varFEM Documentation
Variational formulation based programming for finite element methods in Matlab

Yue Yu[*]

School of Mathematical Sciences, Institute of Natural Sciences, MOE-LSC, Shanghai Jiao Tong University, Shanghai, 200240, P. R. China.

Abstract

This is the documentation for the FEM package - varFEM, with a variety of examples presented to demonstrate the use of the software package. varFEM provides a simple way to realize the programming style in FreeFEM and makes extensive use of the numerical integration and assembly techniques in iFEM.

1 Introduction

FreeFEM is a popular 2D and 3D partial differential equations (PDE) solver based on finite element methods (FEMs) [2], which has been used by thousands of researchers across the world. The highlight is that the programming language is consistent with the variational formulation of the underlying PDEs, referred to as the variational formulation based programming in this article. We intend to develop an FEM package in a similar way of FreeFEM using the language of Matlab, named varFEM. The similarity here only refers to the programming style of the main or test script, not to the internal architecture of the software.

This programming paradigm is usually organized in an object-oriented language, which makes it difficult for readers or users to understand and modify the code, and further redevelop the package (although it is a good way to develop softwares). Upon rethinking the process of finite element programming, it becomes clear that the assembly of the stiffness matrix and load vector essentially reduces to the numerical integration of some typical bilinear and linear forms with respect to basis functions. In this regard, the package iFEM, written in Matlab, has provided robust, efficient, and easy-following codes for many mathematical and physical problems in both two and three dimensions [1]. On this basis, we successfully developed the variational formulation based programming for the conforming $\mathbb{P}_k$-Lagrange ($k \le 3$) FEMs in two and three dimensions by utilizing the Matlab language. The underlying idea can be generalized to other types of finite

[*]terenceyuyue@sjtu.edu.cn

elements for both two- and three-dimensional problems on unstructured simplicial meshes. The package is accessible on https://github.com/Terenceyuyue/varFEM (see the varFEM folder).

The article is organized as follows. In Section 2, we introduce the basic idea in varFEM through a model problem. In Section 3, we demonstrate the use of varFEM for several typical examples, including a complete implementation of the model problem, the vector finite element for the linear elasticity, the mixed FEMs for bihamonic equation and Stokes problem, the iterative scheme for the heat equation and the eigenvalue problem for the Poisson equation. We also demonstrate the ability of varFEM to solve complex problems and three-dimensional problems in Sections 4 and 5, respectively.

2 Variational formulation based programming in varFEM

We introduce the variational formulation based programming in varFEM via a model problem, so as to facilitate the underlying design idea.

2.1 Programming for a model problem

Let $\Omega = (0,1)^2$ and consider the second-order elliptic problem:

$$\begin{cases} -\nabla\cdot(a\nabla u) + cu = f & \text{in } \Omega, \\ u = g_D & \text{on } \Gamma_D, \\ g_R u + a\partial_n u = g_N & \text{on } \Gamma_R, \end{cases} \tag{2.1}$$

where Γ_D and $\Gamma_R = \partial\Omega\backslash\Gamma_D$ are the Dirichlet boundary and Robin boundary, respectively. For brevity, we refer to g_R as the Robin boundary data function and g_N as the Neumann boundary data function. For homogenous Dirichlet boundary condition, the variational problem is to find $u \in V := H_0^1(\Omega)$ such that

$$a(v,u) = \ell(v), \quad v \in V,$$

where

$$a(v,u) = \int_\Omega a\nabla v\cdot\nabla u \mathrm{d}\sigma + \int_\Omega cvu\mathrm{d}\sigma + \int_{\Gamma_R} g_R vu\mathrm{d}s,$$

$$\ell(v) = \int_\Omega fv\mathrm{d}\sigma + \int_{\Gamma_R} g_N v\mathrm{d}s.$$

Here, the test function is placed in the first entry of $a(\cdot,\cdot)$ since

$$a(v,u) = \boldsymbol{v}^T\boldsymbol{A}\boldsymbol{u}, \quad \boldsymbol{A} = (a(\varphi_i,\phi_j))_{m\times n},$$

where $\boldsymbol{v} = (v_1,\cdots,v_m)^T$ and $\boldsymbol{u} = (u_1,\cdots,u_n)^T$, with

$$v = \sum_{i=1}^m v_i\varphi_i, \quad u = \sum_{i=1}^n u_i\phi_i.$$

2.1.1 The assembly of bilinear forms

The first step is to obtain the stiffness matrix associated with the bilinear form

$$\int_\Omega a\nabla v\cdot\nabla u\mathrm{d}\sigma+\int_\Omega cvu\mathrm{d}\sigma$$

on the approximated domain, where for simplicity we have used the original notation Ω to represent a triangular mesh. The computation in varFEM reads

```
% Omega
Coef  = {a, c};
Test  = {'v.grad', 'v.val'};
Trial = {'u.grad', 'u.val'};
kk = int2d(Th,Coef,Test,Trial,Vh,quadOrder);
```

Here, `Th` represents the triangular mesh, which provides some necessary auxiliary data structures. We set up the triple `(Coef,Test,Trial)`, for the coefficients, test functions and trial functions in variational form, respectively. It is obvious that `v.grad` is for ∇v and `v.val` is for v itself. The routine `int2d.m` computes the stiffness matrix corresponding to the bilinear form on the two-dimensional region, i.e.

$$A=(a_{ij}),\quad a_{ij}=a(\Phi_i,\Phi_j),$$

where Φ_i are the global shape functions of the finite element space `Vh`. The integral of the bilinear form, as $(\nabla\Phi_i,\nabla\Phi_j)_\Omega$, will be approximated by using the Gaussian quadrature formula with `quadOrder` being the order of accuracy.

The second step is to compute the stiffness matrix for the bilinear form on the Robin boundary Γ_R:

$$\int_{\Gamma_R} g_R vu\mathrm{d}s.$$

The code can be written as follows.

```
% Gamma_R
Th.elem1d = Th.bdEdgeType{1};
Th.elem1dIdx = Th.bdEdgeIdxType{1};
Coef  = {g_R};
Test  = {'v.val'};
Trial = {'u.val'};
kk = kk + int1d(Th,Coef,Test,Trial,Vh,quadOrder);
```

Here, `int1d.m` gives the contribution to the stiffness matrix on the one-dimensional boundary edges of the mesh. Note that we must provide the connectivity list `elem1d` of the boundary edges of Γ_R and the associated indices `elem1dIdx` in the data structure `edge` introduced later.

2.1.2 The assembly of linear forms

For the linear forms, we first consider the integral for the source term:

$$\int_\Omega fv\mathrm{d}\sigma.$$

The load vector can be assembled as

```
% Omega
```

```
Coef = pde.f;  Test = 'v.val';
ff = int2d(Th,Coef,Test,[],Vh,quadOrder);
```

We set `Trial = []` to indicate the linear form.

The computation of the load vector associated with the Neumann boundary data function g_N, i.e.,

$$\int_{\Gamma_R} g_N v \mathrm{d}s$$

reads

```
% Gamma_R
Coef = g_N;  Test = 'v.val';
ff = ff + int1d(Th,Coef,Test,[],Vh,quadOrder);
```

2.2 Data structures for triangular meshes

We adopt the data structures given in *i*FEM [1]. All related data are stored in the Matlab structure `Th`, which is computed by using the subroutine FeMesh2d.m as

```
Th = FeMesh2d(node,elem,bdStr);
```

The triangular meshes are represented by two basic data structures `node` and `elem`, where `node` is an `N` $\times$ 2 matrix with the first and second columns contain x- and y-coordinates of the nodes in the mesh, and `elem` is an `NT` $\times$ 3 matrix recording the vertex indices of each element in a counterclockwise order, where `N` and `NT` are the numbers of the vertices and triangular elements.

In the current version, we only consider the $\mathbb{P}_k$-Lagrange finite element spaces with k up to 3. In this case, there are two important data structures `edge` and `elem2edge`. In the matrix `edge(1:NE,1:2)`, the first and second rows contain indices of the starting and ending points. The column is sorted in the way that for the k-th edge, `edge(k,1)<edge(k,2)` for $k = 1, 2, \cdots,$ `NE`. The matrix `elem2edge` establishes the map of local index of edges in each triangle to its global index in matrix `edge`. By convention, we label three edges of a triangle such that the i-th edge is opposite to the i-th vertex. We refer the reader to `https://www.math.uci.edu/~chenlong/ifemdoc/mesh/auxstructuredoc.html` for some detailed information.

To deal with boundary integrals, we first exact the boundary edges from `edge` and store them in matrix `bdEdge`. In the input of `FeMesh2d`, the string bdStr is used to indicate the interested boundary part in `bdEdge`. For example, for the unit square $\Omega = (0, 1)^2$,

- `bdStr = 'x==1'` divides `bdEdge` into two parts: `bdEdgeType{1}` gives the boundary edges on $x = 1$, and `bdEdgeType{2}` stores the remaining part.
- `bdStr = {'x==1','y==0'}` separates the boundary data `bdEdge` into three parts: `bdEdgeType{1}` and `bdEdgeType{2}` give the boundary edges on $x = 1$ and $y = 0$, respectively, and `bdEdgeType{3}` stores the remaining part.
- `bdStr = []` implies that `bdEdgeType{1} = bdEdge`.

We also use `bdEdgeIdxType` to record the index in matrix `edge`, and `bdNodeIdxType` to store the nodes for respective boundary parts. Note that we determine the boundary of interest by the coordinates of the midpoint of the edge, so `'x==1'` can also be replaced by a statement like `'x>0.99'`.

2.3 Code design of `int2d.m` and `assem2d.m`

In this article we only discuss the implementation of the bilinear forms in two dimensions.

2.3.1 The scalar case: `assem2d.m`

In this subsection we introduce the details of writing the subroutine `assem2d.m` to assemble a two-dimensional scalar bilinear form

$$a(v,u), \quad v=\varphi_i, \;\; u=\phi_j, \quad i=1,\cdots,m, \;\; j=1,\cdots,n,$$

where the test function v and the trial function u are allowed to match different finite element spaces, which can be found in mixed finite element methods for Stokes problems. For the scalar case, `assem2d.m` is essentially the same as `int2d.m`, while the later one can be used to deal with vector cases like linear elasticity problems. To handle different spaces, we write `Vh = {'P1', ... 'P2'}` for the input of `assem2d.m`, where `Vh{1}` is for v and `Vh{2}` is for u. For simplicity, it is also allowed to write `Vh = 'P1'` when v and u are in the same space.

Let us discuss the case where v and u lie in the same space. Suppose that the bilinear form contains only first-order derivatives. Then the possible combinations are

$$\int_K avu\mathrm{d}x, \quad \int_K av_xu\mathrm{d}x, \quad \int_K av_yu\mathrm{d}\sigma,$$
$$\int_K avu_x\mathrm{d}\sigma, \quad \int_K av_xu_x\mathrm{d}\sigma, \quad \int_K av_yu_x\mathrm{d}\sigma,$$
$$\int_K avu_y\mathrm{d}\sigma, \quad \int_K av_xu_y\mathrm{d}\sigma, \quad \int_K av_yu_y\mathrm{d}\sigma.$$

Of course, we often encounter the gradient form

$$\int_K a\nabla v\cdot\nabla u\mathrm{d}\sigma = \int_K a(v_xu_x+v_yu_y)\mathrm{d}\sigma.$$

We take the second bilinear form as an example. Let

$$a_K(v,u)=\int_K av_xu\mathrm{d}\sigma,$$

and consider the $\mathbb{P}_1$-Lagrange finite element. Denote the local basis functions to be ϕ_1,ϕ_2,ϕ_3. Then the local stiffness matrix is

$$A_K=\int_K a\begin{bmatrix}\partial_x\phi_1\\ \partial_x\phi_2\\ \partial_x\phi_3\end{bmatrix}\begin{bmatrix}\phi_1 & \phi_2 & \phi_3\end{bmatrix}\mathrm{d}\sigma.$$

Let

$$v_1=\partial_x\phi_1, \;\; v_2=\partial_x\phi_2, \;\; v_3=\partial_x\phi_3; \quad u_1=\phi_1, \;\; u_2=\phi_2, \;\; u_3=\phi_3.$$

Then

$$A_K = (k_{ij})_{3\times 3}, \quad k_{ij} = \int_K a v_i u_j \mathrm{d}\sigma.$$

The integral will be approximated by the Gaussian quadrature rule:

$$k_{ij} = \int_K a v_i u_j \mathrm{d}\sigma = |K| \sum_{p=1}^{n_g} w_p a(x_p, y_p) v_i(x_p, y_p) u_j(x_p, y_p),$$

where (x_p, y_p) is the p-th quadrature point. In the implementation, we in advance store the quadrature weights and the values of basis functions or their derivatives in the following form:

$$w_p, \qquad \mathtt{vi}(:,p) = \begin{bmatrix} v_i|_{(x_p^1,y_p^1)} \\ v_i|_{(x_p^2,y_p^2)} \\ \vdots \\ v_i|_{(x_p^{\mathrm{NT}},y_p^{\mathrm{NT}})} \end{bmatrix}, \qquad \mathtt{uj}(:,p) = \begin{bmatrix} u_j|_{(x_p^1,y_p^1)} \\ u_j|_{(x_p^2,y_p^2)} \\ \vdots \\ u_j|_{(x_p^{\mathrm{NT}},y_p^{\mathrm{NT}})} \end{bmatrix}, \qquad p = 1, \cdots, n_g,$$

where `vi` associated with v_i is of size $\mathtt{NT} \times \mathtt{ng}$ with the p-th column given by $\mathtt{vi}(:,p)$. Let $\mathtt{weight} = [w_1, w_2, \cdots, w_{n_g}]$ and `ww = repmat(weight, NT, 1)`. Then k_{ij} for $a = 1$ can be computed as

```
k11 = sum(ww.*v1.*u1,2);
k12 = sum(ww.*v1.*u2,2);
k13 = sum(ww.*v1.*u3,2);
k21 = sum(ww.*v2.*u1,2);
k22 = sum(ww.*v2.*u2,2);
k23 = sum(ww.*v2.*u3,2);
k31 = sum(ww.*v3.*u1,2);
k32 = sum(ww.*v3.*u2,2);
k33 = sum(ww.*v3.*u3,2);
K = [k11,k12,k13, k21,k22,k23, k31,k32,k33];
```

Here we have stored the local stiffness matrix A_K in the form of $[k_{11}, k_{12}, k_{13}, k_{21}, k_{22}, k_{23}, k_{31}, k_{32}, k_{33}]$, and stacked the results of all cells together. By adding the contribution of the area, one has

```
Ndof = 3;
K = repmat(area,1,Ndof^2).*K;
```

For the variable coefficient case, such as $a(x, y) = x + y$, one can further introduce the **coefficient matrix** as

```
cf = @(pz) pz(:,1) + pz(:,2); % x+y;
cc = zeros(NT,ng);
for p = 1:ng
    pz = lambda(p,1)*z1 + lambda(p,2)*z2 + lambda(p,3)*z3;
    cc(:,p) = cf(pz);
end
```

where `pz` are the quadrature points on all elements. The above procedure can be implemented as follows.

```
K = zeros(NT,Ndof^2);
s = 1;
v = {v1,v2,v3}; u = {u1,u2,u3};
for i = 1:Ndof
    for j = 1:Ndof
        vi = v{i}; uj = u{j};
        K(:,s) =  area.*sum(ww.*cc.*vi.*uj,2);
        s = s+1;
    end
end
```

The bilinear form is assembled by using the build-in function `sparse.m` as in *i*FEM. In this case, the code is given as

```
ss = K(:);
kk = sparse(ii,jj,ss,NNdof,NNdof);
```

Here, the triple `(ii, jj, ss)` is called the sparse index. Please refer to the following link: https://www.math.uci.edu/~chenlong/ifemdoc/fem/femdoc.html.

Remark 2.1. For the case where v and u are in different spaces, one just needs to modify the basis functions and the number of local degrees of freedom accordingly. The code can be presented as

```
s = 1;
for i = 1:Ndofv
    for j = 1:Ndofu
        vi = vbase{i}; uj = ubase{j};
        K(:,s) =  K(:,s) + area.*sum(ww.*cc.*vi.*uj,2);
        s = s+1;
    end
end
```

In varFEM, we use `Base2d.m` to load the information of `vi` and `uj`, for example, the following code gives the values of $\partial_x \phi$, where ϕ is a local basis function.

```
v = 'v.dx';
vbase = Base2d(v,node,elem,Vh{1},quadOrder); % v1.dx, v2.dx, v3.dx
```

2.3.2 The vector case: `int2d.m`

Let us consider a typical bilinear form for linear elasticity problems, given as

$$a_K(\boldsymbol{v},\boldsymbol{u}) := \int_K \boldsymbol{\varepsilon}(\boldsymbol{v}) : \boldsymbol{\varepsilon}(\boldsymbol{u}) \mathrm{d}\sigma,$$

where $\boldsymbol{v} = (v_1, v_2)^T$, $\boldsymbol{u} = (u_1, u_2)^T$, and

$$\begin{aligned} \boldsymbol{\varepsilon}(\boldsymbol{v}) : \boldsymbol{\varepsilon}(\boldsymbol{u}) &= v_{1,x}u_{1,x} + v_{2,y}u_{2,y} + \frac{1}{2}(v_{1,y} + v_{2,x})(u_{1,y} + u_{2,x}) & (2.2) \\ &= v_{1,x}u_{1,x} + v_{2,y}u_{2,y} + \frac{1}{2}(v_{1,y}u_{1,y} + v_{1,y}u_{2,x} + v_{2,x}u_{1,y} + v_{2,x}u_{2,x}). & (2.3) \end{aligned}$$

The stiffness matrix can be assembled as

```
Coef  = {1, 1, 0.5, 0.5, 0.5, 0.5};
Test  = {'v1.dx', 'v2.dy', 'v1.dy', 'v1.dy', 'v2.dx', 'v2.dx'};
Trial = {'u1.dx', 'u2.dy', 'u1.dy', 'u2.dx', 'u1.dy', 'u2.dx'};
kk = int2d(Th, Coef, Test, Trial, Vh, quadOrder);
```

We also provide the subroutine `getExtendedvarForm.m` to get the extended combinations (2.3) from (2.2), which has been included in `int2d.m`. Therefore, the bilinear form can be directly assembled as

```
Coef = { 1, 1, 0.5 };
Test  = {'v1.dx', 'v2.dy', 'v1.dy + v2.dx'};
Trial = {'u1.dx', 'u2.dy', 'u1.dy + u2.dx'};
kk = int2d(Th, Coef, Test, Trial, Vh, quadOrder);
```

In the rest of this subsection, we briefly discuss the sparse index. Let $\boldsymbol{v} = (v_1, v_2, v_3)$ and $\boldsymbol{u} = (u_1, u_2, u_3)$, and suppose that $a(\boldsymbol{v}, \boldsymbol{u})$ is a bilinear form. Note that, in general, $v_i (i = 1, 2, 3)$ can be in different spaces, but v_i and u_i are in the same space, otherwise the resulting stiffness matrix is not a square matrix. The stiffness matrix after blocking has the following correspondence:

$$a(\boldsymbol{v}, \boldsymbol{u}) \qquad \leftrightarrow \qquad [\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3] \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} \boldsymbol{u}_1 \\ \boldsymbol{u}_2 \\ \boldsymbol{u}_3 \end{bmatrix},$$

where $\boldsymbol{u}_i$ is the vector of degrees of freedom of u_i. It is easy to see that A_{ij} can obtained as in scalar case by assembling all pairs that contain (v_i, u_j) in $a(\boldsymbol{v}, \boldsymbol{u})$.

Let the sparse index for A_{ij} be $(\boldsymbol{i}_{ij}, \boldsymbol{j}_{ij}, \boldsymbol{s}_{ij})$. Let the numbers of rows and columns of A_{ij} be m_i and n_j, respectively. Then the final sparse assembly index `ii` and `jj` can be written in block matrix as

$$\begin{bmatrix} \boldsymbol{i}_{11} & \boldsymbol{i}_{12} & \boldsymbol{i}_{13} \\ \boldsymbol{i}_{21} + m_1 & \boldsymbol{i}_{22} + m_1 & \boldsymbol{i}_{23} + m_1 \\ \boldsymbol{i}_{31} + m_2 & \boldsymbol{i}_{32} + m_2 & \boldsymbol{i}_{33} + m_2 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} \boldsymbol{j}_{11} & \boldsymbol{j}_{12} + n_1 & \boldsymbol{j}_{13} + n_2 \\ \boldsymbol{j}_{21} & \boldsymbol{j}_{22} + n_1 & \boldsymbol{j}_{23} + n_2 \\ \boldsymbol{j}_{31} & \boldsymbol{j}_{32} + n_1 & \boldsymbol{j}_{33} + n_2 \end{bmatrix},$$

and obtained by straightening them as a column vector along the row vectors.

3 Textbook examples

In this section, we present several examples that frequently encountered in textbooks to demonstrate the use of varFEM.

3.1 Poisson-type problems

We now provide the complete implementation of the model problem (2.1):

$$\begin{cases} -\nabla \cdot (a\nabla u) + cu = f & \text{in } \Omega, \\ u = g_D & \text{on } \Gamma_D, \\ g_R u + a\partial_n u = g_N & \text{on } \Gamma_R. \end{cases}$$

The PDE stored is generated by `Poissondatavar.m`.

The function file reads

```
function uh = varPoisson(Th,pde,Vh,quadOrder)

%% Assemble stiffness matrix
% Omega
Coef  = {pde.a, pde.c};
Test  = {'v.grad', 'v.val'};
```

```
 7 Trial = {'u.grad', 'u.val'};
 8 kk = assem2d(Th,Coef,Test,Trial,Vh,quadOrder);
 9
10 % Robin data
11 bdStr = Th.bdStr;
12 if ¬isempty(bdStr)
13     Th.on = 1;
14     %Th.elem1d = Th.bdEdgeType{1};
15     %Th.elem1dIdx = Th.bdEdgeIdxType{1};
16     Coef  = pde.g_R;  Test  = 'v.val';  Trial = 'u.val';
17     kk = kk + assem1d(Th,Coef,Test,Trial,Vh,quadOrder);
18 end
19
20 %% Assemble the right hand side
21 % Omega
22 Coef = pde.f;  Test = 'v.val';
23 ff = assem2d(Th,Coef,Test,[],Vh,quadOrder);
24 % Neumann data
25 if ¬isempty(bdStr)
26     %Coef = @(p) pde.g_R(p).*pde.uexact(p) + pde.a(p).*(pde.Du(p)*n');
27
28     fun = @(p) pde.g_R(p).*pde.uexact(p);
29     Cmat1 = interpEdgeMat(fun,Th,quadOrder);
30     fun = @(p) repmat(pde.a(p),1,2).*pde.Du(p);
31     Cmat2 = interpEdgeMat(fun,Th,quadOrder);
32     Coef = Cmat1 + Cmat2;
33
34     ff = ff + assem1d(Th,Coef,Test,[],Vh,quadOrder);
35 end
36
37 %% Apply Dirichlet boundary conditions
38 g_D = pde.g_D;
39 on = 2 - 1*isempty(bdStr); % 1 for bdStr= [], 2 for bdStr = 'x==0'
40 uh = apply2d(on,Th,kk,ff,Vh,g_D);
```

In the above code, the structure `pde` stores the information of the PDE, including the exact solution `pde.uexact`, the gradient `pde.Du`, etc. The Neumann data function is $g_N = g_R u + a\partial_n u$, which varies on the boundary edges. For testing purposes, we compute this function by using the exact solution. In Lines 28-32, we use the subroutine `interpEdgeMat.m` to derive the coefficient matrix as in Subsect. 2.3.1. We remark that the `Coef` has three forms:

1. A function handle or a constant.
2. The numerical degrees of freedom of a finite element function.
3. A coefficient matrix resulting from the numerical integration.

In the computation, the first two forms in fact will be transformed to the third one.

In this example, the boundary edges will be divided into at most two parts. For example, when `bdStr = 'x==0'`, the left-end boundary is labelled by 1, and the remaining parts are labelled by 2. In this case, we use `Th.on = 1` to indicate the edges for boundary integral. One can also replace `Th.on = 1` by the following data structures:

```
1 Th.elem1d = Th.bdEdgeType{1};
2 Th.elem1dIdx = Th.bdEdgeIdxType{1};
```

The test script is presented as follows (see `main_varPoisson.m`).

```
%% Parameters
maxIt = 5;
N = zeros(maxIt,1);
h = zeros(maxIt,1);
ErrL2 = zeros(maxIt,1);
ErrH1 = zeros(maxIt,1);

%% Generate an intitial mesh
[node,elem] = squaremesh([0 1 0 1],0.5);
bdStr = 'x==0';

%% Get PDE data
pde = Poissondatavar;
g_R = @(p) 1 + p(:,1) + p(:,2); % 1 + x + y
pde.g_R = g_R;

%% Finite Element
i = 1; % 1,2,3
Vh = ['P', num2str(i)];
quadOrder = i+2;
for k = 1:maxIt
    % refine mesh
    [node,elem] = uniformrefine(node,elem);
    % get the mesh information
    Th = FeMesh2d(node,elem,bdStr);
    % solve the equation
    uh = varPoisson(Th,pde,Vh,quadOrder);
    % record and plot
    N(k) = size(elem,1);
    h(k) = 1/(sqrt(size(node,1))-1);
    if N(k) < 2e3  % show mesh and solution for small size
        figure(1);
        showresult(node,elem,pde.uexact,uh);
        drawnow;
    end
    % compute error
    ErrL2(k) = varGetL2Error(Th,pde.uexact,uh,Vh,quadOrder);
    ErrH1(k) = varGetH1Error(Th,pde.Du,uh,Vh,quadOrder);
end

%% Plot convergence rates and display error table
figure(2);
showrateh(h,ErrH1,ErrL2);
fprintf('\n');
disp('Table: Error')
colname = {'#Dof','h','||u-u_h||','|u-u_h|_1'};
disptable(colname,N,[],h,'%0.3e',ErrL2,'%0.5e',ErrH1,'%0.5e');
```

In the `for` loop, we first load or generate the mesh, which immediately returns the matrix `node` and `elem` to the Matlab workspace. Then we set up the boundary conditions to get the structural information. The subroutine `varPoisson.m` is the function file containing all source code to implement the FEM as given before. When obtaining the numerical solutions, we can visualize the solutions by using the subroutines `showresult.m`. We then calculate the discrete L^2 and H^1 errors via the subroutines `varGetL2Error.m` and `varGetH1Error.m`. The procedure is completed by verifying the rate of convergence through `showrateh.m`.

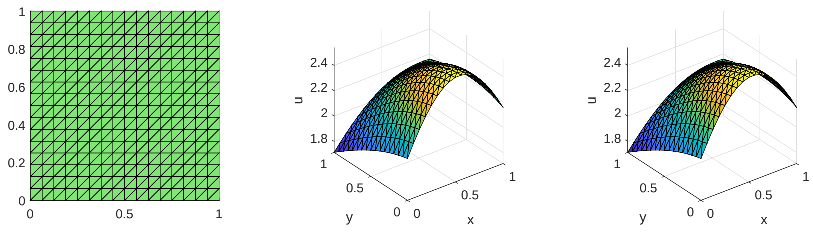
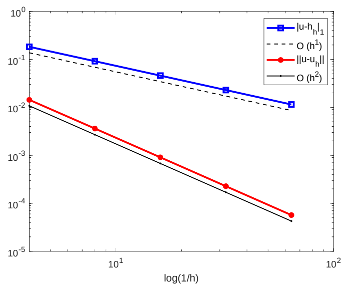
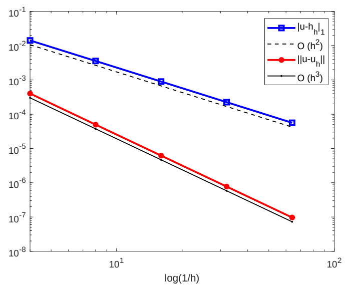


Fig. 1: The nodal values of exact and numerical solutions for the model problem

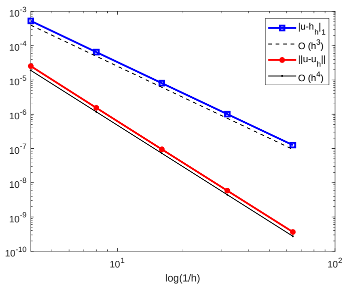
The test script can be easily used to compute $\mathbb{P}_i$-Lagrange element for $i = 1, 2, 3$ (see Line 18 in the test script). The nodal values for the model problem are displayed in Fig. 1. The rates of convergence are shown in Fig. 2, from which we observe the optimal convergence for all cases.



(a) $i = 1$



(b) $i = 2$



(c) $i = 3$

Fig. 2: Convergence rates for the model problem.

We also print the errors on the Matlab command window by using `disptable.m`, with the results for $\mathbb{P}_3$ element given below:

```
Table: Error
    #Dof        h          ||u-u_h||      |u-u_h|_1

    ----    ---------    -----------    -----------


      32    2.500e-01    2.53816e-05    5.28954e-04
     128    1.250e-01    1.53744e-06    6.50683e-05
     512    6.250e-02    9.46298e-08    8.07922e-06
    2048    3.125e-02    5.87001e-09    1.00695e-06
    8192    1.562e-02    3.65444e-10    1.25698e-07
```

The function `disptable.m` is given in iFEM using the build-in function `disp.m`. We give a simple and well-printed implementation just using the built-in function `table.m`.

We next consider the example with a circular domain or an L-shaped domain (see `main_varPoisson_pdetool.m`). Such a domain can be generated by using the pdetool as

```
%% Generate an intitial mesh
g = 'circleg'; %'lshapeg';
[p,e,t] = initmesh(g,'hmax',0.5);
node = p'; elem = t(1:3,:)';
bdStr = 'x>0'; % string for Neumann
```

The results are given in Fig. 3

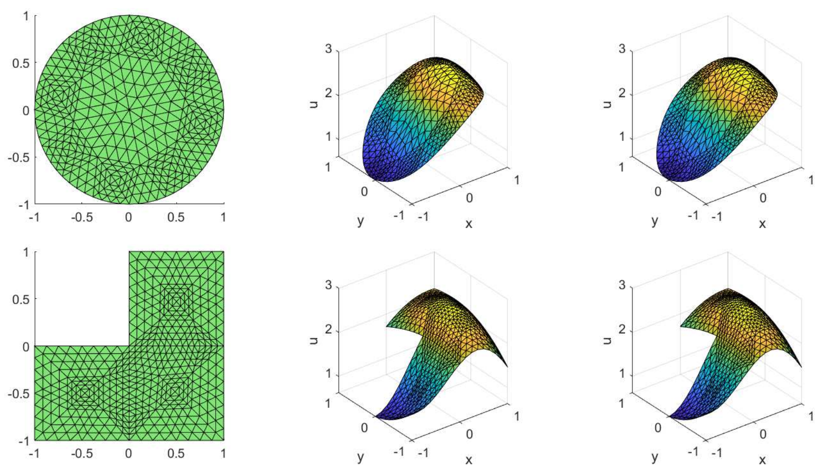


Fig. 3: The nodal values of exact and numerical solutions on circular domain or L-shaped region.

3.2 Linear elasticity problems

The linear elasticity problem is

$$\begin{cases} -\mathrm{div}\boldsymbol{\sigma} = \boldsymbol{f} & \text{in} \quad \Omega, \\ \boldsymbol{u} = \boldsymbol{0} & \text{on} \quad \Gamma_0, \\ \boldsymbol{\sigma n} = \boldsymbol{g} & \text{on} \quad \Gamma_1, \end{cases} \tag{3.1}$$

where $\boldsymbol{n} = (n_1, n_2)^T$ denotes the outer unit vector normal to $\partial\Omega$. The constitutive relation for linear elasticity is

$$\boldsymbol{\sigma}(\boldsymbol{u}) = 2\mu\boldsymbol{\varepsilon}(\boldsymbol{u}) + \lambda(\mathrm{div}\boldsymbol{u})\boldsymbol{I},$$

where $\boldsymbol{\sigma} = (\sigma_{ij})$ and $\boldsymbol{\varepsilon} = (\varepsilon_{ij})$ are the second order stress and strain tensors, respectively, satisfying $\varepsilon_{ij} = \frac{1}{2}(\partial_i u_j + \partial_j u_i)$, λ and μ are the Lamé constants, $\boldsymbol{I}$ is the identity matrix, and $\mathrm{div}\boldsymbol{u} = \partial_1 u_1 + \partial_2 u_2$.

3.2.1 The variational formulation of displacement type

The vector problem can be solved in block form by using `assem2d.m` as scalar cases. The equilibrium equation in (3.1) can also be written in the form

$$-\mu\Delta\boldsymbol{u} - (\lambda + \mu)\mathrm{grad}(\mathrm{div}\boldsymbol{u}) = \boldsymbol{f} \quad \text{in } \Omega, \tag{3.2}$$

which is referred to as the displacement type in what follows. In this case, we can only consider the pure displacement problem, i.e., $\Gamma_0 = \Gamma := \partial\Omega$. The first term $\Delta\boldsymbol{u}$ can be treated as the vector case of the Poisson equation. The variational formulation is

$$\mu\int_\Omega \nabla\boldsymbol{u} \cdot \nabla\boldsymbol{v}\mathrm{d}x + (\lambda + \mu)\int_\Omega (\mathrm{div}\ \boldsymbol{u})(\mathrm{div}\ \boldsymbol{v})\mathrm{d}\sigma = \int_\Omega \boldsymbol{f} \cdot \boldsymbol{v}\mathrm{d}\sigma.$$

The first term of the bilinear form can be split into

$$\mu\int_\Omega \nabla u_1 \cdot \nabla v_1 \mathrm{d}\sigma \quad \text{and} \quad \mu\int_\Omega \nabla u_2 \cdot \nabla v_2 \mathrm{d}\sigma.$$

They generate the same matrix, denoted A, corresponding to the blocks A_{11} and A_{22}, respectively. The computation reads

```
% (v1.grad, u1.grad), (v2.grad, u2.grad)
cf = 1;
Coef  = cf;  Test  = 'v.grad';  Trial = 'u.grad';
A = assem2d(Th,Coef,Test,Trial,Vh,quadOrder);
```

The second term of the bilinear form has the following combinations:

$$\int_\Omega v_{1,x}u_{1,x}\mathrm{d}x, \quad \int_\Omega v_{1,x}u_{2,y}\mathrm{d}x, \quad \int_\Omega v_{2,y}u_{1,x}\mathrm{d}x \quad \int_\Omega v_{2,y}u_{2,y}\mathrm{d}x,$$

which correspond to A_{11}, A_{12}, A_{21} and A_{22}, respectively, and can be computed as follows.

```
% (v1.dx, u1.dx)
cf = 1;
Coef  = cf;  Test  = 'v.dx';  Trial = 'u.dx';
B1 = assem2d(Th,Coef,Test,Trial,Vh,quadOrder);
% (v1.dx, u2.dy)
cf = 1;
Coef  = cf;  Test  = 'v.dx';  Trial = 'u.dy';
B2 = assem2d(Th,Coef,Test,Trial,Vh,quadOrder);
% (v2.dy, u1.dx)
cf = 1;
Coef  = cf;  Test  = 'v.dy';  Trial = 'u.dx';
B3 = assem2d(Th,Coef,Test,Trial,Vh,quadOrder);
% (v2.dy, u2.dy)
cf = 1;
Coef  = cf;  Test  = 'v.dy';  Trial = 'u.dy';
B4 = assem2d(Th,Coef,Test,Trial,Vh,quadOrder);
```

The block matrix is then given by

```
% kk
kk = [  mu*A+(lambda+mu)*B1,          (lambda+mu)*B2;
             (lambda+mu)*B3,     mu*A+(lambda+mu)*B4   ];
kk = sparse(kk);
```

The right-hand side has two components:

$$\int_\Omega f_1 v_1 \mathrm{d}x \quad \text{and} \quad \int_\Omega f_2 v_2 \mathrm{d}x.$$

The load vector is assembled in the following way:

```
%% Assemble right hand side
% F1
Coef = @(pz) pde.f(pz)*[1;0];  Test = 'v.val';
F1 = assem2d(Th,Coef,Test,[],Vh,quadOrder);
% F2
Coef = @(pz) pde.f(pz)*[0;1];  Test = 'v.val';
F2 = assem2d(Th,Coef,Test,[],Vh,quadOrder);
% F
ff = [F1; F2];
```

For the vector problem, we impose the Dirichlet boundary value conditions as follows.

```
%% Apply Dirichlet boundary conditions
g_D = pde.g_D;
on = 1;
g_D1 = @(p) g_D(p)*[1;0];
g_D2 = @(p) g_D(p)*[0;1];
```

```
gBc = {g_D1,g_D2};
Vhvec = {Vh,Vh};
u = apply2d(on,Th,kk,ff,Vhvec,gBc); % note Vhvec
```

Here, g_D1 is for u_1 and g_D2 is for u_2. Note that the finite element spaces Vhvec must be given in the same structure of gBc.

The solutions are displayed in Fig. 4 (The test script is main_varElasticityNavier_block.m).

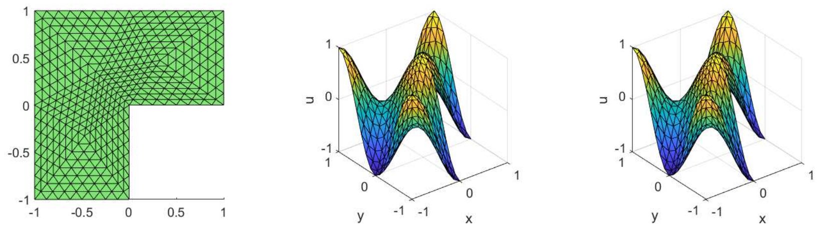


Fig. 4: The nodal values of exact and numerical solutions for the elasticity problem of displacement type.

3.2.2 The variational formulation of tensor type

The bilinear form is

$$a(\boldsymbol{v},\boldsymbol{u}) = 2\mu\int_\Omega \varepsilon_{ij}(\boldsymbol{v})\varepsilon_{ij}(\boldsymbol{u})\mathrm{d}\sigma + \lambda\int_\Omega (\partial_i v_i)(\partial_j u_j)\mathrm{d}\sigma,$$

where the summation is omitted. We can still assemble the stiffness matrix in block form as before (see main_varElasticity_block.m). Here we only present the implementation of vectorial FEMs using int2d.m. The computation of the first term has been given in Subsect. 2.3.2, i.e.,

```
% (Eij(u):Eij(v))
Coef = { 1, 1, 0.5 };
Test  = {'v1.dx', 'v2.dy', 'v1.dy + v2.dx'};
Trial = {'u1.dx', 'u2.dy', 'u1.dy + u2.dx'};
A = int2d(Th,Coef,Test,Trial,Vh,quadOrder);
A = 2*mu*A;
```

The second term can be computed as

```
% (div u,div v)
Coef = 1;
Test  = 'v1.dx + v2.dy' ;
Trial = 'u1.dx + u2.dy';
B = int2d(Th,Coef,Test,Trial,Vh,quadOrder);
B = lambda*B;
% stiffness matrix
kk = A + B;
```

The linear form is

$$\ell(\boldsymbol{v}) = \int_\Omega \boldsymbol{f}\cdot\boldsymbol{v}\mathrm{d}\sigma + \int_{\Gamma_1} \boldsymbol{g}\cdot\boldsymbol{v}\mathrm{d}s.$$

For the first term, one has

```
Coef = pde.f;  Test = 'v.val';
ff = int2d(Th,Coef,Test,[],Vh,quadOrder);
```

Note that we have added the implementation for $\boldsymbol{f} \cdot \boldsymbol{v}$ by just setting Test = 'v.val'. For the second term, one can compute the coefficient matrix for the boundary integral using interpEdgeMat.m and the Neumann condition is then realized as

```
%% Assemble Neumann boundary conditions
if ¬isempty(Th.bdStr)
    g_N = pde.g_N; trg = eye(3);

    Th.on = 1;
    g1 = @(p) g_N(p)*trg(:,[1,3]);   Cmat1 = interpEdgeMat(g1,Th,quadOrder);
    g2 = @(p) g_N(p)*trg(:,[3,2]);   Cmat2 = interpEdgeMat(g2,Th,quadOrder);


    Coef = {Cmat1, Cmat2};  Test = 'v.val';
    ff = ff + int1d(Th,Coef,Test,[],Vh,quadOrder);
end
```

Here, the data g_N is stored as $g_N = [\sigma_{11}, \sigma_{22}, \sigma_{12}]$ in the structure pde.

The Dirichlet condition can be handled as the displacement type:

```
%% Apply Dirichlet boundary conditions
on = 2 - 1*isempty(bdStr);
g_D1 = @(pz) pde.g_D(pz)*[1;0];
g_D2 = @(pz) pde.g_D(pz)*[0;1];
g_D = {g_D1, g_D2};
u = apply2d(on,Th,kk,ff,Vh,g_D);
```

Note that Vh is of vector form.

In the test script (see main_varElasticity.m), we set bdStr = 'y==0 | x==1'. The errors for the $\mathbb{P}_3$ element are listed in Tab. 1.

Tab. 1: The L^2 and H^1 errors for the elasticity problem of tensor form ($\mathbb{P}_3$ element)

♯Dof	h	ErrL2	ErrH1
32	2.500e-01	6.82177e-04	1.34259e-02
128	1.250e-01	3.90894e-05	1.61503e-03
512	6.250e-02	2.32444e-06	1.97527e-04
2048	3.125e-02	1.42085e-07	2.44506e-05
8192	1.562e-02	8.79363e-09	3.04301e-06

3.3 Biharmonic equation with mixed form

This subsection considers the mixed finite elements for biharmonic equation.

3.3.1 The mixed form of the biharmonic equation

We first consider the biharmonic equation with homogenous Dirichlet boundary conditions:

$$\begin{cases} \Delta^2 u = f & \text{in } \ \Omega \subset \mathbb{R}^2, \\ u = \partial_n u = 0 & \text{on } \ \partial\Omega. \end{cases}$$

By introducing a new variable $w = -\Delta u$, the above problem can be written in a mixed form as

$$\begin{cases} -\Delta u = w, \\ -\Delta w = f, \\ u = \partial_n u = 0 \quad \text{on } \partial\Omega. \end{cases}$$

The associated variational problem is: Find $(w, u) \in H^1(\Omega) \times H_0^1(\Omega) =: V \times U$ such that

$$\begin{cases} \displaystyle\int_\Omega \nabla u \cdot \nabla \phi \mathrm{d}\sigma = \int_\Omega w\phi \mathrm{d}\sigma, \quad \phi \in H^1(\Omega), \\ \displaystyle\int_\Omega \nabla w \cdot \nabla \psi \mathrm{d}\sigma = \int_\Omega f\psi \mathrm{d}\sigma, \quad \psi \in H_0^1(\Omega). \end{cases} \tag{3.3}$$

Let

$$a(w, \phi) = -\int_\Omega w\phi \mathrm{d}\sigma \quad \text{and} \quad b(\phi, u) = \int_\Omega \nabla\phi \cdot \nabla u \mathrm{d}\sigma.$$

One has

$$\begin{cases} a(w, \phi) + b(\phi, u) = 0, & \phi \in H^1(\Omega) = V, \\ b(w, \psi) = (f, \psi), & \psi \in H_0^1(\Omega) = U, \end{cases}$$

where $a(\cdot,\cdot) : V \times V \to \mathbb{R}$ and $b(\cdot,\cdot) : V \times U \to \mathbb{R}$.

The functions u and w will be approximated by $\mathbb{P}_1$-Lagrange elements. Let N be the vector of global basis functions. One easily gets

$$\begin{cases} \boldsymbol{\phi}^T A\boldsymbol{w} + \boldsymbol{\phi}^T B\boldsymbol{u} = 0, \\ \boldsymbol{\psi}^T B^T\boldsymbol{w} = \boldsymbol{\psi}^T\boldsymbol{f}, \end{cases}$$

where

$$A = -\int_\Omega N^T N \mathrm{d}\sigma, \qquad B = \int_\Omega \nabla N^T \cdot \nabla N \mathrm{d}\sigma, \qquad \boldsymbol{f} = \int_\Omega N^T f \mathrm{d}\sigma.$$

Note that the local form has the similar structure. The system can be written in block matrix form as

$$\begin{bmatrix} A & B \\ B^T & O \end{bmatrix} \begin{bmatrix} \boldsymbol{w} \\ \boldsymbol{u} \end{bmatrix} = \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{f} \end{bmatrix}. \tag{3.4}$$

Remark 3.1. If $\partial_n u$ does not vanish on $\partial\Omega$, then the mixed variational formulation is

$$\begin{cases} a(w, \phi) + b(\phi, u) = \displaystyle\int_{\partial\Omega} \partial_n u \phi \mathrm{d}s, & \phi \in V, \\ b(w, \psi) = (f, \psi), & \psi \in U. \end{cases}$$

Here, $\partial_n u$ corresponds to the Neumann boundary data for u in the first equation.

3.3.2 The programming in scalar form

We first assemble the linear system by using the block structure in (3.4).

In block matrix form, the stiffness matrix can be computed as follows.

```
%% Assemble stiffness matrix
% matrix A
Coef = 1;  Test = 'v.val';  Trial = 'u.val';
```

```
A = -assem2d(Th,Coef,Test,Trial,'P1',quadOrder);
% matrix B
Coef = 1;  Test = 'v.grad';  Trial = 'u.grad';
B = assem2d(Th,Coef,Test,Trial,'P1',quadOrder);
% kk
O = zeros(size(B));
kk = [A,  B;  B', O];
kk = sparse(kk);
```

The right-hand side is given by

```
%% Assemble right-hand side
Coef = pde.f;  Test = 'v.val';
ff = assem2d(Th,Coef,Test,[],'P1',quadOrder);
O = zeros(size(ff));
ff = [O; ff];
```

The computation of the Neumann boundary condition reads

```
%% Assemble Neumann boundary condition
Th.elem1d = Th.bdEdge; % all boundary edges
%Th.bdEdgeIdx1 = Th.bdEdgeIdx;
%Coef = @(p) pde.Du(p)*n;
Coef = interpEdgeMat(pde.Du,Th,quadOrder);
Test = 'v.val';
ff(1:N) = ff(1:N) + assem1d(Th,Coef,Test,[],'P1',quadOrder);
```

We finally impose the Dirichlet boundary conation as

```
1 %% Apply Dirichlet boundary conditions
2 on = 1;
3 g_D = pde.g_D;
4 gBc = {[],g_D};
5 Vhvec = {'P1','P1'};
6 U = apply2d(on,Th,kk,ff,Vhvec,gBc); % note Vhvec
7 w = U(1:N);  u = U(N+1:end);
```

Note that the Dirichlet data is only for u, so we set `gBc{1} = []` in Line 4.

The test script is `main_varBiharmonicMixedFEM_block.m`. The convergence rates for u and w are shown in Fig. 5, from which we clearly observe the first-order and the second-order convergence in the H^1 norm and L^2 norm for both variables.

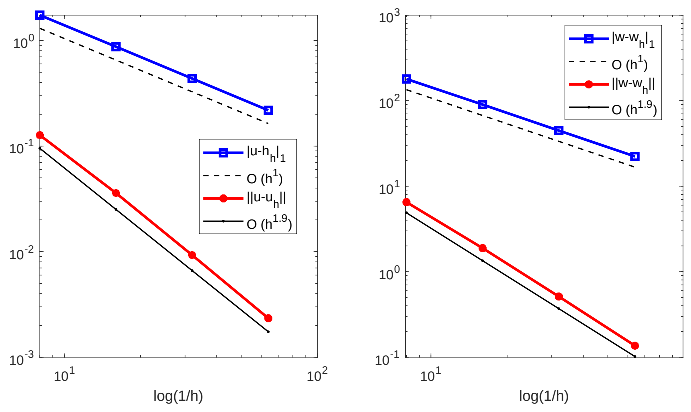


Fig. 5: The convergence rates of the mixed FEM for the biharmonic equation ($\mathbb{P}_1$ element)

3.3.3 The programming in vector form

Let $u_1 = w$, $u_2 = u$, $v_1 = \phi$ and $v_2 = \psi$. Then the problem (3.3) can be regarded as a variational problem of vector form, with $\boldsymbol{u} = (u_1, u_2)^T$ being the trial function and $\boldsymbol{v} = (v_1, v_2)^T$ being the test function. One easily finds that the mixed form (3.3) is equivalent to the following vector form

$$\int_\Omega (-v_1 u_1 + \nabla v_1 \cdot \nabla u_2 + \nabla v_2 \cdot \nabla u_1) \mathrm{d}\sigma = \int_\Omega f v_2 \mathrm{d}x + \int_{\partial\Omega} g_2 v_1 \mathrm{d}s,$$

which is obtained by adding the two equations.

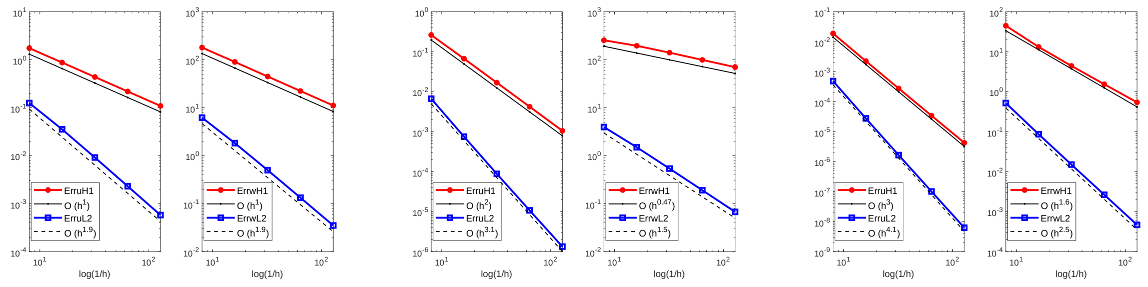
Using `int2d.m` and `int1d.m`, we can compute the vector $\boldsymbol{u}$ as follows.

```
%% Assemble stiffness matrix
Coef = { -1, 1, 1 };
Test  = {'v1.val', 'v1.grad', 'v2.grad'};
Trial = {'u1.val', 'u2.grad', 'u1.grad'};
kk = int2d(Th,Coef,Test,Trial,Vh,quadOrder);

%% Assemble right hand side
Coef = pde.f; Test = 'v2.val';
ff = int2d(Th,Coef,Test,[],Vh,quadOrder);

%% Assemble Neumann boundary conditions
% Get 1D mesh for boundary integrals
Th.on = 1;

% Coef = @(p) pde.Du(p)*n;
Coef = interpEdgeMat(pde.Du,Th,quadOrder);
Test = 'v1.val';
ff = ff + int1d(Th,Coef,Test,[],Vh,quadOrder);

%% Apply Dirichlet boundary conditions
g_D = { [], pde.g_D };
on = 1;
U = apply2d(on,Th,kk,ff,Vh,g_D);
U = reshape(U,[],2);
w = U(:,1);   u = U(:,2);
```

In the above code, `Vh` can be chosen as {`'P1', 'P1'`}, {`'P2', 'P2'`} and {`'P3', 'P3'`}. The results are displayed in Fig. 6. We can find that the rate of convergence for u is optimal but for w is sub-optimal:

- For linear element, optimal order for w is also observed.
- For $\mathbb{P}_2$ element, the order for L^2 is 1.5 and for H^1 is 0.5.
- For $\mathbb{P}_3$ element, the order for L^2 is 2.5 and for H^1 is 1.5.

Note that our results are consistent with that given in iFEM. Obviously, for $\mathbb{P}_2$ and $\mathbb{P}_3$ elements, the rate of w has the behaviour of Δu, which is reasonable since $w = -\Delta u$.



(a) P1 (b) P2 (c) P3

Fig. 6: The convergence rates for the biharmonic equation.

3.4 Stokes problem

The Stokes problem is to find $(\boldsymbol{u}, p)$ such that

$$\begin{cases} -\nu\Delta\boldsymbol{u} - \nabla p = \boldsymbol{f} & \text{in } \Omega, \\ \mathrm{div}\boldsymbol{u} = 0 & \text{in } \Omega, \\ \boldsymbol{u} = \boldsymbol{g} & \text{on } \partial\Omega, \end{cases} \tag{3.5}$$

where, $\Omega \subset \mathbb{R}^d$ $(d = 2, 3)$, $\boldsymbol{u} = (u_1, \cdots, u_d)^T$ is the velocity of the fluid, p is the pressure and $\nu = 1/R_e > 0$ with R_e called Reynolds number.

- Models the steady state flow of viscous fluid.
- Conservation of mass: $\mathrm{div}\boldsymbol{u} = \nabla \cdot \boldsymbol{u} = 0$ ("incompressibility condition").

3.4.1 The variational problem

Multiplying the vector $\boldsymbol{v} \in H_0^1(\Omega)^d =: \boldsymbol{H}^1(\Omega)$ on both sides of the first equation of (3.5), and integrating over Ω gives

$$\int_\Omega -\nu\Delta\boldsymbol{u}\cdot\boldsymbol{v}\mathrm{d}x + \int_\Omega \nabla p\cdot\boldsymbol{v}\mathrm{d}x = \int_\Omega \boldsymbol{f}\cdot\boldsymbol{v}\mathrm{d}x.$$

Integration by parts yields

$$\int_\Omega \sum_{i=1}^d \nu\nabla u_i\cdot\nabla v_i \mathrm{d}x - \int_\Omega p\sum_{i=1}^d \frac{\partial v_i}{\partial x_i}\mathrm{d}x = \int_\Omega \boldsymbol{f}\cdot\boldsymbol{v}\mathrm{d}x, \tag{3.6}$$

where $\boldsymbol{v} = (v_1, \cdots, v_d)^T$.

Let

$$\nabla\boldsymbol{u} = \begin{bmatrix} \frac{\partial u_1}{\partial x_1} & \cdots & \frac{\partial u_1}{\partial x_d} \\ \vdots & \ddots & \vdots \\ \frac{\partial u_d}{\partial x_1} & \cdots & \frac{\partial u_d}{\partial x_d} \end{bmatrix} \quad \text{and} \quad \nabla\boldsymbol{v} = \begin{bmatrix} \frac{\partial v_1}{\partial x_1} & \cdots & \frac{\partial v_1}{\partial x_d} \\ \vdots & \ddots & \vdots \\ \frac{\partial v_d}{\partial x_1} & \cdots & \frac{\partial v_d}{\partial x_d} \end{bmatrix},$$

For two d-order matrices $A = (a_{ij})$ and $B = (b_{ij})$, define the inner product

$$(A, B) = A : B = \sum_{i=1}^d\sum_{j=1}^d a_{ij}b_{ij}.$$

Then the equation (3.6) can be written as

$$\int_\Omega \nu \nabla \boldsymbol{u} : \nabla \boldsymbol{v} \mathrm{d}x - \int_\Omega (\mathrm{div} \boldsymbol{v}) p \mathrm{d}x = \int_\Omega \boldsymbol{f} \cdot \boldsymbol{v} \mathrm{d}x.$$

Let $\boldsymbol{g} = 0$ and define $\boldsymbol{V} = \boldsymbol{H}_0^1(\Omega)$ and $P = L_0^2(\Omega)$. The mixed variational problem is: Find $(\boldsymbol{u}, p) \in \boldsymbol{V} \times Q$ such that

$$\begin{cases} a(\boldsymbol{u}, \boldsymbol{v}) + b(\boldsymbol{v}, p) & = (\boldsymbol{f}, \boldsymbol{v}), \quad \boldsymbol{v} \in \boldsymbol{V}, \\ b(\boldsymbol{u}, q) & = 0, \quad q \in P, \end{cases}$$

where

$$a(\boldsymbol{u}, \boldsymbol{v}) = (\nu \nabla \boldsymbol{u}, \nabla \boldsymbol{v}), \qquad b(\boldsymbol{v}, q) = (\mathrm{div} \boldsymbol{v}, q).$$

Let $\mathcal{T}_h$ be a shape regular triangulation of Ω. We consider the conforming finite element discretizations: $\boldsymbol{V}_h \subset \boldsymbol{V}$ and $P_h \subset P$. Typical pairs $(\boldsymbol{V}_h, P_h)$ of stable finite element spaces include: MINI element, Girault-Raviart element and $\mathbb{P}_k - \mathbb{P}_{k-1}$ elements. For the last one, a special example is the $\mathbb{P}_2 - \mathbb{P}_1$ element, also known as the Taylor-Hood element, which is the one under consideration.

The FEM is to find $(\boldsymbol{u}_h, p_h) \in \boldsymbol{V}_h \times P_h$ such that

$$\begin{cases} a(\boldsymbol{u}_h, \boldsymbol{v}) + b(\boldsymbol{v}, p_h) = F(\boldsymbol{v}), & \boldsymbol{v} \in \boldsymbol{V}_h, \\ b(\boldsymbol{u}_h, q) = 0, & q \in P_h. \end{cases} \tag{3.7}$$

The problem (3.7) can be solved either by discretizing it directly into a system of equations (a saddle point problem), or by adding the two equations, as done for the biharmonic equation. The later one is to find $(\boldsymbol{u}_h, p_h) \in \boldsymbol{V}_h \times L^2(\Omega)$ such that

$$a(\boldsymbol{u}_h, \boldsymbol{v}) + b(\boldsymbol{v}, p_h) + b(\boldsymbol{u}_h, q) - \varepsilon(p_h, q) = F(\boldsymbol{v}), \boldsymbol{v} \in \boldsymbol{V}_h, \quad \in L^2(\Omega), \tag{3.8}$$

where ε is a small parameter to ensure stability or $p_h \in L_0^2$ (You can find the fact by taking $v_1 = v_2 = 0$ and $q = 1$).

In the following, we only consider the two-dimensional case.

3.4.2 The programming in scalar form

We remark that the problem (3.7) (when including the stabilization term) and (3.8) are equivalent. To express it more clearly, we write the first equation of the problem (3.5) specifically by components, with

$$\begin{cases} -\nu \Delta u_1 + \dfrac{\partial p}{\partial x} = f_1, \\ -\nu \Delta u_2 + \dfrac{\partial p}{\partial y} = f_2. \end{cases}$$

The variational form is actually obtained by multiplying v_1 and $v_2 \in H_0^1(\Omega)$ in the first and second rows, respectively, and then adding them up after the integration by parts.

The integration by parts gives

$$\begin{cases} \nu \displaystyle\int_\Omega \nabla u_1 \cdot \nabla v_1 \mathrm{d}x - \int_\Omega p \frac{\partial v_1}{\partial x} \mathrm{d}x = \int_\Omega f_1 v_1 \mathrm{d}x, \\ \nu \displaystyle\int_\Omega \nabla u_2 \cdot \nabla v_2 \mathrm{d}x - \int_\Omega p \frac{\partial v_2}{\partial y} \mathrm{d}x = \int_\Omega f_2 v_2 \mathrm{d}x. \end{cases}$$

Accordingly, the second equation of the problem (3.5) is

$$0 = -\int_\Omega (\mathrm{div}\boldsymbol{u})q\mathrm{d}x = -\int_\Omega \frac{\partial u_1}{\partial x}q\mathrm{d}x - \int_\Omega \frac{\partial u_2}{\partial y}q\mathrm{d}x.$$

The above equations are abbreviated together as

$$\begin{cases} a_1(v_1,u_1) + b_1(v_1,p) = F_1(v_1), \\ a_2(v_2,u_2) + b_2(v_2,p) = F_2(v_2), \\ b_1(q,u_1) + b_2(q,u_2) = 0. \end{cases}$$

The specific definitions of the notations are omitted here for brevity.

To get a stable solution, we add a perturbation term as

$$\begin{aligned} & a_1(v_1,u_1) + a_2(v_2,u_2) \\ & +b_1(v_1,p) + b_2(v_2,p) \\ & +b_1(q,u_1) + b_2(q,u_2) \\ & -\varepsilon(q,p) \\ & = F_1(v_1) + F_2(v_2). \end{aligned} \tag{3.9}$$

When expressing the functions under the basis functions, we have the following matrix expression

$$\begin{aligned} & \boldsymbol{v}_1^T A_1 \boldsymbol{u}_1 + \boldsymbol{v}_2^T A_2 \boldsymbol{u}_2 \\ & +\boldsymbol{v}_1^T B_1 \boldsymbol{p} + \boldsymbol{v}_2^T B_2 \boldsymbol{p} \\ & +\boldsymbol{q}^T C_1 \boldsymbol{u}_1 + \boldsymbol{q}^T C_2 \boldsymbol{u}_2 \\ & +\boldsymbol{q}^T D_\varepsilon \boldsymbol{p} \\ & = \boldsymbol{v}_1^T \boldsymbol{F}_1 + \boldsymbol{v}_2^T \boldsymbol{F}_2, \end{aligned}$$

where $C_1 = B_1^T$ and $C_2 = B_2^T$. The above equation can be written as

$$[\boldsymbol{v}_1^T, \boldsymbol{v}_2^T, \boldsymbol{q}^T] \begin{bmatrix} A_1 & O & B_1 \\ O & A_2 & B_2 \\ C_1 & C_2 & D_\varepsilon \end{bmatrix} \begin{bmatrix} \boldsymbol{u}_1 \\ \boldsymbol{u}_2 \\ \boldsymbol{p} \end{bmatrix} = [\boldsymbol{v}_1^T, \boldsymbol{v}_2^T, \boldsymbol{q}^T] \begin{bmatrix} \boldsymbol{F}_1 \\ \boldsymbol{F}_2 \\ \boldsymbol{0} \end{bmatrix},$$

or

$$\begin{bmatrix} A_1 & O & B_1 \\ O & A_2 & B_2 \\ C_1 & C_2 & D_\varepsilon \end{bmatrix} \begin{bmatrix} \boldsymbol{u}_1 \\ \boldsymbol{u}_2 \\ \boldsymbol{p} \end{bmatrix} = \begin{bmatrix} \boldsymbol{F}_1 \\ \boldsymbol{F}_2 \\ \boldsymbol{0} \end{bmatrix}.$$

This gives the block expression.

In the following we set $\nu = 1$. The finite element spaces are labelled as

```
Vh = {'P2','P2','P1'}; % [u1,u2,p]
quadOrder = 5;
```

The matrices A_1 and A_2 are the same, given by

```
% A1,A2
Coef = 1;  Test = 'v1.grad';  Trial = 'u1.grad';
A1 = assem2d(Th,Coef,Test,Trial,Vh(1),quadOrder);
A2 = A1;
```

The computation of B_1, B_2, C_1 and C_2 reads

```
% B1,B2
Coef = -1;  Test = 'v1.dx'; Trial = 'p.val';
B1 = assem2d(Th,Coef,Test,Trial,Vh([1,3]),quadOrder);
Coef = -1;  Test = 'v2.dy'; Trial = 'p.val';
B2 = assem2d(Th,Coef,Test,Trial, Vh([1,3]), quadOrder);
% C1,C2
C1 = B1'; C2 = B2';
```

Note that the test function and the trial function are in the different spaces. And for the function `assem2d.m`, there is no need to transform `p` to `u3` since the function only identifies `.dx`, `.dy`, `.val` and so on.

Similarly, we have the matrix D and the final stiffness matrix:

```
% D
eps = 1e-10;
Coef = -eps;  Test = 'p.val'; Trial = 'q.val';
D = assem2d(Th,Coef,Test,Trial,Vh(3),quadOrder);

% stiffness matrix
NNdof = size(A1,1);
O = zeros(NNdof,NNdof);
kk = [ A1,   O,   B1;
       O,    A2,  B2;
       C1,   C2,  D  ];
kk = sparse(kk);
```

The right-hand side is given as

```
%% Assemble right hand side
ff = zeros(size(kk,1),1);
% F1
trf = eye(2);
Coef = @(pz) pde.f(pz)*trf(:, 1);  Test = 'v1.val';
F1 = assem2d(Th,Coef,Test,[],Vh(1),quadOrder);

% F2
trf = eye(2);
Coef = @(pz) pde.f(pz)*trf(:, 2);  Test = 'v2.val';
F2 = assem2d(Th,Coef,Test,[],Vh(2),quadOrder);

ff(1:2*NNdof) = [F1;F2];
```

We impose the Dirichlet boundary conditions as follows.

```
%% Apply Dirichlet boundary conditions
tru = eye(2);
g_D1 = @(pz) pde.g_D(pz)*tru(:, 1);
g_D2 = @(pz) pde.g_D(pz)*tru(:, 2);
g_D = {g_D1, g_D2, []};
on = 1;
U = apply2d(on,Th,kk,ff,Vh,g_D);
uh = [U(1:NNdof), U(NNdof+1:2*NNdof)]; % uh = [u1h, u2h]
ph = U(2*NNdof+1:end);
```

Note that `g_D{3} = []` since no constraints are imposed on p.

Example 3.1. *Let* $\Omega = (0,1)^2$. *We choose the load term* $\boldsymbol{f}$ *in such a way that the analytical* *solution is*

$$\boldsymbol{u}(x,y) = \begin{bmatrix} -2^8(x^2-2x^3+x^4)(2y-6y^2+4y^3) \\ 2^8(2x-6x^2+4x^3)(y^2-2y^3+y^4) \end{bmatrix}$$

and $p(x,y) = -2^8(2-12x+12x^2)(y^2-2y^3+y^4)$.

The results are displayed in Fig. 7 and Tab. 2, from which we observe the optimal rates of convergence both for u and p.

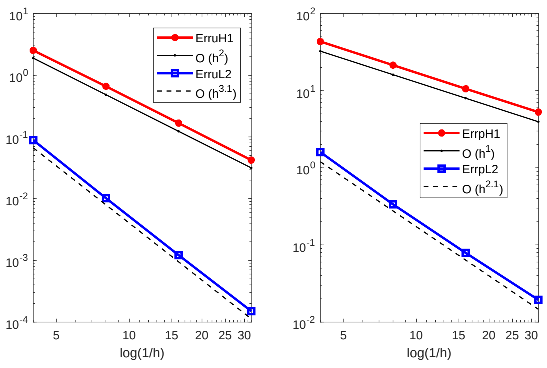


Fig. 7: Convergence rates of the Taylor-Hood element for the Stokes problem

Tab. 2: The discrete errors for the Stokes problem

♯Dof	h	$\|u-u_h\|$	$\|u-u_h\|_1$	$\|p-p_h\|$
32	2.500e-01	8.88464e-02	2.52940e+00	1.59802e+00
128	1.250e-01	1.01868e-02	6.62003e-01	3.36224e-01
512	6.250e-02	1.21537e-03	1.67792e-01	7.88512e-02
2048	3.125e-02	1.50235e-04	4.21077e-02	1.94079e-02
8192	1.562e-02	1.87368e-05	1.05374e-02	4.83415e-03

Remark 3.2. In the lowest order $k = 2$ (for $\boldsymbol{u}$), one can only expect the second-order convergence in L^2 norm in general cases since the Stokes problem corresponds to a fourth-order problem.

3.4.3 The programming in vector form

Noting the equation (3.9), we have the bilinear form:

```
Vh = {'P2','P2','P1'}; quadOrder = 5;

%% Assemble stiffness matrix
vstr = {'v1','v2','q'}; ustr = {'u1','u2','p'};
% [v1,v2,q] = [v1,v2,v3], [u1,u2,p] = [u1,u2,u3]
%    a1(v1,u1) + a2(v2,u2)
% + b1(v1,p)   + b2(v2,p)
% + b1(q,u1)   + b2(q,u2) - eps*(q,p)
eps = 1e-10;
Coef  = { 1, 1,  -1,-1,  -1,-1,  -eps};
Test  = {'v1.grad', 'v2.grad', 'v1.dx',  'v2.dy',  'q.val', 'q.val', 'q.val'};
Trial = {'u1.grad', 'u2.grad', 'p.val',  'p.val', 'u1.dx',  'u2.dy',  'p.val'};
```

```
[Test,Trial] = getStdvarForm(vstr, Test,  ustr, Trial); % [u1,u2,p] --> [u1,u2,u3]
[kk,info] = int2d(Th,Coef,Test,Trial,Vh,quadOrder);
```

Note that the symbols `p` and `q` correspond to `u3` and `v3`, which is realized by using the subroutine `getStdvarForm.m`.

The computation of the right-hand side reads

```
%% Assemble right hand side
trf = eye(2);
Coef1 = @(pz) pde.f(pz)*trf(:, 1);  Coef2 = @(pz) pde.f(pz)*trf(:, 2);
Coef = {Coef1, Coef2};
Test = {'v1.val', 'v2.val'};
ff = int2d(Th,Coef,Test,[],Vh,quadOrder);
```

In this case, one can not use `Coef = pde.f` and `Test = 'v.val'` instead since $\boldsymbol{v} = [v_1, v_2, v_3]^T$ has three components.

We impose the Dirichlet boundary conditions as follows.

```
%% Apply Dirichlet boundary conditions
tru = eye(2);
g_D1 = @(pz) pde.g_D(pz)*tru(:, 1);
g_D2 = @(pz) pde.g_D(pz)*tru(:, 2);
g_D = {g_D1, g_D2, []};
on = 1;
U = apply2d(on,Th,kk,ff,Vh,g_D);
NNdofu = info.NNdofu;
id1 =  NNdofu(1);  id2 = NNdofu(1)+ NNdofu(2);
uh = [ U(1:id1), U(id1+1:id2) ]; % uh = [u1h, u2h]
ph = U(id2+1:end);
```

The omitted numerical results are the same.

3.5 Heat equation

As an example for time-dependent problems, we consider the heat equation:

$$\begin{cases} u_t - \Delta u = f & \text{in } \ \Omega, \\ u(x,y,0) = u_0(x,y) & \text{in } \ \Omega, \\ u = g_D & \text{on } \ \Gamma_D, \\ \partial_n u = g_N & \text{on } \ \Gamma_N. \end{cases}$$

After applying the backward Euler discretization in time, we shall seek $u^n(x,y)$ satisfying for all $v \in H_0^1(\Omega)$:

$$\int_\Omega \Big(\frac{u^n - u^{n-1}}{\Delta t} v + \nabla u^n \cdot \nabla v \Big) \mathrm{d}\sigma = \int_\Omega f^n v \mathrm{d}\sigma + \int_{\Gamma_N} g_N^n v \mathrm{d}s.$$

We fist generate a mesh and compute the mesh information.

```
%% Mesh
% generate mesh
Nx = 10;
[node,elem] = squaremesh([0 1 0 1],1/Nx);
% mesh info
bdStr = 'x==0'; % Neumann
```

```
Th = FeMesh2d(node,elem,bdStr);
% time
Nt = Nx^2;
t = linspace(0,1,Nt+1)';  dt = t(2)-t(1);
```

The PDE data is given by

```
%% PDE data
pde = heatData();
```

The exact solution is chosen as $u = \sin(\pi x)\sin(y)\mathrm{e}^{-t}$.

For fixed Δt, the bilinear form gives the same stiffness matrix in each iteration.

```
%% Bilinear form
Vh = 'P1';  quadOrder = 7;
Coef  = {1/dt, 1};
Test  = {'v.val', 'v.grad'};
Trial = {'u.val', 'u.grad'};
kk = assem2d(Th,Coef,Test,Trial,Vh,quadOrder);
```

The linear form is dependent of time, so we compute it in the iteration:

```
%% Backward Euler
u0 = @(p) pde.uexact(p,t(1));
uh0 = interp2d(u0,Th,Vh); % dof vector
uf = zeros(Nt+1,2);  % record solutions at p-th point
p = 2*Nx;
uf(1,:) = [uh0(p),uh0(p)];
for n = 1:Nt
    % Linear form
    fun = @(p) pde.f(p, t(n+1));
    Coef = fun;  Test = 'v.val';
    ff = assem2d(Th,Coef,Test,[],Vh,quadOrder);
    Coef = uh0/dt;
    ff = ff + assem2d(Th,Coef,Test,[],Vh,quadOrder);

    % Neumann boundary condition
    if ¬isempty(bdStr)
        Th.on = 1;
        fun = @(p) pde.Du(p,t(n+1));
        Coef = interpEdgeMat(fun,Th,quadOrder);
        ff = ff + assem1d(Th,Coef,Test,[],Vh,quadOrder);
    end

    % Dirichlet boundary condition
    ue = @(p) pde.uexact(p,t(n+1));
    on = 2 - 1*isempty(bdStr); % on = 1 for the whole boundary if bdStr = []
    uh = apply2d(on,Th,kk,ff,Vh,ue);

    % Record
    uhe = interp2d(ue,Th,Vh);
    uf(n+1,:) = [uhe(p), uh(p)];

    % Update
    uh0 = uh;
end
```

In the above code, we record solutions at p-th point, where $p = 2N_x$. The results are shown in Fig. 8 and Fig. 9 (see `test_varHeat.m`).

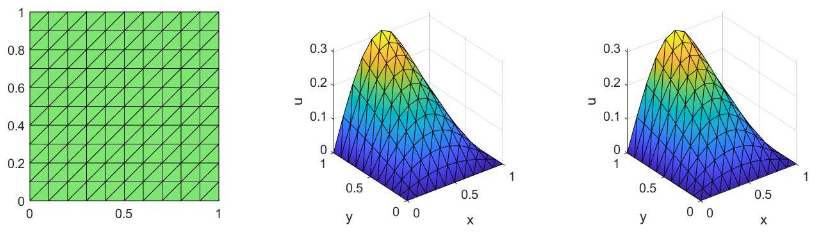


Fig. 8: Exact and numerical solutions for the heat equation

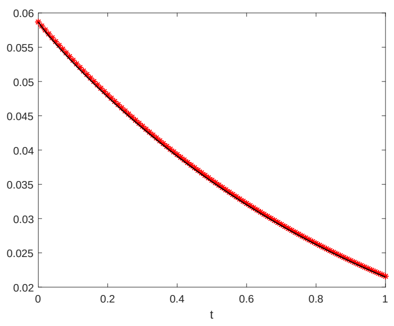
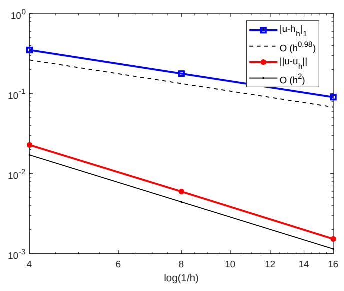
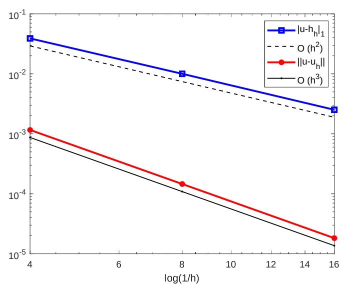


Fig. 9: Exact and numerical solutions at a fixed point for the heat equation

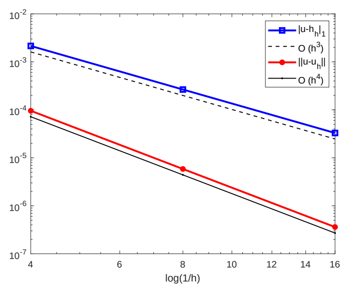
It is well-known that the error behaves as $\mathcal{O}(\Delta t + h^{k+1})$ for the $\mathbb{P}_k$-Lagrange element. To test the convergence order w.r.t h, one can set $\Delta t = \mathcal{O}(h^{k+1})$. The numerical results are consistent with the theoretical prediction as shown in Fig. 10 (see `main_varHeat.m`).



(a) P1



(b) P2



(c) P3

Fig. 10: Convergence rates for the heat equation.

3.6 Navier-Stokes equation

The steady-state flow model of an incompressible Newton fluid is described by the following Navier-Stokes equation

$$\begin{cases} -\nu\Delta \boldsymbol{u} + (\boldsymbol{u}\cdot\nabla)\boldsymbol{u} + \nabla p = \boldsymbol{f} \quad \text{in } \Omega, \\ -\mathrm{div}\boldsymbol{u} = 0 \quad \text{in } \Omega, \\ \boldsymbol{u} = \boldsymbol{g} \quad \text{on } \partial\Omega, \end{cases} \tag{3.10}$$

where $\Omega \subset \mathbb{R}^d$ $(d = 2, 3)$, $\nu = 1/R_e > 0$, $\boldsymbol{f}$ is the body force, $\boldsymbol{u}$ is the velocity, and p is the pressure.

For two-dimensional case,

$$(\boldsymbol{u}\cdot\nabla)\boldsymbol{v} = \begin{bmatrix} u_1\dfrac{\partial v_1}{\partial x} + u_2\dfrac{\partial v_1}{\partial y} \\ u_1\dfrac{\partial v_2}{\partial x} + u_2\dfrac{\partial v_2}{\partial y} \end{bmatrix}.$$

As done for the Stokes problem, define the following spaces

$$\boldsymbol{V} := H_0^1(\Omega)^d, \qquad P := L_0^2(\Omega).$$

The variational problem is to find $(\boldsymbol{u}, p) \in \boldsymbol{V}\times P$ such that

$$\begin{cases} a(\boldsymbol{u},\boldsymbol{v}) + N(\boldsymbol{u};\boldsymbol{u},\boldsymbol{v}) + b(\boldsymbol{v},p) = F(\boldsymbol{v}), \quad \boldsymbol{v}\in\boldsymbol{V}, \\ b(\boldsymbol{u},q) = 0, \quad q\in P, \end{cases} \tag{3.11}$$

where

$$a(\boldsymbol{u},\boldsymbol{v}) = \nu\int_\Omega \nabla\boldsymbol{u} : \nabla\boldsymbol{v}\mathrm{d}x, \qquad b(\boldsymbol{v},q) = -\int_\Omega (\mathrm{div}\boldsymbol{v})p\mathrm{d}x,$$

$$N(\boldsymbol{u};\boldsymbol{v},\boldsymbol{w}) = \int_\Omega (\boldsymbol{u}\cdot\nabla)\boldsymbol{v}\cdot\boldsymbol{w}\mathrm{d}x, \qquad F(\boldsymbol{v}) = \int_\Omega \boldsymbol{f}\cdot\boldsymbol{v}\mathrm{d}x.$$

The mixed finite element method is: Find $(\boldsymbol{u}_h, p_h)\in\boldsymbol{V}_h\times P_h$ such that

$$\begin{cases} a(\boldsymbol{u}_h,\boldsymbol{v}) + N(\boldsymbol{u}_h;\boldsymbol{u}_h,\boldsymbol{v}) + b(\boldsymbol{v},p_h) = F(\boldsymbol{v}), \quad \boldsymbol{v}\in\boldsymbol{V}_h, \\ b(\boldsymbol{u}_h,q) = 0, \quad q\in P_h. \end{cases} \tag{3.12}$$

The two equations can be added together. So we need to find $(\boldsymbol{u}_h, p_h)\in\boldsymbol{V}_h\times P_h$ such that

$$a(\boldsymbol{u}_h,\boldsymbol{v}) + N(\boldsymbol{u}_h;\boldsymbol{u}_h,\boldsymbol{v}) + b(\boldsymbol{v},p_h) + b(\boldsymbol{u}_h,q) = F(\boldsymbol{v}), \quad \boldsymbol{v}\in\boldsymbol{V}_h,\ \ q\in P_h.$$

Since $N(\boldsymbol{u}_h;\boldsymbol{u}_h,\boldsymbol{v})$ is nonlinear, we solve it by using the Newton iteration method: Given the initial data $(\boldsymbol{u}_h^0, p_h^0)$, find $(\boldsymbol{u}_h^{n+1}, p_h^{n+1})$ satisfying

$$\begin{aligned} &a(\boldsymbol{u}_h^{n+1},\boldsymbol{v}) + N(\boldsymbol{u}_h^{n+1};\boldsymbol{u}_h^n,\boldsymbol{v}) + N(\boldsymbol{u}_h^n;\boldsymbol{u}_h^{n+1},\boldsymbol{v}) + b(\boldsymbol{v},p_h^{n+1}) + b(\boldsymbol{u}_h^{n+1},q) \\ &= F(\boldsymbol{v}) + N(\boldsymbol{u}_h^n;\boldsymbol{u}_h^n,\boldsymbol{v}), \quad \boldsymbol{v}\in\boldsymbol{V}_h,\ \ q\in P_h. \end{aligned} \tag{3.13}$$

Note that we still need to include the stabilization term to ensure $p\in L_0^2(\Omega)$.

In the bilinear forms, the stiffness matrices given by the first, the third and the last terms are fixed in the iteration:

```matlab
nu = pde.nu;  eps = 1e-10;
vstr = {'v1','v2','q'}; ustr = {'u1','u2','p'};

%% Fixed bilinear forms
% a(uh^{n+1}, v) +  b(v, ph^{n+1}) + b(uh^{n+1},q) - eps*p*q
Coef = { nu, nu,  -1, -1,  -1, -1, -eps};
Test  = {'v1.grad','v2.grad',  'v1.dx','v2.dy',  'q.val', 'q.val', 'q.val'};
Trial = {'u1.grad','u2.grad',  'p.val','p.val',  'u1.dx', 'u2.dy', 'p.val'};

[Test,Trial] = getStdvarForm(vstr, Test, ustr, Trial);
kk = int2d(Th,Coef,Test,Trial,Vh,quadOrder);
```

Here we use `getStdvarForm.m` to transform the notation to the standard one.

Similarly, we have the fixed load vector:

```
%% Fixed linear forms
f1 = @(p) pde.f(p)*[1;0];
f2 = @(p) pde.f(p)*[0;1];
Coef = {f1, f2};
Test = {'v1.val', 'v2.val'};
ff = int2d(Th,Coef,Test,[],Vh,quadOrder);
```

The iteration is described as follows.

- We first give the initial data

```
% initial data
u1 = @(p) 0*p(:,1);
u2 = @(p) 0*p(:,1);
p = @(p) 0*p(:,1);
```

They must be converted into the dof vectors in view of the iteration.

```
uh1 = interp2d(u1,Th,Vh{1});
uh2 = interp2d(u2,Th,Vh{2});
ph = interp2d(p,Th,Vh{3});
```

- For numerical integration, it is preferable to provide the coefficient matrices for these coefficient functions.

```
u1c = interp2dMat(uh1,'u1.val',Th,Vh{1},quadOrder);
u2c = interp2dMat(uh2,'u2.val',Th,Vh{2},quadOrder);
%pc = interp2dMat(ph,'p.val',Th,Vh{3},quadOrder);

u1xc = interp2dMat(uh1,'u1.dx',Th,Vh{1},quadOrder);
u1yc = interp2dMat(uh1,'u1.dy',Th,Vh{1},quadOrder);
u2xc = interp2dMat(uh2,'u2.dx',Th,Vh{2},quadOrder);
u2yc = interp2dMat(uh2,'u2.dy',Th,Vh{2},quadOrder);
```

Here, `interp2dMat.m` provides a way to get the coefficient matrices only from the dof vectors.

- The triple `(Coef,Test,Trial)` for $N(\boldsymbol{u}_h^{n+1};\boldsymbol{u}_h^n,\boldsymbol{v}) + N(\boldsymbol{u}_h^n;\boldsymbol{u}_h^{n+1},\boldsymbol{v})$ is then given by

```
% bilinear form: N(uh^{n+1}; uh^n, v) + N(uh^n; uh^{n+1}, v)
Coef = {u1xc, u1yc, u2xc, u2yc, ...
    u1c, u2c, u1c, u2c};
Test = {'v1.val','v1.val','v2.val','v2.val', ...
    'v1.val','v1.val','v2.val','v2.val'};
Trial = {'u1.val','u2.val','u1.val','u2.val', ...
    'v1.dx','v1.dy','v2.dx','v2.dy'};
[kkN,info] = int2d(Th,Coef,Test,Trial,Vh,quadOrder);
kkn = kk + kkN;
```

Note that there is no non-standard notation in this part.

- The computation of the remaining load vector reads

```
% linear form: N(uh^n; uh^n, v)
Coef = {u1c.*u1xc+u2c.*u1yc,  u1c.*u2xc+u2c.*u2yc};
Test = {'v1.val', 'v2.val'};
ffn = ff + int2d(Th,Coef,Test,[],Vh,quadOrder);
```

- The Dirichlet boundary conditions are imposed as

```
% Dirichlet boundary conditions
g_D1 = @(p) pde.g_D(p)*[1;0];
g_D2 = @(p) pde.g_D(p)*[0;1];
g_D = {g_D1, g_D2, []};
on = 1;
U = apply2d(on,Th,kkn,ffn,Vh,g_D);

% Update
NNdofu = info.NNdofu;
id1 =  NNdofu(1);  id2 = NNdofu(1)+ NNdofu(2);
uh1 = U(1:id1);
uh2 = U(id1+1:id2);
ph = U(id2+1:end);
```

The iteration stops when the l^2 norm of the errors between the true dof vector and the numerical dof vector is less than 10^{-10} or the number of iterations is greater than 20.

Example 3.2. *Let* $\Omega = (0,1)^2$*. The exact velocity is* $\boldsymbol{u} = (u_1, u_2)^T$*, where*

$$u_1(x,y) = \phi(x)\phi'(y), \qquad u_2(x,y) = -\phi'(x)\phi(y),$$

with $\phi(t) = t^2(t-1)^2$*, which satisfies the homogeneous Dirichlet boundary conditions. The exact pressure is chosen as*

$$p(x,y) = (2x-1)(2y-1).$$

A direct computation gives

$$\begin{aligned} f_1 &= 4y - 4\nu(2y-1)(3x^4 - 6x^3 + 6x^2y^2 - 6x^2y + 3x^2 \\ &\quad - 6xy^2 + 6xy + y^2 - y) + 8x^3y^2(x-1)^2(2x^2-3x+1)(2y^2-3y+1)^2 \\ &\quad - 4x^3y^2(x-1)^2(y-1)^2(2x^2-3x+1)(6y^2-6y+1) - 2, \\ f_2 &= 4x + 4\nu(2x-1)(6x^2y^2 - 6x^2y + x^2 - 6xy^2 \\ &\quad + 6xy - x + 3y^4 - 6y^3 + 3y^2) + 8x^2y^3(y-1)^2(2x^2-3x+1)^2(2y^2-3y+1) \\ &\quad - 4x^2y^3(x-1)^2(y-1)^2(6x^2-6x+1)(2y^2-3y+1) - 2. \end{aligned}$$

The test script is `main_varNavierStokes.m`. The number of steps and the error of dof vectors are printed in the following.

```
Number of iterations = 4,    erru = 2.2143e-17
Number of iterations = 4,    erru = 5.4957e-17
Number of iterations = 4,    erru = 1.5019e-16
Number of iterations = 4,    erru = 6.8219e-16
Number of iterations = 4,    erru = 2.1303e-15
```

The results are displayed in Fig. 11, from which we observe the optimal rates of convergence both for u and p. We can also plot the level lines of the pressure, as given in Fig. 12. Note that the contour figure is obtained by interpolating the finite element function to a two-dimensional cartesian grid (within the mesh). The interpolated values can be created by using the Matlab built-in function `pdeInterpolant.m` in the pdetool. However, the built-in function seems not efficient. For this reason, we provide a new realization of the interpolant, named `varInterpolant2d.m`. With this function, we give a subroutine `varcontourf.m` to draw the level lines.

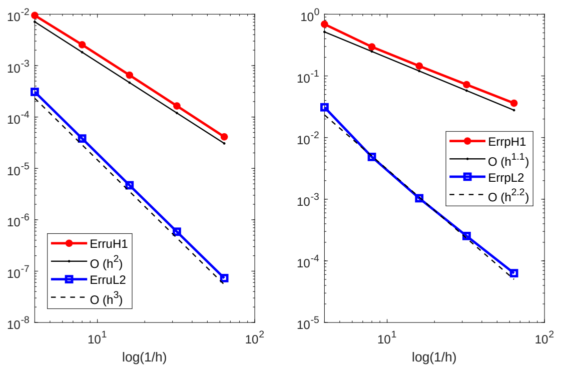


Fig. 11: Convergence rates of the Taylor-Hood element for the Navier-Stokes problem

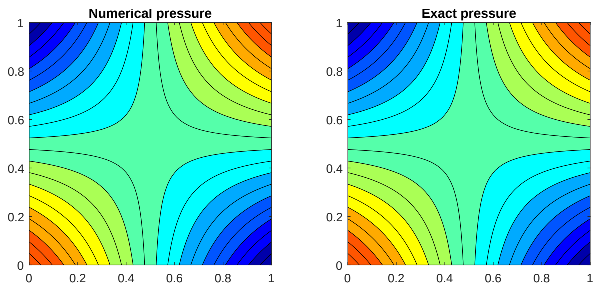


Fig. 12: Contour figure for the pressure

3.7 Variational inequalities

We now focus on the the finite element method to solve a simplified friction problem, which is a typical elliptic variational inequality of the second kind.

Let $\Omega \subset \mathbb{R}^2$ be a bounded domain with a Lipschitz boundary $\Gamma = \partial\Omega$ that is divided into two parts Γ_C and Γ_D. The problem is

$$\begin{cases} -\Delta u + \alpha u = f & \text{in } \Omega, \\ \partial_{\boldsymbol{n}} u \le g, \quad u\partial_{\boldsymbol{n}} u + g|u| = 0, & \text{on } \Gamma_C, \\ u = 0 \quad \text{on} & \Gamma_D, \end{cases} \tag{3.14}$$

where $\alpha > 0$ is a constant, $f \in L^2(\Omega)$, $g \in L^2(\Gamma_C)$, Γ_C is the frictional boundary part and Γ_D is the Dirichlet boundary part.

Define

$$V = \{v \in H^1(\Omega) : v|_{\Gamma_D} = 0\}.$$

The variational inequality is: Find $u \in V$ such that

$$a(u, v-u) + j(v) - j(u) \ge \ell(v-u), \quad v \in V, \tag{3.15}$$

where

$$a(u,v) = \int_\Omega (\nabla u \cdot \nabla v + \alpha uv)\mathrm{d}x, \quad \ell(v) = \int_\Omega fv\mathrm{d}x, \quad j(v) = \int_{\Gamma_C} g|v|\mathrm{d}s.$$

By introducing a Lagrangian multiplier

$$\lambda \in \Lambda = \{\lambda \in L^\infty(\Gamma_C) : |\lambda| \le 1 \quad \text{a.e. on } \Gamma_C\},$$

the inequality problem (3.15) can be rewritten as

$$\begin{cases} a(u,v) + \displaystyle\int_{\Gamma_C} g\lambda v \mathrm{d}s = \ell(v), \quad v \in V, \\ \lambda u = |u| \quad \text{a.e. on } \Gamma_C. \end{cases}$$

For this reason, the discrete problem can be recast as

$$\begin{cases} a_h(u_h, v_h) + \displaystyle\int_{\Gamma_C} g\lambda_h v_h \mathrm{d}s = \ell_h(v_h), \quad v_h \in V_h, \\ \lambda_h u_h = |u_h| \quad \text{a.e. on } \Gamma_C, \end{cases}$$

where $\lambda_h \in L^\infty(\Gamma_C)$ and $|\lambda_h| \le 1$. Then the Uzawa algorithm for solving the above problem is: given any $\lambda_h^{(0)} \in \Lambda$, for $n \ge 1$, find $u_h^{(n)}$ and $\lambda_h^{(n)}$ by solving

$$a_h(u_h^{(n)}, v_h) = \ell_h(v_h) - \int_{\Gamma_C} g\lambda_h^{(n-1)} v_h \mathrm{d}s \tag{3.16}$$

and

$$\lambda_h^{(n)} = P_\Lambda(\lambda_h^{(n-1)} + \rho g u_h^{(n)}),$$

where $P_\Lambda(\mu) = \sup\{-1, \inf\{1, \mu\}\}$ and ρ is a constant parameter.

Example 3.3. *Let $\Omega = (0,1)^2$ and suppose that the frictional boundary condition is imposed on $y = 0$. The function g can be simply chosen as $\sup_{\Gamma_C} |\partial_{\boldsymbol{n}} u|$. The right-hand function f is chosen such that the exact solution is $u = (\sin(x) - x\sin(1))\sin(2\pi y)$.*

The Uzawa iteration stops when $\|\boldsymbol{\chi}(u_h^{n+1} - u_h^n)\|_{l^2} \le \mathrm{tol}$ or $n \ge \mathrm{maxIt}$, where $\boldsymbol{\chi}(u_h)$ is the dof vector. It is evident that the problem (3.16) is exactly the VEM discretization for the reaction-diffusion problems, with the Neumann boundary data replaced by $g\lambda_h^{(n-1)}$. In addition, we only need to assemble the integral on Γ_C in each iteration. Because of this, we will not give the implementation details. We set $\mathrm{tol} = 10^{-8}$, $\mathrm{maxIt} = 500$ and $\rho = 10$. The results are shown in Figs. 13. The test script is `test_Poisson_VI_Uzawa.m`.

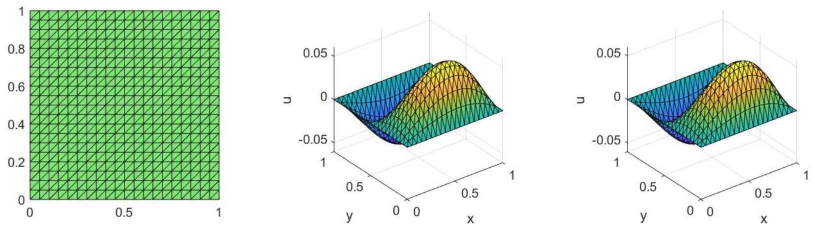


Fig. 13: Numerical and exact solutions for the simplified friction problem ($\alpha = 10^4$)

3.8 Eigenvalue problems

Consider the eigenvalue problem for the Poisson equation in 2-D: Find $\lambda \in \mathbb{R}$ such that

$$\begin{cases} -\Delta u = \lambda u & \text{in } \Omega, \\ u = 0 & \text{on } \partial\Omega, \end{cases}$$

where $u \not\equiv 0$. The variational problem reads as: Find $(\lambda, u) \in \mathbb{R} \times V$, $\|u\| = 1$, such that

$$a(u,v) = \lambda b(u,v), \qquad v \in V = H_0^1(\Omega),$$

where $a(u,v) = (\nabla u, \nabla v)$ and $b(u,v) = (u,v)$.

Let us consider the numerical solution of the conforming Lagrange elements. The matrix form can be written as

$$A\boldsymbol{u} = \lambda \boldsymbol{u}.$$

If the boundary conditions are imposed, one easily gets

$$A(\text{freedof}, \text{freedof}) \cdot \boldsymbol{u}(\text{freedof}) = \lambda \boldsymbol{u}(\text{freedof}),$$

where `freedof` represents the index of the variables.

Example 3.4. *Let $\omega = (0,1)^2$. The eigenvalues are known and given by*

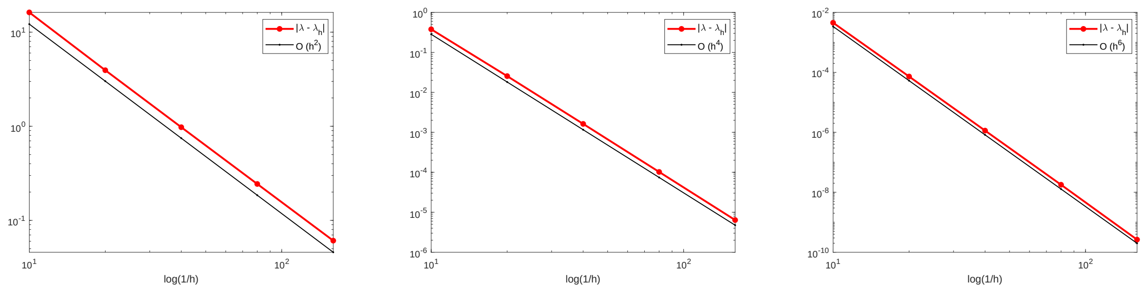
$$\lambda = \pi^2(n^2 + m^2), \qquad n, m = 1, 2, \cdots$$

The code is simple, given as follows.

```
%% Parameters
maxIt = 5;
N = zeros(maxIt,1);
h = zeros(maxIt,1);
Err = zeros(maxIt,1);

%% Generate an intitial mesh
[node,elem] = squaremesh([0 1 0 1],1/5);
bdStr = [];

%% Exact eigenvalues
nm = [1 1;
      1 2; 2 1;
      1 3; 2 2; 3 1;
      1 4; 2 3; 3 2; 4 1];
lamExact = pi^2*(nm(:,1).^2 + nm(:,2).^2);
lamExact = sort(lamExact);  % in ascending order
nlam = length(lamExact);

%% Finite element method
Vh = 'P1'; quadOrder = 7;
for k = 1:maxIt
    % refine mesh
    [node,elem] = uniformrefine(node,elem);
    % get the mesh information
    Th = FeMesh2d(node,elem,bdStr);
    % A
    Coef  = 1;  Test  = 'v.grad';  Trial = 'u.grad';
    A = assem2d(Th,Coef,Test,Trial,Vh,quadOrder);
    NNdof = size(A,1);
    % B
    Coef  = 1;  Test  = 'v.val';  Trial = 'u.val';
    B = assem2d(Th,Coef,Test,Trial,Vh,quadOrder);
    % apply Dirichlet boundary conditions
    gDLogic = 1; % vector case: [u1,u2,u3] = [1,0,1]
    on = 1;
    [A,freedof] = apply2dMat(A,on,Th,Vh,gDLogic);
    B = apply2dMat(B,on,Th,Vh,gDLogic);

    % find eigenvalues
    Uh = zeros(NNdof,nlam);
    [Uh(freedof,:),D] = eigs(A,B,nlam,'smallestabs');
    lam = diag(D);

    % record
    N(k) = size(elem,1);
    h(k) = 1/(sqrt(size(node,1))-1);

    % compute errors
    Err(k) = norm(lam-lamExact)/sqrt(nlam);
end

%% Plot convergence rates and display results
figure,
showrateh(h,Err,'|\lambda - \lambda_h|');

fprintf('\n');
colname = {'Exact eigenvalues',  'Numerical eigenvalues'};
disptable(colname, lamExact,'%0.5f', lam,'%0.5f');
```

- For convenience, we introduce a new function `apply2dMat.m` to get the matrix with boundary condition imposed.
- The eigenvalues are computed by the built-in function `eigs.m`. Please see the Matlab document for details (Just type `open eigs` on the Command Window).

It is known that the error $|\lambda - \lambda_h|$ behaves as $\mathcal{O}(h^k)$ for the $\mathbb{P}_k$-Lagrange element. The numerical results are consistent with the theoretical prediction as shown in Fig. 14.



(a) P1 (b) P2 (c) P3

Fig. 14: Convergence rates for the eigenvalue problem of the Poisson equation.

The exact and numerical eigenvalues for the $\mathbb{P}_1$ element are displayed below.

```
Exact eigenvalues    Numerical eigenvalues
-----------------    ---------------------

     19.73921             19.74111
     49.34802             49.35620
     49.34802             49.36077
     78.95684             78.98727
     98.69604             98.73346
     98.69604             98.73347
    128.30486            128.36202
    128.30486            128.40054
    167.78327            167.88384
    167.78327            167.88633
```

4 Examples in FreeFEM Documentation

We in this section present several examples given in FreeFEM. Many more examples can be found or will be added in the example folder in varFEM.

4.1 Membrane

This is an exmple given in FreeFEM Documentation: Release 4.6 (see Subsection 2.3 - Membrane).

The equation is simply the Laplace equation, where the region is an ellipse with the length of the semimajor axis $a = 2$, and unitary the semiminor axis. The mesh on such a domain can be generated by using the pdetool:

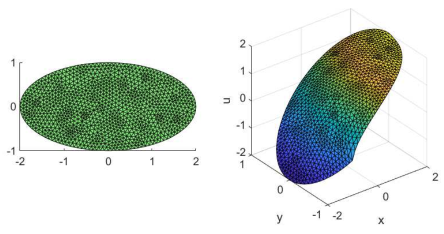
```
%% Mesh
% ellipse with a = 2, b = 1
a = 2; b = 1;
g = ellipseg(a,b);
[p,e,t] = initmesh(g,'hmax',0.2);
[p,e,t] = refinemesh(g,p,e,t);
node = p'; elem = t(1:3,:)';
figure,
subplot(1,2,1),
showmesh(node,elem);
% bdStr
bdNeumann = 'y<0 & x>-sin(pi/3)'; % string for Neumann
% mesh info
Th = FeMesh2d(node,elem,bdNeumann);
```

The Neumann boundary condition is imposed on

$$\Gamma_2 = \{(x,y) : x = a\cos t, \ \ y = b\sin t, \quad t \in (\theta, 2\pi)\}, \qquad \theta = 4\pi/3,$$

which can be identified by setting `bdStr = 'y<0 & x>-sin(pi/3)'` as done in the code.

The remaining implementation is very simple, so we omit the details (see `test_membrane.m`). The nodal values are shown in Fig. 15a.


(a) Nodal values

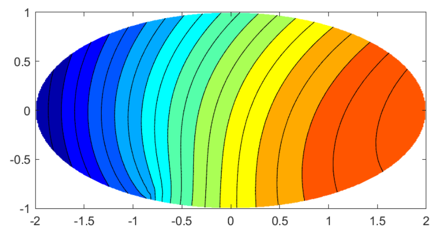

(b) Level lines

Fig. 15: Membrane deformation

We can also plot the level lines of the membrane deformation, as given in Fig. 15b.

4.2 Heat exchanger

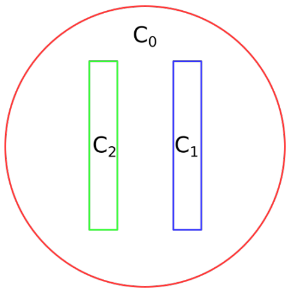


Fig. 16: Heat exchanger geometry (see [2])

The geometry is shown in Fig. 16, where C_1 and C_2 are two thermal conductors within an enclosure C_0. The temperature u satisfies

$$\nabla \cdot (\kappa \nabla u) = 0 \quad \text{in } \Omega.$$

- The first conductor is held at a constant temperature $u_1 = 100°\text{C}$, and the border of enclosure C_0 is held at temperature $20°\text{C}$. This means the domain is $\Omega = C_0 \backslash C_1$, and the boundaries consist of ∂C_0 and ∂C_1.
- The conductor C_2 has a different thermal conductivity than the enclosure C_0: $\kappa_0 = 1$ and $k_2 = 3$.

We use the mesh generated by FreeFEM, which is saved in a .msh file named `meshdata_heatex.msh`. The command in FreeFEM can be written as

```
savemesh(Th, "meshdata_heatex.msh");
```

We read the basic data structures `node` and `elem` via a self-written function:

```
[node,elem] = getMeshFreeFEM('meshdata_heatex.msh');
```

Then the mesh data can be computed as

```
% bdStr
```

```
C0 = 'x.^2 + y.^2 > 3.8^2'; % 1
bdStr = C0;
% mesh info
Th = FeMesh2d(node,elem,bdStr);
```

Note that the remaining boundary is ∂C_1, hence the boundaries of C_0 and the first conductor are labelled as 1 and 2, respectively.

The coefficient κ can be written as

```
%% PDE data
kappa = @(p) 1 + 2*(p(:,1)<-1).*(p(:,1)>-2).*(p(:,2)<3).*(p(:,2)>-3);  % p = [x,y]
```

And the bilinear form and the linear form are assembled as follows.

```
%% Bilinear form
Vh = 'P1'; quadOrder = 5;
Coef = kappa;  Test = 'v.grad';  Trial = 'u.grad';
kk = assem2d(Th,Coef,Test,Trial,Vh,quadOrder);

%% Linear form
ff = zeros(size(kk,1),1);
```

The Dirichlet boundary conditions are imposed in the following way:

```
%% Dirichlet boundary conditions
on = [1,2];
gBc1 = @(p) 20+0*p(:,1);
gBc2 = @(p) 100+0*p(:,1);
uh = apply2d(on,Th,kk,ff,Vh,gBc1,gBc2);
```

Here, `gBc1` is for ∂C_0 or `on(1)`, and `gBc2` is for ∂C_1 or `on(2)`.

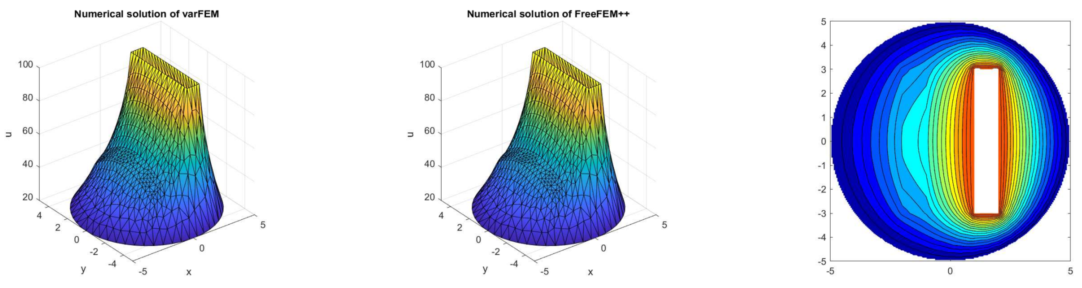
In FreeFEM, the numerical solution can be outputted using the command `ofstream`:

```
ofstream file("sol_heatex.txt");
file<<u[]<<endl;
```

The saved information is then imported by

```
uff = solFreeFEM('sol_heatex.txt');
```

The results are shown in Fig. 17, from which we observe that the varFEM solution is well matched with the one given by FreeFEM (see `test_heatex.m`).



(a) varFEM (b) FreeFEM (c) Level lines

Fig. 17: Numerical solutions given by varFEM and FreeFEM

4.3 Airfoil

This is an exmple given in FreeFem Documentation: Release 4.6 (Subsection 2.7 - Irrotational Fan Blade Flow and Thermal effects).

Consider a wing profile S (the NACA0012 Airfoil) in a uniform flow. Infinity will be represented by a large circle C where the flow is assumed to be of uniform velocity. The domain is outside S, with the mesh shown in Fig. 18. The NACA0012 airfoil is a classical wing profile in aerodynamics, whose equation for the upper surface is

$$y = 0.17735\sqrt{x} - 0.075597x - 0.212836x^2 + 0.17363x^3 - 0.06254x^4.$$

With this equation, we can generate a mesh using the Matlab pdetool, as included in varFEM. The function is `mesh_naca0012.m`. For comparison, we use the mesh generated by FreeFEM.

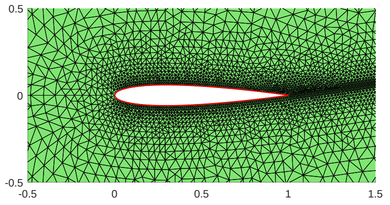
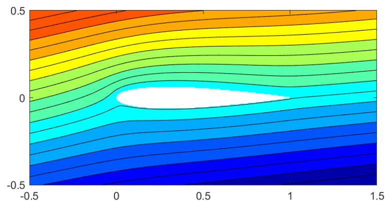


Fig. 18: Mesh zoomed around the NACA0012 airfoil

The programming is very simple, given by

```
%% Parameters
theta = 8*pi/180;
lift = theta*0.151952/0.0872665; % lift approximation formula
uinfty1 = cos(theta);   uinfty2 = sin(theta);

%% Mesh
[node,elem] = getMeshFreeFEM('meshdata_airfoil.msh');
% mesh info
bdStr = 'x.^2 + y.^2 > 4.5^2'; % 1-C
Th = FeMesh2d(node,elem,bdStr);

%% Bilinear form
Vh = 'P2';  quadOrder = 7;
Coef  = 1;
Test  = 'v.grad';
Trial = 'u.grad';
kk = assem2d(Th,Coef,Test,Trial,Vh,quadOrder);

%% Linear form
ff = zeros(size(kk,1),1);

%% Dirichlet boundary conditions
on = [1,2];
gBc1 = @(p) uinfty1*p(:,2) - uinfty2*p(:,1); % on 1-C
gBc2 = @(p) -lift + 0*p(:,1);  % on 2-S
uh = apply2d(on,Th,kk,ff,Vh,gBc1,gBc2);
```

We refer the reader to the FreeFEM Documentation for more details (The code is `potential.edp` in the software). The zoomed solutions of the streamlines are shown in Fig. 19, where the varFEM solution is well matched with the one given by FreeFEM (see `test_Airfoil.m`).


(a) varFEM solution

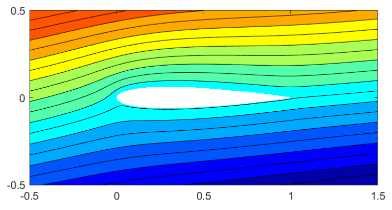

(b) FreeFEM solution

Fig. 19: Zoom around the NACA0012 airfoil showing the streamlines

4.4 Newton method for the steady Navier-Stokes equations

For the introduction of the problem, please refer to FreeFEM Documentation (Subsect. 2.12 - Newton Method for the Steady Navier-Stokes equations).

In each iteration, one needs to solve the following variational problem: Find $(\delta\boldsymbol{u}, \delta p)$ such that

$$DF(\delta\boldsymbol{u}, \delta p; \boldsymbol{u}, p) = F(\boldsymbol{u}, p),$$

where $\boldsymbol{u}$ and p are the solutions given in the last step, and

$$DF(\delta\boldsymbol{u}, \delta p; \boldsymbol{u}, p) = \int_\Omega ((\delta\boldsymbol{u}\cdot\nabla)\boldsymbol{u})\cdot\boldsymbol{v} + ((\boldsymbol{u}\cdot\nabla)\delta\boldsymbol{u})\cdot\boldsymbol{v} + \nu\nabla\delta\boldsymbol{u} : \nabla v - q\nabla\cdot\delta\boldsymbol{u},$$

$$F(\boldsymbol{u}, p) = \int_\Omega ((\boldsymbol{u}\cdot\nabla)\boldsymbol{u})\cdot\boldsymbol{v} + \nu\nabla\boldsymbol{u} : \nabla v - p\nabla\boldsymbol{u} : \nabla\boldsymbol{v} - q\cdot\boldsymbol{u},$$

where $(\boldsymbol{v}, q)$ are the test functions.

The finite element spaces and the quadrature rule are

```
Vh = {'P2','P2','P1'}; % v = [v1,v2,q] --> [v1,v2,v3]
quadOrder = 7;
vstr = {'v1','v2','q'}; ustr = {'du1','du2','dp'};
```

Here `vstr` and `ustr` are for the test and trial functions, respectively.

In the following, we only provide the detail for the assembly of the first term of DF, i.e.,

$$\int_\Omega ((\delta\boldsymbol{u}\cdot\nabla)\boldsymbol{u})\cdot\boldsymbol{v} = \int_\Omega u_{1,x}\cdot v_1\delta u_1 + u_{1,y}\cdot v_1\delta u_2 + u_{2,x}\cdot v_2\delta u_1 + u_{2,y}\cdot v_2\delta u_2. \tag{4.1}$$

In the iteration, $u_{1,x}$, $u_{1,y}$, $u_{2,x}$ and $u_{2,y}$ are the known coefficients, which can be obtained from the finite element function $\boldsymbol{u}$ in the last step. Let us discuss the implementation of varFEM:

- The initial data are

```
% initial data
u1 = @(p) double( (p(:,1).^2 + p(:,2).^2 > 2) );
u2 = @(p) 0*p(:,1);
p = @(p) 0*p(:,1);
```

  We must convert them into the dof vectors in view of the iteration.

```
uh1 = interp2d(u1,Th,Vh{1});
uh2 = interp2d(u2,Th,Vh{2});
ph = interp2d(p,Th,Vh{3});
```

- For numerical integration, it is preferable to provide the coefficient matrices for these coefficient functions.

```
u1c = interp2dMat(uh1,'u1.val',Th,Vh{1},quadOrder);
u2c = interp2dMat(uh2,'u2.val',Th,Vh{2},quadOrder);
pc = interp2dMat(ph,'p.val',Th,Vh{3},quadOrder);

u1xc = interp2dMat(uh1,'u1.dx',Th,Vh{1},quadOrder);
u1yc = interp2dMat(uh1,'u1.dy',Th,Vh{1},quadOrder);
u2xc = interp2dMat(uh2,'u2.dx',Th,Vh{2},quadOrder);
u2yc = interp2dMat(uh2,'u2.dy',Th,Vh{2},quadOrder);
```

  Here, `interp2dMat.m` provides a way to get the coefficient matrices only from the dof vectors.

- The triple `(Coef,Test,Trial)` for (4.1) is then given by

```
Coef = { u1xc, u1yc, u2xc, u2yc};
Test  = { 'v1.val', 'v1.val', 'v2.val', 'v2.val'};
Trial = { 'du1.val', 'du2.val', 'du1.val', 'du2.val'};
```

  A complete correspondence of $DF(\delta\boldsymbol{u}, \delta p; \boldsymbol{u}, p)$ can be listed in the following:

```
Coef = { u1xc, u1yc, u2xc, u2yc,  ... % term 1
    u1c, u2c, u1c, u2c,  ... % term 2
    nu, nu, nu, nu, ... % term 3
    -1, -1, ... % term 4
    -1, -1, ... % term 5
    -eps ... % stablization term
    };
Test  = { 'v1.val', 'v1.val', 'v2.val', 'v2.val', ... % term 1
    'v1.val', 'v1.val', 'v2.val', 'v2.val', ... % term 2
    'v1.dx', 'v1.dy', 'v2.dx', 'v2.dy', ... % term 3
    'v1.dx', 'v2.dy', ... % term 4
    'q.val', 'q.val', ... % term 5
    'q.val' ... % stablization term
    };
Trial = { 'du1.val', 'du2.val', 'du1.val', 'du2.val',... % term 1
    'du1.dx', 'du1.dy', 'du2.dx', 'du2.dy', ... % term 2
    'du1.dx', 'du1.dy', 'du2.dx', 'du2.dy', ... % term 3
    'dp.val', 'dp.val', ... % term 4
    'du1.dx', 'du2.dy', ... % term 5
    'dp.val' ... % stablization term
    };
```

- The computation of the stiffness matrix reads

```
[Test,Trial] = getStdvarForm(vstr, Test, ustr, Trial);
[kk,info] = int2d(Th,Coef,Test,Trial,Vh,quadOrder);
```

  Here `getStdvarForm.m` transforms the user-defined notation to the standard one.

We remark that as in Subsect. 3.6, some fixed stiffness matrices or load vectors can be computed in advance so as to save computational losses. The complete test script is available in `test_NSNewton.m`. The varFEM solutions and the FreeFEM solutions are shown in Fig. 20.

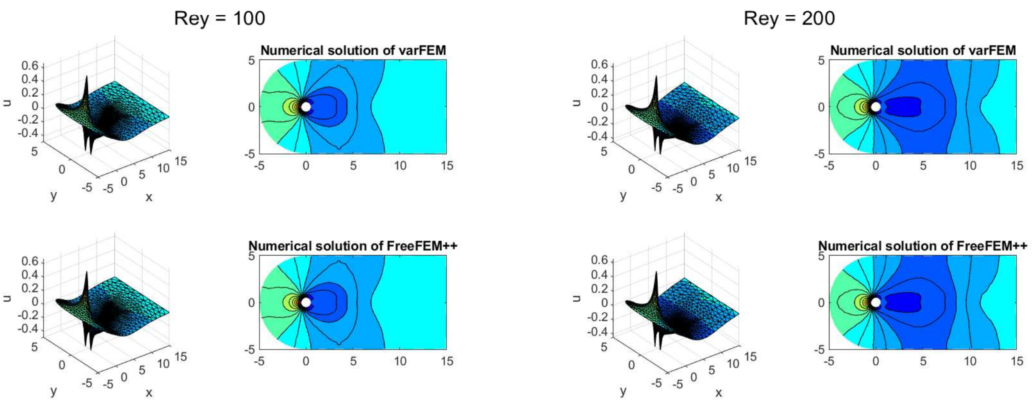


Fig. 20: The pressure p of a flow

4.5 Optimal control

4.5.1 The gradient is not provided

This is an exmple given in FreeFem Documentation: Release 4.6 (Subsection 2.15 - Optimal Control).

For a given target u_d, the problem is to find u such that

$$\min_{z\in\mathbb{R}^3} J(z) = \int_E (u-u_d)^2 = \int_\Omega I_E(u-u_d)^2, \quad z=(b,c,d),$$

where $E \subset \Omega$, I_E is an indication function, and u is the solution of the following PDE:

$$\begin{cases} -\nabla(\kappa(b,c,d)\cdot\nabla u) = 0 & \text{in } \Omega, \\ u = u_\Gamma & \text{on } \Gamma \subset \partial\Omega. \end{cases}$$

Let B, C and D be the separated subsets of Ω. The coefficient κ is defined as

$$\kappa(x) = 1 + bI_B(x) + cI_C(x) + dI_D(x), \quad x \in \Omega.$$

For fixed $z = (b,c,d)$, one can solve the PDE to obtain an approximate solution u_h and the approximate objective function:

$$J_h(z,u_h,u_d) := \int_\Omega I_E(u_h(z)-u_d)^2.$$

We use the built-in function `fminunc.m` in Matlab to find the (local) minimizer. To this end, we first establish a function to get the PDE solution:

```
%% Problem for the PDE constraint
function uh = PDEcon(z,Th)

    % Parameters
    Vh = Th.Vh; quadOrder = Th.quadOrder;
    Ib = @(p) (p(:,1).^2 + p(:,2).^2 < 1.0001);
    Ic = @(p) ( (p(:,1)+3).^2 + p(:,2).^2 < 1.0001);
    Id = @(p) (p(:,1).^2 + (p(:,2)+3).^2 < 1.0001);


    % Bilinear form
    Coef = @(p) 1 + z(1)*Ib(p) + z(2)*Ic(p) + z(3)*Id(p);
    Test = 'v.grad';
    Trial = 'u.grad';
    kk = assem2d(Th,Coef,Test,Trial,Vh,quadOrder);

    % Linear form
    ff = zeros(size(kk,1),1);

    % Dirichlet boundary condition
    gD = @(p) p(:,1).^3 - p(:,2).^3;
    on = 1;
    uh = apply2d(on,Th,kk,ff,Vh,gD);
end
```

The cost function is then given by

```
%% Cost function
function err = J(z,ud,Th)
    Ie = @(p) ((p(:,1)-1).^2 + p(:,2).^2 ≤4);
    uh = PDEcon(z,Th);

    fh = Ie(Th.node).*(uh-ud).^2;
    err = integral2d(Th,fh,Th.Vh,Th.quadOrder);
end
```

Given a vector $z_d = [2, 3, 4]$, we can construct an "exact solution" solution u_d by solving the PDE. The minimizer is then given by

```
%% The constructed solution
zd = [2, 3, 4];
ud = PDEcon(zd,Th);

%% Find the mimimizer
options.LargeScale = 'off';
options.HessUpdate = 'bfgs';
options.Display = 'iter';
z0 = [1,1,1];
zmin = fminunc(@(z) J(z,ud,Th), z0, options)
```

In the Matlab command window, one can get the following information:

```
                                                        First-order
 Iteration  Func-count       f(x)        Step-size       optimality
     0           4          30.9874                          77.2
     1           8          9.87548       0.0129606          21.4
     2          12          6.27654               1          12.2
     3          16          4.76889               1           5.3
     4          20          4.47092               1          3.03
     5          24          3.46888               1          5.66
     6          28          2.13699               1          5.77
     7          32          1.06649               1          4.42
     8          36         0.542858               1          3.14
     9          40         0.295327               1          2.18
    10          44         0.179846               1           1.7
    11          48         0.114221               1          1.28
    12          52         0.063295               1         0.842
    13          56        0.0242571               1         0.366
    14          60       0.00729547               1         0.258
    15          64       0.00170544               1         0.107
    16          68      3.95891e-05               1        0.0257
    17          72      3.54227e-07               1       0.00546
    18          76      3.36066e-10               1      0.000159
    19          80      5.54979e-13               1      3.88e-06
```

```
Local minimum found.

Optimization completed because the size of the gradient is less than
the value of the optimality tolerance.

<stopping criteria details>

zmin =

    2.0000    3.0000    4.0000
```

We can observe that the minimizer is found after 19 iterations, with the solutions displayed in Fig. 21.

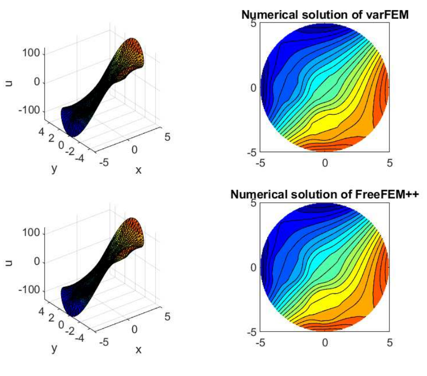


Fig. 21: Numerical solutions of the optimal control problem

4.5.2 The gradient is provided

For the example given in FreeFEM, the optimization problem is solved by the quasi-Newton BFGS method:

```
BFGS(J, DJ, z, eps=1.e-6, nbiter=15, nbiterline=20);
```

Here, `DJ` is the derivatives of J with respect to b, c, d. We also provide an implementation when the gradient is available. In this case, the cost function should be modified as

```
function [err,derr] = J(z,ud,Th)
end
```

Here `err` and `derr` correspond to `J` and `DJ`, respectively. For the details of the implementation, please refer to the test script `test_optimalControlgrad.m` in varFEM. The minimizer is then captured as follows.

```
1 %% Find the mimimizer
2 options = optimoptions('fminunc','Display','iter',...
3     'SpecifyObjectiveGradient',true); % gradient is provided
4 z0 = [1,1,1];
5 zmin = fminunc(@(z) J(z,ud,Th), z0, options)
```

Note that Line 3 indicates that the gradient is provided.

The minimizer is also found, with the printed information given as

```
 Iteration  Func-count       f(x)        Step-size       optimality
     0           1          30.9874                          77.2
     1           2          9.87549      0.0129606           21.4
     2           3          6.27653              1           12.2
     3           4          4.76889              1            5.3
     4           5          4.47091              1           3.03
     5          6         3.46888             1          5.66
     6          7           2.137             1          5.77
     7          8         1.06648             1          4.42
     8          9        0.542849             1          3.14
     9         10        0.295333             1          2.18
    10         11        0.179846             1           1.7
    11         12        0.114223             1          1.28
    12         13       0.0632962             1         0.842
    13         14       0.0242572             1         0.366
    14         15      0.00729561             1         0.258
    15         16       0.0017055             1         0.107
    16         17     3.95888e-05             1        0.0257
    17         18     3.54191e-07             1       0.00546
    18         19     3.47621e-10             1      0.000159
    19         20     4.31601e-13             1      3.87e-06

Local minimum found.

Optimization completed because the size of the gradient is less than
the value of the optimality tolerance.

<stopping criteria details>

zmin =

    2.0000    3.0000    4.0000
```

Compared with the previous implementation, one can find that the latter approach has fewer function calls (although we need to compute the gradient by solving a PDE problem).

5 Three-dimensional problems

5.1 Poisson equation

We consider the model problem (2.1) in three dimensions. The realization is very similar to that in 2-D, with the function file given as follows.

```
function uh = varPoisson3d(Th,pde,Vh,quadOrder)
%varPoisson3 Poisson equation in 3D: Pk Lagrange element (k≤2)

%   This function produces the finite element approximation of the
%   Poisson equation
%
%       -div(a*grad(u)) + cu = f  in \Omega, with
%       Dirichlet boundary condition u=g_D on \Gamma_D,
%       Robin boundary condition     g_R*u + a*grad(u)*n=g_N on \Gamma _R
%

% Quadrature orders for int1d, int2d and int3d
if nargin==2, Vh = 'P1'; quadOrder = 3; end % default: P1
if nargin==3, quadOrder = 3; end

%% Assemble stiffness matrix
% Omega
Coef  = {pde.a, pde.c};
Test  = {'v.grad', 'v.val'};
Trial = {'u.grad', 'u.val'};
kk = assem3d(Th,Coef,Test,Trial,Vh,quadOrder);

% Gamma_R
if ¬isempty(Th.bdStr)
    Th.on = 1;
    Coef  = pde.g_R;
    Test  = 'v.val';
    Trial = 'u.val';
    kk = kk + assem2d(Th,Coef,Test,Trial,Vh,quadOrder);
end

%% Assemble the right hand side
% Omega
Coef = pde.f;  Test = 'v.val';
ff = assem3d(Th,Coef,Test,[],Vh,quadOrder);
% Gamma_R
if ¬isempty(Th.bdStr)
    %Coef = @(p) pde.g_R(p).*pde.uexact(p) + pde.a(p).*(pde.Du(p)*n');

    fun = @(p) pde.g_R(p).*pde.uexact(p);
    Cmat1 = interpFaceMat(fun,Th,quadOrder);
    fun = @(p) repmat(pde.a(p),1,3).*pde.Du(p);
    Cmat2 = interpFaceMat(fun,Th,quadOrder);
    Coef = Cmat1 + Cmat2;

    ff = ff + assem2d(Th,Coef,Test,[],Vh,quadOrder);
end

%% Apply Dirichlet boundary value conditions
g_D = pde.g_D;
on = 2 - 1*isempty(Th.bdStr);
uh = apply3d(on,Th,kk,ff,Vh,g_D);
```

- For 3-D problems, we only provide $\mathbb{P}_k$-Lagrange elements with k up to 2. Note that $k = 3$ is not considered in FreeFEM as well.
- For boundary integrals, `interpEdgeMat.m` is now replaced by `interpFaceMat.m`.

The test script is given in the following.

```
%% Parameters
maxIt = 4;
N = zeros(maxIt,1);
h = zeros(maxIt,1);
ErrL2 = zeros(maxIt,1);
ErrH1 = zeros(maxIt,1);

%% Generate an intitial mesh
[node,elem3] = cubemesh([0 1 0 1 0 1],0.5);
bdStr = 'x==1';

%% Get the data of the pde
pde = Poissondata3var;
g_R = @(p) 1 + p(:,1) + p(:,2) + p(:,3); % 1 + x + y + z
pde.g_R = g_R;

%% Finite Element Method
i = 1; % 1,2
Vh = ['P', num2str(i)];
quadOrder = i+3;
for k = 1:maxIt
    % refine mesh
    % WARNING: order of vertices for each cell must be
    % consistent with the order in iFEM (see line 43 in cubemesh.m).
    [node,elem3] = uniformrefine3(node,elem3);
    % get the mesh information
    Th = FeMesh3d(node,elem3,bdStr);
    Th.solver = 'amg';
    % solve the equation
    uh = varPoisson3d(Th,pde,Vh,quadOrder);
    % record and plot
    N(k) = length(uh);
    h(k) = 1/(size(node,1)^(1/3)-1);
    if N(k) < 2e4  % show mesh and solution for small size
        figure(1);
        showresult3(node,elem3,pde.uexact,uh);
        drawnow;
    end
    % compute error
    ErrL2(k) = varGetL2Error3d(Th,pde.uexact,uh,Vh,quadOrder);
    ErrH1(k) = varGetH1Error3d(Th,pde.Du,uh,Vh,quadOrder);
end

%% Plot convergence rates and display error table
figure(2);
showrateh(h,ErrH1,ErrL2);
fprintf('\n');
disp('Table: Error')
colname = {'#Dof','h','||u-u_h||','|u-u_h|_1'};
disptable(colname,N,[],h,'%0.3e',ErrL2,'%0.5e',ErrH1,'%0.5e');
```

- For different dimensions, we always use `node` to denote the vertices in the mesh. The connectivity lists in 1-D, 2-D and 3-D are given by `elem1d`, `elem` and `elem3d`, respectively. We do not use `elem2d` since two dimensional problems are more typical for the FEMs.

- In the test script, we first generate an initial mesh for the cubic by using `cubemesh.m` given in *i*FEM, which is then uniformly refined in each iteration. It should be pointed out that uniform refine in 3D is not orientation preserved in `cubemesh.m`.

- For this reason, in `FeMesh3d.m` we must switch the vertices such that all signed volume is positive:

  ```
  elem3 = fixorder3(node,elem3);  % with reverse normal vector
  elem3(:,[2 3]) = elem3(:,[3 2]);
  ```

  Note that the `elem3` given by `fixorder3.m` has reverse normal vectors, so we add the second row.

- For small scale linear system, we directly solve it using the backslash command in Matlab, while for large systems the the Conjugate Gradients Squared Method or the algebraic multigrid method is used instead. One can set `Th.solver = 'cg'` or `Th.solver = 'amg'` to indicate the Conjugate Gradients Squared (CGS) Method or the algebraic multigrid method (AMG). Here, the CGS is realized by the built-in function `cgs.m` in Matlab, while the AMG can be found in *i*FEM (see `amg.m` there).

We display the solution at $z = z_{\min}$ in Fig. 22. The convergence rate is shown in Fig. 23, from which we clearly observe the optimal rates both for the H^1 norm and the L^2 norm.

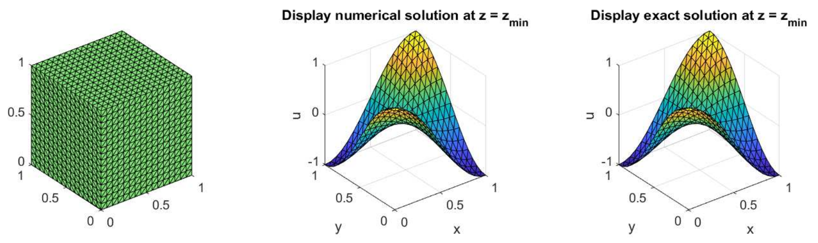
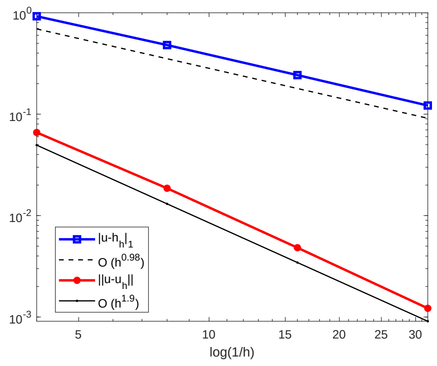
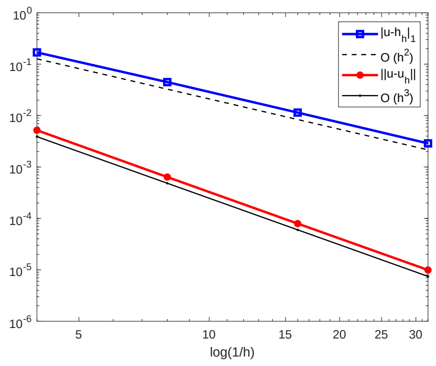


Fig. 22: Numerical and exact solutions for the 3-D Poisson equation



(a) P1



(b) P2

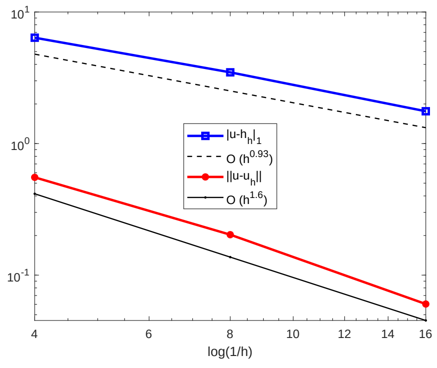
Fig. 23: Convergence rates for the 3-D Poisson equation

5.2 Linear elasticity problems

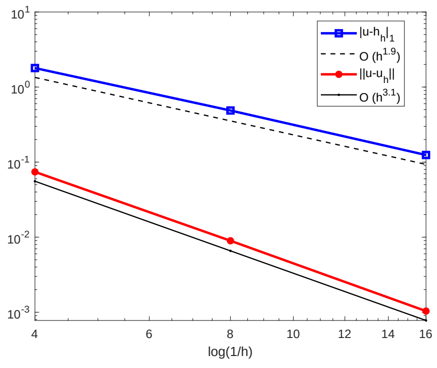
We only present the numerical results for the linear elasticity problem. Please refer to the test script `main_varElasticity3d.m` for details. The exact solution is given by

$$\begin{aligned}
u_1 &= (-1+\cos(2\pi x))\sin(2\pi y) + 1/(1+\lambda)\sin(\pi x)\sin(\pi y),\\
u_2 &= -(-1+\cos(2\pi y))\sin(2\pi x) + 1/(1+\lambda)\sin(\pi x)\sin(\pi y),\\
u_3 &= -(-1+\cos(2\pi z))\sin(2\pi x) + 1/(1+\lambda)\sin(\pi x)\sin(\pi z).
\end{aligned}$$

For the initial mesh with $h = 0.5$, the convergence rates are shown in Fig. 24. We see that the optimal convergence rate is obtained for the $\mathbb{P}_2$ element, while the result is suboptimal for the $\mathbb{P}_1$ element in the L^2 norm. The reason lies in the coarse mesh. One can run the code on a sequence of meshes with smaller sizes. For instance, Fig. 25 shows the result with initial size 0.25, in which case the better rates of convergence are obtained. In particular, when only the Dirichlet boundary conditions are present, the optimal rates are observed. However, the number of cells given by the uniform refinement grows too fast, making it difficult to compute on a personal computer. For example, for the initial mesh with size 0.25, the number of cells will be 3072, 24576, 196608 and 1572864 in the subsequent refinements.

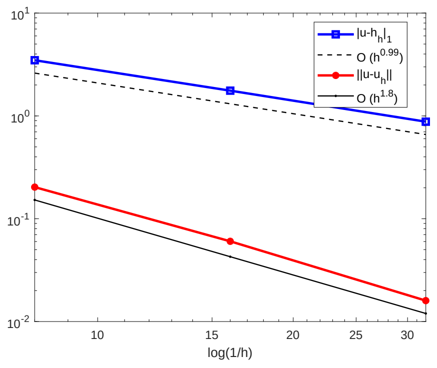


(a) P1

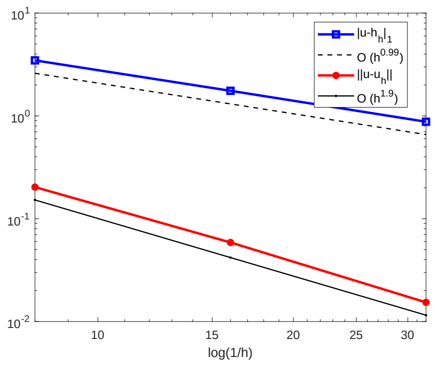


(b) P2

Fig. 24: Convergence rates for the 3-D elasticity problem



(a) `bdStr = 'x==1'`



(b) `bdStr = []`

Fig. 25: Convergence rates for the 3-D elasticity problem ($\mathbb{P}_1$ element)

6 Concluding remarks

In this paper, a Matlab software package for the finite element method was presented for various typical problems, which realizes the programming style in FreeFEM. The usage of the library, named varFEM, was demonstrated through various examples. Possible extensions of this library that are of interest include other types of finite elements and various applications. One can also develop a function to automatically identify the triple `(Coef, Trail, Test)` from the variational expression in FreeFEM.

References

[1] L. Chen. iFEM: an integrated finite element method package in Matlab. Technical report, University of California at Irvine, 2009.

[2] F. Hecht. New development in FreeFem++. *J. Numer. Math.*, 20(3-4):251–265, 2012.