varFEM: 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 paper summarizes the development of varFEM, which provides a realization of the programming style in FreeFEM by using the Matlab language.

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 jFEM, 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 $P_k$-Lagrange ($k \leq 3$) FEMs in two dimensions by utilizing the Matlab language. The underlying idea can be generalized to other types of finite 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 biharmonic equation and Stokes problem, and the iterative scheme for the heat equation. We also demonstrate the ability of varFEM to solve complex problems in Section 4.

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:

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

(2.1)

where $\Gamma_D$ and $\Gamma_R = \partial \Omega \setminus \Gamma_D$ 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 $u \in V := H^1_0(\Omega)$ such that

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

where

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

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

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

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

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

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

on the approximated domain, where for simplicity we have used the original notation $\Omega$ to represent a triangular mesh. The computation in varFEM reads
Here, \( \Theta \) represents the triangular mesh, which provides some necessary auxiliary data structures. We set up the triple \((\text{Coef}, \text{Test}, \text{Trial})\), for the coefficients, test functions and trial functions in variational form, respectively. It is obvious that \( v.\text{grad} \) is for \( \nabla v \) and \( v.\text{val} \) is for \( v \) itself.

The routine \texttt{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 \( \Phi_i \) are the global shape functions of the finite element space \( \mathcal{V}_h \). The integral of the bilinear form, as \( (\nabla \Phi_i, \nabla \Phi_j)_{\Omega} \), will be approximated by using the Gaussian quadrature formula with \texttt{quadOrder} being the order of accuracy.

The second step is to compute the stiffness matrix for the bilinear form on the Robin boundary \( \Gamma_R \):

\[
\int_{\Gamma_R} g_R v u \, ds.
\]

The code can be written as follows.

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

Here, \texttt{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 \texttt{elem1d} of the boundary edges of \( \Gamma_R \) and the associated indices \texttt{elem1dIdx} in the data structure \texttt{edge} introduced later.

### 2.1.2 The assembly of linear forms

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

\[
\int_{\Omega} f v \, ds.
\]

The load vector can be assembled as

```c
1 % Omega
2 Coef = pde.f; Test = 'v.val';
3 ff = int2d(Th,Coef,Test,[],Vh,quadOrder);
```

We set \texttt{Trial = []} to indicate the linear form.

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

\[
\int_{\Gamma_R} g_N v u \, ds
\]

reads
\begin{verbatim}
1 % Gamma_R
2 Coef = g_N; Test = 'v.val';
3 ff = ff + int1d(Th, Coef, Test, [], Vh, quadOrder);
\end{verbatim}

### 2.2 Data structures for triangular meshes

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

\begin{verbatim}
1 Th = FeMesh2d(node, elem, bdStr);
\end{verbatim}

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 \( P^k \)-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 \( k = 1, 2, \ldots, NE \). 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 \url{https://www.math.uci.edu/~chenlong/ifemdoc/mesh/auxstructuredoc.html} for some detailed information.

To deal with boundary integrals, we first extract 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 \( \Omega = (0, 1)^2 \),

- \( \text{bdStr} = \{'x==1'\} \) divides bdEdge into two parts: bdEdgeType\{1\} gives the boundary edges on \( x = 1 \), and bdEdgeType\{2\} stores the remaining part.
- \( \text{bdStr} = \{'x==1', 'y==0'\} \) 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.
- \( \text{bdStr} = [] \) 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 \texttt{int2d.m} and \texttt{assem2d.m}

In this article we only discuss the implementation of the bilinear forms in two dimensions.
2.3.1 The scalar case: \texttt{assem2d.m}

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

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

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, \texttt{assem2d.m} is essentially the same as \texttt{int2d.m}, while the later one can be used to deal with vector cases like linear elasticity problems. To handle different spaces, we write \( V_h = \{ 'P1', \ldots , 'P2' \} \) for the input of \texttt{assem2d.m}, where \( V_h\{1\} \) is for \( v \) and \( V_h\{2\} \) is for \( u \). For simplicity, it is also allowed to write \( V_h = 'P1' \) 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

\[
\int_K avu dx, \quad \int_K av_x u dx, \quad \int_K av_y u d\sigma, \\
\int_K av_x u_x d\sigma, \quad \int_K av_x u_x d\sigma, \quad \int_K av_y u_x d\sigma, \\
\int_K av_y u_x d\sigma, \quad \int_K av_x u_y d\sigma, \quad \int_K av_y u_y d\sigma.
\]

Of course, we often encounter the gradient form

\[
\int_K a \nabla v \cdot \nabla u d\sigma = \int_K a(v_x u_x + v_y u_y) d\sigma.
\]

We take the second bilinear form as an example. Let

\[ a_K(v, u) = \int_K av_x u_x d\sigma, \]

and consider the \( P_1 \)-Lagrange finite element. Denote the local basis functions to be \( \phi_1, \phi_2, \phi_3 \). Then the local stiffness matrix is

\[
A_K = \int_K a \begin{bmatrix}
\partial_x \phi_1 \\
\partial_x \phi_2 \\
\partial_x \phi_3
\end{bmatrix}
\begin{bmatrix}
\phi_1 \\
\phi_2 \\
\phi_3
\end{bmatrix} d\sigma.
\]

Let

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

Then

\[
A_K = (k_{ij})_{3 \times 3}, \quad k_{ij} = \int_K av_i u_j d\sigma.
\]

The integral will be approximated by the Gaussian quadrature rule:

\[
k_{ij} = \int_K av_i u_j d\sigma = |K| \sum_{p=1}^{n_p} w_p a(x_p, y_p) v_i(x_p, y_p) u_j(x_p, y_p),
\]
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:

\[
\begin{align*}
    w_p, v(\cdot, p) = \begin{bmatrix}
        v_1(x_1^p, y_1^p) \\
        v_2(x_1^p, y_1^p) \\
        \vdots \\
        v_{NT}(x_{NT}^p, y_{NT}^p)
    \end{bmatrix}, \quad u(\cdot, p) = \begin{bmatrix}
        u_1(x_1^p, y_1^p) \\
        u_2(x_1^p, y_1^p) \\
        \vdots \\
        u_{NT}(x_{NT}^p, y_{NT}^p)
    \end{bmatrix}, \quad p = 1, \ldots, n_g,
\end{align*}
\]

where \(v_1\) associated with \(v_1\) is of size \(NT \times ng\) with the \(p\)-th column given by \(v(\cdot, p)\). Let \(\text{weight} = [w_1, w_2, \ldots, w_{ng}]\) and \(\text{ww} = \text{repmat}(\text{weight}, NT, 1)\). Then \(k_{ij}\) for \(a = 1\) can be computed as

\[
\begin{align*}
    &1 \quad k_{11} = \text{sum}(\text{ww} \ast v_1 \ast u_1 , 2); \\
    &2 \quad k_{12} = \text{sum}(\text{ww} \ast v_1 \ast u_2 , 2); \\
    &3 \quad k_{13} = \text{sum}(\text{ww} \ast v_1 \ast u_3 , 2); \\
    &4 \quad k_{21} = \text{sum}(\text{ww} \ast v_2 \ast u_1 , 2); \\
    &5 \quad k_{22} = \text{sum}(\text{ww} \ast v_2 \ast u_2 , 2); \\
    &6 \quad k_{23} = \text{sum}(\text{ww} \ast v_2 \ast u_3 , 2); \\
    &7 \quad k_{31} = \text{sum}(\text{ww} \ast v_3 \ast u_1 , 2); \\
    &8 \quad k_{32} = \text{sum}(\text{ww} \ast v_3 \ast u_2 , 2); \\
    &9 \quad k_{33} = \text{sum}(\text{ww} \ast v_3 \ast u_3 , 2); \\
    &10 \quad K = [k_{11}, k_{12}, k_{13}, k_{21}, k_{22}, k_{23}, k_{31}, k_{32}, k_{33}];
\end{align*}
\]

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

\[
\begin{align*}
    &1 \quad \text{Ndof} = 3; \\
    &2 \quad K = \text{repmat}(@area,1,\text{Ndof}^2) \ast K;
\end{align*}
\]

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

\[
\begin{align*}
    &1 \quad \text{cf} = @(pz) \text{pz}(:,1) + \text{pz}(:,2); \ % x+y; \\
    &2 \quad \text{cc} = \text{zeros}(NT,ng); \\
    &3 \quad \text{for} \ p = 1:ng \\
    &4 \quad \text{pz} = \text{lambda}(p,1) \ast z1 + \text{lambda}(p,2) \ast z2 + \text{lambda}(p,3) \ast z3; \\
    &5 \quad \text{cc}(:,p) = \text{cf}(\text{pz}); \\
    &6 \quad \text{end}
\end{align*}
\]

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

\[
\begin{align*}
    &1 \quad K = \text{zeros}(NT,\text{Ndof}^2); \\
    &2 \quad s = 1; \\
    &3 \quad v = \{v1,v2,v3\}; u = \{u1,u2,u3\}; \\
    &4 \quad \text{for} \ i = 1:\text{Ndof} \\
    &5 \quad \text{for} \ j = 1:\text{Ndof} \\
    &6 \quad \text{vi} = \text{v(i)}; \text{uj} = \text{u(j)}; \\
    &7 \quad K(:,s) = \text{area} \ast \text{sum}(\text{ww} \ast \text{cc} \ast \text{vi} \ast \text{uj},2); \\
    &8 \quad s = s+1; \\
    &9 \quad \text{end} \\
    &10 \quad \text{end}
\end{align*}
\]

The bilinear form is assembled by using the build-in function \text{sparse.m} as in iFEM. In this case, the code is given as

\[
\begin{align*}
    &1 \quad ss = K(:,); \\
    &2 \quad kk = \text{sparse}(ii,jj,ss,\text{NNdof},\text{NNdof});
\end{align*}
\]
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

```matlab
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 \(\phi\) is a local basis function.

```matlab
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

\[
a_K(v, u) := \int_K \varepsilon(v) : \varepsilon(u) d\sigma,
\]

where \(v = (v_1, v_2)^T\), \(u = (u_1, u_2)^T\), and

\[
\varepsilon(v) : \varepsilon(u) = v_{1,x}u_{1,x} + v_{2,y}u_{2,y} + \frac{1}{2}(v_{1,y} + v_{2,x})(u_{1,y} + u_{2,x})
\]

\[
= v_{1,x}u_{1,x} + v_{2,y}u_{2,y} + \frac{1}{2}(v_{1,y}u_{1,y} + v_{1,y}u_{2,x} + v_{2,x}u_{1,y} + v_{2,x}u_{2,x}).
\] (2.2)

The stiffness matrix can be assembled as

```matlab
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.dy', '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

```matlab
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 \(v = (v_1, v_2, v_3)\) and \(u = (u_1, u_2, u_3)\), and suppose that \(a(v, u)\) 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:

\[ a(v,u) \leftrightarrow [v_1,v_2,v_3] \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix}, \]

where \( u_i \) 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 \( a(v,u) \).

Let the sparse index for \( A_{ij} \) be \((i_{ij},j_{ij},s_{ij})\). 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

\[
\begin{bmatrix}
i_{11} & i_{12} & i_{13} \\
i_{21} + m_1 & i_{22} + m_1 & i_{23} + m_1 \\
i_{31} + m_2 & i_{32} + m_2 & i_{33} + m_2
\end{bmatrix}
\quad \text{and} \quad
\begin{bmatrix}
\bar{j}_{11} & \bar{j}_{12} + n_1 & \bar{j}_{13} + n_2 \\
\bar{j}_{21} & \bar{j}_{22} + n_1 & \bar{j}_{23} + n_2 \\
\bar{j}_{31} & \bar{j}_{32} + n_1 & \bar{j}_{33} + n_2
\end{bmatrix},
\]

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

### 3 Tutorial examples

In this section, we present several examples to demonstrate the use of varFEM.

#### 3.1 Poisson-type problems

We now provide the complete implementation of the model problem (2.1). The function file reads

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

%% Assemble stiffness matrix
% Omega
Coef = { pde.a, pde.c};
Test = {'v.grad', 'v.val'};
Trial = {'u.grad', 'u.val'};
kk = assem2d(Th,Coef,Test,Trial,Vh,quadOrder); % or assem2d

% Robin data
bdStr = Th.bdStr;
if ~isempty(bdStr)
    Th.elem1d = Th.bdEdgeType{1};
    Th.elem1dIdx = Th.bdEdgeIdxType{1};
    Coef = pde.g_R;
    Test = 'v.val';
    Trial = 'u.val';
    kk = kk + assem1d(Th,Coef,Test,Trial,Vh,quadOrder); % or assem1d
end

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

% Neumann data
if ~isempty(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 = interpEdgeMat(fun,Th,quadOrder);
  fun = @(p) repmat(pde.a(p),1,2).*pde.Du(p);
  Cmat2 = interpEdgeMat(fun,Th,quadOrder);
  Coef = Cmat1 + Cmat2;
  ff = ff + assem1d(Th,Coef,Test,[],Vh,quadOrder);
end

%% Apply Dirichlet boundary conditions
g_D = pde.g_D;
on = 2 - 1.*isempty(bdStr); % 1 for bdStr = [], 2 for bdStr = 'x==0'
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 31-35, 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.

The boundary edges will be divided into at most two parts. For example, when bdStr = ... 'x==0', the Robin boundary part is given by elem1d = Th.bdEdgeType{1}. The remaining part is for the Dirichlet boundary condition, with the index on = 1 given for Th.bdEdgeType.

The test script is presented as follows.

1  %% Parameters
2  maxIt = 5;
3  N = zeros(maxIt,1);
4  h = zeros(maxIt,1);
5  ErrL2 = zeros(maxIt,1);
6  ErrH1 = zeros(maxIt,1);
7 8  %% Generate an initial mesh
9  [node,elem] = squaremesh([0 1 0 1],0.5);
10  bdStr = 'x==0';
11 12  %% Get PDE data
13  pde = Poissondatavar;
14  g_R = @(p) 1 + p(:,1) + p(:,2); % 1 + x + y
15  pde.g_R = g_R;
16 17  %% Finite Element
18  i = 1; % 1,2,3
\begin{verbatim}
19 Vh = ['P', num2str(i)];
20 quadOrder = i+2;
21 for k = 1:maxIt
22    \% refine mesh
23    [node, elem] = uniformrefine(node, elem);
24    \% get the mesh information
25    Th = FeMesh2d(node, elem, bdStr);
26    \% solve the equation
27    uh = varPoisson(Th, pde, Vh, quadOrder);
28    \% record and plot
29    N(k) = size(elem,1);
30    h(k) = 1/(sqrt(size(node,1))-1);
31    if N(k) < 2e3 \% show mesh and solution for small size
32       figure(1);
33       showresult(node, elem, pde.uexact, uh);
34       drawnow;
35    end
36    \% compute error
37    ErrL2(k) = varGetL2Error(Th, pde.uexact, uh, Vh, quadOrder);
38    ErrH1(k) = varGetH1Error(Th, pde.Du, uh, Vh, quadOrder);
39 end
40
41 \% Plot convergence rates and display error table
42 figure(2);
43 showrateh(h, ErrH1, ErrL2);
44 fprintf('\n');
45 disp('Table: Error')
46 colname = {'# Dof', 'h', '|u-u_h|', '|u-u_h|_1'};
47 dispTable(colname, N, h, ErrL2, ErrH1, '0.3e', '0.5e', '0.5e');
\end{verbatim}

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.

The test script can be easily used to compute $P_i$-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.

Fig. 1: The nodal values of exact and numerical solutions for the model problem
We next consider the example with a circular domain or an L-shaped domain. Such a domain can be generated by using the pdetool as

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

The results are given in Fig. 3
3.2 Linear elasticity problems

The linear elasticity problem is

\[
\begin{aligned}
-\text{div}\sigma &= f \quad \text{in } \Omega, \\
u &= 0 \quad \text{on } \Gamma_0, \\
\sigma n &= g \quad \text{on } \Gamma_1,
\end{aligned}
\]

(3.1)

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

\[
\sigma(u) = 2\mu\varepsilon(u) + \lambda(\text{div}\ u)I,
\]

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

### 3.2.1 The programming in scalar form

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

\[
-\mu\Delta u - (\lambda + \mu)\text{grad}(\text{div}\ u) = f \quad \text{in } \Omega,
\]

(3.2)

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

\[
\mu \int_\Omega \nabla u \cdot \nabla v dx + (\lambda + \mu) \int_\Omega (\text{div} \ u)(\text{div} \ v) d\sigma = \int_\Omega f \cdot v d\sigma.
\]

The first term of the bilinear form can be split into

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

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

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

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

\[
\int_\Omega v_{1,x}u_{1,x}dx, \quad \int_\Omega v_{1,x}u_{2,y}dx, \quad \int_\Omega v_{2,y}u_{1,x}dx \quad \int_\Omega v_{2,y}u_{2,y}dx,
\]

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

1. \% (v1.dx, u1.dx)
2. cf = 1;
3. Coef = cf; Test = 'v.dx'; Trial = 'u.dx';
4. B1 = assem2d(Th,Coef,Test,Trial,Vh,quadOrder);
5 \%(v_1.dx, u_2.dy)
6 \( \text{cf} = 1; \)
7 Coef = cf; Test = 'v.dx'; Trial = 'u.dy';
8 B2 = assem2d(Th,Coef,Test,Trial,Vh,quadOrder);
9 \%(v_2.dy, u_1.dx)
10 \( \text{cf} = 1; \)
11 Coef = cf; Test = 'v.dy'; Trial = 'u.dx';
12 B3 = assem2d(Th,Coef,Test,Trial,Vh,quadOrder);
13 \%(v_2.dy, u_2.dy)
14 \( \text{cf} = 1; \)
15 Coef = cf; Test = 'v.dy'; Trial = 'u.dy';
16 B4 = assem2d(Th,Coef,Test,Trial,Vh,quadOrder);

The block matrix is then given by
1 \%
2 \( \text{kk} \)
3 \( \text{kk} = \begin{bmatrix} \mu A + (\lambda + \mu) B_1 & (\lambda + \mu) B_2 \\
4 (\lambda + \mu) B_3 & \mu A + (\lambda + \mu) B_4 \end{bmatrix}; \)
5 \( \text{kk} = \text{sparse}(\text{kk}); \)

The right-hand side has two components:
\[
\int_\Omega f_1 v_1 dx \quad \text{and} \quad \int_\Omega f_2 v_2 dx.
\]

The load vector is assembled in the following way:
1 \%
2 \% Assemble right hand side
3 \%
4 trf = eye(2);
5 Coef = @(pz) pde.f(pz)*trf(:,1); Test = 'v.val'; \% f = [f1, f2]
6 F1 = assem2d(Th,Coef,Test,[],Vh,quadOrder);
7 \%
8 trf = eye(2);
9 Coef = @(pz) pde.f(pz)*trf(:,2); Test = 'v.val';
10 F2 = assem2d(Th,Coef,Test,[],Vh,quadOrder);
11 \%
12 ff = [F1; F2];

For the vector problem, we impose the Dirichlet boundary value conditions as follows.
1 \%
2 \% Apply Dirichlet boundary conditions
3 g_D = pde.g_D;
4 on = 1;
5 g_D1 = @(p) g_D(p)*[1;0];
6 g_D2 = @(p) g_D(p)*[0;1];
7 gBc = {g_D1,g_D2};
8 Vhvec = {Vh,Vh};
9 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.
Fig. 4: The nodal values of exact and numerical solutions for the elasticity problem of displacement type.

### 3.2.2 The programming in vector form

The bilinear form is

\[ a(v, u) = 2\mu \int_{\Omega} \varepsilon_{ij}(v)\varepsilon_{ij}(u)\sigma + \lambda \int_{\Omega} (\partial_i v_i)(\partial_j u_j)\sigma, \]

where the summation is omitted. The computation of the first term has been given in Subsect. 2.3.2, i.e.,

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

The second term can be computed as

```plaintext
1 % (div u, div v)
2 Coef = 1;
3 Test = 'v1.dx + v2.dy';
4 Trial = 'u1.dx + u2.dy';
5 B = int2d(Th, Coef, Test, Trial, Vh, quadOrder);
6 B = lambda*B;
7 % stiffness matrix
8 kk = A + B;
```

The linear form is

\[ \ell(v) = \int_{\Omega} f \cdot v\sigma + \int_{\Gamma} g \cdot v\sigma. \]

For the first term, one has

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

Note that we have added the implementation for \( f \cdot v \) by just setting \( Test = 'v.val' \). For the second term, we first determine the one-dimensional edges:

```plaintext
1 % Get 1D mesh for boundary integrals
2 bdStr = Th.bdStr;
3 if ~isempty(bdStr)
4    Th.elemId = Th.bdEdgeType(1);
5    Th.elemIdIdx = Th.bdEdgeIdxType(1);
6 end
```
The coefficient matrix for the boundary integral can be computed using `interpEdgeMat.m` and the Neumann condition is then realized as

```matlab
1  %% Assemble Neumann boundary conditions
2  if ~isempty(bdStr)
3      g_N = pde.g_N; trg = eye(3);
4  
5      g1 = @(p) g_N(p)*trg(:,[1,3]); Cmat1 = interpEdgeMat(g1,Th,quadOrder);
6      g2 = @(p) g_N(p)*trg(:,[3,2]); Cmat2 = interpEdgeMat(g2,Th,quadOrder);
7  
8      Coef = {Cmat1, Cmat2}; Test = 'v.val';
9      ff = ff + int1d(Th,Coef,Test,[],Vh,quadOrder);
10  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:

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

Note that $Vh$ is of vector form.

In the test script, we set `bdStr = 'y==0 | x==1'`. The errors for the $P_3$ element are listed in Tab. 1.

Tab. 1: The $L^2$ and $H^1$ errors for the elasticity problem of tensor form ($P_3$ element)

| Dof  | $h$      | ErrL2   | ErrH1   |
|------|----------|---------|---------|
|  32  | 2.50e-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 Mixed FEMs for the biharmonic equation

3.3.1 The programming in scalar form

For the mixed finite element methods, we first consider the biharmonic equation with Dirichlet boundary conditions:

\[
\begin{align*}
\Delta^2 u &= f & \text{in } & \Omega \subset \mathbb{R}^2, \\
\Delta u &= 0 & \text{on } & \partial \Omega.
\end{align*}
\]

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

\[
\begin{align*}
-\Delta u &= w, \\
-\Delta w &= f, \\
u &= \partial_n u = 0 & \text{on } & \partial \Omega.
\end{align*}
\]

15
The associated variational problem is: Find \((w, u) \in H^1(\Omega) \times H^1_0(\Omega) =: V \times U\) such that

\[
\begin{cases}
\int_{\Omega} \nabla u \cdot \nabla \phi \, d\sigma = \int_{\Omega} w \phi \, d\sigma, & \phi \in H^1(\Omega), \\
\int_{\Omega} \nabla w \cdot \nabla \psi \, d\sigma = \int_{\Omega} f \psi \, d\sigma, & \psi \in H^1_0(\Omega).
\end{cases}
\] (3.3)

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

One has

\[
\begin{cases}
a(w, \phi) + b(\phi, u) = 0, & \phi \in H^1(\Omega) = V, \\
b(w, \psi) = (f, \psi), & \psi \in H^1_0(\Omega) = U,
\end{cases}
\]

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 \(P_1\)-Lagrange elements. Let \(N\) be the vector of global basis functions. One easily gets

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

where

\[
A = -\int_{\Omega} N^T N d\sigma, \quad B = \int_{\Omega} \nabla N^T \cdot \nabla N d\sigma, \quad f = \int_{\Omega} N^T f d\sigma.
\]

The system can be written in block matrix form as

\[
\begin{bmatrix}
A & B \\
B^T & O
\end{bmatrix}
\begin{bmatrix}
w \\
u
\end{bmatrix}
= \begin{bmatrix} 0 \\ f \end{bmatrix}.
\]

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

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

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

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

```matlab
1 %% Assemble stiffness matrix
2 % matrix A
3 Coef = 1; Test = 'v.val'; Trial = 'u.val';
4 A = -assem2d(Th,Coef,Test,Trial,'P1',quadOrder);
5 % matrix B
6 Coef = 1; Test = 'v.grad'; Trial = 'u.grad';
7 B = assem2d(Th,Coef,Test,Trial,'P1',quadOrder);
8 % kk
9 O = zeros(size(B));
10 kk = [A, B; B', O];
11 kk = sparse(kk);
```

The right-hand side is given by

```matlab
1 %% Assemble right-hand side
2 Coef = pde.f; Test = 'v.val';
3 ff = assem2d(Th,Coef,Test,[],'P1',quadOrder);
4 O = zeros(size(ff));
5 ff = [O; ff];
```
The computation of the Neumann boundary condition reads

```matlab
%% 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

```matlab
%% Apply Dirichlet boundary conditions
on = 1;
g_D = pde.g_D;
gBc = {}; g_D;
Vhvec = {'P1','P1'};
U = apply2d(on,Th,kk,ff,Vhvec,gBc); % note Vhvec
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 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 (\( P_1 \) element)](image)

### 3.3.2 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 \( \mathbf{u} = (u_1,u_2)^T \) being the trial function and \( \mathbf{v} = (v_1,v_2)^T \) being the test function. One easily finds that the mixed form (3.3) is equivalent to the following vector form

\[
\int_{\Omega} (-v_1 u_1 + \nabla v_1 \cdot \nabla u_2 + \nabla v_2 \cdot \nabla u_1) \, d\sigma = \int_{\Omega} f v_2 \, dx + \int_{\partial \Omega} g_2 v_1 \, ds,
\]

which is obtained by adding the two equations.

Using `int2d.m` and `int1d.m`, we can compute the vector \( \mathbf{u} \) as follows.

```matlab
%% 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);
```
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 P₂ element, the order for L² is 1.5 and for H¹ is 0.5.

- For P₃ element, the order for L² is 2.5 and for H¹ is 1.5.

Note that our results are consistent with that given in iFEM. Obviously, for P₂ and P₃ elements, the rate of w has the behaviour of Δu, which is reasonable since w = −Δu.
3.4 Mixed FEMs for the Stokes problem

The Stokes problem with homogeneous Dirichlet boundary conditions is to find \((\mathbf{u}, p)\) such that
\[
\begin{align*}
-\nu \Delta \mathbf{u} - \nabla p &= f \quad \text{in } \Omega, \\
\text{div} \mathbf{u} &= 0 \quad \text{in } \Omega, \\
\mathbf{u} &= 0 \quad \text{on } \partial \Omega.
\end{align*}
\]
Define \(V = H^1_0(\Omega)\) and \(P = L^2_0(\Omega)\). The mixed variational problem is: Find \((\mathbf{u}, p)\) such that
\[
\begin{align*}
\int (a(\mathbf{u}, \mathbf{v}) + b(\mathbf{v}, p)) &= (f, \mathbf{v}), \quad \mathbf{v} \in V, \\
b(u, q) &= 0, \quad q \in P,
\end{align*}
\]
where
\[
a(\mathbf{u}, \mathbf{v}) = (\nu \nabla \mathbf{u}, \nabla \mathbf{v}), \quad b(\mathbf{v}, q) = (\text{div} \mathbf{v}, q).
\]

Let \(T_h\) be a shape regular triangulation of \(\Omega\). We consider the conforming finite element discretizations: \(V_h \subset V\) and \(P_h \subset P\). Typical pairs \((V_h, P_h)\) of stable finite element spaces include: MINI element, Girault-Raviart element and \(P_k - P_{k-1}\) elements. For the last one, a special example is the \(P_2 - P_1\) element, also known as the Taylor-Hood element, which is the one under consideration.

The FEM is to find \((\mathbf{u}_h, p_h)\) such that
\[
\begin{align*}
\int (a(u_h, v) + b(v, p_h)) &= F(v), \quad v \in V_h, \\
b(u_h, q) &= 0, \quad q \in P_h.
\end{align*}
\]
(3.4)

The problem (3.4) 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. We consider the latter one: Find \((\mathbf{u}_h, p_h)\) such that
\[
\begin{align*}
\int (a(\mathbf{u}_h, \mathbf{v}) + b(\mathbf{v}, p_h) + b(\mathbf{u}_h, q) - \varepsilon (p_h, q)) &= F(v), \quad v \in V_h, \\
&\in L^2(\Omega),
\end{align*}
\]
where \(\varepsilon\) is a small parameter to ensure stability.

The bilinear form can be assembled as follows.

Note that the symbols \(p\) and \(q\) correspond to \(u_3\) and \(v_3\), 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 \( \mathbf{v} = [v_1, v_2, v_3]^T \) 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);

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 \( f \) in such a way that the analytical solution is

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

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](image)
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 |

3.5 Time-dependent problems

As an example, we consider the heat equation:

\[
\begin{align*}
\frac{\partial u}{\partial t} - \Delta u &= f \quad \text{in } \Omega, \\
u(x, y, 0) &= u_0(x, y) \quad \text{in } \Omega, \\
u &= g_D \quad \text{on } \Gamma_D, \\
\frac{\partial u}{\partial n} &= g_N \quad \text{on } \Gamma_N.
\end{align*}
\]

After applying the backward Euler discretization in time, we shall seek $u^n(x, y)$ satisfying for all $v \in H^1_0(\Omega):$

\[
\int_{\Omega} \left( \frac{u^n - u^{n-1}}{\Delta t} v + \nabla u^n \cdot \nabla v \right) d\sigma = \int_{\Omega} f^n v d\sigma + \int_{\Gamma_N} g^n_N vds.
\]

We first generate a mesh and compute the mesh information.

The PDE data is given by

\[
\begin{align*}
\text{pde} &= \text{heatData;} \\
\text{u0} &= @(p) \text{pde.uxact}(p, t(1)); \\
\text{uh0} &= \text{interp2d(u0, Th, Vh)}; \quad \text{\% dof vector} \\
\text{uf} &= \text{zeros(Nt+1,2)}; \quad \text{\% record solutions at p-th point} \\
p &= 2*Nx; \\
u(1,:) &= [\text{uh0}(p), \text{uh0}(p)]; \\
\text{for } n = 1:Nt \\
\text{fun} &= @(p) \text{pde.f}(p, t(n+1));
\end{align*}
\]

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

For fixed $\Delta t$, the bilinear form gives the same stiffness matrix in each iteration.
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.elem1d = Th.bdEdgeType{1};
    Th.elem1dIdx = Th.bdEdgeIdxType{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;

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.

**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 \( O(\Delta t + h^{k+1}) \) for the \( P_k \)-Lagrange element. To test the convergence order w.r.t \( h \), one can set \( \Delta t = O(h^{k+1}) \). The numerical results are consistent
with the theoretical prediction as shown in Fig. 10.

![Graphs showing convergence rates for the heat equation.](a) P1  (b) P2  (c) P3)

Fig. 10: Convergence rates for the heat equation.

# 4 Examples in FreeFEM Documentation

We in this section present five 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 example 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:

```plaintext
1 % Mesh
2 % ellipse with a = 2, b = 1
3 a = 2; b = 1;
4 g = ellipseg(a,b);
5 [p,e,t] = initmesh(g,'hmax',0.2);
6 [p,e,t] = refinemesh(g,p,e,t);
7 node = p'; elem = t(1:3,:);
8 figure,
9 subplot(1,2,1),
10 showmesh(node,elem);
11 % bdStr
12 bdNeumann = 'y<0 & x>-sin(pi/3)'; % string for Neumann
13 % mesh info
14 Th = FeMesh2d(node,elem,bdNeumann);
```

The Neumann boundary condition is imposed on

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

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. The nodal values are shown in Fig. 11a.
We can also plot the level lines of the membrane deformation, as given in Fig. 11b. 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 build-in function `pdeInterpolant.m` in the pdetool. However, the build-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 a contour plot.

### 4.2 Heat exchanger

The geometry is shown in Fig. 12, where $C_1$ and $C_2$ are two thermal conductors within an enclosure $C_0$. The temperature $u$ satisfies

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

- The first conductor is held at a constant temperature $u_1 = 100^\circ C$, and the border of enclosure $C_0$ is held at temperature $20^\circ C$. This means the domain is $\Omega = C_0 \setminus C_1$, and the boundaries consist of $\partial C_0$ and $\partial C_1$.
- 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

```matlab
1 savemesh(Th, "meshdata_heatex.msh");
```
We read the basic data structures \texttt{node} and \texttt{elem} via a self-written function:

\begin{verbatim}
[node,elem] = getMeshFreeFEM('meshdata_heatex.msh');
\end{verbatim}

Then the mesh data can be computed as

\begin{verbatim}
1  \% bdStr
2  C0 = 'x.^2 + y.^2 > 3.8^2';  \% 1
3  bdStr = C0;
4  \% mesh info
5  Th = FeMesh2d(node,elem,bdStr);
\end{verbatim}

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

The coefficient \( \kappa \) can be written as

\begin{verbatim}
1 \% PDE data
2  kappa = @(p) 1 + 2*(p(:,1)<-1).*(p(:,1)>-2).*(p(:,2)<3).*(p(:,2)>-3);  \% p = [x,y]
\end{verbatim}

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

\begin{verbatim}
1 \% Bilinear form
2  Vh = 'P1'; quadOrder = 5;
3  Coef = kappa;
4  Test = {'v.grad'}; Trial = {'u.grad'};
5  kk = assem2d(Th,Coef,Test,Trial,Vh,quadOrder);
6
7 \% Linear form
8  ff = zeros(size(kk,1),1);
\end{verbatim}

The Dirichlet boundary conditions are imposed in the following way:

\begin{verbatim}
1 \% Dirichlet boundary conditions
2  on = [1,2];
3  gBc1 = @(p) 20*0*p(:,1);
4  gBc2 = @(p) 100*0*p(:,1);
5  uh = apply2d(on,Th,kk,ff,Vh,gBc1,gBc2);
\end{verbatim}

Here, \( gBc1 \) is for \( \partial C_0 \) or \texttt{on(1)}, and \( gBc2 \) is for \( \partial C_1 \) or \texttt{on(2)}.

In FreeFEM, the numerical solution can be outputted using the command \texttt{ofstream}:

\begin{verbatim}
1 ofstream file("sol_heatex.txt");
2 file<<uh[][]<<endl;
\end{verbatim}

The saved information is then loaded by

\begin{verbatim}
1 uff = solFreeFEM('sol_heatex.txt');
\end{verbatim}

The results are shown in Fig. 13, from which we observe that the \texttt{varFEM} solution is well matched with the one given by FreeFEM.
4.3 Airfoil

This is an example 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. 14. 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.

The programming is very simple, given by

```matlab
%% 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');
bdStr = 'x.^2 + y.^2 > 4.5^2'; % 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. 15, where the varFEM solution is well matched with the one given by FreeFEM.

![Fig. 15: Zoom around the NACA0012 airfoil showing the streamlines](image)

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 \((\delta u, \delta p)\) such that

\[
DF(\delta u, \delta p; u, p) = F(u, p),
\]

where \(u\) and \(p\) are the solutions given in the last step, and

\[
DF(\delta u, \delta p; u, p) = \int_\Omega ((\delta u \cdot \nabla) u) \cdot v + ((u \cdot \nabla) \delta u) \cdot v + \nu \nabla \delta u : \nabla v - q \nabla \cdot \delta u,
\]

\[
F(u, p) = \int_\Omega ((u \cdot \nabla) u) \cdot v + \nu \nabla u : \nabla v - p \nabla u : \nabla v - q \cdot u,
\]

where \((v, q)\) are the test functions.

The finite element spaces and the quadrature rule are

\[
Vh = \{ 'P2', 'P2', 'P1' \}; \% v = [v_1,v_2,q] \rightarrow [v_1,v_2,v_3]
quadOrder = 7;
vstr = \{ 'v1', 'v2', 'q' \}; ustr = \{ 'du1', 'du2', 'dp' \};
\]

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.,

\[
\int_\Omega ((\delta u \cdot \nabla) u) \cdot v = \int_\Omega u_{1,x} \cdot v_1 \delta u_1 + u_{1,y} \cdot v_1 \delta u_2 + u_{2,x} \cdot v_2 \delta u_1 + u_{2,y} \cdot v_2 \delta u_2. \tag{4.1}
\]
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 $u$ in the last step. Let us discuss the implementation of varFEM:

- The initial data are
  
  ```matlab
  1 % initial data
  2 u1 = @(p) double( (p(:,1).^2 + p(:,2).^2 > 2) );
  3 u2 = @(p) 0*p(:,1);
  4 p = @(p) 0*p(:,1);
  ```

  We must convert them into the dof vectors in view of the iteration.
  
  ```matlab
  1 uh1 = interp2d(u1, Th, Vh{1});
  2 uh2 = interp2d(u2, Th, Vh{2});
  3 ph = interp2d(p, Th, Vh{3});
  ```

- For numerical integration, it is preferable to provide the coefficient matrices for these coefficient functions.
  
  ```matlab
  1 u1c = interp2dMat(uh1, 'u1.val', Th, Vh{1}, quadOrder);
  2 u2c = interp2dMat(uh2, 'u2.val', Th, Vh{2}, quadOrder);
  3 pc = interp2dMat(ph, 'p.val', Th, Vh{3}, quadOrder);
  4 u1xc = interp2dMat(uh1, 'u1.dx', Th, Vh{1}, quadOrder);
  5 u1yc = interp2dMat(uh1, 'u1.dy', Th, Vh{1}, quadOrder);
  6 u2xc = interp2dMat(uh2, 'u2.dx', Th, Vh{2}, quadOrder);
  7 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
  
  ```matlab
  1 Coef = { u1xc, u1yc, u2xc, u2yc};
  2 Test = { 'v1.val', 'v1.val', 'v2.val', 'v2.val'};
  3 Trial = { 'du1.val', 'du2.val', 'du1.val', 'du2.val'};
  ```

  A complete correspondence of $DF(\delta u, \delta p; u, p)$ can be listed in the following:
  
  ```matlab
  1 Coef = { u1xc, u1yc, u2xc, u2yc, ... % term 1
  2 u1c, u2c, u1c, u2c, ... % term 2
  3 nu, nu, nu, nu, ... % term 3
  4 -1, -1, ... % term 4
  5 -1, -1, ... % term 5
  6 -eps ... % stabilization term
  7};
  8 Test = { 'v1.val', 'v1.val', 'v2.val', 'v2.val', ... % term 1
  9 'v1.val', 'v1.val', 'v2.val', 'v2.val', ... % term 2
  10 'v1.dx', 'v1.dy', 'v2.dx', 'v2.dy', ... % term 3
  11 'v1.dx', 'v2.dy', ... % term 4
  12 'q.val', 'q.val', ... % term 5
  13 'q.val' ... % stabilization term
  14};
  15 Trial = { 'du1.val', 'du2.val', 'du1.val', 'du2.val',... % term 1
  16 'du1.dx', 'du1.dy', 'du2.dx', 'du2.dy', ... % term 2
  17 'du1.dx', 'du1.dy', 'du2.dx', 'du2.dy', ... % term 3
  18 'dp.val', 'dp.val', ... % term 4
  19 'du1.dx', 'du2.dy', ... % term 5
  ```
The computation of the stiffness matrix reads

```matlab
[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.

The complete test script is available in `test_NSNewton.m`. The varFEM solutions and the FreeFEM solutions are shown in Fig. 16.

![Fig. 16: The pressure p of a flow](image)

### 4.5 Optimal control

#### 4.5.1 The gradient is not provided

This is an example 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 $\Omega$. The coefficient $\kappa$ 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);

% Dirichlet boundary condition
on = 1;
uh = apply2d(on,Th,kk,ff,Vh,gD);
end

The cost function is then given by

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

function zd = [2, 3, 4];
ud = PDEcon(zd,Th);

options.LargeScale = 'off';
options.HessUpdate = 'bfgs';
options.Display = 'iter';
z0 = [1,1,1];
zm = fminunc(@(z) J(z,ud,Th), z0, options)

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

| Iteration | Func-count | f(z)    | Step-size | First-order optimality |
|-----------|------------|---------|-----------|------------------------|
| 0         | 4          | 30.9874 |           | 77.2                   |
| 1         | 8          | 9.07648 | 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         | 2.13989 | 1         | 5.77                   |
Local minimum found.
Optimization completed because the size of the gradient is less than
the value of the optimality tolerance.

<stopping criteria details>

```
min =
```

2.0000  3.0000  4.0000

We can observe that the minimizer is found after 19 iterations, with the solutions displayed in
Fig. 17.

![Numerical solutions of the optimal control problem](image)

Fig. 17: 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:

```matlab
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

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

Here $err$ and $derr$ correspond $J$ 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.

```matlab
%% Find the minimizer
options = optimoptions('fminunc','Display','iter',
    'SpecifyObjectiveGradient',true); % gradient is provided
$z_0 = [1, 1, 1];$
$z_{\text{min}} = \text{fminunc}(g(z), \mathcal{J}(z, \text{ud}, \text{Th}), z_0, \text{options})$

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          | 1.06448    | 1         | 4.42       |
| 8         | 9          | 0.542864   | 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         | 0.063266   | 1         | 0.842      |
| 13        | 14         | 0.0242572  | 1         | 0.366      |
| 14        | 15         | 0.0070561  | 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         | 4.31651e-13| 1         | 3.86e-06   |

Local minimum found.

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

<stopping criteria details>

$z_{\text{min}} =$

| 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 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 several examples. Possible extensions of this
library that are of interest include three-dimensional problems and other types of finite elements.

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.