HiperLife Tutorial: NonLinear Thermal Conduction

Arash Imani

LaCàN

November 3, 2024

1 Problem Definition

Thermal conduction is the diffusion of thermal energy within one material or between materials in contact. Nowadays it is becoming more important to predict heat transport properties according to geometric structures or component materials which add complexity in form of nonlinearity to the PDEs. Here the simplest case of nonlinear heat transfer equation has been chosen to show how to use nonlinear solver within the time discretization.
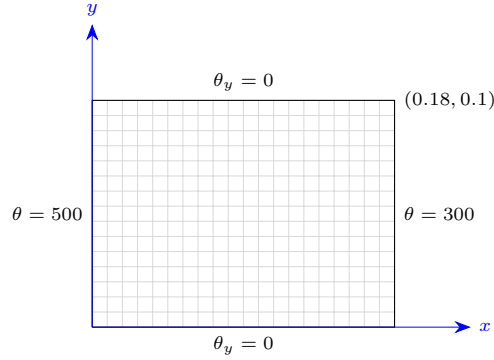

Figure 1: Geometry, BC and computational domain used for the analysis of nonlinear heat transfer.

2 Governing Equations

Consider the transient heat conduction equation [1]

$$\frac{\partial \theta}{\partial t} - \frac{\partial}{\partial x}(\kappa \frac{\partial \theta}{\partial x}) - \frac{\partial}{\partial y}(\kappa \frac{\partial \theta}{\partial y}) = 0 \quad \text{in } \Omega. \tag{1}$$

where θ is the temperature and κ is the conductivity. as it is demonstrated in Figure ??, The domain Ω is a rectangle of dimensions $[1.18 \times 0.1]$ along the x and y coordinates, respectively; the conductivity κ is of the form

$$\kappa = \kappa_0(1 + \beta\theta). \tag{2}$$

where $\kappa_0 = 0.2$ W/$(m \, °C)$ and $\beta = 2 \times 10^{-3} \, °C^{-1}$.the boundary conditions to be

$$\theta(0, y, t) = 500 \, °C, \quad \theta(0.18, y, t) = 300 \, °C, \quad \theta_y(x, 0, t) = \theta_y(x, 0.1, t) = 0. \tag{3}$$

The initial condition is assumed to be

$$\theta(x, y, 0) = 0. \tag{4}$$

This is essentially a one-dimensional problem. We choose a uniform mesh of $4 \times 2Q9$ to model the domain, $\Delta t = 0.005$ and we use Backward temporal approximation scheme.

3 Weak Form

The starting point for the development of the finite element models of Eq. (1) is their weak forms. By considering $\Delta t = t^{n+1} - t^n$, the α-family approximation for the temperature field takes the following form

$$\{\theta\}^{n+1} = \{\theta\}^n + \Delta t(1-\alpha)\{\theta_t\}^n + \Delta t\alpha\{\theta_t\}^{n+1} \,. \tag{5}$$

The time derivatives of θ on the right hand side of this equation can be calculated from Eq. (1) for each step, like this

$$\begin{aligned} \{\theta_t\}^{n+1} &= \nabla^2(\{\kappa\theta\}^{n+1}) \,, \\ \{\theta_t\}^n &= \nabla^2(\{\kappa\theta\}^n) \,. \end{aligned} \tag{6}$$

combining these two equations gives us

$$\{\theta\}^{n+1} - \Delta t\alpha[\nabla^2(\{\kappa\theta\}^{n+1})] - \{\theta\}^n - \Delta t(1-\alpha)[\nabla^2(\{\kappa\theta\}^n)] = 0 \,. \tag{7}$$

The variation formulation of our model problem can be introduced as find $\theta \in V$ such that

$$\mathcal{F}(\theta; v) = 0 \quad \forall v \in \hat{V} \,. \tag{8}$$

where

$$\mathcal{F}(\theta; v) = \int_\Omega v\{\theta\}^{n+1} - \Delta t\alpha v[\nabla^2(\{\kappa\theta\}^{n+1})] - v\{\theta\}^n - \Delta t(1-\alpha)v[\nabla^2(\{\kappa\theta\}^n)] \, \mathrm{d}\Omega \,. \tag{9}$$

and

$$\begin{aligned} \hat{V} &= \{v \in H^1(\Omega) : v = 0 \text{ on } x = 0 \text{ and } x = 0.18\}, \\ V &= \{v \in H^1(\Omega) : v = 500 \text{ on } x = 0 \text{ and } v = 300 \text{ on } x = 0.18\} \,. \end{aligned} \tag{10}$$

The discrete problem arises as usual by restricting V and $\hat{V}$ to a pair of discrete spaces.with $\theta = \sum_{j=1}^N \theta_j \phi_j$. Since $\mathcal{F}$ is a nonlinear function of θ, the variational statement gives rise to a system of nonlinear algebraic equations. applying integration by part, Using Gauss's theorem and also defining heat flux density as $q_{\mathfrak{n}} = [\nabla(\kappa\theta)] \cdot \mathfrak{n}$, the weak form takes the following form

$$\begin{aligned} \mathcal{F}(\theta; v) = &\int_\Omega v\{\theta\}^{n+1} + \Delta t\alpha\nabla v\nabla(\{\kappa\theta\}^{n+1}) \, \mathrm{d}\Omega - \int_\Omega v\{\theta\}^n + \Delta t(1-\alpha)\nabla v\nabla(\{\kappa\theta\}^n) \, \mathrm{d}\Omega \\ &- v\Delta t\int_\Gamma \alpha v q_{\mathfrak{n}}^{n+1} + (1-\alpha)v q_{\mathfrak{n}}^n \, \mathrm{d}\Gamma \,. \end{aligned} \tag{11}$$

Applying boundary conditions ($\Gamma = \Gamma_N \cup \Gamma_D$):

$$\begin{aligned} v &= 0 \quad \text{on } \Gamma_D \,, \\ \nabla[\kappa\theta] \cdot n &= 0 \quad \text{on } \Gamma_N \,. \end{aligned} \tag{12}$$

we get the final form for $\mathcal{F}$

$$\mathcal{F}(\theta; v) = \int_\Omega v\{\theta\}^{n+1} + \Delta t\alpha\nabla v\nabla(\{\kappa\theta\}^{n+1}) \, d\Omega - \int_\Omega v\{\theta\}^n + \Delta t(1-\alpha)\nabla v\nabla(\{\kappa\theta\}^n) \, \mathrm{d}\Omega \,. \tag{13}$$

In order to linearize our discretized nonlinear PDE problem, we may use Newton's method. which for the system $\mathcal{F}_i(\Theta_1, \ldots, \Theta_j) = 0$ it can be formulated by the first terms of a Taylor series approximation for the value of the variational as

$$\sum_{j=1}^N \frac{\partial}{\partial\Theta_j}\mathcal{F}_i(\Theta_1^k, \ldots, \Theta_N^k)\delta\Theta_j = -\mathcal{F}_i(\Theta_1^k, \ldots, \Theta_N^k), \quad i = 1, \ldots, N,$$

$$\Theta_j^{k+1} = \Theta_j^k + \delta\Theta_j, \quad j = 1, \ldots, N \,. \tag{14}$$

where k is an iteration index and n is time step. An initial guess θ^0 must be provided to start the algorithm. We need to compute the $\partial\mathcal{F}_i/\partial\Theta_j$ and the right-hand side vector $-\mathcal{F}_i$. Our present problem has $\mathcal{F}_i$ given by above. the Hessian is given by

$$\mathcal{J}(\theta; v) = \int_\Omega v\frac{\partial\theta^{n+1}}{\partial\theta_j} + \Delta t\alpha\nabla v(\frac{\partial\kappa}{\partial\theta_j}\theta^{n+1} + \kappa\frac{\partial\theta^{n+1}}{\partial\theta_j}) \, \mathrm{d}\Omega \tag{15}$$

2

4 Finite Element Model

Since we are developing the Ritz-Galerkin finite element model, the choice of the weight functions is restricted to the spaces of approximation functions used for the solution field. Suppose that the dependent variable θ approximated by expansions of the form

$$\theta(\mathbf{x},t) = \sum_{m=1}^{M} \phi_m(\mathrm{x})\theta^m(t) = \mathbf{\Phi}^T\boldsymbol{\theta}\,, \tag{16}$$

Lets rewrite this equation for known and unknown variables.

$$\begin{aligned}
\mathcal{J}(\theta;\phi_j,\phi_i) &= \int_\Omega \phi_i\phi_j + \Delta t\alpha\kappa_0\nabla\phi_i(\nabla\phi_j + 2\beta(\phi_j\nabla\theta^{n+1,k} + \theta^{n+1,k}\nabla\phi_j))\ \mathrm{d}\Omega \\
&= \phi_i\phi_j + \Delta t\alpha\kappa_0[\nabla\phi_i\nabla\phi_j + 2\beta(\nabla\phi_i\phi_j\nabla\theta^{n+1,k} + \nabla\phi_i\nabla\phi_j\theta^{n+1,k})]\mathrm{d}\Omega\,.
\end{aligned} \tag{17}$$

The above equations can be written symbolically in matrix form as we are going to solve it,

$$\sum_j \mathcal{J}(\theta;\phi_j,\phi_i)\{\delta\theta_j\}^{n+1} = -\mathcal{F}(\theta;\phi_i)\,. \tag{18}$$

where $\mathcal{F}(\theta;\phi_i)$ is

$$\begin{aligned}
\mathcal{F}(\theta;\phi_i) &= \int_\Omega \phi_i\theta^{n+1,k} + \kappa_0\Delta t\alpha\nabla\phi_i[\nabla\theta^{n+1,k} + 2\beta\nabla\theta^{n+1,k}\theta^{n+1,k}]\mathrm{d}\Omega \\
&\quad - \int_\Omega \phi_i\theta^{n,k} + \kappa_0\Delta t(1-\alpha)\nabla\phi_i[\nabla\theta^{n,k} + 2\beta\nabla\theta^{n,k}\theta^{n,k}]\mathrm{d}\Omega\,.
\end{aligned} \tag{19}$$

The elemental representation of the vector and matrix required for filling function in hiperlife would be like

$$\begin{aligned}
Ak(i,j) =&\, jac \times [\phi_i\phi_j + \Delta t\alpha\kappa_0[\nabla\phi_i\nabla\phi_j + 2\beta(\nabla\phi_i\phi_j\nabla\theta^{n+1,k} + \nabla\phi_i\nabla\phi_j\theta^{n+1,k})]]\,, \\
Bk(i) =&\, jac \times [(\phi_i\theta^{n+1,k} + \kappa_0\Delta t\alpha\nabla\phi_i\nabla\theta^{n+1,k}[1 + 2\beta\theta^{n+1,k}]) \\
&- (\phi_i\theta^{n,k} + \kappa_0\Delta t(1-\alpha)\nabla\phi_i\nabla\theta^{n,k}[1 + 2\beta\theta^{n,k}])]\,.
\end{aligned} \tag{20}$$

Note that $\mathrm{d}A = \mathrm{d}x_1 \times \mathrm{d}x_2 = jac\ \mathrm{d}\xi\mathrm{d}\eta$, which $jac = \det(Jacobian)$.

5 Choice of Elements

Thus, for this simple problem every Lagrange and serendipity family of interpolation functions are admissible for the interpolation of the temperature field, our choice is would be linear quadrilateral element. Linear Quadrilateral Elements is the simplest quadrilateral element consists of four nodes. The associated interpolation functions for geometry and field variables are bilinear.

$$\mathbf{\Phi}_I(\xi,\eta) = \frac{1}{4}(1 + \xi_I\xi)(1 + \eta_I\eta) \quad (I \text{ from 1 to 4}) \tag{21}$$

where ξ_I and η_I are the corner coordinates at element T in domain of $\Omega_T \in (-1,1)^2$. As it shown in Figure 2 we are using 2×2 Gauss–Legendre quadrature integration. We also chose a uniform mesh of size 10×10 to model the domain of our problem.

6 Implementation

In this section, we present the implementation of our solution in the Hiperlife. The program is divided into three separate files, main part which we create our problem by the Hiperlife headers, auxiliary header where we introduce parameters and declare defined functions, and at last auxiliary file, where we define some functions which provide required matrices like the Jacobian and the Hessian.
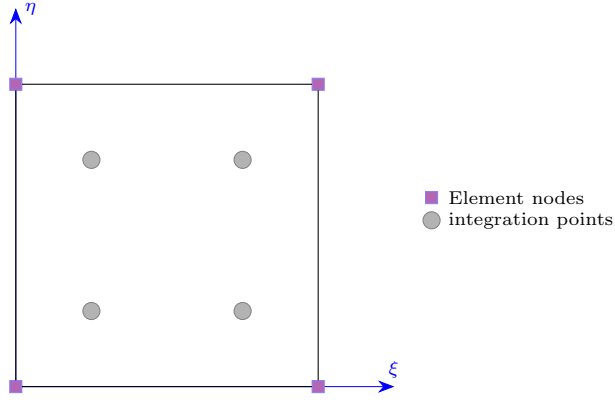
Figure 2: Linear quadrilateral element used for finite element model.

6.1 HeatTransferNonL.cpp

```cpp
/*
 * Heat Transfer conduction nonlinear
 */
// cpp headers
#include <iostream>
#include <fstream>
#include <cmath>

// hiperlife headers
#include "hl_Core.h"
#include "hl_ParamStructure.h"
#include "hl_Parser.h"
#include "hl_TypeDefs.h"
#include "hl_GlobalBasisFunctions.h"
#include "hl_StructMeshGenerator.h"
#include "hl_DistributedMesh.h"
#include "hl_FillStructure.h"
#include "hl_DOFsHandler.h"
#include "hl_SurfLagrParam.h"
#include "hl_HiPerProblem.h"
#include "hl_LinearSolver_Iterative_AztecOO.h"
#include "hl_NonlinearSolver_NewtonRaphson.h"
#include <hl_ConsistencyCheck.h>

// Header to auxiliary functions
#include "AuxHeatTransferNonL.h"

// ———————————————————————————————————————————————————————————————//
/// ————————————————————         MAIN FUNCTION        ————————————————————///
// ———————————————————————————————————————————————————————————————//

int main(int argc, char** argv)
{
        using namespace std;
        using namespace hiperlife;
        using namespace hiperlife::Tensor;

        // ———————————————————————————————————————————————————————————————//
        /// *****                        INITIALIZATION                        *****///
        // ———————————————————————————————————————————————————————————————//

        // Initialize MPI
        hiperlife::Init(argc, argv);

        // ———————————————————————————————————————————————————————————————//
```

4

```cpp
/// *****                        DATA INPUT                        *****///
// ────────────────────────────────────────────────────────────────//
// Time−related parameters
int maxSteps = 1000;
double maxTime  = 0.1;
double maxDeltat = 1.0;
double adaptiveFactor = 1.;

// number of output files
int nSave = 10;

// Put parameters in the user structure
SmartPtr<ParamStructure> paramStr = CreateParamStructure<HeatParams>();

// Data
paramStr−>setRealParameter(HeatParams::delta_t, 0.005);
paramStr−>setRealParameter(HeatParams::alpha, 0.5);
double delta_t = paramStr−>getRealParameter(HeatParams::delta_t);

// ────────────────────────────────────────────────────────────────//
/// *****                        MESH CREATION                        *****///
// ────────────────────────────────────────────────────────────────//

// Create a rectangular structured mesh
SmartPtr<StructMeshGenerator> structMesh = Create<StructMeshGenerator>();
structMesh−>setNDim(3);
structMesh−>setBasisFuncType(BasisFuncType::Lagrangian);
structMesh−>setBasisFuncOrder(2);
structMesh−>setElemType(ElemType::Square);
structMesh−>genRectangle(4, 2, 0.18, 0.1);

// Distributed mesh
SmartPtr<DistributedMesh> disMesh = Create<DistributedMesh>();
disMesh−>setMesh(structMesh);
disMesh−>setBalanceMesh(true);
disMesh−>setElementLocatorEngine(ElementLocatorEngine::BoundingVolumeHierarchy);
disMesh−>Update();

// checking mesh
disMesh−>printFileLegacyVtk("mesh");

// ────────────────────────────────────────────────────────────────//
/// *****                        DOFsHANDLER CREATION                        *****///
// ────────────────────────────────────────────────────────────────//

// DOFHandler
SmartPtr<DOFsHandler> dofHand = Create<DOFsHandler>(disMesh);
dofHand−>setNameTag("dofHand");
dofHand−>setNumDOFs(1);
dofHand−>setDOFs({"theta"});
dofHand−>Update();
// ───────────────────── Initial conditions───────────────────── //
//────────────────────────────────────────────────────────────────//
double f;
for (int i = 0; i < disMesh−>loc_nPts(); i++)
{
        // Coordinate
        std::vector<double> x = disMesh−>nodeCoords(i, IndexType::Local);
        f = 0.0 * x[i];
        // Initial condition
        dofHand−>nodeDOFs−>setValue("theta", i, IndexType::Local, f);
        // ───────────────────── Boundary condition ───────────────── //
        //────────────────────────────────────────────────────────────────//
        if (x[0] < 1e−5)
        {
                dofHand−>nodeDOFs−>setValue("theta", i, IndexType::Local,500.0);
                dofHand−>setConstraint("theta",i, IndexType::Local,0.0);
        }
```

```
114              if (x[0] > (0.18−1e−5))
115              {
116                      dofHand−>nodeDOFs−>setValue("theta", i, IndexType::Local,300.0);
117                      dofHand−>setConstraint("theta",i, IndexType::Local,0.0);
118              }
119          }
120          // Update
121          dofHand−>nodeDOFs0−>setValue(dofHand−>nodeDOFs);
122          dofHand−>nodeDOFs0−>UpdateGhosts();
123          dofHand−>UpdateGhosts();
124          // checking initial and boundary condition
125          dofHand−>printFileLegacyVtk("HeatTransferNonL0");
126          // —————————————————————————————————————————————————————//
127          /// *****                    HIPERPROBLEM CREATION               *****///
128          // —————————————————————————————————————————————————————//
129
130          SmartPtr<HiPerProblem> hiperProbl = Create<HiPerProblem>();
131          hiperProbl−>setParameterStructure(paramStr);
132          hiperProbl−>setDOFsHandlers({dofHand});
133          hiperProbl−>setIntegration("Integ", {"dofHand"});
134          hiperProbl−>setCubatureGauss("Integ", 4);
135          hiperProbl−>setElementFillings("Integ", LS);
136          if (true)
137          {
138                  hiperProbl−>setConsistencyDOFs("dofHand", {"theta"});
139                  hiperProbl−>setElementFillings("Integ", ConsistencyCheck<LS>);
140                  hiperProbl−>setConsistencyCheckType(ConsistencyCheckType::Hessian);
141          }
142          hiperProbl−>Update();
143
144          // —————————————————————————————————————————————————————//
145          /// *****                      SOLVER CREATION                  *****///
146          // —————————————————————————————————————————————————————//
147
148          SmartPtr<AztecOOIterativeLinearSolver> linsolver=
149          Create<AztecOOIterativeLinearSolver>();
150          linsolver−>setHiPerProblem(hiperProbl);
151          linsolver−>setTolerance(1.E−8);
152          linsolver−>setMaxNumIterations(500);
153          linsolver−>setSolver(AztecOOIterativeLinearSolver::Solver::Gmres);
154          linsolver−>setPreconditioner(AztecOOIterativeLinearSolver::Preconditioner::None);
155          linsolver−>setDefaultParameters();
156          linsolver−>setVerbosity(AztecOOIterativeLinearSolver::Verbosity::None);
157          linsolver−>Update();
158
159          // Create nonlinear solver
160          SmartPtr<NewtonRaphsonNonlinearSolver> nonLinSolver =
161          Create<NewtonRaphsonNonlinearSolver>();
162          nonLinSolver−>setLinearSolver(linsolver);
163          nonLinSolver−>setConvRelTolerance(true);
164          nonLinSolver−>setMaxNumIterations(15);
165          nonLinSolver−>setResTolerance(1e−6);
166          nonLinSolver−>setSolTolerance(1e−6);
167          nonLinSolver−>setResMaximum(1e5);
168          nonLinSolver−>setSolMaximum(1e5);
169          nonLinSolver−>setExitRelMaximum(true);
170          nonLinSolver−>setLineSearch(false);
171          nonLinSolver−>setPrintSummary(false);
172          nonLinSolver−>setPrintIntermInfo(true);
173          nonLinSolver−>Update();
174          // —————————————————————————————————————————————————————//
175          /// *****                    SOLVE HIPERPROBLEM                 *****///
176          // —————————————————————————————————————————————————————//
177          // Load loop
178          int timeStep{1};
179          double time{delta_t};
180          double delta_t0 = delta_t;
181          while (timeStep <= maxSteps and time <= maxTime)
```

```cpp
182              {
183                      // Print info
184                      if (hiperProbl->myRank() == 0)
185                      cout<<endl<<endl<<"Time-step:"<<timeStep<<"/"<<maxSteps<<"deltat"<<delta_t
186                      <<"time"<<time<<"/"<<maxTime<<endl;
187
188                      paramStr->setRealParameter(HeatParams::delta_t,delta_t);
189                      paramStr->setRealParameter(HeatParams::time,time);
190                      paramStr->setIntParameter(HeatParams::timeStep,timeStep);
191                      // Initial guess
192                      dofHand->nodeDOFs->setValue(dofHand->nodeDOFs0);
193                      hiperProbl->UpdateGhosts();
194
195                      bool converged = nonLinSolver->solve();
196
197                      // Check convergence
198                      if (converged)
199                      {
200                              // Save solution
201                              dofHand->nodeDOFs0->setValue(dofHand->nodeDOFs);
202                              dofHand->nodeDOFs0->UpdateGhosts();
203
204                              // Save results each nSave time steps
205                              if (timeStep%2 == 0)
206                              {
207                                      // Output results
208                                      string solName = "CondNL." + to_string(timeStep);
209                                      dofHand->printFileLegacyVtk(solName,true);
210                              }
211
212                              // Update load variables
213                              timeStep++;
214                              int iter = nonLinSolver->numberOfIterations();
215                              if (iter < 4)
216                              delta_t /= adaptiveFactor;
217                              else if (iter == 5)
218                              delta_t *= adaptiveFactor;
219                              if (delta_t > maxDeltat)
220                              delta_t = maxDeltat;
221                              if (time < maxTime && time + delta_t >= maxTime)
222                              delta_t = maxTime - time;
223                              time += delta_t;
224                      }
225                      else
226                      {
227                              // End if no adaptivity
228                              if (adaptiveFactor == 1.0)
229                              {
230                                      cout << endl << endl<< "Not-converged" << endl;
231                                      break;
232                              }
233
234                              // Steady state or non-convergence
235                              if (delta_t < 1e-8)
236                              {
237                                      cout <<endl<<endl<<"Steady-state-or-non-convergence"<<endl;
238                                      break;
239                              }
240                              time -=delta_t;
241                              delta_t *= adaptiveFactor;
242                              time += delta_t;
243                      }
244              }
245          hiperlife::Finalize();
246          return 0;
247 }
```

6.2 AuxHeatTransferNonL.h

```cpp
#ifndef AUXHeat_H
#define AUXHeat_H

// C headers
#include <iostream>

// hiperlife headers
#include "hl_Core.h"
#include "hl_ParamStructure.h"
#include "hl_Parser.h"
#include "hl_TypeDefs.h"
#include "hl_GlobalBasisFunctions.h"
#include "hl_StructMeshGenerator.h"
#include "hl_DistributedMesh.h"
#include "hl_FillStructure.h"
#include "hl_DOFsHandler.h"
#include "hl_HiPerProblem.h"
#include "hl_SurfLagrParam.h"
#include "hl_LinearSolver_Iterative_AztecOO.h"
#include "hl_NonlinearSolver_NewtonRaphson.h"

// alpha = 0.5: crank nicolson; delta_t = 0.05
// alpha = 1: backward euler; delta_t = 0.05
struct HeatParams
{
        enum RealParameters
        {
                delta_t,
                time,
                alpha,
        };
        enum IntParameters
        {
                timeStep,
        };
        HL_PARAMETER_LIST DefaultValues
        {
                {"delta_t,", 0.005},
                {"time,", 0.005},
                {"alpha,", 0.5},
                {"timeStep", 0},
        };
};

void LS(hiperlife::FillStructure& fillStr);

#endif
```

6.3 AuxHeatTransferNonL

```cpp
// Header to auxiliary functions
#include "AuxHeatTransferNonL.h"

// Hiperlife headers
#include "hl_Core.h"
#include "hl_ParamStructure.h"
#include "hl_Parser.h"
#include "hl_TypeDefs.h"
#include "hl_GlobalBasisFunctions.h"
#include "hl_StructMeshGenerator.h"
#include "hl_DistributedMesh.h"
#include "hl_FillStructure.h"
#include "hl_DOFsHandler.h"
#include "hl_HiPerProblem.h"
```

```cpp
15  #include "hl_SurfLagrParam.h"
16  #include "hl_LinearSolver_Iterative_AztecOO.h"
17  #include "hl_NonlinearSolver_NewtonRaphson.h"
18
19  using namespace std;
20  using namespace hiperlife;
21  using namespace hiperlife::Tensor;
22
23
24  // Conduction
25
26  void LS(hiperlife::FillStructure& fillStr)
27  {
28          double alpha  = fillStr.getRealParameter(HeatParams::alpha);
29          double delta_t = fillStr.getRealParameter(HeatParams::delta_t);
30          double k0{0.2};
31          double beta{0.002};
32
33          //————————————————————————————————————————————————————————————————————————//
34          // ———————————————————————————————— INPUT DATA ————————————————————————————//
35          //————————————————————————————————————————————————————————————————————————//
36
37          // Dimensions
38          SubFillStructure& subFill = fillStr["dofHand"];
39          int nDOFs = subFill.numDOFs;
40          int eNN   = subFill.eNN;
41          int nDim  = subFill.nDim;
42          int pDim  = subFill.pDim;
43
44          // Nodal values at Gauss points
45          wrapper<double,1> nborDOFs(subFill.nborDOFs.data(),eNN);
46          wrapper<double,1> nborDOFs0(subFill.nborDOFs0.data(),eNN);
47
48          // Shape functions and derivatives at Gauss points
49
50          double jac; // dx * dy = jac * dxi deta
51          wrapper<double,1>  bf(subFill.nborBFs(), eNN); //basis function
52          tensor<double,2> Dbf(eNN,pDim);                    //gradient of basis function
53          GlobalBasisFunctions::gradients(Dbf, jac, subFill);
54
55          //————————————————————————————————————————————————————————————————————————//
56          // —————————————————————— Initializing Hessian & Jacobian —————————————————//
57          //————————————————————————————————————————————————————————————————————————//
58          wrapper<double,2> Ak(fillStr.Ak(0, 0).data(), eNN, eNN);
59          wrapper<double,1> Bk(fillStr.Bk(0).data(),eNN);
60
61          //————————————————————————————————————————————————————————————————————————//
62          // ———————————————————————————————— VARIABLES —————————————————————————————//
63          //————————————————————————————————————————————————————————————————————————//
64
65          // temp
66          double theta = bf*nborDOFs;
67          double theta0 = bf*nborDOFs0;
68          // grad temp
69          tensor<double,1> grad_t(pDim);
70          tensor<double,1> grad_t0(pDim);
71          for (int i = 0; i < eNN; i++)
72          {
73                  for (int d = 0; d < pDim; d++)
74                  {
75                          grad_t(d) += Dbf(i,d)*nborDOFs(i);
76                          grad_t0(d) += Dbf(i,d)*nborDOFs0(i);
77                  }
78          }
79          // (gradient of the bf) *  (gradient of the bf)
80          tensor<double,2> DbfDbf = product(Dbf,Dbf,{{1,1}});
81          // (gradient of the bf) *  (gradient of theta)
82          tensor<double,1> DtDbf = product(grad_t,Dbf,{{0,1}});
```

```
83              tensor<double,1> Dt0Dbf = product(grad_t0,Dbf,{{0,1}});
84              //——————————————————————————————————————————————————————————//
85              //———————————————————— Filling Hessain and Jacobian ————————————————————//
86              //——————————————————————————————————————————————————————————//
87              for (int i = 0; i < eNN; i++)              //i for basis functions
88              {
89                      // Fill Jacobian
90                      Bk(i) += jac * (bf(i)*theta + k0*delta_t*alpha*DtDbf(i)*(1.0+2.0*beta*theta)
91                      - bf(i)*theta0 - k0*delta_t*(1.0-alpha)*Dt0Dbf(i)*(1.0+2.0*beta*theta0));
92                      for (int j = 0; j < eNN; j++)          //j for variable.
93                      {
94                              // Fill Hessian
95                              Ak(i,j) += jac * (bf(i)*bf(j) + delta_t*alpha*k0
96                              * (DbfDbf(i,j) * (1+2.0*beta*theta) + 2.0*beta*bf(j)*DtDbf(i)));
97                      }
98              }
99      }
```

7 Results

In this section, we present the results of our solution. The contour demonstration of temperature in the whole domain is shown in Figure 3.
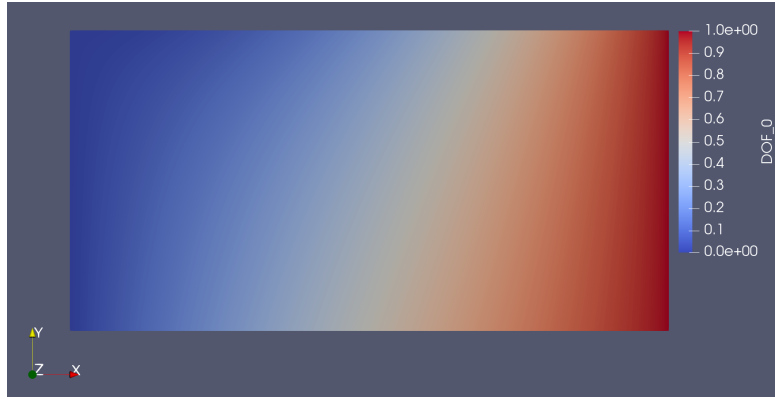

Figure 3: Distrobution of θ in the problem domain.

References

[1] Junuthula Narasimha Reddy. *An Introduction to Nonlinear Finite Element Analysis Second Edition: with applications to heat transfer, fluid mechanics, and solid mechanics.* OUP Oxford, 2014.