad \text{with} \quad \boldsymbol{a}^i_K = \boldsymbol{x}_j. \quad \begin{bmatrix} i \leftrightarrow \textit{local} \text{ vertex number} \\ j \leftrightarrow \textit{global} \text{ node number} \end{bmatrix} \tag{43}$$

Obviously, we have 3 such functions $\lambda_1, \lambda_2, \lambda_3$ on any triangle $K \in \mathcal{M}$ with x_j being the 3 vertices of the triangle respectively, for example, the green surface given in Figure 13 represents the graph of λ_2.

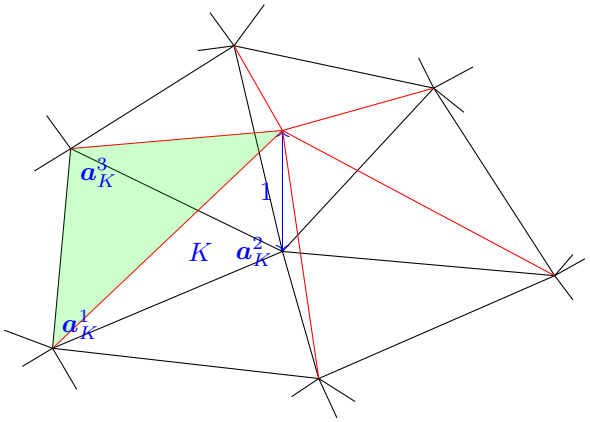


Figure 13: ⇑ a barycentric coordinate function λ_2

The functions $\lambda_1, \lambda_2, \lambda_3$ on the triangle K are also known as barycentric coordinate functions, which owe their name to the fact that they can be regarded as "coordinates of a point with respect to the vertices of a triangle" in the sense that

$$\boldsymbol{x} = \lambda_1(\boldsymbol{x})\boldsymbol{a}^1_K + \lambda_2(\boldsymbol{x})\boldsymbol{a}^2_K + \lambda_3(\boldsymbol{x})\boldsymbol{a}^3_K. \tag{44}$$

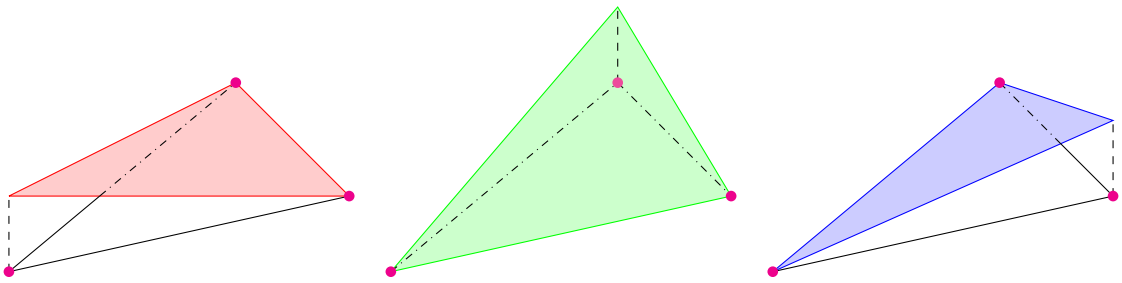
For instance, say, $\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$, $\boldsymbol{a}_K^1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, $\boldsymbol{a}_K^2 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, and $\boldsymbol{a}_K^3 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, then $\lambda_1(\boldsymbol{x}) = 1 - x_1 - x_2$, $\lambda_2(\boldsymbol{x}) = x_1$, $\lambda_3(\boldsymbol{x}) = x_2$. According to (44), the following equation holds true:

$$\boldsymbol{x} = (1 - x_1 - x_2) \begin{bmatrix} 0 \\ 0 \end{bmatrix} + x_1 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + x_2 \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

In addition, as the attribute "barycentric" indicates, the barycentric coordinate functions $\lambda_1, \lambda_2, \lambda_3$ satisfy

$$\lambda_1 + \lambda_2 + \lambda_3 = 1.$$

For the sake of comparison, we plot the graphs of the functions $\lambda_1, \lambda_2, \lambda_3$ in alignment:



Since the barycentric coordinate functions $\lambda_1, \lambda_2, \lambda_3$ are affine linear functions (visually, their graphs are planes) and for any fixed triangle $K \in \mathcal{M}$ they meet the cardinal basis property (30), we can write

$$\lambda_i(\boldsymbol{x}) = \alpha_i + \boldsymbol{\beta}^i \cdot \boldsymbol{x}, \tag{45}$$

and on a triangle K with coordinates $\boldsymbol{a}_K^1 = \begin{bmatrix} a_1^1 \\ a_2^1 \end{bmatrix}$, $\boldsymbol{a}_K^2 = \begin{bmatrix} a_1^2 \\ a_2^2 \end{bmatrix}$, and $\boldsymbol{a}_K^3 = \begin{bmatrix} a_1^3 \\ a_2^3 \end{bmatrix}$, they satisfy

$$\begin{bmatrix} 1 & a_1^1 & a_2^1 \\ 1 & a_1^2 & a_2^2 \\ 1 & a_1^3 & a_2^3 \end{bmatrix} \begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 \\ \beta_1^1 & \beta_1^2 & \beta_1^3 \\ \beta_2^1 & \beta_2^2 & \beta_2^3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{46}$$

Now we define the *element (stiffness) matrix*

$$\mathbf{A}_K := \left[\int_K \mathbf{grad}\, \lambda_i \cdot \mathbf{grad}\, \lambda_j \, \mathrm{d}\boldsymbol{x} \right]_{i,j=1}^{3} \in \mathbb{R}^{3,3}. \tag{47}$$

By (27), we have

$$\mathbf{grad}\, \lambda_i = \boldsymbol{\beta}^i, \tag{48}$$

which, together with (46), suggests an efficient way to compute the element (stiffness) matrix (47):

$$\mathbf{A}_K = |K| \begin{bmatrix} \beta_1^1 & \beta_1^2 & \beta_1^3 \\ \beta_2^1 & \beta_2^2 & \beta_2^3 \end{bmatrix}^\top \begin{bmatrix} \beta_1^1 & \beta_1^2 & \beta_1^3 \\ \beta_2^1 & \beta_2^2 & \beta_2^3 \end{bmatrix} \in \mathbb{R}^{3,3}, \tag{49}$$

where $\begin{bmatrix} \beta_1^1 & \beta_1^2 & \beta_1^3 \\ \beta_2^1 & \beta_2^2 & \beta_2^3 \end{bmatrix}$ can be computed by

$$\begin{bmatrix} \beta_1^1 & \beta_1^2 & \beta_1^3 \\ \beta_2^1 & \beta_2^2 & \beta_2^3 \end{bmatrix} = \left(\begin{bmatrix} 1 & a_1^1 & a_2^1 \\ 1 & a_1^2 & a_2^2 \\ 1 & a_1^3 & a_2^3 \end{bmatrix}^{-1} \right)_{(2:3,:)}. \tag{50}$$

Here the notation $(2:3,:)$ in the above equation (50) is adopted from MATLAB, meaning the submatrix taken from the 2nd row to the 3rd row.

Remark 1. $\mathbf{A}_K$ does not depend on the "size" of triangle K. This is followed by the reasoning below:

- Apparently, translation and rotation of K does not change $\mathbf{A}_K$.
- Scaling of K by a factor $\rho > 0$ has the effect that
  - the area $|K|$ is scaled by ρ^2,
  - the gradients $\mathbf{grad}\,\lambda_i$ are scaled by ρ^{-1} (imagine the graphs of λ_i: when the triangle shrinks with $\rho < 1$, they become steeper, otherwise flatter.)

  Combining the two effects above offset the scaling of the triangle K, thus rendering $\mathbf{A}_K$ invariant.

Assembly of Full Galerkin Matrix

Now we consider the computation of the full Galerkin matrix. This time our idea *assembly* mentioned before comes in handy. The computation of $\mathbf{A}_{ij}$ for $i \neq j$ starts from summing cell contributions

$$(\mathbf{A})_{ij} = \int_{K_1} \mathbf{grad}\, b^j_{h|K_1} \cdot \mathbf{grad}\, b^i_{h|K_1}\, \mathrm{d}\boldsymbol{x} + \int_{K_2} \mathbf{grad}\, b^j_{h|K_2} \cdot \mathbf{grad}\, b^i_{h|K_2}\, \mathrm{d}\boldsymbol{x},$$

which can be visualized as follows:

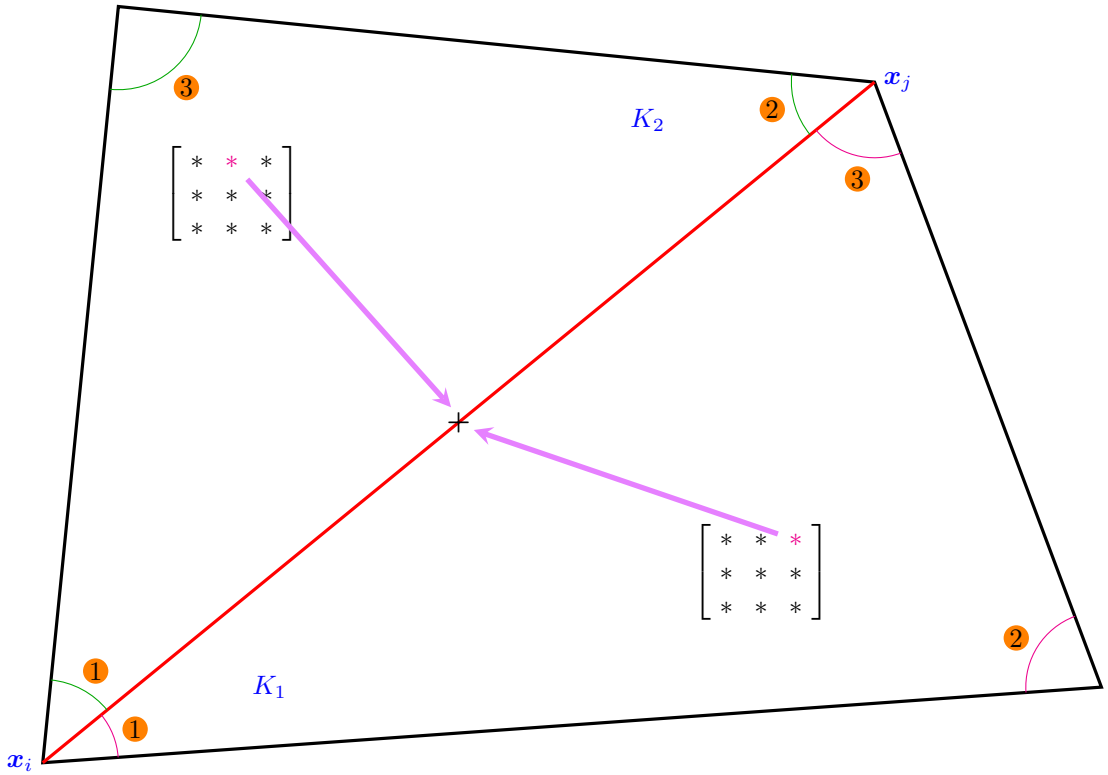


Figure 14: $\mathbf{A}_{ij}$ by summing entries of two element matrices

In the above diagram, 1, 2, and 3 represent the local vertex numbers; the magenta entries $*$ in the element matrices of K_1 and K_2 are the items that have contributions to $\mathbf{A}_{ij}$ (expressed by the edge, note the directivity, however, that $\mathbf{A}_{ij}$ is contributed by 1 3 from K_1 alongside 1 2 from K_2 whereas $\mathbf{A}_{ji}$ is contributed by 3 1 from K_1 and 2 1 from K_2. But in this thesis, the bilinear form $a(\cdot,\cdot)$ we use will all be symmetric thus the Galerkin matrix $\mathbf{A}$ and the element matrix $\mathbf{A}_K$ from (47) will also be symmetric,[5] so quantitatively speaking the directivity doesn't really matter. Otherwise we should make sure the relations are right).

[5] Indeed, whether the Galerkin matrix is symmetric is determined by a couple of factors: if the bilinear form is symmetric, how ever the boundary conditions are imposed, if we choose to use the same kinds of trial and test functions. By the Galerkin method we make it to the 1st and the 3rd ones, while the second one in real-world problems will usually render the final Galerkin matrix non-symmetric. But prior to processing the boundary conditions, the Galerkin matrix can be symmetric. This can be verified from numerical example 1 in Section 5.1.

As for the assembly of the diagonal entry $\mathbf{A}_{ii}$ of the Galerkin matrix $\mathbf{A}$, it can be obtained by summing corresponding diagonal entries of element matrices belonging to triangles adjacent to node x_i (see Figure 15).

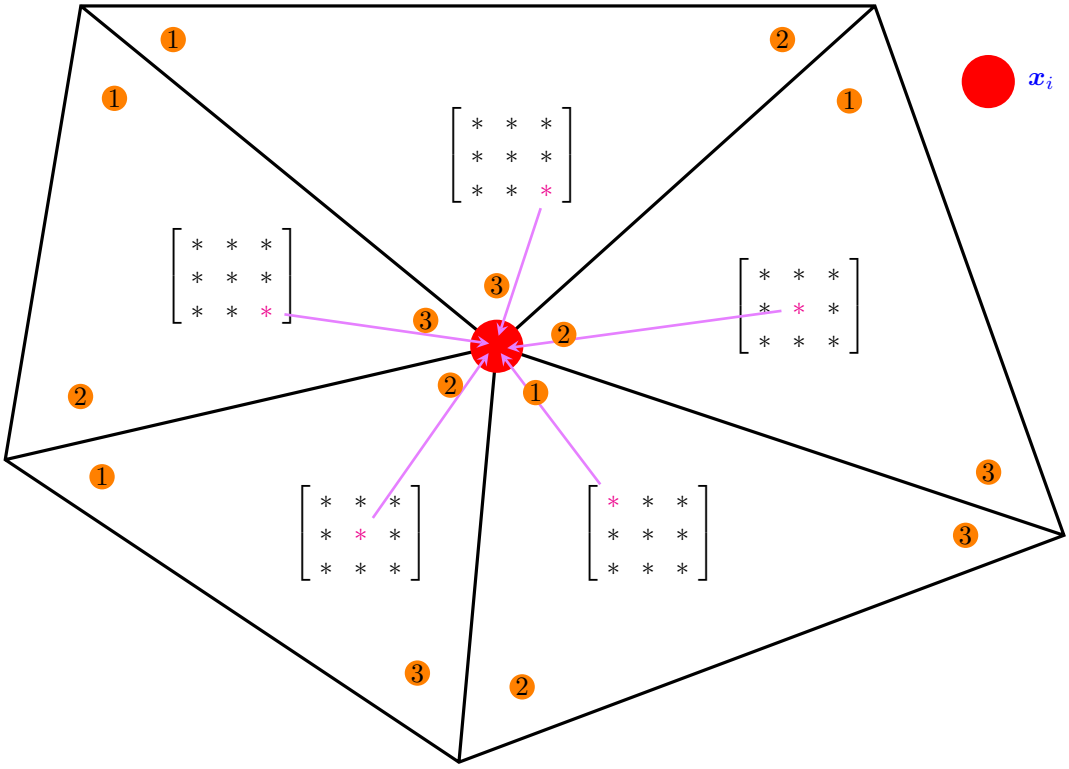


Figure 15: $\mathbf{A}_{ii}$ by summing diagonal entries of element matrices of adjacent triangles

Therefore, combining the two situations above we can develop a vertex-oriented assembly algorithm as follows:

Algorithm 1 Vertex-centered assembly of Galerkin matrix for linear finite elements

for all $e \in \mathcal{E}(\mathcal{M})$ **do**
  $(i, j) :=$ vertex numbers of endpoints of e
  $(\mathbf{A})_{ij} \leftarrow 0,\ (\mathbf{A})_{ji} \leftarrow 0$
  for all trangle K adjacent to e **do**
    find local numbers $l, m \in \{1, 2, 3\}$ of endpoints of e
    $(\mathbf{A})_{ij} \leftarrow (\mathbf{A})_{ij} + (\mathbf{A}_K)_{lm}$ ▷ see Figure 14
    $(\mathbf{A})_{ji} \leftarrow (\mathbf{A})_{ji} + (\mathbf{A}_K)_{ml}$ ▷ see Figure 14
  end for
end for
for all $\boldsymbol{v} \in \mathcal{V}(\mathcal{M})$ **do**
  $j :=$ number of vertex $\boldsymbol{v}$
  $(\mathbf{A})_{ij} \leftarrow 0$
  for all trangle K adjacent to $\boldsymbol{v}$ **do**
    $l :=$ local number of e in K
    $(\mathbf{A})_{jj} \leftarrow (\mathbf{A})_{jj} + (\mathbf{A}_K)_{ll}$ ▷ see Figure 15
  end for
end for

However, this sort of vertex-centered assembly algorithm is a little awkward for implementing, because, as we can see from the algorithm above, we not only have to traverse each edge and find the triangle(s) containing the edge, but also have to traverse each vertex as well as all the triangles adjacent to

the vertex, which means we need somewhat complex data structures for storing these information or some extra procedures in order to get the adjacent triangles in the nested loops so that the implementation is doable. In practice, we adopt another cell-oriented assembly scheme, which only needs to loop over all cells $K \in \mathcal{M}$ and *distribute* all entries of the element matrices $\mathbf{A}_K$ to the corresponding entries of the Galerkin matrix. This is illustrated in Figure 16.

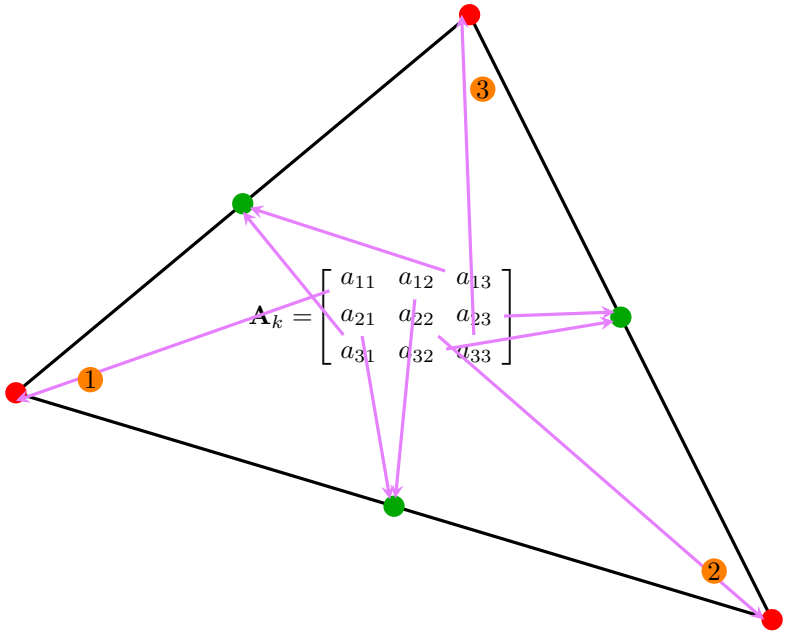


Figure 16: Cell-oriented assembly of Galerkin matrix by distribution from element matrices

The green circles ● can be regarded as the edges to which we can associate the off-diagonal entries of the Galerkin matrix. An edge can represent both $(\mathbf{A})_{ij}$ and $(\mathbf{A})_{ji}$, which accounts for why there are two symmetric off-diagonal entries in an element matrix contributing to an edge ●. The algorithm describing the distribution scheme is given below.

Algorithm 2 Cell-oriented assembly of Galerkin matrix for linear finite elements

SparseMatrix $\mathbf{A} \in \mathbb{R}^{N,N}$, $N := \sharp\mathcal{V}(\mathcal{M})$
$\mathbf{A} := \mathbf{0}$
$M := \sharp\mathcal{M}$ ▷ no. of cells
for $i = 1$ **to** M **do**
  $K \leftarrow$ `mesh.getElemCoords`(i) ▷ obtain cell-shape information
  $\mathbf{A}_K \leftarrow$ `getElemMatrix`(K) ▷ compute element matrix, see (49, 50)
  for $k = 1$ **to** 3 **do**
    for $j = 1$ **to** 3 **do**
      $\mathbf{A}(i : \boldsymbol{x}^i = \boldsymbol{a}_K^k, \ell : \boldsymbol{x}^\ell = \boldsymbol{a}_K^j) \mathrel{+}= \mathbf{A}_K(k, j)$ ▷ see Figure 16
    end for
  end for
end for

2.2.4 Computation of Right-Hand Side Vector

Computation of the right-hand side vector $\vec{\boldsymbol{\varphi}}$, as one possibly can guess, runs parallel to what we have studied in Section 2.2.3. We start from the right-hand side linear form

$$\ell(v) := \int_\Omega f(\boldsymbol{x}) v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x}, \quad v \in H^1(\Omega),\ f \in L^2(\Omega).$$

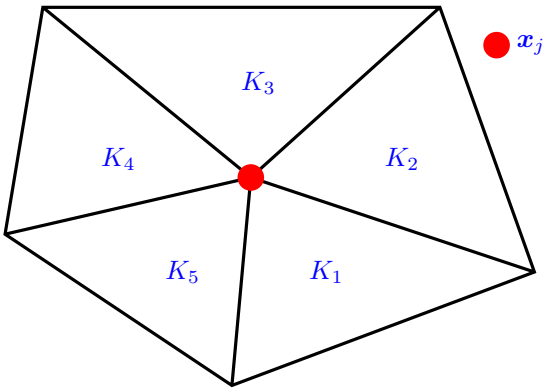
From (20) we have learned that $\vec{\boldsymbol{\varphi}} = \left[\ell(b_h^j)\right]_{j=1}^N$, so we have

$$(\vec{\boldsymbol{\varphi}})_j = \ell(b_h^j) = \int_\Omega f(\boldsymbol{x}) b_h^j(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}, \quad j = 1, ..., N. \tag{51}$$

Similarly, we split the right-hand side linear form into *cell contributions*:

$$(\vec{\boldsymbol{\varphi}})_j = \sum_{l=1}^{N_j} \int_{K_l} f(\boldsymbol{x}) b_{h|K_l}^j(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x},$$

where $K_1, ..., K_{N_j}$ are the triangles adjacent to node $\boldsymbol{x}_j$.



Likewise on a single triangle $K \in \mathcal{M}$ we can define

$$\ell_K(b_h^j) := \int_K f(\boldsymbol{x}) b_{h|K}^j(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \ell_K(\lambda_i), \tag{52}$$

where λ_i is the barycentric coordinate function defined in (43). Thus we can rewrite

$$(\vec{\boldsymbol{\varphi}})_j = \sum_{K, i: \boldsymbol{a}_K^i = \boldsymbol{x}_j} \ell_K(\lambda_i). \tag{53}$$

To implement this sort of collecting scheme we would emulate the clumsy and burdensome algorithm mentioned on page 16 somehow.

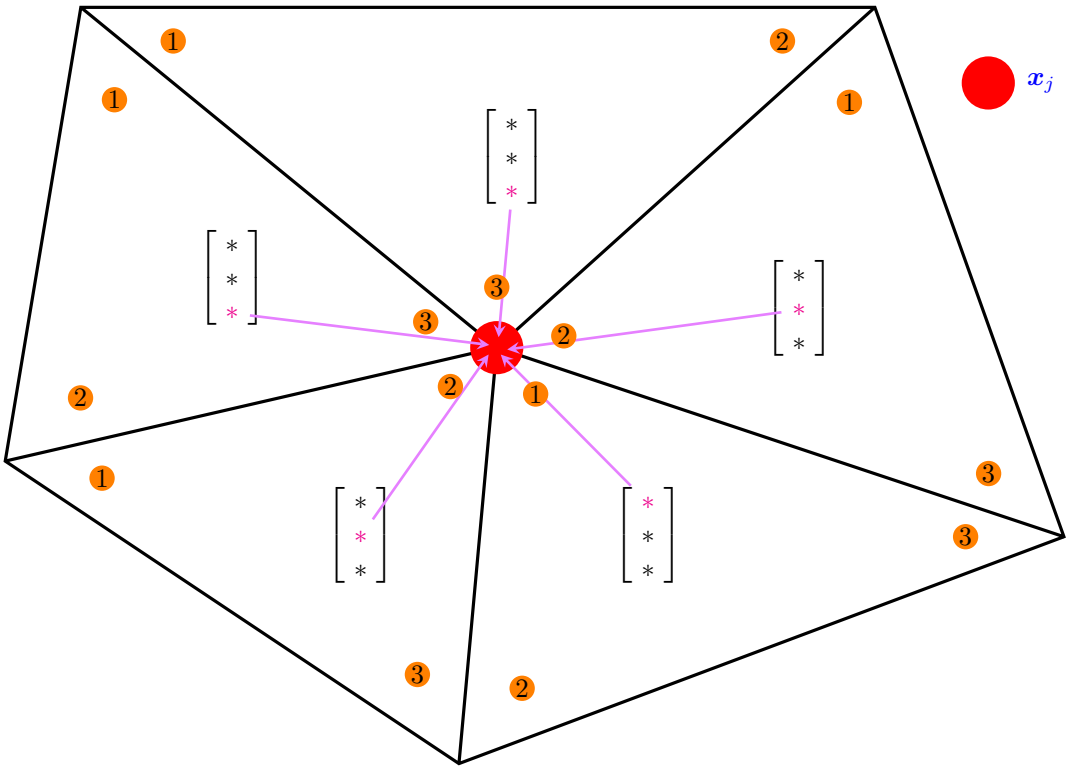


Figure 17: $(\vec{\boldsymbol{\varphi}})_j$ by summing diagonal entries of element vectors of adjacent triangles

But we have a better choice, that is to compute $\vec{\varphi}$ in a cell-oriented fashion, in which way we end up with a counterpart analogous to the element (stiffness) matrix from (47), the

$$\text{element (load) vector:} \quad \vec{\varphi}_K := [\ell_K(\lambda_i)]_{i=1}^3 \in \mathbb{R}^3. \tag{54}$$

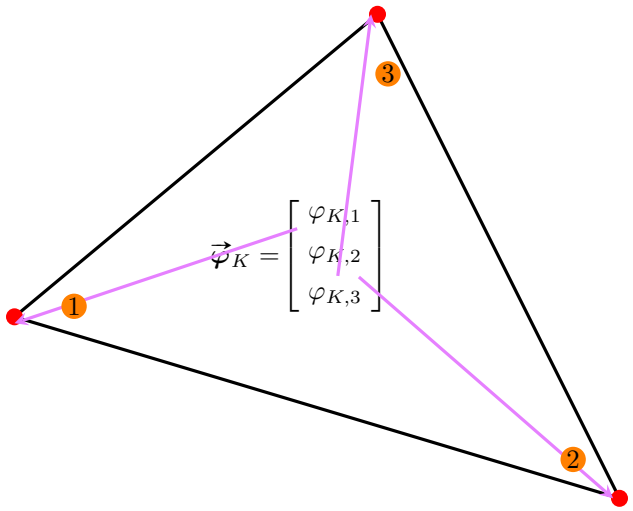
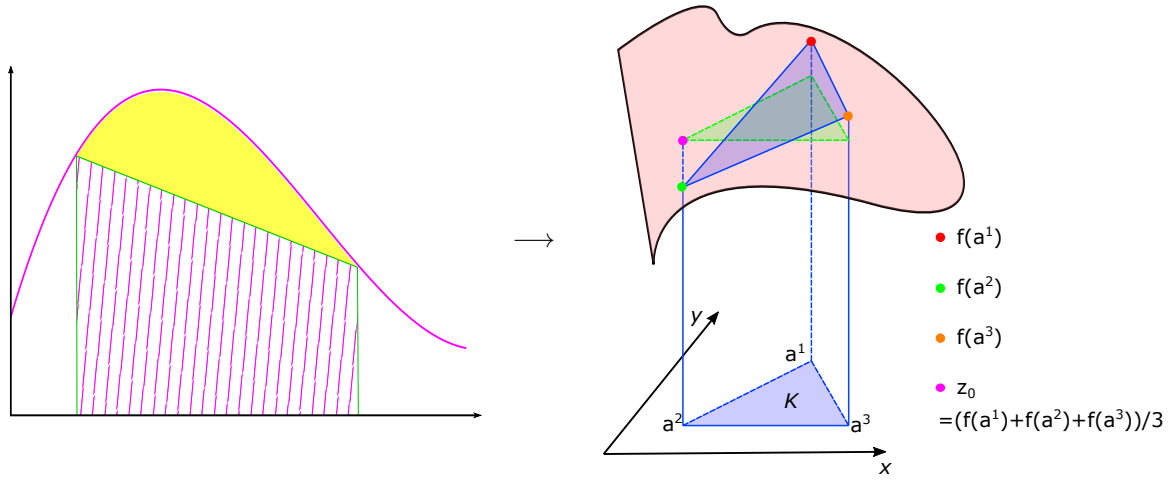


Figure 18: Cell-oriented assembly of right-hand side vector by distribution from element vectors

Now we consider how to compute $\vec{\varphi}_K$. We know that in 1D, by trapezoidal rule, we can approximate $\int_a^b f(x)\,\mathrm{d}x$ by accumulating integrals on many small trapezoids. We can extend it to 2D:



for triangle K with vertices $\boldsymbol{a}^1, \boldsymbol{a}^2, \boldsymbol{a}^3$

$$\int_K f(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \approx \frac{|K|}{3}\left(f(\boldsymbol{a}^1) + f(\boldsymbol{a}^2) + f(\boldsymbol{a}^3)\right). \tag{55}$$

Thus noting the cardinal basis property (30) we obtain

$$\vec{\varphi}_K := [\ell_K(\lambda_i)]_{i=1}^3 \approx \frac{|K|}{3}\begin{bmatrix} f(\boldsymbol{a}^1) \\ f(\boldsymbol{a}^2) \\ f(\boldsymbol{a}^3) \end{bmatrix}. \tag{56}$$

This formula can be used for our computation of the element vector. The algorithm that assembles the right-hand side vector will be presented in Section 4.3 in a more generic fashion.

3 Error Analysis

A rigorous and complete analysis for the error estimates of finite element methods is very complicated, which involves lots of techniques and expertise. Therefore, here we will only show some conclusions that will be used to verify our numerical experiments in Section 5 and Section 6.4. For more information about FEM error analysis one can refer to our text [Li10, Section 2.2, 2.8] or this paper [GB05] by Thomas Grätsch and Klaus-Jürgen Bathe.

For p-th order finite elements, the error measured in the L^∞-norm is of order $O(h^{p+1})$

$$\|u - u_h\|_{L^\infty(\Omega)} \leq Ch^{p+1}\|u\|,$$

in the L^2-norm is of order $O(h^{p+1})$

$$\|u - u_h\|_{L^2(\Omega)} \leq Ch^{p+1}\|u\|,$$

and in the H^1-norm is of order $O(h^p)$

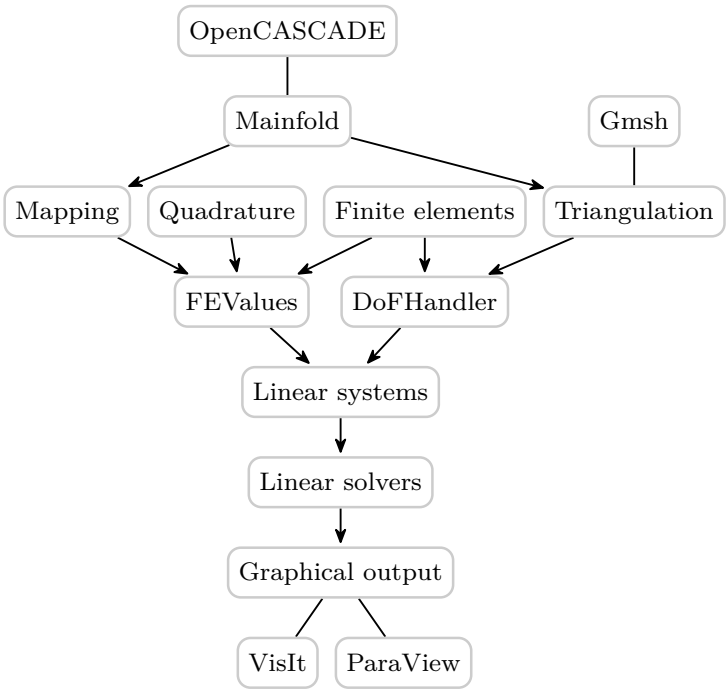
$$\|u - u_h\|_{H^1(\Omega)} \leq Ch^{p}\|u\|.$$

4 Implementation

In this section, we discuss the nuts and bolts of the implementation of finite elements methods. However, implementing a robust, industry-standard FEM library is a demanding job, requiring one to equip with not only a whole lot of mathematical skills but also exceptional programming ability (especially strong background in some compiled programming languages, like, C, C++, Fortran). The good news is that we don't have to implement one from scratch since there are many well-established libraries out there now which we can use and learn from. Moreover, in practice, it is recommended that we use those existing libraries to develop and deploy our own new algorithms for the model problems we need to tackle as it can significantly reduce the development cycle and save us from onerous debugging nightmares. So we are not going to implement a fully functional, versatile, and extensible FEM library. Instead, we will present a minimal implementation in the context of MATLAB, which makes our life easier by providing us with a tremendous number of built-in handy subroutines (functions), and most of all, as well as a powerful yet easy-to-use visualization system. But this doesn't mean our implementation is trivial. We will cover many important aspects concerning implementing a FEM library.

In fact, many FEM libraries share similar frameworks and modules that specify how the procedure goes and what results should be yielded in a particular step. Thus it makes sense to have a look at some successful FEM libraries, from which, at least ideawise, we can draw some considerations.

Hence, we start with looking at an industrial-strength open-source FEM library deal.II, an outline of how its primary groups of classes (main modules) interact is depicted in the following graph (simplified, omitted modules PETSc, Trilinos, CUDA, UMFPACK for linear systems and linear solvers):



What we will discuss and try to implement is the 3rd level to the 5th level (top down). We shall first give some brief explanations to the modules in the above graph. For the detailed documentation one can refer to https://www.dealii.org/current/doxygen/deal.II/index.html.

Mainfolds describe the shape of cells and, more generally, the geometry of the domain on which one wants to solve an equation. The geometries can be obtained from the 3D modeling kernel OpenCASCADE via its APIs.

Triangulation module does the meshing job that takes input from the geometries in Mainfold. This is usually done with the help of some external mesh generation libraries or tools, e.g. Gmsh.

Finite element classes describe the properties of a finite element space as defined on the *unit* cell. This includes, for example, how many degrees of freedom are located at vertices, on lines, or in the interior of cells. In addition, values and gradients of individual shape functions at points on the unit cell

are also of course provided. The DoFHandler class allocates spaces so that each vertex, line, or cell of the triangulation has the correct number of them. It also gives them a global numbering.

The quadrature module is a set of rules that describe the location of quadrature points on the unit cell, and the weights of quadrature points thereon.

Mapping classes make computing the matrix and right-hand side entries or other quantities on each cell of a triangulation numerically practical by mapping the shape functions, quadrature points, and quadrature weights from the unit cell to each cell of a triangulation. The FEValues class is the result of finite element shape functions and their gradients being evaluated in quadrature points defined by a quadrature formula when mapped to the real cell.

After knowing the values and gradients of shape functions on individual cells (by FEValues) and the global numbers of the degrees of freedom on a cell (by DoFHandler), what we should do next is to assemble the system matrix (and right hand side) of the linear system. This is done in the module Linear systems. Then we apply some appropriate solvers to solve this linear system of equations, which is followed by some post-processing visualization if one wants to.

We can clearly see that this outline largely agrees with what we have examined in Section 2, but with more implementation details. Our own implementation will also cover these aspects though not in a very systematic way.

4.1 Mesh Generation, Index Mapping, and Mesh Refinement

We will begin with triangulations which are fundamental to finite element methods. In industry, there are quite a number of choices to do the meshing job, a popular one is Gmsh, which is based around four modules: Geometry, Mesh, Solver and Post-processing, and it can be used at 3 levels: through the GUI, through the dedicated .geo language, through the C++, C, Python, and Julia API.[GR09]

Below is an example of Gmsh-constructed and -rendered geometry (mesh) model. It can be expressed in a few lines of code in the highly-concise .geo language, which can be found in here.

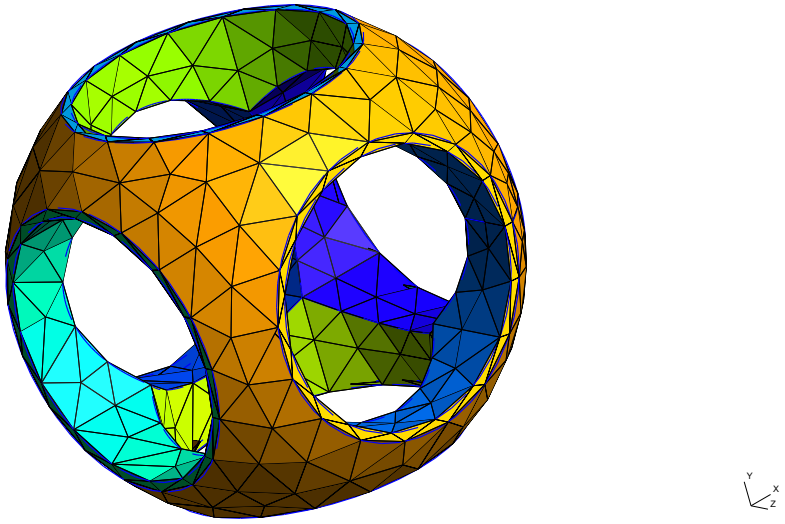
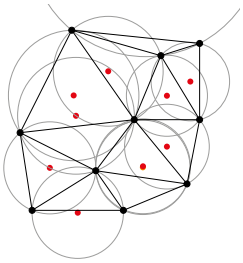


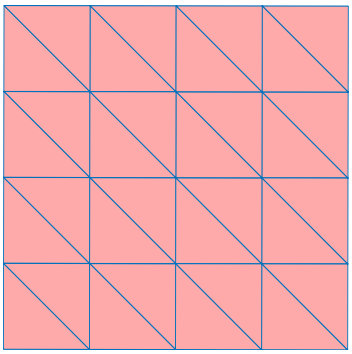
Figure 19: A sphere after deleting 3 orthogonal cylinders from its center



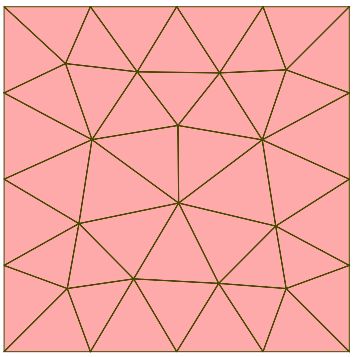
In MATLAB we can use its built-in `delaunay` subroutine for mesh generation. The Delaunay triangulation, according to Wolfram MathWorld, is a triangulation which is equivalent to the nerve of the cells in a Voronoi diagram, i.e., that triangulation of the convex hull of the points in the diagram in which every circumcircle of a triangle is an empty circle.

The Computational Geometry Algorithms Library (CGAL) which provides easy access to efficient and reliable geometric algorithms also incorporates a bundle of triangulations and Delaunay triangulations packages along with a large number of other powerful data structures and algorithms like Voronoi diagrams, cell complexes and polyhedra, convex hull algorithms, spatial searching and sorting, etc. The mesh generator DistMesh that developed by Per-Olof Persson and Gilbert Strang in the Department of Mathematics at MIT is a good alternative as well in the context of MATLAB.

Now we observe the mesh quality generated by the MATLAB `delaunay` subroutine. For simplicity and comparison, we assume the computational domain is a square and put the Gmsh counterpart by side. The results are represented in the two figures below.

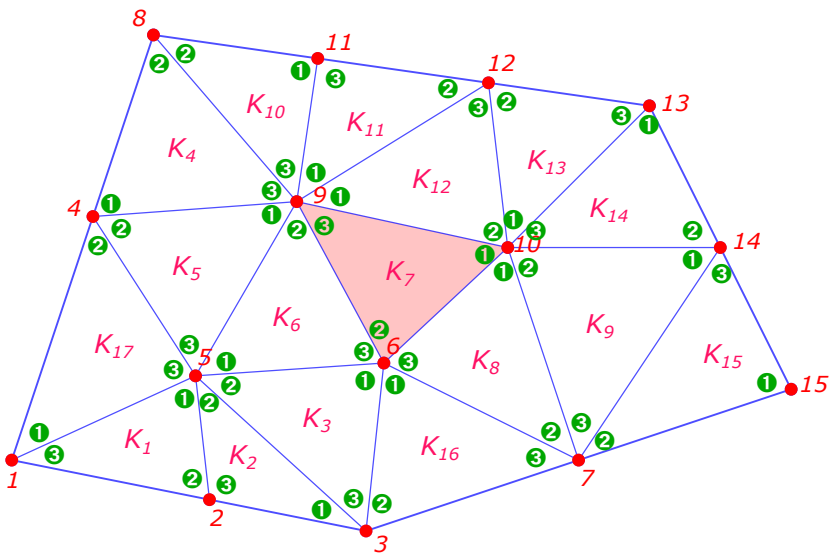


(a) A uniform triangular mesh generated via MATLAB `delaunay`



(b) A somewhat random triangular mesh generated via Gmsh

In fact, the subroutine `delaunay` takes two arguments in 2D with syntax like this: `TRI = delaunay(X,Y)`. And we of course for convenience chose to use the equidistant points as the input which led to the uniformly distributed triangular mesh. The Gmsh counterpart appears to be somewhat random but with almost equally sized cells. It was generated from the Gmsh GUI, possibly under the hood running some other algorithms in the kernel. The MATLAB code for generating uniform triangular meshes on a rectangular region is given in Code listing 1.



♯	❶	❷	❸
K_1	5	2	1
K_2	3	5	2
K_3	6	5	3
K_4	4	8	9
K_5	9	4	5
K_6	5	9	6
K_7	10	6	9
K_8	10	7	6
K_9	14	10	7
K_{10}	11	8	9
K_{11}	9	12	11
K_{12}	9	10	12
K_{13}	10	12	13
K_{14}	13	14	10
K_{15}	15	7	14
K_{16}	6	3	7
K_{17}	1	4	5

Figure 20: Index mapping by d.o.f. mapper

We have learned from Section 2 as well as the deal.II library that the global node numbers and local vertex numbers are soooo important to the assembly of Galerkin matrices and right-hand side vectors.

Hence, we zero in on the triangulation numbering and index mapping.

First, we consider the vertices (points/nodes). Simply enough we can use an array for storing them, specifically, a `nP-by-nDim` array. That is, there are `nP` (number of points) rows and each row represents the point's coordinates. In the 2D case, we can also use 3 columns to represent the 2D coordinates leaving the 3rd column (coordinate of z) being 0. This will comply with the 3D case. Thus to number the vertices we just need to record their corresponding row numbers (indices). These numbers are our global node numbers.

Then, the elements of the mesh (triangles) can be represented by their corresponding vertex numbers (indices). Note that we also have to pay attention to the local numbers. Hence we store an element in the order with respect to the local vertex numbers ❶, ❷, and ❸. This is illustrated in Figure 20 and we call it index mapping by d.o.f. mapper. Actually, this is also what `TRI = delaunay(X,Y)` is done for us: it takes the points `(X,Y)` and creates a 2D Delaunay triangulation represented by `TRI` which is a matrix of size `mtri-by-3`, where `mtri` is the number of triangles. Each row of `TRI` specifies a triangle defined by indices with respect to the points.

Mathematically, we can denote the local $\mapsto$ global index mapping (d.o.f. mapper) by $\texttt{dofh} \in \mathbb{N}^{\sharp\mathcal{M},3}$ with the meaning as follows:

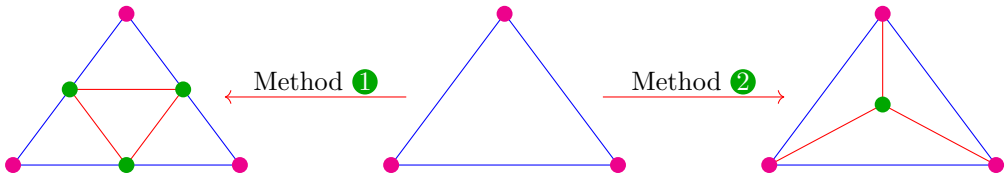
$$\begin{array}{rcl} \texttt{dofh}(k,l) & = & \text{global number of vertex } l \text{ of } k\text{-th cell} \in \{1,...,N\}, \\ \boldsymbol{x}_{\texttt{dofh}(k,l)} & = & \boldsymbol{a}^l \text{ when } \boldsymbol{a}^1, \boldsymbol{a}^2, \boldsymbol{a}^3 \text{ are the vertices of } K_k, \end{array} \tag{57}$$

for $l \in \{1,2,3\},\ k \in \{1,...,M\},\ M := \sharp\mathcal{M},\ N := \sharp\mathcal{V}(\mathcal{M})$.

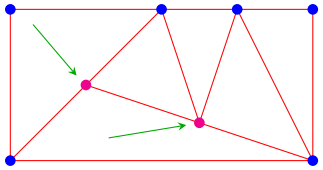
For example, the three vertices of the pink triangle K_7 in Figure 20 can be represented with indices $\texttt{dofh}(7,1) = 10,\ \texttt{dofh}(7,2) = 6,$ and $\texttt{dofh}(7,3) = 9$, respectively.

The d.o.f. mapper `dofh` as described in (57) in a sense amounts to the `DoFHandler` class in the deal.ii library, or rather it is a crude and simple function that implements the index mapping.

Next we give a short discussion about mesh refinement. We know that the accuracy of finite element methods is closely related to the mesh size $h_{\mathcal{M}}$. So naturally people would like to get finer meshes. We can achieve this by dividing the existing cells into sub-cells. Take the triangular meshes for an example, one way is to add 3 more nodes at the midpoints of the 3 edges of a cell and cut this triangle along these new edges connected by the midpoints. Thus a triangular cell is divided into four congruent half-sized triangles.



Another approach for the triangular meshes is to add some node(s) inside a cell, for instance, adding a new node in the barycenter of a triangle. Adding more than one inner point is also possible (e.g. adding one node in the barycenter then another node, again, at the barycenter of a divided sub-triangle). In practice, the first approach is often adopted as it creates regular/uniform finer meshes, which is usually desirable.



In the end, it should be emphasized that the triangulation or mesh refinement should not contain or give rise to any "hanging nodes", that is it shouldn't violate the restriction that each triangle side (if not a boundary edge) is entirely shared by two adjacent triangles (implied by principle (iv) of triangulation as defined on page 5).

4.2 Local Computations

In Section 2.2.3 we have explored how to compute the Galerkin matrix with omitting the diffusion coefficient $\boldsymbol{\alpha}$ as well as the whole second term $\gamma(\boldsymbol{x})u$ in (3) for the sake of simplicity. Now we add them back and focus on the computation of the element matrix that corresponds to the bilinear form

$$a(u,v) = \int_\Omega \boldsymbol{\alpha}(\boldsymbol{x})\mathbf{grad}\, u \cdot \mathbf{grad}\, v + \gamma(\boldsymbol{x})uv \,\mathrm{d}\boldsymbol{x}, \quad u, v \in H^1(\Omega).$$

The element matrix for the above bilinear form is analogous to the one presented in (47), that is

$$\mathbf{A}_K := [a_K(\lambda_i, \lambda_j)]_{i,j=1}^3 \in \mathbb{R}^{3,3}, \tag{58}$$

$$a_K(\lambda_i, \lambda_j) = \int_K \boldsymbol{\alpha}(\boldsymbol{x})\mathbf{grad}\, \lambda_i \cdot \mathbf{grad}\, \lambda_j + \gamma(\boldsymbol{x})\lambda_i\lambda_j \,\mathrm{d}\boldsymbol{x}, \tag{59}$$

where λ_i, λ_j are the barycentric coordinate functions defined in (43).

We start our investigation from a lemma below, which comes in handy in a few situations.

Lemma 4.1 (Integration of powers of barycentric coordinate functions [Hip21, Lemma 2.7.5.5])**.** *For any non-degenerate d-simplex K with barycentric coordinate functions $\lambda_1, ..., \lambda_{d+1}$ and exponents $\alpha_j \in \mathbb{R}, j = 1, ..., d+1$,*

$$\int_K \lambda_1^{\alpha_1} \cdots \lambda_{d+1}^{\alpha_{d+1}} \,\mathrm{d}\boldsymbol{x} = d!\,|K| \frac{\alpha_1!\,\alpha_2! \cdot \cdots \cdot \alpha_{d+1}!}{(\alpha_1 + \alpha_2 + \cdots + \alpha_{d+1} + d)!} \quad \forall \boldsymbol{\alpha} \in \mathbb{R}^{d+1}.$$

A direct application of this lemma for now is to compute the element mass matrix

$$\mathbf{A}_{K_M} := \left[\int_K \lambda_i\lambda_j \,\mathrm{d}\boldsymbol{x}\right]_{i,j=1}^3 \in \mathbb{R}^{3,3}.$$

In fact, applying the above lemma with $d = 2$ we immediately obtain

$$\int_K \lambda_\ell(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = \frac{|K|}{3}, \quad \int_K \lambda_\ell(\boldsymbol{x})^2 \,\mathrm{d}\boldsymbol{x} = \frac{|K|}{6}, \quad \int_K \lambda_i(\boldsymbol{x})\lambda_j(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = \frac{|K|}{12}, \tag{60}$$

which suggest we can compute the element mass matrix by

$$\mathbf{A}_{K_M} = \frac{|K|}{12}\begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix}. \tag{61}$$

Equipped with this formula we are now able to solve the following PDE model problem:

$$-\Delta u + ku = f,$$

where k is a constant.

Of course, all the stuff we have studied before will also be used.

Woohoo! But wait a minute, what we get in here is an augmented version, $\gamma(\boldsymbol{x})$ is not necessarily a constant, plus we have the scary first term in (59) to handle, which seems nontrivial at all.

Hence, we turn to an approximation technique called quadrature rules that use weighted sum of function values at specific points to approximate the definite integral of a function. And we call such a way of doing this on an element $K \in \mathcal{M}$ a local quadrature rule, which can be expressed as follows:

$$\int_K f(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \approx \sum_{l=1}^{P_K} \omega_l^K f(\boldsymbol{\xi}_l^K), \quad \boldsymbol{\xi}_l^K \in K,\ \omega_l^K \in \mathbb{R},\ P_K \in \mathbb{N}. \tag{62}$$

Here, ω_l^K are the weights, $\boldsymbol{\xi}_l^K$ are the quadrature points, and P means P-point local quadrature rule.

Typically, these quadrature points defined by a quadrature formula are known on *reference elements*, that is, e.g. the unit interval $[0, 1] \in \mathbb{R}$ in 1D, the unit triangle ◺ connected by points $(0, 0), (1, 0), (0, 1) \in \mathbb{R}^2$ in 2D. Moreover, the function values and their gradients on a reference element can be rather easily computed. So an important step is to transform the quadrature points from the unit cell (reference element) to a real cell of a mesh. Then we are able to evaluate the function values and gradients on the real cells and subsequently compute (59). To achieve these we shall make a few preparations.

Local Shape Functions

First, we introduce an important notion appeared in the outline of deal.ii library on page 21, which is shape functions. In fact, we have encountered shape functions at the very beginning—the basis functions b_h^i are actually also called global shape functions (GSF)/global basis functions/degrees of freedom (DOFs). What we are currently interested in is the local shape functions, defined as below.

Definition 4.1 (Local shape functions (LSF))**.** Given a finite element function space on a mesh $\mathcal{M}$ with global shape functions $b_h^i, i = 1, ..., N$, for every mesh entity K we define

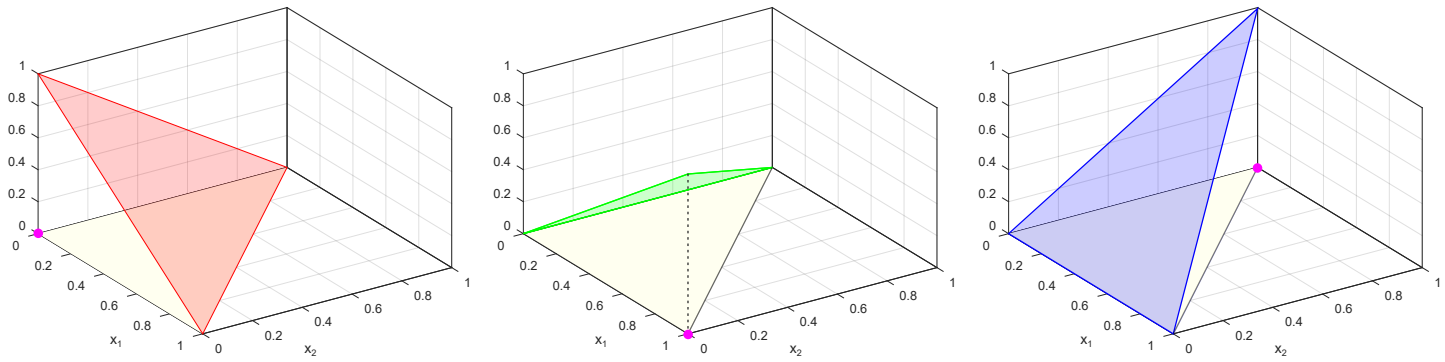
$$\{b_K^j\}_{j=1}^{Q(K)} := \{b_{h|K}^j, K \subset \text{interior of supp}(b_h^j)\} := \text{set of local shape functions (LSF)},$$

that is the local shape functions are the basis functions that cover K, restricted to K.

Note that $Q(K)$ is the number of local shape functions on the cell K, e.g. $Q(K) = 3$ for a triangular cell and $Q(K) = 4$ for a quadrilateral cell.

It turns out that on a triangular mesh there are exactly 3 local shape functions on each cell K, which indeed are the barycentric coordinate functions $\lambda_1, \lambda_2, \lambda_3$ introduced in Section 2.2.3.

Below is an important example of local shape functions on the unit triangle $\widehat{K}$ with vertices $\boldsymbol{a}_{\widehat{K}}^1 = \begin{bmatrix}0\\0\end{bmatrix}$, $\boldsymbol{a}_{\widehat{K}}^2 = \begin{bmatrix}1\\0\end{bmatrix}$, and $\boldsymbol{a}_{\widehat{K}}^3 = \begin{bmatrix}0\\1\end{bmatrix}$:



(a) $b_{\widehat{K}}^1(\boldsymbol{x}) := 1 - x_1 - x_2$ (b) $b_{\widehat{K}}^2(\boldsymbol{x}) := x_1$ (c) $b_{\widehat{K}}^3(\boldsymbol{x}) := x_2$

Figure 21: Local shape functions on the unit triangle $\widehat{K}$

We call the above local shape functions on the unit triangle (reference element) $\widehat{K}$ *reference shape functions* for the sake of convenience. These reference shape functions are simple enough to evaluate their function values and gradients since their analytic formulas are directly known and are arguably the most straightforward affine linear functions in 2D.

Affine Equivalence

Next, we introduce the affine transformation of triangles and pullback of functions, both of which are very practical and useful.

Lemma 4.2 (Affine transformation of triangles [Hip21, Lemma 2.7.5.14])**.** *For any non-degenerate triangle $K \subset \mathbb{R}^2$ ($|K| > 0$) with numbered vertices there is a unique affine transformation $\boldsymbol{\Phi}_K$, $\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}}) = \mathbf{F}_K\widehat{\boldsymbol{x}} + \boldsymbol{\tau}_K$, with $K = \boldsymbol{\Phi}_K(\widehat{K})$ and preserving the numbering of the vertices.*

This lemma tells us that all cells of a triangulation are affine images of the unit triangle $\widehat{K}$, see Figure 22 for an illustration. For a general triangle K with vertices $\boldsymbol{a}^1, \boldsymbol{a}^2, \boldsymbol{a}^3$, the affine mapping $\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}}) := \mathbf{F}_K\widehat{\boldsymbol{x}} + \boldsymbol{\tau}_K$ can be set with $\mathbf{F}_K := \begin{bmatrix}\boldsymbol{a}^2 - \boldsymbol{a}^1 & \boldsymbol{a}^3 - \boldsymbol{a}^1\end{bmatrix}$, $\boldsymbol{\tau}_K := \boldsymbol{a}^1$, i.e.

$$\mathbf{F}_K := \begin{bmatrix}a_1^2 - a_1^1 & a_1^3 - a_1^1\\ a_2^2 - a_2^1 & a_2^3 - a_2^1\end{bmatrix}, \quad \boldsymbol{\tau}_K := \begin{bmatrix}a_1^1\\ a_2^1\end{bmatrix}. \tag{63}$$

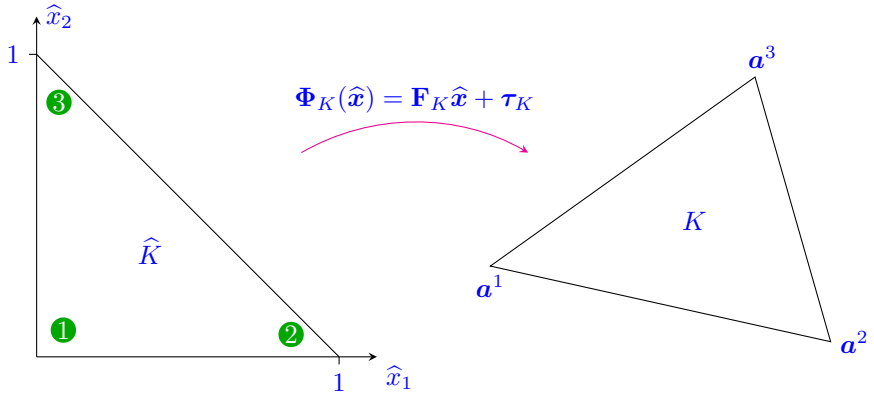
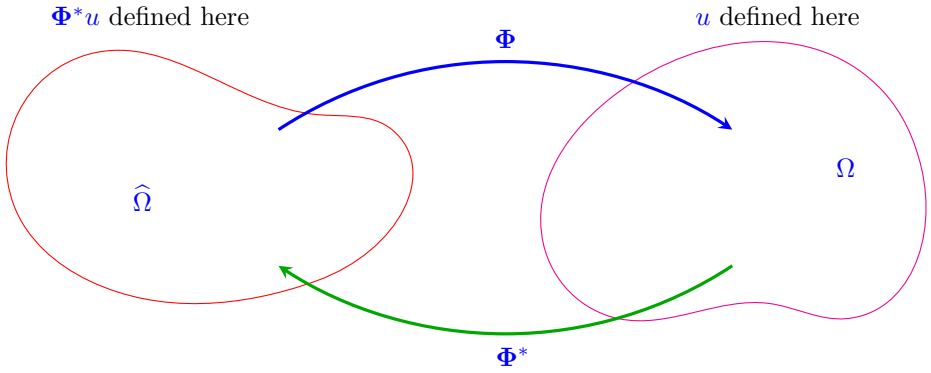


Figure 22: The affine mapping $\boldsymbol{\Phi}_K$ that transforms the unit triangle $\widehat{K}$ to a general triangle K

Definition 4.2 (Pullback)**.** Given domains $\Omega, \widehat{\Omega} \subset \mathbb{R}^d$ and a bijective mapping $\boldsymbol{\Phi} : \widehat{\Omega} \mapsto \Omega$, the pullback $\boldsymbol{\Phi}^* u : \widehat{\Omega} \mapsto \mathbb{R}$ of a function $u : \Omega \mapsto \mathbb{R}$ is a function defined on $\widehat{\Omega}$ by

$$\boldsymbol{\Phi}^* u(\widehat{\boldsymbol{x}}) := u(\boldsymbol{\Phi}(\widehat{\boldsymbol{x}})), \quad \widehat{\boldsymbol{x}} \in \widehat{\Omega}.$$

What the definition above is saying can be visualized in the following picture. Here is also an excellent explanation for pullback, which can be summarized as geometric objects (e.g. points, vectors) "go forward" and functions on them "go back".



We're concerned with the pullback of local shape functions. Let $K \in \mathcal{M}, \widehat{K}$ be the unit triangle, and $\boldsymbol{\Phi}_K$ the unique affine mapping $\widehat{K} \mapsto K$, which respects the local numbering of the vertices of $\widehat{K}$ and K: $\boldsymbol{\Phi}_K(\widehat{\boldsymbol{a}}^i) = \boldsymbol{a}^i,\ i = 1, 2, 3$. We write

- b_K^1, b_K^2, b_K^3 for the local shape functions on K, and
- $\widehat{b}^1, \widehat{b}^2, \widehat{b}^3$ for the local shape functions on $\widehat{K}$, i.e. reference shape functions (see Figure 21),

then we have a fundamental relationship in regard to the pullback $\boldsymbol{\Phi}_K^*$:

$$\widehat{b}^i = \boldsymbol{\Phi}_K^* b_h^i \quad \Leftrightarrow \quad \widehat{b}^i(\widehat{\boldsymbol{x}}) = b_K^i(\boldsymbol{x}), \quad \boldsymbol{x} = \boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}}). \tag{64}$$

This relationship is illustrated in Figure 23 which takes $\widehat{b}^1 \mapsto b_K^1$ for an example. A direct observation can be made from the fact that $\widehat{b}^1(A) = b_K^1(A') = 1$, $\widehat{b}^1(B) = b_K^1(B') = \widehat{b}^1(C) = b_K^1(C') = 0$. For a general case where the point $P \in \widehat{K}$, and the transformed point $P' = \boldsymbol{\Phi}_K(P) \in K$, all we have to do is notice a property that affine transformation possesses: it preserves the ratios of the lengths of parallel line segments. Therefore,

$$\frac{|EF|}{|BC|} = \frac{|E'F'|}{|B'C'|}, \quad \Rightarrow \quad \frac{|AP|}{|AH|} = \frac{|A'P'|}{|A'H'|}, \quad \Rightarrow \quad \frac{|PQ|}{|AD|} = \frac{|P'Q'|}{|A'D'|}.$$

Note that $|AD| = |A'D'| = 1$, thus we obtain $\widehat{b}^i(P) = |PQ| = |P'Q'| = b_K^i(P')$, which proves (64).

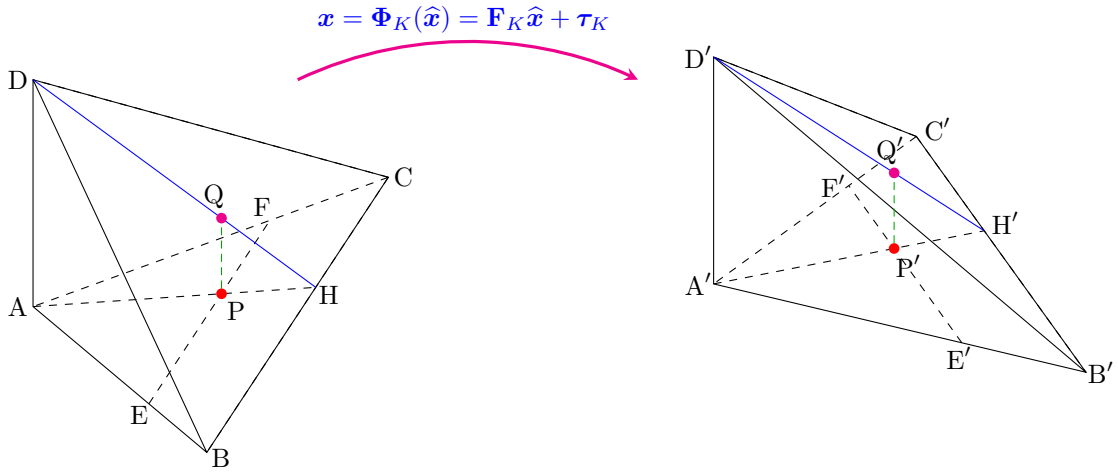


Figure 23: The affine mapping $\boldsymbol{\Phi}_K$ preserves the values of local shape functions:
$P \in \widehat{K}, P' = \boldsymbol{\Phi}_K(P) \in K, \quad \Rightarrow \quad \widehat{b}^i(P) = |PQ| = |P'Q'| = b^i_K(P').$

Local Quadrature

With the above preliminaries we are now all set for the final work of computing the element matrix in (58, 59). Let's dive in.

We write $\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}}) := \mathbf{F}_K\widehat{\boldsymbol{x}} + \boldsymbol{\tau}_K$ for the affine transformation from the reference triangle $\widehat{K}$ to the general triangle K. By the transformation formula for integrals we can pull back integrals over K to $\widehat{K}$:

$$\int_K f(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \int_{\widehat{K}} f(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}}))\,|\det \mathbf{F}_K|\,\mathrm{d}\widehat{\boldsymbol{x}}. \tag{65}$$

Combining this formula with (62) yields

$$\int_K f(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \approx |\det \mathbf{F}_K| \sum_{l=1}^{P} \widehat{\omega}_l\, f(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\xi}}_l)). \tag{66}$$

The above formulas can be used for computing the element reaction matrix (the second term) in (59):

$$\begin{aligned}
&\int_K \gamma(\boldsymbol{x})\ \underbrace{\lambda_i(\boldsymbol{x})}_{=b^i_K(\boldsymbol{x})}\ \underbrace{\lambda_j(\boldsymbol{x})}_{=b^j_K(\boldsymbol{x})}\ \mathrm{d}\boldsymbol{x} \\
&= \int_{\widehat{K}} \gamma(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}}))\ \underbrace{b^i_K(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}}))}_{\text{by (64), } =\widehat{b}^i(\widehat{\boldsymbol{x}})}\ \underbrace{b^j_K(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}}))}_{=\widehat{b}^j(\widehat{\boldsymbol{x}})}\ |\det \mathbf{F}_K|\,\mathrm{d}\widehat{\boldsymbol{x}} \\
&= |\det \mathbf{F}_K| \sum_{l=1}^{P} \widehat{\omega}_l\, \gamma(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\xi}}_l))\, \widehat{b}^i(\widehat{\boldsymbol{\xi}}_l)\, \widehat{b}^j(\widehat{\boldsymbol{\xi}}_l).
\end{aligned} \tag{67}$$

Here, $\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\xi}}_l), l = 1, ..., P$ are the transformed quadrature points, which together with $|\det \mathbf{F}_K|$ can be computed by (63), and the reference shape functions values $\widehat{b}^i(\widehat{\boldsymbol{\xi}}_l), i = 1, 2, 3, l = 1, ..., P$ on the quadrature points $\widehat{\boldsymbol{\xi}}_l$ defined on $\widehat{K}$ can be precomputed by `[1-x1-x2,x1,x2]` with `(x1,x2)` being these quadrature points.

The element vector (54) can also be computed thus:

$$\begin{aligned}
(\vec{\boldsymbol{\varphi}}_K)_i = \int_K f(\boldsymbol{x}) b^i_K(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} &= \int_{\widehat{K}} f(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}}))\,\widehat{b}^i(\widehat{\boldsymbol{x}})\,|\det \mathbf{F}_K|\,\mathrm{d}\widehat{\boldsymbol{x}} \\
&= |\det \mathbf{F}_K| \sum_{l=1}^{P} \widehat{\omega}_l\, f(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\xi}}_l))\,\widehat{b}^i(\widehat{\boldsymbol{\xi}}_l).
\end{aligned} \tag{68}$$

See Code listing 6 for its MATLAB implementation.

So, now there is only the first term (the element diffusion matrix) in (59) left. Let's zero in on it. This time we will write $f(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}}))$ as the pullback form $(\boldsymbol{\Phi}_K^* f)(\widehat{\boldsymbol{x}})$, then

$$\begin{aligned}&\int_K \boldsymbol{\alpha}(\boldsymbol{x})\mathbf{grad}\, b_K^i(\boldsymbol{x})\cdot\mathbf{grad}\, b_K^j(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\\ =&\int_{\widehat{K}} (\boldsymbol{\Phi}_K^*\boldsymbol{\alpha})(\widehat{\boldsymbol{x}})(\underbrace{\boldsymbol{\Phi}_K^*(\mathbf{grad}\, b_K^i)}_{=?})(\widehat{\boldsymbol{x}})\cdot(\underbrace{\boldsymbol{\Phi}_K^*(\mathbf{grad}\, b_K^j)}_{=?})(\widehat{\boldsymbol{x}})\,|\det \mathrm{D}\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}})|\,\mathrm{d}\widehat{\boldsymbol{x}}.\end{aligned} \tag{69}$$

The vexing problem is how do we compute the $=?$ parts in (69)? The local shape functions b_K^i, (affine) linear functions though, are sort of elusive. One may think of using the brute-force way of doing it, that is to calculate the analytic formulas (planes) of the local shape functions on the general triangle K, and consequently their gradients can be computed. This sounds not that scary, huh? Well, but actually we don't have to do so. The following lemma leads to a cleaner and faster solution.

Lemma 4.3 (Transformation formula for gradients [Hip21, Lemma 2.8.3.10])**.** *For differentiable $u : K \mapsto \mathbb{R}$ and any diffeomorphism $\boldsymbol{\Phi} : \widehat{K} \mapsto K$ we have*

$$(\mathbf{grad}_{\widehat{\boldsymbol{x}}}(\boldsymbol{\Phi}^* u))(\widehat{\boldsymbol{x}}) = (\mathrm{D}\boldsymbol{\Phi}(\widehat{\boldsymbol{x}}))^\top \underbrace{(\mathbf{grad}_{\boldsymbol{x}} u)(\boldsymbol{\Phi}(\widehat{\boldsymbol{x}}))}_{=\boldsymbol{\Phi}^*(\mathbf{grad}\, u)(\widehat{\boldsymbol{x}})} \quad \forall \widehat{\boldsymbol{x}} \in \widehat{K}.$$

Proof. By the *chain rule* the components of the gradient vector become

$$(\mathbf{grad}\, \boldsymbol{\Phi}^* u(\widehat{\boldsymbol{x}}))_i = \frac{\partial \boldsymbol{\Phi}^* u}{\partial \widehat{x}_i}(\widehat{\boldsymbol{x}}) = \frac{\partial}{\partial \widehat{x}_i} u(\boldsymbol{\Phi}(\widehat{\boldsymbol{x}})) = \sum_{j=1}^d \frac{\partial u}{\partial x_j}(\boldsymbol{\Phi}(\widehat{\boldsymbol{x}}))\frac{\partial \boldsymbol{\Phi}_j}{\partial \widehat{x}_i}(\widehat{\boldsymbol{x}}),$$

then in the vector form we have

$$\begin{bmatrix}\frac{\partial \boldsymbol{\Phi}^* u}{\partial \widehat{x}_i}(\widehat{\boldsymbol{x}})\\ \vdots \\ \frac{\partial \boldsymbol{\Phi}^* u}{\partial \widehat{x}_d}(\widehat{\boldsymbol{x}})\end{bmatrix} = (\mathbf{grad}_{\widehat{\boldsymbol{x}}}(\boldsymbol{\Phi}^* u))(\widehat{\boldsymbol{x}}) = (\mathrm{D}\boldsymbol{\Phi}(\widehat{\boldsymbol{x}}))^\top \begin{bmatrix}\frac{\partial u}{\partial x_i}(\boldsymbol{\Phi}(\widehat{\boldsymbol{x}}))\\ \vdots \\ \frac{\partial u}{\partial x_d}(\boldsymbol{\Phi}(\widehat{\boldsymbol{x}}))\end{bmatrix} = (\mathrm{D}\boldsymbol{\Phi}(\widehat{\boldsymbol{x}}))^\top (\mathbf{grad}_{\boldsymbol{x}} u)(\boldsymbol{\Phi}(\widehat{\boldsymbol{x}})).$$

Here, $\mathrm{D}\boldsymbol{\Phi}(\widehat{\boldsymbol{x}}) \in \mathbb{R}^{d,d}$ is the Jacobian of $\boldsymbol{\Phi}$ at $\widehat{\boldsymbol{x}} \in \widehat{K}$, the general formula for Jacobian matrix is

$$\mathrm{D}\boldsymbol{f}(\boldsymbol{x}) = \left[\frac{\partial f_i}{\partial x_j}\right]_{i,j=1}^{m,n} = \begin{bmatrix}\frac{\partial f_1}{\partial x_1}(\boldsymbol{x}) & \cdots & \frac{\partial f_1}{\partial x_n}(\boldsymbol{x})\\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1}(\boldsymbol{x}) & \cdots & \frac{\partial f_m}{\partial x_n}(\boldsymbol{x})\end{bmatrix} \in \mathbb{R}^{m,n}$$

□

Therefore, by using the above lemma we arrive at

$$\begin{aligned}&\int_K \boldsymbol{\alpha}(\boldsymbol{x})\mathbf{grad}\, b_K^i(\boldsymbol{x})\cdot\mathbf{grad}\, b_K^j(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\\ =&\int_{\widehat{K}} \boldsymbol{\alpha}(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}}))\left((\mathrm{D}\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}}))^{-\top}\mathbf{grad}_{\widehat{\boldsymbol{x}}}\widehat{b}^i(\widehat{\boldsymbol{x}})\right)\cdot\left((\mathrm{D}\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}}))^{-\top}\mathbf{grad}_{\widehat{\boldsymbol{x}}}\widehat{b}^j(\widehat{\boldsymbol{x}})\right)|\det \mathrm{D}\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}})|\,\mathrm{d}\widehat{\boldsymbol{x}}\\ =&\sum_{l=1}^P \widehat{\omega}_l\,\boldsymbol{\alpha}(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\xi}}_l))\left((\mathrm{D}\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\xi}}_l))^{-\top}\mathbf{grad}\,\widehat{b}^i(\widehat{\boldsymbol{\xi}}_l)\right)\cdot\left((\mathrm{D}\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\xi}}_l))^{-\top}\mathbf{grad}\,\widehat{b}^j(\widehat{\boldsymbol{\xi}}_l)\right)|\det \mathrm{D}\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\xi}}_l)|\end{aligned} \tag{70}$$

$$= |\det \mathbf{F}_K| \sum_{l=1}^P \widehat{\omega}_l\,\boldsymbol{\alpha}(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\xi}}_l))\left(\mathbf{F}_K^{-\top}\mathbf{grad}\,\widehat{b}^i(\widehat{\boldsymbol{\xi}}_l)\right)\cdot\left(\mathbf{F}_K^{-\top}\mathbf{grad}\,\widehat{b}^j(\widehat{\boldsymbol{\xi}}_l)\right). \tag{71}$$

Terrific! The formula (71) for the entries of the element diffusion matrix is totally manageable! And in the linear finite element method, the gradients of the reference shape functions $\widehat{b}^1, \widehat{b}^2, \widehat{b}^3$ are ever straightforward—they are constants which can be written as `[-1 -1;1 0;0 1]` in MATLAB. The MATLAB implementation for computing the element matrix (58, 59) is given in Code listing 4.

4.3 Assembly Algorithms

Now we present the generic assembly algorithms for the Galerkin matrix and the right-side vector with introducing an abstract "d.o.f. mapper/handler" facility `locglobmap`, defined as follows:

$$\texttt{locglobmap}(K, i) = j, \quad \text{if } b^j_{h|K} = b^i_K, \quad i = \{1, ..., Q(K)\}.$$

Note that according to Definition 4.1 every mesh entity K is also endowed with a set of local shape functions $\{b^1_K, ..., b^{Q(K)}_K\}$, not merely cells.

The `locglobmap` can be deemed as the generalized version of `dofh` introduced in (57), with a relationship by

$$\texttt{dofh}(k, l) = \texttt{locglobmap}(K, l), \quad \text{if } K \text{ has index } k, \quad l \in \{1, 2, 3\}.$$

The abstract algorithms are given below. Typically, on a triangular mesh $Q(K) \equiv 3$ holds for all mesh entities $K \in \mathcal{M}$.

Algorithm 3 Abstract assembly routine for finite element Galerkin matrices

procedure ASSEMBLEGALERKINMATRIX(Mesh $\mathcal{M}$)
  $\mathbf{A} = N \times N$ sparse matrix ▷ allocate zero sparse matrix
  for all $K \in \mathcal{M}$ **do** ▷ loop over all cells
    $Q(K) = \texttt{no\_loc\_shape\_functions}(K)$
    $\mathbf{A}_K = \texttt{getElementMatrix}(K)$ ▷ compute element matrix, see (49, 50, 61)/(67, 71)
    Vector $idx = \{\texttt{locglobmap}(K, 1), ..., \texttt{locglobmap}(K, Q(K))\}$ ▷ get global indices
    for $i = 1$ **to** $Q(K)$ **do**
      for $j = 1$ **to** $Q(K)$ **do**
        $\mathbf{A}(idx(i), idx(j)) \mathrel{+}= \mathbf{A}_K(i, j)$ ▷ see Figure 16
      end for
    end for
  end for
  return A
end procedure

Algorithm 4 Generic assembly algorithm for finite element R.H.S. vectors

procedure ASSEMBLERHSVECTOR(Mesh $\mathcal{M}$)
  $\vec{\varphi} = \text{Vector}(N)$ ▷ preallocate appropriate memory
  for all $K \in \mathcal{M}$ **do** ▷ loop over all cells
    $Q(K) = \texttt{no\_loc\_shape\_functions}(K)$
    $\vec{\varphi}_K = \texttt{getElementVector}(K)$ ▷ compute element vector, see (56)/(68)
    Vector $idx = \{\texttt{locglobmap}(K, 1), ..., \texttt{locglobmap}(K, Q(K))\}$ ▷ get global indices
    for $i = 1$ **to** $Q(K)$ **do**
      $\vec{\varphi}(idx(i)) = \vec{\varphi}(idx(i)) + \vec{\varphi}_K(i)$ ▷ see Figure 18
    end for
  end for
  return $\vec{\varphi}$
end procedure

4.4 Incorporation of Boundary Conditions

Up to this point, we have finished the local computations of the element Galerkin matrix and the element vector as well as the assembly process. Thus there is only one thing left before we solve the linear system of equations, that is the imposition of boundary conditions. Here we will focus on two kinds of boundary conditions: Dirichlet boundary conditions and Neumann boundary conditions, the former of which are

also referred to as *essential* boundary conditions because they are directly imposed on trial space and (in homogeneous form) on test space, and the latter of which are also referred to as *natural* boundary conditions because they are enforced only through the variational equation (the Neumann boundary conditions "naturally" emerge when removing constraints on the boundary).

In Section 1 we considered the simplest homogeneous Neumann problem for the purpose of developing the Galerkin discretization strategy with relative ease. Now we first examine the non-homogeneous Neumann boundary value problem

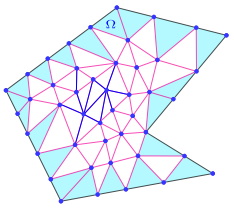
$$\begin{aligned} -\nabla \cdot (\boldsymbol{\alpha}(\boldsymbol{x})\nabla u) + \gamma(\boldsymbol{x})u &= f, \quad \text{in } \Omega, \\ \boldsymbol{\alpha}(\boldsymbol{x})\nabla u \cdot \boldsymbol{n} &= g_n, \quad \text{on } \partial\Omega. \end{aligned} \tag{72}$$

The corresponding variational formulation is

$$u \in H^1(\Omega): \int_\Omega (\boldsymbol{\alpha}(\boldsymbol{x})\nabla u \cdot \nabla v + \gamma(\boldsymbol{x})uv)\,\mathrm{d}\boldsymbol{x} = \int_\Omega fv\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} g_n v\,\mathrm{d}s \quad \forall v \in H^1(\Omega). \tag{73}$$

Substituting the test function v with (14) the right-hand side becomes

$$\int_\Omega fv_h\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} g_n v_h\,\mathrm{d}s = \sum_{i=1}^{N} \nu_i \left[\ell(b_h^i) + \int_{\partial\Omega} g_n b_h^i\,\mathrm{d}s\right]. \tag{74}$$



Hence, compared to the right-hand side vector derived in (15–20) we just need to add $\int_{\partial\Omega} g_n b_h^i\,\mathrm{d}s$ to each component of the right-hand side vector $\vec{\boldsymbol{\varphi}}$. Note that $\int_{\partial\Omega} g_n b_h^i\,\mathrm{d}s$ is non-zero only when the vertex $\boldsymbol{x}_i$ is on the boundary $\partial\Omega$, therefore only the additions to the components whose indices are the global node numbers of boundary vertices are required, in which cases the additions become

$$\Delta\vec{\boldsymbol{\varphi}}_i = \int_{\partial\Omega} g_n b_h^i\,\mathrm{d}s = \int_{e_1} + \int_{e_2} g_n b_h^i\,\mathrm{d}s, \quad e_1, e_2 \in \mathcal{E}_B(\mathcal{M}),\ e_1 \cap e_2 = \boldsymbol{x}_i \in \mathcal{V}_B(\mathcal{M}). \tag{75}$$

These contributions can also be computed by cell-oriented or rather (boundary) edge-oriented assembly on $\mathcal{M}_{|\partial\Omega}$, that is we traverse each boundary cell K/edge e and then distribute the cell contribution to the two vertices of the edge e. Note that on each boundary cell K only the two local shape functions whose nodal values are over the two vertices of the edge e do have contributions.

To implement this one must at least provide a facility or a mesh data structure that enables us to loop over the boundary edges of a particular type (Neumann, Dirichlet, Robin, etc). The Nektar++ framework which uses XML to specify the boundary conditions by tags and values [Can+15] is a great resource that we can learn from.

Next, consider the non-homogeneous Dirichlet boundary value problem

$$-\nabla \cdot (\boldsymbol{\alpha}(\boldsymbol{x})\nabla u) + \gamma(\boldsymbol{x})u = f, \quad \text{in } \Omega, \tag{76}$$

$$u = g_d, \quad \text{on } \partial\Omega, \tag{77}$$

with variational formulation

$$\begin{matrix} u \in H^1(\Omega) \\ u = g_d \text{ on } \partial\Omega \end{matrix} : \int_\Omega (\boldsymbol{\alpha}(\boldsymbol{x})\nabla u \cdot \nabla v + \gamma(\boldsymbol{x})uv)\,\mathrm{d}\boldsymbol{x} = \int_\Omega fv\,\mathrm{d}\boldsymbol{x} \quad \forall v \in H_0^1(\Omega). \tag{78}$$

A problem arises if we as before use $U = V$, since in this way $u \in U = V \subset H_0^1(\Omega)$ which conflicts with the non-homogeneous essential boundary condition (77).

Therefore, we introduce a small trick—offset functions, which can be used to convert (78) into a variational problem with the same trial and test space:

$$(78) \quad \Leftrightarrow \quad u = u_0 + w, \quad w \in H_0^1(\Omega): \ a(w, v) = \ell(v) - a(u_0, v) \quad \forall v \in H_0^1(\Omega), \tag{79}$$

with offset function $u_0 \in H^1(\Omega)$ satisfying

$$u_0 = g_d \quad \text{on } \partial\Omega. \tag{80}$$

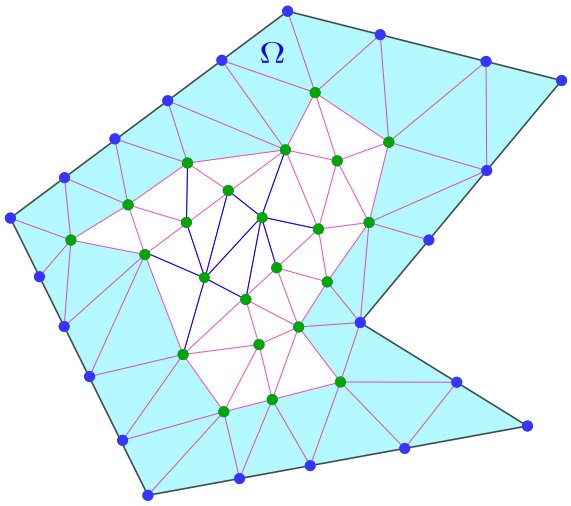
So, in order to obtain the Galerkin solution $u_h = u_0 + w_h$ all we need to do is find

$$w_h \in V_{0,h} := \mathcal{S}^0_{1,0}(\mathcal{M}): \ a(w_h, v_h) = \ell(v_h) - a(u_0, v_h) \quad \forall v_h \in V_{0,h}. \tag{81}$$

The finite element subspace $V_{0,h} := \mathcal{S}^0_{1,0}(\mathcal{M}) \subset H^1_0(\mathcal{M})$ is obtained by

$$\mathcal{S}^0_{1,0}(\mathcal{M}) := \mathcal{S}^0_1(\mathcal{M}) \cap H^1_0(\mathcal{M}) = \operatorname{span}\{b^j_h : \boldsymbol{x}_j \in \Omega \text{ (interior node)}\},$$

that is by dropping all those nodal basis functions (global shape functions) associated with nodes on $\partial\Omega$.



"Location" of nodal basis functions:
●, ● → nodal basis functions of $\mathcal{S}^0_1(\mathcal{M})$
● → nodal basis functions of $\mathcal{S}^0_{1,0}(\mathcal{M})$

■ → maximum support of offset function u_0:

$$\operatorname{supp}(u_0) \subset \bigcup \{K \in \mathcal{M} : \overline{K} \cap \partial\Omega \neq \varnothing\}.$$

We write

$$\begin{aligned} \mathfrak{B}_0 := \{b^1_h, ..., b^{N_0}_h\} \quad &\hat{=} \quad \text{nodal basis of } \mathcal{S}^0_{1,0}(\mathcal{M}) \\ &\quad\ \ \text{(tent functions associated with interior nodes),} \\ \mathfrak{B} := \mathfrak{B}_0 \cup \{b^{N_0+1}_h, ..., b^N_h\} \quad &\hat{=} \quad \text{nodal basis of } \mathcal{S}^0_1(\mathcal{M}) \\ &\quad\ \ \text{(extra basis functions associated with nodes} \in \partial\Omega\text{).} \end{aligned}$$

Here, $N := \sharp\mathcal{V}(\mathcal{M}) = \dim \mathcal{S}^0_1(\mathcal{M})$ (no. of total nodes),
$N_0 := \sharp\{\boldsymbol{x} \in \mathcal{V}(\mathcal{M}), \boldsymbol{x} \notin \partial\Omega\} = \dim \mathcal{S}^0_{1,0}(\mathcal{M})$ (no. of interior nodes).

Moreover, we denote

$$\begin{aligned} \mathbf{A}_0 \in \mathbb{R}^{N_0,N_0} \quad &\hat{=} \quad \text{Galerkin matrix for discrete trial/test space } \mathcal{S}^0_{1,0}(\mathcal{M}), \\ \mathbf{A} \in \mathbb{R}^{N,N} \quad &\hat{=} \quad \text{Galerkin matrix for discrete trial/test space } \mathcal{S}^0_1(\mathcal{M}). \end{aligned}$$

Then the Galerkin matrix $\mathbf{A}$ can be partitioned into four blocks:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_0 & \mathbf{A}_{0\partial} \\ \mathbf{A}^\top_{0\partial} & \mathbf{A}_{\partial\partial} \end{bmatrix}, \qquad \begin{aligned} \mathbf{A}_{0\partial} &:= \Big(a(b^j_h, b^i_h)\Big)_{\substack{i=1,...,N \\ j=N_0+1,...,N}} \in \mathbb{R}^{N_0, N-N_0}, \\ \mathbf{A}_{\partial\partial} &:= \Big(a(b^j_h, b^i_h)\Big)_{\substack{i=N_0+1,...,N \\ j=N_0+1,...,N}} \in \mathbb{R}^{N-N_0, N-N_0}. \end{aligned}$$

Thus if we ignore the essential boundary conditions and assemble the linear system of equations we can write $\mathbf{A}\vec{\boldsymbol{\mu}} = \vec{\boldsymbol{\varphi}}$ in the block-partitioned form:

$$\begin{bmatrix} \mathbf{A}_0 & \mathbf{A}_{0\partial} \\ \mathbf{A}^\top_{0\partial} & \mathbf{A}_{\partial\partial} \end{bmatrix} \begin{bmatrix} \vec{\boldsymbol{\mu}}_0 \\ \vec{\boldsymbol{\mu}}_\partial \end{bmatrix} = \begin{bmatrix} \vec{\boldsymbol{\varphi}}_0 \\ \vec{\boldsymbol{\varphi}}_\partial \end{bmatrix}. \tag{82}$$

Here, $\vec{\boldsymbol{\mu}}_0 \hat{=}$ coefficients for interior basis functions $b^1_h, ..., b^{N_0}_h$,
$\vec{\boldsymbol{\mu}}_\partial \hat{=}$ coefficients for basis functions $b^{N_0+1}_h, ..., b^N_h$ associated with nodes located on $\partial\Omega$.
Be aware that the coefficient vector $\vec{\boldsymbol{\mu}}$ amounts to the finite element approximation of u, therefore

$$\vec{\boldsymbol{\mu}}_\partial = \vec{\boldsymbol{\gamma}} := \text{values of } g_d \text{ at boundary nodes}, \tag{83}$$

which means we can compute $\vec{\boldsymbol{\mu}}_0$ by

$$\mathbf{A}_0\vec{\boldsymbol{\mu}}_0 = \vec{\boldsymbol{\varphi}}_0 - \mathbf{A}_{0\partial}\vec{\boldsymbol{\gamma}}. \tag{84}$$

We can also change the matrix and right-hand side of the linear system of equations $\mathbf{A}\vec{\boldsymbol{\mu}} = \vec{\boldsymbol{\varphi}}$ in a way such that it would transform (82) into

$$\begin{bmatrix} \mathbf{A}_0 & \mathbf{A}_{0\partial} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \vec{\boldsymbol{\mu}}_0 \\ \vec{\boldsymbol{\mu}}_\partial \end{bmatrix} = \begin{bmatrix} \vec{\boldsymbol{\varphi}}_0 \\ \vec{\boldsymbol{\gamma}} \end{bmatrix}. \tag{85}$$

Our initial aim is to obtain $\vec{\boldsymbol{\mu}}_0 = [\mu_1, ..., \mu_{N_0}]^\top$ since its components are the coefficients of the basis expansion of w_h in space $\mathcal{S}^0_{1,0}(\mathcal{M})$, i.e. $w_h = \sum_{i=1}^{N_0} \mu_i b_h^i$. But (85) suggests we can meanwhile obtain the whole approximate solution $\vec{\boldsymbol{\mu}}$ on the mesh $\mathcal{M}$.

One might ask why bother deriving these and why not just directly use (82) to compute our final solution for the Dirichlet boundary value problem (76, 77)? And thus we don't even need to make any modifications on the Galerkin matrix $\mathbf{A}$ and the right-hand side vector $\vec{\boldsymbol{\varphi}}$.

Yes, the solutions of (82) and (85) are exactly the same. The reason why we derived (85) and prefer it is because *in practice the fixed degrees of freedom (global shape/basis functions) corresponding to components of $\vec{\boldsymbol{\gamma}}$ are usually rather erratically dispersed among all DOFs, as opposed to being numbered in a way to yield a nice block-partitioning as in* (82).

If we let $\mathbf{A}_*$ represent the practical Galerkin matrix for the discrete trial/test space $\mathcal{S}^0_1(\mathcal{M})$, and the corresponding coefficient vector and right-hand side vector be $\vec{\boldsymbol{\mu}}_*$, $\vec{\boldsymbol{\varphi}}_*$ respectively, then we can show that the solution of the original linear system of equations $\mathbf{A}_*\vec{\boldsymbol{\mu}}_* = \vec{\boldsymbol{\varphi}}_*$ is different from the one of (82) and consequently different from that of (85) (otherwise why bother to take the Dirichlet boundary condition (77) into account!).

To verify this we only need to show that the transformation between the two linear systems of equations

$$\begin{bmatrix} \mathbf{A}_* & \vec{\boldsymbol{\mu}}_* & \vec{\boldsymbol{\varphi}}_* \end{bmatrix} \quad \rightarrow \quad \begin{bmatrix} \mathbf{A}_0 & \mathbf{A}_{0\partial} & \vec{\boldsymbol{\mu}}_0 & \vec{\boldsymbol{\varphi}}_0 \\ \mathbf{A}_{0\partial}^\top & \mathbf{A}_{\partial\partial} & \vec{\boldsymbol{\mu}}_\partial & \vec{\boldsymbol{\varphi}}_\partial \end{bmatrix}$$

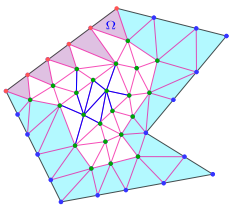
can not be obtained by simply switching rows.

This is actually very obvious: when we transform

$$\begin{bmatrix} \vec{\boldsymbol{\mu}}_* & \vec{\boldsymbol{\varphi}}_* \end{bmatrix} \quad \rightarrow \quad \begin{bmatrix} \vec{\boldsymbol{\mu}}_0 & \vec{\boldsymbol{\varphi}}_0 \\ \vec{\boldsymbol{\mu}}_\partial & \vec{\boldsymbol{\varphi}}_\partial \end{bmatrix}$$

by row-switching, all the interior DOFs of $\mathbf{A}_*$ are also transformed into the first N_0 rows, but in order to obtain $\mathbf{A}_0$ we still need some extra swapping operations on the columns of $\mathbf{A}_*$.

In practice, we leverage the scattered version of (85), that is, we first find the positions (global indices) of those boundary DOFs, then set all these rows of $\mathbf{A}_*$ to 0 with exceptions that in each of these rows set the entry whose column number is equal to its row number with value 1, and set these rows of $\vec{\boldsymbol{\varphi}}_*$ to its corresponding Dirichlet data $\vec{\boldsymbol{\gamma}}$.



Lastly, have a look at the mixed Neumann-Dirichlet problem

$$\begin{aligned} -\nabla\cdot(\boldsymbol{\alpha}(\boldsymbol{x})\nabla u) + \gamma(\boldsymbol{x})u &= f, \quad \text{in } \Omega, \\ u &= g_d, \quad \text{on } \Gamma \subset \Omega, \\ \boldsymbol{\alpha}(\boldsymbol{x})\nabla u\cdot\boldsymbol{n} &= g_n, \quad \text{on } \partial\Omega\setminus\Gamma, \end{aligned} \tag{86}$$

with variational formulation

$$\begin{matrix} u\in H^1(\Omega) \\ u = g_d \text{ on } \Gamma \end{matrix} \; : \; \int_\Omega (\boldsymbol{\alpha}(\boldsymbol{x})\nabla u\cdot\nabla v + \gamma(\boldsymbol{x})uv)\,\mathrm{d}\boldsymbol{x} = \int_\Omega fv\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega\setminus\Gamma} g_n v\,\mathrm{d}s \tag{87}$$

for all $v \in H^1(\Omega)$ with $v_{|\Gamma} = 0$.

Equipped with the knowledge learned in this subsection we just need to modify the integral domain in (75) with $\partial\Omega\setminus\Gamma$ and restrict the Dirichlet data $\vec{\boldsymbol{\gamma}}$ only to Γ, and then combine them both. Tada! Done! Everything runs in a similar manner as above.

4.5 Considerations for Higher Order Finite Elements

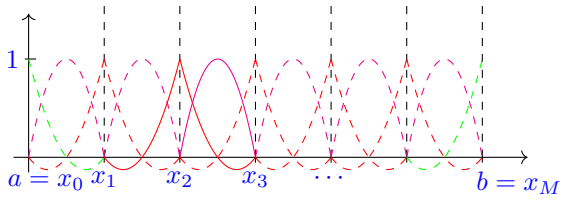
This subsection attempts to give some supplementary information about higher order finite elements, in particular, the quadratic Lagrangian finite elements will be considered under the context of two-point boundary value problems. (Disclaimer: We will only provide some basic ideas here, never meant to be rigorous or complet. It may also contain incorrekt information.)

Still, we consider problem P1. This time we will use (piecewise) quadratic functions to approximate the solution instead of linear functions as in Section 2.1.3. Similarly, we denote the piecewise quadratic function space by

$$\mathcal{S}^0_{2,0}(\mathcal{M}) := \left\{ \begin{array}{l} v \in C^0[a,b] : v_{|[x_{i-1},x_i]} \text{ quadratic}, \\ i = 1, ..., M, \ v(a) = v(b) = 0 \end{array} \right\},$$

$$N := \dim \mathcal{S}^0_{2,0}(\mathcal{M}) = 2M - 1.$$

We first give an visual of the quadratic Lagrangian interpolation (basis) functions:



⟵

The interpolation nodes are comprised of the interior nodes and midpoints, i.e.

$$\mathcal{N} := \mathcal{V}_I(\mathcal{M}) \cup \{\text{midpoints of intervals}\}.$$

Here the green parts are only for the space

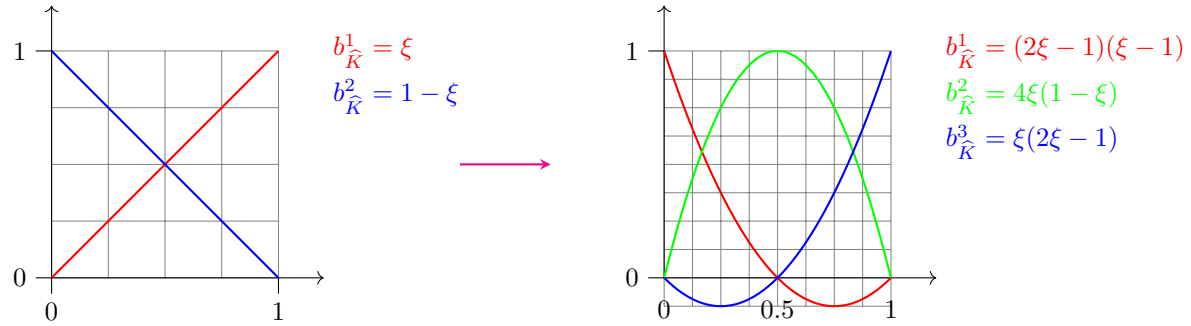
$$\mathcal{S}^0_2(\mathcal{M}) := \left\{ v \in C^0[a,b] : v_{|[x_{i-1},x_i]} \text{ quadratic}, \quad \forall i = 1, ..., M \right\},$$

$$N := \dim \mathcal{S}^0_2(\mathcal{M}) = 2M + 1.$$

The space $\mathcal{S}^0_2(\mathcal{M})$ will be used instead when dealing with non-homogeneous Dirichlet TPBVPs.

We can see that these basis functions satisfy the augmented nodal value property ($\cup$ nodal values at midpoints) and they can be split into two categories: one corresponds to the interior nodes and the other corresponds to the midpoints of those intervals. We denote them by $b^i_h, i = 1, ..., M-1$ and $b^{i+\frac{1}{2}}_h, i = 0, ..., M-1$ respectively. For example, in the above picture, the red solid graph represents b^2_h whose support spans two adjacent intervals and the magenta solid graph represents $b^{2\frac{1}{2}}_h$ whose support only spans one interval.

If we restrict our attention onto one interval, we can extract three local shape functions, whose standard versions (reference shape functions, on $[0,1]$) can be rather easily acquired and are shown below. For the sake of comparison we also plot the reference shape functions of 1st order on its left.



Since $b^i_h, i = 1, ..., M-1$ and $b^{i+\frac{1}{2}}_h, i = 0, ..., M-1$ are the basis functions of the quadratic finite element space $\mathcal{S}^0_{2,0}(\mathcal{M})$, then the Galerkin approximation $u_h \in U_h := \mathcal{S}^0_{2,0}(\mathcal{M})$ can be uniquely expressed in their linear combinations

$$u_h = \sum_{i=1}^{M-1} \mu_i b^i_h + \sum_{i=0}^{M-1} \mu_{i+\frac{1}{2}} b^{i+\frac{1}{2}}_h,$$

with

$$\mu_i = u_h(x_i), \quad \mu_{i+\frac{1}{2}} = u_h(x_{i+\frac{1}{2}}),$$

that is the basis expansion coefficients are given by the function values of u_h at these interpolation nodes.

The Galerkin matrix now becomes a $(2M-1)$-by-$(2M-1)$ matrix with entries

$$(\mathbf{A})_{ij} = a(b_h^j, b_h^i) = \int_a^b \left(p \frac{\mathrm{d}b_h^j}{\mathrm{d}x} \frac{\mathrm{d}b_h^i}{\mathrm{d}x} + q b_h^j b_h^i \right) \mathrm{d}x, \quad i,j = \frac{1}{2}, 1, ..., (M-1)\frac{1}{2}.$$

Of course, the fractional indices may sound a little odd. Therefore in practice we use the transformation

$$(i,j) \quad \rightarrow \quad (s,t) := (2i, 2j)$$

to locate the real positions of the entries.

The element matrix is defined on a cell $[x_{i-1}, x_i]$ by

$$\mathbf{A}_K := \left[a_K(b_K^j, b_K^l) \right]_{j,l=1}^3 = \left[\int_{x_{i-1}}^{x_i} \left(p \frac{\mathrm{d}b_K^j}{\mathrm{d}x} \frac{\mathrm{d}b_K^l}{\mathrm{d}x} + q b_K^j b_K^l \right) \mathrm{d}x \right]_{j,l=1}^3.$$

Doing the linear transformation $x = x_{i-1} + h_i t,\ h_i := x_i - x_{i-1}$ we can compute the entries of the element matrix by

$$\begin{aligned}
(\mathbf{A}_K)_{jl} &= \int_{x_{i-1}}^{x_i} \left(p \frac{\mathrm{d}b_K^j}{\mathrm{d}x} \frac{\mathrm{d}b_K^l}{\mathrm{d}x} + q b_K^j b_K^l \right) \mathrm{d}x \\
&= \int_0^1 p(x_{i-1} + h_i t) \frac{\mathrm{d}\overbrace{b_K^j(x_{i-1} + h_i t)}^{=\widehat{b}^j(t)}}{h_i\, \mathrm{d}t} \mathrm{d}\overbrace{b_K^l(x_{i-1} + h_i t)}^{=\widehat{b}^l(t)} + \\
&\quad \int_0^1 q(x_{i-1} + h_i t) \overbrace{b_K^j(x_{i-1} + h_i t)}^{=\widehat{b}^j(t)} \overbrace{b_K^l(x_{i-1} + h_i t)}^{=\widehat{b}^l(t)} h_i\, \mathrm{d}t \\
&= \frac{1}{h_i} \int_0^1 p(x_{i-1} + h_i t) \frac{\mathrm{d}\widehat{b}^j(t)}{\mathrm{d}t} \frac{\mathrm{d}\widehat{b}^l(t)}{\mathrm{d}t}\, \mathrm{d}t + h_i \int_0^1 q(x_{i-1} + h_i t)\, \widehat{b}^j(t)\, \widehat{b}^l(t)\, \mathrm{d}t.
\end{aligned}$$

Here $\widehat{b}^j(t) := b_{\widehat{K}}^j(t)$, $j = 1,2,3$ and hence

$$\frac{\mathrm{d}\widehat{b}^1(t)}{\mathrm{d}t} = 4t - 3, \qquad \frac{\mathrm{d}\widehat{b}^2(t)}{\mathrm{d}t} = 4 - 8t, \qquad \frac{\mathrm{d}\widehat{b}^3(t)}{\mathrm{d}t} = 4t - 1.$$

The element vector is defined on an interval $[x_{i-1}, x_i]$ by

$$\vec{\boldsymbol{\varphi}}_K := \left[\ell_K(b_K^j) \right]_{j=1}^3 = \left[\int_{x_{j-1}}^{x_i} f(x) b_K^j(x)\, \mathrm{d}x \right]_{j=1}^3,$$

with entries

$$(\vec{\boldsymbol{\varphi}}_K)_j = h_i \int_0^1 f(x_{i-1} + h_i t)\, \widehat{b}^j(t)\, \mathrm{d}t.$$

Once acquired the element matrices and element vectors we would like to assemble the linear system of equations. This process is illustrated in Figure 24.

We will apply a little trick here. First, it should be noticed that in the 1st cell and last cell there are only two local shape functions in them respectively, that is, b_K^1 (right half of b_h^0) is absent in $[x_0, x_1]$ and b_K^3 (left half of b_h^M) is absent in $[x_{M-1}, x_M]$. This means the 1st element matrix should remove all the entries concerning b_K^1 which are the 1st row and the 1st column, and the last element matrix should get rid of all the entries related to b_K^3 which are the 3rd row and the 3rd column. Next, we consider the assembly process in a way that it were in $\mathcal{S}_2^0(\mathcal{M})$ (pretend each cell has full/3 local shape functions), which will lead to a $(2M+1)$-by-$(2M+1)$ Galerkin matrix $\mathbf{A}$. The assembly is done by looping over all

cells (loop variable $j = 1, ..., M-1$) and putting the two adjacent element matrices (corresponding to the cell $[x_{j-1}, x_j]$ and cell $[x_j, x_{j+1}]$ respectively) together in a manner such that the a_{33} (lower right) entry of the former element matrix coincides with the a_{11} (upper left) entry of the latter element matrix. The value at this coinciding position is the sum of these two entries. After the loop we just need to extract and reassign the Galerkin matrix: `A` $\leftarrow$ `A(2:2M,2:2M)`.

The assembly of the RHS vector is done in the exactly same manner: by letting the 3rd entry of the former element vector coincide with the 1st entry of the latter element vector and then summing them up. Finally, let `phi` $\leftarrow$ `phi(2:2M)`.

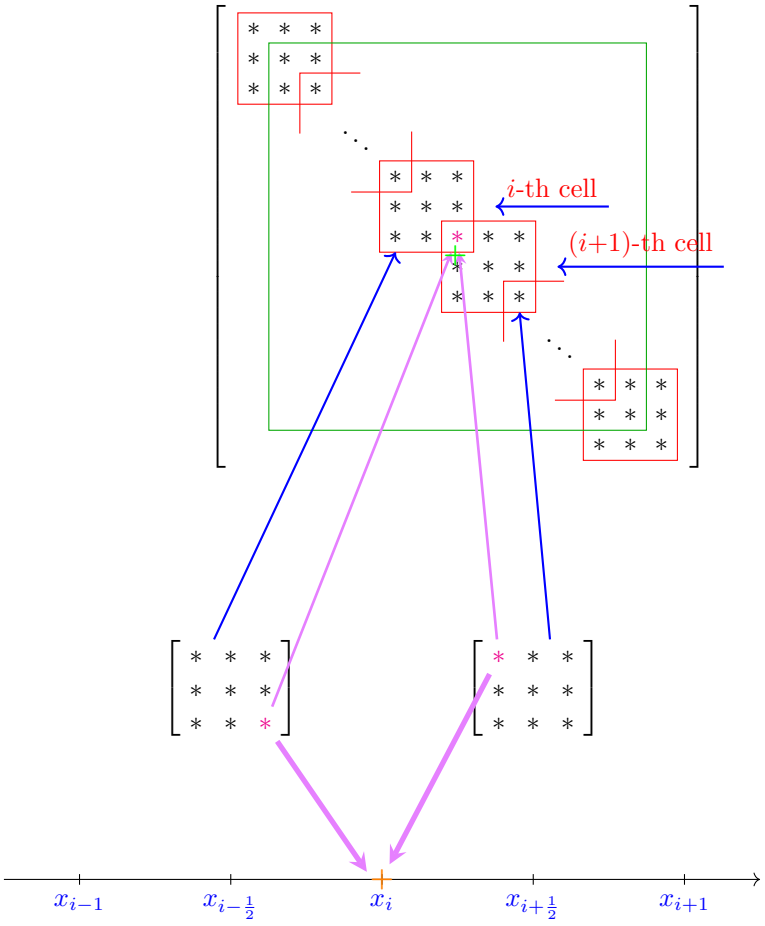


Figure 24: Assembly of quadratic FE Galerkin matrix

5 Numerical Experiments

In this section, we are going to give some numerical examples that use the Galerkin finite element method studied in previous sections to solve some boundary value problems. A few more examples are provided in Section 6.4 as comparisons to the weak Galerkin (WG) FEM.

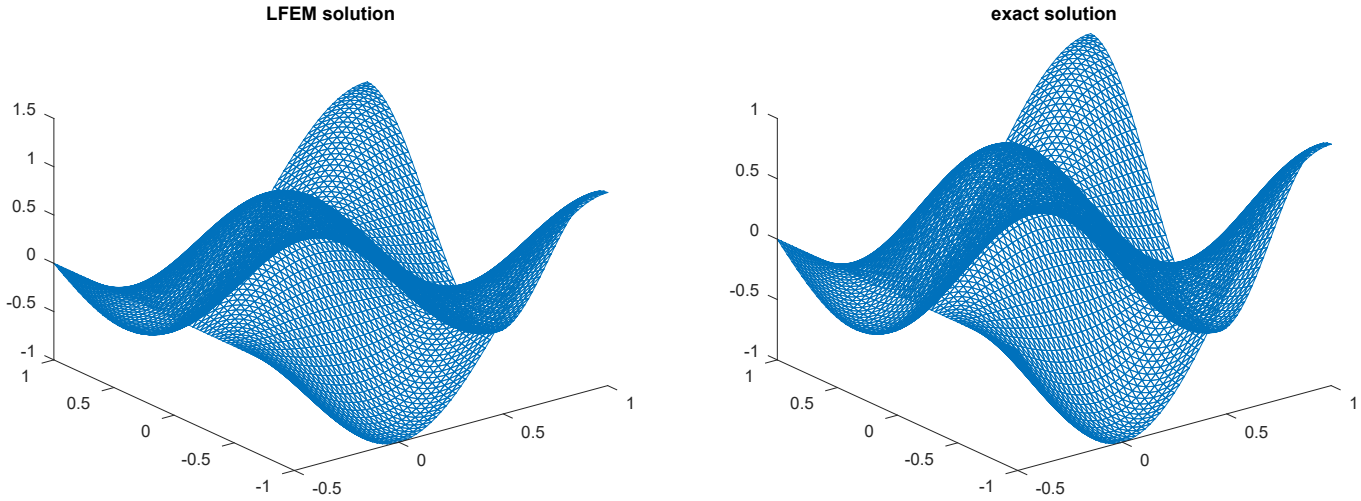
5.1 Example 1

Our first example is a mixed Neumann-Dirichlet boundary value problem:

$$\begin{aligned} -\Delta u &= 2\pi^2 \cos(\pi x)\cos(\pi y), && \text{in } \Omega \\ \nabla u \cdot \boldsymbol{n} &= \pi \sin(\pi x)\cos(\pi y), && \text{on } \Gamma \\ u &= \cos(\pi x)\cos(\pi y), && \text{on } \partial\Omega \setminus \Gamma, \end{aligned}$$

where $\Omega = \{(x,y)\,|\, -\frac{1}{2} < x < 1,\ -1 < y < 1\}$, $\Gamma = \{(x,y)\,|\, x = -\frac{1}{2},\ -1 \le y \le 1\}$.

Here, the analytic solution is $u = \cos(\pi x)\cos(\pi y)$. We used the Galerkin discretization scheme for linear finite elements developed in Section 2 to solve this BVP. We first give an visual of the solution:



One may find that the height of the two graphs seem to disagree. The fact is that the linear FEM solution has some values at some certain points exceed 1 which is the maximum value of the analytic solution and is represented in the right-hand side graph. The exceeding values are actually very small. The detailed discretization errors in various norms are shown in Table 2 below. By the way, if we adopt the general way of computing the right-hand side vector studied in Section 4.2, i.e. by (68) as opposed to (56), we can indeed gain the same looking of graph as the right-hand side one.

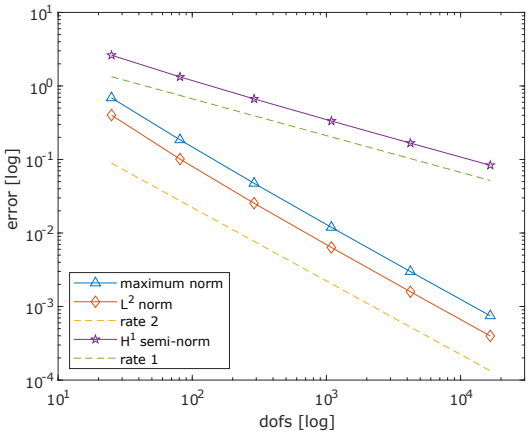


Figure 25: Convergence rates for Example 1

The visualization of these results in the table is represented in Figure 25. The MATLAB implementation for Example 1 is given in Code listing 8.

Table 2: Discretization errors and convergence rates in various norms for Example 1

N	$h_{\mathcal{M}} := N^{-\frac{1}{2}}$	$\|e_u\|_\infty$	order	$\|e_u\|_{L^2(\Omega)}$	order	$\|e_u\|_{H^1(\Omega)}$	order	$\|e_u\|_{H^1(\Omega)}$	order
25	0.2000	0.6884		0.4015		2.5890		2.6199	
81	0.1111	0.1852	1.8942	0.1011	1.9889	1.3244	0.9671	1.3282	0.9800
289	0.0588	0.0474	1.9646	0.0254	1.9931	0.6663	0.9911	0.6668	0.9942
1089	0.0303	0.0119	1.9904	0.0064	1.9979	0.3337	0.9976	0.3338	0.9984
4225	0.0154	0.0030	1.9975	0.0016	1.9994	0.1669	0.9994	0.1669	0.9996
16641	0.0078	0.0007	1.9994	0.0004	1.9999	0.0835	0.9998	0.0835	0.9999

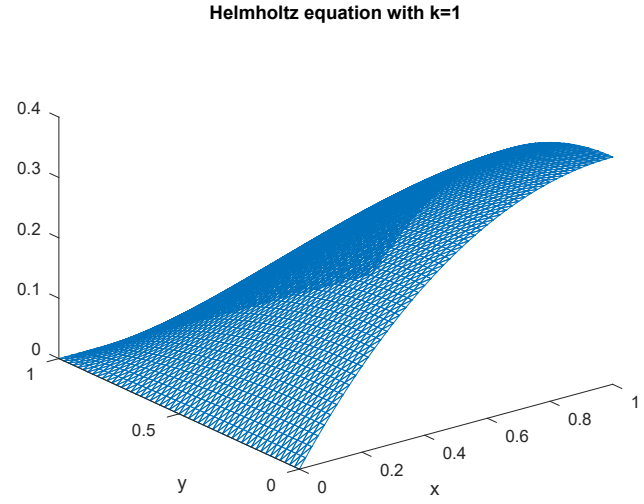
5.2 Example 2

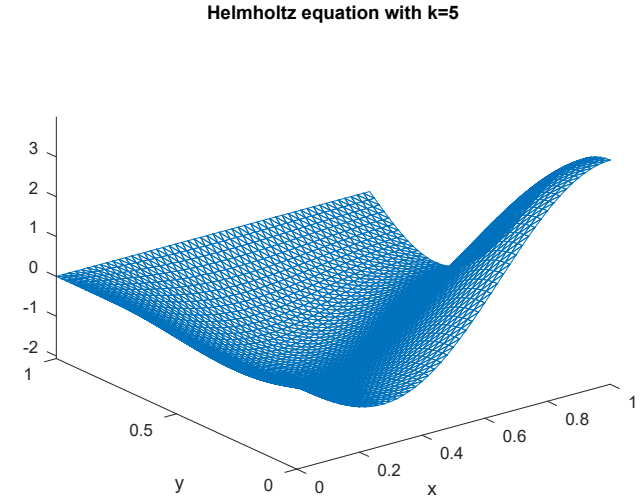
Consider the boundary value problem for Helmholtz equations:

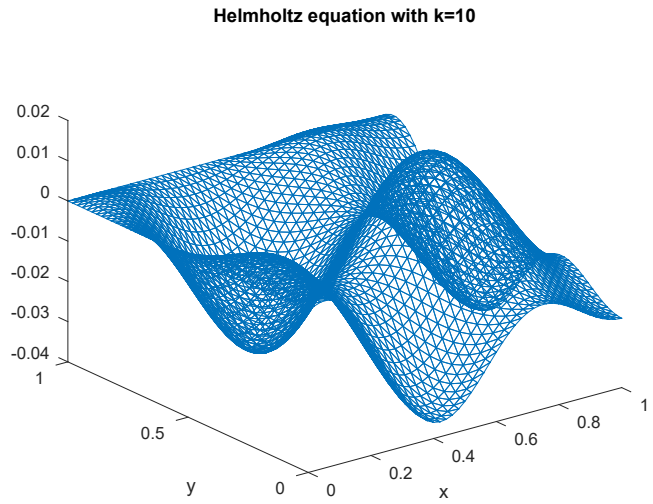
$$\begin{aligned} -\Delta u - k^2 u &= 1, \quad \text{in } G =]0,1[\times]0,1[, \\ u &= 0, \quad \text{on } \Gamma_1 = \{x = 0, 0 \le y \le 1\} \cup \{0 \le x \le 1, y = 1\}, \\ \nabla u \cdot \boldsymbol{n} &= 0, \quad \text{on } \Gamma_2 = \{0 \le x \le 1, y = 0\} \cup \{x = 1, 0 \le y \le 1\}, \end{aligned}$$

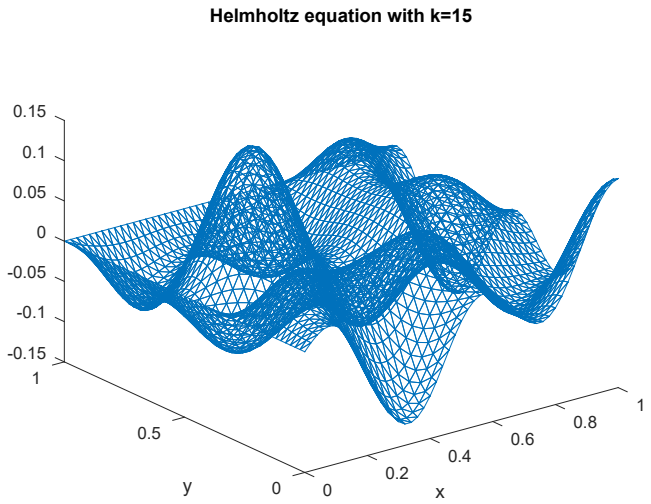
where $k = 1, 5, 10, 15, 20, 25$.

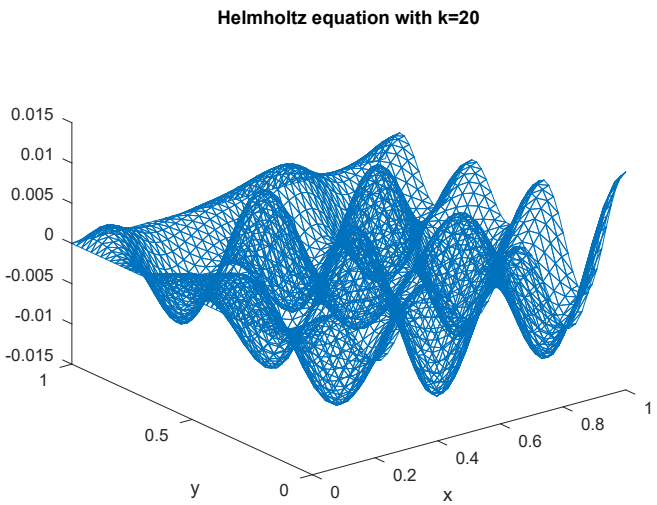
The implementation for Example 2 is given in Code listing 9 and the results are as follows:

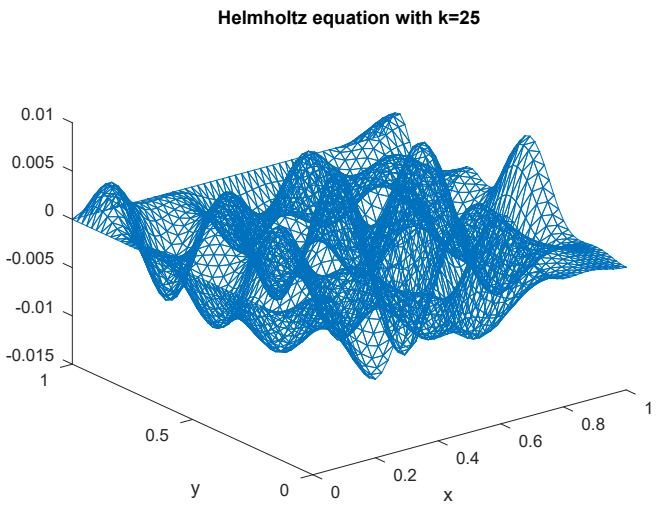












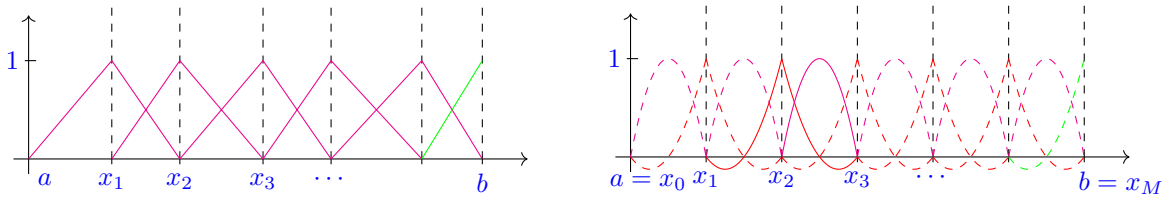
5.3 Examples for TPBVP

In this subsection we will have a look at the mixed two-point boundary value problem:

$$-\frac{\mathrm{d}}{\mathrm{d}x}\left(p(x)\frac{\mathrm{d}u}{\mathrm{d}x}\right)+q(x)u=f(x),\quad x\in(a,b),$$
$$u(a)=0,\ u'(b)=0,$$

where $p(x)\in C^1(\bar{I}),\ p(x)\geq p_{min}>0,\ q(x)\in C^1(\bar{I}),\ q(x)\geq 0,\ f(x)\in L^2(I),\ I=]a,b[$.

The basis functions of 1st and 2nd order for the above mixed boundary value problem now become something like these:



The overall procedures are similar to what we have discussed in Section 2.2.1 and Section 4.5 for the 1st order and 2nd order respectively. For example, in the quadratic element version, we just need to let `A ← A(2:2M+1,2:2M+1)`.

We will consider two test cases both of which set $p(x)=1$, $q(x)=0$, $[a,b]=[0,3]$.

Test case 1:

$$f=\frac{\pi^2}{4}\sin(\frac{\pi}{2}x),\quad \text{with } u=\sin(\frac{\pi}{2}x).$$

Test case 2:

$$f=\pi^2\sin(\pi x),\quad \text{with } u=\sin(\pi x)+\pi x.$$

The approximations are given in Figure 26 and the graphs concerning the convergence rates of 1st order and 2nd order for Test case 1 are represented in Figure 27.

We can conclude from Figure 27 that the convergence rates for *linear* finite elements are

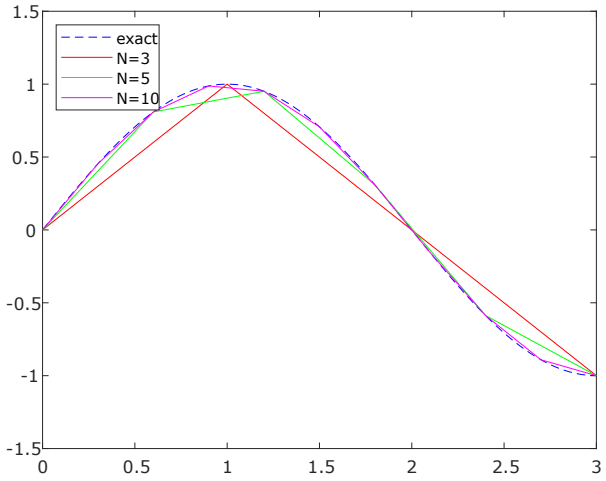
$$L^\infty \text{ norm}: O(h^2),\qquad L^2 \text{ norm}: O(h^2),\qquad H^1 \text{ norm}: O(h),$$

and for *quadratic* finite elements are

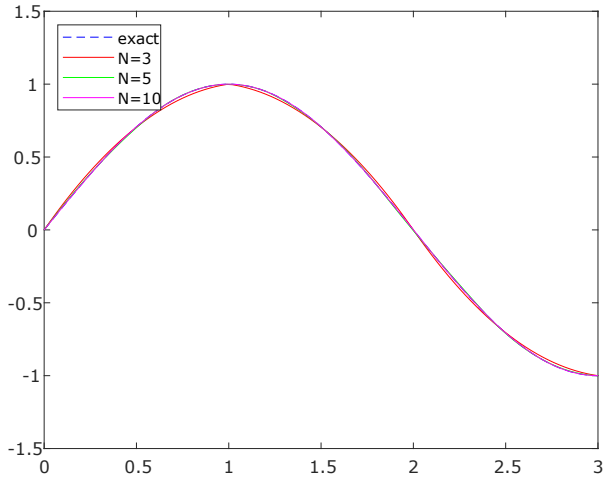
$$L^\infty \text{ norm}: O(h^3),\qquad L^2 \text{ norm}: O(h^3),\qquad H^1 \text{ norm}: O(h^2).$$

These results agree with the theory, that is for p-th order finite elements:

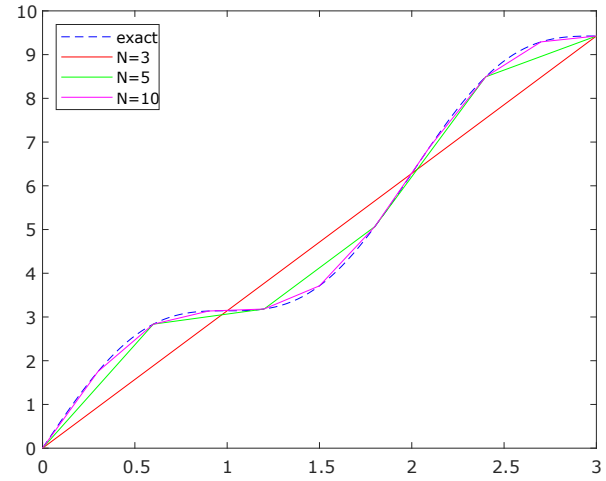
$$L^\infty \text{ norm}: O(h^{p+1}),\qquad L^2 \text{ norm}: O(h^{p+1}),\qquad H^1 \text{ norm}: O(h^p).$$



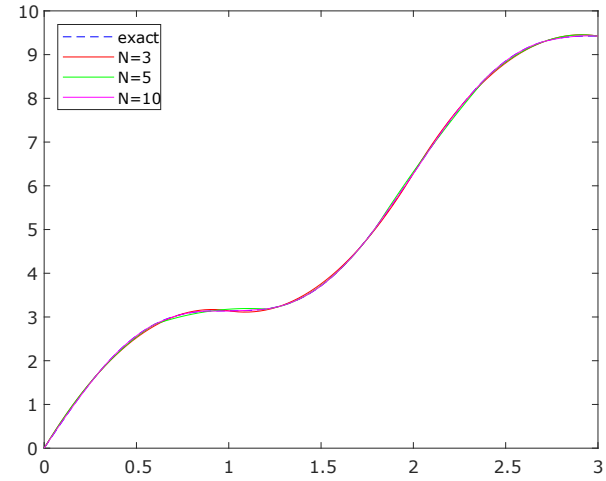
(a) Test case 1 by LFEM



(b) Test case 1 by QFEM

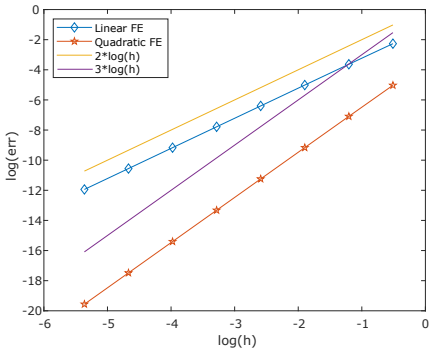


(c) Test case 2 by LFEM

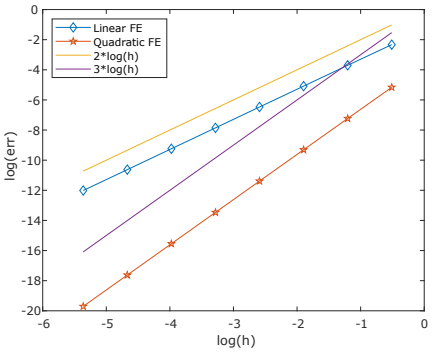


(d) Test case 2 by QFEM

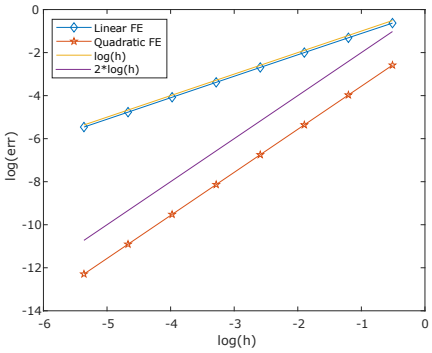
Figure 26: Approximations via Linear FEM & Quadratic FEM



(a) Convergence in L^∞ norm



(b) Convergence in L^2 norm



(c) Convergence in H^1 norm

Figure 27: Convergence rates for Test case 1

6 Further Reading—A Weak Galerkin FEM

In the end, as a further reading, we introduce a more advanced technique—the weak Galerkin method. It was first introduced and analyzed by Junping Wang and Xiu Ye in their paper [WY13] for solving second-order elliptic problems. The weak Galerkin is an original finite element method based on variational principles for weak (generalized) functions and the weak gradients defined thereon.

To illustrate the main ideas of weak Galerkin, the model problem we will consider here is the Dirichlet problem for 2nd-order elliptic equations

$$-\nabla\cdot(\mathcal{A}\nabla u) = f, \quad \text{in } \Omega, \tag{88}$$

$$u = g, \quad \text{on } \partial\Omega, \tag{89}$$

where Ω is an open bounded domain in $\mathbb{R}^d$ (polygonal for $d=2$, polyhedral for $d=3$) and $\mathcal{A}$ is a symmetric uniformly positive definite $d\times d$ matrix-valued function.

6.1 Weak Gradients and Discrete Weak Gradients

As the weak Galerkin draws its strength from introducing the weak gradient operator, we shall first elaborate on the definitions.

Let T be any polytopal domain with boundary ∂T. A weak function on the region T refers to a function $v=\{v_0,v_b\}$ satisfying $v_0\in L^2(T)$ and $v_b\in L^2(\partial T)$. One can interpret the 1st component v_0 as the value of $v\in T$, and the 2nd component v_b as the value of v on ∂T. We point out that v_b may not necessarily be associated with the trace of v_0 on ∂T. We denote the space of weak functions on T by $W(T)$, that is,

$$W(T)=\{v=\{v_0,v_b\}:\ v_0\in L^2(T),\ v_b\in L^2(\partial T)\}.$$

For ease of writing, we use the notations $(v,w)_D \hat{=} \int_D vw\,\mathrm{d}\boldsymbol{x}$ with $D\in\mathbb{R}^d$ and $\langle v,w\rangle_\gamma \hat{=} \int_\gamma vw\,\mathrm{d}s$ with $\gamma\in\mathbb{R}^{d-1}$.

Definition 6.1. $\forall v\in W(T)$, we define the weak gradient of v as a linear functional $\nabla_w v$ in the dual space of $H^1(T)$ whose action on each $q\in[H^1(T)]^d$ is given by

$$\langle\nabla_w v,q\rangle_T := -(v_0,\nabla\cdot q)_T + \langle v_b, q\cdot\boldsymbol{n}\rangle_{\partial T}. \tag{90}$$

Here, $\boldsymbol{n}$ is the outward normal direction to ∂T.

Note that for any $v\in W(T)$, the right-hand side of (90) defines a bounded linear functional on the normed linear space $H^1(T)$. Therefore, the weak gradient $\nabla_w v$ is well defined. In addition, if the component of v are restrictions of a function $u\in H^1(T)$ in T and on ∂T, respectively, then we arrive at

$$-(v_0,\nabla\cdot q)_T + \langle v_b, q\cdot\boldsymbol{n}\rangle_{\partial T} = -(u,\nabla\cdot q)_T + \langle u, q\cdot\boldsymbol{n}\rangle_{\partial T} = -(\nabla u\cdot q,1)_T,$$

which suggests $\nabla_w v=\nabla u$, i.e. the strong gradient of u.

In another view, we can the embed Sobolev space $H^1(T)$ into the space $W(T)$ through an inclusion map: $H^1(T)\mapsto W(T)$ which is defined by

$$i_w(\phi)=\{\phi|_T,\phi|_{\partial T}\},\quad \phi\in H^1(T).$$

Aided by the inclusion map i_w, one can view the Sobolev space $H^1(T)$ as a subspace of $W(T)$ via distinguishing each $\phi\in H^1(T)$ with $i_w(\phi)$. Similarly, a weak function $v=\{v_0,v_b\}\in W(T)$ is said to be in $H^1(T)$ if it can be recognized by a function $\phi\in H^1(T)$ through $i_w(\phi)$. This, again, suggests that the weak gradient amounts to the classical gradient (that is, $\nabla_w v=\nabla u$) for smooth functions $v\in H^1(T)$.

The discrete weak gradient operator is introduced in the sense such that ∇_w is approximated in a polynomial subspace of the dual $[H^1(T)]^d$. We will adopt the notation $[P_r(T)]^d$ to mean the set of polynomials on T with degree $\le r$.

Definition 6.2. The discrete weak gradient operator, denoted by $\nabla_{w,r,T}$, is defined as the unique polynomial $(\nabla_{w,r,T}v)\in[P_r(T)]^d$ that meets

$$(\nabla_{w,r,T}v,q)_T := -(v_0,\nabla\cdot q) + \langle v_b, q\cdot\boldsymbol{n}\rangle_{\partial T},\quad q\in[P_r(T)]^d. \tag{91}$$

Using integration by parts to the first term on the right-hand side of (91), one can rephrase (91) in the following form

$$(\nabla_{w,r,T} v, q) = (\nabla v_0, q)_T + \langle v_b - v_0, q \cdot \boldsymbol{n} \rangle_{\partial T}, \quad q \in [P_r(T)]^d. \tag{92}$$

Remark 2. We point out that in the design of numerical methods for partial differential equations, the classical gradient operator $\nabla = (\partial_{x_1}, \partial_{x_2})$ should be employed to functions with some certain degree of smoothness. For instance, in the standard Galerkin FEM, continuous piecewise polynomials over a prescribed finite element partition is often implied on such a "smoothness". By introducing the weak gradient operator, derivatives can be taken for functions *without any continuity* across cells on a mesh (amazing). In this way, the notion of weak gradient *permits the use of generalized functions* in approximation.

6.2 Weak Galerkin Finite Element Schemes

Denote by $\mathcal{T}_h$ a partition of the domain Ω comprising polygons in 2D or polyhedrons in 3D qualified with a set of rules as specified in [WY14]. We denote the set of all edges or flat faces in $\mathcal{T}_h$ by $\mathcal{E}_h$. The set of all interior edges or faces is represented by $\mathcal{E}_h^0 = \mathcal{E}_h \setminus \partial\Omega$. $\forall T \in \mathcal{T}_h$, its diameter is denoted by h_T, and the mesh size is denoted by $h = \max_{T \in \mathcal{T}_h} h_T$ for $\mathcal{T}_h$. We assume the mesh is quasi-uniform, that is to say there exists a constant satisfying $h \leq C h_T$, $\forall T \in \mathcal{T}_h$.

For a given integer $k \geq 1$, denote by V_h the weak Galerkin finite element space concerning T_h defined by

$$V_h = \{v = \{v0, v_b\} : \, v_0|_T \in P_k(T), \, v_b|e \in P_{k-1}(e), \, e \subset \partial T, \, T \in \mathcal{T}_h\} \tag{93}$$

and

$$V_h^0 = \{v : \, v \in V_h, \, v_b = 0 \text{ on } \partial\Omega\}. \tag{94}$$

It should be emphasized that for any function $v \in V_h$ there exists a single value v_b on each edge $e \in \mathcal{E}_h$.

For every element $T \in \mathcal{T}_h$, we denote the L^2 projection from $L^2(T)$ to $P_k(T)$ by Q_0 and denote the L^2 projection from $L^2(e)$ to $P_{k-1}(e)$ by Q_b. Denote by $\mathbb{Q}_h$ the L^2 projection from $[L^2(T)]^d$ to the discrete gradient space $[P_{k-1}(T)]^d$. Let $V = H^1(\Omega)$. A projection operator is defined by $Q_h : V \mapsto V_h$ satisfying

$$Q_h v = \{Q_0 v_0, Q_b v_b\}, \quad \{v_0, v_b\} = i_w(v) \in W(T), \quad T \in \mathcal{T}_h \tag{95}$$

We denote the discrete weak gradient operator on the finite element space V_h by $\nabla_w, k-1$, then we can compute it by using (91) on each element T, that is

$$(\nabla_{w,k-1} v)|_T = \nabla_{w,k-1,T}(v|_T), \quad v \in V_h.$$

We will drop the subscript $k-1$ in $\nabla_{w,k-1}$ for the discrete weak gradient from now on for the sake of simplicity.

Consider two forms on V_h:

$$a(v, w) = \sum_{T \in \mathcal{T}_h} (a \nabla_w v, \nabla_w w)_T,$$

$$s(v, w) = \rho \sum_{T \in \mathcal{T}_h} h_T^{-1} \langle Q_b v_0 - v_b, Q_b w_0 - w_b \rangle_{\partial T},$$

where $\rho > 0$. We will take $\rho = 1$ throughout this section for the sake of simplicity. We denote a stabilization $a_s(\cdot, \cdot)$ of $a(\cdot, \cdot)$ with

$$a_s(v, w) = a(v, w) + s(v, w).$$

Weak Galerkin Algorithm 1. We are assigned to find $u_h = \{u_0, u_b\} \in V_h$ such that $u_b = Q_b g$ on $\partial\Omega$ and satisfying:

$$a_s(u_h, v) = (f, v_0), \quad \forall v = \{v_0, v_b\} \in V_h^0. \tag{96}$$

To justify the well-posedness of the scheme (96), we let

$$|\!|\!| v |\!|\!| := \sqrt{a_s(v, v)} \quad \forall v \in V_h. \tag{97}$$

A semi-norm in V_h is defined by the functional $\|\cdot\|$. Moreover, a norm in V_h^0 is also defined. To confirm this, all we need to do is check the positivity property for $\|\cdot\|$. In the end, we suppose $v \in V_h^0$ and $\|v\| = 0$. It follows that

$$(a\nabla_w v, \nabla_w v) + \sum_{T\in\mathcal{T}_h} h_T^{-1}\langle Q_b v_0 - v_b, Q_b v_0 - v_b\rangle_{\partial T} = 0,$$

which suggests that $\nabla_w v = 0$ for every element T and $Q_b v_0 = v_b$ on ∂T. Then by $\nabla_w v = 0$ and (92) we have $\forall q \in [P_{k-1}(T)]^d$

$$\begin{aligned} 0 &= (\nabla_w v, q)_T \\ &= (\nabla v_0, q)_T - \langle v_0 - v_b, q\cdot \boldsymbol{n}\rangle_{\partial T} \\ &= (\nabla v_0, q)_T - \langle Q_b v_0 - v_b, q\cdot \boldsymbol{n}\rangle_{\partial T} \\ &= (\nabla v_0, q)_T. \end{aligned}$$

By letting $q = \nabla v_0$ in the above equation gives $\nabla v_0 = 0$ on $T \in \mathcal{T}_h$. Therefore, $v_0 = const \quad \forall T \in \mathcal{T}_h$. This, alone with the fact that $Q_b v_0 = v_b$ on ∂T and $v_b = 0$ on $\partial\Omega$, suggests $v_0 = v_b = 0$.

Lemma 6.1. *The weak Galerkin finite element scheme (96) has one and only one solution.*

Proof. Obviously, we just need to shown the uniqueness. Let $u_h^{(1)}$ and $u_h^{(2)}$ be two solutions of (96), it follows that $e_h = u_h^{(1)} - u_h^{(2)}$ satisfies

$$a_s(e_h, v) = 0, \quad \forall v \in V_h^0.$$

Note that $e_h \in V_h^0$, then substituting v with e_h in the above equation gives

$$\|e_h\|^2 = a_s(e_h, e_h) = 0.$$

This suggests $e_h \equiv 0$, or equivalently, $u_h^{(1)} \equiv u_h^{(1)}$. This yields the assertion of the lemma. □

6.3 Error Analysis for Weak Galerkin

The overall error analysis for the weak Galerkin is rather complicated, so we will simply give some conclusions. For more information about the proofs, one can refer to [WY15, Section 5, 6].

Error Equation

Let $u_h = \{u_o, u_b\} \in V_h$ be the weak Galerkin finite element solution arising from the numerical scheme (96). Assume u the exact solution of (88, 89). The L^2 projection of u in the finite element space V_h is given by

$$Q_h u = \{Q_0 u, Q_b u\}.$$

Let

$$e_h = \{e_0, e_b\} = \{Q_0 u - u_0, Q_b u - u_b\}$$

be the error between the WG finite element solution and the L^2 projection of the exact solution.

Lemma 6.2. *Let e_h be the error of the weak Galerkin finite element solution arising from (96). Then we have*

$$a_s(e_h, v) = \ell_u(v) + s(Q_h u, v), \quad \forall v \in V_h^0, \tag{98}$$

where $\ell_u(v) = \sum_{T\in\mathcal{T}_h}\langle a(\nabla u - \mathbb{Q}_h\nabla u)\cdot\boldsymbol{n}, v_0 - v_b\rangle_{\partial T}$.

Error Estimates

Theorem 6.3. *Let $u_h \in V_h$ be the weak Galerkin finite element solution of the model problem (88, 89) arising from (96). Assume the exact solution $u \in H^{k+1}(\Omega)$. Then, there exists a constant C such that*

$$|||u_h - Q_h u||| \leq Ch^K \|u\|_{k+1}. \tag{99}$$

Corollary 6.4. *Let $u_h \in V_h$ be the weak Galerkin finite element solution of the model problem (88, 89) arising from (96). Assume the exact solution $u \in H^{k+1}(\Omega)$. Then, there exists a constant C such that*

$$\|u - u_h\|_{1,h} \leq Ch^k \|u\|_{k+1}. \tag{100}$$

Consider the dual problem

$$\begin{cases} \text{Find } \Phi \in H_1^0(\Omega) \text{ such that} \\ -\nabla \cdot (a\nabla\Phi) = e_0 \quad \text{in } \Omega. \end{cases} \tag{101}$$

The usual H^2-regularity is assumed, which means there exists a Constant C such that

$$\|\Phi\|_2 \leq C\|e_0\|. \tag{102}$$

Theorem 6.5. *Let $u_h \in V_h$ be the weak Galerkin finite element solution of problem (88, 89) arising from (96). Assume the exact solution $u \in H^{k+1}(\Omega)$. Moreover, assume the dual problem (101) has the usual H^2-regularity. Then, there exists a constant C such that*

$$\|u - u_0\| \leq Ch^{k+1} \|u\|_{k+1}. \tag{103}$$

6.4 Comparison to Standard FEM

In this subsection, we will check out differences of the quality of approximation between the weak Galerkin finite element method and the standard Galerkin FEM developed in previous sections by a few numerical experiments. For simplicity, the domain is set with $\Omega =]0,1[\times]0,1[$ for *all* the comparison examples below, and in the rest of this subsection we will just call these two methods WG and SG. The MATLAB implementation for all these comparison examples by SG is given in Code listing 10.

For the weak Galerkin, the error is defined by $e_h = u_h - Q_h u = \{e_0, e_b\}$, where $e_0 = u_0 - Q_0 u$ and $e_b = u_b - Q_b u$. Here $Q_h u = \{Q_0 u, Q_b u\}$ with Q_h as the L^2 projection onto approximately defined spaces.

Comparison Example 1

The first comparison example is set with

$$\mathcal{A} = \begin{bmatrix} x^2 + y^2 + 1 & xy \\ xy & x^2 + y^2 + 1 \end{bmatrix}, \quad \text{with } u = \sin(\pi x)\cos(\pi y).$$

To match the equations, the Dirichlet boundary function g and source function f are thus set to be

$$\begin{aligned} f = &-\pi[3x\cos(\pi x)\cos(\pi y) - 2\pi xy\cos(\pi x)\sin(\pi y) - \\ &2\pi(x^2 + y^2 + 1)\sin(\pi x)\cos(\pi y) - 3y\sin(\pi x)\sin(\pi y)], \end{aligned}$$

and

$$g = \begin{cases} 0, & x = 0 \text{ or } 1,\ 0 \leq y \leq 1, \\ sin(\pi x) & y = 0,\ 0 \leq x \leq 1, \\ -sin(\pi x) & y = 1,\ 0 \leq x \leq 1, \end{cases} . \tag{104}$$

The corresponding numerical results by WG and SG are shown in Table 3 and Table 4, respectively.

Table 3: Comparison example 1 by WG

h	$\|\|\|e_h\|\|\|$	Order	$\|e_0\|$	Order
1/4	1.3240e+00		1.5784e+00	
1/8	6.6333e−01	9.9710e-01	3.6890e−01	2.0972
1/16	3.3182e−01	9.9933e-01	9.0622e−02	2.0253
1/32	1.6593e−01	9.9983e-01	2.2556e−02	2.0064
1/64	8.2966e−02	9.9998e-01	5.6326e−03	2.0016
1/128	4.1483e−02	1.0000	1.4078e−03	2.0004

Table 4: Comparison example 1 by standard FEM

$h_{\mathcal{M}} := n^{-1}$	$\|e_u\|_\infty$	order	$\|e_u\|_{L^2(\Omega)}$	order	$\|e_u\|_{H^1(\Omega)}$	order	$\|\|e_u\|\|_{H^1(\Omega)}$	order
1/4	0.0293		0.0644		0.8442		0.8466	
1/8	0.0081	1.8620	0.0176	1.8739	0.4325	0.9648	0.4329	0.9678
1/16	0.0021	1.9284	0.0045	1.9657	0.2176	0.9908	0.2177	0.9917
1/32	0.0005	1.9868	0.0011	1.9912	0.1090	0.9977	0.1090	0.9979
1/64	0.0001	1.9978	0.0003	1.9978	0.0545	0.9994	0.0545	0.9995
1/128	0.0000	1.9989	0.0001	1.9994	0.0273	0.9999	0.0273	0.9999
$O(h^r), r =$		1.9599		1.9713		0.9921		0.9928

Comparison Example 2

The Poisson problem is considered in our second example:

$$-\Delta u = f$$

for the PDE model as in (88). Still, we use the exact solution $u = \sin(\pi x)\cos(\pi y)$ as the first example did. Hence the source function f becomes

$$f = 2\pi^2 \sin(\pi x)\cos(\pi y)$$

and boundary function g is given by (104) as before.

The results for Comp Ex 2 by WG and SG are shown in Table 5 and Table 6, respectively.

The next two comparison examples are taken from the paper [Mu+12].

Comparison Example 3

The 3rd comparison example we use here is a degenerated diffusion problem.

Consider

$$\begin{aligned} -\nabla \cdot (xy\nabla u) &= f, \quad \text{in } \Omega \\ u &= 0. \quad \text{on } \partial\Omega \end{aligned}$$

The exact solution is set to be $u = x(1-x)y(1-y)$ given that it vanishes on the boundary.

The interesting thing is the diffusion tensor $\mathcal{A} = xy$ is not uniformly positive definite (see the 2nd footnote on page 2), that is, we can't find such a positive number $\epsilon^- > 0$ such that

$$((xy)(z)) \cdot \boldsymbol{z} \geq \epsilon^- \|\boldsymbol{z}\|^2 \quad \forall \boldsymbol{z} \in \mathbb{R}^2$$

holds for almost all $(x, y) \in \Omega =]0, 1[\times]0, 1]$. Hence, the convergence rates in these norms by which we have usually measured are not specified. This applies both to WG and SG. So we cannot expect the orders of the convergent rates.

These numerical results are given in Table 7 and Table 8 for WG and SG, respectively.

Table 5: Comparison example 2 by WG

h	$\|\|\|e_h\|\|\|$	$\|e_h\|$	$\|e_h\|_{\mathcal{E}_h}$
1/2	2.7935e-01	6.1268e-01	5.7099e-02
1/4	1.4354e-01	1.5876e-01	1.3892e-02
1/8	7.2436e-02	4.0043e-02	3.5430e-03
1/16	3.6315e-02	1.0033e-02	8.9325e-04
1/32	1.8170e-02	2.5095e-03	2.2384e-04
1/64	9.0865e-03	6.2747e-04	5.5994e-05
1/128	4.5435e-03	1.5687e-04	1.4001e-05
1/256	2.2718e-03	3.9219e-05	3.5003e-06
$O(h^r), r =$	9.9388e-01	1.9931	1.9961

Table 6: Comparison example 2 by SG

$h_{\mathcal{M}} := n^{-1}$	$\|e_u\|_\infty$	order	$\|e_u\|_{L^2(\Omega)}$	order	$\|e_u\|_{H^1(\Omega)}$	order	$\|e_u\|_{H^1(\Omega)}$	order
1/4	0.0223		0.0636		0.8440		0.8464	
1/8	0.0059	1.9235	0.0172	1.8863	0.4325	0.9646	0.4328	0.9676
1/16	0.0015	1.9525	0.0044	1.9699	0.2176	0.9908	0.2177	0.9916
1/32	0.0004	1.9752	0.0011	1.9924	0.1090	0.9977	0.1090	0.9979
1/64	0.0001	1.9988	0.0003	1.9981	0.0545	0.9994	0.0545	0.9995
1/128	0.0000	1.9992	0.0001	1.9995	0.0273	0.9999	0.0273	0.9999
$O(h^r), r =$		1.9714		1.9744		0.9921		0.9928

Table 7: Comparison example 3 by WG

h	$\|\nabla_d e_h\|$	$\|e_0\|$	$\|e_b\|$	$\|\nabla_d u_h \nabla u\|$	$\|u_0 - u\|$	$\|e_0\|_\infty$
1/8	5.61e-02	3.32e-03	6.60e-03	5.75e-02	5.48e-03	1.27e-02
1/16	4.03e-02	1.38e-03	2.81e-03	4.09e-02	2.59e-03	4.90e-03
1/32	2.95e-02	5.68e-04	1.16e-03	2.96e-02	1.23e-03	2.21e-03
1/64	2.15e-02	2.35e-04	4.83e-04	2.15e-02	5.97e-04	1.16e-03
1/128	1.55e-02	9.93e-05	2.02e-04	1.55e-02	2.91e-04	5.99e-04
$O(h^r), r =$	0.4614	1.2687	1.2594	0.4697	1.0579	1.0912

Table 8: Comparison example 3 by SG

$h_{\mathcal{M}} := n^{-1}$	$\|e_u\|_\infty$	order	$\|e_u\|_{L^2(\Omega)}$	order	$\|e_u\|_{H^1(\Omega)}$	order	$\|e_u\|_{H^1(\Omega)}$	order
1/4	0.0043		0.0056		0.0590		0.0593	
1/8	0.0019	1.1499	0.0015	1.8432	0.0304	0.9557	0.0305	0.9602
1/16	0.0008	1.3176	0.0004	1.8922	0.0153	0.9909	0.0153	0.9922
1/32	0.0003	1.5415	0.0001	1.9172	0.0076	1.0011	0.0076	1.0015
1/64	0.0001	1.6451	0.0000	1.9340	0.0038	1.0028	0.0038	1.0029
1/128	0.0000	1.6871	0.0000	1.9460	0.0019	1.0022	0.0019	1.0022
$O(h^r), r =$		1.4789		1.9089		0.9928		0.9939

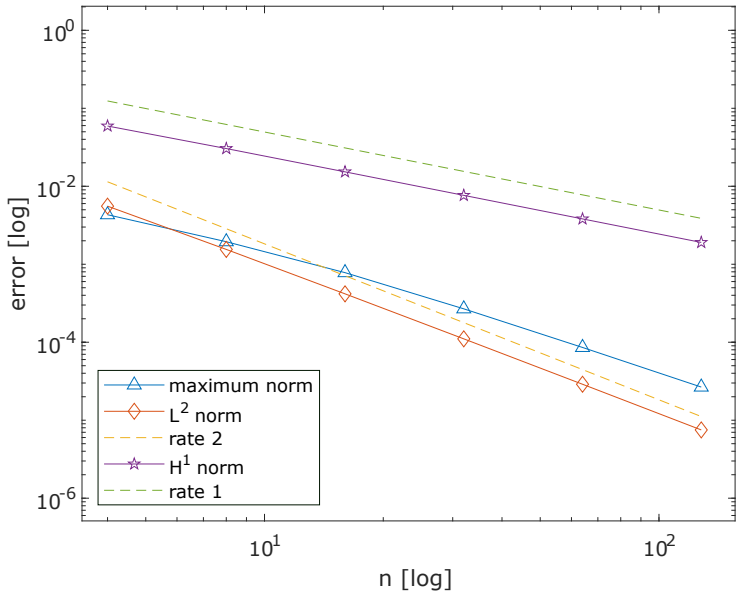


Figure 28: Comparison example 3 by SG

Comparison Example 4

The last (4th) comparison example is an anisotropic problem.

Consider

$$-\nabla \cdot (\mathcal{A}\nabla u) = f,$$

with the diffusion tensor

$$\mathcal{A} = \begin{bmatrix} k^2 & 0 \\ 0 & 1 \end{bmatrix} \quad \text{for } k \neq 0.$$

We set the analytic solution $u = \sin(2\pi x)\sin(2k\pi y)$. Thus the source function

$$f = 8k^2\pi^2 \sin(2\pi x)\sin(2k\pi y)$$

and Dirichlet boundary function

$$g = 0 \quad \text{on } \partial\Omega.$$

We will test two cases with $k = 3$ and $k = 9$ by WG and SG.

For $k = 3$, Table 9 and Table 10 represent the numerical results for WG and SG, respectively.

For $k = 9$, these results are shown in Table 11 and Table 12 for WG and SG, respectively.

From these results in Comp Ex 4 we can see that the SG attains its expected convergent orders (optimal convergence rates) in a rather slow pace compared to the WG.

Table 9: Comparison example 4 with $k = 3$ by WG

h	$\|\nabla_d e_h\|$	$\|e_0\|$	$\|e_b\|$	$\|\nabla_d u_h \nabla u\|$	$\|u_0 - u\|$	$\|e_0\|_\infty$
1/8	1.48e+00	1.95e-02	4.61e-02	2.70e+00	1.29e-01	4.13e-02
1/16	7.39e-01	5.11e-03	1.16e-02	1.35e+00	6.53e-02	1.06e-02
1/32	3.69e-01	1.29e-03	2.92e-03	6.80e-01	3.27e-02	2.67e-03
1/64	1.84e-01	3.24e-04	7.33e-04	3.40e-01	1.63e-02	6.68e-04
1/128	9.23e-02	8.12e-05	1.83e-04	1.70e-01	8.18e-03	1.66e-04
$O(h^r), r =$	0.0010	1.9793	1.9942	0.9972	0.9975	1.9906

Table 10: Comparison example 4 with $k = 3$ by SG

$h_{\mathcal{M}} := n^{-1}$	$\|e_u\|_\infty$	order	$\|e_u\|_{L^2(\Omega)}$	order	$\|e_u\|_{H^1(\Omega)}$	order	$\|e_u\|_{H^1(\Omega)}$	order
1/4	0.6727		0.4879		9.7186		9.7308	
1/8	0.2425	1.4719	0.2875	0.7629	6.8684	0.5008	6.8745	0.5013
1/16	0.0648	1.9050	0.0946	1.6046	3.7577	0.8701	3.7589	0.8709
1/32	0.0180	1.8447	0.0254	1.8967	1.9204	0.9684	1.9206	0.9688
1/64	0.0045	1.9951	0.0065	1.9739	0.9654	0.9922	0.9654	0.9923
1/128	0.0011	1.9988	0.0016	1.9935	0.4834	0.9981	0.4834	0.9981
$O(h^r), r =$		1.8616		1.6994		0.8888		0.8892

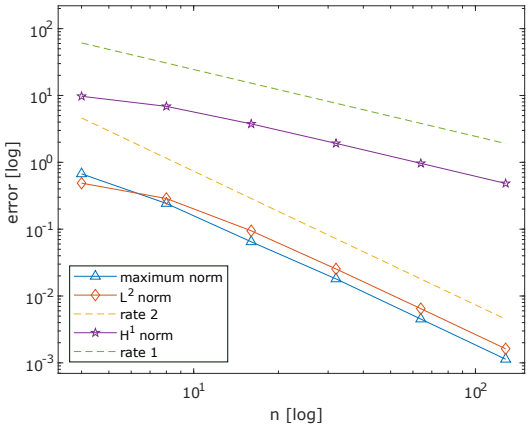


Figure 29: Comp Ex 4 with $k = 3$ by SG

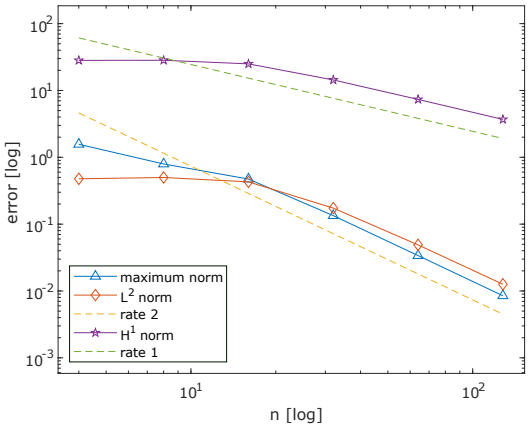


Figure 30: Comp Ex 4 with $k = 9$ by SG

Table 11: Comparison example 4 with $k = 9$ by WG

h	$\|\nabla_d e_h\|$	$\|e_0\|$	$\|e_b\|$	$\|\nabla_d u_h \nabla u\|$	$\|u_0 - u\|$	$\|e_0\|_\infty$
1/4	7.98e+00	6.80e−02	2.93e−01	1.58e+01	2.52e−01	1.49e−01
1/8	3.89e+00	2.07e−02	7.44e−02	8.18e+00	1.30e−01	4.22e−02
1/16	1.91e+00	5.43e−03	1.88e−02	4.12e+00	6.53e−02	1.09e−02
1/32	9.54e−01	1.37e−03	4.72e−03	2.06e+00	3.27e−02	2.74e−03
1/64	4.76e−01	3.44e−04	1.18e−03	1.03e+00	1.63e−02	6.84e−04
$O(h^r), r =$	1.0161	1.9160	1.9897	0.9857	0.9883	1.9492

Table 12: Comparison example 4 with $k = 9$ by SG

$h_{\mathcal{M}} := n^{-1}$	$\|e_u\|_\infty$	order	$\|e_u\|_{L^2(\Omega)}$	order	$\|e_u\|_{H^1(\Omega)}$	order	$\|e_u\|_{H^1(\Omega)}$	order
1/4	1.5663		0.4770		28.1833		28.1873	
1/8	0.7947	0.9788	0.4986	-0.0638	28.2982	-0.0059	28.3026	-0.0059
1/16	0.4728	0.7492	0.4292	0.2163	25.0052	0.1785	25.0089	0.1785
1/32	0.1341	1.8185	0.1738	1.3042	14.3673	0.7994	14.3683	0.7995
1/64	0.0339	1.9850	0.0489	1.8285	7.3446	0.9680	7.3448	0.9681
1/128	0.0085	1.9975	0.0126	1.9577	3.6889	0.9935	3.6890	0.9935
$O(h^r), r =$		1.5177		1.0733		0.6087		0.6088

7 Conclusion

In our thesis, we first derived the weak formulations of homogeneous Dirichlet two-point boundary value problems and in parallel we obtained the counterpart of 2nd-order elliptic BVPs. Then we managed to use the Galerkin discretization scheme to form our linear system of equations. Thus the problem turned into how to compute the element matrices (vectors) and assemble the final Galerkin matrix (RHS vector). We could do it in either vertex-centered or cell-oriented way. Vertex-centered assembly seemed to be direct to obtain the Galerkin matrix, but to find all the adjacent triangles to the vertex or edge requires a bit more effort or some complex data structures for storing the mesh information. Thus it was rather cumbersome. The cell-oriented assembly only need to distribute the cell contributions to the correct DOFs, hence we just need an index mapper. Assembly's done by looping over each cells and distributing the local contributions to the global DOFs. Cool! We also learned that why "assembly" became important even in 1D for TPBVP. For the linear finite elements it may seem trivial to assemble the 2-by-2 element stiffness matrix, but for higher order finite elements it really shines. This is because higher order elements requires many more interpolation points and thus giving rise to a wide area of span for a single DOF, which to it rather hard to directly compute each entry of the global stiffness matrix. Hence, rooted by the core idea of assembly and distribution acquired in 2D, we are now able to apply this idea to handle higher order finite elements in 1D with ease (The author was ever confused with why doing such a fuss to use element matrices and assemble them, we could be quick and dirty—just do it directly. That's because he had not got it. Considering higher order finite elements contributes to understanding the idea of assembly). In the end of the first part we performed a few numerical experiments and observed the optimal convergence rates, which verified the assertion of the theory.

In the second part we studied the definitions of weak gradients and discrete weak gradients. We found the charming property that them brought us: allowing approximating the solution with discontinuous piecewise polynomials across cells, thus permitting the use of generalized functions. Without the requirement of being "smooth", the weak Galerkin can be applied to a wide area of interests. We checked out the quality of approximation by WG and SG through a couple of numerical experiments, and found out that WG really kicks SG's ass in some cases. After all, it can be deemed as a generalized Galerkin method.

References

[Tur+56] M. J. Turner et al. "Stiffness and deflection of complex structures." In: *Journal of the Aeronautical Sciences* 23 (1956), pp. 805–824.

[GB05] Thomas Grätsch and Klaus-Jürgen Bathe. "A posteriori error estimation techniques in practical finite element analysis." In: *Computers and Structures* 83 (2005), pp. 235–265. DOI: 10.1016/j.compstruc.2004.08.011.

[Goc06] Mark S. Gockenbach. *Understanding and Implementing the Finite Element Method*. Society for Industrial and Applied Mathematics, 2006. Chap. 2.2.1-2.2.2, pp. 21–27. ISBN: 0-89871-614-4. URL: https://1lib.us/book/656138/15c86e.

[GR09] C. Geuzaine and J.-F. Remacle. "Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities." In: *International Journal for Numerical Methods in Engineering* 79 (11 2009), pp. 1309–1331. URL: http://gmsh.info/doc/preprints/gmsh_paper_preprint.pdf.

[BF10] Richard L. Burden and J. Douglas Faires. *Numerical Analysis*. 9th. Cengage Learning, 2010, p. 746.

[Li10] Ronghua Li. *Numerical Methods for Partial Differential Equations*. 2th. Higher Education Press, 2010.

[Mu+12] Lin Mu et al. "A COMPUTATIONAL STUDY OF THE WEAK GALERKINMETHOD FOR SECOND-ORDER ELLIPTIC EQUATIONS." In: (2012). URL: https://arxiv.org/abs/1111.0618.

[WY13] Junping Wang and Xiu Ye. "A weak Galerkin finite element method for second-order elliptic problems." In: *Journal of Computational and Applied Mathematics* 241 (2013), pp. 103–115. DOI: 10.1016/j.cam.2012.10.003.

[WY14] Junping Wang and Xiu Ye. "A weak Galerkin mixed finite element method for second-order elliptic problems." In: *Mathematics of Computation* 83.289 (2014), pp. 2101–2126. URL: https://arxiv.org/abs/1202.3655.

[Can+15] C.D. Cantwella et al. "Nektar++: An open-source spectral/hp element framework." In: *Computer Physics Communications* 192 (2015), pp. 205–219. DOI: 10.1016/j.cpc.2015.02.008.

[WY15] Junping Wang and Xiu Ye. "A weak Galerkin finite element method with polynomial reduction." In: *Journal of Computational and Applied Mathematics* 285 (2015), pp. 45–58. DOI: 10.1016/j.cam.2015.02.001.

[Hip21] Ralf Hiptmair. *Numerical Methods for Partial Differential Equations*. 2021. URL: https://www.sam.math.ethz.ch/~grsam/NUMPDEFL/NUMPDE.pdf.

Appendix

```
function [p,t]=generateMesh(x,y,n_x,n_y)
h_x=(x(2)-x(1))/n_x;
h_y=(y(2)-y(1))/n_y;
[X,Y] = meshgrid(x(1):h_x:x(2), y(1):h_y:y(2));
N=(n_x+1)*(n_y+1);
x=reshape(X,N,1);
y=reshape(Y,N,1);
p=[x,y];
t=delaunay(x,y);
end
```

Code listing 1: (Uniform) mesh generation on a rectangular region

```
function S=getArea(triangle)
% triangle is a 3x2 matrix
S=0.5*( (triangle(2,1)-triangle(1,1))* ...
        (triangle(3,2)-triangle(1,2))- ...
        (triangle(2,2)-triangle(1,2))* ...
        (triangle(3,1)-triangle(1,1)) );
end
```

Code listing 2: Compute the area of a triangle

```
% -Δu + ku = f, @para mass_coef = k
% ∫_Ω kuv dx is termed as mass matrix
function A = assembleMatrix(p, t, mass_coef)
if nargin==2
    mass_coef=0;
end
N=size(p,1);
nCells=size(t,1);
ii=zeros(9*nCells,1); % prealloacate memory
jj=ii;
vv=ii;
idx=0;
for k=1:nCells
    E=t(k,:); % triangle element by index
    Ak=getElementMatrix(p(E,:), mass_coef);
    for i=1:3
        for j=1:3
            idx=idx+1;
            ii(idx)=E(i); jj(idx)=E(j);
            vv(idx)=Ak(i,j);
        end
    end
end
A=sparse(ii,jj,vv,N,N);
end

function Ak = getElementMatrix(triangle, mass_coef)
Ak=ones(3);
Ak(:,2:3)=triangle;
Ak=Ak\eye(3); % inv(Ak)
Ak=Ak(2:3,:);
area=getArea(triangle);
Ak=area*(Ak.')*Ak;
if mass_coef ~= 0
    Ak=Ak+mass_coef*area/12*[2 1 1;1 2 1; 1 1 2];
end
end
```

Code listing 3: Assembly of the Galerkin matrix (stiffness matrix)

```
% −∇ · (α∇u) + γu = f
function A = assembleReactionDiffusionMatrix(p, t, alpha, gamma)
N=size(p,1);
nCells=size(t,1);
ii=zeros(9*nCells,1); % prealloacate memory
jj=ii;
vv=ii;
idx=0;
for k=1:nCells
    vidx=t(k,:); % global vertex indices of k-th cell
    Ak=getElementMatrix(p,vidx,alpha,gamma);
    for i=1:3
        for j=1:3
            idx=idx+1;
            ii(idx)=vidx(i); jj(idx)=vidx(j);
            vv(idx)=Ak(i,j);
        end
    end
end
A=sparse(ii,jj,vv,N,N);
end

function Ak = getElementMatrix(p, vidx, alpha, gamma)
% 6-point quadrature rule of order 4 on the unit triangle [0 0;1 0;0 1]
w = [1/60; 1/60; 1/60; 9/60; 9/60 ; 9/60];
x = [1/2 0; 1/2 1/2; 0 1/2; 1/6 1/6; 1/6 2/3; 2/3 1/6];
ref_shape_val = [1-x(:,1)-x(:,2), x(:,1), x(:,2)]; % values on quadrature points
grad_ref_shape = [-1 1 0;  % gradients of reference shape functions
                  -1 0 1];

bK = p(vidx(1),:);
BK = [p(vidx(2),:)-bK; p(vidx(3),:)-bK];
det_BK = abs(det(BK));
inv_BK = BK\eye(2); % inv(BK)

% transform quadrature points
y = x*BK+bK;

gamma_val = gamma(y(:,1),y(:,2));
Ak = zeros(3);
for i=1:6 % loop over quadrature points
    alpha_val = alpha(y(i,1),y(i,2)); % a 2-by-2 symmetric matrix or a scalar

    trf_grad = inv_BK*grad_ref_shape; % transformed gradients, 2-by-3
    Ak_alpha = (alpha_val*trf_grad).'*trf_grad;

    Ak_gamma = gamma_val(i)*(ref_shape_val(i,:).')*ref_shape_val(i,:);

    Ak = Ak+w(i)*(Ak_alpha+Ak_gamma)*det_BK;
end
end
```

Code listing 4: Assembly of diffusion and reaction matrix

```
function phi = assembleVector(p,t,f)
nCells=size(t,1);
phi=zeros(size(p,1),1);
for k=1:nCells
    E=t(k,:); % element by index
    phi(E)=phi(E)+getElementVector(p(E,:),f);
end
end

function phi_loc = getElementVector(triangle,f)
```

```
phi_loc=f(triangle(:,1),triangle(:,2));
phi_loc=getArea(triangle)/3*phi_loc;
end
```

Code listing 5: Assembly of the right-hand side vector (load vector)

```
function phi = assembleVectorByGaussQuad(p, t, f)
nCells = size(t,1);
phi = zeros(size(p,1),1);
for k = 1:nCells
    vidx = t(k,:); % global vertex indices of k-th cell
    phi(vidx) = phi(vidx)+getElementVector(p,vidx,f);
end
end

function phi_loc = getElementVector(p, vidx, f)
% 6-point quadrature rule of order 4 on the unit triangle [0 0;1 0;0 1]
w = [1/60; 1/60; 1/60; 9/60; 9/60 ; 9/60];
x = [1/2 0; 1/2 1/2; 0 1/2; 1/6 1/6; 1/6 2/3; 2/3 1/6];
ref_shape_val = [1-x(:,1)-x(:,2), x(:,1), x(:,2)]; % values on quadrature points

bK = p(vidx(1),:);
BK = [p(vidx(2),:)-bK; p(vidx(3),:)-bK];
det_BK = abs(det(BK));

% transform quadrature points
y = x*BK+bK;

f_val = f(y(:,1),y(:,2));
phi_loc = sum(w.*f_val.*ref_shape_val)*det_BK;
phi_loc = phi_loc.';
end
```

Code listing 6: Assembly of the RHS vector by Gaussian quadrature

```
function [L_inf_err,L2_err,H1_semi_err,H1_err] = errorEstimates(p,t,u,u_r,grad_u_r)

% 6-point quadrature rule of order 4 on the unit triangle [0 0;1 0;0 1]
w = [1/60; 1/60; 1/60; 9/60; 9/60 ; 9/60];
x = [1/2 0; 1/2 1/2; 0 1/2; 1/6 1/6; 1/6 2/3; 2/3 1/6];

nCells = size(t,1);
ref_shape_val = [1-x(:,1)-x(:,2), x(:,1), x(:,2)]; % values on quadrature points
grad_ref_shape = [-1 -1;1 0;0 1]; % gradients of reference shape functions

L_inf_err = max(abs(u_r(p(:,1),p(:,2))-u));
L2_err = 0; H1_semi_err = 0; H1_err = 0;
for k = 1 : nCells
    % We can also use `GaussTriaQuad()` to calculate the local L2 error:
    % update u_err = @(x,y) (u_r(x,y)-u_K(x,y)).^2 for each loop, where
    %        u_K   = @(x,y) "plane over triangle K with hights u(vidx)"
    % then
    %        loc_L2_err = GaussTriaQuad(triangle, u_err);

    vidx = t(k,:); % global vertex indices of k-th cell

    bK = p(vidx(1),:);
    BK = [p(vidx(2),:)-bK; p(vidx(3),:)-bK];
    det_BK = abs(det(BK));

    % transform quadrature points
    y = x*BK+bK;

    u_EX = u_r(y(:,1),y(:,2));
```

```
    grad_u_EX = grad_u_r(y(:,1),y(:,2));

    u_FE = u(vidx(1))*ref_shape_val(:,1)+u(vidx(2))*ref_shape_val(:,2)+ ...
                                        u(vidx(3))*ref_shape_val(:,3);

    grad_u_FE = (u(vidx(1))*grad_ref_shape(1,:)+ ...
                 u(vidx(2))*grad_ref_shape(2,:)+ ...
                 u(vidx(3))*grad_ref_shape(3,:))*(inv(BK)).';

    tmp = sum(w.*(u_EX-u_FE).^2)*det_BK;
    L2_err = L2_err+tmp;

    tmp2 = sum(w.*sum((grad_u_EX-grad_u_FE).^2,2))*det_BK;
    H1_semi_err = H1_semi_err+tmp2;
    H1_err = H1_err+tmp+tmp2;

end

L2_err = sqrt(L2_err);
H1_semi_err = sqrt(H1_semi_err);
H1_err = sqrt(H1_err); % = sqrt(L2_err^2+H1_semi_err^2)

end
```

Code listing 7: Error estimator

```
% u = cos(πx) cos(πy)
% −Δu = 2π^2 cos(πx) cos(πy), in Ω
% ∇u · n = π sin(πx) cos(πy), on Γ
% u = cos(πx) cos(πy), on ∂Ω \ Γ
% where Ω = {(x, y) | − 1/2 < x < 1, −1 < y < 1},
% Γ = {(x, y) | x = −1/2, −1 ≤ y ≤ 1}

N=6;
L_inf_err=zeros(N,1);
L2_err=zeros(N,1);
H1_semi_err=zeros(N,1);
H1_err=zeros(N,1);
dofs=zeros(N,1);
n=4;
for i=1:N
    [dofs(i),L_inf_err(i),L2_err(i),H1_semi_err(i),H1_err(i)]=run_example1(n);
    n=n*2;
end
L_inf_order=zeros(N-1,1);
L2_order=zeros(N-1,1);
H1_semi_order=zeros(N-1,1);
H1_order=zeros(N-1,1);
for i=1:N-1
    L_inf_order(i)=log2(L_inf_err(i)/L_inf_err(i+1));
    L2_order(i)=log2(L2_err(i)/L2_err(i+1));
    H1_semi_order(i)=log2(H1_semi_err(i)/H1_semi_err(i+1));
    H1_order(i)=log2(H1_err(i)/H1_err(i+1));
end
dofs
disp('L_inf_err   L2_err   H1_semi_err   H1_err')
disp([L_inf_err,L2_err,H1_semi_err,H1_err])
disp('L_inf_order   L2_order   H1_semi_order   H1_order')
disp([L_inf_order,L2_order,H1_semi_order,H1_order])

% plot
figure(3)
% L^2/L^∞ norm: ||u − u_h||_* = O(M^−2) = O(h_M^2) = O(N^−1)
% H^1 semi-norm: |u − u_h|_{H^1(Ω)} = O(M^−1) = O(h_M) = O(N^−1/2)
loglog(dofs,L_inf_err,'-^', ...
```

```
    dofs,L2_err,'-d', ...
    dofs,exp(-1*log(dofs)+0.8),'--', ...
    dofs,H1_semi_err,'-p', ... dofs,H1_err,'-h', ...
    dofs,exp(-1/2*log(dofs)+1.9),'--');
xlabel('dofs [log]'); xlim([10 3*10^4]);
ylabel('error [log]');
legend('maximum norm','L^2 norm','rate 2','H^1 semi-norm', ... 'H^1 norm',
    'rate 1','Location','SouthWest');
% title('Convergence Rates');
% Since L^2 norm errors are rather small when compared to H^1 semi-norm,
% H^1 semi-norm errors are almost equal to H^1 semi norm, so in the plot
% H^1 semi-norm and H^1 norm almost coincide if we draw them together.

%=========================================================================

function [dofs,L_inf_err,L2_err,H1_semi_err,H1_err] = run_example1(n)
x=[-1/2 1]; y=[-1 1];
% n_x=50; n_y=50;
n_x=n; n_y=n;

[p,t] = generateMesh(x,y,n_x,n_y);
dofs = size(p,1);

% alpha = @(x,y) 1; % alpha = @(x,y) 1*[1 0;0 1];
% gamma = @(x,y) x.*0+y.*0;
A = assembleMatrix(p,t); % A = assembleReactionDiffusionMatrix(p,t,alpha,gamma);
f = @(x,y) 2*pi^2*cos(pi*x).*cos(pi*y);
phi = assembleVector(p,t,f); % phi = assembleVectorByGaussQuad(p,t,f);

% process boundary conditions
nb_func = @(y) -pi*cos(pi*y);
phi = processNeumannBoundary(phi,p,nb_func);
db_func = @(x,y) cos(pi*x).*cos(pi*y);
[A,phi] = processDirichletBoundary(A,phi,p,db_func);

% solve the linear system of equations
u = A\phi;

u_r = @(x, y) cos(pi*x).*cos(pi*y);
grad_u_r = @(x, y) [-pi*sin(pi*x).*cos(pi*y),-pi*cos(pi*x).*sin(pi*y)];
[L_inf_err,L2_err,H1_semi_err,H1_err] = errorEstimates(p,t,u,u_r,grad_u_r);

% t is specified as a 4-by-Nt matrix in pdemesh, although we don't
% use the 4th row in a 2D mesh. To be compatible with 3D meshes?
t=[t';zeros(1,size(t,1))];
p=p'; % required to be 2-by-nP

figure(1)
pdemesh(p,0123,t,u) % 2nd arg `e` is unused, but has to be a matrix
title('LFEM solution')

figure(2)
pdemesh(p,0123,t,u_r(p(1,:),p(2,:)))
title('exact solution')

end

function phi = processNeumannBoundary(phi,p,nb_func)
% g = \nabla u \cdot n = \pi sin(\pi x) cos(\pi y)
% \int_\Gamma gv ds = \sum_i \int_{\Gamma_i} g (v_h^{s(i)} + v_h^{e(i)}) ds

pos = find(p(:,1)==-1/2);
edges = [pos(1:end-1),pos(2:end)];
nE = size(edges,1);
for i = 1:nE
    idx_s = edges(i,1); idx_e = edges(i,2);
```

```
    y_s = p(idx_s,2); y_e = p(idx_e,2);
    diff_y = y_e-y_s;
    phi(idx_s) = phi(idx_s)+ ...
        diff_y * Gaussquad(@(t)nb_func(y_s+t*diff_y).*t, 0, 1);
    phi(idx_e) = phi(idx_e)+ ...
        diff_y * Gaussquad(@(t)nb_func(y_s+t*diff_y).*(1-t), 0, 1);
end
end

function [A,phi] = processDirichletBoundary(A,phi,p,db_func)
db_pos = find(p(:,1)==1|p(:,2)==-1|p(:,2)==1);
A(db_pos,:) = 0;
A(db_pos,db_pos) = eye(size(db_pos,1));
phi(db_pos) = db_func(p(db_pos,1),p(db_pos,2));
end
```

Code listing 8: Example 1

```
% Exercise 2.7.2 in our text, p82
% -\Delta u - k^2 u = 1, in (0,1)\times(0,1),
% u = 0, on \Gamma_1 = {x=0, 0\le y\le 1}\cup{0\le x\le 1, y=1}
% \nabla u\cdot n = 0, on \Gamma_2 = {0\le x\le 1, y=0}\cup{x=1, 0\le y\le 1}

for k = [1,5,10,15,20,25]
    run_exercise_2_7_2(k)
end

%=====================================================================

function run_exercise_2_7_2(k)
x=[0 1]; y=[0 1];
n_x=50; n_y=50;

[p,t] = generateMesh(x,y,n_x,n_y);

A = assembleMatrix(p,t,-k^2);
f = @(x,y) 1+x.*0+y.*0;
phi = assembleVector(p,t,f);

% process Dirichlet boundary condition
db_func = @(x,y) x.*0+y.*0;
[A,phi] = processDirichletBoundary(A,phi,p,db_func);

% solve the linear systems of equations
u = A\phi;

t=[t';zeros(1,size(t,1))];
p=p'; % required to be 2-by-nP

figure
pdemesh(p,0123,t,u) % 2nd arg `e` is unused, but has to be a matrix
title(['Helmholtz equation with k=',num2str(k)])
xlabel('x')
ylabel('y')
fig2svg(['../svg/Helmholtz_k=',num2str(k),'.svg']);

end

function [A,phi] = processDirichletBoundary(A,phi,p,db_func)
db_pos = find(p(:,1)==0|p(:,2)==1);
A(db_pos,:) = 0;
A(db_pos,db_pos) = eye(size(db_pos,1));
phi(db_pos) = db_func(p(db_pos,1),p(db_pos,2));
end
```

Code listing 9: Exercise 2.7.2 in our text [Li10]

```
% FEM for Elliptic BVP
% -∇·(α∇u)+γu = f, in Ω,
% u = g_d, on ∂Ω,
% where Ω = (0,1)×(0,1).

u_case = 1; % <--- choose which one to run
N=6;
n=4; % h(M) := n^-1
n_arr = 2.^(2:N+1).';
L_inf_err=zeros(N,1);
L2_err=zeros(N,1);
H1_semi_err=zeros(N,1);
H1_err=zeros(N,1);
for i=1:N
    [~,L_inf_err(i),L2_err(i),H1_semi_err(i),H1_err(i)]=run_ellbvp(n,u_case);
    n=n*2;
end
L_inf_order=zeros(N-1,1);
L2_order=zeros(N-1,1);
H1_semi_order=zeros(N-1,1);
H1_order=zeros(N-1,1);
for i=1:N-1
    L_inf_order(i)=log2(L_inf_err(i)/L_inf_err(i+1));
    L2_order(i)=log2(L2_err(i)/L2_err(i+1));
    H1_semi_order(i)=log2(H1_semi_err(i)/H1_semi_err(i+1));
    H1_order(i)=log2(H1_err(i)/H1_err(i+1));
end
p_L_inf = polyfit(log2(n_arr),log2(L_inf_err),1);
p_L2 = polyfit(log2(n_arr),log2(L2_err),1);
p_H1_semi = polyfit(log2(n_arr),log2(H1_semi_err),1);
p_H1 = polyfit(log2(n_arr),log2(H1_err),1);
n_arr
disp('L_inf_err   L2_err   H1_semi_err   H1_err')
disp([L_inf_err,L2_err,H1_semi_err,H1_err])
disp('L_inf_order   L2_order   H1_semi_order   H1_order')
disp([L_inf_order,L2_order,H1_semi_order,H1_order])
disp('Orders by least square approximation')
disp(-[p_L_inf(1),p_L2(1),p_H1_semi(1),p_H1(1)])

% plot
figure(3)
b = [1.9,0.8; -0.7,-1.7; 5.5,4.3];
% L^2/L^∞ norm: ||u-u_h||_* = O(h_M^2) = O(n^-2)
% H^1 semi-norm: |u-u_h|_{H^1(Ω)} = O(h_M) = O(n^-1)
loglog(n_arr,L_inf_err,'-^', ...
    n_arr,L2_err,'-d', ...
    n_arr,exp(-2*log(n_arr)+b(u_case,2)),'--', ... % adjust the coef b in exp to
    n_arr,H1_err,'-p', ...                         % make the ref line work better
    n_arr,exp(-1*log(n_arr)+b(u_case,1)),'--');
xlabel('n [log]');
ylabel('error [log]');
legend('maximum norm','L^2 norm','rate 2','H^1 norm', ...
    'rate 1','Location','SouthWest');
% title('Convergence Rates');

%==========================================================================

function [dofs,L_inf_err,L2_err,H1_semi_err,H1_err] = run_ellbvp(n, u_case)
x=[0 1]; y=[0 1];
% n_x=50; n_y=50;
n_x=n; n_y=n;

[p,t] = generateMesh(x,y,n_x,n_y);
dofs = size(p,1);
```

```
65
66  if u_case == 1
67      u_r = @(x, y) sin(pi*x).*cos(pi*y);
68      grad_u_r = @(x, y) [pi*cos(pi*x).*cos(pi*y), -pi*sin(pi*x).*sin(pi*y)];
69      alpha = @(x,y)[x.^2+y.^2+1,x.*y; x.*y,x.^2+y.^2+1];
70      gamma = @(x,y) 0.*x.*y;
71      f = @(x,y) -pi*( 3*x.*cos(pi*x).*cos(pi*y)-2*pi*x.*y.*cos(pi*x).*sin(pi*y)- ...
72                  2*pi*(x.^2+y.^2+1).*sin(pi*x).*cos(pi*y)-3*y.*sin(pi*x).*sin(pi*y) );
73  %     alpha = @(x,y) 1; % alpha = @(x,y) 1*[1 0;0 1];
74  %     gamma = @(x,y) x.*0+y.*0;
75  %     f = @(x,y) 2*pi^2*sin(pi*x).*cos(pi*y);
76  elseif u_case == 2
77      u_r = @(x, y) x.*(1-x).*y.*(1-y);
78      grad_u_r = @(x, y) [(1-2*x).*y.*(1-y), x.*(1-x).*(1-2*y)];
79      % Here the diffusion coefficient α is not uniformly positive
80      % definite, so the rates of convergence (in L^∞, L^2, H^1 norms)
81      % are unspecified though the exact solution is smooth.
82      alpha = @(x,y) x.*y; % alpha = @(x,y) x.*y.*[1 0;0 1];
83      gamma = @(x,y) 0.*x.*y;
84      f = @(x,y) -( (1-4*x).*y.^2.*(1-y)+x.^2.*(1-x).*(1-4*y) );
85  elseif u_case == 3
86      k = 3; % k = 9;
87      u_r = @(x, y) sin(2*pi*x).*sin(2*k*pi*y);
88      grad_u_r = @(x, y) [2*pi*cos(2*pi*x).*sin(2*k*pi*y), 2*k*pi*sin(2*pi*x).*cos(2*k*pi*y)];
89      alpha = @(x,y) [k^2, 0; 0, 1];
90      gamma = @(x,y) 0.*x.*y;
91      f = @(x,y) 8*k^2*pi^2*sin(2*pi*x).*sin(2*k*pi*y);
92  end
93
94  A = assembleReactionDiffusionMatrix(p,t,alpha,gamma);
95  % phi = assembleVector(p,t,f);
96  phi = assembleVectorByGaussQuad(p,t,f);
97
98  % process boundary conditions
99  % see also "example1.m" for how to process Neumann BCs
100 [A,phi] = processDirichletBoundary(A,phi,p,u_case);
101
102 % solve the linear system of equations
103 u = A\phi;
104
105 % get discretization errors in various norms
106 [L_inf_err,L2_err,H1_semi_err,H1_err] = errorEstimates(p,t,u,u_r,grad_u_r);
107
108 % t is specified as a 4-by-Nt matrix in pdemesh, although we don't
109 % use the 4th row in a 2D mesh. To be compatible with 3D meshes?
110 t=[t';zeros(1,size(t,1))];
111 p=p'; % required to be 2-by-nP
112
113 figure(1)
114 pdemesh(p,0123,t,u) % 2nd arg `e` is unused, but has to be a matrix
115 xlabel('x'); ylabel('y');
116 title('LFEM solution')
117
118 figure(2)
119 pdemesh(p,0123,t,u_r(p(1,:),p(2,:)))
120 xlabel('x'); ylabel('y');
121 title('exact solution')
122 end
123
124 function [A,phi] = processDirichletBoundary(A,phi,p,u_case)
125 if u_case == 1
126     db_pos1 = find(p(:,1)==0|p(:,1)==1);
127     db_pos2 = find(p(:,2)==0);
128     db_pos3 = find(p(:,2)==1);
129     db_pos = [db_pos1; db_pos2; db_pos3];
130     db_pos = unique(db_pos);
```

```
    phi(db_pos1) = zeros(size(db_pos1,1),1);
    phi(db_pos2) = sin(pi*p(db_pos2,1));
    phi(db_pos3) = -sin(pi*p(db_pos3,1));
else%if u_case == 2 || u_case ==3
    db_pos = find(p(:,1)==0|p(:,1)==1|p(:,2)==0|p(:,2)==1);
    phi(db_pos) = 0;
end
A(db_pos,:) = 0;
A(db_pos,db_pos) = eye(size(db_pos,1));
end
```

Code listing 10: Examples used for comparison with the WG FEM

```
% Gaussian quadrature over a triangle
function S = GaussTriaQuad(triangle, f)
% @para triangle: a 3x2 matrix, 3 pairs of (x,y) coordinates
% @para f: function handle of a two-variale func to be integrated

% 6-point quadrature rule of order 4 on the unit triangle [0 0;1 0;0 1]
w = [1/60; 1/60; 1/60; 9/60; 9/60 ; 9/60];
x = [1/2 0; 1/2 1/2; 0 1/2; 1/6 1/6; 1/6 2/3; 2/3 1/6];

% An affine mapping that transforms the unit triangle to a general
% triangle:
%
%  |\        \Phi(x^) = F * x^ + \tau         /\
%  |  \     --------------------------->     /   \
%  |____\                                  /______\
%   (K^)                                    (K)

bK = triangle(1,:);
BK = [triangle(2,:)-bK; triangle(3,:)-bK];
det_BK = abs(det(BK));

% transform quadrature points and integrate
x = x*BK+bK;
S = sum(w.*f(x(:,1),x(:,2)))*det_BK;
end
```

Code listing 11: Gaussian quadrature over a triangle

```
% Gaussian quadrature
function T = Gaussquad(f,a,b,n)
if nargin<4
    n=8;
end
% Gaussian quarature points & weights on [-1,1]
if n == 2
    w = [1,1];                      % weights
    p = [-1/sqrt(3),1/sqrt(3)];     % points
elseif n == 4
    w = [0.3478548451,0.3478548451,0.6521451549,0.6521451549];
    p = [0.8611363116,-0.8611363116,0.3399810436,-0.3399810436];
elseif n == 8
    w = [0.1012285363,0.1012285363,0.2223810345,0.2223810345,0.3137066459,...
                0.3137066459,0.3626837834,0.3626837834];
    p = [0.9602898565,-0.9602898565,0.7966664774,-0.7966664774,0.5255324099,...
                        -0.5255324099,0.1834346425,-0.1834346425];
else
    error('n must be either 2 or 4 or 8')
end

% take a linear transform: x = t*(b-a)/2 + (a+b)/2, where t belongs to [-1,1]
% int(f,[a,b]) = (b-a)/2 * int(f(t*(b-a)/2 + (a+b)/2), [-1,1])
```

```
w = 0.5*(b-a)*w;
p = 0.5*(b-a)*p+0.5*(a+b);

T = w*f(p).';
end
```

Code listing 12: Gaussian quadrature in 1D

```
[x,y] = meshgrid((1:2:10)/10,(1:2:10)/10);
tri = delaunay(x,y);
z = peaks(5);
z = z + .3*rand(5); % oscillation from the Gaussian distributions
zmax=max(max(z)); zmin=min(min(z));
a=.05; b =1;
z = a+(z-zmin)/(zmax-zmin)*(b-a); % transform to [a,b]

figure(1)
trisurf(tri,x,y,z)
figure(2)
triplot(tri,x,y)

% plot the mesh in 3d coordinate system
figure(1)
hold on
t=[.1, .9]; z0=[0,0];
for v=.1:.2:.9
    plot3(t,[v,v],z0,'r');
    plot3([v,v],t,z0,'r');
end
for v=.3:.2:.9
    plot3([.1,v],[v,.1],z0,'r')
end
for v=.3:.2:.7
    plot3([v,.9],[.9,v],z0,'r')
end

% You may want to adjust the figure to a satisfying position first
% fig2svg("../svg/trisurf_piecewise_affine_linear_function_example.svg")
%%% NOTE %%%
% When exporting to svg, the red mesh will cover (be on the upper layer)
% part of the surface plot if we first draw the surface then the mesh.
% But we can use some svg editor (for example, Inkscape) to raise the layer
% of the surface so that we can use the svg in our paper.
```

Code listing 13: Code using `trisurf` for drawing the piecewise affine linear function in Figure 9