Finite element method, Matlab implementation

Main program

The main program is the actual finite element solver for the Poisson problem. In general, a finite element solver includes the following typical steps:

1. Define the problem geometry and boundary conditions, mesh generation. In this example, we download a precomputed mesh.

2. Assemble the required global matrices and implement the boundary conditions

3. Solve the resulting linear system of equations

4. Visualize and/or post-process the approximate solution

In more advanced finite element software packages the user interface varies, but these are the core features that the solver must have. In FreeFem++ and FEniCS finite element packages, the user is only required to insert the weak forms of equations.

Our example code below does not consist any problem definition interface, but is restricted to solve the Poisson problem on a given mesh. Moreover, the homogeneous Dirichlet boundary condition is fixed on the entire boundary.

```matlab
% Olli Mali 3.2.2015
%
% Computes the finite element approximation of the
% Poisson problem.
%
clear all

% Load a pregenerated mesh data, generated by Matlab pdetoolbox
% p 2 x N nodes of the mesh
% e 7 x E boundary edges of the mesh
% t 4 x M triangles of the mesh (bottom row extra
% label)
load mesh_morko N = size(p,2);```
% Assemble global matrices and vector
f = @(x) ones(1, size(x,2)); % define right hand side
don = 3; % order of integration

quad = AssembleGlobalMatrices(t,p,f,ord);

% Apply homogeneous Dirichlet boundary condition and solve
bnodes2nodes = unique([e(1,:); e(2,:)]); % locate boundary nodes
dofs = setdiff(1:N, bnodes2nodes); % create indeces of nonboundary nodes
u_h = zeros(N,1); % initialize solution to zeros
u_h(dofs) = S(dofs, dofs) \ b(dofs); % solve for nonzero values

% Plot triangular solution
figure(1); clf; trisurf(t(1:3,:)', p(1,:)', p(2,:)', u_h)

Matrix assembly

The heart of the finite element code is the matrix assembly. It is often even more time consuming than solving the respective linear system.

function [S,M,b] = AssembleGlobalMatrices(t,p,f,ord)
% Assembles the global stifness matrix
% a_ji = (nabla psi_i, nabla psi_j)
% and the right hand side b_j = (f,psi_j).
% Olli Mali 20.1.2015
% IN:
% t 3 x M M triangles
% p 2 x N N nodes (1st row x-coord, 2nd row y-coord)
% f fhandle handle to a function
% ord integration quadrature
%
% OUT:
% S N x N stifness matrix
% K N x N mass matrix
% b N x 1 right hand side load
%
First we define from the inputted mesh data the amount of nodes and elements. Note that we make no different treatment for the boundary nodes.
They are handled “later”, when defining the system of linear equations.

\[ M = \text{size}(t,2); \quad \% \text{NOF elements} \]
\[ N = \text{size}(p,2); \quad \% \text{NOF nodes} \]

For the sake of performance we allocate the memory for the global matrices. This is practically necessary, since otherwise Matlab reconstructs matrices every time new elements are added. Moreover, the matrices are defined as sparse, i.e., zeros are not stored. Instead, matrix elements and their coordinates (in one representation or another) are stored.

\[ \text{St} = \text{sparse}(N,N); \quad \% \text{Initialize sparse matrices} \]
\[ \text{Ma} = \text{sparse}(N,N); \]
\[ \text{b} = \text{zeros}(N,1); \]

We obtain the integration points and weights on the reference element. They can be precomputed and depend only on the respective integration order.

\[ \% \text{Compute quadrature on the reference element} \]
\[ [\text{iw}, \text{ip}] = \text{RefElemQuad}(\text{ord}); \]

The values of basis functions and their gradients are computed at the integration points on the reference element. Note that if one would change the basis functions, this would be the only place where the change had to be implemented (if the amount of local basis functions changes, then the size of local matrices has to be changed).

\[ \% \text{Compute basis functions and their gradients values at the} \]
\[ \% \text{integration points of the reference element} \]
\[ [\text{basis}_\text{ip,grad}\_\text{basis}_\text{ip}] = \text{basis}\_\text{linear}(\text{ip}); \]

\[ \% \text{Initialize local matrices} \]
\[ \text{Local}_\text{M}_\text{matr} = \text{zeros}(3,3); \]
\[ \text{Local}_\text{S}_\text{matr} = \text{zeros}(3,3); \]
\[ \text{Local}_\text{load} = \text{zeros}(3,1); \]

Go through all elements to constructs the inner products of global basis functions using the local basis functions (restrictions of global basis functions on an element).

\[ \text{for } k = 1:M \quad \% \text{go through all elements} \]
Construct the elementwise affine mapping using the coordinates of the nodes (corners of the triangle). Compute also the determinant and the inverse of the transpose required for the change of variables.

\% corners of the k'th element
nodes = p(:,t(1:3,k));

\% Compute the affine mapping \( F(x) = Cx + cv \) and its determinant
C = [ nodes(:,2)-nodes(:,1) , nodes(:,3)-nodes(:,1) ];
cv = nodes(:,1);
Cdet = C(1,1)*C(2,2)-C(1,2)*C(2,1);

\% Compute the transpose of the inverse of \( C \)
CinvT = (1/Cdet) * [ C(2,2) , -C(2,1) ; -C(1,2), C(1,1) ];

Use the affine mapping to map the integration points on the reference element to the current element, i.e., compute the local integration points. Then compute the value of \( f \) at these points

\% Map the integration points to local element
ip_local = C * ip + kron(cv,ones(1,length(ip)));

\% Calculate values at local points
f_at_ip = f( ip_local );

Compute the inner products of local basis functions, products of the gradients, and the product with \( f \), using the change of variables and the integration quadrature.

\% Assemble the local matrices
for i = 1 : 3
    Local_load( i ) = (f_at_ip .* basis_ip(i,:)) * iw' * Cdet;
    for j = 1 : 3
        Local_M_matr(i,j) = basis_ip(i,:).*basis_ip(j,:)*iw'*Cdet;
        Local_S_matr(i,j) = (CinvT * grad_basis_ip(:,1,i))' * ... 
                          (CinvT * grad_basis_ip(:,1,j)) * Cdet;
        \% Selecting gradient value at first ip is enough since
% gradient is constant at each element.
end
end

Add the contributions of the local basis functions to the global matrix. In other words, the elementwise contributions of the inner products (integrals) of global basis functions are added to the corresponding element in the global matrix.

% Add the contribution of local matrices to the global.
% b(t(1:3,k)) = b(t(1:3,k)) + Local_load;
% St(t(1:3,k), t(1:3,k)) = St(t(1:3,k), t(1:3,k)) + Local_S_matr;
% Ma(t(1:3,k), t(1:3,k)) = Ma(t(1:3,k), t(1:3,k)) + Local_M_matr;

end
end

Linear basis functions

This function computes the value of the linear local basis functions on a reference triangle at given points. Moreover, it returns the respective values of gradients.

function [value,d_value] = basis_linear(x)

% Evaluates the basis functions for 2D triangular
% linear element
%
% i  function  node
%
% 1  1-x-y  (0,0)
% 2  x       (1,0)
% 3  y       (0,1)
%
% IN: x,  (2 x M) Set of points
%
% OUT: value,  (3 x M) values of basis functions on points x.
% d_value,  (2 x 3 x M) values of gradients on points x
M = size(x,2);

value = zeros( 3 , M );

value(1,:) = ones(1,M) - x(1,:) - x(2,:);
value(2,:) = x(1,:);
value(3,:) = x(2,:);

d_value=zeros(2,M,3);
v = ones(1,M);

d_value(:,:,1) = [-v ; -v];
d_value(:,:,2) = [ v ; zeros(1,M)];
d_value(:,:,3) = [ zeros(1,M) ; v ];

Integration quadrature on a reference element

function [ w,p ] = RefElemQuad( order )

% Return symmetric Gaussian integration quadrature on
% the reference element (0,0)-(1,0)-(0,1).
% Olli Mali 20.1.2015
%
% IN:
% order integer order of the quadrature
% OUT:
% w 1xN weights of the quadrature
% p 2xN points of the quadrature
%
switch order
  case 1
    w = 1/2;
    p = [ 1/3 ; 1/3 ];
  case 2
    w = (1/6)*[1,1,1];
    p = [ 0, 0.5, 0.5; 0.5, 0, 0.5];
  case 3
    w = (1/48)*[-27, 25, 25, 25];
    p = [ 1/3, 2/15, 2/15, 11/15 ; ... 
         1/3, 11/15, 2/15, 2/15 ];
  case 4
    w1 = 0.22338158967801;
w2 = 0.10995174365532;
w = [ w1, w1, w1, w2, w2, w2 ];
a = 0.44594849091597;
b = 0.10810301816807;
c = 0.09157621350977;
d = 0.81684757298046;
p = [ a, a, b, c, c, d ; a, b, a, c, d, c ];
case 5
  w2 = 0.13239415278851;
w3 = 0.12593918054483;
w = [ 0.225, w2, w2, w2, w3, w3 ];
a = 0.47014206410511;
b = 0.05971587178977;
c = 0.10128650732346;
d = 0.79742698535309;
p = [ 1/3, a, a, b, c, c, d ; 1/3 a, b, a, c, d, c ];
otherwise
  error(['RefElemQuad: input parameter order wrongly defined. '
        'Should be integer from 1 to 5.']);
end