22

Introduction to Finite Element Packages

22.1 Programming of finite element methods in Matlab: iFEM package

In this section we give description of popular data structures, representing the finite element mesh, the boundary of Ω, the stiffness matrix, and other data needed to compute the finite element solution. We also provide subroutines (written in matlab) which illustrate how such data structures are used and manipulated. These subroutines can be found in iFEM [**?**], an innovative finite element method package in MATLAB. In this novel coding style, the sparse matrix and its operation is used extensively in the data structure and algorithms.

22.1.1 Triangulations and related data structures

We shall discuss the data structure to represent triangulations and boundary data for the domain Ω. The matrices `node(1:N,1:d)` and `elem(1:NT,1:d+1)` are used to represent a d-dimensional triangulation embedded in $\mathbb{R}^d$, where N is the number of vertices and NT is the number of elements. These two matrices represent two different structure of a triangulation: `elem` for the topology and `node` for the embedding.

The matrix `elem` represents a set of abstract simplices. The index set $\{1, 2, \cdots, N\}$ is called the global index set of vertices. Here an vertex is thought as an abstract entity. By definition, `elem(t,1:d+1)` are the global indices of $d + 1$ vertices which form the abstract d-simplex `t`. Note that any permutation of vertices of t will represent the same abstract simplex.

The matrix `node` gives the geometric realization of the simplicial complex. For example, for a 2-D triangulation, `node(k,1:2)` contain x- and y-coordinates of the k-th nodes. The geometric realization introduces an ordering of the simplex. We shall always order the vertices of a simplex such that the signed area is positive. That is in 2-D, three vertices of a triangle is ordered anti-clockwise and in 3-D, the ordering of vertices follows the right-hand rule. Note that even with the orientation requirement, certain permutation of vertices is still allowed.

As an example, `node` and `elem` matrices to represent the triangulation of the L-shape domain $(-1, 1) \times (-1, 1) \backslash ([0, 1] \times [0, -1])$ in the Figure 22.1.

To build the boundary condition into the matrix equation, we first discuss the data structure to represent Dirichlet boundary Γ_D and Neumann boundary Γ_N.

For 2-D triangulations, we shall use `bdEdge(1:NT,1:3)` to indicate which edge of each element is on the boundary. The value is the type of boundary condition: 1 for first type, i.e., Dirichlet boundary edges, 2 for second type, i.e., Neumann boundary edges, and 0 for non-boundary, i.e., interior edges. For example, `bdEdge(t,:)=[1 0 2]` means, the edge opposite to `elem(t,1)` is a Dirichlet boundary face, the one to `elem(t,3)` is of Neumann type, and the other is an interior edge. The third type of boundary condition, i.e., Robin boundary condition, can be easily added but is not discussed in this article. We can extract boundary edges from `bdEdge` using the following code:

```
totalEdge = [elem(:,[2,3]); elem(:,[3,1]); elem(:,[1,2])];
isBdEdge = reshape(bdEdge,3*NT,1);
```

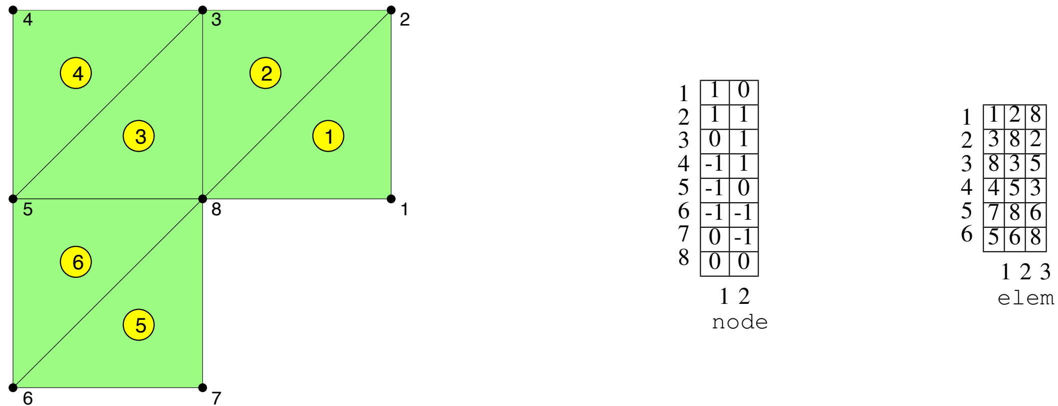

Fig. 22.1. Left: A triangulation of a L-shape domain. Right: `node` and `elem` matrices for the L-shape domain.

```
Dirichlet = totalEdge((isBdEdge == 1),:);
Neumann = totalEdge((isBdEdge == 2),:);
```

Similarly in 3-D, we use `bdFace(1:NT,1:4)` to indicate which face of each element is on the boundary and extract different type of boundary faces in a similar way.

22.1.2 Sparse matrices and data structures related to them

One of the nice features of finite element methods is the sparsity of the matrix obtained via the discretization. Although the matrix is $N \times N = N^2$, there are only cN nonzero entries in the matrix with a small constant c. Sparse matrix is the corresponding data structure to take advantage of this sparsity. Sparse matrix algorithms require less computational time by avoiding operations on zero entries and sparse matrix data structures require less computer memory by not storing many zero entries. We refer to books [**?**, **?**] for detailed description on sparse matrix data structure and [**?**] for a quick introduction on popular data structures of sparse matrix. In particular, the sparse matrix data structure and operations has been added to MATLAB by Gilbert, Moler and Schereiber and documented in [**?**].

There are different types of data structures for the sparse matrix. All of them share the same basic idea: use a single array to store all nonzero entries and two additional integer arrays to store the indices of nonzero entries.

A natural scheme, known as *coordinate format*, is to store both the row and column indices. In the sequel, we suppose A is a $m \times n$ matrix containing only nnz nonzero elements. Let us look at the following simple example:

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 4 \\ 0 & 0 & 0 \\ 0 & 9 & 0 \end{bmatrix}, \quad i = \begin{bmatrix} 1 \\ 2 \\ 4 \\ 2 \end{bmatrix}, \quad j = \begin{bmatrix} 1 \\ 2 \\ 2 \\ 3 \end{bmatrix}, \quad s = \begin{bmatrix} 1 \\ 2 \\ 9 \\ 4 \end{bmatrix}. \tag{22.1}$$

In this example, i vector stores row indices of non-zeros, j column indices, and s the value of non-zeros. All three vectors have the same length $nnz = 4$. Note that the two indices vectors i and j contains redundant information. We can compress the column index vector j to a column pointer vector with length $n + 1$. The value $j(k)$ will be the pointer to the beginning of k-th column in the vectors of i and s, and $j(n+1) = nnz+1$. This scheme is known as *Compressed Sparse Column (CSC)* scheme and is used in MATLAB sparse matrices package. For example, in CSC format, the vector to store the column pointer will be $j = [\,1\ 2\ 4\ 5\,]^t$. Comparing with coordinate format, CSC format saves storage for $nnz - n - 1$ integers which could be nonnegligilble when the number of nonzeros is much larger than that of the column. In CSC format it is

efficient to extract a column of a sparse matrix. For example, the k-th column of a sparse matrix can be built from the index vector i and the value vector s ranging from $j(k)$ to $j(k+1)-1$. There is no need of searching index arrays. An algorithm that builds up a sparse matrix one column at a time can be also implemented efficiently [?].

22.1.3 Assembling of matrix equation

In this section, we discuss how to obtain a matrix equation for the linear finite element method of solving the Poisson equation

$$-\Delta u = f \text{ in } \Omega, \qquad u = g_D \text{ on } \Gamma_D, \qquad \nabla u \cdot n = g_N \text{ on } \Gamma_N, \tag{22.2}$$

where $\partial\Omega = \Gamma_D \cup \Gamma_N$ and $\Gamma_D \cap \Gamma_N = \emptyset$. We assume Γ_D is closed and Γ_N open.

Denoted by $H^1_{g,D}(\Omega) = \{v \in L^2(\Omega), \nabla v \in L^2(\Omega) \text{ and } v|_{\Gamma_D} = g_D\}$. Using integration by parts, the weak form of the Poisson equation (22.2) is to find $u \in H^1_{g,D}(\Omega)$ such that for all $v \in H^1_{0_D}$

$$a(u,v) := \int_\Omega \nabla u \cdot \nabla v \, \mathrm{dxdy} = \int_\Omega f v \, \mathrm{dxdy} + \int_{\Gamma_N} g_N v \, dS\,. \tag{22.3}$$

Let $\mathcal{T}$ be a triangulation of Ω. We define the linear finite element space on $\mathcal{T}$ as

$$\mathbb{V}_\mathcal{T} = \{v \in C(\bar{\Omega}) : v|_\tau \in \mathcal{P}_1(\tau), \forall \tau \in \mathcal{T}\}.$$

For each vertex v_i of $\mathcal{T}$, let ϕ_i be the piecewise linear function such that $\phi_i(v_i) = 1$ and $\phi_i(v_j) = 0$ if $j \neq i$. Then it is easy to see $\mathbb{V}_\mathcal{T}$ is spanned by $\{\phi_i\}_{i=1}^N$. The linear finite element method for solving (22.2) is to find $u \in \mathbb{V}_\mathcal{T} \cap H^1_{g,D}(\Omega)$ such that (22.3) holds for all $v \in \mathbb{V}_\mathcal{T} \cap H^1_{0,D}(\Omega)$.

22.1.4 Assembling the stiffness matrix

For any function $v \in \mathbb{V}_\mathcal{T}$, there is a unique representation: $v = \sum_{i=1}^N v_i\phi_i$. We define an isomorphism $\mathbb{V}_\mathcal{T} \cong \mathbb{R}^N$ by

$$v = \sum_{i=1}^N v_i\phi_i \longleftrightarrow \boldsymbol{v} = (v_1, \cdots, v_N)^t, \tag{22.4}$$

and call $\boldsymbol{v}$ the vector representation of v (with respect to the basis $\{\phi_i\}_{i=1}^N$). We introduce the *stiffness matrix*

$$\boldsymbol{A} = (a_{ij})_{N\times N}, \quad \text{with} \quad a_{ij} = a(\phi_j, \phi_i).$$

In this subsection, we shall discuss how to form the matrix $\boldsymbol{A}$ efficiently in MATLAB.

Standard assembling process

By the definition, for $1 \le i, j \le N$,

$$a_{ij} = \int_\Omega \nabla\phi_j \cdot \nabla\phi_i \, \mathrm{dx} = \sum_{\tau\in\mathcal{T}} \int_\tau \nabla\phi_j \cdot \nabla\phi_i \, \mathrm{dx}.$$

For each simplex τ, we define the local stiffness matrix $A^\tau = (a^\tau_{ij})_{(d+1)\times(d+1)}$ as

$$a_{ij}^{\tau} = \int_{\tau} \nabla\lambda_i \cdot \nabla\lambda_j \, dx, \text{ for } 1 \le i, j \le d+1.$$

The computation of a_{ij} will be decomposed into the computation of local stiffness matrix and the summation over all elements.

A standard procedure to compute the local stiffness matrix is to transfer the computation to a reference simplex through an affine map. We include the two dimensional case here for the comparison and completeness.

We call the triangle $\hat{\tau}$ spanned by $\hat{v}_1 = (1,0), \hat{v}_2 = (0,1)$ and $\hat{v}_3 = (0,0)$ *a reference triangle* and use $\hat{\boldsymbol{x}} = (\hat{x}, \hat{y})^t$ for the vector in that coordinate. For any $\tau \in \mathcal{T}$, we treat it as the image of $\hat{\tau}$ under an affine map: $F : \hat{\tau} \to \tau$. One of such affine map is to match the local indices of three vertices, i.e., $F(\hat{v}_i) = v_i, i = 1,2,3$:

$$F(\hat{\boldsymbol{x}}) = B^t(\hat{\boldsymbol{x}}) + c,$$

where

$$B = \begin{bmatrix} x_1 - x_3 & y_1 - y_3 \\ x_2 - x_3 & y_2 - y_3 \end{bmatrix}, \text{ and } c = (x_3, y_3)^t.$$

We define $\hat{u}(\hat{\boldsymbol{x}}) = u(F(\hat{\boldsymbol{x}}))$. Then $\hat{\nabla}\hat{u} = B\nabla u$ and $\mathrm{dxdy} = |\det(B)|\mathrm{d}\hat{x}\mathrm{d}\hat{y}$. We change the computation of the integral in τ to $\hat{\tau}$ by

$$\begin{aligned}\int_{\tau} \nabla\lambda_i \cdot \nabla\lambda_j \mathrm{dxdy} &= \int_{\hat{\tau}} (B^{-1}\hat{\nabla}\hat{\lambda}_i) \cdot (B^{-1}\hat{\nabla}\hat{\lambda}_j)|\det(B)|\mathrm{d}\hat{x}\mathrm{d}\hat{y} \\ &= \frac{1}{2}|\det(B)|(B^{-1}\hat{\nabla}\hat{\lambda}_i) \cdot (B^{-1}\hat{\nabla}\hat{\lambda}_j).\end{aligned}$$

In the reference triangle, $\hat{\lambda}_1 = \hat{x}, \hat{\lambda}_2 = \hat{y}$ and $\hat{\lambda}_3 = 1 - \hat{x} - \hat{y}$. Thus

$$\hat{\nabla}\hat{\lambda}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \ \hat{\nabla}\hat{\lambda}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \text{ and } \hat{\nabla}\hat{\lambda}_3 = \begin{bmatrix} -1 \\ -1 \end{bmatrix}.$$

We then end with the following subroutine to compute the local stiffness matrix in one triangle τ. The advantage of this approach is that by modifying the subroutine `localstiffness`, one can easily adapt to new elements and new equations.

To get the global stiffness matrix, we apply a `for` loop of all elements and distribute element-wise quantity to node-wise quantity. A straightforward MATLAB code is like

```
function A = assemblingstandard(node,elem)
N=size(node,1); NT=size(elem,1);
A = sparse(N,N);
for t=1:NT
    At=locatstiffness(node(elem(t,:),:));
    for i=1:3
        for j=1:3
            A(elem(t,i),elem(t,j))=A(elem(t,i),elem(t,j))+At(i,j);
        end
    end
end
```

22.1.5 Assembling of the right hand side

We define the vector $\boldsymbol{f} = (f_1, \cdots, f_N)^t$ by $f_i = \int_{\Omega} f\phi_i$, where ϕ_i is the hat basis at the vertex v_i. Although the 1-point quadrature is adequate for linear element on quasi-uniform meshes, to reduce the error introduced by the numerical quadrature on adaptive meshes, we compute the load term $\int_{\Omega} f\phi_i$ by 3-points quadrature rule in 2-D and 4-points rule in 3-D which are exact for quadratic polynomials.

We list the 2-D code below as an example to emphasis again that the command `accumarray` is used to avoid the slow `for` loop over all elements.

```
mid1 = (node(elem(:,2),:)+node(elem(:,3),:))/2;
mid2 = (node(elem(:,3),:)+node(elem(:,1),:))/2;
mid3 = (node(elem(:,1),:)+node(elem(:,2),:))/2;
bt1 = area.*(f(mid2)+f(mid3))/6;
bt2 = area.*(f(mid3)+f(mid1))/6;
bt3 = area.*(f(mid1)+f(mid2))/6;
b = accumarray(elem(:),[bt1;bt2;bt3],[N 1]);
```

22.1.6 Implementation of boundary conditions

We list the code for 2-D case and briefly explain it.

```
%-------------------- Dirichlet boundary conditions------------------------
isBdNode = false(N,1); isBdNode(Dirichlet) = true;
bdNode = find(isBdNode);
freeNode = find(¬isBdNode);
u = zeros(N,1); u(bdNode) = g_D(node(bdNode,:));
b = b - A*u;
%-------------------- Neumann boundary conditions -------------------------
Nve = node(Neumann(:,1),:) - node(Neumann(:,2),:);
edgeLength = sqrt(sum(Nve.^2,2));
mid = (node(Neumann(:,1),:) + node(Neumann(:,2),:))/2;
b = b + accumarray([Neumann(:),ones(2*size(Neumann,1),1)], ...
                   repmat(edgeLength.*g_N(mid)/2,2,1),[N,1]);
```

Line 1-3 will find all Dirichlet boundary nodes. The Dirichlet boundary condition is posed by assign the function values at Dirichlet boundary nodes. Note that the vector `u` is initialized as zero vector. Therefore after line 5, the vector `u` will represent a function $u_D \in H_{g_D,\Gamma_D}$. Writing $u = \tilde{u} + u_D$, finding u is equivalent to finding $\tilde{u} \in \mathbb{V}_{\mathcal{T}} \cap H_0^1(\Omega)$ such that $a(\tilde{u}, v) = (f, v) - a(u_D, v) + (g_N, v)_{\Gamma_N}$ for all $v \in \mathbb{V}_{\mathcal{T}} \cap H_0^1(\Omega)$. The modification of the right hand side $(f, v) - a(u_D, v)$ is realized by the code `b=b-A*u` in line 6. The boundary integral involving the Neumann boundary part is computed in line 8–12. Note that the code is speed up using `accumarray`.

Since u_D and $\tilde{u}$ use disjoint nodes set, one vector `u` is used to represent both. The addition of $\tilde{u} + u_D$ is realized by assign values to different node sets of the same vector `u`. We have assigned the value to boundary nodes in line 5. We can compute $\tilde{u}$, i.e., the value at other nodes (denoted by `freeNode`), by

(22.5) `u(freeNode)=A(freeNode,freeNode)\b(freeNode).`

Here `A\b` computes $A^{-1}b$ using MATLAB build int direct solver. In the next section, we shall discuss more efficient solvers for this algebraic system.

22.1.7 Numerical quadrature

In the implementation, we need to compute various integrals on a simplex. In this subsection, we will present several numerical quadrature rules for simplexes in 1, 2, and 3 dimensions.

The numerical quadrature is to approximate an integral by weighted average of function values at sampling points p_i:

$$\int_\tau f(\mathbf{x})\mathrm{d}\mathbf{x} \approx I_n(f) := \sum_{i=1}^n f(p_i) w_i |\tau|.$$

The order of a numerical quadrature is defined as the largest integer k such that inf $f = I_n(f)$ when f is a polynomial of degree less than equal to k.

A numerical quadrature is determined by the quadrature points and corresponding weight: $(p_i, w_i), i = 1, \cdots, n$. For a d-simplex τ, let $\mathbf{x}_i, i = 1, c \cdots, d+1$ be vertices of τ. The simplest one is the one point rule:

$$I_1(f) = f(c_\tau)|\tau|, c_\tau = \frac{1}{d+1}\sum_{i=1}^{d+1}\mathbf{x}_i.$$

A very popular one is the trapezoidal rule:

$$I_1(f) = \frac{1}{d+1}\sum_{i=1}^{d+1} f(\mathbf{x}_i)|\tau|.$$

Both of them are of order one, i.e., exact for linear polynomial. For second order quadrature, in 1-D, the Simpson rule is quite popular

$$\int_a^b f(x)dx \approx (b-a)\frac{1}{6}\left(f(a) + 4f((a+b)/2) + f(b)\right).$$

For a triangle, a second order quadrature is using three middle points $m_i, i = 1, 2, 3$ of edges:

$$\int_\tau f(x)dx \approx |\tau|\frac{1}{3}\sum_{i=1}^{3} f(m_i).$$

These rules are popular due to the reason that the points and the weight are easy to memorize. No such rule exists for 3-D second order quadrature rule.

A criterion for choosing quadrature points is to attain a given precision with the fewest possible function evaluations. A simple question: for the two first order quadrature rules given above, which one shall we use? Restricting to one simplex, the answer is obvious. When considering an integral over a triangulation, the trapezoidal rule is better since it only evaluates the function at N vertices while the center rule needs NT evaluation. It is a simple exercise to show $NT \approx 2N$ asymptotically.

Another criterion will be related to the inverse of matrix. For example, mass lumping can be realized by the trapezoidal rule. We will discuss this in the future chapters.

In 1-D, the Gauss quadrature use n points to achieve the order $2n-1$ which is the highest order for n points. The Guass points are roots of orthogonal polynomials and can be found in almost all books on numerical analysis. We collect some quadrature rules for triangles and tetrahedron which is less well documented in the literature. We present the points in the barycentric coordinate $p = (\lambda_1, \cdots, \lambda_{d+1})$. The Cartesian coordinate of p is obtained by $\sum_{i=1}^{d+1}\lambda_i\mathbf{x}_i$. The high order rules are less desirable since too many points are needed.

The 2-D quadrature points can be found in the paper [**?**] and the 3-D case is in [**?**]. 16 digits accurate quadrature points is included in iFEM.

```
%% iFEM tutorial

%% ------------------------------Installation--------------------------------
% * Download iFEM https://bitbucket.org/ifem/ifem/get/tip.zip
% * Unzip the file to where you like.
% * In MATLAB, go to the iFEM folder .
% * Run setpath.m
% * https://www.math.uci.edu/¬chenlong/ifemdoc/introduction.html

%% ----------------------------Problem setting-------------------------------
%---------------u = sin(2*pi* x) * cos(2*pi* y)-----------------------------
%       -\Delta u = f          in \Omega = (0,1)^2
%               u = 0          on \partial \Omega
%               f = 8*pi^2 * sin(2*pi* x) * cos(2*pi* y)
uexact = @(coord) sin(2*pi*coord(:,1)).* sin(2*pi*coord(:,2));
f = @(coord) 8*pi^2 *sin(2*pi*coord(:,1)).*sin(2*pi*coord(:,2));
g_D = @(coord) zeros(size(coord,1),1);
```

```
%% Mesh data structure
%----------------Generate mesh for the unit square-------------------------
h = 1/2^8;

disp('# of dof:'); Nodf = (1/h)^2
[node,elem] = squaremesh([0,1,0,1],h);
% showmesh(node,elem);
% findnode(node);
% findelem(node,elem);


%------------set all boundary edges to Dirichlet type----------------------
bdFlag = setboundary(node,elem,'Dirichlet');  % bdFlag: the types of three
                                               %  vertices of each element

%-------extract boundary edges wiht Dirichlet BC and Neumann BC------------
totalEdge = [elem(:,[2,3]); elem(:,[3,1]); elem(:,[1,2])];
Dirichlet = totalEdge(bdFlag(:) == 1,:);
% findedge(node,Dirichlet);
Neumann = totalEdge(bdFlag(:) == 2,:);


%% DOF mapping
dofmap = elem; % for P1 Lagrange element


%% Assembling the stiffness matrix
% tic; disp('assemblingstandard'); A = assemblingstandard(node,elem); toc;
% tic; disp('assemblingsparse'); A = assemblingsparse(node,elem); toc;
tic; disp('assembling'); A = assembling(node,elem); toc;


%% right hand side  (f, \phi_i)
% midpoint quadrature rule  % f(mid_i) * lambda(mid_i) * w_i * area
mid1 = (node(elem(:,2),:)+node(elem(:,3),:))/2;
mid2 = (node(elem(:,3),:)+node(elem(:,1),:))/2;
mid3 = (node(elem(:,1),:)+node(elem(:,2),:))/2;
area = abs(simplexvolume(node,elem)) ;
i1 = elem(:,1);
bt1 = area.*(f(mid2)*.5+f(mid3)*.5)/3;     %\lambda_1
i2 = elem(:,2);
bt2 = area.*(f(mid3)*.5+f(mid1)*.5)/3;      %\lambda_2
i3 = elem(:,3);
bt3 = area.*(f(mid1)*.5+f(mid2)*.5)/3;      %\lambda_3
N = size(node,1);
b = accumarray([i1; i2; i3],[bt1;bt2;bt3],[N 1]);


%% Boundary condition
%-------------------- Dirichlet boundary conditions------------------------
%
% Au = b   => A(u_0+u_g) = b  =>   Au_0 = b - Au_g
%
isBdNode = false(N,1);
isBdNode(Dirichlet) = true;
bdNode = find(isBdNode);
freeNode = find(~isBdNode);
u = zeros(N,1);
u(bdNode) = g_D(node(bdNode,:));
b = b - A*u;
% %-------------------- Neumann boundary conditions -------------------------
```

```
% if (¬isempty(Neumann))
%     Nve = node(Neumann(:,1),:) - node(Neumann(:,2),:);
%     edgeLength = sqrt(sum(Nve.^2,2));
%     mid = (node(Neumann(:,1),:) + node(Neumann(:,2),:))/2;
%     b = b + accumarray([Neumann(:),ones(2*size(Neumann,1),1)], ...
%         repmat(edgeLength.*g_N(mid)*.5,2,1),[N,1]);
% end


%% solve the linear system
%    solve Au_0 = b - Au_g on the freeNode
u(freeNode) = A(freeNode,freeNode) \ b(freeNode);


%% plot the solution
%showresult(node,elem,u,[-62,58]);

%% compute L2 error and H1 error
 u_I = uexact(node);
 e = U_I - u; % u_I- u_h
 %  u -u_h
 disp('mesh size' ); all_h = [all_h h]
 disp('H1 error of (u_I-u_h)');all_error = [all_error  e'*A*e]
```

22.2 FEniCS tutorial

This section gives a concise and gentle introduction to finite element programming in Python based on the popular FEniCS software library. The readers are encouraged to learn more from the FEniCS tutorial [?] and the comprehensive FEniCS book [?]. FEniCS can be programmed in both C++ and Python, but this tutorial focuses exclusively on Python programming, since this is the simplest and most effective approach for beginners.

22.2.1 Preliminaries

Overview on FEniCS project

The FEniCS Project is a research and software project aimed at creating mathematical methods and software for automated computational mathematical modeling. This means creating easy, intuitive, efficient, and flexible software for solving partial differential equations (PDEs) using finite element methods. It consists of a number of software components: DOLFIN, FFC, FIAT, UFL, mshr and a few others.

- DOLFIN is the computational high-performance C++ backend of FEniCS. It implements data structures such as meshes, function spaces and functions, compute-intensive algorithms such as finite element assembly and mesh refinement, and the interfaces to linear algebra solvers and data structures such as PETSc. DOLFIN also implements the FEniCS problem-solving environment in both C++ and Python.
- UFL implements the abstract mathematical language by which users may express variational problems.
- FIAT is the finite element backend of FEniCS, responsible for generating finite element basis functions.
- FFC is the code generation engine of FEniCS (the form compiler), responsible for generating efficient C++ code from high-level mathematical abstractions.
- mshr provides FEniCS with mesh generation capabilities.

Obtaining the software

We highlight two main options for installing the FEniCS software: Docker containers and Ubuntu packages. While the Docker containers work on all operating systems, the Ubuntu packages only work on Ubuntu-based systems.

Installation using Docker containers

A modern solution to the challenge of software installation on diverse software platforms is to use so-called containers. To use FEniCS containers, you must first install the Docker platform. Docker installation is simple and instructions are available on the Docker web page[c0]. Once you have installed Docker, just copy the following line into a terminal window:

```
Terminal $> curl -s https://get.fenicsproject.org | bash
```

The command above will install the program *fenicsproject* on your system. This program lets you easily create FEniCS container on your system:

```
Terminal $> fenicsproject run
```

This command has several useful options, such as easily switching between the latest release of FEniCS, the latest development version and many more. To learn more, type *fenicsproject help*.

Installation using Ubuntu packages

For users of Ubuntu GNU/Linux, FEniCS can also be installed easily via the standard Ubuntu package manager *apt-get*. Just copy the following lines into a terminal window:

```
Terminal $>  sudo add-apt-repository ppa:fenics-packages/fenics
Terminal $>  sudo apt-get update
Terminal $>  sudo apt-get install fenics
Terminal $>  sudo apt-get dist-upgrade
```

This will add the FEniCS package archive (PPA) to your Ubuntu computer's list of software sources and then install FEniCS. It will will also automatically install packages for dependencies of FEniCS.

Testing your installation

Once you have installed FEniCS, you should make a quick test to see that your installation works properly. To do this, type the following command in a FEniCS-enabled[c0] terminal:

```
Terminal $> python  -c  import 'fenics'
```

If all goes well, you should be able to run this command without any error message.

[c0] https://www.docker.com
[c0] For users of FEniCS containers, this means first running the command *fenicsproject run*.

22.2.2 Solving the Poisson equation

In this section, we show how the Poisson equation, the most basic of all PDEs, can be quickly solved with a few lines of FEniCS code. We introduce the most fundamental FEniCS objects such as *Mesh*, *Function*, *FunctionSpace*, *TrialFunction*, and *TestFunction*, and learn how to write a basic PDE solver, including how to formulate the variational problem, apply boundary conditions, call the FEniCS solver, and plot the solution.

Solving boundary-value problems in FEniCS consists of the following steps:

1. Identify the computational domain Ω, the PDE, its boudary condition, and the source terms f.
2. Reformulate the PDE as a finite element variational problem.
3. Write a Python program which defines the computational domain, the variational problem, the boundary conditions, and the source terms, using the corresponding FEniCS abstraction.
4. Call FEniCS to solve the boundary-value problem and, optionally, extend the program to compute derived quantities such as fluxes and averages, and visualize the results.

Mathematical problem formulation

We consider the following boundary-value problem:

$$\begin{aligned} -\nabla^2 u(x) &= f(x), \quad x \in \Omega, \\ u(x) &= u_D(x), \quad x \in \partial\Omega. \end{aligned} \tag{22.6}$$

Here $u = u(x)$ is the unknown function, $f = f(x)$ is a prescribed function, ∇^2 is the Laplace operator, Ω is the spatial domain and $\partial\Omega$ is the boundary of Ω.

By multiplying a testing function with the PDE, integrating over the domain, and integration by parts to keep the order of the derivative as small as possible, it turns out the following canonical variational problems: find $u \in V$ such that

$$a(u, v) = L(v) \quad \forall v \in \hat{V}. \tag{22.7}$$

For the Poisson equation, we have:

$$a(u, v) = \int_\Omega \nabla u \cdot \nabla v \, dx, \qquad L(v) = \int_\Omega f v \, dx,$$

$$V = \{v \in H^1(\Omega) : v = u_D \text{ on } \partial\Omega\}, \qquad \hat{V} = \{v \in H^1(\Omega) : v = 0 \text{ on } \partial\Omega\}.$$

From the mathematics literature, $a(u, v)$ is known as a bilinear form and $L(v)$ as a linear form. We shall, in every linear problem we solve, identify the terms with the unkown u and collect them in $a(u, v)$, and similarly collect all terms with only known functions in $L(v)$. The formulas for a and L can then be expressed directly in our FEniCS programs.

Choosing a test problem

We just manufacture some quadratic function in 2D as the exact solution, say

$$u_e(x, y) = 1 + x^2 + 2y^2.$$

By inserting it into the Poisson equation, we have

$$f(x, y) = -6, \quad u_D = u_e(x, y) = 1 + x^2 + 2y^2,$$

regardless of the shape of the domain as long as u_e is prescribed along the boundary. Here, for simplicity, the domain is the unit square $\Omega = [0, 1] \times [0, 1]$.

FEniCS implementation

A FEniCS program for solving our test problem for the Poisson equation in 2D with the given choices of Ω, u_D , and f may look as follows:

```
from fenics import *

# Create mesh and define function space
mesh = UnitSquareMesh(8, 8)
V = FunctionSpace(mesh, P, 1)

# Define boundary condition
u_D = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]', degree=2)

def boundary(x, on_boundary):
    return on_boundary

bc = DirichletBC(V, u_D, boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0)
a = dot(grad(u), grad(v))*dx
L = f*v*dx

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# Plot solution and mesh
plot(u)
plot(mesh)

# Save solution to file in VTK format
vtkfile = File('poisson/solution.pvd')
vtkfile << u

# Compute error in L2 norm
error_L2 = errornorm(u_D, u, 'L2')

# Compute maximum error at vertices
vertex_values_u_D = u_D.compute_vertex_values(mesh)
vertex_values_u = u.compute_vertex_values(mesh)
import numpy as np
error_max = np.max(np.abs(vertex_values_u_D - vertex_values_u))

# Print errors
print('error_L2  =', error_L2)
print('error_max =', error_max)

# Hold plot
interactive()
```

Runing the program

The FEniCS program must be available in a plain text file, written with a text editor. The example program can be found in the file *~/demo/documented/poisson/python/demo_poisson.py*.

Open a terminal window, move to the directory containing the program and type the following command:

```
Terminal $> python demo_poisson.py
```

Note that this command must be run in a FEniCS-enabled terminal. For users of the FEniCS Docker containers, this means that you must type this command after you have started a FEniCS container using *fenicsproject run*.

Dissection of the program

We shall now dissect our FEniCS program in detail. The listed FEniCS program defines a finite element mesh, a finite element function space V on this mesh, boundary conditions for u (the function u_D), and the bilinear and linear forms $a(u, v)$ and $L(v)$. Thereafter, the solution u is computed. At the end of the program, we compare the numerical and the exact solutions. We also plot the solution using the *plot* command and save the solution to a file for external post-processing.

The important first line

The first line in the program,

```
1 from fenics import *
```

imports the key classes *UnitSquareMesh*, *FunctionSpace*, *Function*, and so forth, from the FEniCS library. All FEniCS programs for solving PDEs by the finite element method normally start with this line.

Generating simple meshes

The statement

```
1 mesh = UnitSquareMesh(8, 8)
```

defines a uniform finite element mesh over the unit square $[0, 1] \times [0, 1]$. The mesh consists of cells, which in 2D are triangles with straight sides. The parameters 8 and 8 specify that the square should be divided into 8×8 rectangles, each divided into a pair of triangles. The total number of triangles (cells) thus becomes 128. The total number of vertices in the mesh is $9 \cdot 9 = 81$.

Defining the finite element function space

Once the mesh has been created, we can create a finite element function space V:

```
1 V = FunctionSpace(mesh, 'P', 1)
```

The second argument ‘P’ specifies the type of element. The type of element here is P, implying the standard Lagrange family of elements. You may also use ‘Lagrange’ to specify this type of element. FEniCS supports all simplex element families and the notation defined in the Periodic Table of the Finite Elements[c0]. The third argument 1 specifies the degree of the finite element. In this case, the standard $P1$ linear Lagrange element, which is a triangle with nodes at the three vertices. Some finite element practitioners refer to this element as the ”linear triangle”. The computed solution u will be continuous across elements and linearly varying in x and y inside each element. Higher-degree polynomial approximations over each cell are trivially obtained by increasing the third parameter to *FunctionSpace*, which will then generate function spaces of type $P2$, $P3$, and so forth. Changing the second parameter to ‘DP’ creates a function space for discontinuous Galerkin methods.

[c0] https://www.femtable.org

Defining the trial and test functions

In mathematics, we distinguish between the trial and test spaces V and $\hat{V}$. The only difference in the present problem is the boundary conditions. In FEniCS we do not specify the boundary conditions as part of the function space, so it is suffcient to work with one common space V for both the trial and test functions in the program:

```
u = TrialFunction(V)
v = TestFunction(V)
```

Defining the boundary conditions

The next step is to specify the boundary condition: $u = u_D$ on $\partial\Omega$. This is done by

```
bc = DirichletBC(V, u_D, boundary)
```

where u_D is an expression defining the solution values on the boundary, and *boundary* is a function (or object) defining which points belong to the boundary.

The variable u_D refers to an *Expression* object, which is used to represent a mathematical function. The typical construction is

```
 u_D = Expression(formula, degree=1)
```

where *formula* is a string containing a mathematical expression. The formula must be written with C++ syntax and is automatically turned into an effcient, compiled C++ function.

The expression may depend on the variables x[0] and x[1] corresponding to the x and y coordinates. In 3D, the expression may also depend on the variable x[2] corresponding to the z coordinate. With our choice of $u_D(x, y) = 1 + x^2 + 2y^2$, the formula string can be written as 1+x[0]*x[0] + 2*x[1]*x[1]:

The function *boundary* specifies which points that belong to the part of the boundary where the boundary condition should be applied:

```
def boundary(x, on_boundary):
  return on_boundary
```

The argument *on_boundary* may also be omitted, but in that case we need to test on the value of the coordinates in x:

```
def boundary(x, on_boundary):
  return x[0] == 0 or x[1] == 0 or x[0] == 1 or x[1] == 1
```

Defining the source term

Before defining the bilinear and linear forms $a(u, v)$ and $L(v)$ we have to specify the source term f:

```
f = Expression('-6', degree=0)
```

When f is constant over the domain, f can be more effciently represented as a *Constant*:

```
f = Constant(-6)
```

Defining the variational problem

We now have all the ingredients we need to define the variational problem:

```
a = dot(grad(u), grad(v))*dx
L = f*v*dx
```

In essence, these two lines specify the PDE to be solved. Note the very close correspondence between the Python syntax and the mathematical formulas $\nabla u \cdot \nabla v \, dx$ and $f v \, dx$. This is a key strength of FEniCS: the formulas in the variational formulation translate directly to very similar Python code, a feature that makes it easy to specify and solve complicated PDE problems. The language used to express weak forms is called UFL (Unified Form Language) and is an integral part of FEniCS.

Forming and solving the linear system

Having defined the finite element variational problem and boundary condition, we can now ask FEniCS to compute the solution:

```
u = Function(V)
solve(a == L, u, bc)
```

Note that we first defined the variable u as a *TrialFunction* and used it to represent the unknown in the form a. Thereafter, we redefined u to be a *Function* object representing the solution; i.e., the computed finite element function u. This redefinition of the variable u is possible in Python and is often used in FEniCS applications for linear problems. The two types of objects that u refers to are equal from a mathematical point of view, and hence it is natural to use the same variable name for both objects.

Plotting the solution using the plot command

Once the solution has been computed, it can be visualized by the *plot* command:

```
plot(u)
plot(mesh)
interactive()
```

Note the call to the function *interactive* after the *plot* commands. This call makes it possible to interact with the plots (rotating and zooming). The call to *interactive* is usually placed at the end of a program that creates plots.

Plotting the solution using ParaView

22.3 Introduction to ANSYS Free Student Software

ANSYS provides free student software products perfect for work done outside the classroom, such as homework, capstone projects, student competitions and more. The renewable products can be downloaded by students across the globe. ANSYS Student products can be installed on any supported MS Windows 64-bit machine. There are three options you can access the free software and can make engineering simulations quickly.

Discovery Live Student Based on the brand new instantaneous simulation technology, ANSYS Discovery Live Student has integrated geometry modeling based on ANSYS SpaceClaim technology and a completely meshless and instantaneous thermal, structural, and fluids solvers. ANSYS Discovery Live Student lets you learn about physics without the headache of learning how to use a complex simulation tool. Be an early adopter and reap the benefits of the simulation tool of the future.

ANSYS AIM Student The next-generation academic simulation experience, ANSYS AIM Student has integrated geometry modeling based on ANSYS SpaceClaim technology and structural, fluids and electromagnetics solvers. ANSYS AIM Student is great if you are new to simulation and need an intuitive, cutting-edge simulation tool. Many professors are switching to ANSYS AIM, so join the revolution in engineering education.

ANSYS Student ANSYS Student is the ANSYS Workbench-based bundle of ANSYS Mechanical, ANSYS CFD, ANSYS Autodyn, ANSYS SpaceClaim and ANSYS DesignXplorer. ANSYS Student is used by hundreds of thousands of students globally. It is a great choice if your professor is already using it for your course or if you are already familiar with the ANSYS Workbench platform.

You can learn more about Discovery Live Student, ANSYS AIM Student and ANSYS Student from the website `https://www.ansys.com/academic/free-student-products`.

22.4 Computing exercises

Exercise 1. Using any computing language, for linear finite element method for the Poisson equation with pure Dirichlet boundary condition,

1. write a code for a uniform grid on the unit square and $(0,1)^2$ and solve the Poisson equation on the unit square with the exact solution given by

$$u(x,y) = \sin(x(1-x))y^2(1-y)^2. \tag{22.8}$$

   Assume that u_h is the linear finite element approximation defined on a uniform triangulation with mesh size h. Evaluate, for $h = 1/4, 1/8, 1/16$

$$\|u_I - u_h\|_{0,\Omega}, \quad |u_I - u_h|_{1,\Omega} \text{ and } \|u_I - u_h\|_{0,\infty,\Omega}$$

   where u_I is the interpolation of u. Determine the order of the above quantities with respect to h for each of the three different quadrature schemes (that are exact for $\mathcal{P}_k$ with $k = 0, 1, 2$ respectively) applied to evaluate the right hand side of the corresponding algebraic systems.
2. write a code for any general unstructured grid given by the data structures described in this chapter.

Exercise 2. Use iFEM to implement the finite element method for solving the Poisson equation in a general polygonal domain using the piecewise linear finite element.

Exercise 3. Use iFEM to implement some simple and popular finite element pairs for solving the Stokes equations in two dimensions.

Exercise 4. Use iFEM to implement numerical methods for solving time-dependent Navier-Stokes equations in two dimensions.