# *femg* — A Suite of MATLAB Finite Element Codes

Robert Scheichl and Ivan Graham

This code solves the problem

$$-\Delta u = f \text{ on } \Omega \,, \text{ subject to } \; u = g \text{ on } \Gamma \,,$$

where $\Omega$ is a domain in $\mathbb{R}^2$ with a polygonal boundary $\Gamma$. The method used is the finite element method with piecewise linear basis functions on a triangular mesh on $\Omega$. The imposed boundary condition, $u = g$ on the boundary of the domain is commonly called a "Dirichlet" boundary condition and this terminology is used below and in the comments in the code. The code consists of the following main modules:

- **femg.m** This is the main program. It first computes an initial mesh specified in **initial_mesh.m**. Then it allows the user to perform any number of uniform refinements to it using **refine.m**. The initial and the final meshes are visualised using the function **visualise.m**. Then the element stiffness matrices for all the elements are calculated using **elt_stiffness.m** and the stiffness matrix is assembled from these using **global_stiffness.m**. For convenience the matrix assembled initially includes rows and columns corresponding to all the nodes in the mesh. It is called $\hat{A}$ (=`Ahat`) in the code. To impose the Dirichlet boundary condition we simply select the rows and columns of the matrix $\hat{A}$ corresponding to interior nodes. These are also called "degrees of freedom" and the resulting stiffness matrix is called $A$ in the code. (This matrix corresponds to the matrix $A$ in lectures.)

  The system to be solved is $A\mathbf{U} = \mathbf{f} - A^D\mathbf{g}$. You will have to work out yourself what the entries of $A^D$ and of $g$ are and how this is implemented in the code. The vector **f** is assembled by the function `rhs.m` which takes as input the function `source_term.m`, which should be the right-hand side of Poisson's equation. The vector **g** is constructed by evaluating the Dirichlet data function $g(\mathbf{x})$ at the boundary nodes of the mesh. This linear system is solved using the built-in MATLAB function \ (essentially a direct solve using Gaussian elimination). The finite element solution is plotted using the built-in MATLAB functions `meshgrid`, `griddata` and `surf`. If the right-hand side was chosen such that the exact solution is known we can compare the finite element solution to the exact solution at the nodes of the mesh. This is done here and the error is then also plotted.

  The last line of the code calculates an error indicator. You will have to convince yourself that this is the quantity specified in the assignment.

- **initial_mesh.m** In this function the user can specify the domain $\Omega$ by giving the nodes and the elements of an initial (coarse) mesh. These are stored as two two-dimensional arrays. Each row of the array `nodes` contains the coordinates of the nodes together

with a "boundary indicator" in the third column: a 1 to signify a Dirichlet node and a 0 to signify an interior node. The node numbering coincides with the order in which the nodes are listed in this array. The array `elements` specifies the elements of the mesh. Each element is specified by a row which lists its nodes in anticlockwise order (the global-to-local mapping discussed in lectures). The elements are also numbered using the order in which they appear in the list.

An example of a simple triangulation of the unit square is given in the initial version of **initial_mesh.m**. You can edit this to create whatever coarse mesh you require.

- **visualise.m** This function takes the nodes and elements of the current mesh and visualises it. The colour scheme used is specified in the comments lines. Note that the nodes and elements of the current mesh can also be listed on the screen by just adding for example the line

  ```
  nodes
  ```

  to the program `femg.m`.

- **refine.m** This takes a current mesh specified by `nodes` and `elements` and computes the new mesh, obtained by applying *uniform refinement* to each element of the current mesh. (This means that each element of the old mesh is divided into four equal elements by joining the mid-points of its sides.) The utility function `visited.m` is used here.

- **elt_stiffness.m** This takes the elements and the nodes of a mesh and computes the element stiffness matrices using the formulae derived in lectures. Note that the element stiffness matrices are stored in a cell array: This is an array, each element of which is a $3 \times 3$ matrix. It uses the area function `mu.m` described below.

- **global_stiffness.m** This takes the element stiffness matrices, together with the elements and the nodes and assembles the global stiffness matrix using the procedure described in lectures. (Note that it is crucial here for efficiency that we loop over the elements and add for each element the entries in the element matrices to the partially assembled global stiffness matrix. It would be very expensive to loop over the rows and columns of $\hat{A}$.

- **rhs.m** as described above. It employs calls to the function `source_term.m` to evaluate the right-hand side of Poisson's equation using the quadrature rule described in lectures.

These depend on two utility functions:

- **visited.m** Given two nodes with global numbers `node1` and `node2` and a table of edges `edges`, this function checks whether the edge between the two nodes `node1` and `node2` has already been visited in the current refinement sweep. (The edge table is emptied after each refinement sweep.)

- **mu.m** Finds the area of a triangle with three given nodes.