A learner's $\mathbb{P}_1-$FEM code

Francisco–Javier Sayas
University of Delaware

Last update: March 11, 2015

Contents

Foreword. Many of the ideas of this program/document are borrowed from the following technical report of the Technical University of Vienna:

> S. Funken, D. Praetorius, P. Wissgott. *Efficient implementation of adaptive P1-FEM in MATLAB*

The above document deals with adaptivity (which I won't) in the two dimensional setting. I'll try to explain things directly and you'll very easily see how to change the type of finite element, because most of the information related to the particular finite element is given in the data structure and in the local matrices on the reference element. The treatment of source terms and Neumann boundary conditions will be developed only for the lowest order elements.

1 The triangulation and its data structure

The simplest version of a tetrahedral mesh is a data structure that contains the following fields

```
T =
    coordinates: [2x137 double]
       elements: [3x218 double]
        neumann: [2x18 double]
      dirichlet: [2x38 double]
```

In this case, what we have is:

- 137 nodes on the grid, whose coordinates are given in the columns of `T.coordinates`
- 218 triangular elements, whose vertices are given in the columns of `T.elements`. Local numbering gives positive orientation of the triangle. This will have implications on the sign of a determinant below.
- 38 Dirichlet edges on the boundary. The numbering is done so that the interior domain is on the left.
- 18 Neumann edges on the boundary. The numbering is done as in the Dirichlet case and will have consequences on how the normal is computed.

All this information can be easily used to create easily many lists and quantities that are needed for Finite Element computations. A very easy one is the construction of the lists of all Dirichlet and Free nodes.

```
nC=size(T.coordinates,2);
Dir=unique(T.dirichlet);
Free=(1:nC)'; Free(Dir)=[];
```

Let us introduce some notation. Given an element K with vertices (x_1, y_1), (x_2, y_2) and (x_3, y_3), we consider the transformation

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_2 - x_1 & x_3 - x_1 \\ y_2 - y_1 & y_3 - y_1 \end{bmatrix} \begin{bmatrix} \xi \\ \eta \end{bmatrix} + \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

that maps the reference triangle

$$\widehat{K} := \{(\xi, \eta) \,:\, \xi, \eta \geq 0, \xi + \eta \leq 1\}$$

into K. Then

$$\mathrm{B}_K := \begin{bmatrix} x_2 - x_1 & x_3 - x_1 \\ y_2 - y_1 & y_3 - y_1 \end{bmatrix}, \qquad \mathrm{C}_K := (\det \mathrm{B}_K)\, \mathrm{B}_K^{-1}\, \mathrm{B}_K^{-\top} = \begin{bmatrix} c_{11} & c_{12} \\ c_{12} & c_{22} \end{bmatrix}.$$

Finally

$$\mathrm{G}_K := \mathrm{B}_K^{-\top} \begin{bmatrix} -1 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} g_{11} & g_{12} & g_{13} \\ g_{21} & g_{22} & g_{23} \end{bmatrix}.$$

Remark. The fact that the triangulation has been given with positive orientation for each element means that

$$|\det \mathrm{B}_K| = \det \mathrm{B}_K = (x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1).$$

The elements of C_K can be given with explicit formulas:

$$\begin{aligned} c_{11} &= \frac{1}{\det \mathrm{B}_K}\big((x_3-x_1)^2+(y_3-y_1)^2\big) \\ c_{22} &= \frac{1}{\det \mathrm{B}_K}\big((x_2-x_1)^2+(y_2-y_1)^2\big) \\ c_{12} &= -\frac{1}{\det \mathrm{B}_K}\big((x_2-x_1)(x_3-x_1)+(y_2-y_1)(y_3-y_1)\big). \end{aligned}$$

The elements of G_K can be computed explicitly too

$$\begin{aligned} \mathrm{G}_K &= \frac{1}{\det \mathrm{B}_K}\begin{bmatrix} (y_2-y_1)-(y_3-y_1) & y_3-y_1 & -(y_2-y_1) \\ (x_3-x_1)-(x_2-x_1) & -(x_3-x_1) & x_2-x_1 \end{bmatrix} \\ &= \frac{1}{\det \mathrm{B}_K}\begin{bmatrix} y_2-y_3 & y_3-y_1 & y_1-y_2 \\ x_3-x_2 & x_1-x_3 & x_2-x_1 \end{bmatrix}. \end{aligned}$$

The meaning of these two matrices will be shown when needed: C_K will be used in the computation of local stiffness matrices and G_K in the evaluation of gradients.

Local edge-numbering. If an element K is described by the triplet $[n_1^K, n_2^K, n_3^K]$ we will consider that its edges are given by the pairs

$$e_1^K := (n_1^K, n_2^K), \qquad e_2^K := (n_2^K, n_3^K), \qquad e_3^K := (n_3^K, n_1^K).$$

(In that precise order and with that orientation.)

2 How to create a mesh using the PDE Toolbox

I have modified a small script (given to me by my colleague Salim Meddahi, from the University of Oviedo, several years ago) to create the basic data structure from what PDE–Tool produces by simply clicking on the different elements. Here's what you have to do:

1. Open the MATLAB's PDE Toolbox by typing `pdetool`
2. Create the domain. Clicking on it you can modify coordinates.
3. Assign boundary conditions. It is immaterial what coefficients you assign. You just have to choose between Dirichlet and Neumann. (The default option is Dirichlet.)
4. You will have to remove all subdomain borders if your domain is composed by several subdomains. You can find this in the `Boundary` menu.
5. Create the grid.
6. Export the elements you will need, keeping the names of variables that MATLAB suggests:
   - In the `Boundary` menu, choose `Export Decomposed Geometry, Boundary Cond's,...`
   - In the `Mesh` menu, choose `Export Mesh,...`
7. Finally, go back to the workspace and run the script `basicFEMgrid`. You will obtain the four basic fields of triangulation data structure named `T`.

Here is the code and some explanations on how it works. You can perfectly ignore this part. It is technical and we will be using it as a black box.

```
% basicFEMgrid — Last modified: February 18, 2015

% This script assumes that you have created a mesh with the PDE Toolbox and
% exported the decomposed geometry and mesh. (See documentation for more
% instructions.)

T.coordinates=p;        % Nodes
T.elements=t(1:3,:);    % Connectivity

% Reorientation

j = find(e(6,:)==0);    % edges that have the exterior on the left
e([1 2],j)=e([2 1],j);

% Neumann edges

NeuList=find(b(2,e(5,:))==0);
T.neumann=e([1 2],NeuList);

% Dirichlet edges

DirList=find(b(2,e(5,:))≠0);
T.dirichlet=e([1 2],DirList);
```

How the code works. The key is understanding the matrices that are exported from the PDE Toolbox. The sides of the polygon that we have created are numbered. For simplicity we will refer to them as boundary subdomains. For easy reference

$$N_{\text{nodes}}, \qquad N_{\text{elt}} \qquad \text{and} \qquad N_{\text{bdedg}}$$

will respectively denote the number of vertices (nodes), triangles (elements) and edges on the boundary of the triangulation.

- `p` has two rows and N_{nodes} columns. We only need to copy it in `T.coordinates`.
- `t` has four rows and N_{elt} columns. The fourth row contains the index of the subdomain. We will not be using it, so we ignore this row and copy the rest in `T.elements`.
- `b` has 10 rows and one column for each boundary subdomain. In the second row there is a number 1 if the side (subdomain) is Dirichlet and a number 0 if it is Neumann.
- `e` has 7 rows and as N_{bdedg} columns. The two first rows contain the indices of the nodes of the edge. The fifth row contains the index corresponding to the boundary subdomains where this edges lies.
- The sixth row of `e` contains the index of the plane subdomain that lies on the left. If this number is zero, it means that the exterior domain is on the left and the edge has to be reoriented for what we need.[1]

The command

```
e(5,:)
```

gives a list of the boundary subdomain for each of the boundary edges. Therefore

```
b(2,e(5,:))
```

[1]The PDE–Toolbox default option gives each closed curve with negative (clockwise) orientation. If the closed curve surrounds a hole in the obstacle, this is the right orientation. Otherwise, it is not.

is a list of ones and zeros: there is a one for edges on the Neumann boundary and a zero for edges on the Dirichlet boundary. Finally,

```
      NeuList=find(b(2,e(5,:))==0);
      DirList=find(b(2,e(5,:))≠0);
```

are the lists of Neumann and Dirichlet edges, constructed by looking at what positions of the vector `b(2,e(5,:))` there is a zero (Neumann) or a non–zero (Dirichlet). Once we know these lists, we can drop all other rows of `e` and separate what is left in the two lists `T.dirichlet` and `T.neumann`.

3 A reordered mesh

In some cases (mesh refinements using the newest-vertex-bisection method) it is convenient to have triangles numbered so that the first edge is the longest one. This is done by `reorderFEMgrid.m`. Recall that the first edge of a triangle is the one running from the first node to the second one.

```
function T=reorderFEMgrid(T)

% T=reorderFEMgrid(T)
%
% Input:
%     T   : basic mesh data structure with fields
%           .coordinates, .elements, .dirichlet, .neumann
% Output:
%     T   : same triangulation
%           REORDERED so that longest edge is the first one
%
% Last modified: February 18, 2015

firstnode =T.coordinates(:,T.elements(1,:));
secondnode=T.coordinates(:,T.elements(2,:));
thirdnode =T.coordinates(:,T.elements(3,:));
lengths   =zeros(size(T.elements));
lengths(1,:)=sum((secondnode-firstnode).^2,1);
lengths(2,:)=sum((thirdnode-secondnode).^2,1);
lengths(3,:)=sum((firstnode-thirdnode).^2,1);
lengths     =sqrt(lengths);
[~,where]=max(lengths,[],1);
itssecond = find(where==2);
itsthird  = find(where==3);
T.elements(:,itssecond)=T.elements([2 3 1],itssecond);
T.elements(:,itsthird)=T.elements([3 1 2],itsthird);

return
```

4 The expanded data structure

The triangulation can be enriched with some additional pre-computed quantities. It becomes

```
T =
    coordinates: [2x137 double]
       elements: [3x218 double]
        neumann: [2x18 double]
      dirichlet: [2x38 double]
          baryc: [2x218 double]
         normal: [2x18 double]
       midptNeu: [2x18 double]
```

```
          detB: [1x218 double]
           c11: [1x218 double]
           c22: [1x218 double]
           c12: [1x218 double]
           g11: [1x218 double]
           g12: [1x218 double]
           g13: [1x218 double]
           g21: [1x218 double]
           g22: [1x218 double]
           g23: [1x218 double]
```

These are the new fields:

- `T.baryc` contains the barycenters of all the triangles. Row number i contains the x and y coordinates of the barycenter of triangle number i.
- `T.normal` contains the non-unitary normal vectors of the Neumann edges, pointing outwards and with the length of the edge.
- `T.midptNeu` contains the midpoints of the Neumann edges.
- The coefficients of the matrices C_K and G_K (see Section 1), as well as the determinants $\det \mathrm{B}_K$ are stored in row vectors with N_{elt} elements.

The formulas of Section 1 explain what is done in this function.

```
function T=expandFEMgrid(T)

% T=expandFEMgrid(T)
%
% Input:
%     T  : a triangulation with the original four basic fields
%          T.coordinates, T.elements, T.dirichlet, T.neumann
% Output:
%     T  : an expanded version of the triangulation with some new fields
%
% Last modified: February 18, 2015

T.baryc=(T.coordinates(:,T.elements(1,:))...
         +T.coordinates(:,T.elements(2,:))...
         +T.coordinates(:,T.elements(3,:)))/3;
if length(T.neumann)>0
    v=T.coordinates(:,T.neumann(2,:))...
        -T.coordinates(:,T.neumann(1,:));
    T.normal=[v(2,:);-v(1,:)];           % non-normalized normals
    T.midptNeu=0.5*T.coordinates(:,T.neumann(1,:))...
               +0.5*T.coordinates(:,T.neumann(2,:));
end
x12=T.coordinates(1,T.elements(2,:))-T.coordinates(1,T.elements(1,:));  % x(2)-x(1)
y12=T.coordinates(2,T.elements(2,:))-T.coordinates(2,T.elements(1,:));  % y(2)-y(1)
x13=T.coordinates(1,T.elements(3,:))-T.coordinates(1,T.elements(1,:));  % x(3)-x(1)
y13=T.coordinates(2,T.elements(3,:))-T.coordinates(2,T.elements(1,:));  % y(3)-y(1)
T.detB=x12.*y13-x13.*y12;

% coefficients for stiffness matrix

T.c11=(x13.^2+y13.^2)./T.detB;
T.c22=(x12.^2+y12.^2)./T.detB;
T.c12=-(x12.*x13+y12.*y13)./T.detB;

% coefficients for gradients

T.g11=(y12-y13)./T.detB;
T.g12=y13./T.detB;
```

```
T.g13=-y12./T.detB;
T.g21=(x13-x12)./T.detB;
T.g22=-x13./T.detB;
T.g23=x12./T.detB;

return
```

5 Stiffness and mass matrices

The stiffness matrix for the $\mathbb{P}_1$ element associated to the triangulation `T` above is the matrix

$$s_{ij} := \int_\Omega \nabla\varphi_j \cdot \nabla\varphi_i,$$

where $\{\varphi_i\}$ is the nodal basis of the space of continuous finite element functions. The mass matrix is the following matrix

$$m_{ij} := \int_\Omega \varphi_j\, \varphi_i.$$

As in previous sections, the number of nodes of the triangulation (equaling the dimension of the Finite Element space) N_{nodes} and the number of elements N_{elt}.

Change of variables to the reference element. With the notation for the map $F_K : \widehat{K} \to K$ that we have introduced in Section 1, it can be proved that

$$\int_K \nabla u \cdot \nabla v = c_{11}^K \int_{\widehat{K}} \partial_\xi u\, \partial_\xi v + c_{12}^K \int_{\widehat{K}} (\partial_\xi u\, \partial_\eta v + \partial_\eta u\, \partial_\xi v) + c_{22}^K \int_{\widehat{K}} \partial_\eta u\, \partial_\eta v,$$

where $\nabla(u \circ F_K^{-1}) = (\partial_\xi u\,, \partial_\eta u)^\top$. Also

$$\int_K u\, v = \det \mathrm{B}_K \int_{\widehat{K}} (u \circ F_K^{-1})\, (v \circ F_K^{-1}).$$

The $\mathbb{P}_1$ basis functions on the reference element are

$$N_1 = 1 - \xi - \eta, \qquad N_2 = \xi, \qquad N_3 = \eta.$$

In the physical element K, the basis functions are

$$N_\alpha^K := N_\alpha \circ F_K^{-1}.$$

The chain rule can be used to prove that

$$\nabla N_\alpha^K = \mathrm{B}_K^{-\top} \nabla N_\alpha,$$

with the assumption that gradients are column vectors and noticing that all the above gradients are constant vectors.

There are four relevant matrices on the reference element:

$$\mathrm{K}_{11} = \left[\int_{\widehat{K}} \partial_\xi N_\alpha \, \partial_\xi N_\beta\right]_{\alpha,\beta=1}^{3} = \frac{1}{2}\begin{bmatrix} 1 & -1 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$\mathrm{K}_{22} = \left[\int_{\widehat{K}} \partial_\eta N_\alpha \, \partial_\eta N_\beta\right]_{\alpha,\beta=1}^{3} = \frac{1}{2}\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$$

$$\mathrm{K}_{12} = \left[\int_{\widehat{K}} \partial_\eta N_\alpha \, \partial_\xi N_\beta\right]_{\alpha,\beta=1}^{3} = \frac{1}{2}\begin{bmatrix} 1 & 0 & -1 \\ -1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}^{\top} = \mathrm{K}_{21}^{\top}$$

$$\mathrm{M} = \left[\int_{\widehat{K}} N_\alpha \, N_\beta\right]_{\alpha,\beta=1}^{3} = \frac{1}{24}\begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix}$$

A simple computation shows then that

$$\mathrm{S}_K := \left[\int_K \nabla N_\beta^K \cdot \nabla N_\alpha^K\right]_{\alpha,\beta=1}^{3} = c_{11}^K \mathrm{K}_{11} + c_{12}^K(\mathrm{K}_{12} + \mathrm{K}_{21}) + c_{22}^K \mathrm{K}_{22}$$

and

$$\mathrm{M}_K := \left[\int_K N_\beta^K \, N_\alpha^K\right]_{\alpha,\beta=1}^{3} = (\det \mathrm{B}_K)\mathrm{M}.$$

Computation of stiffness and mass matrices. The set of all local and mass matrices, displayed in a $3 \times N_{\mathrm{elt}}$ matrix, can be computed using Kronecker products (`kron` in MATLAB). For instance

$$\left[\begin{array}{cccc} \det \mathrm{B}_1 & \det \mathrm{B}_2 & \dots & \det \mathrm{B}_{N_{\mathrm{elt}}} \end{array}\right] \otimes \mathrm{M} = \left[\begin{array}{c|c|c|c} \mathrm{M}_1 & \mathrm{M}_2 & \cdots & \mathrm{M}_{N_{\mathrm{elt}}} \end{array}\right].$$

The collection of the local stiffness matrices can be done in a similar way. We will momentarily reshape them (although this is not necessary) to three-dimensional $3 \times 3 \times N_{\mathrm{elt}}$ arrays

$$\left[\begin{array}{c} M_1 \end{array}\right]\left[\begin{array}{c} M_2 \end{array}\right]\left[\begin{array}{c} \dots \end{array}\right]\left[\begin{array}{c} M_{N_{\mathrm{elt}}} \end{array}\right] \qquad \left[\begin{array}{c} S_1 \end{array}\right]\left[\begin{array}{c} S_2 \end{array}\right]\left[\begin{array}{c} \dots \end{array}\right]\left[\begin{array}{c} S_{N_{\mathrm{elt}}} \end{array}\right].$$

The assembly process can be realized almost automatically by setting up some matrices that show the elements and their locations in the global matrices and then using the `sparse` matrix builder of MATLAB. The underlying idea is very simple. Given an element K with vertices $[n_1^K, n_2^K, n_3^K]$, the following three matrices

$$\mathrm{S}_K = \begin{bmatrix} s_{11}^K & s_{12}^K & s_{13}^K \\ s_{21}^K & s_{22}^K & s_{23}^K \\ s_{31}^K & s_{32}^K & s_{33}^K \end{bmatrix} \qquad \mathrm{R}_K = \begin{bmatrix} n_1^K & n_1^K & n_1^K \\ n_2^K & n_2^K & n_2^K \\ n_3^K & n_3^K & n_3^K \end{bmatrix} \qquad \mathrm{C}_K = \begin{bmatrix} n_1^K & n_2^K & n_3^K \\ n_1^K & n_2^K & n_3^K \\ n_1^K & n_2^K & n_3^K \end{bmatrix}$$

contain the necessary information to place each element of S_K in the correct location. For instance, the element s_{23}^K will be place in row number n_2^K (this is the number we find in R_{23}^K and column number n_3^K (which we find looking at C_K). Note that the element blocks C_K are just the transposes of the blocks R_K. If we are able to construct the three dimensional $3 \times 3 \times N_{\mathrm{elt}}$ arrays

$$\left[\begin{array}{c} R_1 \end{array}\right]\left[\begin{array}{c} R_2 \end{array}\right]\left[\begin{array}{c} \dots \end{array}\right]\left[\begin{array}{c} R_{N_{\mathrm{elt}}} \end{array}\right] \qquad \left[\begin{array}{c} C_1 \end{array}\right]\left[\begin{array}{c} C_2 \end{array}\right]\left[\begin{array}{c} \dots \end{array}\right]\left[\begin{array}{c} C_{N_{\mathrm{elt}}} \end{array}\right]$$

then the FEM-assembly process for the mass and stiffness matrices can be done using the command `sparse` with these two matrices as first arguments and the collection of local mass and stiffness matrices

$$\left[\ M_1\ \right]\left[\ M_2\ \right]\left[\ \ldots\ \right]\left[\ M_{N_{\text{elt}}}\ \right] \qquad \left[\ S_1\ \right]\left[\ S_2\ \right]\left[\ \ldots\ \right]\left[\ S_{N_{\text{elt}}}\ \right].$$

as third argument. The construction of the index matrices is shown with an example:

```
>> T.elements=[1 2 4;...
               4 2 5;...
               3 5 2;...
               3 6 5]';
>> nE=size(T.elements,2);     % = 4
>> R=repmat((1:3)',1,3)
R =
     1     1     1
     2     2     2
     3     3     3
>> R=reshape(T.elements(R,:),3,3,nE)
R(:,:,1) =
     1     1     1
     2     2     2
     4     4     4
R(:,:,2) =
     4     4     4
     2     2     2
     5     5     5
R(:,:,3) =
     3     3     3
     5     5     5
     2     2     2
R(:,:,4) =
     3     3     3
     6     6     6
     5     5     5
>> C=permute(R,[2 1 3])
C(:,:,1) =
     1     2     4
     1     2     4
     1     2     4
C(:,:,2) =
     4     2     5
     4     2     5
     4     2     5
C(:,:,3) =
     3     5     2
     3     5     2
     3     5     2
C(:,:,4) =
     3     6     5
     3     6     5
     3     6     5
```

The full code is given below:

```
function [S,M]=matricesFEM(T)

% [S,M]=matricesFEM(T)
%
% Input:
%     T    : expanded triangulation data structure
% Output:
%     S    : stiffness matrix (sparse)
%     M    : mass matrix (sparse)
%
```

```
% Last modified: February 20, 2015

nE=size(T.elements,2);

% Matrices in the reference element

K11=0.5*[1 -1 0;-1 1 0;0 0 0];
K22=0.5*[1 0 -1;0 0 0;-1 0 1];
K12=0.5*[1 0 -1;-1 0 1;0 0 0]';
M=1/24*[2 1 1;1 2 1;1 1 2];

% Assembly

R=repmat((1:3)',1,3);
R=reshape(T.elements(R,:),3,3,nE);
C=permute(R,[2 1 3]);
S=kron(T.c11,K11)+kron(T.c22,K22)+kron(T.c12,K12+K12');
S=sparse(R(:),C(:),S(:));
M=kron(T.detB,M);
M=sparse(R(:),C(:),M(:));

return
```

6 Load and traction

Given a function $f : \Omega \to \mathbb{R}$, the load vector is

$$f_i := \int_\Omega f\varphi_i.$$

Finally, given $\mathbf{g} : \Gamma_N \to \mathbb{R}^2$, the (normal) traction vector is

$$g_i := \int_{\Gamma_N} (\mathbf{g} \cdot \mathbf{n})\, \varphi_i,$$

where $\mathbf{n}$ is the exterior normal vector and $\{\varphi_i\}$ is the nodal basis for the $\mathbb{P}_1$ FE space. Of the vector $\mathbf{g}$, we only use the normal component for the traction vector. Instead of the vector valued function $\mathbf{g}$ we could use the scalar valued function $g = \mathbf{g} \cdot \mathbf{n}$. The changes in our program are minimal for that. Equivalently, if we are given g, one of the many choices to recreate a vector field whose normal component is g consists of defining $\mathbf{g} = g\,\mathbf{n}$.

The load vector. We will program an approximated version of the load vector, where f is substituted by a piecewise constant function, whose value on each triangle K is the value of f at the barycenter of K. The local load vector is then

$$f_\alpha^K := \int_K f\, N_\alpha^K \approx f(\mathbf{b}_K) \int_K N_\alpha^K = \frac{\det \mathrm{B}_K}{6} f(\mathbf{b}_K) =: f^K, \qquad \alpha = 1, 2, 3,$$

where $\mathbf{b}_K$ is the barycenter of K. The three components of this approximation are equal. If the element K has nodes $[n_1^K, n_2^K, n_3^K]$ the value that we have just computed has to be added to the positions n_1^K, n_2^K, n_3^K of the global load vector. Here is a simple way of doing the assembly of the load vector for a vectorized function of two variables `f`.

```
nC=size(T.coordinates,2);
f=(f(T.baryc(1,:),T.baryc(2,:)).*T.detB)/6;
f=repmat(f,3,1);
ld=accumarray(T.elements(:),f(:),[nC,1]);
```

Note that we have created a row vector $1 \times N_{\text{nodes}}$ with the values f^K. We pile these three copies on top of each other to create the $3 \times N_{\text{nodes}}$ matrix with the values f_α^K. Finally,

The traction vector. For the traction vector we apply a similar strategy of numerical approximation. In this case, we count locally using the Neumann boundary edges. We need two functions of one variable (the variable will be the parameter in the line integral)

$$\psi_1 = 1 - t, \qquad \psi_2 = t$$

and the parametrization of the edge e with vertices (x_1, y_1) and (x_2, y_2):

$$[0,1] \ni t \longmapsto \phi_e(t) = (1-t) \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} + t \begin{bmatrix} x_2 \\ y_2 \end{bmatrix}.$$

If e is a boundary edge, we need to compute the integrals

$$g_\alpha^e := \int_e (\mathbf{g} \cdot \mathbf{n}_e)(\psi_\alpha \circ \phi_e^{-1}) \approx |e|\, \mathbf{g}(\mathbf{m}_e) \cdot \mathbf{n}_e \int_0^1 \psi_\alpha(t) \mathrm{d}t = \tfrac{1}{2}|e|\, \mathbf{g}(\mathbf{m}_e) \cdot \mathbf{n}_e,$$

where $|e|$ is the length of e, $\mathbf{m}_e$ is the midpoint of the edge e and $\mathbf{n}_e$ is the vector in the normal direction with unit length. Note that the normal vectors that we have precomputed are $|e|\, \mathbf{n}_e$. The assembly process is done as in the case of the load vector, although we now use the adjacency information contained in `T.neumann`

```
g=0.5*sum(g(T.midptNeu(1,:),T.midptNeu(2,:)).*T.normal,1);
g=repmat(g,2,1);
trc=accumarray(T.neumann(:),g(:),[nC,1]);
```

For this to work, the vectorized function `g` has to return a $2 \times N$ matrix when the input is composed of two row vectors with N components.

```
function [ld,trc,Dir,Free]=vectorsFEM(T,f,g)

% [ld,trc,Dir,Free]=vectorsFEM(T,f,g)
%
% Input:
%     T    : expanded triangulation data structure
%     f    : vectorized function of two variables
%     g    : vectorized function of two variables
%            with values in R^2
% Output:
%     ld   : load vector (associated to f)
%     trc  : traction vector (associated to g)
%     Dir  : list of Dirichlet nodes
%     Free : list of free (independent, non-Dirichlet) nodes
%
% Last modified: February 20, 2015

nC=size(T.coordinates,2);

% Load vector

f=(f(T.baryc(1,:),T.baryc(2,:)).*T.detB)/6;
f=repmat(f,3,1);
ld=accumarray(T.elements(:),f(:),[nC,1]);

% Traction vector

if length(T.neumann)>0
    g=0.5*sum(g(T.midptNeu(1,:),T.midptNeu(2,:)).*T.normal,1);
    g=repmat(g,2,1);
    trc=accumarray(T.neumann(:),g(:),[nC,1]);
else
    trc=zeros(nC,1);
```

```
end

% Dir and Free nodes

Dir=unique(T.dirichlet);
Free=(1:nC)'; Free(Dir)=[];

return
```

7 Evaluation of the gradient

Let $u_h = \sum_j u_j \varphi_j$. If we want to evaluate ∇u_h in K (this is a constant vector), we can do as follows:

$$\begin{aligned}
\nabla u_h &= \sum_{\alpha=1}^{3} u_{n_\alpha} \nabla N_\alpha^K = \sum_{\alpha=1}^{3} u_{n_\alpha} \mathrm{B}_K^{-\top} \nabla N_\alpha \\
&= \mathrm{B}_K^{-\top} \left[\begin{array}{c|c|c} \nabla N_1 & \nabla N_2 & \nabla N_3 \end{array} \right] \begin{bmatrix} u_{n_1} \\ u_{n_2} \\ u_{n_3} \end{bmatrix} = \mathrm{B}_K^{-\top} \begin{bmatrix} -1 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_{n_1} \\ u_{n_2} \\ u_{n_3} \end{bmatrix} \\
&= \mathrm{G}_K \begin{bmatrix} u_{n_1} \\ u_{n_2} \\ u_{n_3} \end{bmatrix}.
\end{aligned}$$

Here we have assumed that $\{n_1, n_2, n_3\}$ are the global numbers of the three vertices of K. This leads to the following simple piece of code (we will use it in the next section) to simultaneously evaluate the gradient in all the triangles.

```
T=expandFEMgrid(T);
ux = sum(uh(T.elements).*[T.g11; T.g12; T.g13]);
uy = sum(uh(T.elements).*[T.g21; T.g22; T.g23]);
```

Partial differentiation of a general u_h can be easily accomplished defining a discrete differential operator through an anonymous function. Once the following functions have been defined, computing a partial derivative will be as simple as writing `dx(uh)`.

```
dx = @(uh) sum(uh(T.elements).*[T.g11; T.g12; T.g13]);
dy = @(uh) sum(uh(T.elements).*[T.g21; T.g22; T.g23]);
```

Another way of computing gradients is through the construction of the sparse $N_{\mathrm{elt}} \times N_{\mathrm{nodes}}$ matrices G^x and G^y

$$\mathrm{G}^\star_{K,i} = \partial_\star \varphi_i|_K \qquad \star \in \{x, y\},$$

so that computation of a partial derivative is just multiplication by the coprresponding matrix.

```
M=size(T.elements,2);     % number of triangles
N=size(T.coordinates,2); % number of nodes
Gx =sparse(1:M,T.elements(1,:),T.g11,M,N)...
    +sparse(1:M,T.elements(2,:),T.g12,M,N)...
    +sparse(1:M,T.elements(3,:),T.g13,M,N);
Gy =sparse(1:M,T.elements(1,:),T.g21,M,N)...
    +sparse(1:M,T.elements(2,:),T.g22,M,N)...
    +sparse(1:M,T.elements(3,:),T.g23,M,N);
```

8 L^2 and H^1 errors

The aim of the following piece of code is the approximation of

$$\left(\int_\Omega |u_h - u|^2\right)^{1/2} \qquad \text{and} \qquad \left(\int_\Omega |\nabla u_h - \nabla u|^2\right)^{1/2}$$

where u_h is a discrete function on the triangulation and both u and $\nabla u = (u_x, u_y)^\top$ can be computed by means of vectorized functions.

A basic quadrature rule on $\widehat{K}$. We assume that we have a formula to approximate integrals on $\widehat{K}$:

$$\int_{\widehat{K}} f \approx \sum_{\ell=1}^{N_{\text{quad}}} \omega_\ell \, f(\xi_\ell, \eta_\ell).$$

With this formula, we can approximate

$$\int_K u = \det \mathrm{B}_K \int_{\widehat{K}} u \circ F_K \approx \det \mathrm{B}_K \sum_\ell \omega_\ell \, u(\underbrace{F_K(\xi_\ell, \eta_\ell)}_{=:(x_\ell^K, y_\ell^K)}).$$

We therefore have to address two relatively simple problems:

- The computation of (x_ℓ^K, y_ℓ^K).
- The evaluation of u_h in these points.

Computation of the quadrature points and interpolation. The fastest way to express the formula in order to apply linear transformations is to consider the barycentric coordinates:

$$(\lambda_\ell^1, \lambda_\ell^2, \lambda_\ell^3) = (1 - \xi_\ell - \eta_\ell, \xi_\ell, \eta_\ell).$$

Then, if the triangle K has nodes $\{n_1, n_2, n_3\}$, we compute

$$x_\ell^K = \lambda_\ell^1 x_{n_1} + \lambda_\ell^2 x_{n_2} + \lambda_\ell^3 x_{n_3} = \begin{bmatrix} \lambda_\ell^1 & \lambda_\ell^2 & \lambda_\ell^3 \end{bmatrix} \begin{bmatrix} x_{n_1} \\ x_{n_2} \\ x_{n_3} \end{bmatrix}$$

It is then simple to see how if we create an $N_{\text{quad}} \times 3$ matrix

$$\Lambda := \begin{bmatrix} \lambda_1^1 & \lambda_1^2 & \lambda_1^3 \\ \hline \lambda_2^1 & \lambda_2^2 & \lambda_2^3 \\ \hline \vdots & \vdots & \vdots \\ \hline \lambda_{N_{\text{quad}}}^1 & \lambda_{N_{\text{quad}}}^2 & \lambda_{N_{\text{quad}}}^3 \end{bmatrix}$$

with the barycentric coordinates of all quadrature points, and a $3 \times N_{\text{elt}}$ matrix X with the x-coordinates of all the nodes (vertices) of all the elements, then

$$\Lambda X$$

is an $N_{\text{quad}} \times N_{\text{elt}}$ matrix with the x-coordinates of all the quadrature points of all the elements. A similar computation can be used to create the $N_{\text{quad}} \times N_{\text{elt}}$ matrix Y with all the y-coordinates of all the quadrature points of all the elements.

The evaluation of u_h at the quadrature points is just linear interpolation of the values of u_h at the vertices. Therefore

$$u_h(x_\ell^K, y_\ell^K) = u_{n_1}\lambda_\ell^1 + u_{n_2}\lambda_\ell^2 + u_{n_3}\lambda_\ell^3 = \begin{bmatrix} \lambda_\ell^1 & \lambda_\ell^2 & \lambda_\ell^3 \end{bmatrix} \begin{bmatrix} u_{n_1} \\ u_{n_2} \\ u_{n_3} \end{bmatrix}$$

The computation of the values of u_h in all quadrature points of all the elements can be done exactly as the computation of the coordinates of these points.

Computation of the L^2-error. We just need to compute

$$\sum_K \det \mathrm{B}_K \Big(\sum_\ell \omega_\ell \, |u_h(x_\ell^K, y_\ell^K) - u(x_\ell^K, y_\ell^K)|^2 \Big).$$

This can be done in three steps that we will group in the program:

(a) Compute the errors in each of the quadrature points

$$e_\ell^K := |u_h(x_\ell^K, y_\ell^K) - u(x_\ell^K, y_\ell^K)|^2,$$

storing the result in an $N_{\text{quad}} \times N_{\text{elt}}$ matrix.

(b) Compute the errors on each of the elements (still not corrected by the Jacobians)

$$e_K := \begin{bmatrix} \omega_1 & \dots & \omega_{N_{\text{quad}}} \end{bmatrix} \begin{bmatrix} e_1^K \\ \vdots \\ e_{N_{\text{quad}}}^K \end{bmatrix}$$

(c) Add the contribution of each of the elements, correcting with the Jacobian

$$\sum_K \det \mathrm{B}_K \, e_K.$$

Computation of the H^1-error. Apart from having to consider separately the partial derivatives

$$\int_\Omega |\partial_x u_h - \partial_x u|^2 + \int_\Omega |\partial_y u_h - \partial_y u|^2,$$

the only novelty arises from the fact that $(\partial_x u_h, \partial_y u_h)$ is constant on K (its evaluation is shown in the previous section). If we do not want to repeat the vector of values of the partial derivatives of u_h at each of the quadrature points, we can use `bsxfun` for the repeated differences. Here's a toy example of what we end up doing to subtract the values $\partial_x u_h - \partial_x u(x_\ell^K, y_\ell^K)$.

```
>> A=[1 2 3 4]'
A =
     1
     2
     3
     4
>> B=[2 3 4 1;4 3 1 2;1 2 3 4]'
B =
     2     4     1
     3     3     2
     4     1     3
     1     2     4
>> bsxfun(@minus,A,B)
ans =
    -1    -3     0
    -1    -1     0
    -1     2     0
     3     2     0
```

The final code uses a 7−point quadrature rule, that is invariant by affine transformations and that uses two points on each median of the triangle.

```
function [l2error,h1error]=computeFEMerror(T,uh,u,ux,uy)

% [l2error,h1error]=computeFEMerror(T,uh,u,ux,uy)
%
% Input:
%      T    : basic triangulation
%      uh   : vector of nodal values of u_h
%      u    : function
%      ux   : partial_x u
%      uy   : partial_y u
% Output:
%      l2error : \| u-u_h\|_{L^2}
%      h1error : \|\grad u -\grad u_h\|_{L^2}
%
% Last modified: February 20, 2015

% Seven point quadrature formula with order 5

nodes = [0.333333333333333 0.333333333333333 0.333333333333333; ...
       0.059715871789770 0.470142064105115 0.470142064105115; ...
       0.470142064105115 0.059715871789770 0.470142064105115; ...
       0.470142064105115 0.470142064105115 0.059715871789770; ...
       0.797426985353087 0.101286507323456 0.101286507323456; ...
       0.101286507323456 0.797426985353087 0.101286507323456; ...
       0.101286507323456 0.101286507323456 0.797426985353087];
weights = 0.5*[0.225000000000000 ...
               0.132394152788506 ...
               0.132394152788506 ...
               0.132394152788506 ...
               0.125939180544827 ...
               0.125939180544827 ...
               0.125939180544827];

x=T.coordinates(1,:); x=nodes*x(T.elements);      % x coordinates of quadrature points
y=T.coordinates(2,:); y=nodes*y(T.elements);      % y coordinates of quadrature points
T=expandFEMgrid(T);

% L2 error

uint=nodes*uh(T.elements);    % Value of u_h at all the quadrature points
u   =u(x,y);                  % Value of u at all the quadrature points
l2error=sqrt(dot( T.detB, weights*(uint-u).^2 ));

% H1 error

uhx = sum(uh(T.elements).*[T.g11; T.g12; T.g13]);
uhy = sum(uh(T.elements).*[T.g21; T.g22; T.g23]);
ux  = ux(x,y);
uy  = uy(x,y);
error=bsxfun(@minus,uhx,ux).^2+bsxfun(@minus,uhy,uy).^2;  % Errors in quad points
h1error=sqrt(dot( T.detB, weights*error ));

return
```

9 A simple PDE solver

The following program (given as a function, although it could easily be given as a script) solves the problem

$$-\Delta u + cu = f \qquad \text{in } \Omega,$$

with boundary conditions

$$u = u_D \quad \text{on } \Gamma_D, \qquad \partial_\nu u = \mathbf{u}_N \cdot \mathbf{n} \quad \text{on } \Gamma_N,$$

where $c \geq 0$ is a constant.

```
function uh=P1FEMdiffusion(T,c,f,uD,uN)

% function u=P1FEMdiffusion(T,c,f,uD,uN)
%
% -\Delta u + c u = g     in Omega
%               u = uD    on Gamma_D
%    \partial_n u = uN.n  on Gamma_N
% Input:
%     T     : basic triangulation
%     c     : constant parameter
%     f     : vectorized function of two variables (source)
%     uD    : vectorized function of two variables (Dirichlet B.C.)
%     uN    : vectorized function of two variables
%             with two components   (uN.n = Neumann B.C.)
% Output:
%     uh    : P1-FEM approximation
%
% Last modified: February 25, 2015

T=expandFEMgrid(T);
nC=size(T.coordinates,2);
uh=zeros(nC,1);

[S,M]=matricesFEM(T);
[ld,trc,Dir,Free]=vectorsFEM(T,f,uN);

matrix  = S+c*M;
uh(Dir) = uD(T.coordinates(1,Dir),T.coordinates(2,Dir));
rhs     = ld+trc-matrix(:,Dir)*uh(Dir);
uh(Free)= matrix(Free,Free)\rhs(Free);

return
```

10 Edge numbering

The goal of the function `edgesFEMgrid.m` is the production of two new fields for the triangulation `T`:

- A field `T.edges` that stores a $3 \times N_{\text{edge}}$ matrix. The first two rows contain the first and second vertex of each edge. Edges are oriented by following the order given in this list. We assume that boundary edges are always positively oriented, meaning that when going from the first to the second we leave the domain to the left. The third row contains the flag:

  0 when the edge is interior,

  1 when the edge is Dirichlet,

  2 when the edge is Neumann.

  The function below numbers first all interior edges, then all Dirichlet edges, and finally all Neumann edges. This is however not the only way to do it and we will not assume that this is the case whenever we use the function.

- A field `T.edgebyelt` stores a $3 \times N_{\text{elt}}$ matrix with the edge numbers for the elements. We organize the information in the following way. Assume that an element K is ordered (positively) as

$$\begin{bmatrix} n_1^K & n_2^K & n_3^K \end{bmatrix}$$

We next look for the edges

$$e_1^K = [n_1^K, n_2^K], \qquad e_2^K = [n_2^K, n_2^K], \qquad e_3^K = [n_3^K, n_1^K]$$

in the global list of edges. If $e_1^K = e_j$ ($j \in \{1, \ldots, N_{\text{edge}}\}$), we store the number j in the first row and K-th column. If $[n_2^K, n_1^K] = e_j$ (that is, we find the edge reversed in the global list), we store the number $j - 1$ instead. This is done for the second and third edge as well. Therefore the K-th column of `T.edgebyelt` contains the global reference of the local edges (from first node to second, second to third, and third to first) and the orientation as the sign of the entry.

This is the structure of this function. The way it is programmed, it uses several sparse matrices to keep connectivity information.

1. Create a sparse matrix `edges` with the pairs (n_1^K, n_2^K), (n_2^K, n_3^K), (n_3^K, n_1^K), storing the number one in each entry. Since all elements are positively oriented, if the pairs (i, j) and (j, i) are marked in the matrix, it is beacuse the edge is interior. Boundary edges do not appear in symmetric locations of the matrix.

2. We then delete the boundary edges and keep only the upper triangular part of the remaining matrices. At this moment we can count the number of interior edges: $N_{\text{int}-\text{edg}}$. We already knew the number of Dirichlet and Neumann edges.

3. The list `T.edges` is composed of the pairs (i, j) such that `edges`$(i, j) = 1$, followed by `T.dirichlet` and `T.neumann`.

4. At the same time that we create the list `T.edges` we modify the matrix in the following way:

   - In the locations where `edges` is non-zero, we place the numbers $\{1, \ldots, N_{\text{int}-\text{edge}}\}$ in the same order that we placed these pairs in the list of edges. We the write `edges=edges-edges'` to place negative values in the locations of reversed edges.
   - We place the numbers $\{N_{\text{int}-\text{edge}} + 1, N_{\text{dir}} + N_{\text{int}-\text{edge}}\}$ on the locations of Dirichlet edges.
   - We place the numbers $\{N_{\text{int}-\text{edge}} + N_{\text{dir}} + 1, N_{\text{int}-\text{edge}} + N_{\text{dir}} + N_{\text{neu}}\}$ on the locations of the Neumann edges.

5. The final step consists of going element by element and locating the pairs

   $$(n_1^K, n_2^K), \quad (n_2,^K, n_3^K), \quad (n_3^K, n_1^K)$$

   in the matrix `edges`. These are the columns of `T.edgebyelt`. Note that this can be done in an easy way using the function `sub2ind`. Basically, if `rows` and `cols` are lists of the same length

   $$\texttt{A}(\texttt{sub2ind}([\texttt{N}, \texttt{N}], \texttt{rows}, \texttt{cols}))$$

   gives the entries of the matrix `A` that are located in the positions

   $$(\texttt{rows}(i), \texttt{cols}(i)).$$

   Since we are reading from a sparse matrix, we need to transform the output to a full matrix.

```
function T=edgesFEMgrid(T)

% T=edgesFEMgrid(T)
% Input:
%     T    : basic or expanded triangulation data structure
% Output:
%     T    : fields added
%            T.edges, T.edgebyelt
```

```
%
% Last modified: February 25, 2015

N=size(T.coordinates,2);
Ndir = size(T.dirichlet,2);
Nneu = size(T.neumann,2);

edges = sparse(T.elements,T.elements([2 3 1],:),1); % all edges
edges = edges+edges';
edges(find(edges==1))=0;   % eliminate bd edges
[i,j] = find(edges);
Nint  = sum(i<j);
edges = sparse(i,j,i<j);   % lower triangular

listInt=1:Nint;
listDir=1+Nint:Nint+Ndir;
listNeu=1+Nint+Ndir:Nint+Ndir+Nneu;

T.edges=[];
if Nint>0
    [i,j]=find(edges);
    T.edges=[T.edges [i'; j'; zeros(1,Nint)]];
    edges=sparse(i,j,listInt,N,N);
    edges=edges-edges';
end
if Ndir>0
    T.edges=[T.edges [T.dirichlet ; ones(1,Ndir)]];
    edges=edges+...
          sparse(T.dirichlet(1,:),T.dirichlet(2,:),listDir,N,N);
end
if Nneu>0
    T.edges=[T.edges [T.neumann; 2*ones(1,Nneu)]];
    edges=edges+...
          sparse(T.neumann(1,:),T.neumann(2,:),listNeu,N,N);
end
T.edgebyelt=...
    [full(edges(sub2ind([N,N],T.elements(1,:),T.elements(2,:))));...
     full(edges(sub2ind([N,N],T.elements(2,:),T.elements(3,:))));...
     full(edges(sub2ind([N,N],T.elements(3,:),T.elements(1,:))))];

return
```

11 Uniform (red) refinement of a triangulation

In a uniform red refinement, every triangle is subdivided into four equally shaped and sized triangles by joining the midpoints of the edges. This is the gist of the process:

- We start by creating the lists `T.edges` and `T.edgebyelt` using `edgesFEMgrid.m`.
- Next, we add N_{nodes} to the edge number of each of the edges and assign the vertex number $N_{\text{nodes}}+k$ to the edge number k. This gives a global numbering of all old and new nodes, keeping the old nodes at the beginning of the list. Coordinates can be easily computed at this stage.
- Finally, we can refine the triangulation. If a triangle has vertices (n_1, n_2, n_3) and midpoints (n_4, n_5, n_6) (n_4 is the midpoint of n_1 and n_2; n_5 that of n_2 and n_3; n_6 that of n_3 and n_1), the new triangles are

$$(n_1, n_4, n_6) \quad (n_2, n_5, n_4) \quad (n_3, n_6, n_5) \quad (n_4, n_5, n_6).$$

```
function T=refineFEMgridRed(T)
```

```
% T=refineFEMgridRed(T)
%
% Input:
%     T    : basic triangulation
% Output:
%     T    : red refinement of T (basic triangulation data struct)
%
% Last modified: March 2, 2015

T=edgesFEMgrid(T);
nC=size(T.coordinates,2);
nElt=size(T.elements,2);

% Renumbering edges & constructing new mesh

midpoints=0.5*(T.coordinates(:,T.edges(1,:))+...
               T.coordinates(:,T.edges(2,:)));
T.coordinates=[T.coordinates midpoints];
local = [T.elements; nC+abs(T.edgebyelt)];
local = local([1 4 6 2 5 4 3 6 5 4 5 6],:);
T.elements=reshape(local,3,4*nElt);

% Dirichlet and Neumann edges

if size(T.dirichlet,2)
    local = [T.dirichlet; nC+find(T.edges(3,:)==1)];
    T.dirichlet=[local([1 3],:) local([3 2],:)];
end
if size(T.neumann,2)
    local = [T.neumann; nC+find(T.edges(3,:)==2)];
    T.neumann=[local([1 3],:) local([3 2],:)];
end

T=rmfield(T,{'edges','edgebyelt'});

return
```

12 Newest Vertex Bisection

```
function T=refineFEMgridNVB(T,marked)

% T=refineFEMgridNVB(T,marked)
%
% Input:
%    T      : triangulation in basic form (no need for longest edge at
%             beginning)
%    marked : list of marked elements
% Output:
%    T      : newest vertex bisection refined triangulation
%
% Mostly direct adaptation of code by Funken, Praetorious & Wissgott
%
% Last modified: March 9, 2015

T=reorderFEMgrid(T);
T=edgesFEMgrid(T);
T.edgebyelt=abs(T.edgebyelt);

nElt = size(T.elements,2);
nEd  = size(T.edges,2);
nC   = size(T.coordinates,2);
nDir = size(T.dirichlet,2);
```

```
nNeu = size(T.neumann,2);

T.diredge = find(T.edges(3,:)==1);
T.neuedge = find(T.edges(3,:)==2);

% Mark edges: at the end of this search we have a list of all marked Edges
% that will welcome a node. Every triangle has either its first (longest)
% edge marked (and possible more) or none at all.

markEdge = zeros(1,nEd);
markEdge(T.edgebyelt(:,marked)) = 1;
checknew = 1;
while checknew
    markElt = markEdge(T.edgebyelt);
    newElts = find( ¬markElt(1,:) & (markElt(2,:) | markElt(3,:)) );
    markEdge(T.edgebyelt(1,newElts)) = 1;
    checknew=length(newElts);  % ≠0 if there are new edges
end
idx = find(markEdge);
nNewN = sum(markEdge(:));
markEdge(idx) = nC + (1:nNewN);
midpoints = (T.coordinates(:,T.edges(1,idx))...
             +T.coordinates(:,T.edges(2,idx)))/2;
T.coordinates=[T.coordinates midpoints];

if nDir
    refDir = markEdge(T.diredge); % 0s or numbers
    markedEdges = find(refDir);
    if ¬isempty(markedEdges)
        T.dirichlet = [T.dirichlet(:,¬refDir), ...
                       [T.dirichlet(1,markedEdges);refDir(markedEdges)],...
                       [refDir(markedEdges);T.dirichlet(2,markedEdges)]];
    end
end

if nNeu
    refNeu    = markEdge(T.neuedge);
    markedEdges = find(refNeu);
    if ¬isempty(markedEdges)
        T.neumann = [T.neumann(:,¬refNeu), ...
                     [T.neumann(1,markedEdges);refNeu(markedEdges)],...
                     [refNeu(markedEdges);T.neumann(2,markedEdges)]];
    end
end

% New nodes for refinement of elements: newNodes contains a zero when an
% edge does not provide a new node and the number of node when it does
% It is 3 x nElt

newNodes = markEdge(T.edgebyelt);

% Determine type of refinement for each element

markedEdges=sign(newNodes);
first  = markedEdges(1,:);
second = markedEdges(2,:);
third  = markedEdges(3,:);

none = ¬first;
bisec1   = first & ¬second & ¬third;
bisec12  = first & second & ¬third;
bisec13  = first & ¬second & third;
all      = first & second & third;

T.elements = [T.elements(:,none),...
    [T.elements(3,bisec1);T.elements(1,bisec1);newNodes(1,bisec1)],...
```

```
     [T.elements(2,bisec1);T.elements(3,bisec1);newNodes(1,bisec1)],...
    [T.elements(3,bisec12);T.elements(1,bisec12);newNodes(1,bisec12)],...
     [newNodes(1,bisec12);T.elements(2,bisec12);newNodes(2,bisec12)],...
     [T.elements(3,bisec12);newNodes(1,bisec12);newNodes(2,bisec12)],...
    [newNodes(1,bisec13);T.elements(3,bisec13);newNodes(3,bisec13)],...
     [T.elements(1,bisec13);newNodes(1,bisec13);newNodes(3,bisec13)],...
     [T.elements(2,bisec13);T.elements(3,bisec13);newNodes(1,bisec13)],...
    [newNodes(1,all);T.elements(3,all);newNodes(3,all)],...
     [T.elements(1,all);newNodes(1,all);newNodes(3,all)],...
     [newNodes(1,all);T.elements(2,all);newNodes(2,all)],...
     [T.elements(3,all);newNodes(1,all);newNodes(2,all)]];
return
```

13 Variable coefficients and boundary mass

Pieces of the stiffness matrix. We want to compute the local matrices

$$\int_K k_{ij}\,\partial_j N_\alpha^K \partial_i N_\beta^K, \qquad \alpha,\beta\in\{1,2,3\}, \quad K\in\mathcal{T}_h, \quad i,j\in\{1,2\}\equiv\{x,y\},$$

for given vectorized functions of two variables k_{ij}. We will only compute the pairs $(1,1)$, $(2,2)$, and $(1,2)$. Approximating the variable coefficient k_{ij} by its value at the barycenter we can proceed as follows:

$$\begin{aligned}\int_K k_{ij}\partial_j N_\alpha^K \partial_i N_\beta^K &\approx k_{ij}(\mathbf{b}_K)\int_K (\mathbf{e}_j^\top\nabla N_\alpha^K)\cdot(\mathbf{e}_i^\top\nabla N_\beta^K)\\ &= k_{ij}(\mathbf{b}_K)\,\det \mathrm{B}_K\int_{\widehat K}(\mathbf{e}_j^\top \mathrm{B}_K^{-\top}\nabla\widehat N_\alpha)\cdot(\mathbf{e}_i^\top \mathrm{B}_K^{-\top}\nabla\widehat N_\beta)\\ &= \int_{\widehat K}(\mathrm{D}_K^{ij}\nabla\widehat N_\alpha)\cdot\nabla\widehat N_\beta\end{aligned}$$

where

$$\mathrm{D}_K^{ij} := k_{ij}(\mathbf{b}_K)\,\det \mathrm{B}_K(\mathrm{B}_K^{-1}\mathbf{e}_i)(\mathrm{B}_K^{-1}\mathbf{e}_j)^\top = \frac{k_{ij}(\mathbf{b}_K)}{\det \mathrm{B}_K}\mathbf{d}_{K,i}\mathbf{d}_{K,j}^\top.$$

In the last expression, $\mathbf{d}_{K,i}$ are the columns of

$$(\det \mathrm{B}_K)\,\mathrm{B}_K^{-1} = \begin{bmatrix} y_3-y_1 & -(x_3-x_1)\\ -(y_2-y_1) & x_2-x_1\end{bmatrix},$$

and $\mathbf{v}_i^K=(x_i^K,y_i^K)$ are the three vertices of K. Shortening

$$x_{21}=x_2-x_1,\quad x_{31}=x_3-x_1,\quad y_{21}=y_2-y_1,\quad y_{31}=y_3-y_1,$$

we can expand

$$\mathrm{D}_K^{11} = \frac{k_{11}(\mathbf{b}_K)}{\det \mathrm{B}_K}\begin{bmatrix} y_{13}^2 & -y_{12}y_{13}\\ -y_{12}y_{13} & y_{12}^2\end{bmatrix},$$

$$\mathrm{D}_K^{22} = \frac{k_{22}(\mathbf{b}_K)}{\det \mathrm{B}_K}\begin{bmatrix} x_{13}^2 & -x_{12}x_{13}\\ -x_{12}x_{13} & x_{12}^2\end{bmatrix},$$

$$\mathrm{D}_K^{12} = \frac{k_{12}(\mathbf{b}_K)}{\det \mathrm{B}_K}\begin{bmatrix} -y_{13}x_{13} & x_{12}y_{13}\\ y_{12}x_{13} & -x_{12}y_{12}\end{bmatrix}.$$

Note finally that everything can be written in terms of the elements of the matrices D_K^{ij}

$$\left[\int_{\widehat K}(\mathrm{D}_K^{ij}\nabla\widehat N_\alpha)\cdot\nabla\widehat N_\beta\right]_{\alpha,\beta} = d_{K,11}^{ij}\mathrm{K}_{11}+d_{K,12}^{ij}\mathrm{K}_{12}+d_{K,21}^{ij}\mathrm{K}_{21}+d_{K,22}^{ij}\mathrm{K}_{22},$$

which leads to Kronecker product computations (see Section 5). The assembly process is done with the same strategies as the mass and stiffness matrices.

Variable mass matrix. Using the same kind of ideas, it is easy to compute

$$\int_K c\, N_\alpha^K N_\beta^K \approx c(\mathbf{b}_K) \det \mathrm{B}_K \mathrm{M}_{\alpha\beta}$$

(see Section 5.)

Boundary mass matrix. A final matrix we will compute is

$$\int_{\Gamma_N} k\, \varphi_i \varphi_j, \qquad i,j = 1,\dots,N_{\text{nodes}},$$

where k is a function. Using the parametrized one-dimensional basis functions of Section 6, $\psi_1 = 1-t$, $\psi_2 = t$, and the parametrization $\phi_e : [0,1] \to e$, we compute

$$\int_e k\, (\psi_\alpha^\circ \phi_e^{-1})(\psi_\beta \circ \phi_e^{-1}) \approx |e| k(\mathbf{m}_e) \int_0^1 \psi_\alpha(t)\psi_\beta(t)\mathrm{d}t, \qquad \alpha,\beta = 1,2.$$

The matrix in the reference interval is

$$\mathrm{B} = \frac{1}{6}\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

and, therefore, the local matrices are

$$|e| k(\mathbf{m}_e)\mathrm{B}.$$

These matrices can be computed using a Kronecker product. The assembly is done using the information of `T.dirichlet` and `T.neumann`. The process is very similar to the one used for assembling the mass and stiffness matrix, substituting the list `T.elements` by the joint list of all boundary edges `[T.dirichlet T.neumann]`.

```
function [Sxx,Sxy,Syy,M,MB]=matricesFEMenhanced(T,kxx,kxy,kyy,c,k)

% [Sxx,Sxy,Syy,M,MB]=matricesFEMenhanced(T,kxx,kxy,kyy,c,k)
%
% Input:
%     T    : expanded triangulation data structure
%     kxx, kxy, kyy, c, k : vectorized functions of two variables
% Output:
%     Sxx, Sxy, Syy : variable stiffness matrices (sparse)
%                     Sab = \int kab \partial_b\phi_i \partial_a\phi_j
%     M             : variable mass matrix (sparse)
%                           \int c \phi_i \phi_j
%     MB            : boundary variable mass matrix (sparse)
%                           \int_\Gamma_N k \phi_i \phi_j
% Last modified: March 9, 2015

nE=size(T.elements,2);
nC=size(T.coordinates,2);

% Matrices in the reference element

K11=0.5*[1 -1 0;-1 1 0;0 0 0];
K22=0.5*[1 0 -1;0 0 0;-1 0 1];
K12=0.5*[1 0 -1;-1 0 1;0 0 0]';
M  =1/24*[2 1 1;1 2 1;1 1 2];
MB =1/6*[2 1;1 2];

% Evaluations of coefficients in barycenters

x=T.baryc(1,:); y=T.baryc(2,:);
kxx=kxx(x,y)./T.detB;
kyy=kyy(x,y)./T.detB;
```

```
kxy=kxy(x,y)./T.detB;
c  =c(x,y).*T.detB;

% Geometric coefficients

x12=T.coordinates(1,T.elements(2,:))-T.coordinates(1,T.elements(1,:));  % x(2)-x(1)
y12=T.coordinates(2,T.elements(2,:))-T.coordinates(2,T.elements(1,:));  % y(2)-y(1)
x13=T.coordinates(1,T.elements(3,:))-T.coordinates(1,T.elements(1,:));  % x(3)-x(1)
y13=T.coordinates(2,T.elements(3,:))-T.coordinates(2,T.elements(1,:));  % y(3)-y(1)

% Computation and assembly of mass and stiffness

Sxx=kron(kxx.*y13.^2,K11)+kron(kxx.*y12.^2,K22)...
    -kron(kxx.*y12.*y13,K12+K12');
Syy=kron(kyy.*x13.^2,K11)+kron(kyy.*x12.^2,K22)...
    -kron(kyy.*x12.*x13,K12+K12');
Sxy=-kron(kxy.*y13.*x13,K11)-kron(kxy.*x12.*y12,K22)...
    +kron(kxy.*x12.*y13,K12)+kron(kxy.*x13.*y12,K12');
M  =kron(c,M);

R=repmat((1:3)',1,3);
R=reshape(T.elements(R,:),3,3,nE);
C=permute(R,[2 1 3]);

Sxx=sparse(R(:),C(:),Sxx(:));
Syy=sparse(R(:),C(:),Syy(:));
Sxy=sparse(R(:),C(:),Sxy(:));
M=sparse(R(:),C(:),M(:));

% Computation and assembly of boundary mass

Boundary =T.neumann;
Midpoints=0.5*(T.coordinates(:,Boundary(1,:))...
              +T.coordinates(:,Boundary(2,:)));
Lengths  =sqrt(sum((T.coordinates(:,Boundary(2,:))...
                   -T.coordinates(:,Boundary(1,:))).^2));
k=k(Midpoints(1,:),Midpoints(2,:)).*Lengths;
MB=kron(k,MB);

nBelt=size(Boundary,2);
R=repmat((1:2)',1,2);
R=reshape(Boundary(R,:),2,2,nBelt);
C=permute(R,[2 1 3]);
MB=sparse(R(:),C(:),MB(:),nC,nC);

return
```