Chair for Computational Analysis of Technical Systems

RWTH AACHEN UNIVERSITY

Parallel Computing for Computational Mechanics

Summer semester 2022

Homework 2

---

1 Problem Introduction

1.1 Geometry and Boundary Conditions

In this homework, we are solving the 2D unsteady heat diffusion equation on a plane disk. On the problem domain Ω in Figure 1, a central subdomain with radius R_1, is heated with constant heat source P. The outer circumference of the disk is subject to a Dirichlet boundary condition of constant temperature T_D. Figure 1 and Table 1 describe the computational domain with boundary conditions.

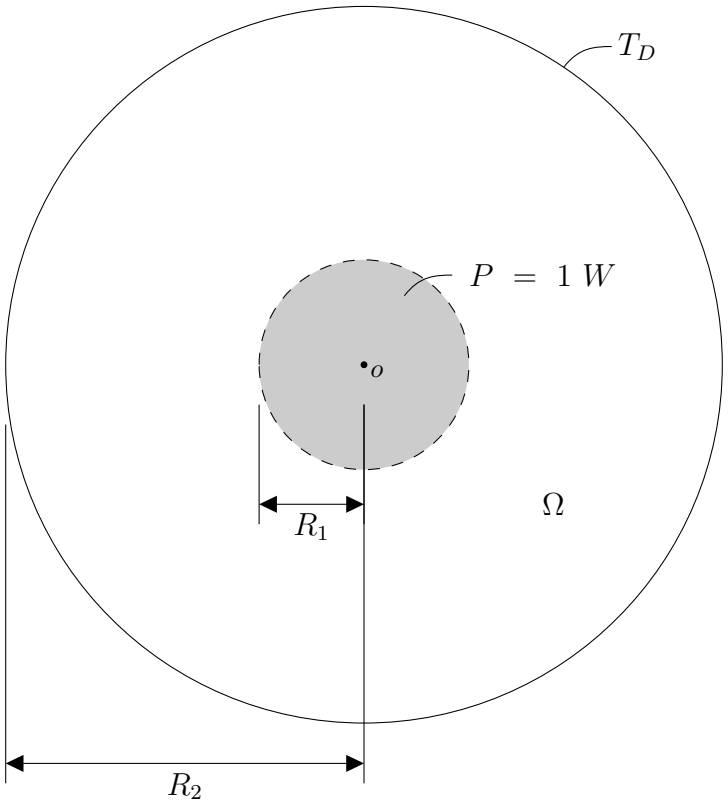

Figure 1: Geometry and boundary conditions.

We choose this problem because it comes with an analytical solution for the temperature as a function of the disk radius. We will use the finite element method to obtain the temperature distribution on the disk.

Table 1: Problem parameters for the heated disk problem.

Parameter	Variable	Value
Inner radius	R_1	0.01 [m]
Outer radius	R_2	0.1 [m]
Heat source	P	1.0 [W]
Dirichlet BC temperature	T_D	300 [K]

1.2 Mathematical Formulation of the problem

Referring to the previous exercise sessions, we can start at the partial differential equation (PDE) for heat diffusion with Dirichlet and Neumann boundary conditions as:

$$\frac{\partial T}{\partial t} - \alpha \nabla^2 T = f \qquad \text{in } \Omega,\ \forall\, t > 0, \tag{1a}$$

$$T = T_D \qquad \text{on } \Gamma_D,\ \forall\, t > 0, \tag{1b}$$

$$\nabla T \cdot n = h \qquad \text{on } \Gamma_h,\ \forall\, t > 0 \tag{1c}$$

$$T = T_0 \qquad \text{on } \Omega,\ \text{at } t = 0; \tag{1d}$$

where T is the temperature, α is the thermal diffusivity, f is a source term, T_D is the temperature boundary condition on the Dirichlet part of the boundary Γ_D, h is the heat flux prescribed on the Neumann part of the boundary, Γ_h. Ω is the domain on which we want to solve the equation. T_0 represents the initial temperature distribution at $t = 0$. In our case, the source term is non-zero for the region inside of R_1. After applying the Galerkin finite element method to the continous PDE, we arrive at the discretized variational formulation: find $T^h \in \mathcal{S}^h$, such that $\forall w^h \in \mathcal{V}^h$

$$\sum_{B \in \eta \setminus \eta_D} \left[(N_A, N_B) \frac{\partial T_B}{\partial t} + a(N_A, N_B) T_B \right] = (N_A, f) - \sum_{B \in \eta_D} a(N_A, N_B) T_D(\boldsymbol{x_B}), \quad \forall A \in \eta \setminus \eta_D. \tag{2}$$

(For a more in detail description, take a look at the FEM Reader from the second exercise session). This variational formulation eventually leads to the matrix form of equations:

$$[M] \left\{ \dot{T} \right\} + [K] \{T\} = \{F\}. \tag{3}$$

1.3 Analytical Solution

For the problem described above, an analytical solution for the temperature $\tilde{T}$ exists:

$$\tilde{T}(r) = \begin{cases} T_0 - \frac{Q}{2\pi\alpha} \left(\frac{1}{2} \left(\frac{r^2}{R_1^2} - 1 \right) + \ln\left(\frac{R_1}{R_2} \right) \right) & \text{if } r < R_1, \\ T_0 - \frac{Q}{2\pi\alpha} \ln\left(\frac{r}{R_2} \right) & \text{if } r \geq R_1, \end{cases} \tag{4}$$

where Q describes the volumetric heat source, which is obtained by dividing the heat source by an out-of-plane thickness of the disk, $d_z = 0.1\ m$, so that $Q = \frac{P}{d_z}$.

2 Tasks Overview

You will have to add code to a skeleton FEM solver code that does the following:

- **Task 1:** In `calculateElementMatrices()` determine the contributions to the element level matrices $[M]$, $[K]$ and $\{F\}$. At first, you will have to initialize those matrices. Then you fill in the respective contributions for each element. To make use of a simple explicit solver, you have to apply a technique called *mass lumping* (explained below). Finally, you will have to copy all the element level matrices to the respective global element.
- **Task 2:** For the Dirichlet boundary condition on the outer perimeter, complete the function `applyDirichletBC()`. If any boundary condition is set to Dirichlet according to the settings, the value for all the nodes on this boundary should be set to the value specified by the boundary condition.
- **Task 3:** Complete the function `explicitSolver()`. For this, you first clear the RHS matrices/vector. Then you evaluate the RHS at element level and accumulate the values from the local nodes. Calculate the global error by evaluating the local error and dividing it by the number of nodes solved. If a certain error threshold is undergone, break the solver loop. Complete the function `CompareAnalytical`, that calculates the analytical solution for each node based on its coordinates and compares it to the numerical solution.
- **Task 4:** Validate your implementation by running the code on different meshes.

3 Background

In this section, we want to present some additional theoretical background needed to solve the tasks. First, we will have a look at how we can discretize the time dependent terms of the heat equation, then we will present a simplification to explicitly solve the matrix equations.

3.1 Temporal Discretization

A first order explicit time integration scheme could be written like:

$$\left.\frac{dT}{dt}\right|_n = \frac{T^{n+1} - T^n}{\Delta t} + \mathcal{O}\left(\Delta t\right), \tag{5}$$

with n being the current time step and $n+1$ being the subsequent time step. The system of equations in matrix form can be expressed as:

$$[M]\left\{\dot{T}\right\} + [K]\{T\} = \{F\}. \tag{6}$$

(7)

After inserting Equation 5 into Equation 7, we obtain the following expression:

$$[M]^n \frac{\{T\}^{n+1} - \{T\}^n}{\Delta t} + [K]^n \{T\}^n = \{F\}^n . \tag{8}$$

We can then keep the unknown terms at the left hand side (LHS) and move the known terms to the RHS. Finally, we can isolate the unknown on the LHS by inverting the matrix M:

$$[M]^n \{T\}^{n+1} = [M]^n \{T\}^n + \Delta t \left(\{F\}^n - [K]^n \{T\}^n\right) \tag{9}$$

$$\{T\}^{n+1} = \left([M]^n\right)^{-1} \left([M]^n \{T\}^n + \Delta t \left(\{F\}^n - [K]^n \{T\}^n\right)\right). \tag{10}$$

3.2 Explicit Solver

A major part of any computational mechanics code is the solver. Our code aims to first simplify the LHS matrix into a diagonal matrix. This allows to easily invert the matrix by taking the inverse of each diagonal element. The technique used for this purpose is called *mass lumping*. It is an approximation to the original system, where the loss of accuracy is put up with by an increase in solving speed. There exist different mass lumping techniques. The one we are using here is adding up all contributions in one row of the mass matrix into its diagonal element. We apply mass lumping on the element level mass matrix and then store the result also in the global mass matrix. The superscript L refers to the lumped matrix, whereas C is used for the original (i.e., consistent) mass matrix. Mass lumping can be expressed as:

$$M^D = \sum_{i=1}^{nen} M_{ii}^C \tag{11}$$

$$M^T = \sum_{i=1}^{nen} \sum_{j=1}^{nen} M_{ij}^C \tag{12}$$

$$M_{ij}^L = \begin{cases} M_{ij}^C \frac{M^T}{M^D} & , \text{if } i = j, \\ 0 & , \text{if } i \neq j, \end{cases} \tag{13}$$

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

4 Implementation Work

In the following sections, we will go through all the implementation tasks you have to fulfill. First, we give an overview over the code provided in this homework.

4.1 Code Overview

The code you have to complete is located in the *skeleton* folder and contains the following files:

- `CMakeLists.txt` The instructions for the build process using `CMake`.
- `constants.h` Global constants.
- `postProcessor.cpp` Code for converting the solution to .vtk files.
- `postProcessor.h` Header file for `postProcessor.cpp`.
- `settings.cpp` Code for parsing the settings from the settings.in file.
- `settings.h` Header file for `settings.cpp`
- `solver.cpp` Code for filling the matrices, settings the boundary conditions and solving the linear system.
- `solver.h` Header file for the solver.
- `tri.cpp` Code for the discretization using finite elements.
- `tri.h` Header file for `tri.cpp`

All tasks you have to complete are located within `solver.cpp` and `postProcessor.cpp`.

4.2 Task 1: `calculateElementMatrices`

Compute the element level matrices and vector $[M^e]$, $[K^e]$, and $\{F^e\}$. Add the contributions as we have derived in the previous sessions. The following pseudo code can be applied for looping:

```
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]
```

After you have set up the element level contributions, lump the mass matrix as described above. In this work we use the isoparametric concept as discussed during the exercises. Hence we define both the solution and the geometry in a parametric space using a reference element.

Hints:

- The routine `calculateElementMatrices` is called in the `solverControl` routine, which loops over all the elements, and takes the current element as an input.
- 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.
- Add the total mass at each node to the local node structure with the `addMass` helper function.
- Copy the values you have obtained for $[M]$, $[K]$, and $\{F\}$ to the corresponding elements using the setter functions.
- When calculating the source term, you have to include the contribution to each element that has one or more quadrature points inside the source term region. Use the function `calculateRealPosition` to find the global position of the quadrature point and determine whether the current value has to be included for the source term contribution based on this evaluation.

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

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

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

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

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

4.3 Task 2: `applyDirichletBC`

In this task, you have to apply the Dirichlet boundary condition. Therefore, you will determine whether any of the boundaries is set to the type Dirichlet (`getType()==1`). Then, you loop over all nodes. In our case, the Dirichlet nodes are located on the outer perimeter of the domain. If the distance of a node to the origin is equal to the outer radius, you have to set the boundary type to 1. Use the helper functions to get and set the Dirichlet temperature at the respective node.
Hints:

- To define the loop breaking condition, we need to know the number of unknowns of problem the system. Implement a counter, that counts all the unknowns (i.e., all non-Dirichlet nodes).
- Can you directly use the exact spatial location of node a to determine if it is located on the outer perimeter?

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".

4.4 Task 3: `explicitSolver`

This task is the most comprehensive. We can split it into subtasks:

- `Solver`: Construct the RHS of element-level matrix equation as provided by Equation (9) and broadcast the result to the correct location in the global system. Then, go over all nodes and solve for the temperature $\{T\}^{n+1}$ by inverting the lumped mass matrix as given in Equation (10).
- `Loop breaking condition`: Although we are solving an unsteady case here, we are expecting the simulation to get to a steady solution. To determine, whether we have reached a steady state, we are comparing newly computed solution $\{T\}^{n+1}$ to the previous one ($\{T\}^{n}$). We calculate the root mean square of this difference and break the loop, when the current value called ϵ_c is below a threshold of 1×10^{-7}, with respect to the initial ϵ_0:

$$\epsilon_{RMS} = \sqrt{\frac{1}{nn}\sum_{i=1}^{nn}\left(T_i^{n+1} - T_i^{n}\right)^2} \tag{14}$$

$$\epsilon_{rel} = \epsilon_{RMS}(t = t_n) \tag{15}$$

$$\epsilon_0 = \epsilon_{RMS}(t = 0) \tag{16}$$

$$\epsilon_c = \frac{\epsilon_{rel}}{\epsilon_0}, \tag{17}$$

- `CompareAnalytical`: To evaluate the accuracy of the solver, the numerical solution T is compared against the analytical solution $\tilde{T}$. In `postProcessor.cpp`, complete the function `compareAnalytical`. This function should loop over all the nodes and get the corresponding solution, x, and y position. For each node the analytical solution at the current node needs to be computed and compared against

the numerical solution. For this, use the root mean square of the difference between numerical and analytical solution as given by the expression below:

$$\Delta_{T_{RMS}} = \sqrt{\frac{1}{nn}\sum_{i=1}^{nn}\left(T_i - \tilde{T}_i\right)^2}, \tag{18}$$

where T_i and $\tilde{T}_i$ represent the numerical and analytical solution at node i respectively.

Important: Dirichlet nodes should not be included in the computation of the error, hence, nn refers to the number of non-Dirichlet nodes.

The pseudo code of the task 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
```

Hints:

- The inverse of a diagonal matrix is computed simply by replacing the diagonal entries by its reciprocal.
- The $[K]$ matrix is stored using the approach depicted in Figure 2.
- Print the $\Delta_{T_{RMS}}$ to the terminal to evaluate the accuracy of your simulation w.r.t. the analytical solution.

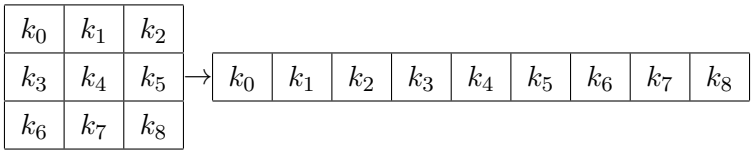

Figure 2: Storage strategy applied to $[K]$. Left: actual dimensions of $[K]$ and its entries. Right: $[K]$ stored row-wise in memory. For example, $K_{3,2}$ can be accessed with `K[7]`.

4.5 Task 4: Code validation

In this final task of this assignment, we will validate the code you have just implement. Complete the following tasks by using the RWTH ITC cluster:

1. Run the given problem with different mesh density (coarse, fine, finest).
2. Compare the results computed on the given meshes on a plot and comment on the convergence.
3. Try increasing the time step size in the input files. How does this affect the numerical solution and why?

5 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 ($0 \leq i \leq 2$).

6 Complementary information

During this homework you should also familiarize yourself with the cluster batch-job system. This system manages the allocated computing time for individual users and projects. A special computing project (and corresponding computing time) has been created for this class. To make use of this project can run you simulation using the following 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
```

Important: The allocated computing time for this class is limited and shared among all participating students in the class. Please be considerate when running you simulations using the batch-system and only run simulations when you are sure that they are set up correctly.

Contact:
Maximilian Schuster · schuster@cats.rwth-aachen.de
Michel Make · make@cats.rwth-aachen.de