2D Unsteady Heat Equation Simulation

Rohan Krishna Balaji Maximilian Schuster

Laboratory for Parallel Programming
rohan.balaji@rwth-aachen.de schuster@cats.rwth-aachen.de

1 Introduction to Finite Element Methods

The finite element method (FEM) as described in [1] is a numerical technique for solving partial differential equations (PDE's).Its first essential characteristic is that the continuum field, or domain, is subdivided into cells, called elements, which form a grid. The elements (in 2D) have a triangular or a quadrilateral form and can be rectilinear or curved.

The second essential characteristic of the FEM is that the solution of the discrete problem is assumed a priori to have a prescribed form. The solution has to belong to a function space, which is built by varying function values in a given way, for instance linearly or quadratically, between values in nodal points. The nodal points, or nodes, are typical points of the elements such as vertices, mid-side points, midelement points, etc. Due to this choice, the representation of the solution is strongly linked to the geometric representation of the domain.

The third essential characteristic is that a FEM does not look for the solution of the PDE itself, but looks for a solution of an integral form of the PDE. The most general integral form is obtained from a weak formulation. By this formulation the method acquires the ability to naturally incorporate differential type boundary conditions and allows easily the construction of higher order accurate methods. The ease in obtaining higher order accuracy and the ease of implementation of boundary conditions form a second important advantage of the FEM.

2 Description of the Problem

Here in this Project we consider a disc, with three types of meshes, namely Coarse, Fine and Finest. The aim of this project is to solve the heat diffusion equation in 2D to obtain the temperature distribution in a plane disk heated by a localized heat source.This heat source is assumed to be acting only on a subset of the domain represented by a small disk of radius 0.01. The disk has a radius, R2 = 0.1m. A fixed temperature, To= 300K is set on the disk boundary, and a heat source of total power P = 1W is applied on a small circular area (radius R1 = 0.01m) centered at the origin.

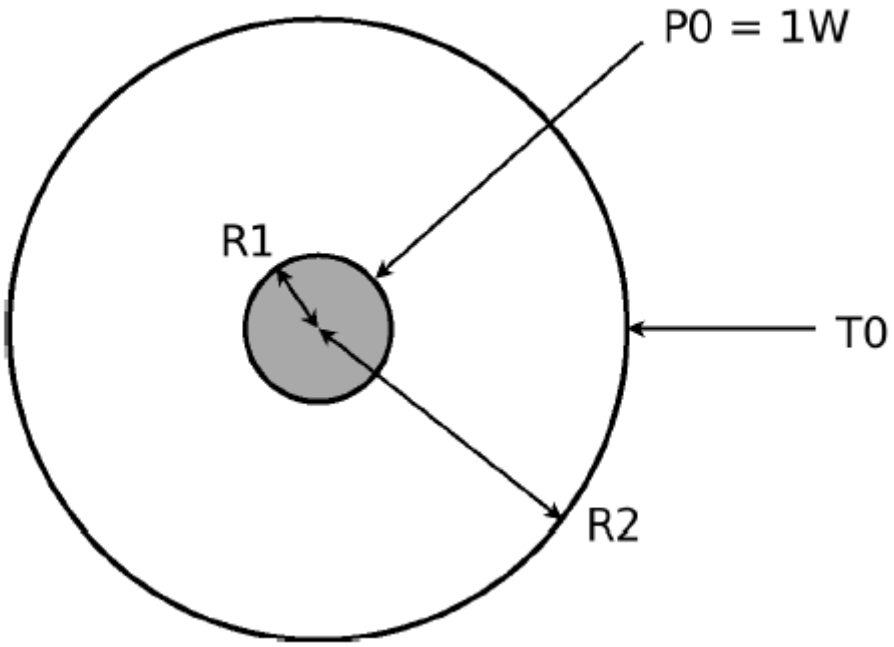

Fig1: Geometry and Boundary Conditions

The Analytical solution to this problem is given by

$$T(r) = \begin{cases} T_0 - \frac{Q}{2\pi\alpha}(\frac{1}{2}(\frac{r^2}{R_1^2}) + \ln\frac{R_1}{R_2}) & r < R_1 \\ T_0 - \frac{Q}{2\pi\alpha}(\ln\frac{r}{R_2}) & r \geq R_1 \end{cases} \tag{1}$$

3 Method

In our simulation problem we solve the unsteady heat difussion equation formulated as

$$\begin{cases} \frac{\partial T}{\partial t} - \alpha\nabla^2 T = f & disc\ domain \\ T = T_o & on\ boundary \end{cases} \tag{2}$$

T, α, and f are respectively the temperature, the thermal diffusivity, and the heat source. The heat source changes based on the location of the node:

$$f(r) = \begin{cases} \frac{Q}{\pi R_1^2} & r < R_1 \\ 0 & r \geq R_1 \end{cases} \tag{3}$$

$Q = P/d_z$ is the volumetric heat source. dz = 0.1m is the out-of-plane thickness. As described in [2] and prolem statement we need to first find the weak formulation for the given problem, which is obtained be muliplying the test function to the PDE and integrating over the whole domain. Further we apply Gauss divergence theorem to

convert volume integral to the surface integral.

$$\int_{\Omega} w\frac{\partial T}{\partial t}d\Omega + \int_{\Omega} \alpha \nabla w \nabla T d\Omega = \int_{\Omega} w f d\Omega \tag{4}$$

Where is the computational domain. Because the problem in question has only a Dirichlet boundary condition, the Neumann boundary term arising from the divergence theorem is zero. Using the Galerkin finite element method, the following space discretization in the Matrix form can be obtained, With [M] is called the mass matrix and [K] the stiffness matrix. F is the source vector.

$$[M]\dot{T} + [k]T = F \tag{5}$$

the added C++ code for the element matrix computation, including mass lumping is shown here,

Listing 1: Element Matrices Calculations

```
1  void femSolver::calculateElementMatrices(const int e)
2  {
3      // Add code here
4    //M and K matrix
5   double M[nen][nen];double K[nen][nen];double F[nen];
6   for(int i=0;i<nen;i++)
7     {
8       F[i]=0;
9       for(int j=0;j<nen;j++)
10        {
11      M[i][j]=0;      //initialisation of M and K matrices
12      K[i][j]=0;
13        }
14    }
15    double f=31830.9886183;
16    for(int m=0;m<nGQP;m++)
17      {
18        for(int i=0;i<nen;i++)
19      {
20        for(int j=0;j<nen;j++)
21        // Calculating the M ang K matrices using getter functions
22          {
23
24            M[i][j]+= (mesh->getME(m)->getWeight())*(mesh->getME(m)->getS(i))
25            *(mesh->getME(m)->getS(j))*abs((mesh->getElem(e)->getDetJ(m)));
26            K[i][j]+= (settings->getD())*(mesh->getME(m)->getWeight())
27            *((mesh->getElem(e)->getDSdX(m,i))*
28            (mesh->getElem(e)->getDSdX(m,j))+(mesh->getElem(e)->getDSdY(m,i))
```

```
29              *(mesh->getElem(e)->getDSdY(m,j)))
30              *abs((mesh->getElem(e)->getDetJ(m)));
31            }
32
33       }
34       }
35    /* F Matrix*/
36    double * xyz= mesh->getXyz();
37    double x[nen]; double y[nen]; int myNode;
38    for(int i=0; i<nen;i++)          //initialising x and y
39      {
40        x[i]=0;
41        y[i]=0;
42      }
43   for(int m=0;m<nGQP;m++)
44     {
45        for(int i=0;i<nen;i++)
46      {
47        myNode= mesh->getElem(e)->getConn(i);
48  //Calculating F matrix if the distance /radius is less than 0.01
49        x[i]= xyz[myNode*nsd+xsd];
50        y[i]= xyz[myNode*nsd+ysd];
51        if(sqrt((x[i]*x[i])+(y[i]*y[i]))<0.01)
52          {
53          F[i]+=f*(mesh->getME(m)->getWeight())*
54          (mesh->getME(m)->getS(i))*
55          abs((mesh->getElem(e)->getDetJ(m)));
56          }
57        else
58          F[i]+=0;
59      }
60     }
61
62   double Me=0; double Mf=0;
63   for (int i=0;i<nen;i++)                  //mass LUMPING
64      {
65      for(int j=0;j<nen;j++)
66        {
67      Me+=M[i][j];
68        }
69      }
70      for(int j=0;j<nen;j++)
71        {
72      Mf+=M[j][j];
```

```
73          }
74        for (int i=0;i<nen;i++)
75          {
76        M[i][i]=M[i][i]*(Me/Mf);
77          }
78
79
80
81
82  //saving computed element matrices by looping over element nodes
83   for (int i=0;i<nen;i++)
84       {
85         myNode= mesh->getElem(e)->getConn(i);
86        for(int j=0;j<nen;j++)
87          {
88            mesh->getNode(myNode)->addMass(M[i][i]);
89            mesh->getElem(e)->setM(i,M[i][i]);
90            mesh->getElem(e)->setK(i,j,K[i][j]);
91            mesh->getElem(e)->setF(i,F[i]);
92          }
93       }
94       return;
95  }
```

After calculating the matrices and storing, next step would be to impose the boundary conditions (BC) for the given problem. Usually there are two major types of Boundary namely, Dirichlet and Neumann Boundary conditions . In out problem we only impose the Dirichelt BC, that is prescribing the Temperature to be 300K at radius of 0.1 dim from the center. That part of the code is given below

Listing 2: Imposing Dirchlet Boundary Condition

```
1   void femSolver::applyDrichletBC()
2   {
3       // Add code here
4      double * xyz= mesh->getXyz(); int nn=mesh->getNn();
5      int ne=mesh->getNe(); double * temp=mesh->getT();
6       double x; double y; int FG=1;
7       //Initialising the temp array
8     for(int i=0;i<nn;i++)
9       {
10        temp[i]=0;
11      }
12      //Traversing through the nodes and taking the x,y
13      //coordinates on the domain
```

```
14   for(int i=0;i<nn;i++)
15     {
16       x= xyz[i*nsd+xsd];
17       y= xyz[i*nsd+ysd];
18       //if the dist is approx 0.1 set the temperature of the
19       //node as needed
20       if(sqrt((x*x)+(y*y))>=0.1-1e-6 && sqrt((x*x)+(y*y))<= 0.1+1e-6 )
21         {
22           mesh->getNode(i)->setBCtype(1);
23           temp[i]=settings->getBC(FG)->getValue1();
24         }
25     }
26      return;
27 }
```

After storing the matrices and imposing boundary conditions next step would be to solve the system numerically. We do this by using First Order time integration scheme. Which results in,

$$[M]T^(n+1) = [M]T^n + \Delta t(F^n - [K]T^n) \tag{6}$$

Listing 3: Explicit Solver

```
1
2  void femSolver::explicitSolver()
3  {
4      // Add code here
5
6    int iter = settings->getNIter();
7    double dt = settings->getDt();
8    int ne = mesh->getNe();
9    int nn = mesh->getNn();
10   double * MTnew = mesh->getMTnew();
11   double * T = mesh->getT();
12   double * massG = mesh->getMassG();
13   double scale,res,gerror;
14   double lim = pow(10,-7);
15   int i;
16
17   for(i=1; i<=iter; i++)       //loop over iterations
18     {
19
20       for(int j=0; j<ne; j++)      // loop over Elements
21     {
22       double * M = mesh->getElem(j)->getMptr();
23       double * K = mesh->getElem(j)->getKptr();
```

```
24          double * F = mesh->getElem(j)->getFptr();
25
26          for(int k=0; k<nen; k++)      // calculate MTnew
27            {
28              int n = mesh->getElem(j)->getConn(k);      //global node number
29              double KT = 0.0;
30              double MT = 0.0;
31
32              for(int l=0; l<nen; l++)
33            {
34              int p = mesh->getElem(j)->getConn(l);      //global node number
35              KT = KT + K[k*nen+l]*T[p];
36            }
37
38              MT = M[k]*T[n];
39              MTnew[n] = MTnew[n]+ MT + dt*(F[k] - KT);
40              // MTnew for global node 'n'
41
42            }
43        }
44
45          gerror = 0.0;    // global error
46          int nd = 0;      // non-Dirichlet nodes
47
48          for(int j=0; j<nn; j++)     // loop over nodes
49        {
50          int type = mesh->getNode(j)->getBCtype();
51          if(type!=1)      //non-dirichlet node
52            {
53              nd = nd +1;
54              double Tnew = MTnew[j]/massG[j];      // new Temp at current node
55              double error = (Tnew-T[j])*(Tnew-T[j]);
56              gerror = gerror + error;
57              T[j] = Tnew;      // store new Temperature
58            }
59
60          MTnew[j]=0.0;     //make 0 for next outer loop iteration
61        }
62
63          gerror = sqrt(gerror/nd);      // compute global error
64
65          if(i==1)
66        {
67              scale = gerror;      // scaled value for 1st iteration
```

```
68          cout<<"Non-Dirichlet Nodes = "<<nd<<endl;
69        }
70          res = gerror/scale;       //rms error or residual
71          if(res<lim)
72        break;
73        }
74
75        cout<<"RMS Error = "<<res<<endl;
76        cout<<"Iterations run = "<<i<<endl;
77
78        return;
79  }
```

This particular problem also has an analytical solution as shown in method section, the node wise analytical values are computed and compared with the Finite Element numerical code. The difference is called Discretisation error, which is which is calculated as RMS value and the final error is printed.

Listing 4: Comparing with Analytical

```
1  void postProcessor::compareAnalytical()
2  {
3       int nn = mesh->getNn();
4       double * T = mesh->getT();
5       double * xyz = mesh->getXyz();
6       double x, y, radius, Ta;
7       // Add code here
8       double To=300;double  Q=10; double alpha= settings->getD();
9       double pi=3.14; double coeff;
10      double error=0;  x=0; y=0; radius=0;
11      //declaring and initialising req variables
12      coeff= (double)(Q/(2*pi*alpha));
13      //Loop over the nodes and
14      for(int i=0; i<nn;i++)
15        {
16      x= xyz[i*nsd+xsd];
17
18      y= xyz[i*nsd+ysd];
19      radius= sqrt((x*x +y*y));
20      Ta=0;
21      // Finding analytical temperature based on distace from center
22      if (radius< 0.01)
23        {
24       Ta= To-(coeff)*(0.5*(((double)(radius*radius)/0.01*0.01)-1)+log(0.1));
25        }
```

```
26      else if (radius>=0.01)
27        {
28          Ta= To-(coeff)*log(radius/0.1);
29        }
30      else
31        cout<<"error in code"<<endl;
32       //Calculating disceretzation error wrt to
33        //Analytical and Numerical temperature
34      error=error+ ((T[i]-Ta)*(T[i]-Ta));
35        }
36      error= sqrt((double)(error)/nn);
37      cout<<" Discretization Error= "<<error<<endl;
38
39      return;
40  }
```

4 Results

4.1 Coarse Mesh

Tmin=300 K
Tmax=303.42 K
Discretization error Coarse= $3.4524 * 10^{-1}$

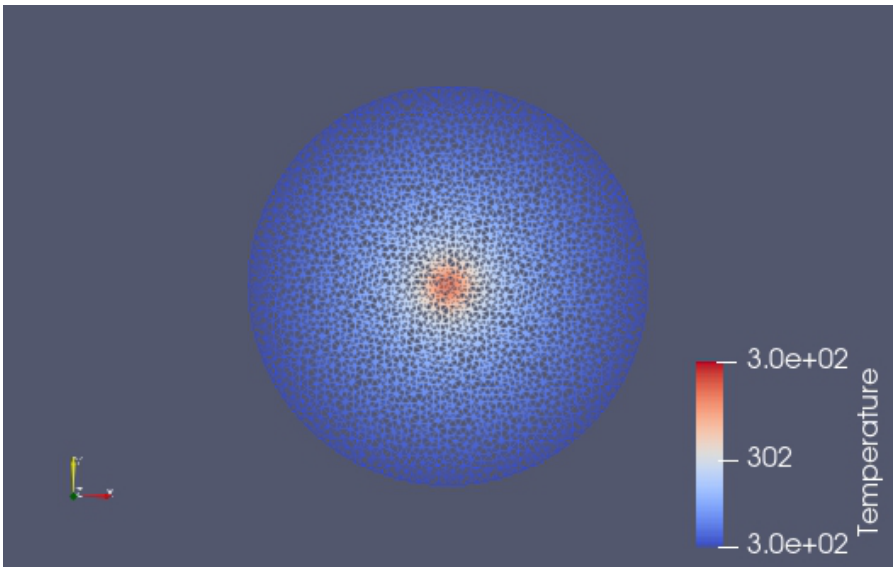

Fig:2 Temperature Distribution on Disk

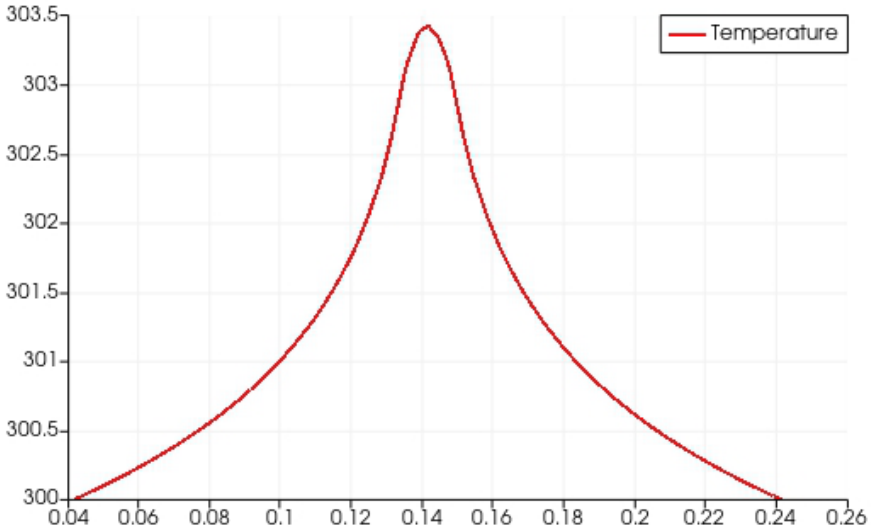

Fig:3 Temperature Distribution Graph

4.2 Fine Mesh

Tmin=300 K
Tmax=304.02 K
Discretization error Coarse= $1.481 * 10^{-1}$

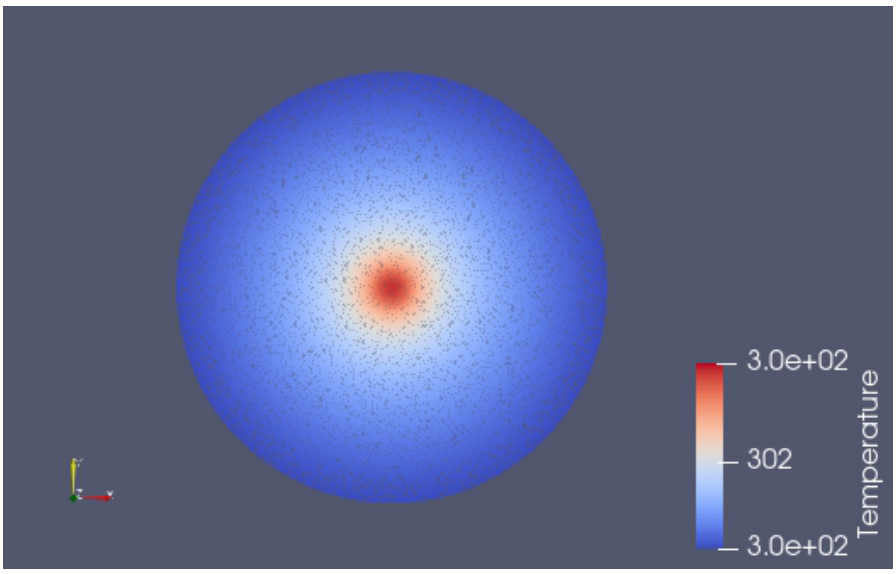

Fig:4 Temperature Distribution on Disk

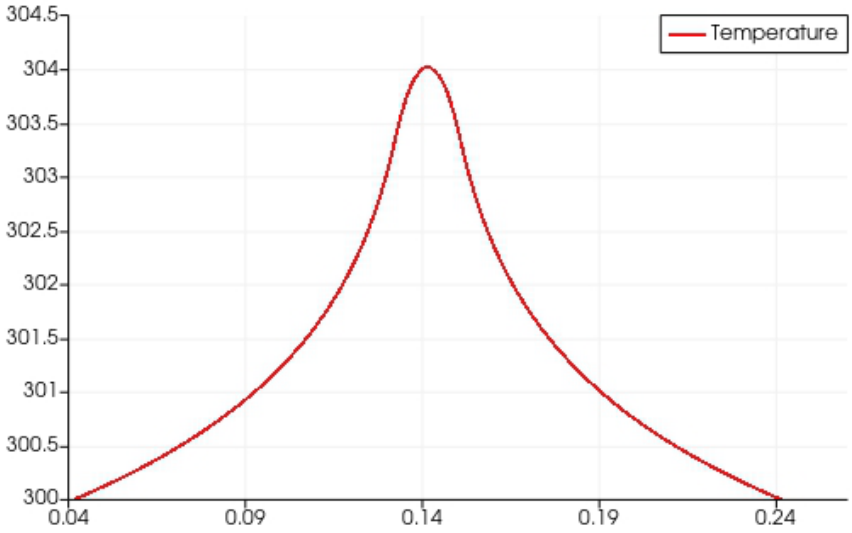

Fig:5 Temperature Distribution Graph

4.3 Finest Mesh

Tmin=300 K
Tmax=304.24 K
Discretization error Coarse= $9.2814 * 10^{-2}$

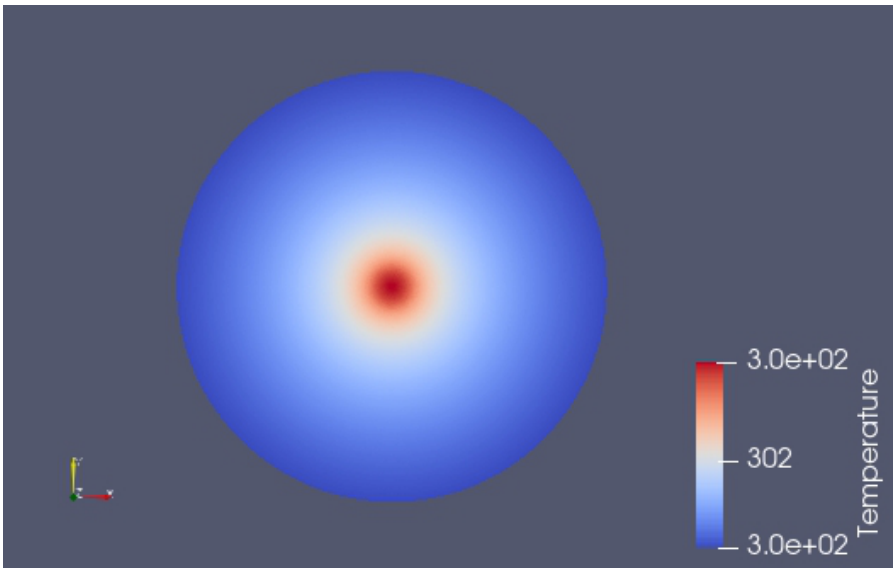

Fig:6 Temperature Distribution on Disk

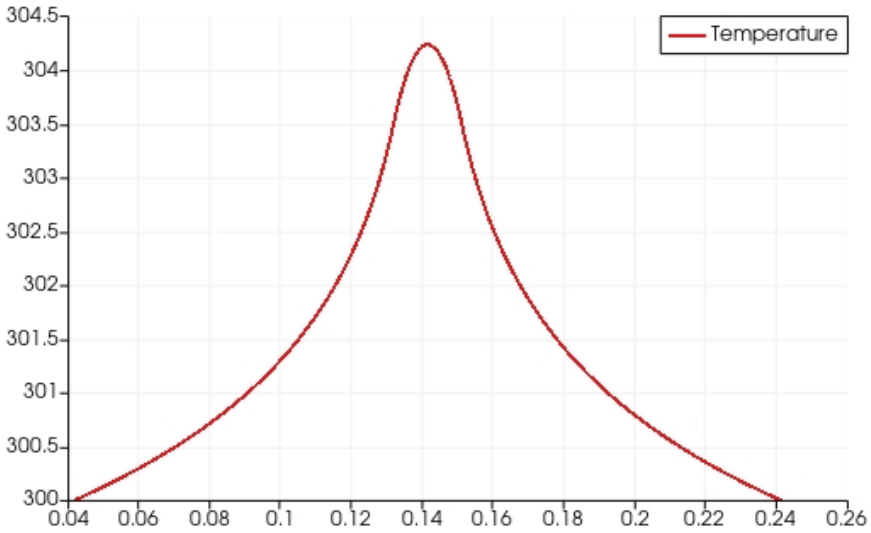

Fig:7 Temperature Distribution Graph

4.4 Errors

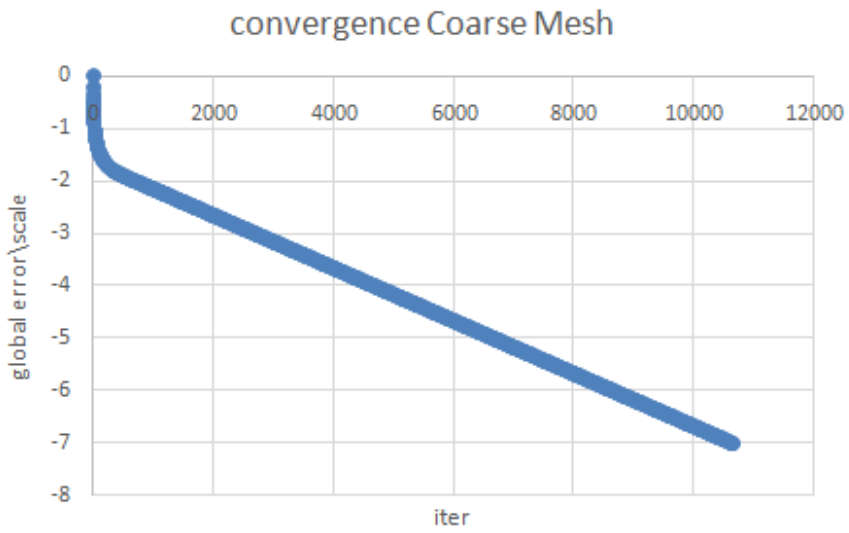

Fig:8 Num of iter vs log(Global error/scale)

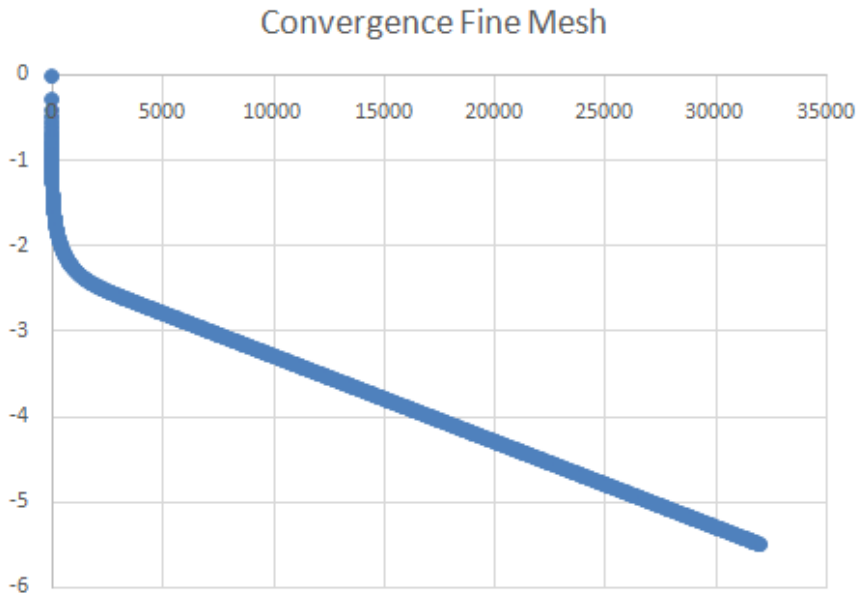

Fig:9 Num of iter vs log(Global error/scale)

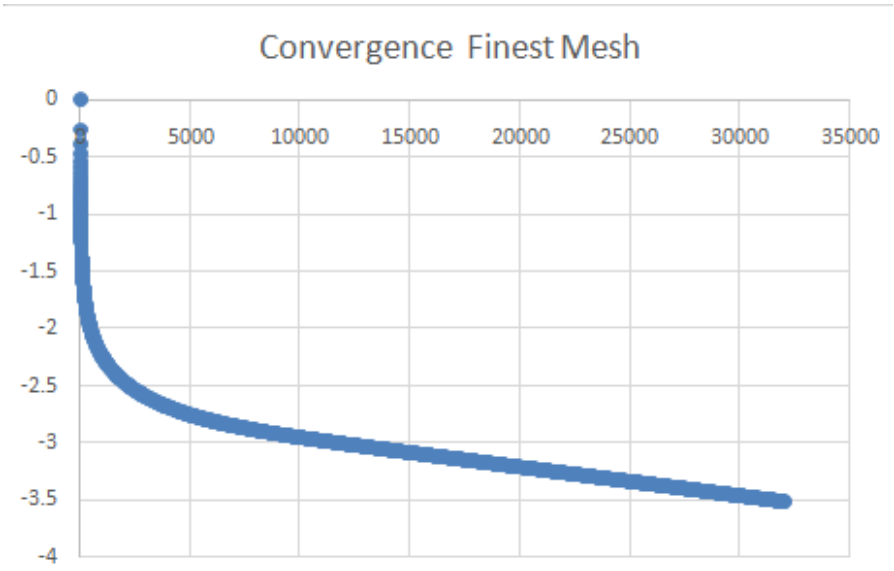

Fig:10 Num of iter vs log(Global error/scale)

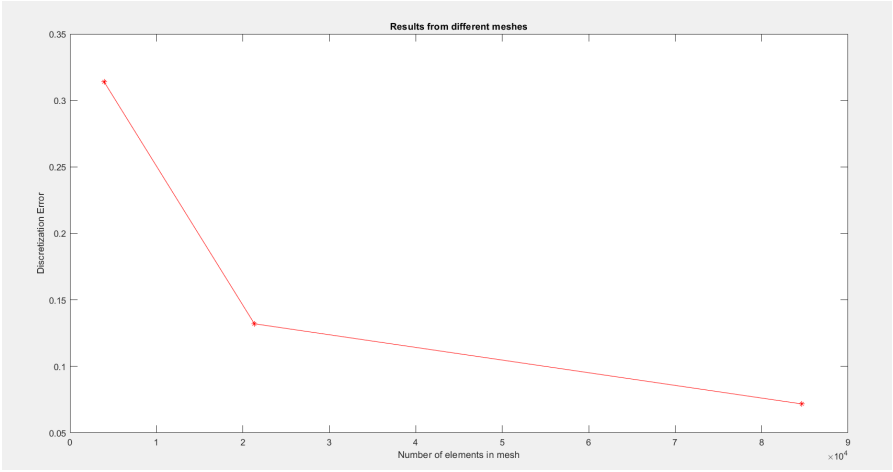

Fig:11 Num of elements vs Discretization error)

From the discretization, Global error of the three meshes i.e coarse, fine and finest as shown in result section, we can see that as the mesh size decreases the error reduces, thus giving better result of temperature and its distribution closer to the analytical solution

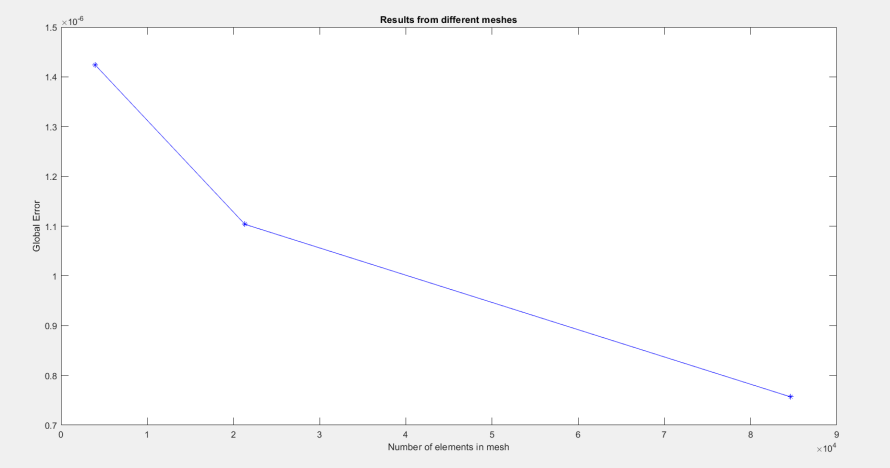

Fig:12 Num of elements vs Global error

5 Conclusion

From the discretization, Global error of the three meshes i.e coarse, fine and finest as shown in result section, we can see that as the mesh size decreases the error reduces, thus giving better result of temperature and its distribution closer to the analytical solution, on the filp side we can also see that the time required to the code for finer meshes increases significantly. Thus its better to use a mesh which maintains balance among accuracy and time. In our case Fine mesh achieves acceptable accurate results in reasonalble time.

Q1: To test a node position, can you compare directly the node distance from the center with the theoretical radius of 1? Is there any numerical limitation?
ans: Although we can compare the distance with theoretical value 1, there is a possibility that for some nodes the values of x,y $\in$ f(node *nsd + x or ysd) values squares sum may not be equal to one, further we also need to consider a floating point computation error, so it is preferred to take a tolerence of maybe one percent.

Q2: Try increasing the time step size in the input files. What happens and why? ans: As the time step increases The discretisation error value increases as numerical solution deviate from analytical solution, which is because the values are calculated by explicit first order scheme which comes from Taylor series expansion, where dt should be small and if it increases, then the linear effect increases and it does not fit the expansion.

References

[1] E Dick. Introduction to finite element methods in computational fluid dynamics. In *Computational fluid dynamics*. Springer, 2009.

[2] Jean Donea and Antonio Huerta. *Finite element methods for flow problems*. John Wiley & Sons, 2003.