ME257 - Homework 3

Submitted by : Saurabh Sharma

Attached with this report are $C++$ source files and the various data files need to run the program. The data files are names *coodinates.dat*, *connectivity.dat* and *oundarynodes.dat*. A solution file titled *sol.dat* is generated upon execution of the code.

(1)

The domain Ω was created using $pde-toolkit$ of MATLAB and figure 1 shows the domain discretised into triangular elements. The discretization shown in figure 1 is for a maximum element size of 0.2 and the size of the mesh can be controlled by the parameter $Hmax$ of the method $generateMesh()$. The required data files are generated using a MATLAB program.

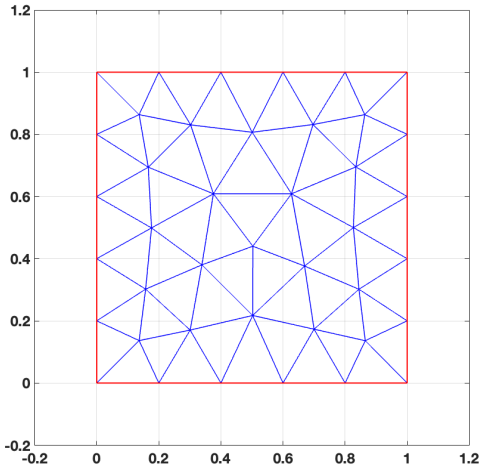


Figure 1: Discretized domain with a maximum element size of 0.2

(2)

A structure `Mesh M` is defined as per the problem statement. One additional variable `int nBdNodes` is used to store the number of nodes on the boundary $d\Omega$.

```
void ReadMesh(std::string coordFileName, std::string connFileName, std::string bdFileName, Mesh & M)
```

The above method is used to read the *.dat* files and store the data in appropriate variables. A check is coded in, to ensure the required files are available in the current directory.

(3)

The following method is used to perform the local to global map. Every node in the local numbering of an element is mapped to the global number system. This method is also used in the subsequent steps to define the linear map from the physical space (x, y) to the parametric space (ξ, η).

```
int Local2GlobalMap(const Mesh &M, int elem_num, int loc_node_num)
```

Figure 2 shows the mapping between the physical space to the parametric space using a linear map $\phi(\xi, \eta)$. The local node number for a representative triangle T_i is indicated in yellow while its corresponding global node numbering is indicated in green. The method `Local2GlobalMap()` takes the element number and the local node number and gives out its corresponding global number. In this implementation, the node number is done in counter-clock wise direction and the local node number 1 is always mapped to node 1 of the parametric triangle. A similar approach is followed for the other nodes as well. There is an inbuilt check to ensure that the element number and local node number for which the global numbering is requested, are not out of bounds.

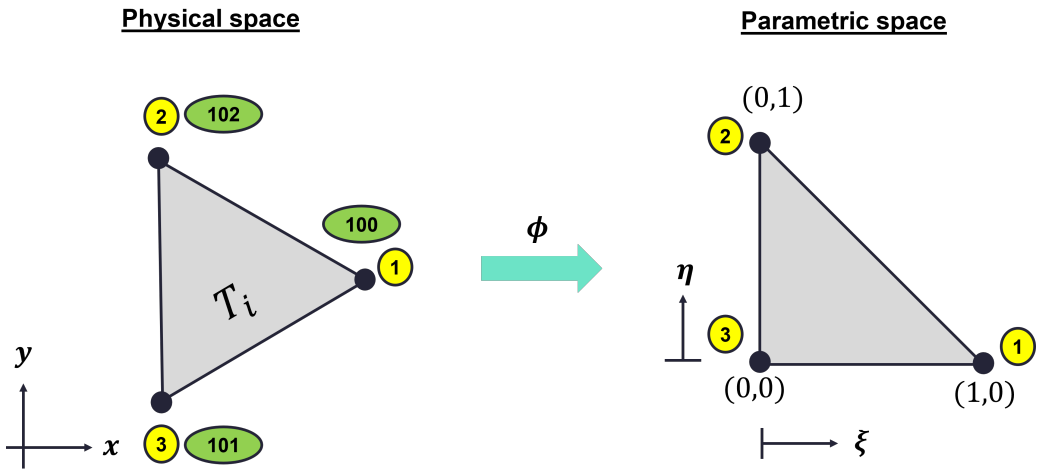


Figure 2: Mapping between physical space to parametric space

(4)

The following method is used to compute the 3×3 element stiffness matrix K_e. The derivatives of the shape functions are hard coded to `Eigen::Matrix<double, 3, 2> DerivativeOfNhats`. The gradient $\nabla\phi$ of the linear map ϕ is stored as a matrix in `Eigen::Matrix2d gradPhi` for easy manipulation for subsequent steps like finding inverse, determinant and dot product with derivative of basis functions.

```
int void ComputeElementStiffnessMatrix(const Mesh &M, int elemNum, Eigen::Matrix<double,3,3> &Ke)
```

A vector of Eigen data structure `Triplet` is used to collect all the non-zero terms of the global stiffness matrix. The global stiffness matrix is then created using the triplets into a spare matrix format

and stored in `Eigen::SparseMatrix<double, Eigen::RowMajor, int> GlobalStiffnessMatrix`.

(5)

The following method is used to compute the 3×1 element force vector F_e. Based on the forcing function f, the component of the element force vector are defined. An integer identifier `FunctionIdentifier` is used to switch between the different forcing functions f. Table 1 gives the list of forcing functions and their corresponding identification index. The values of F_e are hard coded for case 0 and for all the remaining 4 cases, **the integrals are calculated by using a three point Gauss quadrature rule for triangles**. The weights and the evaluation points for the quadrature can be found in lines 283-285 of the code. A function `double getBasisFunction(double Xi, double Eta, int i)` is used to get the values of the three basis functions at the specified quadrature points.

Table 1: Forcing functions corresponding to the user input

`FunctionIdentifier`	$f(x, y)$
0	0
1	2
2	$sin(3\pi x)$
3	$sin(3\pi x) + sin(3\pi y)$
4	$cos(7\pi x) + cos(7\pi y)$

(6)

The method `SetDirichletBCs()` is used to set the Dirichlet boundary conditions where the rows corresponding to the Dirichlet boundary in both the global stiffness matrix and the global force vector are set to zero. Subsequently, the diagonal element of the corresponding node in the global stiffness matrix is set to 1.0.

(7)

The method `printSolutions()` is defined to write an output files "*sol.dat*" which has the nodal coordinates and the approximated solution corresponding to that node. The file is then read in MATLAB and the contours are generated.

(8)

A sparse matrix solver `Eigen::SparseLU<Eigen::SparseMatrix< double, Eigen::RowMajor> > Solver` is defined and the solution is stored in the a vector `Eigen::VectorXd DOFSolution (nNodes)`.

(9)

The following checks are implemented to ensure the correctness of the code

- Check to ensure the necessary inputs files are present in the current working directory is defined in the method `ReadMesh()`
- Check implemented in method `Local2GlobalMap()` to ascertain that the input values for the mapping function is not out of bounds, thereby ensure the right global index is obtained.
- The partition of unity conditions is used to evaluate the values for the basis functions in the method `getBasisFunction()`
- A check is performed to ensure that the Jacobian for the transformation is strictly greater than zero in the method `ComputeElementStiffnessMatrix()`. This check ensures that there are no negative areas, which indicates the parametric element is not overlapping itself due to the mapping
- The Jacobian is nothing but the ratio of the area of the triangle in the physical space to the area of the right triangle in the parametric space. The area of the triangle in the physical space is estimated using the nodal coordinates and the Jacobian is explicitly calculated (lines 183-187). The value of Jacobian is computed as determinant of $\nabla\phi$. This check ensures that the linear map $\phi(\xi, \eta)$ and the derivative of the basis function are rightly coded in
- The test case of forcing function $f(x, y) = 0$ is a validation case, as the solution for this problem is zero everywhere in the domain as described in point 11.

(10)

The method to enforce boundary conditions adopted in this exercise is commonly referred to as, $application of displacement boundary conditions by modification$. This method preserves the order of all the equations and no rearrangement of the matrix vector system is needed which in itself is a time consuming process. Hence, in commercial finite element codes, the modification strategy is adopted for enforcing displacement boundary conditions for better efficiency.

The method works by setting the rows corresponding to the node with prescribed displacement to zero, except for the diagonal entry. This ensures the boundary condition is trivially satisfied. When the vector is multiplied with this modified stiffness matrix only the term $K_n n \times u_n$ survives which equal to the prescribed value. In this case the prescribed values are zero, hence no changes are reflected on to the force vector. In a general case of the the prescribed boundary conditions not being equal to zero, the contributions arising from the multiplication of the given row with the vector needs to be added to the corresponding entries of the force vector.

No, the symmetry of the stiffness matrix is **lost** after the application of the constraints.

(11)

For the case with $f(x, y) = 0$, the solution over the entire domain has to be zero. The code reproduces the same observation as seen in figure 3. A mechanical analogy is to imagine a thin unit square plate fixed on all the boundaries and if there are no external forces acting on the body (including gravity), the plate will not deform and will have zero displacements everywhere over the plate.

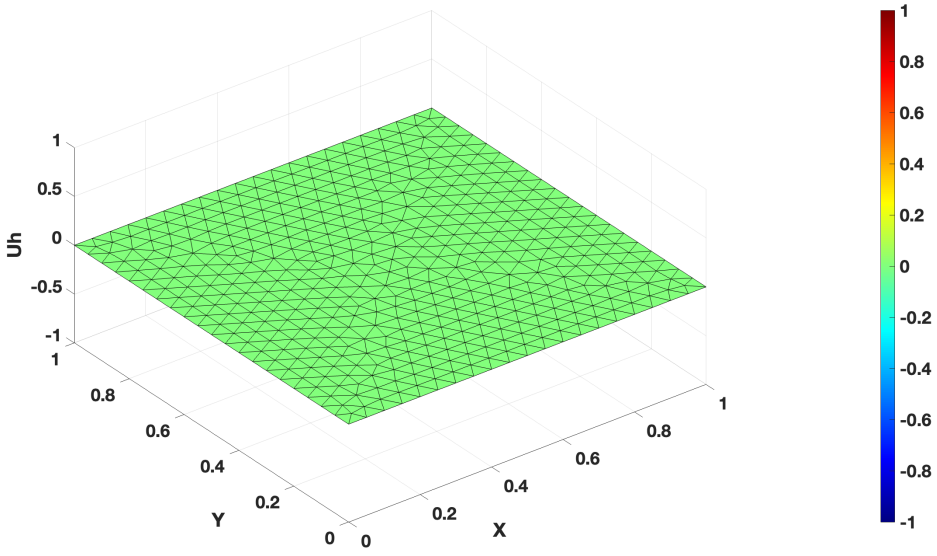


Figure 3: Solution for $f(x, y) = 0$

(12)

Yes, it is possible to choose $f(x, y)$, such that the finite element solution coincides with the exact solution. The finite element solution is a piece wise linear solution defined over the domain and if $f(x, y)$ is constructed in such a way that is also piece wise line over Ω, the error would be zero.

The code was tested for different possibilities of f and the predicted solutions are shown in figure 4 - 7. The mesh size for these runs was 0.05. The predicted solutions were compared with solutions provided by MATLAB and both the solutions were nearly identical.

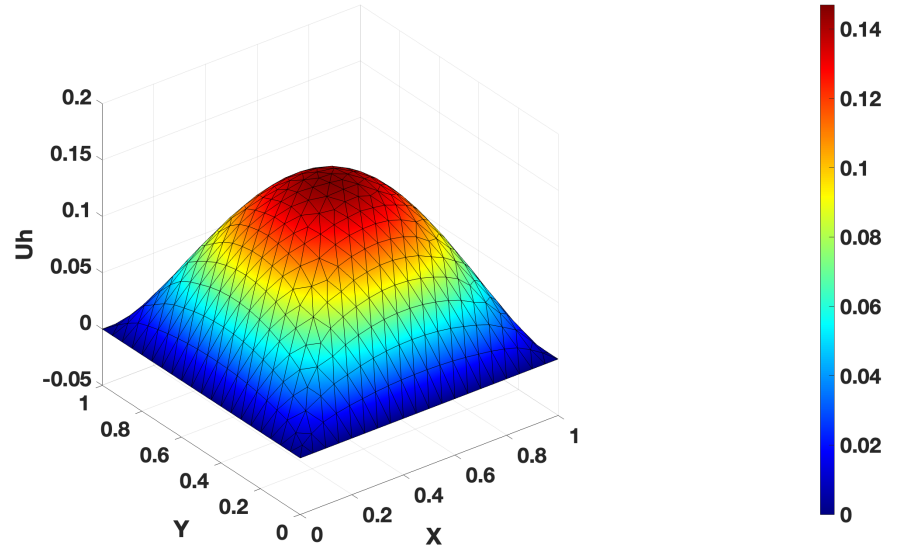


Figure 4: Solution for $f(x, y) = 2$

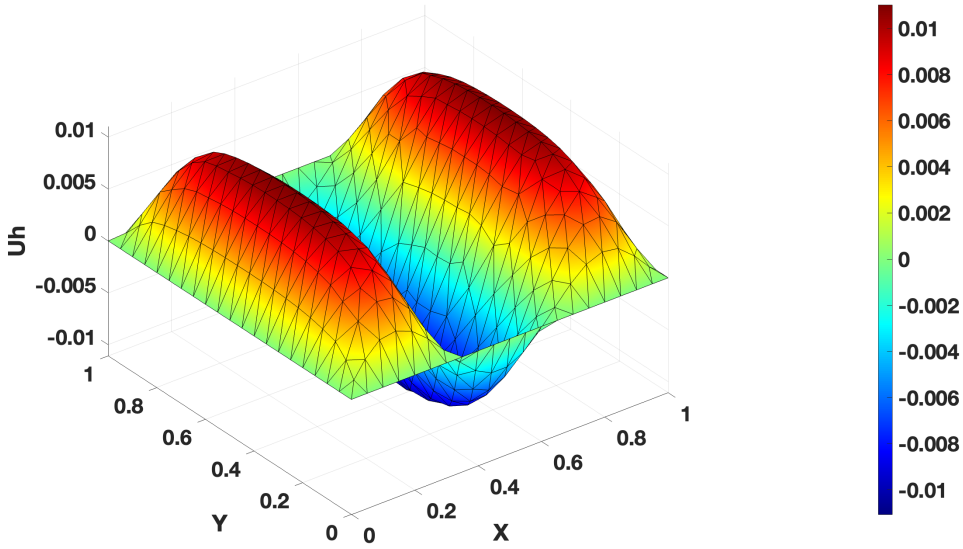


Figure 5: Solution for $f(x, y) = sin(3\pi x)$

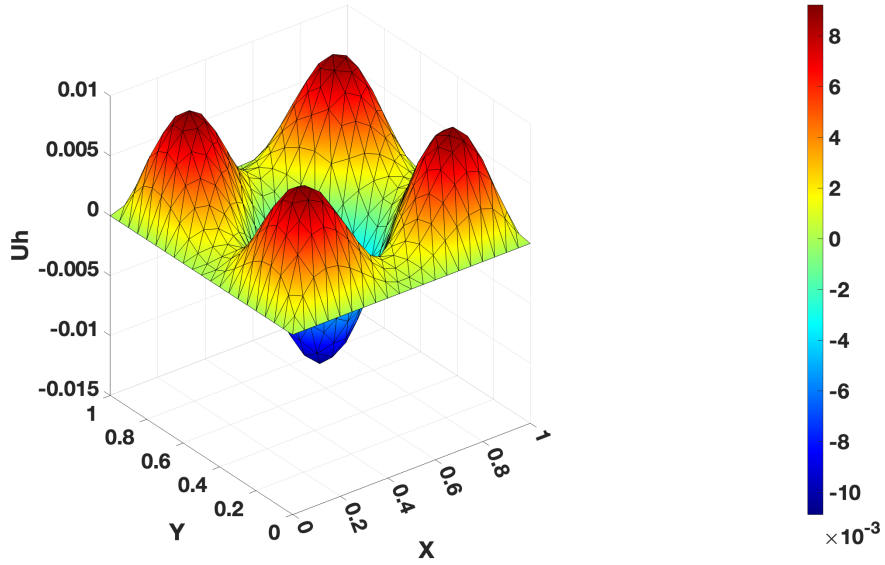


Figure 6: Solution for $f(x, y) = sin(3\pi x) + sin(3\pi y))$

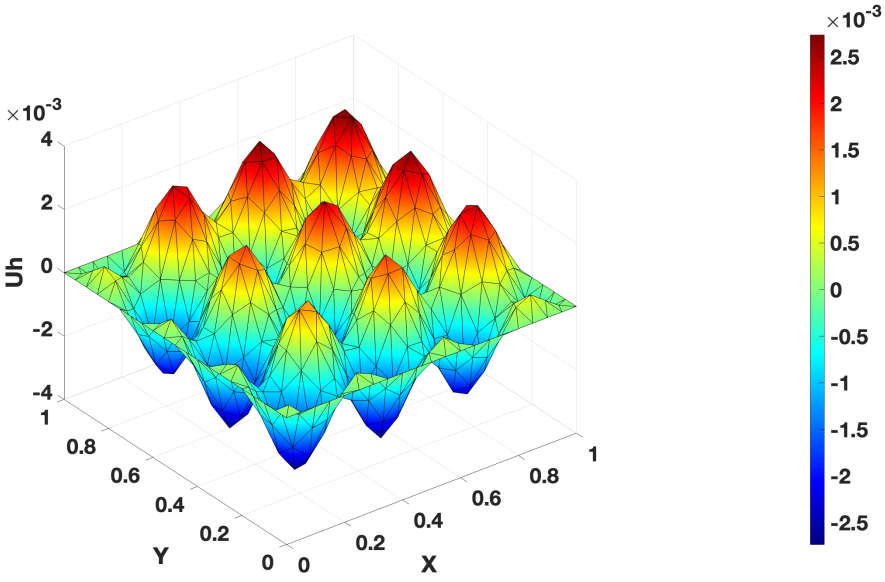


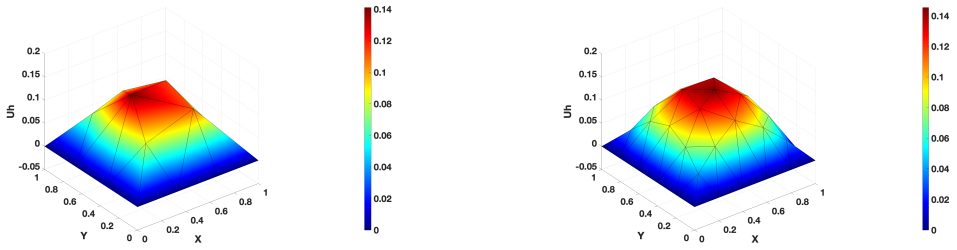
Figure 7: Solution for $f(x, y) = cos(7\pi x) + cos(7\pi y))$

(13)

Analytical solution is available for a simple 2D heat diffusion condition where the forcing function $f(x, y)$ is constant heat flux over the domain Ω. The forcing function $f(x, y)$ is taken as 2.0. The domain Ω is $[0, 1]^2$ with homogeneous boundary conditions on $d\Omega$. This case is chosen by setting parameter `FunctionIdentifier` equal to 1.

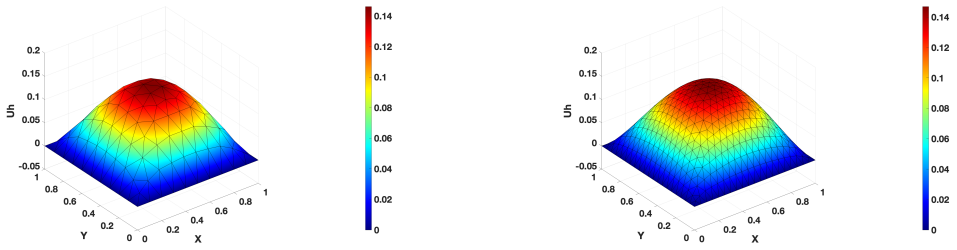
The analytical solution for this problem is given by the following relation [1]. In this exercise the summation is done upto 50 terms. The solution was estimated starting with a mesh size of 0.4 and halved subsequently. Figure 8 shows the evolution of the solution with mesh refinement.

$$u(x,y) = (1-x)x - \frac{8}{\pi^3}\sum_{n=1}^{\infty}\left[\frac{sinh[(2n-1)\pi(1-y)]+sinh[(2n-1)\pi y]}{sinh[(2n-1)\pi]}\right]\frac{sin[(2n-1)\pi x]}{(2n-1)^3}$$



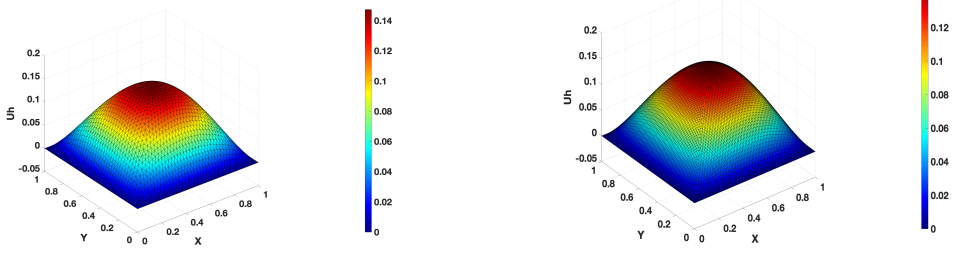
(a) Mesh size = 0.4

(b) Mesh size = 0.2



(c) Mesh size = 0.1

(d) Mesh size = 0.05



(e) Mesh size = 0.025

(f) Mesh size = 0.0125

Figure 8: Evolution of solution for $\Delta u + 2 = 0$ for different mesh sizes

(14)

The error between the analytical solution and the finite element approximation was calculated as defined in the problem statement. Table 2 gives the error for different mesh sizes.

Table 2: Error estimate for model problem $\Delta u + 2 = 0$ as a function of mesh size

Mesh size	Error
0.4	2.6726×10^{-3}
0.2	1.1105×10^{-3}
0.1	2.2673×10^{-4}
0.05	4.8852×10^{-5}
0.025	9.2208×10^{-6}
0.0125	1.7242×10^{-7}

Figure 9 show the variation of the error with the mesh size and as expected the error reduces with smaller mesh size.

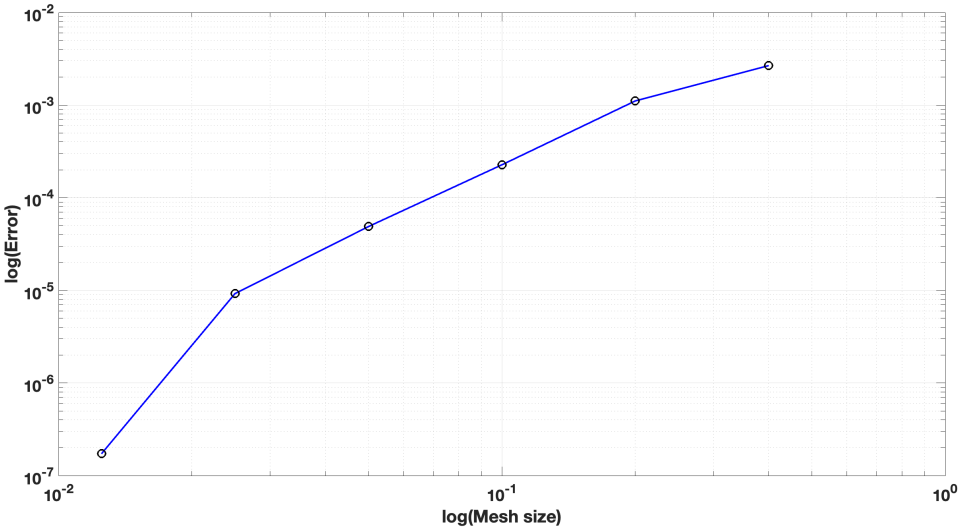


Figure 9: Variation of error with mesh size

References

[1] Kuhnert J. and Tiwari S., Grid free method for solving the Poisson equation, Fraunhofer Institut Techno-und Wirtschaftsmathematik, 2001.

——— o ——— o ———