Project A: Rotationally symmetric conductivity

Lisa Pizzo

January 25, 2026

Setup of the project

We consider the time dependent heat conductivity problem

$$c(x)\frac{\partial u(x,t)}{\partial t} - \nabla_x^T(\lambda(x)\nabla_x u(x,t)) = f(x) \quad (x,t) \in \Omega \times (0,T]$$

$$-\lambda(x)\frac{\partial u(x,t)}{\partial \vec{n}} = \alpha(x)(u(x,t) - u_{out}(x,t)) \quad (x,t) \in \partial\Omega \times (0,Z]$$

$$u(x,0) = u_0(x) \quad x \in \overline{\Omega} = \overline{\Omega}^{wall} \cup \overline{\Omega}^{fluid} \cup \overline{\Omega}^{air},$$

with the initial condition $u_0(x,0)$ and $x = (x_1, x_2, x_3)$. It consist of 3 different regions: The ceramic mug Ω^{wall} itself, the inner part with fluid Ω^{fluid} and the air region between surface of fluid and the top edge of the cup Ω^{air}.

The appropriate material coefficients c, λ, α differ with respect to the various subdomains. There are no inner heat sources f and only Robin boundary conditions are assumed. We assume that all coefficients above do not change in time or depend on the solution u and we do not consider heat radiation.

Rotational symmetry

Let us consider the mug without the handle so that we achieve the rotationally symmetric geometry Ω^{rot}.

The transformation of coordinates $((x_1, x_2, x_3) \to (\phi, r, z))$ and the assumption that no quantity is a function of ϕ results in the rotationally symmetric PDE

$$\begin{aligned}
rc(\cdot)\frac{\partial u(\cdot,t)}{\partial t} - \nabla_{(r,z)}^T\big(r\lambda(\cdot)\nabla_{(r,z)}u(\cdot,t)\big) &= rf(\cdot,t), && (\cdot,t) \in \Omega^{rot} \times (0,T] \\
-r\lambda(x)\frac{\partial u(\cdot,t)}{\partial \vec{n}} &= r\alpha(\cdot)\big(u(\cdot,t) - u_{out}(\cdot,t)\big), && (\cdot,t) \in \Gamma^{Robin} \times (0,T] \\
\lambda(x)\frac{\partial u(\cdot,t)}{\partial \vec{n}} &= 0, && (0,z,t) \in \Gamma^{symm} \times (0,T] \\
u(\cdot,0) &= u_0(\cdot), && (\cdot) \in \overline{\Omega} = \overline{\Omega}^{wall} \cup \overline{\Omega}^{fluid} \cup \overline{\Omega}^{air}
\end{aligned} \tag{1}$$

with the initial condition $u_0(x,0)$, the symmetry boundary $\Gamma^{symm} := \{(r,z) : r = 0\}$, the Robin boundary $\Gamma^{Robin} := \partial\Omega^{rot} \backslash \Gamma^{symm}$

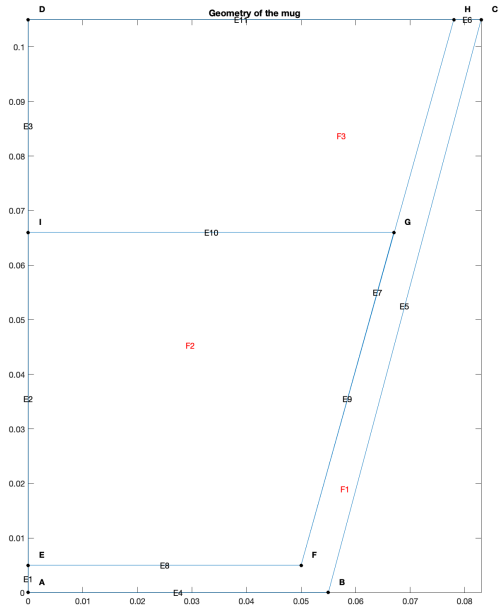

Figure 1: Rotationally symmetric mug geometry (Ω^{rot})

Task 1: Mesh definition

Statement:
Generate the 2D mesh with the 3 material domains Ω^{wall}, Ω^{fluid}, Ω^{air}. Derive the geometry description by changing `generate_mesh/chip_2materials.m` and using MatLab. The script above generates two files `chip_2materials.txt` (coordinates of vertices and the finite element connectivity) and `chip_2materials sd.txt` (material number per element).

Solution:
The mug geometry consists of three distinct subdomains:

- **Ceramic wall** (subdomain 1)
- **Fluid** inside the mug (subdomain 2)
- **Air** above the fluid (subdomain 3)

Defining the geometry correctly is crucial because later tasks assign material properties based on subdomain labels. MATLAB PDE Toolbox supports two approaches:

1. Legacy `initmesh` format (matrix-based), which is cumbersome for multi-domain geometries.
2. Modern `createpde()` with `geometryFromEdges`, which simplifies geometry creation, avoids overlapping edges, and allows easy labeling of subdomains.

For our mug, the modern approach is preferred.

```
% Axisymmetric mug
clear; clc; close all;

%  Create PDE model
model = createpde();

%  Points (meters)
A = [0,      0];
B = [0.055, 0];
C = [0.083, 0.105];
H = [0.078, 0.105];
F = [0.050, 0.005];
E = [0,      0.005];
G = [0.067, 0.066];
I = [0,      0.066];
D = [0,      0.105];

%  Geometry matrix (edges)
g1  = [2; A(1); E(1); A(2); E(2); 1; 0]; % Axis - ceramic
g2  = [2; E(1); I(1); E(2); I(2); 2; 0]; % Axis - fluid
g3  = [2; I(1); D(1); I(2); D(2); 3; 0]; % Axis - air
g4  = [2; A(1); B(1); A(2); B(2); 1; 0]; % Outer ceramic
g5  = [2; B(1); C(1); B(2); C(2); 1; 0]; % Outer ceramic
g6  = [2; C(1); H(1); C(2); H(2); 1; 3]; % Top rim: (C-H) ceramic-air
g7  = [2; H(1); F(1); H(2); F(2); 1; 3]; % Inner ceramic wall (H-F) ceramic-air
g8  = [2; F(1); E(1); F(2); E(2); 1; 2]; % Inner ceramic bottom (F-E) ceramic-fluid
g9  = [2; F(1); G(1); F(2); G(2); 2; 3]; % Fluid surface (F-G) fluid-air
g10 = [2; G(1); I(1); G(2); I(2); 2; 3]; % Fluid surface (G-I) fluid-air
g11 = [2; D(1); H(1); D(2); H(2); 3; 0]; % Air top boundary: (D-H) air-outside

% Assemble geometry
g = [g1 g2 g3 g4 g5 g6 g7 g8 g9 g10 g11];
geometryFromEdges(model, g);
figure(1);
pdegplot(model, 'EdgeLabels','on', 'FaceLabels','on');
axis equal;
title('Geometry with edge and face labels');

%  Generate linear mesh (3 nodes per element)
mesh = generateMesh(model, 'Hmax', 0.006, 'GeometricOrder','linear');

figure(2);
pdemesh(model);
axis equal;
title('Generated mesh');
```

Explanation of key steps:

- `model = createpde()`: creates a PDE model object that stores geometry, mesh, boundary conditions, and PDE coefficients.
- Each edge is defined in the PDE Toolbox "edge format":
  - `2` indicates a straight line segment,
  - `(x1,z1)` and `(x2,z2)` are the start and end points,
  - `left` and `right` indicate the subdomains on each side of the edge.
- `geometryFromEdges(model,g)`: imports the geometry matrix into the PDE model, creating a 2D geometry object with all edges and subdomains.
- `generateMesh(model,'Hmax',0.006)`: generates a triangular mesh with maximum element size `Hmax = 0.006 m`.

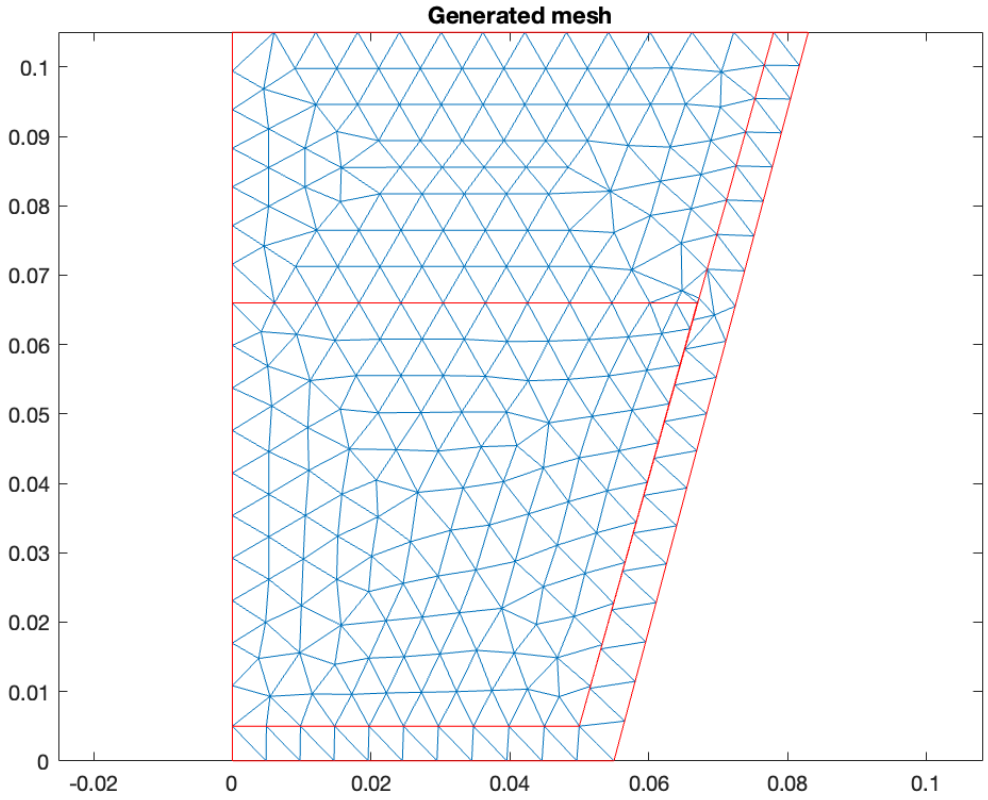

Figure 2: Triangular mesh of the axisymmetric mug domain

Task 2: Jacobi solver with constant lambda

Statement:
Follow the code structure in `mgrid_2/main.cpp` for the Jacobi branch (`#undef MG`) and check whether the stationary Dirichlet problem can be solved.

Solution:
We solve the stationary Laplace problem

$$-\nabla \cdot (\lambda \nabla u) = 0,$$

with zero Dirichlet boundary conditions on the outer boundary. For this first test, we assume $\lambda = 1$ throughout the domain. The Jacobi iterative method is used to solve the linear system resulting from finite element discretization.

```
nodes = mesh.Nodes;          % coordinates of all mesh nodes
elements = mesh.Elements;    % which nodes make up each triangle element

Nnodes = size(nodes,2);
Nelems = size(elements,2);

lambda = ones(Nelems,1);     % thermal conductivity


```

```
% Initialize global stiffness matrix and RHS
K = sparse(Nnodes, Nnodes);
F = zeros(Nnodes,1);

% Assemble K and F
for e = 1:Nelems %Loop over each triangle element
    vert = elements(:,e);   %nodes of element
    x = nodes(1,vert);
    y = nodes(2,vert);

    Ae = polyarea(x,y);     % Compute area of the triangle

    % Linear triangle gradients
    b = [y(2)-y(3); y(3)-y(1); y(1)-y(2)]; % derivative with respect to x
    c = [x(3)-x(2); x(1)-x(3); x(2)-x(1)]; % derivative with respect to y

    % Element stiffness matrix
    Ke = (lambda(e)/(4*Ae)) * (b*b.' + c*c.');

    % Assemble
    K(vert,vert) = K(vert,vert) + Ke;

    % Element load vector (f=0)
    F(vert) = F(vert) + zeros(3,1);
end

% Find boundary nodes
% Boundary nodes are identified via edge multiplicity
% all edges -> 2x(3*Nelems) array
edgesAll = [elements([1 2],:), elements([2 3],:), elements([3 1],:)];
% sort nodes of each edge, ensure that [i,j] and [j,i] are the same
edgesSorted = sort(edgesAll,1);
% identifies unique edges and assignes indices ic
[~,~,ic] = unique(edgesSorted','rows');
% counts how many times each edge appears in the mesh
counts = accumarray(ic,1);
%edges belonging to only 1 element: hence on the boundary
boundaryEdges = find(counts==1);
% nodes belonging to these boundary edges
boundaryNodes = unique(edgesSorted(:,boundaryEdges));

% Jacobi iteration
maxIter = 5000; tol = 1e-6;
D = diag(K);
R = K - diag(D);
u = zeros(Nnodes,1);

for iter = 1:maxIter
    u_new = (F - R*u) ./ D;
    u_new(boundaryNodes) = 0; % Enforce Dirichlet BC
    if norm(u_new - u, inf) < tol % Convergence check
        fprintf('Jacobi converged in %d iterations.\n', iter);
        break;
    end
    u = u_new;
end

% Plot solution
figure(3)
pdeplot(model, 'XYData', u, 'Mesh','on');
axis equal;
title('Stationary Dirichlet solution via Jacobi');
colorbar;
```

Explanation of key steps:

- **Mesh extraction:** `nodes` and `elements` store the coordinates of the mesh nodes and the connectivity of each triangle.
- **Assembly of K and F:** Each triangle element contributes a local stiffness matrix `Ke` which is assembled into the global stiffness matrix `K`. The local gradients of the shape functions are constant for linear triangles.
- **Boundary nodes:** Nodes on edges that belong to only one element are detected and used to enforce the Dirichlet boundary condition.
- **Jacobi iteration:** The linear system $Ku = F$ is solved iteratively. Boundary nodes are kept at zero during iterations.
- **Plot:** The resulting solution u is plotted over the mesh. Since the boundary values are zero and there is no internal source, the solution is $u = 0$ everywhere in this case.

Result and discussion:

```
Jacobi (Task 2) converged in 1 iterations.
```

This because the initial guess $u = 0$ already satisfies the PDE and boundary conditions.

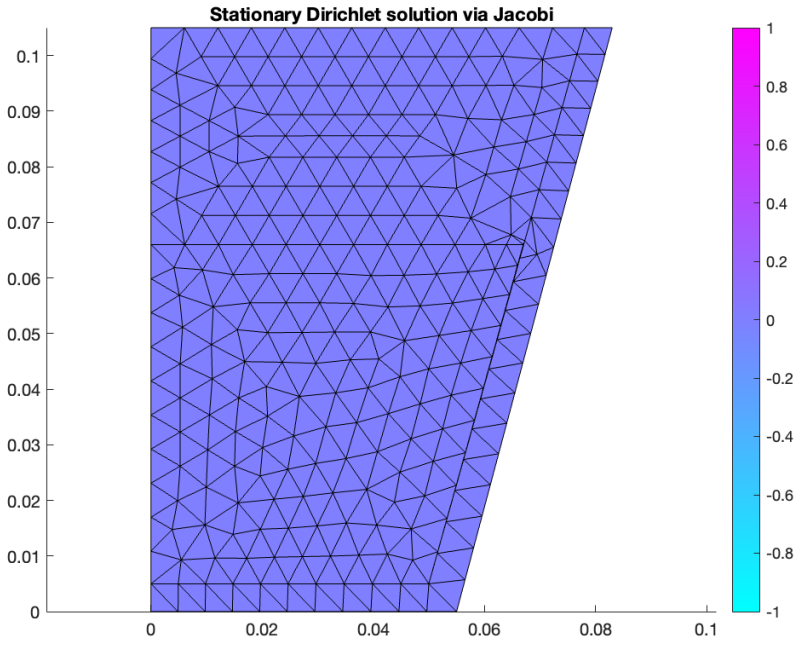

Figure 3: Task 2: Stationary Dirichlet solution using Jacobi iteration

Task 3: Laplace with multiple lambdas

Statement:
Implement for class `FEM_Matrix` a new Method `CalculateLaplace_mult` derived from `CalculateLaplace` that takes into account constant but different conductivities for your domains.

Solution:
Unlike Task 2, each subdomain has a different constant λ:

- λ_{wall} for ceramic wall
- λ_{fluid} for fluid
- λ_{air} for surrounding air

The finite element assembly is similar to Task 2, but each element stiffness is scaled by the element's conductivity according to its subdomain label.

```
function [K, F] = CalculateLaplace_mult(model, lambda_wall, lambda_fluid, lambda_air)
% PDE Toolbox compatible version

mesh = model.Mesh;
nodes    = mesh.Nodes;
elements = mesh.Elements;

Nnodes = size(nodes,2);
Nelems = size(elements,2);

K = sparse(Nnodes, Nnodes);
F = zeros(Nnodes,1);

%build region array manually
regions = zeros(Nelems,1);
wallElems  = findElements(mesh,'region','Face',1);
fluidElems = findElements(mesh,'region','Face',2);
airElems   = findElements(mesh,'region','Face',3);
regions(wallElems)  = 1;
regions(fluidElems) = 2;
regions(airElems)   = 3;

for e = 1:Nelems %assembly loop
    vert = elements(:,e);
    x = nodes(1,vert);
    y = nodes(2,vert);

    Ae = polyarea(x,y);

    b = [y(2)-y(3); y(3)-y(1); y(1)-y(2)];
    c = [x(3)-x(2); x(1)-x(3); x(2)-x(1)];

    % conductivity by material
    switch regions(e)
        case 1
            lambda = lambda_wall;
        case 2
            lambda = lambda_fluid;
        case 3
            lambda = lambda_air;
        otherwise
            error('Element %d has no material assignment', e);
    end

```

```
    Ke = (lambda/(4*Ae)) * (b*b.' + c*c.');
    K(vert,vert) = K(vert,vert) + Ke;
end
end
```

Explanation of key steps:

- Element wise conductivity: Each triangle uses λ_e based on subdomain.
- Assembly: Local matrices K_e are scaled by λ_e and added to the stiffness matrix.

$$K_e = \lambda_e \int_{T_e} \nabla\phi_i \cdot \nabla\phi_j \, d\Omega,$$

- Jacobi solver remains identical to Task 2

Result and discussion:

```
Jacobi (Task 3) converged in 1 iterations.
```

Solution is identical to Task 2 because since the problem is linear and homogeneous, piecewise constant λ does not affect the zero solution.

Task 4: Robin boundary condition

Statement:
Change the boundary conditions from Dirichlet boundary conditions to Robin boundary conditions. Derive a method `ApplyRobinBC_mult` from `ApplyDirichletBC` in class `FEM_Matrix`.

Solution:
Robin boundary conditions:

$$-\lambda \frac{\partial u}{\partial n} = \alpha \left(u - u_{\text{out}}\right) \quad \text{on } \partial\Omega,$$

represents convective heat transfer with the environment. In this equation α denotes the heat transfer coefficient and u_{out} the ambient temperature.

For linear elements, the weak form contribution for each boundary edge E of length L is:

$$K_e^{(R)} = \frac{\alpha L}{6} \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}, \qquad F_e^{(R)} = \frac{\alpha u_{\text{out}} L}{2} \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

```
function [K, F] = ApplyRobinBC_mult(model, K, F, alpha, u_out)
% ApplyRobinBC_mult
% Implements Robin BC:
%   lambda * du/dn + alpha*u = alpha*u_out
%
% alpha   : heat transfer coefficient
% u_out   : outside temperature

mesh = model.Mesh;
nodes = mesh.Nodes;
elements = mesh.Elements;

% --- Find boundary edges ---
edgesAll = [elements([1 2],:), elements([2 3],:), elements([3 1],:)];
edgesSorted = sort(edgesAll,1);
[edgesUnique,~,ic] = unique(edgesSorted','rows');
counts = accumarray(ic,1);
boundaryEdges = edgesUnique(counts==1,:);   % Nx2 array

% --- Loop over Robin boundary edges ---
for k = 1:size(boundaryEdges,1)
    i = boundaryEdges(k,1);
    j = boundaryEdges(k,2);

    ri = nodes(1,i);
    rj = nodes(1,j);

    if ri == 0 && rj == 0
    % Edge lies on symmetry axis r = 0
    % -> homogeneous Neumann BC, no Robin contribution
        continue;
    end

    xi = nodes(:,i);
    xj = nodes(:,j);

    L = norm(xi - xj);   % edge length

    % Robin boundary element matrices
    Ke = alpha * L / 6 * [2 1; 1 2];
    Fe = alpha * u_out * L / 2 * [1; 1];

    % Assemble
```

```
    K([i j],[i j]) = K([i j],[i j]) + Ke;
    F([i j])       = F([i j]) + Fe;
end
end
```

Explanation of key steps:

- Boundary detection: Same as Task 2, edges belonging to one element are boundary edges.
- Robin contribution: Local K_e and F_e are computed per edge and assembled globally.
- Symmetry axis handling: Boundary edges located on the symmetry axis $r = 0$ are explicitly excluded from the Robin assembly, since a homogeneous Neumann condition holds there and is naturally satisfied by the weak formulation.
- Jacobi solver: Uses updated K and F including Robin contributions.

Result and discussion:

`The Jacobi solver converges after approximately 2929 iterations.`

Temperature is nearly uniform at $u \approx u_{\text{out}} = 18°\text{C}$. Small variations of order 10^{-4} are observed between different parts of the boundary and are due to discretization and stopping tolerance.

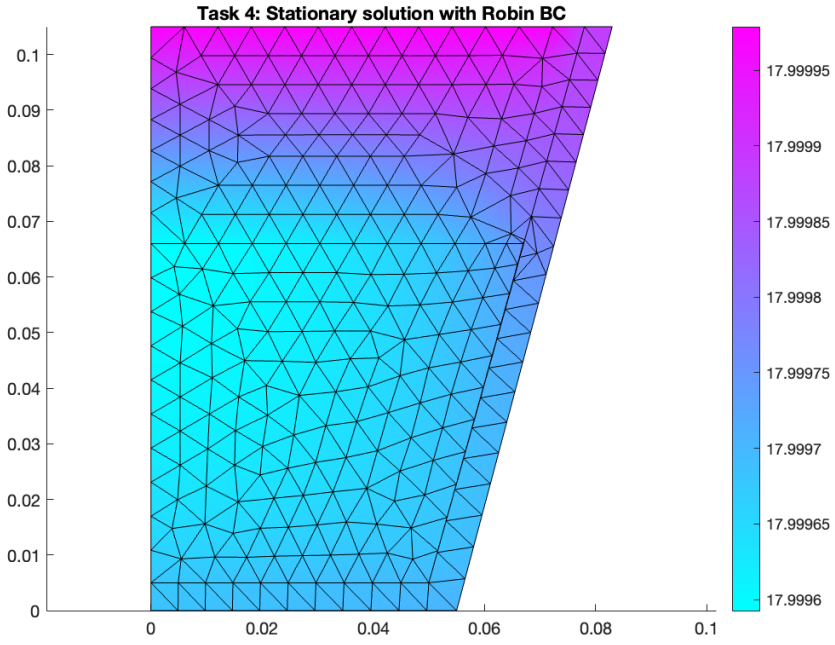

Figure 4: Task 4: Temperature distribution with Robin boundary conditions

Task 5: Axisymmetric Laplace + Robin BC

Statement:
Implement for class `FEM_Matrix` a new Method `CalculateLaplace_mult_rot` and a method `ApplyRobinBC_mult_rot` that realize the finite element matrix computation for the Laplace part (without $\frac{\partial u}{\partial t}$) in equation (1).

Solution:
Axisymmetry introduces a radial weighting factor r in integrals:

- Stiffness: $K_e \to r_{avg} K_e$
- Robin: $K_e^{(R)} \to r_{mid} K_e^{(R)}$ and $F_e^{(R)} \to r_{mid} F_e^{(R)}$

where r_{avg} is the element centroid radius and r_{mid} the midpoint of the boundary edge.

Robin boundary conditions:

$$-r\lambda \frac{\partial u}{\partial n} = r\alpha\,(u - u_{\text{out}}) \quad \text{on } \partial\Omega_{\text{rot}} \setminus \Gamma_{\text{symm}},$$

with $\Gamma_{\text{symm}} = \{r = 0\}$ denoting the symmetry axis, where a homogeneous Neumann condition is naturally satisfied.

```
function [K, F] = CalculateLaplace_mult_rot(model, lambda_wall, lambda_fluid,
    lambda_air)
mesh = model.Mesh;
nodes = mesh.Nodes;
elements = mesh.Elements;

Nnodes = size(nodes,2);
Nelems = size(elements,2);

K = sparse(Nnodes, Nnodes);
F = zeros(Nnodes,1);

regions = zeros(Nelems,1);
regions(findElements(mesh,'region','Face',1)) = 1;
regions(findElements(mesh,'region','Face',2)) = 2;
regions(findElements(mesh,'region','Face',3)) = 3;


for e = 1:Nelems
    vert = elements(:,e);

    x = nodes(1,vert);    % r-coordinates
    y = nodes(2,vert);    % z-coordinates

    Ae = polyarea(x,y);

    b = [y(2)-y(3); y(3)-y(1); y(1)-y(2)];
    c = [x(3)-x(2); x(1)-x(3); x(2)-x(1)];

    rbar = mean(x);       % <-- axisymmetric weight
    switch regions(e)
        case 1
            lambda = lambda_wall;
        case 2
            lambda = lambda_fluid;
        case 3
            lambda = lambda_air;
    end
    Ke = rbar * (lambda/(4*Ae)) * (b*b.' + c*c.');
```

```
    K(vert,vert) = K(vert,vert) + Ke;
end
end
```

```
function [K, F] = ApplyRobinBC_mult_rot(model, K, F, alpha, u_out)
mesh = model.Mesh;
nodes = mesh.Nodes;
elements = mesh.Elements;

edgesAll = [elements([1 2],:), elements([2 3],:), elements([3 1],:)];
edgesSorted = sort(edgesAll,1);
[edgesUnique,~,ic] = unique(edgesSorted','rows');
counts = accumarray(ic,1);
boundaryEdges = edgesUnique(counts==1,:);

for k = 1:size(boundaryEdges,1)
    i = boundaryEdges(k,1);
    j = boundaryEdges(k,2);

    ri = nodes(1,i);
    rj = nodes(1,j);

    if ri == 0 && rj == 0
    % Edge lies on symmetry axis r = 0
    % -> homogeneous Neumann BC, no Robin contribution
        continue;
    end

    xi = nodes(:,i);
    xj = nodes(:,j);

    L = norm(xi - xj);
    rbar = 0.5 * (xi(1) + xj(1));   % r at edge midpoint

    Ke = rbar * alpha * L / 6 * [2 1; 1 2];
    Fe = rbar * alpha * u_out * L / 2 * [1; 1];

    K([i j],[i j]) = K([i j],[i j]) + Ke;
    F([i j]) = F([i j]) + Fe;
end
end
```

Result and discussion:

`The Jacobi solver converges after 3354 iterations.`

Temperature field remains nearly uniform, representing a physically consistent 3D solution obtained by rotating the 2D axisymmetric solution. Robin boundary contributions are applied only on $\partial\Omega_{\text{rot}} \setminus \Gamma_{\text{symm}}$, while boundary edges on the symmetry axis $r = 0$ are explicitly excluded in the implementation, since the homogeneous Neumann condition on Γ_{symm} is naturally satisfied. This confirms that the axisymmetric formulation is consistent with the full 3D model.

Small variations of order 10^{-4} are observed.

The two-dimensional visualization of the solution in the (r, z)-plane is very similar to the result obtained in Task 4 and is therefore not shown again.

It is possible to find an animation of this 3D plot into the "Images and Videos" folder.

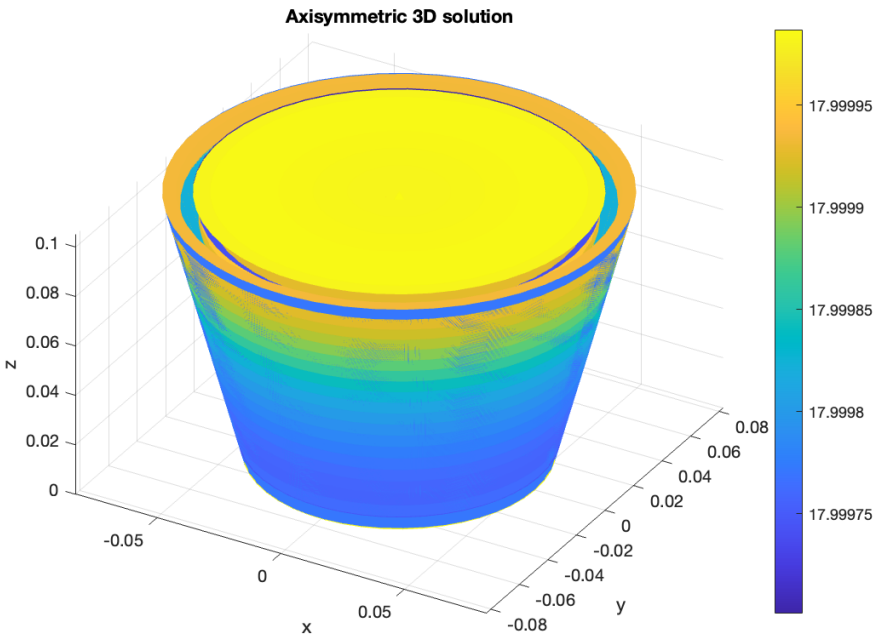

Figure 5: Task 5: Axisymmetric stationary solution with Robin BC

Task 6: Axisymmetric mass matrix assembly

Statement:
Implement a method `AddMass_mult_rot` that realizes internally the calculation of the mass matrix entries

$$M_{i,j} := \int \int rc(r,z)\phi_i\phi_j dr dz$$

similar to the element calculation for the Laplace part of the matrix $K_{i,j}$ and adds it to the existing matrix entries.

Solution:
The mass matrix represents heat storage and it is defined

$$M_{i,j} := \int_{\Omega_{\text{rot}}} r\, c(r,z)\, \phi_i\phi_j\, dr\, dz,$$

where $c(r,z)$ denotes the heat capacity and ϕ_i are the linear finite element shape functions.

For linear triangles and axisymmetry, element contribution:

$$M_e = r_{avg}\, c_e\, \frac{A_e}{12} \begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{pmatrix},$$

where c_e is material specific heat capacity.

```
function M = AddMass_mult_rot(model, M, c_wall, c_fluid, c_air)
mesh = model.Mesh;
nodes = mesh.Nodes;
```

```
elements = mesh.Elements;

Nelems = size(elements,2);

regions = zeros(Nelems,1);
regions(findElements(mesh,'region','Face',1)) = 1; % wall
regions(findElements(mesh,'region','Face',2)) = 2; % fluid
regions(findElements(mesh,'region','Face',3)) = 3; % air

for e = 1:Nelems
    vert = elements(:,e);

    x = nodes(1,vert);    % r-coordinates
    y = nodes(2,vert);    % z-coordinates

    Ae = polyarea(x,y);  % element area
    rbar = mean(x);      % mean radius (axisymmetric weight)

    switch regions(e) % Select heat capacity
        case 1
            c = c_wall;
        case 2
            c = c_fluid;
        case 3
            c = c_air;
    end
    % Axisymmetric element mass matrix
    Me = rbar * c * Ae / 12 * [2 1 1; 1 2 1; 1 1 2];
    % Assemble
    M(vert,vert) = M(vert,vert) + Me;
end
end
```

Task 7: Initial solution

Statement:
Write a function (or method) `Init_Solution_mult` that initializes the solution in the whole domain depending on the subdomain. That function might be useful in sub-task (i) as well as in (ii).

Solution:

- Each node should be initialized once.
- Initial temperatures: `u_wall`, `u_fluid`, `u_air` for respective subdomains.
- Shared nodes are assigned only once using a boolean mask

```
function u0 = Init_Solution_mult(model, u_wall, u_fluid, u_air)
mesh = model.Mesh;
nodes = mesh.Nodes;
elements = mesh.Elements;

Nnodes = size(nodes,2);
Nelems = size(elements,2);

u0 = zeros(Nnodes,1); %initialize solution vector

regions = zeros(Nelems,1);
regions(findElements(mesh,'region','Face',1)) = 1; % wall
```

```
regions(findElements(mesh,'region','Face',2)) = 2; % fluid
regions(findElements(mesh,'region','Face',3)) = 3; % air

% Creates a boolean array to ensure each node is assigned a temperature only
% once
nodeAssigned = false(Nnodes,1);

for e = 1:Nelems
    vert = elements(:,e); %current element
    switch regions(e) %determines which temperature to assign
        case 1
            u_val = u_wall;
        case 2
            u_val = u_fluid;
        case 3
            u_val = u_air;
    end

    %loops over the three nodes of the current element
    % if a node has not been assigned -> u_val and marks assigned in the
    % boolean vector
    for k = 1:3
        i = vert(k);
        if ~nodeAssigned(i)
            u0(i) = u_val;
            nodeAssigned(i) = true;
        end
    end
end
end
```

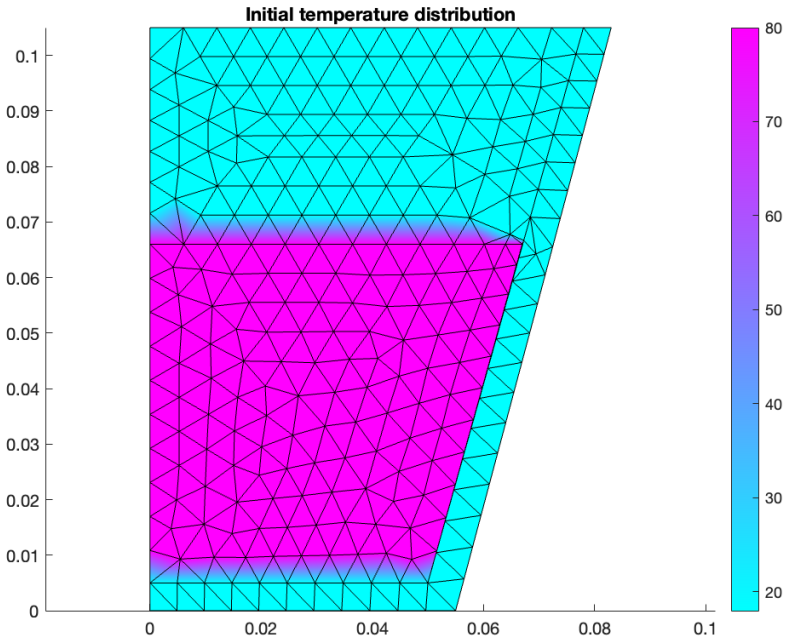

Figure 6: Task 7: Initial temperature distribution in all subdomains

Task 8: Time discretization of the heat equation

Statement:
Use an explicit scheme to discretize in time so that we end up with the discrete formulation of (1)

$$(\frac{1}{\tau}M + K)\underline{u}^{k+1} = \underline{f}^{k+1} + \frac{1}{\tau}M \cdot \underline{u}^{k} \tag{2}$$

with τdenoting the chosen time step.

Solution:
This method allows us to compute the solution $\underline{u}^{k+1}$ from the previous time step $\underline{u}^{k}$.

```
tau = 0.1;              % time step in seconds
T_end = 400;            % total simulation time (seconds)
Nt = ceil(T_end/tau);   % number of time steps

A = (1/tau)*M+K; % Left-hand side matrix
[L,U,P,Q] = lu(A); % A is constant -> factorize it once: PAQ=LU

u = u0; % Initialize solution

for k = 1:Nt
    b = (1/tau)*M*u + F;   % F is the load vector, F=0

    % Solve for next time step
    u_next = Q*(U\(L\(P*b))); % efficient solution using LU factors

    u = u_next; % Update

    if mod(k,20) == 0 %Plot every 20 steps
        figure(8)
        pdeplot(model, 'XYData', u, 'Mesh','on');
        axis equal
        title(['Temperature at t = ', num2str(k*tau), ' s']);
        colorbar
        drawnow
    end
end
```

Explanation of key steps:

- **Time discretization:** The transient heat equation is discretized in time using a first-order scheme, leading to a linear system at each time step.
- **System matrix:** The matrix $(1/\tau)M + K$ combines heat capacity effects and thermal diffusion including Robin boundary conditions.
- **LU factorization:** Since the system matrix is constant, it is factorized once to efficiently solve the system at every time step.
- **Time stepping:** Starting from the initial temperature distribution u_0, the solution is advanced iteratively in time.
- **Visualization:** The temperature distribution is plotted periodically to observe the transient heat propagation.

Result and discussion:
The implemented scheme produces a stable and physically consistent transient temperature

evolution of the mug. Heat diffuses from the hot fluid into the ceramic wall and surrounding air, as expected from the governing heat equation.

To illustrate the time-dependent heat transfer within the mug, snapshots of the temperature distribution are recorded at selected physical times. Figure 7 shows the evolution of the temperature field at $T = 0, 50, 100, 150, 200, 250, 300, 350$, and 400 s.

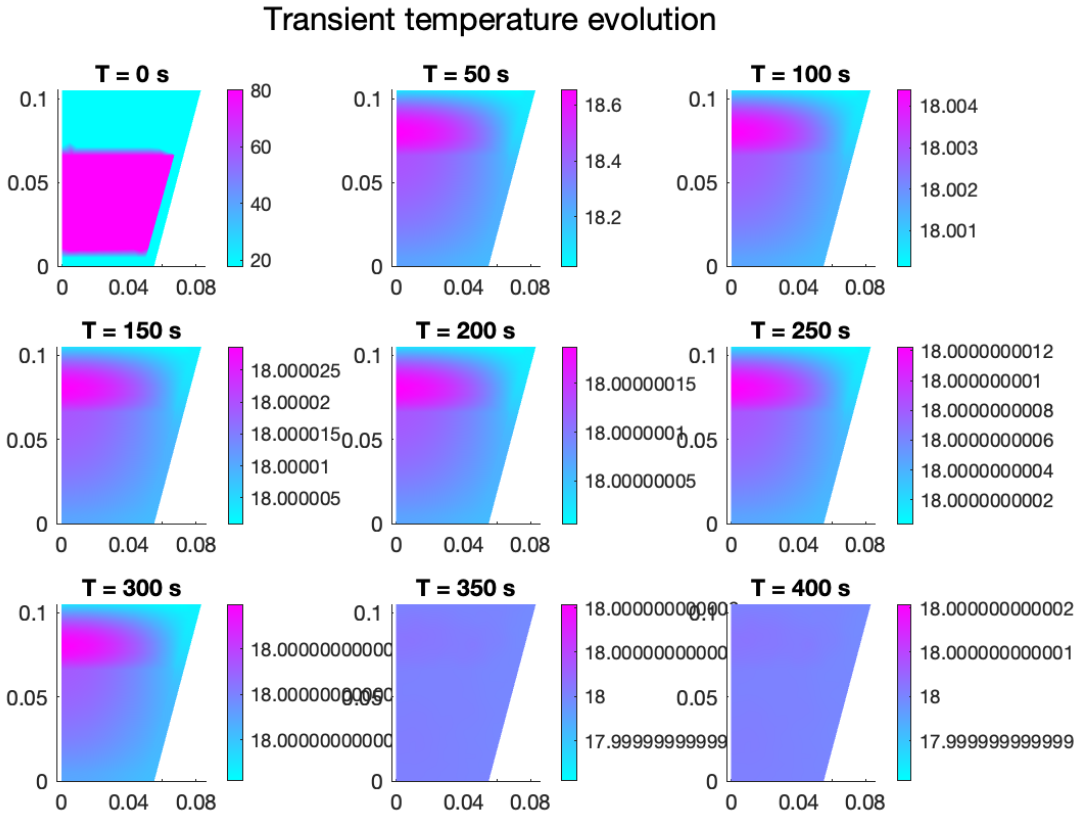

Figure 7: Temperature distribution in the mug at selected time instances.

It is possible to find an animation of this 9 plots into the "Images and Videos" folder.

Task 9: Time-dependent solution of the heat equation

Statement:
Solve (2) for the following sub-task: The initial temperature of the cup is the same as the surrounding air $u_0^{air,wall}$=18°C. The cup is filled with warm water such that u_0^{fluid}=80°C. **Question:** How long does it take (T_{warm}) until the mug is heated to the optimal temperature of 67°C?

Solution:
The rotationally symmetric, time-dependent heat equation is given by

$$rc(\mathbf{x})\frac{\partial u(\mathbf{x},t)}{\partial t} - \nabla_{(r,z)}^T \left(r\lambda(\mathbf{x})\nabla_{(r,z)} u(\mathbf{x},t) \right) = 0, \quad (\mathbf{x},t) \in \Omega_{\text{rot}} \times (0,T].$$

Using an explicit Euler discretization in time with time step τ, we obtain the linear system

$$\left(\frac{1}{\tau} M + K \right) \underline{u}^{k+1} = \frac{1}{\tau} M \, \underline{u}^k,$$

where M is the mass matrix assembled using the spatially varying heat capacity and K is the stiffness matrix resulting from the Laplace operator with Robin boundary conditions.

The initial condition is defined piecewise as

$$u(\mathbf{x},0) = \begin{cases} 80°\text{C}, & \mathbf{x} \in \Omega_{\text{fluid}}, \\ 18°\text{C}, & \mathbf{x} \in \Omega_{\text{wall}} \cup \Omega_{\text{air}}. \end{cases}$$

```
%% Task 9 (i): Heating time using inner ceramic wall temperature
T_target = 67;   % [C]

innerWallNodes = findNodes(model.Mesh,'region','Edge',8); %Edge 8 = ceramic-fluid
u = u0;

% Storage
timeVec = (0:Nt-1)' * tau;
innerWallTemp = zeros(Nt,1);
Twarm = NaN;

for k = 1:Nt
    b = (1/tau)*M*u + F;

    u = Q*(U\(L\(P*b)));

    % Average inner wall temperature
    innerWallTemp(k) = mean(u(innerWallNodes));

    % Check heating criterion
    if innerWallTemp(k) >= T_target
        Twarm = k * tau;
        fprintf('Task 9 (i): Inner wall reaches %.1f C at T = %.1f s\n', ...
                T_target, Twarm);
        break
    end
end

%% --- Plot inner wall temperature evolution
figure(9)
plot(timeVec(1:k), innerWallTemp(1:k), 'LineWidth', 2)
hold on
yline(T_target,'r--','67C','LineWidth',1.5)
xlabel('Time [s]')
```

```
ylabel('Average inner wall temperature [C]')
title('Heating of the inner ceramic wall')
grid on
```

Explanation of key steps:

- **Target definition:** The desired temperature $T_{\text{target}} = 67°\text{C}$ is defined according to the problem statement.
- **Inner wall identification:** The nodes belonging to the ceramic–fluid interface are extracted using `findNodes`. The average temperature over these nodes is used as a representative inner wall temperature.
- **Time stepping:** Starting from the initial temperature distribution u_0, the solution is advanced in time using the explicit formulation derived in Task 8. At each step, the right-hand side vector is assembled as

$$b = \frac{1}{\tau} M u^k + F.$$

- **Linear system solve:** The resulting linear system is solved using a precomputed LU factorization of the matrix $\left(\frac{1}{\tau}M + K\right)$, ensuring efficient time stepping.
- **Heating criterion:** After each time step, the mean temperature at the inner ceramic wall is computed and compared to T_{target}. If the target temperature is reached, the corresponding heating time T_{warm} is stored and the simulation is stopped.
- **Post-processing:** The temporal evolution of the average inner wall temperature is plotted together with the target temperature line to visualize the heating behavior.

Results and discussion:
The figure shows the transient temperature evolution inside the mug for several time instances.

Although the initial hot fluid causes a rapid increase of temperature in the surrounding regions during the first seconds, the maximum temperature inside the domain never reaches the target value of 67°C. Instead, the temperature peaks at approximately 55°C shortly after the start of the simulation and subsequently decreases, approaching the ambient temperature of 18°C as time progresses.

This indicates that, under the current model assumptions and parameter choices, the heat losses dominate over the heat storage, preventing the mug from warming up to the desired temperature. This is due to the absence of a sustained heat source in the model. The initial hot fluid provides only a finite amount of thermal energy, which is rapidly redistributed and subsequently dissipated through the Robin boundary conditions. As a result, the system relaxes towards the ambient temperature.

Remarks on subtasks (ii) and (iii). In this project, we focused on subtask (i) as explicitly required in Task 9. Subtasks (ii) and (iii) would require modifying the initial condition at time T_{warm} and/or the Robin boundary coefficient α on the top boundary.

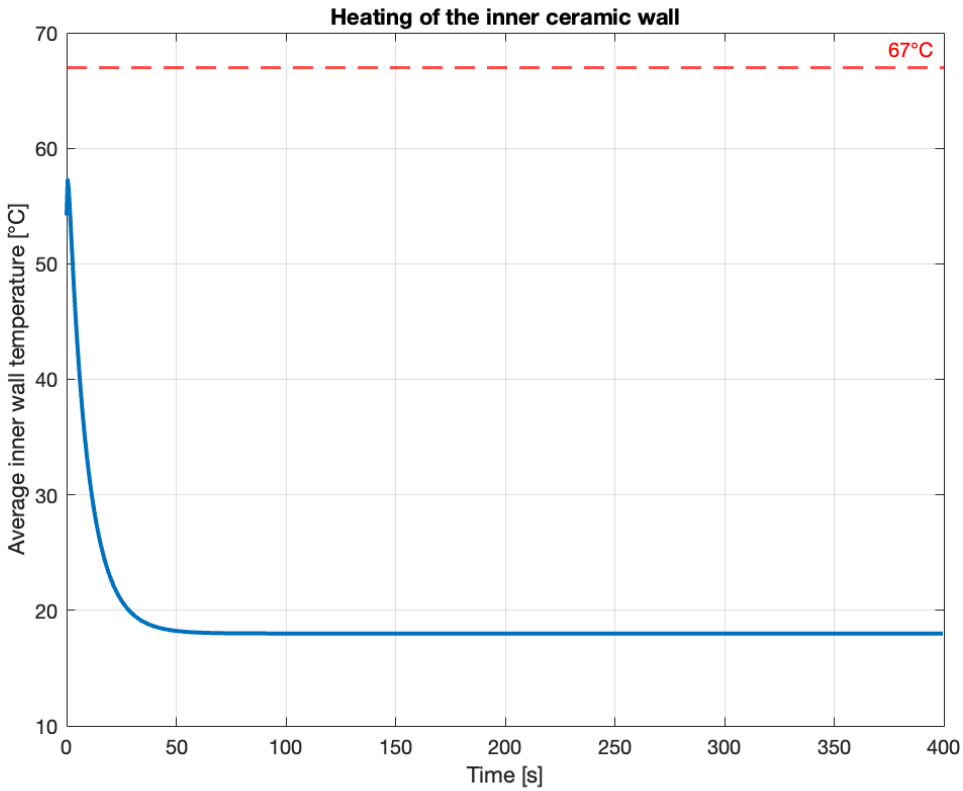

Figure 8: Transient temperature evolution