FINITE ELEMENT METHOD IN 2D WITH MATLAB PDE TOOLBOX

[1]

In this lab we will use the PDE Toolbox in MATLAB to solve partial differential equations in two dimensions. The finite element method for solving PDEs in two (or three) dimensions works essentially in the same as in one dimension, the main difference being that a domain in two or three dimensions can be much more complex than a one-dimensional interval.

For a domain $\Omega \subset \mathbb{R}^2$, we consider the (elliptic) equation

$$-\nabla \cdot \big(c\nabla u\big) = f \quad \text{in } \Omega, \tag{1a}$$

with boundary conditions

$$\begin{cases} \nabla u \cdot n + qu = g & \text{on } \Gamma_N \\ u = r & \text{on } \Gamma_D \end{cases} \tag{1b}$$

Here Γ_N and Γ_D denotes disjoint parts of the boundary where the (generalised) Neumann and Dirichlet boundary conditions apply, respectively.

1. A RECAP OF THE FINITE ELEMENT METHOD IN TWO DIMENSIONS

The variational form of the elliptic problem (1) reads as follows: Find u such that $u = r$ on Γ_D and for all test functions v ($v = 0$ on Γ_D),

$$\iint_\Omega \nabla u \cdot \nabla v \, dx + \int_{\Gamma_N} quv \, ds = \iint_\Omega fv \, dx + \int_{\Gamma_N} gv \, ds. \tag{2}$$

For the finite element method we also need a mesh, or triangulation of Ω. We denote the mesh $\mathcal{T}_h$, with

$$\mathcal{T}_h = \big(\{p\}_{i=1}^n, \{e\}_{i=1}^m, \{t\}_{l=1}^l\big), \tag{3}$$

where

- $p_1, \ldots, p_n$ are the vertices (nodes)
- $e_1, \ldots, e_l$ are the edges
- $t_1, \ldots, t_m$ are the triangles, in the mesh
- h is the diameter of the largest triangle, i.e. $h = \max_{1 \le i \le M} \operatorname{diam}(t_i)$.

The finite element space is defined to be the continuous functions that are piecewise linear with respect to the mesh $\mathcal{T}_h$, i.e.

$$V_h = \big\{u \in C(\Omega) : u|_{K_i} \in \mathcal{P}_1(K_i),\ i = 1, \ldots, m\big\}$$

where $\mathcal{P}_1(K_i)$ is the space of first order polynomials (plane) on the triangle K_i. A basis $\{\phi_j\}_{j=1}^n$ for V_h is uniquely determined from the nodal property

$$\phi_j(x_i) = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j. \end{cases}$$

[1] 2017, Magne Nordaas, Matematiska vetenskaper, Chalmers

Setting $u_h = \sum_{j=1}^n \xi_j \phi_j$ and $v = \phi_i$ in (2), we have

$$\sum_{j=1}^{n} \xi_j \underbrace{\iint_\Omega \nabla\phi_j \cdot \nabla\phi_i \, dx + \int_{\Gamma_N} q\phi_j\phi_i \, ds}_{=A_{ij}} = \underbrace{\iint_\Omega f\phi_i \, dx + \int_{\Gamma_N} g\phi_i \, ds}_{=b_i}, \qquad i = 1, \ldots, n.$$

This is a linear system,

$$\boldsymbol{A\xi} = \boldsymbol{b},$$

which can be solved for the nodal values $\xi_1, \ldots, \xi_n$.

1.1. **Finite element error in two dimensions.** The error estimates for the finite element method in two dimensions is similar to those for one dimensional problem. With h as the mesh parameter defined above, we have the estimate

$$\|u - u_h\|_2 = \sqrt{\iint_\Omega (u - u_h)^2 \, dx} \leq Ch^2, \tag{4}$$

where C is a constant that depends on u, on the quality[2] of the mesh $\mathcal{T}_h$, but not on the mesh parameter h.

2. Introduction to PDE Toolbox

PDE Toolbox is an extension to MATLAB for solving certain PDEs on two- and three-dimensional domains. It can solve symmetric elliptic, parabolic and hyperbolic PDEs, and systems of such PDEs.

PDE Toolbox has functions that handles the low-level parts of the finite element method for us. For example, it has functions for mesh generation and matrix assembly, and solvers for stationary and time-dependent problems. All we need to do is to provide the problem specification. This is done in a number of steps as described below.

2.1. **Overview.** PDE Toolbox has a graphical user interface (GUI), that can be started with the command `pdetool` or from the Apps panel in MATLAB. It is also possible to use the toolbox without using the GUI and instead write a script using PDEToolbox functions, but in this lab, we will only work with the GUI. The next sections describes in detail how to set up and solve a PDE with the PDE Toolbox.

(1) **Defining the geometry in *draw mode*.** The *domain* Ω in (1) is an important part of the problem formulation, and a precise description of it have to be provided to PDE Toolbox before we can do anything else. We can describe the geometry by using Constructive Solid Geometry (CSG). This means that the domain is constructed from a collection of basic solid objects (polygons and ellipsoids in two dimensions) through a sequence of boolean operations.

(2) **Setting boundary conditions in *boundary mode*.** When the geometry has been described, the boundary $\partial\Omega$ is known, and boundary conditions (1b) can be set. The type of of the boundary condition and the parameter

[2]One measure of mesh quality is the largest triangle angle over all the triangles in the mesh. An angle close to π indicates that the mesh quality is poor.

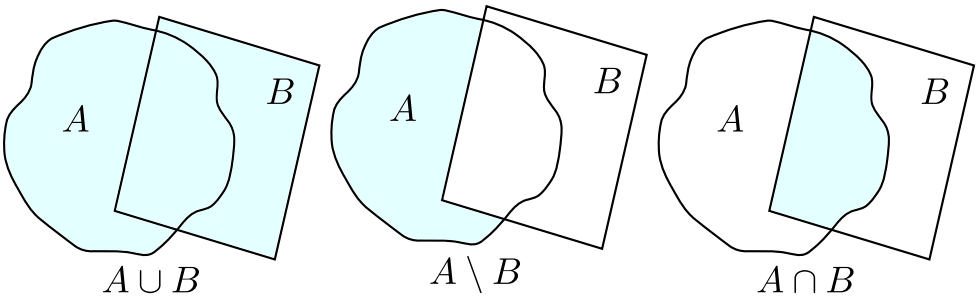

FIGURE 1. Visualising set operations: Union (left), difference (middle) and intersection (right)

values have to be specified for each part of the boundary. PDE Toolbox supports Dirichlet, Neumann and Robin boundary conditions.

(3) **Setting equation parameters in *PDE mode*.** The parameters c and f in (1a) have to provided. These parameters can be specified as scalar constants or as functions.

(4) **Meshing *PDE mode*.** The mesh (3) have to be constructed. The toolbox can do this automatically.

(5) **Solving and postprocessing.** At the end, the problem have to solved. The toolbox has functions for solving both stationary and time-dependent problems. It has functions for plotting, and it also possible to export the data for further postprocessing.

3. Defining the geometry

In *draw* mode you can draw a number of basic solids – rectangles, ellipsoids and polygons. This can be done from the drop-down `draw` menu, or by clicking the buttons below it. It is also possible to select solids to move or rotate them while in `draw` mode.

The domain is constructed from the basic solids using set operations, which can be typed in the formula box under the icons. These operations are

- `+` for union ($\cup$)
- `*` for intersections ($\cup$)
- `-` for difference ($\setminus$)
- `%` only uses the solid to define subdomains.

Set operations are in general not commutative or associative, so it is necessary to use parenthesis to ensure that they carried out in the right order. Note that by default, the domain is the union of all the basic solids.

It is hard to get precise results when drawing freehand. Enabling `Snap` under the `Options` menu helps a lot, as it will make all drawings snap to grid lines. The gridlines need to be set to align with geometric features, by choosing `Grid Spacing` and `Axes Limits` from the same menu.

3.1. Setting boundary conditions.

When the geometry has been described, MATLAB will automatically determine the boundary. Enter boundary mode by clicking on the button marked $\partial\Omega$, or selecting `Boundary Mode` from the `Boundary` menu.

The boundary will divided into a number of segments, that can be selected with a left mouse button click. Press shift when clicking to select additional segments. The selected boundary segments can be assigned a boundary condition by clicking `Specify Boundary Conditions` in the `Boundary` menu, which opens up a dialog box. On the left side you can select between Dirichlet and Neumann type conditions, and on the right parameter values can be typed in.

The Neumann boundary condition is specified as

$$n \cdot \nabla u + qu = g \quad \text{on } \Gamma_N,$$

or

$$n \cdot c\nabla u + qu = g \quad \text{on } \Gamma_N,$$

and the Dirichlet boundary condition is specified as

$$hu = r \quad \text{on } \Gamma_D.$$

The parameters g and r correspond to g and r in (1b). A nonzero value for q results in a generalised Neumann condition, also called a *Robin* boundary condition. The parameter h (not to be confused with the mesh size parameter also denoted h) can always be set to 1 for Dirichlet boundary conditions.

The boundary values can be constants or functions. Functions are written same way as in a normal MATLAB script, and with `x` and `y` used to represent the x and y coordinates. For example, if we want the boundary value to be $\sin(\pi x)\sin(\pi y)$, we type in `sin(pi*x).*sin(pi*y)`. Note that the operations have to be elementwise, i.e. use `.*` and `.^` instead of `*` and `^`.

3.2. **Setting PDE parameters.** PDE mode is activated with `PDE Mode` in the `PDE` menu. In this mode, the subdomains are visible, and can be selected. Clicking the buttons marked `PDE` or selecting `PDE specification` from the `PDE` menu opens a dialogue box where the parameters in the equation can be set. If one or more subdomains are highlighted, the values will be applied to the selected subdomains. Otherwise, the values will be applied to all subdomains. Functions can be entered the same way as for boundary conditions.

The type of equation can also be set, the options being elliptic (Poisson's equation) parabolic (heat equation) or hyperbolic (wave equation).

3.3. **Meshing.** Meshing geometries in two or three dimensions is much more complicated than in one dimension. Fortunately, MATLAB has functions that can do the meshing automatically. Enter mesh mode by selecting `Mesh Mode` from the `Mesh` menu. There are two buttons for meshing. The button with the simple triangle creates an initial mesh. The button with the inscribed triangle refines the current mesh, by subdividing each triangle into four new triangles.

Some parameters for the meshing process can changed by selecting `Parameters` from the `Mesh` menu. In particular, targets can be set for the upper and lower limits of the triangle sizes.

3.4. **Solving and postprocessing.** Finally, solving is as simple as clicking the button marked with the equality sign, or selecting `Solve PDE` from the `Solve` menu. Some parameters of the solver can be set selecting `Parameters` from the `Solve` menu, depending on wether the problem is time-dependent or not. For stationary problems (elliptic PDEs), we can opt to use adaptive or nonlinear solvers. For time-dependent

problems, we can only set parameters for the time stepping, e.g. the final time and the length of time intervals.

Once the solution is computed, it will automatically be plotted. Some plot configuration is possible when selecting `Plot > Parameters`. For example, contour lines and gradient arrows can be plotted. There is also an option to generate a movie to visualise the solution of time-dependent problems.

Sometimes the PDE toolbox may not provide all the tools we need. We can export the data, and work with it in a standard MATLAB script. Selecting `Mesh > Export Mesh` saves the mesh data to the work space, and selecting `Solve > Export Solution` saves the solution vector.

4. Exercises

Exercise 1. First, let us verify that MATLAB is solving the partial differential equations correctly. We do this the same way we did in the previous labs, using the method of manufactured solutions and checking that the error estimate (4) holds. At the same time, we go through the steps of solving PDEs with the toolbox.

For this exercise we consider the problem

$$-\nabla \cdot \big(c\nabla u\big) = f \quad \text{in } \Omega, \tag{5a}$$

with the Dirihlet boundary condition

$$u = r \quad \text{on } \Gamma \tag{5b}$$

with the choosen the exact solution to (5) to be

$$u(x, y) = \sin(\pi x)\sin(\pi y). \tag{5c}$$

(a) Enter *draw mode* and draw an arbitrary polygon. It can be a single polygon, or a composite object consisting of multiple polygons.
(b) Enter *boundary mode* and set the boundary conditions to Dirichlet on all edges, and set the parameter `r` to be the function (5c). Make sure the parameter `h` is set to 1.
(c) Enter *PDE mode* and set the equation type to elliptic. Set `c` to 1 and set `f` to to be consistent with the chosen exact solution (5c).
(d) Enter *mesh mode* and generate a mesh. This is done by clicking the button marked with a simple triangle, or by selecting `Initialize Mesh` from the `Mesh` menu. Solve the equation (make sure that the adaptive solver is *not* activated) and *export* the mesh and solution.

You should now have variables `u`, `p`, `e` and `t` in your MATLAB workspace, provided you exported the data with the default variable names. The function `Errornorm2D` (see *Appendix A* at the end of this file) from the script `Errornorm2D.m` can be used to compute the error of the finite element solution. For example, if the mesh and solution was exported with the default variable names, the following code will work:

```
U = @(x) sin(pi*x(1)) * sin(pi*x(2));
error = Errornorm2D(U, u, p, t)
```

We now want to compute the error for different mesh resolutions, to verify that the numerical solutions converge to the exact solution as expected.

First, compute the error and write down the result with the initial mesh you have after steps (a)–(d) above. Then, refine the mesh once by clicking the button with the inscribed triangle, or selecting `Mesh > Refine Mesh` from the menu. This will result in a new and finer mesh, with the four times as many triangles as the first mesh. Solve the equation again, and export both the new solution and the new mesh to compute the error for the refined mesh. Make sure to write down the result. Repeat the process for a second refinement.

You should now have three values for the error, for the three different levels of mesh refinement. Compare the errors to the error estimate (4). Does it appear to be in agreement? Keep in mind that the mesh size parameter h in (4) is divided by two with each mesh refinement, since each triangle is subdivided in four triangles.

Exercise 2. For this exercise we look at a more complex problem. Consider the geometry in figure 2, with $\Omega = \Omega_1 \cup \Omega_2 \cup \Omega_3 \cup \Omega_4$ and boundary $\Gamma = \partial\Omega$.

$$-\operatorname{div}\big(c\nabla u\big) = f \quad \text{in } \Omega, \tag{6a}$$

with *Robin* boundary conditions

$$c\nabla u \cdot n + qu = 0 \quad \text{on } \Gamma, \tag{6b}$$

Where $q = 0.1$ and the coefficient c and source f are piecewise constant functions as follows:

$$c(x) = \begin{cases} 1 & \text{if } x \in \Omega_1 \cup \Omega_2 \\ 10^{-3} & \text{if } x \in \Omega_3 \\ 10 & \text{if } x \in \Omega_4 \end{cases} \tag{6c}$$

$$f(x) = \begin{cases} 1 & \text{if } x \in \Omega_1 \\ -1 & \text{if } x \in \Omega_2 \\ 0 & \text{if } x \in \Omega_3 \cup \Omega_4. \end{cases} \tag{6d}$$

(a) Draw the domain shown in figure 2 with PDE toolbox. This requires two rectangles and two circles. To make drawing easy, make suitable adjustments to the axes limits and gridlines in the `Options` menu.

(b) Set the boundary condition in boundary mode, and set values for c and f in PDE mode. Make sure that the PDE type is set to elliptic, and that values are given correctly for each subdomain.

(c) Solve the equation and enable contours from `Plot > Parameters`, and try different mesh levels of mesh refinement. How many times do you have to refine the initial mesh before you get a solution that you deem to be good? Export the mesh and the solution, and make a surface plot with the following command:

```
>> pdesurf(p, t, u) % Here p, t and u are exported data
```

Save the figure.

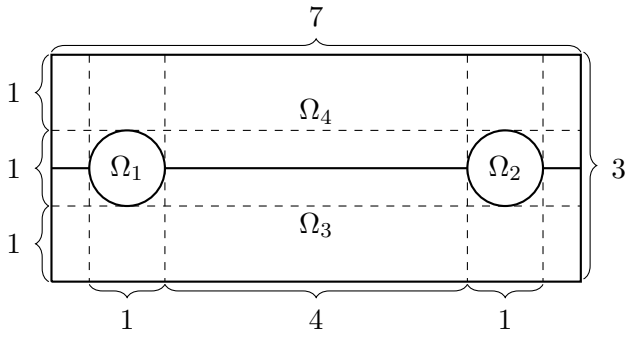

FIGURE 2.
A rectangular domain, consisting of four subdomains $\Omega_1, \Omega_2, \Omega_3$ and Ω_4. The different subdomains have different material properties (i.e. different values for c). The subdomain Ω_1 has a heat source with $f(x) = 1$, and Ω_2 has a heat sink with $f(x) = -1$. The measurements and parameters are given in dimensionless units.

(d) Equation (6a) is the stationary limit of the heat equation

$$\partial_t u - \mathrm{div}\big(\mathrm{c}\nabla \mathrm{u}\big) = \mathrm{f} \quad \text{in } \Omega,$$

with the same parameters and boundary conditions as before. We will now solve this heat equation.

Enter PDE mode and change the equation type to parabolic, with the parameter `d` set to 1. Make sure that all the subdomains still have the correct parameter values, and that you have refined the initial mesh at least three times.

From the menu, select `Solve > Parameters` and set time to be `0:0.005:1`, then solve. Export the mesh and solution, and plot the solution with

```
>> pdesurf(p, t, u(:, end)) % u(:, end) is final solution
```

Finally, make a movie of the time evolution by selecting from the menu `Plot > Parameters` and clicking the button marked animation (you can leave the presented values unchanged). Has the solution become stationary? Compare to your figure from (c).

Appendix A. Errornorm2D

```
function [ error ] = Errornorm2D(u_e, u_h, p, t)
% Errornorm2DComputes the L^2 error (with the following input)
%     u_e  --  exact solution
%     u_h  --  nodal values of numerical solution
%     p    --  nodal coordinates
%     t    --  the triangles in the mesh
    % Reference element quadrature points
    q = [0.816847572980459, 0.091576213509771; ...
         0.091576213509771, 0.816847572980459; ...
         0.091576213509771, 0.091576213509771; ...
         0.108103018168070, 0.445948490915965; ...
         0.445948490915965, 0.108103018168070; ...
         0.445948490915965, 0.445948490915965]';
    w = zeros(6,1);
    w(1:3) = 0.109951743655322;
    w(4:6) = 0.223381589678011;  w = w/2;  m = length(w);
    % Tabulate reference basis
    basis = {@(x) 1-x(1)-x(2); @(x) x(1); @(x) x(2)};
    B = zeros(m, 3);
    for i=1:m
        for j=1:3
            B(i, j) = basis{j}(q(:, i));
        end
    end
    % Loop over triangles:
    error2 = 0;
    for k = 1:length(t)
        % Determine trinagle coordinates
        vertices = t(1:3, k);
        coordinates = p(:,vertices);
        % Obtain the affine map from reference simplex
        A = coordinates(:,2:end)-repmat(coordinates(:,1),1,2);
        b = coordinates(:,1);
        J = abs(det(A)); % Jacobian
        % Map reference quadrature points
        Q = repmat(b,1,6) + A * q;
        % evaluate the numerical solution
        u_h_k = B * u_h(vertices);
        % evaluate exact solution
        u_e_k = zeros(m,1);
        for j=1:m
            u_e_k(j) = u_e(Q(:,j));
        end
        % Sum local L^2 error contribution
        error2 = error2 + J * sum(w.*(u_e_k - u_h_k).^2);
    end
    error = sqrt(error2);
end
```