PoiSolver

PoiSolver is a Finite Element Solver written in R for Poisson equation. PoiSolver employes piecewise linear basis functions on uniform triangulation. It solves a Poisson equation on a two dimensional rectangular domain with homogeneous boundary conditions of Dirichlet type. We illustrate PoiSolver through an example in which we present both the relevant theory of the finite element method a long with the implementation for a given example.

Finite element method overview

Consider a two dimensional open bounded convex (https://en.wikipedia.org/wiki/Convex_set) domain denoted by $\Omega \subset \mathbb{R}^2$ on the positive real $x-y$ plane. Let $u(x,y)$ satisfy the probelm $\Delta\, u(x,y) = f(x,y)$ in Ω and $u(x,y) = 0$ on $\partial\Omega$. Let the solution and test spaces be denoted by $H^1_\Omega = \{w \in H^1 : \|w(x,y)\|^2 < \infty,\ \|\nabla w(x,y)\|^2 < \infty\}$ and $H^1_{\Omega_0} = \{w \in H^1 :\ w(x,y) = 0,\ \ x,y \in \partial\Omega\}$ respectively, where H^1_Ω and $H^1_{\Omega_0}$ are a special class Sobelov spaces (https://en.wikipedia.org/wiki/Hilbert_space). Assuming that the problem satisfies all the conditions stated by Lax-Milgram Theorem (http://mathworld.wolfram.com/Lax-MilgramTheorem.html), we multiply both sides of the equation by a test function $w(x,y) \in H^1_{\Omega_0}$ and integrate by parts with application of Green's Theorem (https://en.wikipedia.org/wiki/Green%27s_theorem), then the weak formulation of the problem is to find $u \in H^1_\Omega$ such that $\int_\Omega \nabla u \cdot \nabla w\, dxdy = \int_\Omega fw\, dxdy$ for all $w \in H^1_{\Omega_0}$. Given that the problem is well-posed (https://en.wikipedia.org/wiki/Well-posed_problem) in Hilbert space of functions of which both H^1_Ω and $H^1_{\Omega_0}$ are subsets, then we can define finite dimensional solution and test spaces namely $V^h \subset H^1_\Omega$ and $V^h_0 \subset H^1_{\Omega_0}$ respectively. Let Ω^h be a triangulated approximation of Ω, containing N discrete number of nodes and K triangles, then using the finite dimentional solution and test spaces, the finite element formulation of the problem is to find $u^h \in V^h$ such that $\int_{\Omega^h} \nabla u^h \cdot \nabla w^h\, dxdy = \int_{\Omega^h} fw^h\, dxdy$ for all $w^h \in V^h_0$. We expand u^h and w^h in terms of its finite i.e. N dimensional basis functions (https://en.wikipedia.org/wiki/Basis_function) in the form $u^h = \sum_{i=1}^N U_i\phi_i$ and $w^h = \sum_{j=1}^N \phi_j$. Substituting these in the finite element formulation we obtain a discrete set of linear equations of the from $S\,U = L$, where S is the stiffness matrix, U is the vector of unknowns containing the approximate solution values at each node and L is the right hand-side load vector. The entries of S and L are given by $[S]_{i,j} = \int_{\Omega^h} \nabla\phi_i \cdot \nabla\phi_j\, dxdy$, $[L]_j = \int_{\Omega^h} f\phi_j\, dxdy$. The entries of U are the finite element approximate solution values to be found from the linear system. For code implementation we start with a scheme to descritise Ω. This is achieved by creating an $N \times N$ grid points within Ω, which provides a uniform quadrilateral mesh. Then we triangulate the quadrilateral mesh by drawing a line of slope -1 diangonally through each square in order to divide every quadrilateral element into two adjacent triangles, which leads to the construction of a uniform triangulated domain Ω^h. Let $\mathcal{T}$ denote the uniform triangulation of Ω, consisting of triangles. The algorithm is programmed to compute locally on each triangle $K \in \mathcal{T}$, the entries of the matrix S and the load vector L. The entries of the local 3×3 matrix S and 3×1 vector L are constructed by the formulea $[S]_{i,j} = \sum_{K\in\mathcal{T}} \int_{\Omega^h} \nabla\phi_i \cdot \nabla\phi_j\, dxdy$ and $[L]_j = \sum_{K\in\mathcal{T}} \int_{\Omega^h} f\phi_j\, dxdy$.

Documentation of the code

We go through the implementation process for the problem $\Delta\, u(x,y) = 1$ that is posed on $\Omega = (0\ 1) \times (0\ 1)$. It satisfies homogeneous Dirichlet type boundary conditions of the form $u(x,y) = 0, \quad \forall (x,y) \in \partial\Omega$. The first few lines of the code mainly assign numerical values for variables to be used. The variables that user may require to interfere with in using `PoiSolver.R` code are `L`, `N` and `F`,

which respectively stor the values for the side length of the two dimensional rectangular Ω, number of mesh points that discretises `L` and finally the local contributions from the right hand-side expression of the equation that are stored in `F`. The documentation is presented in the form of enumerated blocks of code.

1. The variable

```
L = 1
```

   stores the value for the side length of Ω.

2. The current code is implemented so that it only works for rectangular domains that have the same side lengths i.e. square only, therefore, the variable

```
N = 50
```

   is the mesh refinement controlling parameter i.e. it is the number of points that discretises `L`.

3. The line

```
X = seq(0,L,len=N+1)
```

   is equivalent to `linspace` function in MATLAB. It outputs the actual values of the `N+1` grid points.

4. We can creat a quadrilateral grid that consists of N^2 points each of which is equipped with the numerical values of (x, y), for the coordinates in two matrices namely `x` and `y`, which were obtained using the following built-in functions of R.

```
m = length(X); n=length(X);
x = matrix(rep(X,each=n),nrow=n);
y = matrix(rep(X,m),nrow=n)
```

5. Now we reshape the matrices x and y into column vectors, which is accomplished by

```
x = c(x)
y = c(y)
```

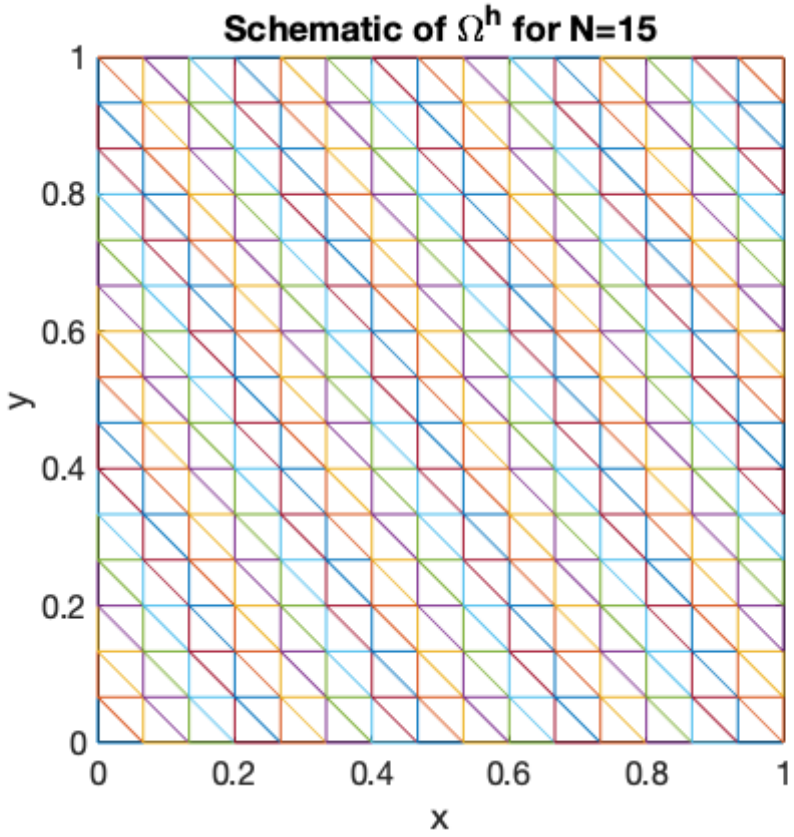
6. The variables `GNodes` is meant to stand for Global Nodes and it is instantiated in terms of `N`. The variable `NumTRI` stores the number of triangles that a discretisation of this type outputs. If we have N^2 quadrilateral elements and each square is divided by two triangles we obtain $2 \times N^2$ triangles. The array `LocNodes` stores the connectivity array of the vertices of all the triangles, hence it is a matrix with `NumTRI` rows and 3 columns.

```
GNodes = (N+1)^2
NumTRI = 2*N^2
LocNodes = matrix(0,NumTRI,3)
```

7. Triangulation of the domain is obtained such that the vertices of each triangle is locally counted in anti-clockwise orientation and such that the local counting of vertices of each triangle fills the global connectivity array `LocNodes` in the correct order. Note that we must fill in `4+2=6` vertices for each square, because each square is divided into two triangles by a diagonal line, however, each triangle has one distinct but two shared vertices. Therefore, it can be noted that the first three lines inside the nested for loop correspond to the three vertices of all the lower triangles on each square and the latter three lines correspond to all the three vertices of all the upper triangles. The nested loop that distributes such order of counting within the connectivity array is given by

```
for (i in 1:N){
  for (j in 1:N){
LocNodes[i+2*(j-1)*N,1] = i+(j-1)*(N+1)
LocNodes[i+2*(j-1)*N,2] = i+j*(N+1)
LocNodes[i+2*(j-1)*N,3] = (i+1)+(j-1)*(N+1)
LocNodes[i+N+2*(j-1)*N,1] = i+1+j*(N+1)
LocNodes[i+N+2*(j-1)*N,2] = (i+1)+(j-1)*(N+1)
LocNodes[i+N+2*(j-1)*N,3] = i+j*(N+1)
}
}
```

This piece of code returns a triangulated Ω^h, that looks like



however, the instruction that plots this particular Figure is not included in the script, since the graphical output of the mesh in this case scenario does not form the essence of the objectives. The Figure is here purely for visualisation of the mesh.

8. We need to introduce two types of global arrays namely a system matrix `Sparsity` and a right hand-side vector `LoadVect`, which must be of size $(N+1)^2 \times (N+1)^2$ and $(N+1)^2 \times 1$ respectively. These are introduced by

```
Sparsity <- matrix(0, GNodes, GNodes)
LoadVect = matrix(0, GNodes, 1)
```

9. The next segment of code is a for loop that executes a series of computations on each one of the triangles. It starts with

```
for (n in 1:NumTRI)
  {# This is the start of the for loop on all triangles
```

- The first part within the loop defines the local position vectors `r1`, `r2`, and `r3` in terms of `x` and `y` coordinate values through which we refer to different vertices of each triangle.

```
  r1 = matrix(c(x[LocNodes[n,1]],y[LocNodes[n,1]]),nrow=2,byrow=FALSE)
  r2 = matrix(c(x[LocNodes[n,2]],y[LocNodes[n,2]]),nrow=2,byrow=FALSE);
  r3 = matrix(c(x[LocNodes[n,3]],y[LocNodes[n,3]]),nrow=2,byrow=FALSE);
```

- The second part within the loop defines a 2×2 Jacobian matrix `J` for the mapping from an arbitrary triangle in Ω^h to a reference triangle that has vertices in order $(0,0),(1,0)$ and $(0,1)$.

```
  J = matrix(c(r2[1]-r1[1],r2[2]-r1[2],r3[1]-r1[1],r3[2]-r1[2]), nrow=2, byrow=T
RUE)
```

  The Jacobian of the mapping namely `J` serves to reduce the computational cost by a significant amount, particularly due to a property of integration for computing the integral of a function on a reference domain with a given mapping between the arbitrary domain and the reference domain. Further details on this topic can be found on Integral domain transformation (http://www.iue.tuwien.ac.at/phd/nentchev/node58.html).
- The variable name `Astiff` is the local 3×3 stiffness matrix, which is computed and looped over all the triangles.

```
Astiff = (1/(2*det(J)))*matrix(c(sum((r2-r3)*(r2-r3)),sum((r2-r3)*(r3-r1)),sum
((r2-r3)*(r1-r2)),
sum((r2-r3)*(r3-r1)),sum((r3-r1)*(r3-r1)),sum((r3-r1)*(r1-r2)),
sum((r2-r3)*(r1-r2)),sum((r3-r1)*(r1-r2)),sum((r1-r2)*(r1-r2))),
nrow=3,byrow=TRUE);
```

- A nested loop (within the main loop over all triangles) is required to place the contributions from all the local stiffness matrices `Astiff` into the correct position in terms of entries of global stiffness matrix i.e. into the `Sparsity`. This is coded as

```
for (i in 1:3){
for (j in 1:3){
  Sparsity[LocNodes[n,i],LocNodes[n,j]]=Sparsity[LocNodes[n,i],LocNodes[n,j]]+As
tiff[i,j]
}
  }
```

- We need to introduce the coordinate names for axes on reference triangle, which are required to be equipped with a transformation (mapping) formula between variables (`x` , `y`) to variables (`xx` , `yy`). The values of the variables `ksi` and `eta` are related to quadrature formula. This mapping is given by

```
ksi = 1/3;
eta = 1/3;
xx = (1-ksi-eta)*r1[1]+ksi*r2[1]+eta*r3[1];
yy = (1-ksi-eta)*r1[2]+ksi*r2[2]+eta*r3[2];
```

- We further need to calculate the contribution of the local load vector `F` and then for the value to be correctly positioned as an entry of the global load vector namely `LoadVect`. The following lines provide the code to accomplish this. Note that for any function on the right hand-side of the equation (other than 1 in the current example) this is where the code will need changing, because this is where the local load vector is constructed.

```
F[1] = (1-ksi-eta)*det(J)*1/2;
F[2] = (ksi)*det(J)*1/2;
F[3] = (eta)*det(J)*1/2;
for (i in 1:3){
  LoadVect[LocNodes[n,i]] = LoadVect[LocNodes[n,i]]+ F[i]
  }
# This is the end of the for loop on all triangles
}
```

10. Next we need to implement the prescribed boundary conditions of Dirichlet type. We impose the boundary conditions through placing instructions on the entries of `Spartsity` and `LoadVect`. Note that the `if` statement explicitly specifies the nodes for which either `x,y = 0` or `x,y = L` must be treated as the boundary node. The code to implement boundary condition is given by

```
for (i in 1:GNodes){
if (x[i]==0 | y[i]==0 | x[i]==L | y[i]==L){
  LoadVect[i] = 0
  Sparsity[i,] = 0
  Sparsity[i,i] = 1
 }
}
```

11. Finally the line

```
U = solve(Sparsity,LoadVect)
```

solves the linear system and stores the values of the numerical solution at each node in the variable `U`.

12. The next step is to write and store the numerical solution `U` and other files that are necessary for visualisation of the solution. These consist of a vector storing the coordinates of x axis, a vector that stores coordinates of y axis and a connectivity array that is usually essential for triangular surface plots for visualisation. These four lines of code will write and store data files namely `xcoordinates.txt`, `ycoordinates.txt`, `Solutions.txt` and `triangles.txt`, of which the first three are column vectors of the same dimension and the fourth one `triangles.txt` stores the connectivity array. All files are written as text files and stored in the current directory.

```
write.table(x,file="./xcoordates.txt",row.names=FALSE,col.names=FALSE)
write.table(y,file="./ycoordates.txt",row.names=FALSE,col.names=FALSE)
write.table(U,file="./Solutions.txt",row.names=FALSE,col.names=FALSE)
write.table(LocNodes,file="./triangles.txt",row.names=FALSE,col.names=FALSE)
```

Visualisation on MATLAB

The current instant of the output produced by PoiSolver is plotted using MATLAB and the four text files of data that are produced by PoiSolver are called from within a MATLAB script called `plotsolution.m`. The first four lines of this script

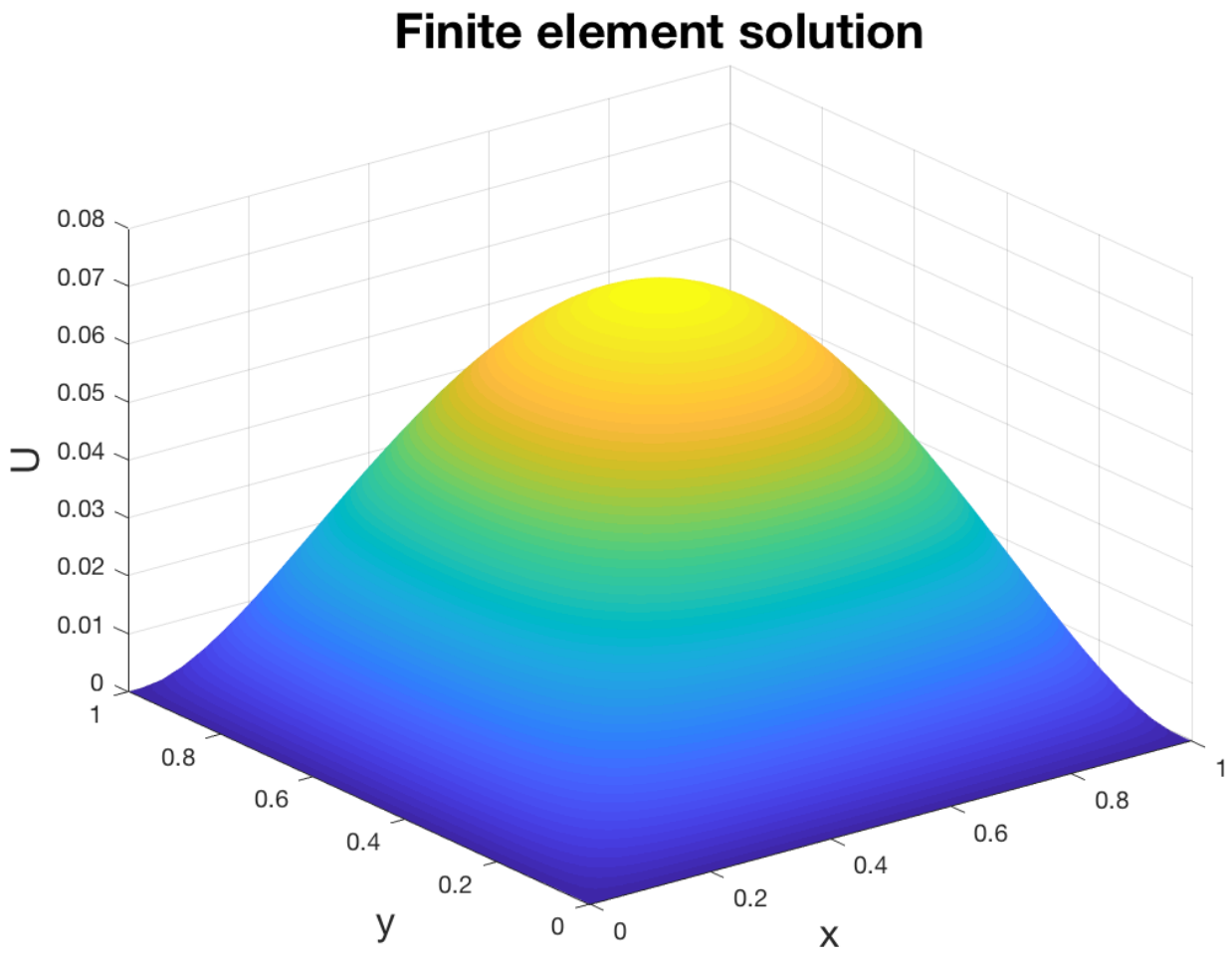
```
coordx = importdata('xcoordates.txt');
coordy = importdata('ycoordates.txt');
triangles = importdata('triangles.txt');
numsol = importdata('Solutions.txt');
```

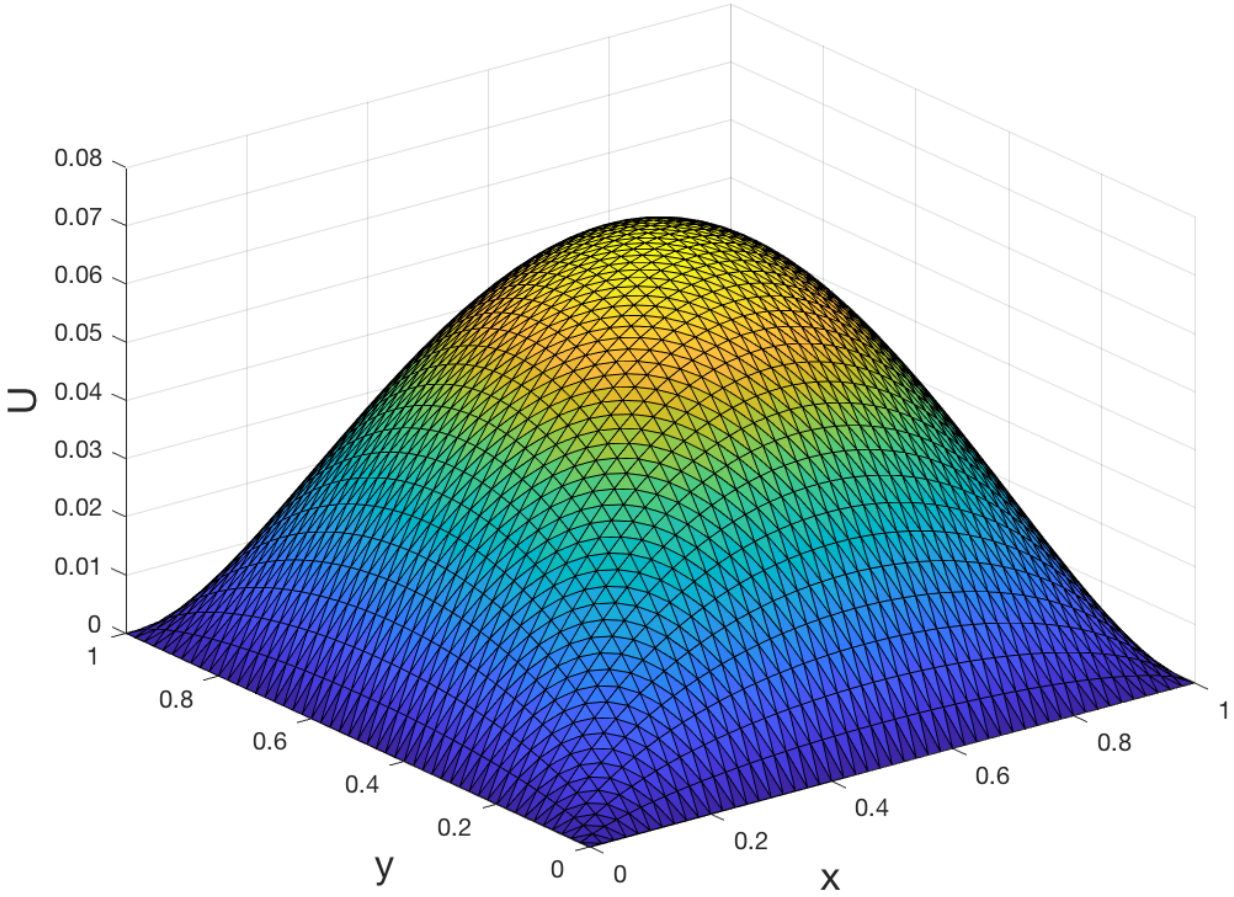
import the data from the output files of `PoiSolver.r`. Then the in-built MATLAB function `trisurf` can be used to visualise the solution using these lines of code.

```
figure(1)
trisurf(triangles,coordx,coordy,numsol)
shading interp
title('Finite element solution','fontsize',20)
xlabel('x','fontsize',16)
ylabel('y','fontsize',16)
zlabel('U','fontsize',16)
```

Note that `shading interp` removes the lines of triangulation from the surface plot. If it is desired to see the triangulation line on the surface plot, then it can be commented out using `%`. The first and second Figures below respectively show the solution surfaces without the triangulation lines appearing on the plot and with

the triangulation lines appearing on the plot.




Finite element solution
U
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0
y
1
0.8
0.6
0.4
0.2
0
x
0
0.2
0.4
0.6
0.8
1