Uncertainty Quantification and Quasi-Monte Carlo

Sommersemester 2025

Vesa Kaarnioja
vesa.kaarnioja@fu-berlin.de

FU Berlin, FB Mathematik und Informatik

Third lecture, April 28, 2025

Recap: Weak formulation

Let $D \subset \mathbb{R}^d$ be an open and bounded Lipschitz domain. We consider the problem

$$\begin{cases} -\nabla \cdot (a(\boldsymbol{x}) \nabla u(\boldsymbol{x})) = f(\boldsymbol{x}), & \boldsymbol{x} \in D, \\ u|_{\partial D} = 0, \end{cases} \tag{1}$$

where $f : D \to \mathbb{R}$ is the *source* and $a : D \to \mathbb{R}$ is the *diffusion coefficient*.

Uniform ellipticity assumption: There exist constants $a_{\min}, a_{\max} > 0$ such that

$$0 < a_{\min} \leq a(\boldsymbol{x}) \leq a_{\max} < \infty \quad \text{for all } \boldsymbol{x} \in D.$$

Definition

Let $a \in L^\infty(D)$ and $f \in L^2(D)$. Then $u \in H_0^1(D)$ is called a weak solution to (1) if

$$B(u, v) = F(v) \quad \text{for all } v \in H_0^1(D), \tag{2}$$

where

$$B(u, v) = \int_D a(\boldsymbol{x}) \nabla u(\boldsymbol{x}) \cdot \nabla v(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x}$$

and

$$F(v) = \int_D f(\boldsymbol{x}) v(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x}.$$

Galerkin method

1. Let $V_m = \operatorname{span}\{\phi_i\}_{i=1}^m \subset H_0^1(D)$ be a finite-dimensional subspace.
2. Find $u_m \in V_m$ s.t.

$$B(u_m, \phi) = F(\phi) \quad \text{for all } \phi \in V_m. \tag{3}$$

Lemma

The problem (3) has a unique solution which also satisfies the so-called Galerkin orthogonality

$$B(u - u_m, \phi) = 0 \quad \textit{for all } \phi \in V_m,$$

where u is the solution to (2).

Proof. The existence of a unique solution is an immediate consequence of the Lax–Milgram lemma applied to (3) in a subspace $V_m \subset H_0^1(D)$. The orthogonality follows from

$$B(u-u_m, \phi) = B(u, \phi) - B(u_m, \phi) = F(\phi) - F(\phi) = 0 \quad \text{for all } \phi \in V_m. \qquad \square$$

Let $V_m := \operatorname{span}\{\phi_i\}_{i=1}^m \subset H_0^1(D)$. Note that the problem of finding $u_m \in V_m$ such that

$$B(u_m, \phi) = F(\phi) \quad \text{for all } \phi \in V_m$$

is equivalent to

$$B(u_m, \phi_j) = F(\phi_j) \quad \text{for all } j \in \{1, \dots, m\}.$$

Since $u_m \in V_m$, we can write it as $u_m(\boldsymbol{x}) = \sum_{i=1}^m c_i \phi_i(\boldsymbol{x})$ using *undetermined coefficients* $\boldsymbol{c} = (c_i)_{i=1}^m \subset \mathbb{R}$. Thus the problem of finding $u_m \in V_m$ is equivalent to solving the coefficients $\boldsymbol{c}$ satisfying

$$\sum_{i=1}^m c_i B(\phi_i, \phi_j) = F(\phi_j) \quad \text{for all } j \in \{1, \dots, m\},$$

which can be expressed as a linear system

$$\boldsymbol{A}\boldsymbol{c} = \boldsymbol{F},$$

where $\boldsymbol{A} = (A_{i,j})_{i,j=1}^m$ and $\boldsymbol{F} = (F_i)_{i=1}^m$ are such that

$$A_{i,j} = B(\phi_i, \phi_j) = \int_D a(\boldsymbol{x}) \nabla \phi_i(\boldsymbol{x}) \cdot \nabla \phi_j(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x}, \quad F_i = \int_D f(\boldsymbol{x}) \phi_i(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x}.$$

The Galerkin solution is a "quasi-optimal" approximation of the weak solution of the PDE in V_m.

Lemma (Céa's lemma)

Let $u \in H_0^1(D)$ be the solution to $B(u, \phi) = F(\phi)$ for all $\phi \in H_0^1(D)$ and let $u_m \in V_m$ be the solution to $B(u_m, \phi) = F(\phi)$ for all $\phi \in V_m$. Then

$$\|u - u_m\|_{H_0^1(D)} \leq \frac{a_{\max}}{a_{\min}} \inf_{v \in V_m} \|u - v\|_{H_0^1(D)}.$$

Proof. Let $v \in V_m$. Then by the coercivity and continuity of B, there holds

$$\begin{aligned} a_{\min}\|u - u_m\|^2_{H_0^1(D)} &\leq B(u - u_m, u - u_m) \\ &= B(u - u_m, u - v) + \underbrace{B(u - u_m, \underbrace{v - u_m}_{\in V_m})}_{=0} \\ &\leq a_{\max}\|u - u_m\|_{H_0^1(D)}\|u - v\|_{H_0^1(D)}. \end{aligned}$$

Hence $a_{\min}\|u - u_m\|_{H_0^1(D)} \leq a_{\max}\|u - v\|_{H_0^1(D)}$. □

Finite element method

One could choose the space $V_m \subset H_0^1(D)$ to be virtually anything. The finite element method is a particular way of constructing this finite dimensional space.

In 2D, we approximate the geometry D by constructing a triangulation.

That is, the computational domain D is represented as the union of non-overlapping triangles called *elements*. The elements are assumed to cover the whole D (and only D). In 2D the elements are typically triangles or quadrilaterals, but they could be practically of any shape. In 3D the elements are typically tetrahedra or hexahedra. Prisms and pyramids are also widely used.

If the domain D is a polyhedron, then the division to elements is accurate. If the domain has, e.g., curved edges, then it cannot be approximated accurately with linear elements. This introduces additional error to the numerical approximation.

A single element is denoted by K. The collection of elements is called a mesh and denoted with $\mathcal{T}_h$, indexed by the diameter of the maximum element in the mesh. The size of the elements plays a key role in the convergence analysis of the method. For a well-defined method, reducing the size of the elements, i.e., refining the mesh, improves the solution (or at least does not make it worse).

The mesh is a discretization of the domain. It does not define a function space. To define the global space, we define the local space in each of the elements. The global space is a piecewise combination of the local, elementwise spaces.

Assume that the domain D is a 2D polyhedral domain, e.g., unit square, and that it has been divided into triangles. The simplest possible subspace to $H_0^1(D)$ is a piecewise linear, continuous space

$$V_h := \{v \in H_0^1(D) \mid v \in \mathcal{P}^1(K)\ \forall K \in \mathcal{T}_h\}.$$

The continuity is enforced by our requirement that the functions belong to $H^1(D)$.

Let $\mathcal{T}_h$ be a triangulation of the domain D with FE nodes $(\boldsymbol{n}_i)_{i=1}^N$, where $m < N$ nodes are in the interior of the domain and $N - m$ nodes are on the boundary ∂D. For later convenience, let us denote $\texttt{interior} := \{i \in \{1, \ldots, N\} \mid \boldsymbol{n}_i \notin \partial D\}$.

We can choose piecewise linear basis functions $\phi_i = \phi_{\boldsymbol{n}_i}$ such that

$$\phi_{\boldsymbol{n}_i}(\boldsymbol{n}_j) = \delta_{i,j},$$

that is, $\phi_{\boldsymbol{n}_i}(\boldsymbol{n}_i) = 1$ and $\phi_{\boldsymbol{n}_i}(\boldsymbol{n}_j) = 0$ whenever $i \neq j$ over the FE nodes $(\boldsymbol{n}_i)_{i=1}^N$.

Since V_h is finite-dimensional, it is spanned by a set of global basis functions

$$V_h = \operatorname{span}\{\phi_{\boldsymbol{n}_i}\}_{i \in \texttt{interior}}.$$

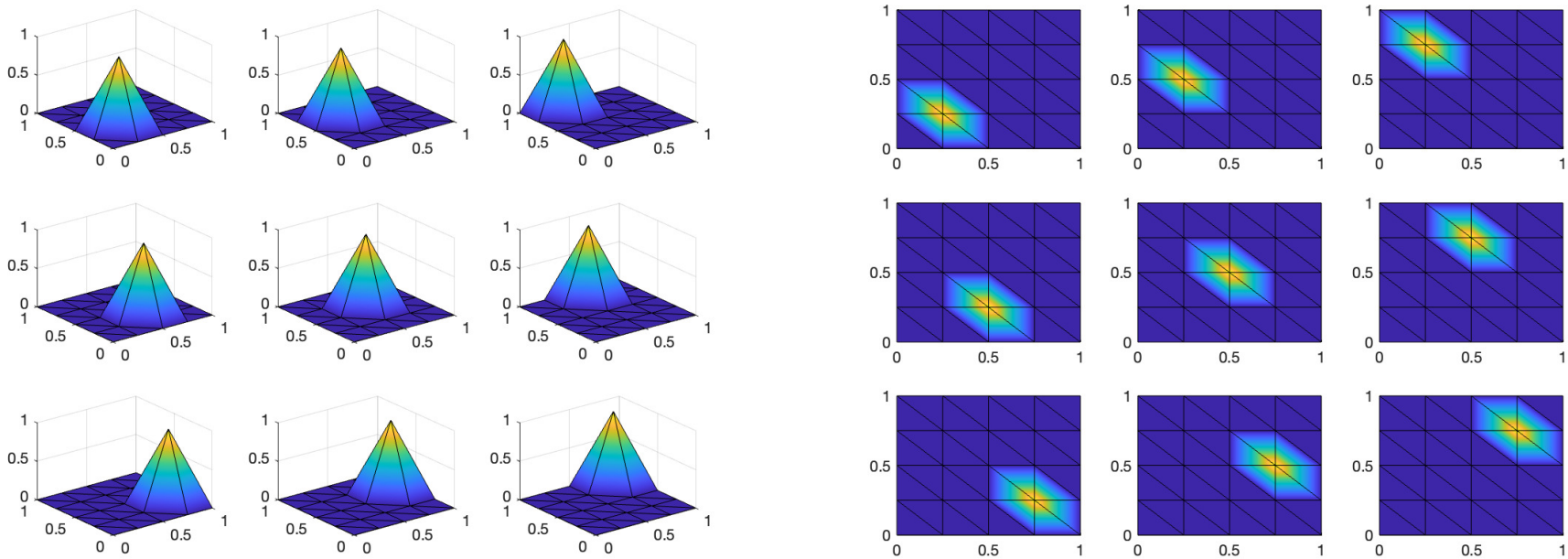


Figure: Left: An illustration of global, piecewise linear FE basis functions spanning V_h over a regular, uniform triangulation of $(0,1)^2$. Right: Bird's-eye view of the same global FE basis functions.

The goal is to find the FE solution $u_h \in V_h$ such that

$$\int_D a(\boldsymbol{x}) \nabla u_h(\boldsymbol{x}) \cdot \nabla \phi(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} = \int_D f(\boldsymbol{x}) \phi(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} \quad \text{for all } \phi \in V_h.$$

For simplicity, suppose that $f(\boldsymbol{x}) := \sum_{i=1}^N f_i \phi_{\boldsymbol{n}_i}(\boldsymbol{x})$ for some known coefficients $\boldsymbol{f} := (f_i)_{i=1}^N \subset \mathbb{R}$.[†] We can write $u_h = \sum_{i=1}^N c_i \phi_{\boldsymbol{n}_i} \in V_h$ and *enforce the zero Dirichlet boundary condition by setting* $c_i = 0$ *for any* $\boldsymbol{n}_i \in \partial D$. Testing the variational formulation against the FE basis functions $\phi = \phi_j$ for all $j \in \texttt{interior}$:

$$\sum_{i \in \texttt{interior}} c_i \underbrace{\int_D a(\boldsymbol{x}) \nabla \phi_{\boldsymbol{n}_i}(\boldsymbol{x}) \cdot \nabla \phi_{\boldsymbol{n}_j}(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x}}_{=:A_{i,j}} = \sum_{i=1}^N f_i \underbrace{\int_D \phi_{\boldsymbol{n}_i}(\boldsymbol{x}) \phi_{\boldsymbol{n}_j}(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x}}_{=:M_{i,j}}.$$

Thus the problem is to solve the FE expansion coefficients $\boldsymbol{c} = (c_i)_{i \in \texttt{interior}}$ from the equation

$$\boldsymbol{A}_{\texttt{interior},\texttt{interior}} \boldsymbol{c} = \boldsymbol{M}_{\texttt{interior},:} \boldsymbol{f},$$

where the matrix $\boldsymbol{A} = (A_{i,j})_{i,j=1}^N$ is called the *stiffness matrix* and $\boldsymbol{M} = (M_{i,j})_{i,j=1}^N$ is called the *mass matrix*.

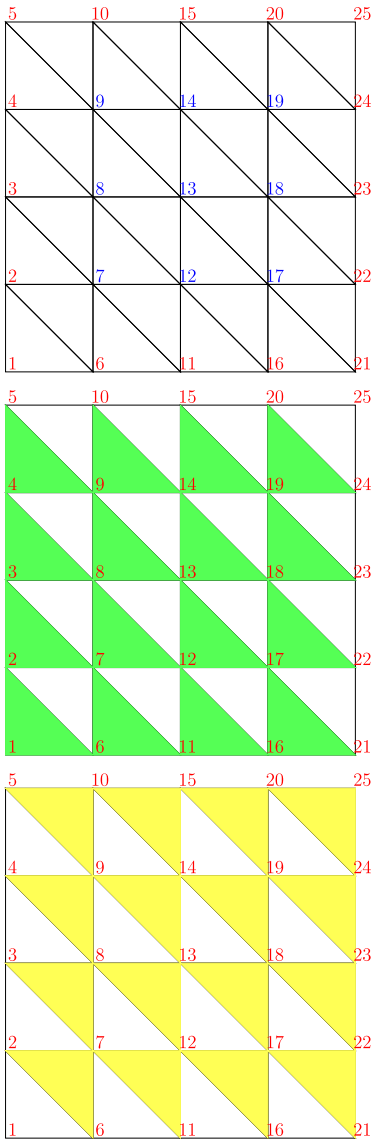
[†] *Note that here we do not require* $f = 0$ *on* ∂D*! For general* $f \in L^2(D)$*, instead of the mass matrix, one would use (Gaussian) quadratures to form the right-hand side.*

Our goals in finite element programming:

- Construct a data structure to represent the topology of the finite element mesh.
  - If the FE nodes are given as rows of an array "`nodes`", then the elements are triangles with vertices `nodes[i,:]`, `nodes[j,:]`, `nodes[k,:]` for certain indices `i`, `j`, `k`.
  - We can represent the elements as an array "`element`", where each row contains the *indices* corresponding to the nodes which form a triangle in our mesh.
  - Since we focus on homogeneous zero Dirichlet boundary conditions, we can enforce the boundary condition by setting the FE expansion coefficients of the FE solution to be zero at the boundary nodes. This is equivalent to choosing a subspace V_h consisting only of those FE basis functions $\phi_{\boldsymbol{n}_i}$ corresponding to FE nodes in the interior of the mesh, i.e., $\boldsymbol{n}_i \notin \partial D$. Thus it is helpful to store the indices of the nodes lying in the interior of the domain into a vector called "`interior`".
- Assembly of finite element matrices (stiffness and mass matrix).
  - All triangles in our FE mesh can be mapped affinely onto a reference triangle of our choosing (say, $\{(x, y) \in \mathbb{R}^d \mid 0 \leq x \leq 1,\ 0 \leq y \leq 1 - x\}$) which we can exploit in the construction of the FE matrices.

Finite element programming (in Python)

Triangulation of $D = (0,1)^2$




```
import numpy as np

def generateFEmesh(level):
    # Create a regular uniform triangulation
    # of the unit square (0,1)**2
    n1 = 2**level+1 # number of nodes in 1D
    # Topology: FE nodes, mesh elements, interior, centers
    X,Y = np.meshgrid(np.arange(0,n1)/(n1-1),
                      np.arange(0,n1)/(n1-1))
    nodes = np.array([X.flatten(),Y.flatten()]).T
    element = []; interior = []
    for i in range(0,n1-1):
        for j in range(0,n1-1):
            element.append([j*n1+i,(j+1)*n1+i,j*n1+i+1])
            element.append([(j+1)*n1+i,(j+1)*n1+i+1,j*n1+i+1])
            if i < n1-2 and j < n1-2:
                interior.append((j+1)*n1+i+1)
    centers = np.mean(nodes[element[:]],axis=1)
    return nodes,element,interior,centers
```

Mass matrix

Let $(K_\ell)_{\ell=1}^{\texttt{nelem}}$ be non-overlapping mesh elements s.t. $D = \bigcup_{\ell=1}^{\texttt{nelem}} K_\ell$. Let us first consider constructing the global mass matrix:

$$M_{i,j} = \int_D \phi_{\boldsymbol{n}_i}(\boldsymbol{x})\phi_{\boldsymbol{n}_j}(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \sum_{\ell=1}^{\texttt{nelem}} \int_{K_\ell} \phi_{\boldsymbol{n}_i}(\boldsymbol{x})\phi_{\boldsymbol{n}_j}(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}.$$

We can think of the elements of the global mass matrix as a sum of locally defined mass matrices in each "active" element. Recall that we already gave a labeling to the FE nodes earlier, and each row of matrix `element` contains the indices of FE nodes which form an element in our FE mesh.

initialize $\texttt{M}_{\texttt{i},\texttt{j}} = 0$, $\texttt{i},\texttt{j} \in \{0,\ldots,\texttt{ncoord}-1\}$
for $\texttt{k} \in \{0,\ldots,\texttt{nelem}-1\}$, **do**

1. set $\texttt{ind} = \texttt{element}[\texttt{k}]$
2. let K be the element with vertices $\texttt{nodes}[\texttt{ind}]$
3. compute local mass matrix $\texttt{L} \in \mathbb{R}^{3\times 3}$, where $\texttt{L}_{\texttt{i},\texttt{j}} = \int_K \phi_{\boldsymbol{n}_i}(\boldsymbol{x})\phi_{\boldsymbol{n}_j}(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$, $i,j \in \{0,1,2\}$
4. set $\texttt{M}_{\texttt{ind},\texttt{ind}} = \texttt{M}_{\texttt{ind},\texttt{ind}} + \texttt{L}$

end for

Let us concentrate on step 3.

Local mass matrix

- Let $K \subset \mathbb{R}^2$ be an arbitrary triangle with vertices $\boldsymbol{n}_1$, $\boldsymbol{n}_2$, $\boldsymbol{n}_3$, and $\hat{K} = \{(x_1, x_2) \mid 0 \le x_1 \le 1,\ 0 \le x_2 \le 1 - x_1\}$ the reference triangle.
- Let $\hat{\phi}_1(\boldsymbol{x}) = 1 - x_1 - x_2$, $\hat{\phi}_2(\boldsymbol{x}) = x_1$, $\hat{\phi}_3(\boldsymbol{x}) = x_2$ be the local basis.
- The affine mapping $F_K \colon \hat{K} \to K$, $F_K(\boldsymbol{x}) := B\boldsymbol{x} + \boldsymbol{n}_1$, $B = [\boldsymbol{n}_2 - \boldsymbol{n}_1, \boldsymbol{n}_3 - \boldsymbol{n}_1]$, can be used to write the global basis functions as $\phi_{\boldsymbol{n}_i}(\boldsymbol{x}) = \hat{\phi}_i(F_K^{-1}(\boldsymbol{x}))$. Change of variables:

$$\int_K \phi_{\boldsymbol{n}_i}(\boldsymbol{x})\phi_{\boldsymbol{n}_j}(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = |\det B| \int_{\hat{K}} \hat{\phi}_i(\boldsymbol{x})\hat{\phi}_j(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \begin{cases} \frac{|\det B|}{12} & \text{if } i = j, \\ \frac{|\det B|}{24} & \text{if } i \neq j \end{cases}$$

that is

$$\left(\int_K \phi_{\boldsymbol{n}_i}(\boldsymbol{x})\phi_{\boldsymbol{n}_j}(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \right)_{i,j=1}^{3} = |\det B| \begin{pmatrix} \frac{1}{12} & \frac{1}{24} & \frac{1}{24} \\ \frac{1}{24} & \frac{1}{12} & \frac{1}{24} \\ \frac{1}{24} & \frac{1}{24} & \frac{1}{12} \end{pmatrix}.$$

Stiffness matrix

We also need to construct the global stiffness matrix

$$A_{i,j} = \int_D a(\boldsymbol{x}) \nabla \phi_{\boldsymbol{n}_i}(\boldsymbol{x}) \cdot \nabla \phi_{\boldsymbol{n}_j}(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x}.$$

To simplify the analysis, let us suppose that the diffusion coefficient $a(\boldsymbol{x})$ has been discretized as a piecewise constant function over the mesh elements, i.e.,

$$a(\boldsymbol{x}) = \sum_{\ell=1}^{\texttt{nelem}} a_\ell \chi_{K_\ell}(\boldsymbol{x}), \quad \chi_{K_\ell}(\boldsymbol{x}) := \begin{cases} 1 & \text{if } \boldsymbol{x} \in K_\ell, \\ 0 & \text{otherwise.} \end{cases}$$

Here, we can take a_ℓ to be the value of a evaluated at the center point of element K_ℓ. Then

$$A_{i,j} = \sum_{\ell=1}^{\texttt{nelem}} a_\ell \int_{K_\ell} \nabla \phi_{\boldsymbol{n}_i}(\boldsymbol{x}) \cdot \nabla \phi_{\boldsymbol{n}_j}(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x}.$$

Idea: Precompute the stiffness tensor $S_{i,j,\ell} := \int_{K_\ell} \nabla \phi_{\boldsymbol{n}_i}(\boldsymbol{x}) \cdot \nabla \phi_{\boldsymbol{n}_j}(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x}$. Given a, the stiffness matrix is a tensor-vector contraction $A = S \times_3 \boldsymbol{a}$, where $\boldsymbol{a}$ is a vector containing values of a at element center points.

The *idealized* construction of the stiffness tensor is as follows:

initialize $\mathtt{S}_{\mathtt{i},\mathtt{j},\mathtt{k}} = 0$, $\mathtt{i}, \mathtt{j} \in \{0, \ldots, \mathtt{ncoord} - 1\}$, $\mathtt{k} \in \{0, \ldots, \mathtt{nelem} - 1\}$

for $\mathtt{k} \in \{0, \ldots, \mathtt{nelem} - 1\}$, **do**

1. set $\mathtt{ind} = \mathtt{element}[\mathtt{k}]$
2. let K be the element with vertices $\mathtt{nodes}[\mathtt{ind}]$
3. compute local stiffness matrix $\mathtt{L} \in \mathbb{R}^{3\times 3}$, where $\mathtt{L}_{\mathtt{i},\mathtt{j}} = \int_K \nabla\phi_{\boldsymbol{n}_i}(\boldsymbol{x}) \cdot \nabla\phi_{\boldsymbol{n}_j}(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$, $\mathtt{i}, \mathtt{j} \in \{0, 1, 2\}$
4. set $\mathtt{S}_{\mathtt{ind},\mathtt{ind},\mathtt{k}} = \mathtt{L}$

end for

Problem: *Scipy does not support sparse tensors!* :(

Workaround: reshape the $n \times n \times m$ tensor into an $n^2 \times m$ matrix!

initialize $\mathtt{grad}_{\mathtt{i},\mathtt{j},\mathtt{k}} = 0$ for $\mathtt{i}, \mathtt{j} \in \{0, \ldots, \mathtt{ncoord} * \mathtt{ncoord} - 1\}$, $\mathtt{k} \in \{0, \ldots, \mathtt{nelem} - 1\}$

for $\mathtt{k} \in \{0, \ldots, \mathtt{nelem} - 1\}$, **do**

1. set $\mathtt{ind} = \mathtt{element}[\mathtt{k}]$
2. let K be the element with the vertices $\mathtt{nodes}[\mathtt{ind}]$
3. compute local stiffness matrix $\mathtt{L} \in \mathbb{R}^{3\times 3}$, where $\mathtt{L}_{\mathtt{i},\mathtt{j}} = \int_K \nabla\phi_{\boldsymbol{n}_i}(\boldsymbol{x}) \cdot \nabla\phi_{\boldsymbol{n}_j}(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$, $i, j \in \{0, 1, 2\}$
4. initialize $\mathtt{dummy} = 0_{\mathtt{ncoord},\mathtt{ncoord}}$; set $\mathtt{dummy}_{\mathtt{ind},\mathtt{ind}} = \mathtt{L}$
5. set $\mathtt{grad}[:, \mathtt{k}] = \mathtt{dummy.flatten}()$

end for

Local stiffness matrix

- Let $K \subset \mathbb{R}^2$ be an arbitrary triangle with vertices $\boldsymbol{n}_1$, $\boldsymbol{n}_2$, $\boldsymbol{n}_3$, and $\hat{K} = \{(x_1, x_2) \mid 0 \le x_1 \le 1,\ 0 \le x_2 \le 1 - x_1\}$ the reference triangle.
- Let $\hat{\phi}_1(\boldsymbol{x}) = 1 - x_1 - x_2$, $\hat{\phi}_2(\boldsymbol{x}) = x_1$, $\hat{\phi}_3(\boldsymbol{x}) = x_2$ be the local basis.
- The affine mapping $F_K \colon \hat{K} \to K$, $F_K(\boldsymbol{x}) := B\boldsymbol{x} + \boldsymbol{n}_1$, $B = [\boldsymbol{n}_2 - \boldsymbol{n}_1, \boldsymbol{n}_3 - \boldsymbol{n}_1]$, can be used to write the global basis functions as $\phi_{\boldsymbol{n}_i}(\boldsymbol{x}) = \hat{\phi}_i(F_K^{-1}(\boldsymbol{x}))$. *Note that there holds* $\nabla\phi_{\boldsymbol{n}_i}(\boldsymbol{x}) = B^{-\mathrm{T}}(\nabla\hat{\phi}_i)(F_K^{-1}(\boldsymbol{x}))$. Change of variables:

$$\int_K \nabla\phi_{\boldsymbol{n}_i}(\boldsymbol{x}) \cdot \nabla\phi_{\boldsymbol{n}_j}(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = |\det B| \int_{\hat{K}} B^{-\mathrm{T}}\nabla\hat{\phi}_i(\boldsymbol{x}) \cdot B^{-\mathrm{T}}\nabla\hat{\phi}_j(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}.$$

Define $G^{\mathrm{T}} := (\nabla\hat{\phi}_1, \nabla\hat{\phi}_2, \nabla\hat{\phi}_3) = \begin{pmatrix} -1 & 1 & 0 \\ -1 & 0 & 1 \end{pmatrix}$. Then

$$\left(\int_K \nabla\phi_{\boldsymbol{n}_i}(\boldsymbol{x}) \cdot \nabla\phi_{\boldsymbol{n}_j}(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \right)_{i,j=1}^{3} = \frac{|\det B|}{2} G B^{-1} B^{-\mathrm{T}} G^{\mathrm{T}}. \qquad (4)$$

Remark. With a bit of linear algebra, one can check that (4) is equal to $\frac{D^{\mathrm{T}}D}{4\mathrm{area}(K)}$, $D := [\boldsymbol{n}_3 - \boldsymbol{n}_2, \boldsymbol{n}_1 - \boldsymbol{n}_3, \boldsymbol{n}_2 - \boldsymbol{n}_1]$.

The shoelace formula

```
def shoelace(g):
    # Compute the area of a triangle with
    # vertices g[0], g[1], and g[2]
    return abs(np.linalg.det([g[0],g[1]])
               + np.linalg.det([g[1],g[2]])
               + np.linalg.det([g[2],g[0]]))/2
```

Assembly of the finite element matrices

```
from scipy import sparse

def generateFEmatrices(nodes,element):
   ncoord = len(nodes); nelem = len(element)
   mass_data = []; mass_rows = []; mass_cols = []
   grad_data = []; grad_rows = []; grad_cols = []
   localmass = np.array([[1/12,1/24,1/24],[1/24,1/12,1/24],[1/24,1/24,1/12]])
   for k in range(nelem):
      ind = element[k]; g = nodes[ind]
      detB = abs(np.linalg.det([g[1]-g[0],g[2]-g[0]]))
      Dt = np.array([g[2]-g[1],g[0]-g[2],g[1]-g[0]])
      triarea = shoelace(g)
      localgrad = Dt@Dt.T/4/triarea
      for i in range(3):
         for j in range(3):
            mass_rows.append(ind[i]); mass_cols.append(ind[j]);
            mass_data.append(detB*localmass[i,j])
            grad_rows.append(ind[i]*ncoord+ind[j]); grad_cols.append(k)
            grad_data.append(localgrad[i,j])
   mass = sparse.csr_matrix((mass_data,(mass_rows,mass_cols)),
                            shape=(ncoord,ncoord))
   grad = sparse.csr_matrix((grad_data,(grad_rows,grad_cols)),
                            shape=(ncoord*ncoord,nelem))
   return grad,mass
```

FEM programs in Python

```
level = 5 # discretization level
# Generate FE mesh
nodes,element,interior,centers = generateFEmesh(level)
ncoord = len(nodes) # number of coordinates
# Generate FE matrices
grad,mass = generateFEmatrices(nodes,element)
```

To obtain the stiffness matrix for a piecewise constant diffusion coefficient, we can use the following simple routine.

```
def UpdateStiffness(grad,a):
# Given vector a containing the values of the diffusion
# coefficient at the element center points, return the
# corresponding stiffness matrix
    n = np.sqrt(grad.shape[0]).astype(int)
    vec = grad @ sparse.csr_matrix(a.reshape((a.size,1)))
    return sparse.csr_matrix.reshape(vec,(n,n)).tocsr()
```

Finite element method in 2D – summary

1. Form a triangulation $\mathcal{T}_h$ of the domain D. Let $(\boldsymbol{n}_j)_{j=1}^N$ be the finite element nodes. Form the list `interior` containing the indices of interior nodes and the element connectivity matrix `element`. Denote by $m = |\texttt{interior}|$ the number of degrees of freedom.
2. Form the stiffness matrix $A \in \mathbb{R}^{m \times m}$ and mass matrix $M \in \mathbb{R}^{N \times N}$.
3. Form the loading vector $\boldsymbol{b} \approx M_{\texttt{interior},:}\boldsymbol{f}$, where $\boldsymbol{f} = [f(\boldsymbol{n}_1), \ldots, f(\boldsymbol{n}_N)]^{\mathrm{T}}$.
4. Solve $\boldsymbol{c} = (c_j)_{j=1}^m \in \mathbb{R}^m$ from $A\boldsymbol{c} = \boldsymbol{b}$.
5. The finite element solution is given by

$$u_h(\boldsymbol{x}) = \sum_{j=1}^{m} c_j \phi_j(\boldsymbol{x}), \quad \text{where } \phi_j = \phi_{\boldsymbol{n}_j}.$$

Remark: The global basis functions ϕ_j are typically *never constructed in practice!* Instead, note that $u_h(\boldsymbol{n}_j) = c_j$. Therefore, the nodal values of the FE solution are precisely the FE expansion coefficients – if one needs to evaluate the FE solution for $\boldsymbol{x} \in K$, one can use linear interpolation between the vertices of the triangle element K.

Computing norms of finite element solutions

Let $V_h \subset H_0^1(D)$ be a finite element space spanned by piecewise linear, continuous FE basis functions $\{\phi_i\}_{i=1}^m$ in the *interior of the domain*. Let

$$u_h(\boldsymbol{x}) = \sum_{i=1}^{m} c_i \phi_i(\boldsymbol{x}) \in V_h.$$

If $M_{i,j} = \int_D \phi_i(\boldsymbol{x})\phi_j(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$ is the mass matrix and $S_{i,j} = \int_D \nabla\phi_i(\boldsymbol{x}) \cdot \nabla\phi_j(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$ is the stiffness matrix of the Dirichlet Laplacian $-\Delta$ with homogeneous zero Dirichlet boundary conditions, then

$$\|u_h\|_{L^2(D)} = \sqrt{\boldsymbol{c}^{\mathrm{T}} M \boldsymbol{c}} \quad \text{and} \quad \|u_h\|_{H_0^1(D)} = \sqrt{\boldsymbol{c}^{\mathrm{T}} S \boldsymbol{c}},$$

where $\boldsymbol{c} = (c_i)_{i=1}^m$. These identities imply that M and S are positive definite.

Numerical example

Consider the elliptic PDE problem

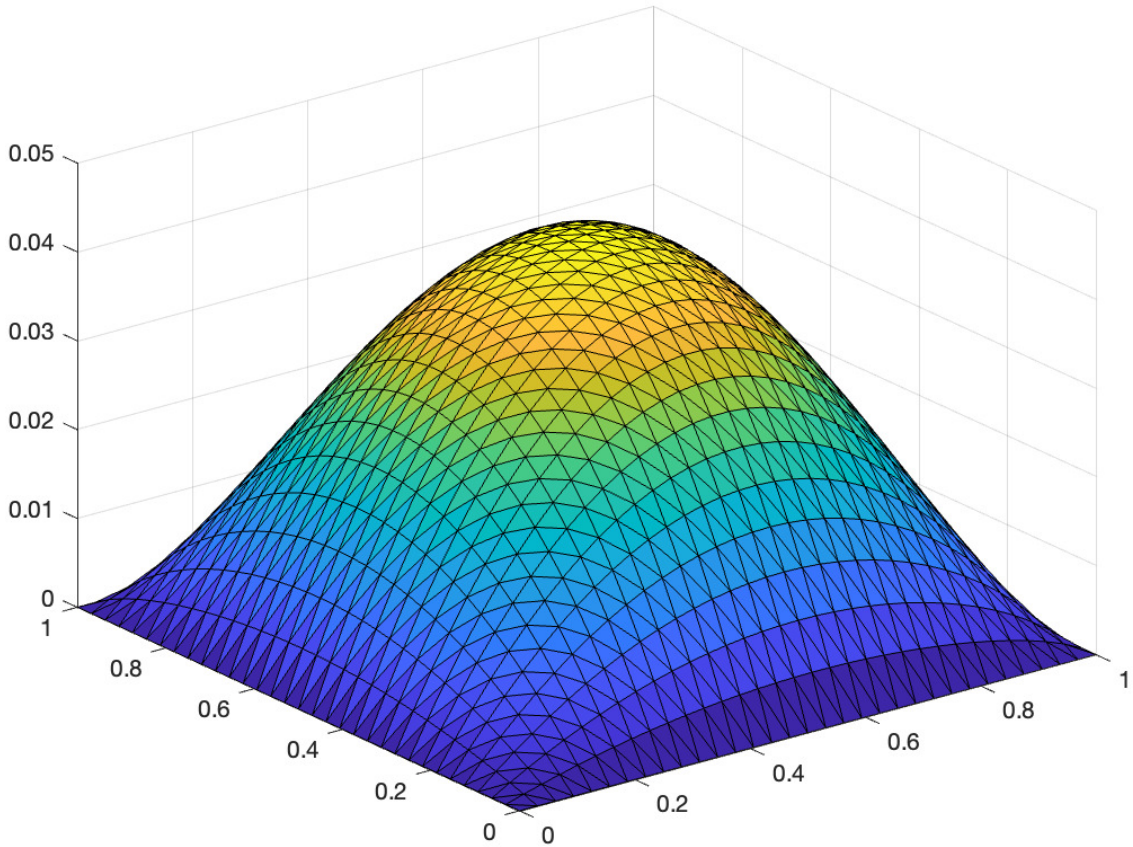
$$\begin{cases} -\nabla \cdot ((1 + x^2 + y^2)\nabla u(x, y)) = x + y, \quad (x, y) \in (0, 1)^2, \\ u|_{\partial D} = 0. \end{cases}$$

We can solve this problem using the code developed above as follows.

```
level = 5 # discretization level
nodes,element,interior,centers = generateFEmesh(level) # generate FE mesh
ncoord = len(nodes) # number of coordinates
grad,mass = generateFEmatrices(nodes,element) # generate FE matrices
a = lambda x: 1+np.sum(x**2,axis=1) # diffusion coefficient
f = lambda x: np.sum(x,axis=1) # source term
rhs = mass[interior,:]@f(nodes) # precompute the loading vector
aval = a(centers) # evaluate diffusion coefficient at element centers
stiffness = UpdateStiffness(grad,aval) # assemble stiffness matrix
sol = np.zeros(ncoord) # initialize solution vector
# Solve the PDE
sol[interior] = sparse.linalg.spsolve(stiffness[np.ix_(interior,interior)],rhs)

# Visualize the solution
import matplotlib.pyplot as plt
fig = plt.figure(figsize=plt.figaspect(1.0))
ax = fig.add_subplot(1,1,1,projection='3d')
ax.plot_trisurf(nodes[:,0],nodes[:,1],sol,triangles=element,cmap=plt.cm.rainbow)
plt.show()
```

FE solution



This note illustrates possibly the simplest (nontrivial) implementation of conforming h-FEM. "Conforming" means that the FE space V_h is a proper subspace of the solution space $H_0^1(D)$. With this method, the only way to increase the approximation accuracy is by mesh refinement. One could generalize the method in a various number of ways:

- Using higher-order piecewise polynomial basis functions leads to p- and hp-FEM. The idea is to use higher-order polynomials and larger elements in regions of the computational domain where the PDE solution is smooth; conversely, one would use lower order polynomial basis functions and smaller elements near singularities (caused by obtuse angles in the geometry, etc.). A proper refinement strategy with hp-FEM can lead to exponentially convergent implementations.
- One can even use discontinuous basis functions, but the method becomes non-conforming. This has the benefit of improved parallelization and easy adaptation, but the implementation details are significantly more involved.
- Instead of discretizing the diffusion coefficient as a piecewise constant function over the elements, a better approach would be to compute the local stiffness matrices $\int_K a(\boldsymbol{x})\nabla\phi_{\boldsymbol{n}_i}(\boldsymbol{x})\cdot\nabla\phi_{\boldsymbol{n}_j}(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$ using Gaussian quadratures for triangles. Similarly, the loading term $\int_K f(\boldsymbol{x})\phi_{\boldsymbol{n}_i}(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$ could also be computed using a Gaussian quadrature. For simplicity of presentation, the details are omitted.
- One could easily extend the method for more nontrivial boundary conditions: non-homogeneous Dirichlet, Neumann, Robin, mixed boundary conditions, etc. This results in additional "book-keeping" and the details are omitted.
- Many practitioners rely on automated mesh generation using software such as Netgen, etc. When the domain has curved boundaries, one usually either ignores the geometry modeling error (if there is reason to believe it is negligible) or uses, e.g., curved finite elements.
- Instead of using a direct solver like `scipy.sparse.linalg.spsolve` to solve the FE system, algebraic multigrid methods (and/or iterative solvers) can be used to improve the computational complexity. Nonlinear PDEs lead to nonlinear discretized FE systems.