Parallel Computing for Computational Mechanics

Summer semester 2020

Project 1

---

1 Introduction

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 (the gray area in Figure 1). This model has an analytical solution to which the obtained numerical results can be compared.

The disk has a radius, $R2 = 0.1\text{m}$. A fixed temperature, $T_0 = 300\text{K}$ is set on the disk boundary, and a heat source of total power $P = 1\text{W}$ is applied on a small circular area (radius $R1 = 0.01\text{m}$) centered at the origin.

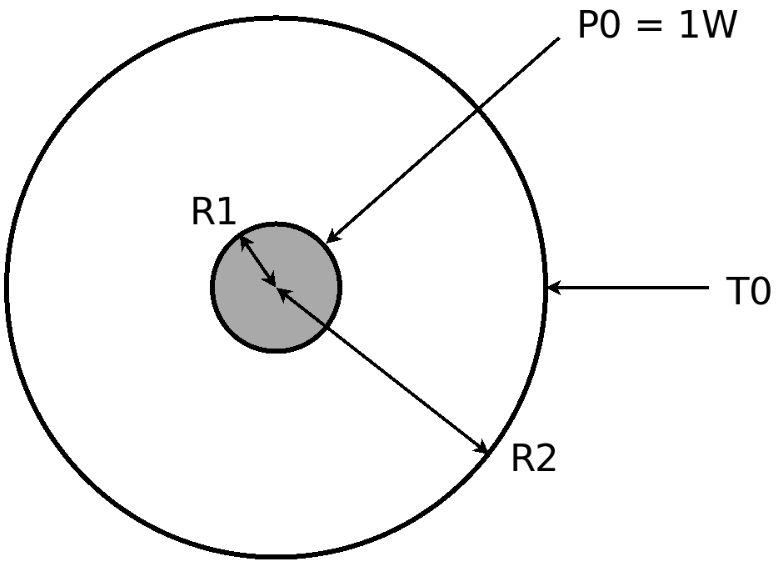

Figure 1: Geometry and boundary conditions

The formulation of the problem to solve is:

$$\begin{cases} \frac{\partial T}{\partial t} - \alpha \nabla^2 T = f & \text{in the disk domain,} \\ T = T_0 & \text{on the disk boundary.} \end{cases}$$

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:

$$f(r) = \begin{cases} \frac{Q}{\pi R_1^2} & \text{if } r < R_1, \\ 0 & \text{if } r \geq R_1. \end{cases}$$

$Q = P/d_z$ is the volumetric heat source. $d_z = 0.1\mathrm{m}$ is the out-of-plane thickness. The analytical solution of this problem is:

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

2 Derivation

The Gauss divergence theorem is applied to the weighted residual form:

$$\int_\Omega w\frac{\partial T}{\partial t}d\Omega + \int_\Omega \alpha\nabla w \cdot \nabla T \, d\Omega = \int_\Omega wf d\Omega.$$

Where Ω 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 can be obtained:

$$\sum_{j=1}^{nn}\left[\int_\Omega S_i S_j d\Omega\right]\frac{\partial T_j}{\partial t} + \sum_{j=1}^{nn}\left[\int_\Omega \alpha\left(\frac{\partial S_i}{\partial x}\frac{\partial S_j}{\partial x} + \frac{\partial S_i}{\partial y}\frac{\partial S_j}{\partial y}\right)d\Omega\right]T_j = \sum_{j=1}^{nn}\left[\int_\Omega S_i f d\Omega\right],$$

where S_i and S_j are respectively the trial function and the test function. In matrix form, the equation reads:

$$[M]\left\{\dot{T}\right\} + [K]\{T\} = \{F\}.$$

$[M]$ is called the mass matrix and $[K]$ the stiffness matrix. $\{F\}$ is the source vector. In order to solve this equation, you will have to:

1. Compute $[M]$, $[K]$, and $\{F\}$ (in `calculateElementMatrices`),
2. apply the necessary boundary condition (in `applyDrichletBC`),
3. build the system of equations and solve it (in `explicitSolver`).

3 `calculateElementMatrices`

3.1 Matrix computation

You have to compute first $[M^e]$, $[K^e]$, and $\{F^e\}$ for each element. The following pseudo code is given:

```
loop over quadrature points (nGQP)
    loop over the element nodes (i)
        loop over the element nodes (j)
            compute M[i][j]
            compute K[i][j]
    compute F[i]
```

To simplify the process, the element-wise entries of $[M^e]$, $[K^e]$, and $\{F^e\}$ are given as:

$$\begin{aligned} M_{ij}^e &= \int_{\Omega^e} (S_i S_j) \left|J\right| dx\, dy, \\ &= \sum_{m=1}^{nGQP} w(m)\, S_i(\zeta_m, \eta_m)\, S_j(\zeta_m, \eta_m) \left|J_m\right|, \end{aligned} \tag{1}$$

$$\begin{aligned} K_{ij}^e &= \int_{\Omega} \alpha \left(\frac{\partial S_i}{\partial x}\frac{\partial S_j}{\partial x} + \frac{\partial S_i}{\partial y}\frac{\partial S_j}{\partial y} \right) \left|J\right| dx\, dy, \\ &= \sum_{m=1}^{nGQP} \alpha w(m) \left(S_{i,x}(\zeta_m, \eta_m)\, S_{j,x}(\zeta_m, \eta_m) + S_{i,y}(\zeta_m, \eta_m)\, S_{j,y}(\zeta_m, \eta_m) \right) \left|J_m\right|, \end{aligned} \tag{2}$$

$$\begin{aligned} F_i^e &= \int_{\Omega} S_i f \left|J\right| dx\, dy, \\ &= \sum_{m=1}^{nGQP} f w(m)\, S_i(\zeta_m, \eta_m) \left|J_m\right| \end{aligned} \tag{3}$$

Description of the terms used:

- $w(m)$: Gauss weights,
- $S_i(\zeta_m, \eta_m)$: shape function in reference element,
- $|J_m|$: determinant of the Jacobian.
- $S_{i,x}(\zeta_m, \eta_m)$: shape function derivative with respect to x.

Because the mesh used is unstructured, the isoparametric formulation is used. The idea is to define the shape functions and shape function derivatives in a reference element and use a mapping from the reference element to an arbitrary deformed element (real element). More information on the procedure can be found in the "extra" folder. ζ and η are the coordinates in the reference element.

Hints:

- Make sure the matrices are initialized and do not overwrite the contributions from the previous quadrature points.
- The source term needs to be conditionally applied based on the position of the current node.

3.2 Mass lumping

To lump the mass matrix, the off-diagonal terms have to be condensed into the main diagonal effectively converting the mass matrix into a vector. The task here is to compute the lumped mass matrix, M^L, from the previously computed consistent mass matrix, M^C:

$$M_{ii}^L = M_{ii}^C \frac{M_e}{\sum_{j=1}^{nn} M_{jj}^C},$$

and

$$M_e = \sum_{i=1}^{nn} \sum_{j=1}^{nn} M_{ij}^C,$$

where i and j are the entry addresses in the mass matrix.

3.3 Save computed matrices

For each element, the different entries of the corresponding matrices have to be stored. The following helper functions have to be used:

1. `mesh->getNode(node)->addMass(M[i][i])`: add the mass value for node "node",
2. `mesh->getElem(e)->setM(i, M[i][i])`: set the mass value for element "e" entry "i",
3. `mesh->getElem(e)->setK(i,j,K[i][j])`: set the entry [i,j] of matrix K for element "e",
4. `mesh->getElem(e)->setF(i, F[i])`: set the entry [i] of vector F for element "e".

The mass information has to be stored in two classes. As "vector" with the first helper function and as "matrix" with the second helper function.

For debugging purpose, the matrices for element 3921 for the coarse mesh are:

$$K = \begin{bmatrix} 5.1716 \times 10^{-1} & -3.8007 \times 10^{-1} & -1.3709 \times 10^{-1} \\ -3.8007 \times 10^{-1} & 7.6273 \times 10^{-1} & -3.8265 \times 10^{-1} \\ -1.3709 \times 10^{-1} & -3.8265 \times 10^{-1} & 5.1975 \times 10^{-1} \end{bmatrix},$$

$$M = \begin{bmatrix} 2.0573 \times 10^{-6} \\ 2.0573 \times 10^{-6} \\ 2.0573 \times 10^{-6} \end{bmatrix},$$

$$F = \begin{bmatrix} 6.5485 \times 10^{-2} \\ 6.5485 \times 10^{-2} \\ 6.5485 \times 10^{-2} \end{bmatrix}.$$

4 `applyDrichletBC`

The meshes from Homework 1 will be used for this project. The origin of the meshes is at the center of the two concentric circles.

Step 1: For each of the nodes with coordinates (x, y), test whether this node is on the boundary (at distance of 0.1 from the center) (see Figure 1).
Question: 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?

Step 2: If the node is on a boundary:

- set the node boundary type to 1,

- get the temperature for the corresponding boundary from "settings.in",
- set the temperature in global "T" array for the current node.

Some helper functions:

- `mesh->getNode(i)->setBCtype(1)`: set the boundary condition type of node "i" to 1 (Dirichlet)
- `settings->getBC(FG)->getValue1())`: get the Dirichlet value for face group "FG".

Hint: Look into the code to find how to get the node coordinates from the `xyz` array.

5 `explicitSolver`

5.1 Time discretization

An explicit first order time integration scheme will be used.

$$\left.\frac{dT}{dt}\right|_n = \frac{T^{n+1} - T^n}{\Delta t} + \mathcal{O}\left(\Delta t\right),$$

Where n and $n+1$ are respectively the current time level and the next time level. This approximation is added to the matrix form given earlier:

$$[M]^n \frac{\{T\}^{n+1} - \{T\}^n}{\Delta t} + [K]^n \{T\}^n = \{F\}^n .$$

The unknown quantities are moved to the left hand side and the known quantities are moved to the right hand side:

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

5.2 Pseudocode

The pseudo code of the process is given as:

```
loop over the number of iterations:
    loop over the elements:
        get the pointers for the different matrices ([M], [K], and [F])
        use them to build the right hand side and store it in MTnew (vector)
    loop over the nodes:
        if the current node is not a boundary node:
            compute the new temperature at the current node
            compute partial error between new and old temperature
            store new temperature in T array (overwritten)
    compute global error
    if first time step:
        store initial value of global error for scaling
    scale global error with initial error
    if scaled error below 1e-7
        break iteration loop
```

5.3 Matrix access

To build the right-hand side, you have to access the M and K matrix. A representation of how the K matrix is stored is:

$$\begin{array}{|c|c|c|}\hline k_0 & k_1 & k_2 \\ \hline k_3 & k_4 & k_5 \\ \hline k_6 & k_7 & k_8 \\ \hline \end{array} \rightarrow \begin{array}{|c|c|c|c|c|c|c|c|c|}\hline k_0 & k_1 & k_2 & k_3 & k_4 & k_5 & k_6 & k_7 & k_8 \\ \hline \end{array}$$

Figure 2: K Matrix: Matrix representation (left) and representation in memory for access (right)

For example, $K_{3,2}$ can be accessed with `K[7]`.

5.4 Residual and Root Mean Square

The residual as a root mean square (RMS) of the temperature of two consecutive iterations can be computed as:

$$R_{RMS} = \sqrt{\frac{1}{nn}\sum_{i=1}^{nn}\left(T_i^{n+1} - T_i^n\right)^2},$$

with

- T_i^{n+1}: new temperature at node i,
- T_i^n: old temperature at node i.

Important: Dirichlet nodes should not be included in the computation of the residual. nn refers here to the number of non-Dirichlet nodes.

6 Code validation

The code you are developing will be an unsteady code. To get a steady result from your unsteady code, you should run the simulation in time until the solution is not changing anymore. So you will wait long enough for the physical system to reach a steady-state. To be sure that the solution is not changing, you should use a threshold on the residual and check whether this value is dropping below a certain predefined value. This **residual** should be defined as the RMS value of the change of temperature between two consecutive iterations, as stated above. This should also be normalized with the value at the initial time step. So you will have a residual of unity at the first time step and it will decrease in time until it reaches a user-defined value (here it must be 10^{-7}).

Complete the function `compareAnalytical()` in `postProcessor.cpp` with the node-wise computation of the analytical value. Compare this value with the outcome of the finite element code, this we could call the **discretization error**, which shows us how accurate our finite element code reproduces the correct solutioin. Print the **discretization error** at the end of your program.
To make this easier, the **discretization error** should also have the form of a RMS of the difference between

analytical and numerical solution:

$$\epsilon_{disc} = \sqrt{\frac{1}{nn}\sum_{i=1}^{nn}(T_N^n - T_A^n)^2},$$

with

- T_N^n: numerical solution at node i,
- T_A^n: analytical solution at node i.

Hint: you can also use this function to generate an analytical solution that can be opened in Paraview.
Tasks:

1. Run the problem with different mesh density (coarse, fine, finest).
2. Compare the results from the different meshes on a plot and comment on the convergence.
3. Try increasing the time step size in the input files. What happens and why?

7 Descriptions of some helper functions

The following helper functions are available.

- `mesh->getME(p)->getS(i)`: Get shape function for quadrature point `p` and element node `i`.
- `mesh->getElem(e)->getDetJ(p)`: Get determinant of the Jacobian for quadrature point `p`.
- `mesh->getME(p)->getWeight()`: Get Gauss weight for quadrature point `p`.
- `mesh->getElem(e)->getDSdX(p,i)`: Get shape function derivative for quadrature point `p` and element node `i`.
- `settings->getD()`: Get thermal diffusivity (from `settings.in`).
- `mesh->getNn()`: Get number of nodes.
- `mesh->getNe()`: Get number of elements.
- `settings->getNIter()`: Get number of time step iterations.
- `settings->getDt()`: Get time step step size.
- `mesh->getT()`: Get pointer to temperature array.
- `mesh->getXyz()`: Get pointer to node coordinates array.
- `mesh->getMassG()`: Get pointer to global mass array.
- `mesh->getMTnew()`: Get pointer to assembled Right Hand Side vector.

- `mesh->getNode(i)->getBCtype()`: Get type of node "i" ($0 =$ inside node, $1 =$ Dirichlet boundary condition).
- `mesh->getElem(e)->getMptr()`: Get pointer to mass matrix of element `e`.
- `mesh->getElem(e)->getKptr()`: Get pointer to stiffness matrix of element `e`.
- `elem->getFptr()`: Get pointer to the source vector of element `e`.
- `mesh->getElem(e)->getConn(i)`: Get global node ID from node `i` of element `e`. Each element has 3 nodes ($0 \leq i \leq 2$).

8 Project Report and Submission

Turn in the completed code and a report (PDF file) containing:

- Excerpts of the code with comments.
- Mesh picture(s) of the resulting temperature distribution.
- The plots of the temperature distribution and the error.
- The answer to any question in the text above.

We expect you to write a consistent report, comparable to a mini-paper. Therefore, you should use proper citations and also the right format for embedding formulas, tables and figures. A paper includes an introduction, a description of the problems, a method section, a results section and a conclusion.
All this should be wrapped within a single tar file `pr1.tar.gz`. Please do not use any other name or format for the archive. Furthermore, the following guidelines apply:

- Solutions are accepted until May 17 2020, 11:59 pm.
- Change only the parts of the code labelled "`// Add code here`" in `solver.cpp` and `postProcessor.cpp`.
- Do not add additional source files.
- The solution to this project shall be done individually (not as a team) and handed in online by using the $RWTHmoodle$ system.
- File `pr1.tar.gz` containing the needed files can be found on the $RWTHmoodle$ page.
- Use the provided template (`main.tex` inside the `tex-template` folder) to write your report.

Guidelines for the code:

- A compiling and running program must be provided.
- At the end of the program, the $error_{RMS}$ should be below 10^{-7}.
- The code must run on the RWTH ITC Cluster.

Complementary information

During this project you should also familiarize yourself with the cluster batch-job system. A special computing project has been created for this class, to benefit from it you have to follow these steps:

Log in to the required machine with the command

```
ssh -x your-tim-id@login18-1.hpc.itc.rwth-aachen.de
```

In order to submit a job, a `run.j` file is provided inside the `test` folder, which you can submit with

```
sbatch run.j
```

Listing all your pending and running jobs is possible with the `squeue` command:

```
squeue -u your-tim-id
```

Furthermore, the `scancel` command is available to kill and remove a submitted job:

```
scancel your-job-id
```

Contact:
Maximilian Schuster · schuster@cats.rwth-aachen.de
Violeta Karyofylli · karyofylli@cats.rwth-aachen.de