Modelling Temperature and Stresses in a Charging Battery

Finite Element Method and Introduction to Strength of Materials

FHLF25

Aron Allen och Erik Thylefors

Lunds Tekniska Högskola

May 29, 2024

Contents

Introduction **1**

Tasks 2

Procedure **2**

Stationary problem 7

Transient problem 7

Stress problem 8

Results **12**

Stationary problem 14

Transient problem 15

Stress problem 25

Discussion **29**

Stationary problem 29

Transient problem 29

Stress problem 30

Computer code **31**

constants.py 31

strategies.py 33

project.py 33

model.py 36

pre processor.py 38

solver.py 42

visualizer.py 49

Introduction

The goal of this project is to formulate and solve the heat equation and the von Mises stress equation using the finite element method on a mockup geometry of a battery cell.

The geometry consists of a trapezoid with seven cooling channels along the horizontal midline. Furthermore, the battery cell is symmetrical along the vertical midline.

The thermal boundary conditions are a series of natural conditions prescribing isolation along the bottom border and newton convection along the top border as well as the borders along the cooling channels. Furthermore, the temperature within the cooling channels as in figure 1 are $T_{in} = 277$ K and $T_{out} = 285$ K and the convection constant is $\alpha_c = 120$ W/m^{2}K. The boundary conditions along the top boundary are $T_\infty = 293$ K and $\alpha_n = 40$ W/m^{2}K. Along the symmetry line, we prescibe no heat flow by symmetry. The stress boundary conditions are both natural and essential. Along the top boundary zero traction is prescribed while along the bottom boundaries have zero displacement. The geometry and the boundary conditions are shown in figure 1.

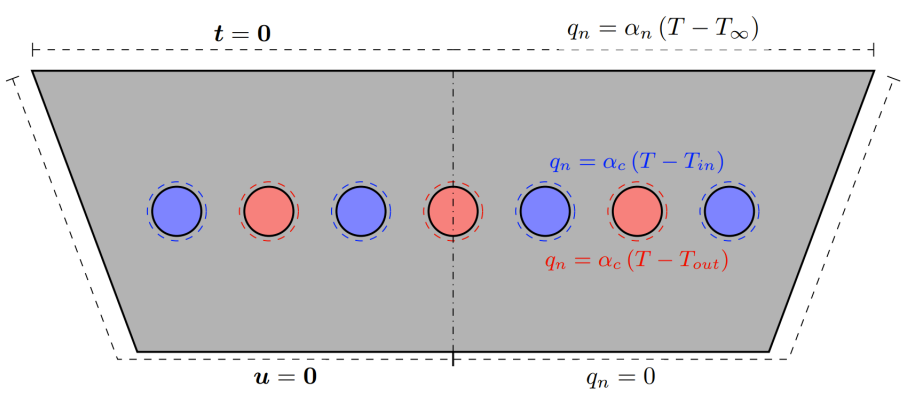

Figure 1: Geometry of the battery cell with boundary conditions. [2]

Tasks

The first task was to compute the stationary temperature distribution in the battery. To determine the maximum and minimum temperatures are in the battery, and to work out which parts of the battery that reaches these temperatures. Lastly we had to make out the largest temperature deviaton from the initial temperature.

The second task was to compute the transient temperature distribution of the first hour considering both of the charging strategies. Then we would plot the maximum and minimum temperatures as well as the largest temperature deviation from the ambient temperature over time. Finally we would plot the temperature distribution at six interesting time-steps.

The third task was to determine when the largest von Mises stresses would appear. Thereafter we would compute the displacement field and the von Mises stresses due to the thermal strains at the time of maxiumum temperature deviation from the inital state. Lastely we would determine wether or not the battery wold sustain at least one charging sequence if the material starts to yield at $\sigma_{eff} = 30$ MPa and to discuss the problems with the current design.

Procedure

The heat equation is a diffusion equation modelling the propagation of heat. In the general form we have [1]

$$c\rho\dot{T} + \mathrm{div}(\mathbf{q}) = f$$

for the density ρ and specific heat capacity c and heat supply f. Given a constitutive law for heat flow one can get a second order parabolic partial differential equation. We assume

that the constitutive relation is Fourier's law [3] which is given by

$$\mathbf{q} = -\mathbf{D}\nabla T \tag{1}$$

where $\mathbf{D}$ is a matrix that is used to model different materials, $\mathbf{q}$ the heat flow and T the temperature. We assume that the material is isotropic and homogenous which means that $\mathbf{D} = k\mathbf{I}$ for some positive constant k [3]. Using this constitutive relation we get the heat equation

$$c\rho\dot{T} - k\Delta T = f \tag{2}$$

where f is a function modelling heat production in the material. We find the weak formulation used for the finite element method by multiplying by an arbitrary smooth function and integrating the resulting relation over the whole space U resulting in

$$\int_U c\rho\dot{T}v - k\Delta T v \mathrm{dx} = \int_U f v \mathrm{dx}. \tag{3}$$

Integrating by parts gives

$$\int_U c\rho\dot{T}v\mathrm{dx} - \int_{\partial U} kv\frac{\partial T}{\partial \nu}\mathrm{dS} + \int_U k\nabla T^T \nabla v \mathrm{dx} = \int_U f v \mathrm{dx} \tag{4}$$

where ν is the outward pointing normal. If we let $\mathcal{L}_h$ be boundary with prescribed heat flow h and $\mathcal{L}_g$ be the boundary with prescribed temperature g we can rewrite this relation to

$$\int_U c\rho\dot{T}v\mathrm{dx} + \int_U k\nabla T^T \nabla v \mathrm{dx} = \int_{\mathcal{L}_h} kvh\mathrm{d}\mathcal{L} + \int_{\mathcal{L}_g} kv\frac{\partial T}{\partial \nu}\mathrm{d}\mathcal{L} + \int_U f v \mathrm{dx} \tag{5}$$

Next one creates a mesh of certain elements, in this case triangles, with the given form functions which must be able to model any fixed temperature and fixed heat flow on any

element [3]. These form functions are extended to global form functions recorded in a matrix $\mathbf{N}$. Prescribing the node temperatures in a vector $\mathbf{a}$ we get an approximation of the temperature

$$T_{approx} = \mathbf{N}\mathbf{a}$$

Using the Galerkin method we wish that our approximation solves the weak formulation for the subspace of our function space spanned by the form functions, hence for $\mathbf{v} = \mathbf{N}\mathbf{b} = \mathbf{b}^T\mathbf{N}^T$ [3]. If we let $B = \nabla N$ we get

$$\mathbf{b}^T\left(\int_U c\rho\mathbf{N}^T\mathbf{N}\dot{\mathbf{a}}\mathrm{dx} + \int_U \mathbf{B}^T k\mathbf{B}\mathrm{dx}\mathbf{a}\right) = \mathbf{b}^T\left(\int_{\mathcal{L}_h} k\mathbf{N}^T h\mathrm{d}\mathcal{L} + \int_{\mathcal{L}_g} k\mathbf{N}^T\frac{\partial T}{\partial \nu}\mathrm{d}\mathcal{L} + \int_U \mathbf{N}^T f\mathrm{dx}\right)$$

Since this equalty should hold for all vectors $\mathbf{b}$, the two matrices must be equal giving us the FEM formulation

$$\int_U c\rho\mathbf{N}^T\mathbf{N}\mathrm{dx}\dot{\mathbf{a}} + \int_U \mathbf{B}^T k\mathbf{B}\mathrm{dx}\mathbf{a} = \int_{\mathcal{L}_h} k\mathbf{N}^T h\mathrm{d}\mathcal{L} + \int_{\mathcal{L}_g} k\mathbf{N}^T\frac{\partial T}{\partial \nu}\mathrm{d}\mathcal{L} + \int_U \mathbf{N}^T f\mathrm{dx}$$

Defining the matrices

$$\mathbf{C} = \int_U c\rho\mathbf{N}^T\mathbf{N}\mathrm{dx}$$

$$\mathbf{K} = \int_U \mathbf{B}^T k\mathbf{B}\mathrm{dx}$$

and the vectors

$$\mathbf{f}_l = \int_U \mathbf{N}^T f\mathrm{dx}$$

$$\mathbf{f}_b = \int_{\mathcal{L}_g \bigcup \mathcal{L}_h} k\mathbf{N}^T\frac{\partial T}{\partial \nu}\mathrm{d}\mathcal{L}$$

$$\mathbf{f} = \mathbf{f}_l + \mathbf{f}_b$$

we get the standard formulation

$$\mathbf{C}\dot{\mathbf{a}} + \mathbf{K}\mathbf{a} = \mathbf{f} \tag{6}$$

However, for the actual problem we are not prescribed heat flow and temperatures along the boundaries but rather heat flow and convective boundary condiitons. This means that along some boundary we have the relation [3]

$$q = \alpha \Delta T = \alpha(T - T_\infty)$$

for some stable outer temperature T_∞ and heat transfer coefficient α with q being the heat flow over the boundary. Inserting this into the relation

$$-k\frac{\partial T}{\partial \nu} = -D\nabla T \cdot \nu = \mathbf{q} \cdot \nu = q = \alpha \Delta T$$

Thus, since $T = \mathbf{N}\mathbf{a}$ our FEM formulation becomes

$$\int_U c\rho \mathbf{N}^T \mathbf{N} \mathrm{dx} \partial_t \mathbf{a} + \int_U \mathbf{B}^T k \mathbf{B} \mathrm{dx} \mathbf{a} = \int_{\mathcal{L}_h} k \mathbf{N}^T h \mathrm{d}\mathcal{L} + \int_{\mathcal{L}_c} \mathbf{N}^T \alpha (T_\infty - \mathbf{N}\mathbf{a}) \mathrm{d}\mathcal{L} + \int_U \mathbf{N}^T f \mathrm{dx}$$

Since the thickness is constant and no quantities depend on the thickness we can integrate over the thickness of the body giving a factor t in each integral. This gives us that the FEM formulation is equivalent to

$$\mathbf{C}\dot{\mathbf{a}} + \mathbf{K}\mathbf{a} = \mathbf{f} \tag{7}$$

where we have defined the matrices

$$\mathbf{C} = \int_U c\rho \mathbf{N}^T \mathbf{N} t \mathrm{dx}$$

$$\mathbf{K} = \int_U \mathbf{B}^T k \mathbf{B} t \mathrm{dx} + \int_{\mathcal{L}_c} \mathbf{N}^T \alpha \mathbf{N} t \mathrm{d}\mathcal{L}$$

and the vectors

$$\mathbf{f}_l = \int_U \mathbf{N}^T f t \mathrm{dx}$$

$$\mathbf{f}_c = \int_{\mathcal{L}_c} \mathbf{N}^T \alpha T_\infty t \mathrm{d}\mathcal{L}$$

$$\mathbf{f}_b = \int_{\mathcal{L}_h} k \mathbf{N}^T h t \mathrm{d}\mathcal{L}$$

$$\mathbf{f} = \mathbf{f}_l + \mathbf{f}_c + \mathbf{f}_b$$

To get the formulation when the boundary conditions contain prescribed values, prescribed heat flows and convection one would simply add back the terms of the boundary $\mathcal{L}_g$ in the force vector $\mathbf{f}$. To calculate the convection contribution to the force vector and the stiffness matrix we calculate the integral of a form function along the boundary. Since the elements are triangles let the length of the side along the boundary be L and let the boundary be along the x axis. The form function for the element will then be given by.

$$N_1 = x/L, \quad N_2 = (x - L)/L$$

For the force vector these integrals become simply

$$\int_0^L N_1 \mathrm{dx} = \left[\frac{x^2}{2L}\right]_0^L = L/2$$

and

$$\int_0^L N_2 \mathrm{dx} = \left[\frac{x^2 - 2xL}{2L}\right]_0^L = -L/2$$

while for the stiffness matrix we must calculate the integral of the products

$$\int_0^L N_1 N_1 \mathrm{dx} = \int_0^L \frac{x^2}{L^2} \mathrm{dx} = \left[\frac{x^3}{3L^2}\right]_0^L = \frac{L}{3}$$

$$\int_0^L N_2 N_2 \mathrm{dx} = \int_0^L \left(\frac{x-L}{L}\right)^2 \mathrm{dx} = \left[\frac{1}{3}\left(\frac{x-L}{L}\right)^3\right]_0^L = \frac{L}{3}$$

$$\int_0^L N_1 N_2 \mathrm{dx} = \int_0^L \left(\frac{x-L}{L}\right) \frac{x}{L} \mathrm{dx} = \left[\frac{x^3}{3L^2} - \frac{x^2}{2L}\right]_0^L = \frac{L}{6}$$

Using these calculations one can compute the contribution to the stiffness matrix and convection vector from the convection boundary conditions.

Stationary problem

In the case of the stationary problem we have that $\dot{T} = 0$. Hence we get that our FEM formulation becomes

$$\mathbf{Ka} = \mathbf{f}$$

for the matrix $\mathbf{K}$ and $\mathbf{f}$ as above, taking into account the convective boundray conditions. This linear equation system can then be solved using CALFEM solveq function, or any other method for solving linear equation systems.

Transient problem

In the case of the transient problem a fully implicit time scheme was used. This means that we approximated the time derivative by a finite difference

$$\dot{\mathbf{a}} \approx \frac{\mathbf{a}(t+h) - \mathbf{a}(t)}{h}$$

for some time step h. This was then inserted into equation 7 evaluated at $t+h$ giving us the linear equation system

$$\mathbf{C}\frac{\mathbf{a}(t+h)-\mathbf{a}(t)}{h}+\mathbf{K}\mathbf{a}(t+h)=\mathbf{f}(t+h)$$

which is an implicit time scheme. [1] Solving for $\mathbf{a}(t+h)$ gives

$$\left(\mathbf{K}+\frac{\mathbf{C}}{h}\right)\mathbf{a}(t+h)=f(t+h)-\frac{\mathbf{C}}{h}\mathbf{a}(t) \tag{8}$$

which may then be solved numerically. The initial condition was given by $\mathbf{a}(0)=T_0=293K$.

Stress problem

We now turn to the stress problem. Here we also wish to formulate a weak solution of a differential equations and then proceed to find a FEM formulation of the problem. Since the battery has thickness of 1.6 m and 0.8 m wide at its widest, the problem is assumed to be well modelled by plane strain [3]. The stress problem is defined by the equilibrium equations [3]

$$\begin{cases} \partial_x\sigma_{xx}+\partial_y\sigma_{xy}+\partial_z\sigma_{xz}+b_x=0 \\ \partial_x\sigma_{yx}+\partial_y\sigma_{yy}+\partial_z\sigma_{yz}+b_x=0 \\ \partial_x\sigma_{zx}+\partial_y\sigma_{zy}+\partial_z\sigma_{zz}+b_x=0 \end{cases}$$

where $\sigma_{x_i x_j}$ are the components of the stress tensor and b_i are the force components. By introducting the derivative operator

$$\tilde{\nabla}^T = \begin{bmatrix} \partial_x & 0 & 0 & \partial_y & \partial_z & 0 \\ 0 & \partial_y & 0 & \partial_x & 0 & \partial_z \\ 0 & 0 & \partial_z & 0 & \partial_x & \partial_y \end{bmatrix}$$

This can be written more compactly as

$$\tilde{\nabla}^T \sigma + \mathbf{b} = 0$$

where we have written the stress components in Voigt notation

$$\sigma = \begin{bmatrix} \sigma_{xx} & \sigma_{yy} & \sigma_{zz} & \sigma_{xy} & \sigma_{xz} & \sigma_{yz} \end{bmatrix}^T.$$

Now to get a weak formulation we multiply by an arbitrary funtion v and integrate over the domain of our problem

$$\int_U v \tilde{\nabla}^T \sigma \mathrm{dx} + \int_U bv \mathrm{dx} = 0. \tag{9}$$

Integration by parts gives

$$\int_U (\tilde{\nabla}^T v) \sigma \mathrm{dx} = \int_{\partial U} v \mathbf{t} \mathrm{dx} + \int_U bv \mathrm{dx}$$

where $\mathbf{t}$ is the traction vector. Now we want to use this relation to solve for the displacements in the body, which can then be used to find the strains and stresses. Here we have the constitutive relation Hookes generalized law relating stresses and strains by

$$\sigma = \mathbf{D}\epsilon - \mathbf{D}\epsilon_0$$

where ϵ is the strain tensor in Voigt notation

$$\epsilon = \begin{bmatrix} \epsilon_{xx} & \epsilon_{yy} & \epsilon_{zz} & \gamma_{xy} & \gamma_{xz} & \gamma_{yz} \end{bmatrix}^T$$

ϵ_0 is the strain caused by thermal strain

$$\epsilon_0 = -\frac{E}{1-2\nu}\alpha T \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}^T$$

and $\mathbf{D}$ is the matrix

$$\mathbf{D} = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & \nu & 0 & 0 & 0 \\ \nu & 1-\nu & \nu & 0 & 0 & 0 \\ \nu & \nu & 1-\nu & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2}(1-2\nu) & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2}(1-2\nu) & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{2}(1-2\nu) \end{bmatrix}$$

.[3] Inserting this relation into our weak formulation gives

$$\int_U (\tilde{\nabla}^T v)\mathbf{D}\epsilon = \int_{\partial U} v\mathbf{t}\mathrm{dx} + \int_U bv\mathrm{dx} + \int_U (\tilde{\nabla}^T v)\mathbf{D}\epsilon_0\mathrm{dx}$$

Now we use the relation between stresses and strains $\epsilon = \tilde{\nabla}\mathbf{u}$ where $\mathbf{u}$ is the displacement vector. [3] Using this gives us a weak formulation in terms of our displacement vector

$$\int_U (\tilde{\nabla}^T v)\mathbf{D}\tilde{\nabla}\mathbf{u} = \int_{\partial U} v\mathbf{t}\mathrm{dx} + \int_U bv\mathrm{dx} + \int_U (\tilde{\nabla}^T v)\mathbf{D}\epsilon_0\mathrm{dx}.$$

Now we use that we have plane strain problem, i.e. there can only be strains in the plane so only $\epsilon_{xx}, \epsilon_{yy}$ and ϵ_{xy} are nonzero [3]. This gives us that our problem becomes two

dimensional with displacements only in the plane and we can integrate over the thickness of the body and get a factor t in each integral. We introduce a triangular grid for the problem and shape functions for each element that correspond to the x or y displacement for each node. This can be assembled into a global shape function $\mathbf{N}$ and our approximation becomes

$$\mathbf{u} \approx \mathbf{N}\mathbf{a}$$

for a vector $\mathbf{a}$ containing the displacements in x and y for each node. By using Galerkin's method we want this approximation to satisfy the weak equation for all functions in the same linear subspace spanned by the global shape functions, i.e. all functions v of the form $v = \mathbf{N}\mathbf{b}$ for some vector $\mathbf{b}$. This gives us

$$\mathbf{b}^T \left(\int_U (\mathbf{B}^T \mathbf{D} \mathbf{B} \mathbf{a} t \mathrm{dx} \right) = \mathbf{b}^T \left(\int_{\partial U} \mathbf{N}^T \mathbf{t} t \mathrm{dx} + \int_U b \mathbf{N}^T t \mathrm{dx} + \int_U \mathbf{B}^T \mathbf{D} \epsilon_0 t \mathrm{dx} \right)$$

where we have used the notation $\mathbf{B} = \tilde{\nabla}^T \mathbf{N}$. Since the relation must hold for arbitrary v, hence arbitrary $\mathbf{b}$ we get that the two matrices on each side must be equal, which gives us the FEM formulation.

$$\int_U \mathbf{B}^T \mathbf{D} \mathbf{B} t \mathrm{dx} \mathbf{a} = \int_{\partial U} \mathbf{N}^T \mathbf{t} t \mathrm{dx} + \int_U b \mathbf{N}^T t \mathrm{dx} + \int_U \mathbf{B}^T \mathbf{D} \epsilon_0 t \mathrm{dx}.$$

Introducing the matrices as in the heat problem we can rewrite it as

$$\mathbf{K}\mathbf{a} = \mathbf{f}$$

Now in the given problem we have $\mathbf{b} = 0$, hence the integral term containing that vanishes. The traction was prescribed for the upper boundary to be zero and the displacement for the lower part of the boundary was zero. By the symmetry, we can have no displacement

in x along the symmetry axis, nor any shear forces. To calculate the forces due to thermal strains, the temperature of each element at the time of maximal deviation from the original temperaure was approximated to be the mean value of the three temperatures of each node, from which we could calculate ϵ_0 for each element. Since we have triangular elements, the matrix $\mathbf{B}$ is constant and depend only upon the differences of coordinates whereby one could caluclate the force from thermal strain in each element and then solve the problem using CALFEM solveq.

Results

In this section plots of the temperature distribution as well as displacements and von misses stress are presented. Graphs of the maximum temperature, minimum temperature and maximum temperature deviation compared to the initial temperature are plotted against time.

An example result of the meshing using a element size factor of 0.01 of the problem is displayed in figure 2. In the plots shown below different element size factors were used in order either increase the resolution of the mesh or to limit the computation time and memory usage.

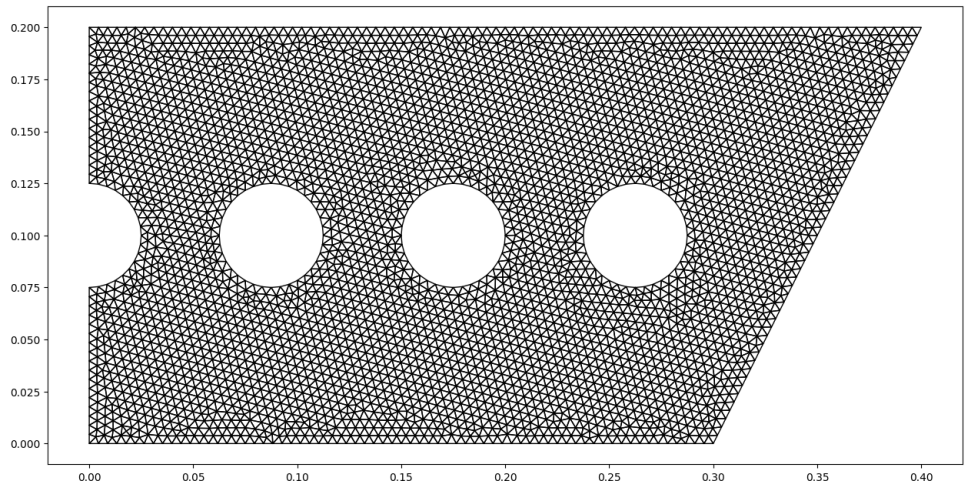

Figure 2: Example mesh of the geometry.

To establish the geometry certain markers were placed to outline the boundaries of the battery cell. This resulted in the CALFEM geometry shown in figure 3

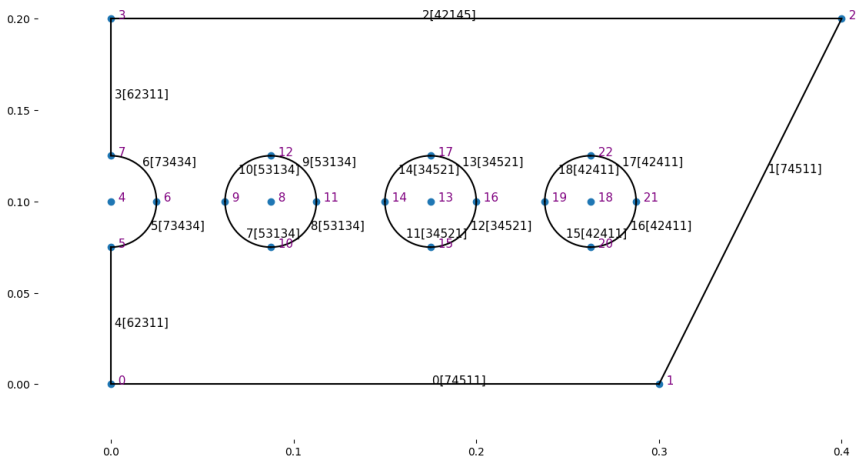

Figure 3: CALFEM geometry of the problem.

Stationary problem

In figure 4 the temperature distribution of the battery for the stationary problem is plotted. This is the solution given no heat flow along the lower boundary and Newtons convection along the top boundary as well as the boundary of the cooling channels. The maximum temperature was 293 K and the minimum temperature of the battery cell was 277 K. This was do to the boundary conditions with those temperatures.

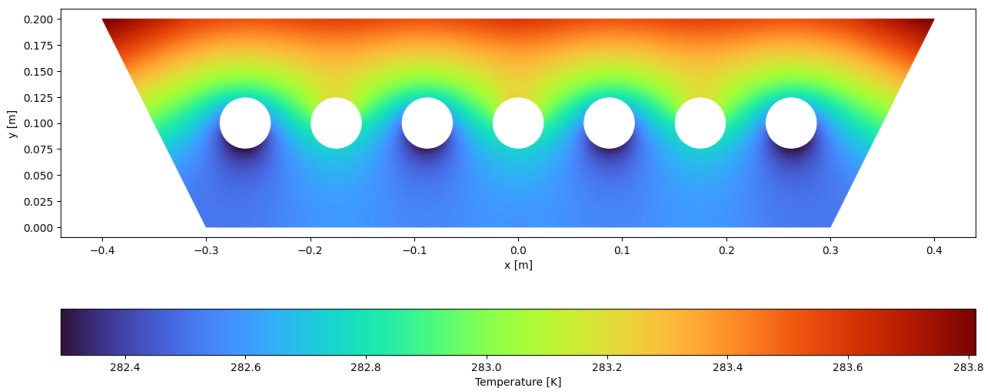

Figure 4: Temperature distribution for the stationary convection problem.

Transient problem

In figure 5 and figure 6 the temperature distributions at the time of largest temperature deviation form the initial temperature are plotted when using the smart charging strategy and normal charging strategy respectively. In figure 7 and figure 8 the temperature distributions at the time of the minimum temperature are plotted for the smart and normal charging strategy respectively. In figure 9 and 10 the maximum temperature deviation from the initial temperature using the smart strategy and the normal strategy are plotted respectively. In figure 11 and figure 12 the maximum temperatures are plotted using the smart and normal charging strategies respectively. In figure 13 and figure 14 the minimum temperatures are plotted using the smart and normal charging strategies respectively. In figure 15 and figure 16 the temperature distribution at time $t = 600$ s. In figure 17 and figure 18 the temperature distribution at time $t = 132$s is plotted. In figure 19 and figure

20 the temperature distribution at time $t = 1000$s is plotted. In figure 21 and figure 22 the temperature distribution at time $t = 2000$s is plotted.

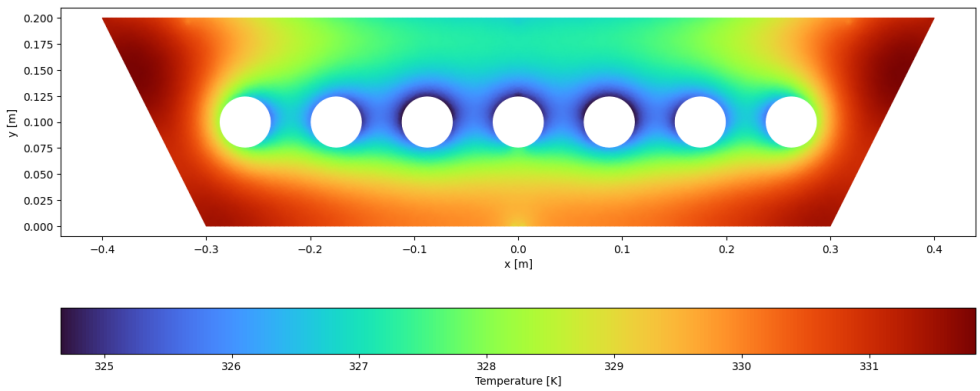

Figure 5: Temperature distribution for the transient problem using the smart charging strategy at maximum temperature deviation.

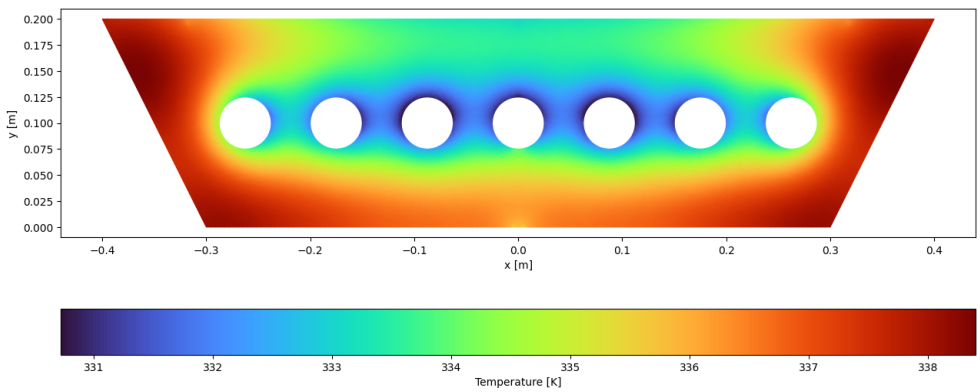

Figure 6: Temperature distribution for the transient problem using the normal charging strategy at maximum temperature deviation.

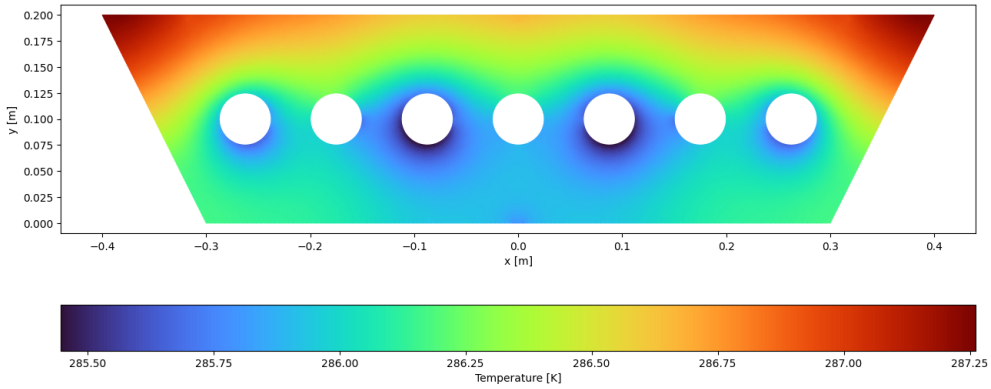

Figure 7: Temperature distribution for the transient problem using the smart charging strategy at minimum temperature.

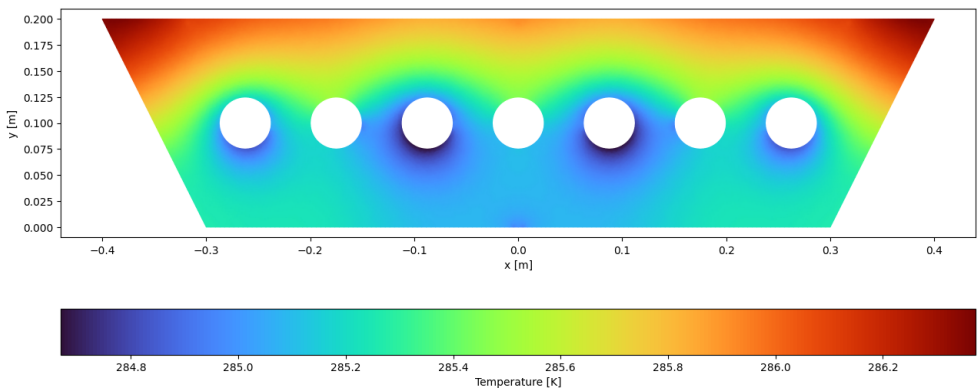

Figure 8: Temperature distribution for the transient problem using the normal charging strategy at minimum temperature.

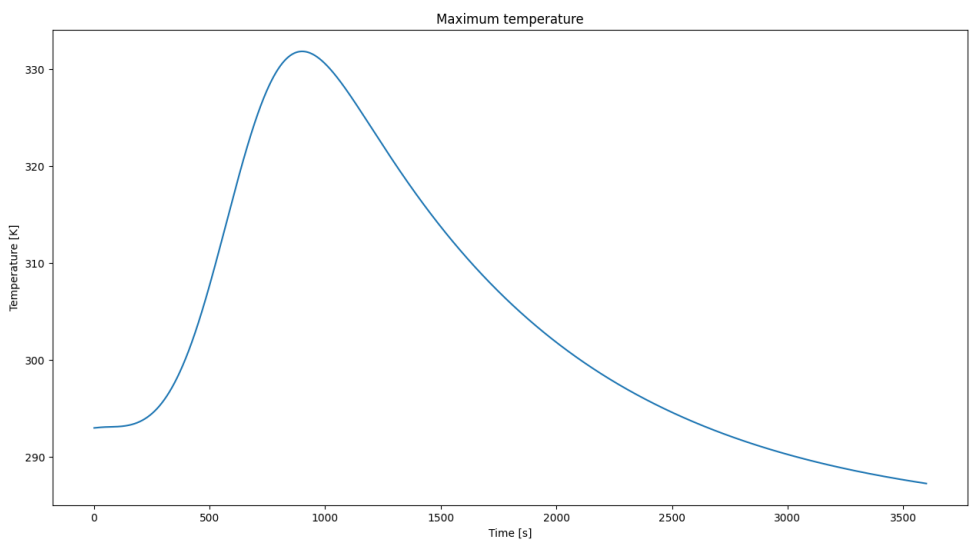

Figure 9: Maximum temperature deviation from the initial temperature using the smart charging strategy.

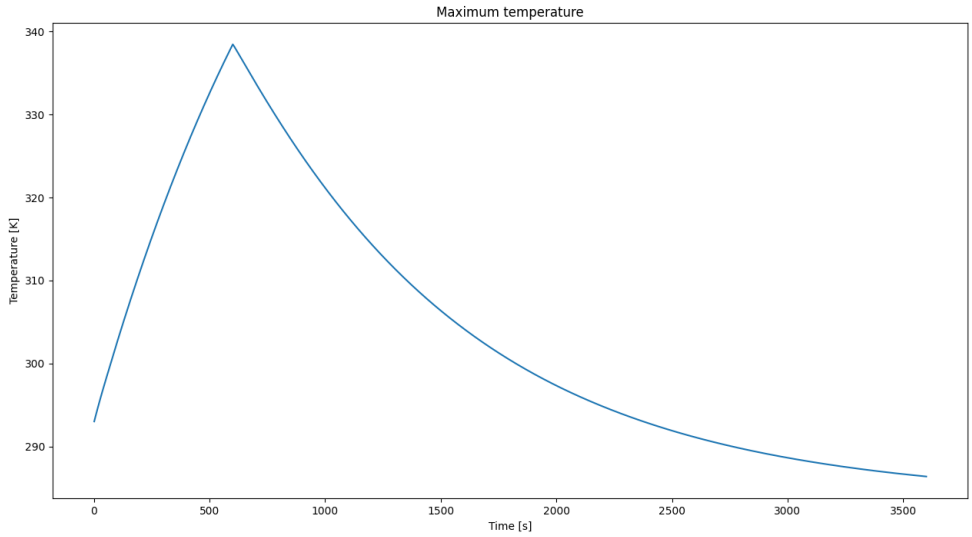

Figure 10: Maximum temperature deviation from the initial temperature using the normal charging strategy.

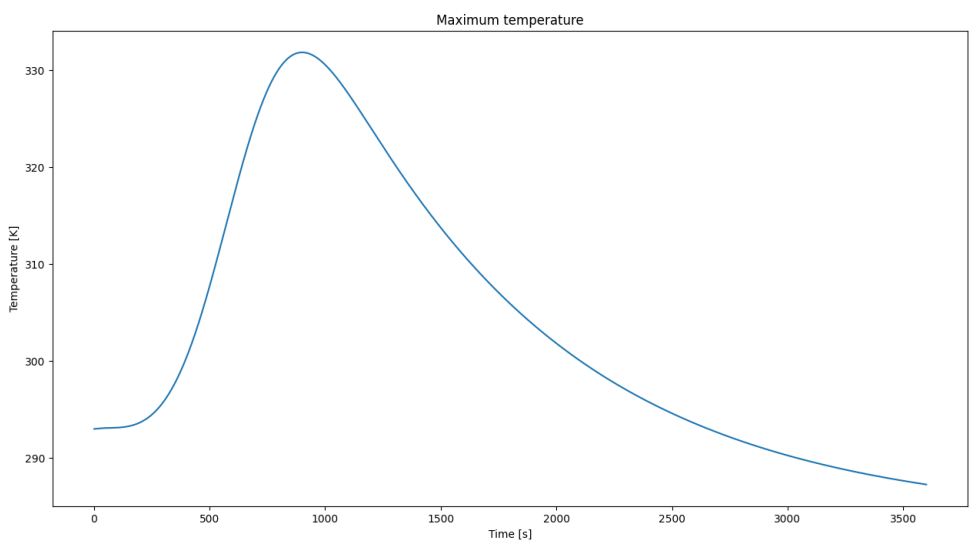

Figure 11: Maximum temperature using the smart charging strategy.

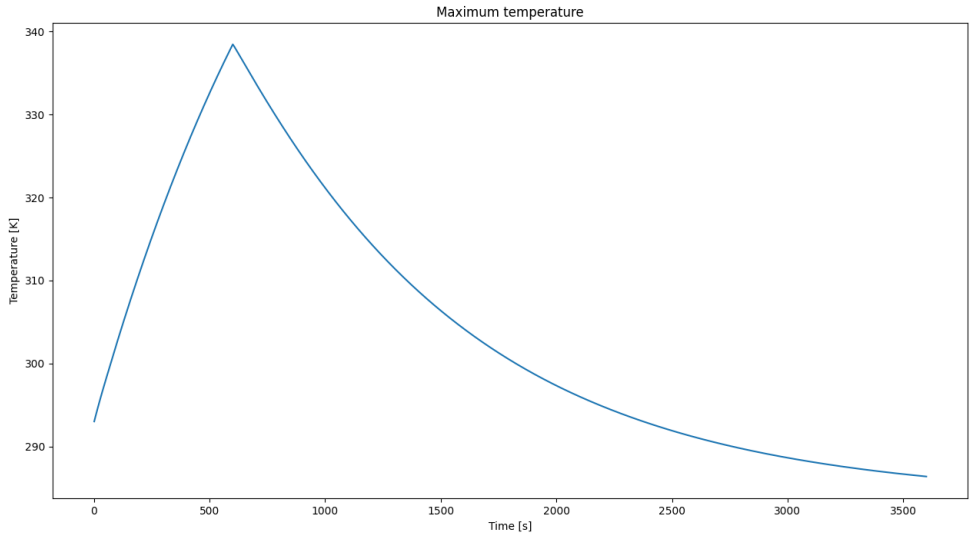

Figure 12: Maximum temperature using the normal charging strategy.

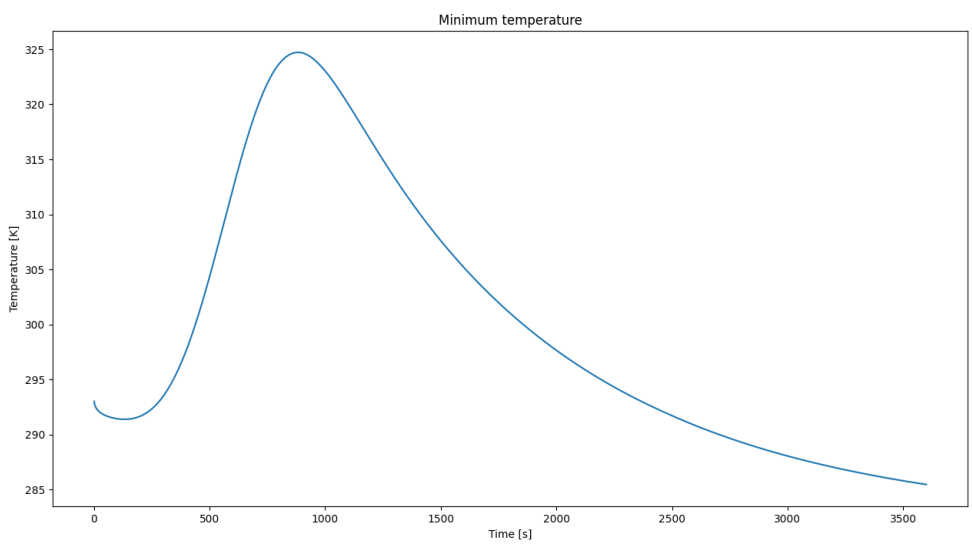

Figure 13: Minimum temperature using the smart charging strategy.

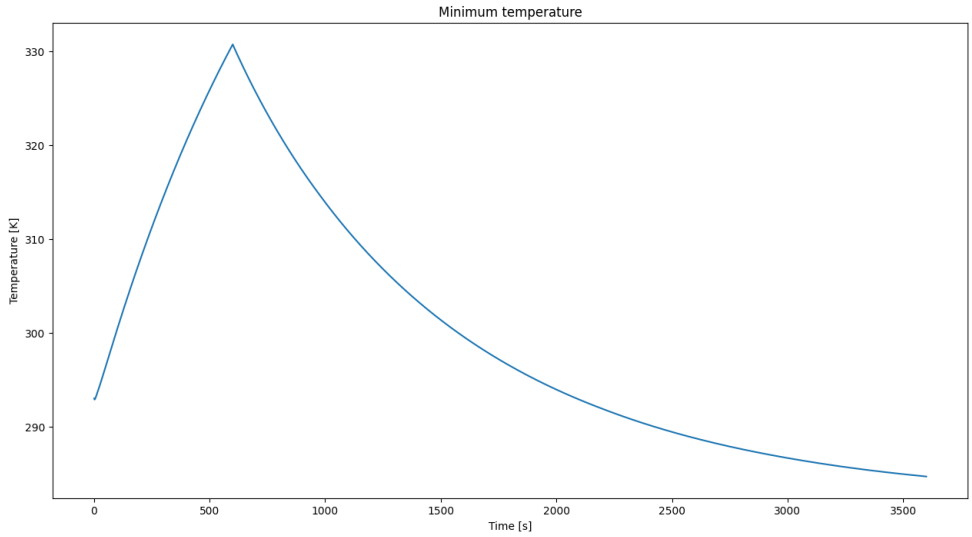

Figure 14: Minimum temperature using the normal charging strategy.

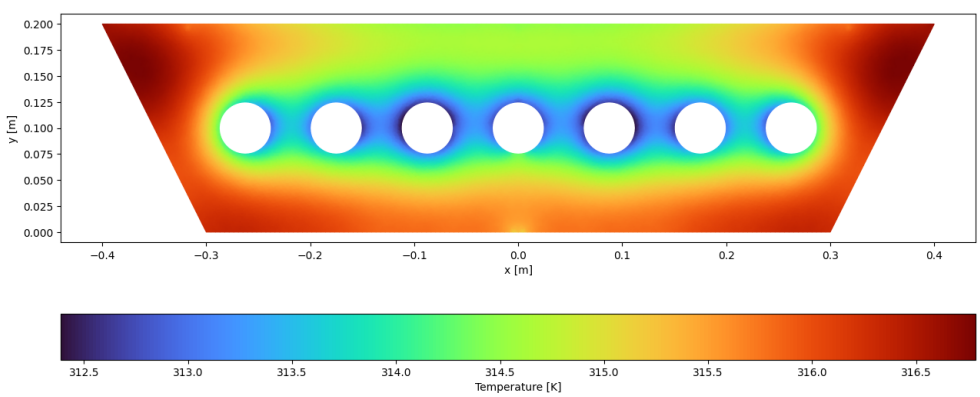

Figure 15: Temperature distribution using the smart charging strategy at time 600s

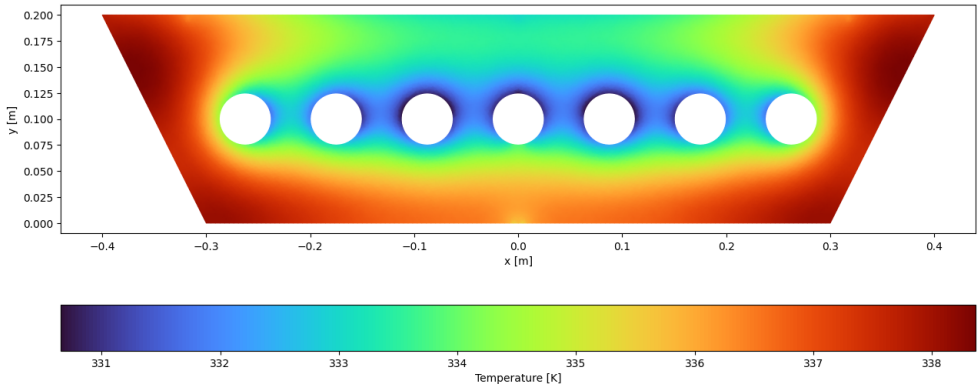

Figure 16: Temperature distribution using the normal charging strategy at time 600s

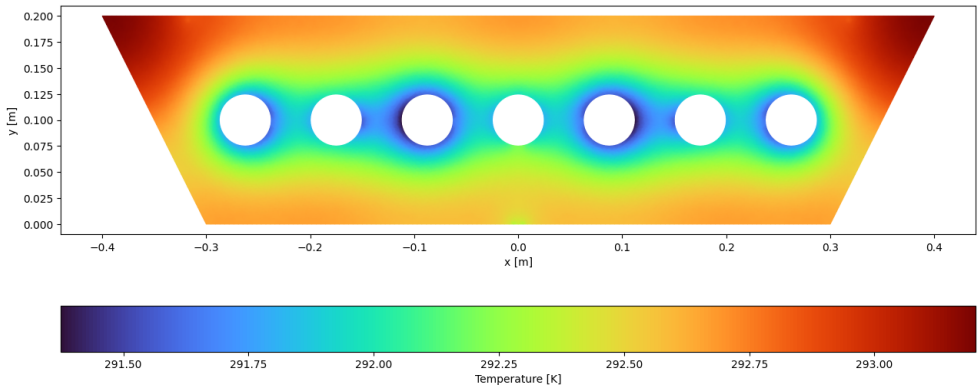

Figure 17: Temperature distribution using the smart charging strategy at time 132s

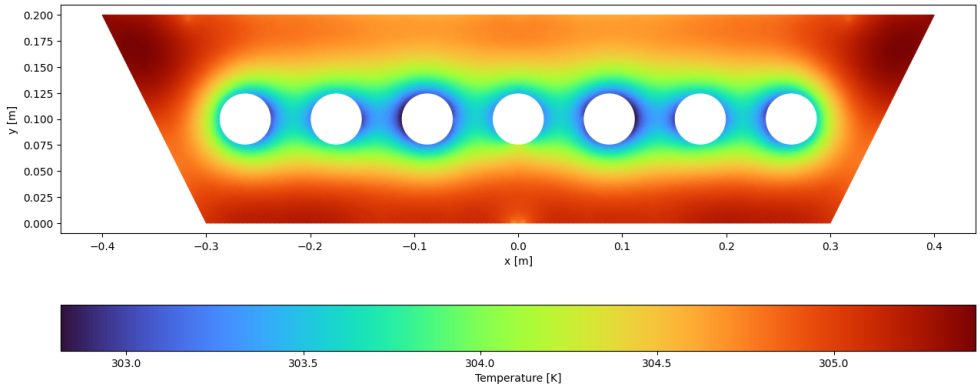

Figure 18: Temperature distribution using the normal charging strategy at time 132s

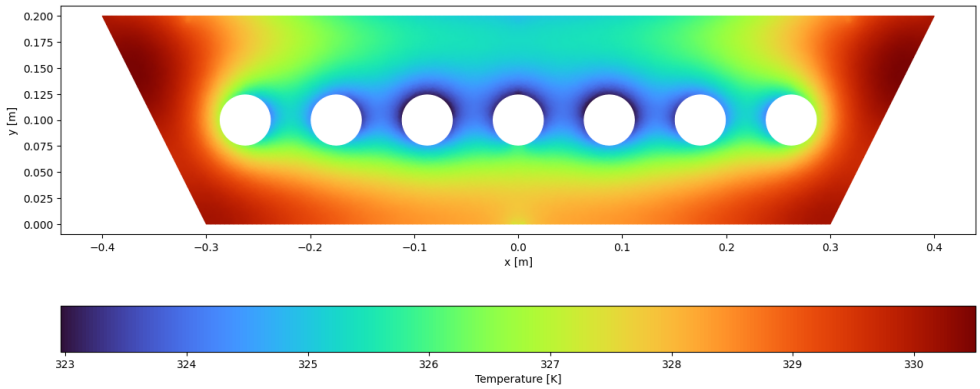

Figure 19: Temperature distribution using the smart charging strategy at time 1000s

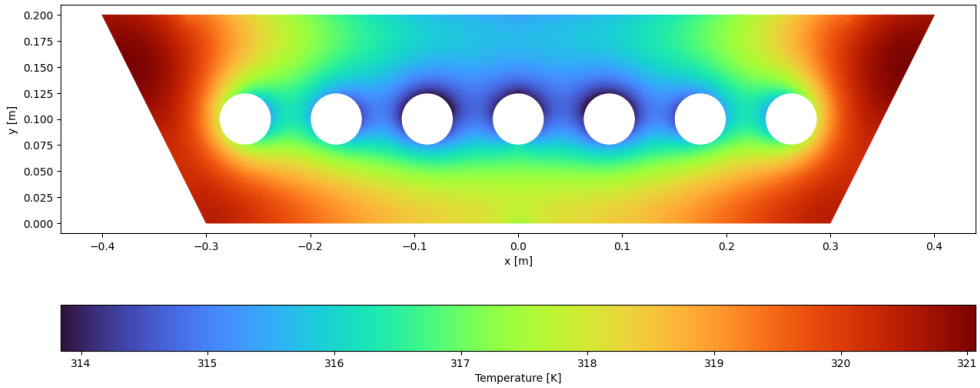

Figure 20: Temperature distribution using the normal charging strategy at time 1000s

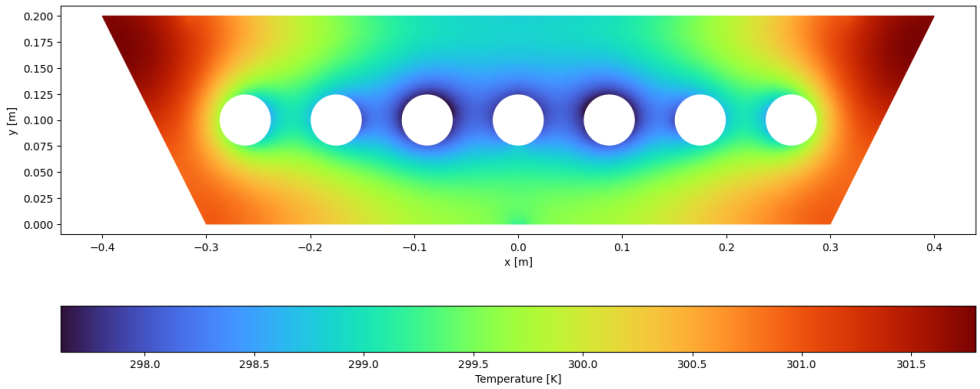

Figure 21: Temperature distribution using the smart charging strategy at time 2000s

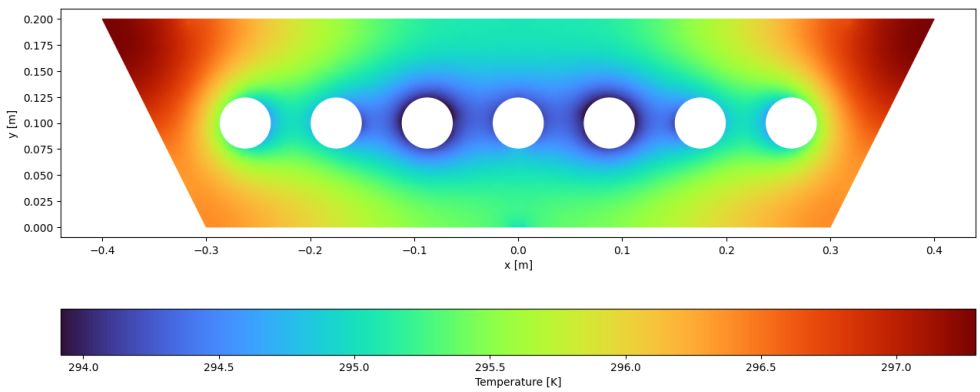

Figure 22: Temperature distribution using the smart charging strategy at time 2000s

Stress problem

All the displacement plots are using a magnification factor of 25 in order to show the results. In figure 23 and figure 24 the displacements at the time of maximum temperature deviation from the initial temperature is plotted respectively. In figure 25 and figure 26 the displacements and the von mises stress at the time of maximum temperature deviation from the initial temperature is plotted respectively. In figure 27 and figure 28 the von Mises stress at the time of maximum temperature deviation from the initial temperature is plotted respectively

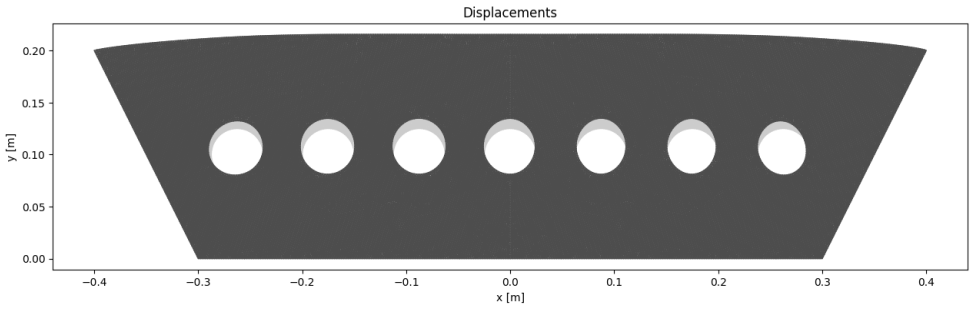

Figure 23: The displacements at the time of maximum temperature deviation using the smart charging strategy.

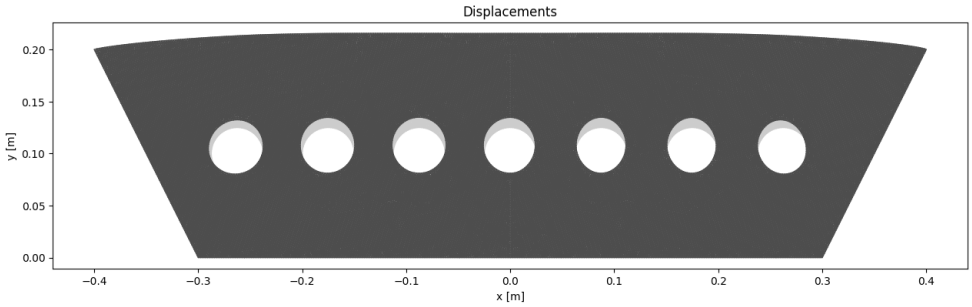

Figure 24: The displacements at the time of maximum temperature deviation using the normal charging strategy.

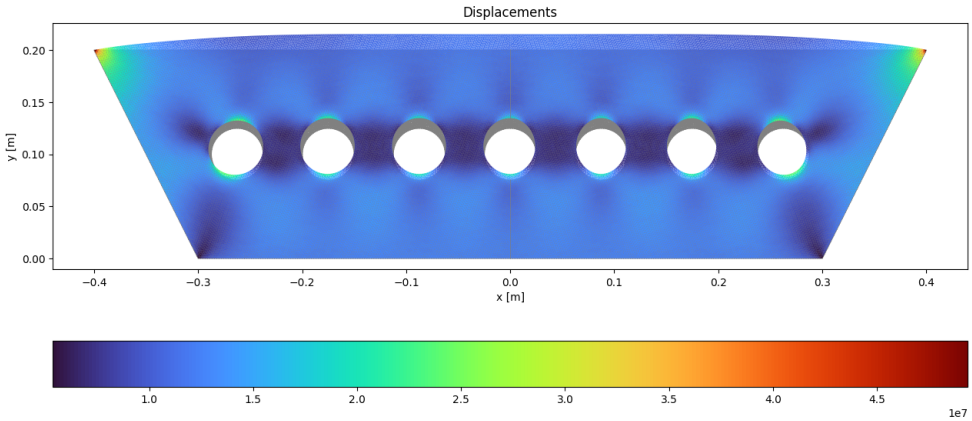

Figure 25: The displacements and the von Mises stress at the time of maximum temperature deviation using the smart charging strategy.

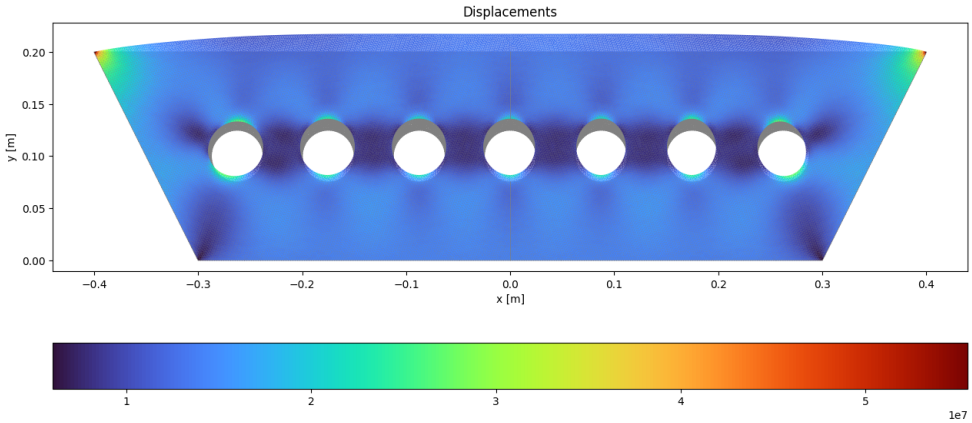

Figure 26: The displacements and the von Mises stress at the time of maximum temperature deviation using the normal charging strategy.

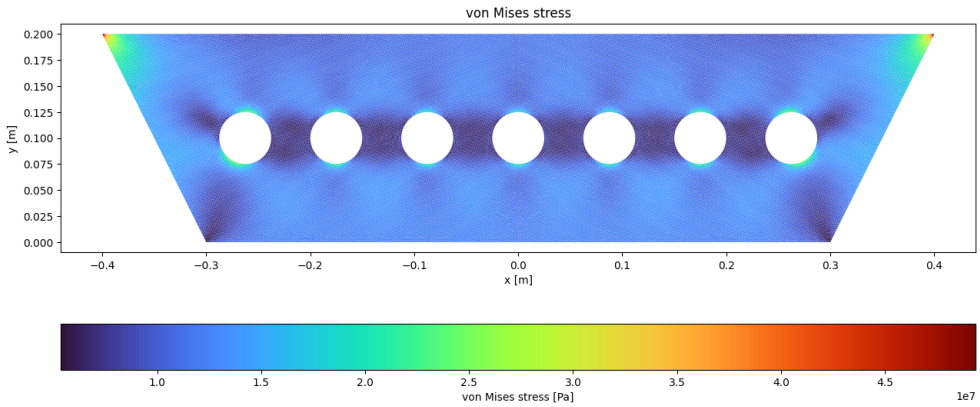

Figure 27: The von Mises stress at the time of maximum temperature deviation using the smart charging strategy.

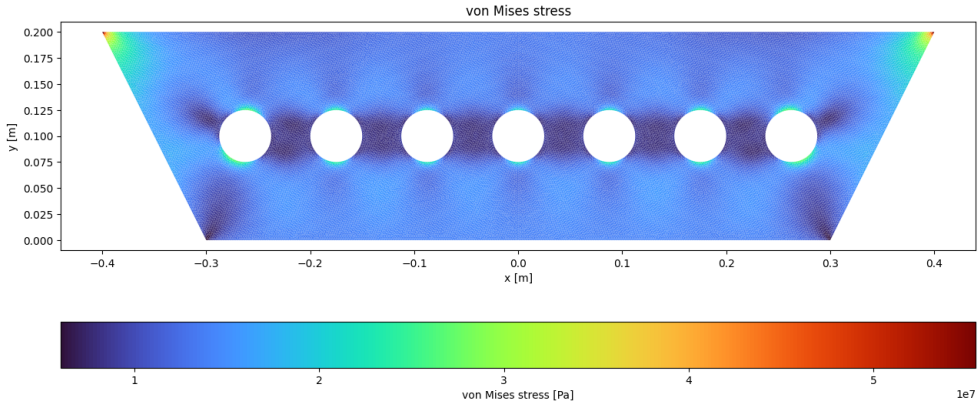

Figure 28: The von Mises stress at the time of maximum temperature deviation using the normal charging strategy.

Discussion

The result clearly shows a difference between the two charging stategies and how the temperature distribution evolves over time.

Stationary problem

For the stationary problem we see that the battery is at its coolest in the bottom part which is furthest away from the top boundary which has a convective heat flow from the outside temperature of 293 K. In general the temperature of the battery has lowered quite alot from the beginning temperature. The temperature in the battery is quite even which is to be expected from a stationary problem and the maximum and minimum temperatures occur along the boundary which agrees with the theory of harmonic functions.

Transient problem

The result shows that there is quite a large difference between the two charging strategies. Using the normal charging strategy, the battery is immediately heated and quickly reaches its maximum temperature at 600 s when the charging stops. It reaches temperatures upwards of 338 K at its maximum. After reaching its maximum temperature it quickly cools off. Interestingly the difference between minimum and maximum temperatures over time are quite constant meaning that the battery is heated up quite evenly and that the cooling of the cooling channels is quickly dispersed throughout the battery.

Using the smart charging strategy, the battery is at first cooled down when the cooling channels start cooling and is then succesively heated up. At around a 1000s the maximum temperature of the battery is achieved and it then cools off from the convection. At its maximum it reaches roughly 332 K which is much lower than the temperature reached by

the normal charging strategy.

Whether the normal or smart charging strategy is preferable is not obvious. The smart charging strategy reaches lower temperatures but is warmer longer. Hence the strategy that is preferable depends on wheter the battery is able to withstand higher temperatures or be warmer longer without a loss of ability to store energy.

To improve the model and get more reliable results one should take into consideration that the cooling channels probably will not immediately be able to cool at 277 K and that the temperature at which they carry the cooling liquid back will be dependent on the temperature at the borders of the cooling channel. Hence to improve the mathematical model one should probably try and model some of the dependence the cooling channel temperature has on the surrounding temperature. Furthermore if the battery is used in an enclosed space, the top boundary might not be well modelled as being exposed to room temperature since the battery is quite large and the heat leaving the battery will heat up the surroundings. These two modelling errors might produce temperatures that are too low using both charging strategies and could therefore cause there to be even more stress in the battery than assumed.

Stress problem

For the stress problem we see that both charging strategies make the battery break in the upper corners, however using the smart charging strategy, the stress is reduced somewhat. We see that the stress in the majority of the battery is well under the yielding limit but that the corners experience stress roughly twice the maximum. The cooling channels also experience some stress. This is to be expected since this is where we see large temperature differences and because of the boundary conditions we have prescribed. Since the top corners are part of two boundaries the corner element has one side that must be completly

fixed and one side that can experience no traction. Since the side not experienceing any traction must be able to move freely while the other side is fixed, the element must be stretched to account for the displacements, causing the high stress that we observe. This is a problem with the model since in reality we can not keep any boundaries fixed, however it is indicative that the battery would break if one tried charging it. To improve the model one could try to ascribe Robin boundary conditions modelling the possible displacements that the battery might experience along the boundary where no displacement is described. This would make the top corners no longer become singularities that experience more and more stress the finer the mesh becomes and is therefore probably a better model of the battery. One should also consider whether the battery ought to have its sides fixed or if one could try and let them move by having some springs trying to force the battery to remain in its original shape. This would reduce the stress that the top corners experience and might result in a battery that does not break when charging it.

Since the corners are singularities that experience more and more stress as the mesh size decreases, it is quite impossible to say whether the battery will break or not and is a source of error in the way we have described the boundary conditions. However, as stated above, it indicates that the battery probably will break.

Computer code

constants.py

```
# Arbitrary id:s for boundaries
left_element = 62311
bottomright_element = 74511
```

```
top_element = 42145
half_circle = 73434
circle1 = 53134
circle2 = 34521
circle3 = 42411

# Convection boundaries
boundary_markers = [top_element, half_circle, circle1, circle2, circle3]

# Boundary constants
T_0 = 293
T_inf = 293
T_in = 277
T_out = 285
T_initial = 293
a_c = 120
a_n = 40
qn = 0

# Geometry constants given in m
upper_length = 0.4
lower_length = 0.3
height = 0.2
thickness = 1.6
radius = 0.025
circle_distance = 0.0875

# Material properties
E = 5*1.e9
nu = 0.36
alpha = 60*1.e-6
```

```
rho = 540
cp = 3600
k = 80
alphan = 40
alphac = 120
```

strategies.py

```
import numpy as np

def f1(t):
    return 100*np.exp(-144*((600-t) / 3600)**2)*1000

def f2(t):
    if (600 - t) / 3600 > 0:
        return 88.42*1000
    else:
        return 0
```

project.py

```
import matplotlib as mpl
import numpy as np

from model import model
from solver import solver
from visualizer import visualizer
```

```
mpl.use('TkAgg')

# Parameters
N = 2001 # number of timesteps
t0 = 0  # initial time
t1 = 2001   # end time
elType = 2  # type of meshing element
elSizeFactor = 0.01 # size of mesh

# Create the geometry and mesh
geometry = model.geometry()
temperature_mesh = model.create_mesh(geometry=geometry, elType=elType, ...
    dofsPerNode=1, elSizeFactor=elSizeFactor) # mesh for heat equation
stress_mesh = model.create_mesh(geometry=geometry, elType=elType, ...
    dofsPerNode=2, elSizeFactor=elSizeFactor) # mesh for von mises problem

# Solver
temperature_solver = solver(temperature_mesh) # create solver object for ...
    the heat equation
a_stationary = temperature_solver.stationary_solver() # compute solution ...
    for stationary heat equation
a_convection = temperature_solver.convection_solver()   # compute ...
    solution for convection heat equation
A_transient, tgrid, tmax, tmin = temperature_solver.transient_solver(t0, ...
    t1, N)   # compute solution for transient heat equation

stress_solver = solver(stress_mesh) # create solver object for the von ...
    mises problem
vonMises, displacements = ...
    stress_solver.von_mises_solver(A_transient[tmax])    # compute ...
```

```
solution for the von mises problem

# Visualizer
vis = visualizer(geometry)  # create visualization object from the geometry
vis.draw_mesh(temperature_mesh)

# Visualize the solutions for the heat equation
vis.visualize_temperature_solution(data=a_stationary, ...
    mesh=temperature_mesh, vmin=np.min(a_stationary), ...
    vmax=np.max(a_stationary), draw_mesh=False, draw_geometry=False, ...
    cmap='turbo')
vis.visualize_temperature_solution(data=a_convection, ...
    mesh=temperature_mesh, vmin=np.min(a_convection), ...
    vmax=np.max(a_convection), draw_mesh=False, draw_geometry=False, ...
    cmap='turbo')
values = A_transient[tmax] # chose which time-frame to display
vis.visualize_temperature_solution(data=values, mesh=temperature_mesh, ...
    vmin=np.min(values), vmax=np.max(values), draw_mesh=False, ...
    draw_geometry=False, cmap='turbo')

# Visualize the solution for the von mises problem
vis.visualize_stress_solution(data=vonMises, disp=displacements, ...
    mesh=stress_mesh, dofsPerNode=stress_mesh.dofs_per_node, ...
    elType=elType, draw_values=True, draw_displacements=True)
vis.show()

# plot the transient temperature plot
vis.plotminmax(A_transient, tgrid, N)
```

model.py

```
import calfem.geometry as cfg
import calfem.mesh as cfm
from constants import left_element, bottomright_element, top_element, ...
    half_circle, circle1, circle2, circle3, upper_length, lower_length, ...
    height, thickness, radius, circle_distance

class model():

    # --Defining geometry--
    def geometry():
        g = cfg.Geometry()
        g.point([0.0,0.0])   # point 0
        g.point([lower_length, 0.0])  # point 1
        g.point([upper_length, height]) # point 2
        g.point([0.0, height]) # point 3

        g.spline([0,1], marker = bottomright_element) # line 0
        g.spline([1,2], marker = bottomright_element) # line 1
        g.spline([2,3], marker = top_element) # line 2

        # Half circle and lower boundary
        g.point([0.0, height/2]) # point 4
        g.point([0.0, height/2 - radius]) # point 5
        g.point([radius, height/2]) # point 6
        g.point([0.0, height/2 + radius]) # point 7

        g.spline([3,7], marker=left_element) # line 3
```

```
        g.spline([5,0], marker=left_element) # line 4
        g.circle([5,4,6], marker=half_circle) # line 5
        g.circle([6,4,7], marker=half_circle) # line 6

        # circle 1
        g.point([circle_distance, height/2])         # point 8
        g.point([circle_distance - radius, height/2])  # point 9
        g.point([circle_distance, height/2 - radius]) # point 10
        g.point([circle_distance + radius, height/2])  # point 11
        g.point([circle_distance, height/2 + radius]) # point 12

        g.circle([9,8,10], marker=circle1)   # line 7
        g.circle([10,8,11], marker=circle1)  # line 8
        g.circle([11,8,12], marker=circle1)  # line 9
        g.circle([12,8,9], marker=circle1)   # line 10

        # circle 2
        g.point([2*circle_distance, height/2])  # point 13
        g.point([2*circle_distance - radius, height/2]) # point 14
        g.point([2*circle_distance, height/2 - radius]) # point 15
        g.point([2*circle_distance + radius, height/2]) # point 16
        g.point([2*circle_distance, height/2 + radius]) # point 17

        g.circle([14,13,15], marker=circle2) # line 11
        g.circle([15,13,16], marker=circle2) # line 12
        g.circle([16,13,17], marker=circle2) # line 13
        g.circle([17,13,14], marker=circle2) # line 14

        # circle 3
        g.point([3*circle_distance, height/2])  # point 18
        g.point([3*circle_distance - radius, height/2]) # point 19
```

```
        g.point([3*circle_distance, height/2 - radius]) # point 20
        g.point([3*circle_distance + radius, height/2]) # point 21
        g.point([3*circle_distance, height/2 + radius]) # point 22

        g.circle([19,18,20], marker=circle3) # line 15
        g.circle([20,18,21], marker=circle3) # line 16
        g.circle([21,18,22], marker=circle3) # line 17
        g.circle([22,18,19], marker=circle3) # line 18

        g.surface([0,1,2,3,6,5,4], [[7,8,9,10],[11,12,13,14],[15,16,17,18]])

        return g

    # --Create the mesh--
    def create_mesh(geometry, elType, dofsPerNode, elSizeFactor):
        mesh = cfm.GmshMesh(geometry, return_boundary_elements=True)

        mesh.elType = elType # Type of meshing element.
        mesh.dofsPerNode = dofsPerNode # Degress of freedom per node.
        mesh.elSizeFactor = elSizeFactor # Element size Factor

        return mesh
```

pre processor.py

```
import calfem.utils as cfu
import calfem.core as cfc
import numpy as np
import scipy.sparse as sp
```

```
from plantml import plantml
from constants import top_element, half_circle, circle1, circle2, ...
    circle3, T_out, T_in, T_inf, boundary_markers, thickness, alphan, ...
    alphac, cp, rho, k

class pre_processor():
    def __init__(self, mesh):
        self.coords, self.edof, self.dofs, self.bdofs, ...
            self.elementmarkers, self.boundary_elements = mesh.create()
        self.num_dofs = len(self.dofs)

    def stationary_boundary_conditions(self):
        bc = np.array([], 'i')
        bcVal = np.array([], 'f')

        # Apply the boundary conditions for the stationary problem
        bc, bcVal = cfu.apply_bc(self.bdofs, bc, bcVal, half_circle, ...
            T_out, 0)
        bc, bcVal = cfu.apply_bc(self.bdofs, bc, bcVal, circle1, T_in, 0)
        bc, bcVal = cfu.apply_bc(self.bdofs, bc, bcVal, circle2, T_out, 0)
        bc, bcVal = cfu.apply_bc(self.bdofs, bc, bcVal, circle3, T_in, 0)
        bc, bcVal = cfu.apply_bc(self.bdofs, bc, bcVal, top_element, ...
            T_inf, 2)
        return bc, bcVal

    def convection_boundary_conditions(self):
        boundary_node_list = []
        K_c = np.zeros([self.num_dofs, self.num_dofs])
        F_c = np.zeros([self.num_dofs, 1])

        # Retrieve the boundary nodes
```

```
        for marker in boundary_markers:
            for i in range(len(self.boundary_elements[marker])):
                nodes = ...
                    self.boundary_elements[marker][i]['node-number-list']
                boundary_node_list.append([nodes, marker])

        # Compute and apply the boundary conditions for the convection ...
            problem
        for nodes in boundary_node_list:
            nx0, ny0 = self.coords[nodes[0][0] - 1]
            nx1, ny1 = self.coords[nodes[0][1] - 1]

            L = np.linalg.norm(np.array([nx1-nx0, ny1-ny0]))

            if nodes[1] == top_element:
              alpha = alphan
              temp = T_inf

            else:
                alpha = alphac

                if nodes[1] == half_circle or nodes[1] == circle2:
                    temp = T_out

                else:
                    temp = T_in

            K_c[nodes[0][0], nodes[0][0]] += alpha*thickness*L/3
            K_c[nodes[0][1], nodes[0][1]] += alpha*thickness*L/3
            K_c[nodes[0][0], nodes[0][1]] += alpha*thickness*L/6
            K_c[nodes[0][1], nodes[0][0]] += alpha*thickness*L/6
```

```python

            F_c[nodes[0][0]] += temp*alpha*thickness*L/2
            F_c[nodes[0][1]] += temp*alpha*thickness*L/2

        K_c = sp.csr_matrix(K_c)
        F_c = sp.csr_matrix(F_c)

        return K_c, F_c

    def transient_pre_processor(self):
        K = np.zeros([self.num_dofs, self.num_dofs])
        F = np.zeros([self.num_dofs, 1])
        V = np.zeros([self.num_dofs , 1])
        C = np.zeros([self.num_dofs, self.num_dofs])
        ex, ey = cfc.coordxtr(self.edof, self.coords, self.dofs)

        ptype = 1
        ep = [ptype, thickness]
        D = k * np.eye(2)

        # Compute the K and C matrix of the problem
        for eltopo, elx, ely in zip(self.edof, ex, ey):
            Ke = cfc.flw2te(elx, ely, ep, D)
            Ce = plantml(elx, ely, cp*rho)
            cfc.assem(eltopo, C, Ce)
            cfc.assem(eltopo, K, Ke)

        # Assign the same boundary conditions as in the convection problem
        K_c, F_c = self.convection_boundary_conditions()

        K += K_c
```

```
        F += F_c

        # Compute the surface integral of each element
        for e in self.edof:
            L1 = np.abs(np.subtract(self.coords[e[0] - 1], ...
                self.coords[e[1] - 1]))
            L2 = np.abs(np.subtract(self.coords[e[0] - 1], ...
                self.coords[e[2] - 1]))

            A = np.abs(L1[0] * L2[1] - L1[1] * L2[0])
            Ve = A*thickness/3

            V[e[0] - 1] += Ve
            V[e[1] - 1] += Ve
            V[e[2] - 1] += Ve

        return K, F, V, C
```

solver.py

```
import calfem.core as cfc
import calfem.utils as cfu
from pre_processor import pre_processor
import numpy as np
import scipy.sparse as sp
import colorama

from constants import bottomright_element, left_element, T_initial, ...
    thickness, k, E, nu, alpha
```

```
from strategies import f1, f2


class solver():

    def __init__(self, mesh):
        self.coords, self.edof, self.dofs, self.bdofs, ...
            self.elementmarkers, self.boundary_elements = mesh.create()
        self.num_dofs = np.size(self.dofs)
        self.prep = pre_processor(mesh)

    def stationary_solver(self):
        K = np.zeros([self.num_dofs, self.num_dofs])
        F = np.zeros([self.num_dofs, 1])
        ex, ey = cfc.coordxtr(self.edof, self.coords, self.dofs)
        ptype = 1
        ep = [ptype, thickness]
        D = k * np.eye(2)

        # Calculate the stiffness matrix for the problem
        for eltopo, elx, ely in zip(self.edof, ex, ey):
            Ke = cfc.flw2te(elx, ely, ep, D)
            cfc.assem(eltopo, K, Ke)

        bc, bcVal = self.prep.stationary_boundary_conditions()

        # Solve the equation system with stationary boundary conditions
        a, r = cfc.solveq(K, F, bc, bcVal)
        return a

    def convection_solver(self):
```

```
        K = np.zeros([self.num_dofs, self.num_dofs])
        F = np.zeros([self.num_dofs, 1])
        ex, ey = cfc.coordxtr(self.edof, self.coords, self.dofs)
        ptype = 1
        ep = [ptype, thickness]
        D = k * np.eye(2)

        # Assemble the stiffness matrix for the problem
        print('Assembling stiffness matrix...')
        idx = 0
        for eltopo, elx, ely in zip(self.edof, ex, ey):
            Ke = cfc.flw2te(elx, ely, ep, D)
            cfc.assem(eltopo, K, Ke)
            idx += 1
            self.progress_bar(idx, len(self.edof))

        # Assign the convection boundary conditions
        K_c, F_c = self.prep.convection_boundary_conditions()

        K += K_c
        F += F_c
        K = sp.csr_matrix(K)
        F = sp.csr_matrix(F)
        a = sp.linalg.spsolve(K, F) # Solve the equation system

        return a

    def transient_solver(self, t0, t1, N):
        print('Performing pre-processing...')
        K_b, F_b, V, C = self.prep.transient_pre_processor()
        A = np.zeros((N, self.num_dofs))
```

```
        h = (t1-t0) / N
        tgrid = np.linspace(t0, t1, N)[1:]

        K = K_b + C/h
        K = sp.csr_matrix(K)
        F = sp.csr_matrix(np.zeros([self.num_dofs, 1]))

        # Assign the initial temperatures
        for i in range(self.num_dofs):
            A[0][i] = T_initial

        # Solve the transient equation system
        print('Solving transient equation system...')
        for idx, t in enumerate(tgrid):
            F = F_b + V * f1(t) + np.array([np.matmul(C/h, A[idx])]).T
            a = sp.linalg.spsolve(K, F)
            A[idx + 1] = a
            self.progress_bar(t, t1)


        # Find the time-frame with maximum and minimum temperature
        tmax = 0
        tmin = 0
        B = np.abs(A-T_initial)
        prevmax = np.max(B[0])
        prevmin = np.min(A[0])
        for i in range(N):
            nextmax = np.max(B[i])
            nextmin = np.min(A[i])
            if nextmax > prevmax:
                prevmax = nextmax
```

```
                tmax = i
            if nextmin < prevmin:
                prevmin = nextmin
                tmin = i

        return A, tgrid, tmax, tmin

    def von_mises_solver(self, temperature):
        ptype = 2
        ep = [ptype, thickness]
        D = cfc.hooke(ptype, E, nu)
        ex, ey = cfc.coordxtr(self.edof, self.coords, self.dofs)
        K = np.zeros([self.num_dofs, self.num_dofs])
        F = np.zeros([self.num_dofs, 1])

        # Assemble the stiffness matrix for the stress problem
        print('Assembling stiffness matrix...')
        idx = 0
        for eltopo, elx, ely in zip(self.edof, ex, ey):
            Ke = cfc.plante(elx, ely, ep, D)
            cfc.assem(eltopo, K, Ke)
            idx += 1
            self.progress_bar(idx, len(self.edof))


        # Calculate the thermal stresses
        Δ_T_list = []
        idx = 0
        print('Calculating thermal stresses...')
        for element, elx, ely in zip(self.edof, ex, ey):
            Δ_T = 0
```

```
            for dof in element:
                Δ_T += (temperature[int(np.round(dof/2)) - 1] - ...
                    T_initial) / 6

            Δ_T_list.append(Δ_T)
            eps0 = (1)*alpha*Δ_T*np.array([1, 1, 0])
            xi, xj, xk = elx[0], elx[1], elx[2]
            yi, yj, yk = ely[0], ely[1], ely[2]

            B = 1/2*np.matrix(
                [[yj-yk, 0, yk-yi, 0, yi-yj, 0],
                 [0, xk-xj, 0, xi-xk, 0, xj-xi],
                 [xk-xj, yj-yk, xi-xk, yk-yi, xj-xi, yi-yj]]
            )
            D0 = np.matmul(np.delete(np.delete(D, 2, axis=0), 2, ...
                axis=1), eps0)
            f0 = np.matmul(D0, B) * thickness

            for dof, f in zip(element, f0.T):
                F[dof - 1] += f[0,0]
            idx += 1
            self.progress_bar(idx, len(self.edof))

        # Assign the boundary conditions for the stress
        bc = np.array([],'i')
        bcVal = np.array([],'f')
        bc, bcVal = cfu.applybc(self.bdofs, bc, bcVal, ...
            bottomright_element, 0, 0)
        bc, bcVal = cfu.applybc(self.bdofs, bc, bcVal, left_element, 0, 1)

        print('Solving for boundary conditions...')
```

```
        a,r = cfc.solveq(K,F,bc,bcVal)


        ed = cfc.extract_eldisp(self.edof, a)
        vonMises = np.zeros(self.edof.shape[0])

        print('Calculating von Mises stresses...')
        for i in range(self.edof.shape[0]):
            # Determine element stresses and strains in the element.
            es, et = cfc.plants(ex[i,:], ey[i,:], ep, D, ed[i,:])

            # Calculate and append effective stress to list.
            es = es[0]
            sigeff = np.sqrt(np.power(es[0],2) + np.power(es[1],2) + ...
                np.power(es[2],2) - es[0]*es[1] - es[0]*es[2] - ...
                es[1]*es[2] + 3*np.power(es[3],2))
            vonMises[i] = sigeff
            ## es: [sigx sigy tauxy]
            ## et: [epsx epsy gamxy]
            self.progress_bar(i, len(self.edof) - 1)

        return vonMises, a

    # Method of displaying the progress in the terminal
    def progress_bar(self, progress, total):
        percent = 100 * (progress / float(total))
        bar = '   ' * int(percent) + '-' * (100 - int(percent))
        color = colorama.Fore.RED
        if percent > 33:
            color = colorama.Fore.YELLOW
        if percent > 66:
```

```
            color = colorama.Fore.GREEN
        print(color + f"\r|{bar}| {percent:.2f}%", end='\r')
        if progress == total:
            print(colorama.Fore.MAGENTA + f"\r|{bar}| ...
                {percent:.2f}%", end='\r')
            print(colorama.Fore.RESET)
```

visualizer.py

```
import calfem.vis_mpl as cfv
import numpy as np
import matplotlib.pyplot as plt

from constants import T_initial

class visualizer():
    def __init__(self, geometry):
        self.geometry = geometry

    def visualize_temperature_solution(self, data, mesh, vmin, vmax, ...
        draw_mesh = True, draw_geometry = True, cmap = 'turbo'):
        coords, edof, dofs, bdofs, elementmarkers, boundary_elements = ...
            mesh.create()
        left_coords = np.array([[-c[0], c[1]] for c in coords])

        cfv.figure()

        # Apply the colormap
```

```
        cfv.draw_nodal_values_shaded(data, coords, edof, colormap = ...
            cmap, title='', vmin=vmin, vmax=vmax)
        cfv.draw_nodal_values_shaded(data, left_coords, edof, colormap = ...
            cmap, title='', vmin=vmin, vmax=vmax)
        cfv.colorbar('horizontal')

        # Draw the mesh
        if draw_mesh:
            cfv.drawMesh(
            coords=coords,
            edof=edof,
            dofs_per_node=mesh.dofsPerNode,
            el_type=mesh.elType,
            filled=False,
            title=""
            )

        # Showing geometry
        if draw_geometry:
            cfv.draw_geometry(self.geometry)

        # Set axis and colorbar label
        plt.xlabel('x [m]')
        plt.ylabel('y [m]')
        for c in cfv.gca().collections:
            if c.colorbar:
                c.colorbar.set_label('Temperature [K]')

    def visualize_stress_solution(self, data, disp, mesh, dofsPerNode, ...
        elType, draw_values = True, draw_displacements = False):
```

```
        coords, edof, dofs, bdofs, elementmarkers, boundary_elements = ...
            mesh.create()
        left_coords = np.array([[-c[0], c[1]] for c in coords])
        cfv.figure()
        if draw_values:
            cfv.draw_element_values(data, coords, edof, dofsPerNode, ...
                elType, None, draw_elements=False, ...
                draw_undisplaced_mesh=False, title="von Mises stress", ...
                cmap='turbo')
            cfv.draw_element_values(data, left_coords, edof, ...
                dofsPerNode, elType, None, draw_elements=False, ...
                draw_undisplaced_mesh=False, title="von Mises stress", ...
                cmap='turbo')
            cfv.colorbar('horizontal')

            plt.xlabel('x [m]')
            plt.ylabel('y [m]')
            for c in cfv.gca().collections:
                if c.colorbar:
                    c.colorbar.set_label('von Mises stress [Pa]')
        if draw_displacements:
            cfv.figure()
            #cfv.draw_mesh(coords, edof, dofsPerNode, elType, title='Mesh')
            cfv.draw_displacements(disp, coords, edof, dofsPerNode, ...
                elType, draw_undisplaced_mesh=True, ...
                title='Displacements', magnfac=25)
            cfv.draw_displacements(disp, left_coords, edof, dofsPerNode, ...
                elType, draw_undisplaced_mesh=True, ...
                title='Displacements', magnfac=25)
            plt.xlabel('x [m]')
            plt.ylabel('y [m]')
```

```
            cfv.figure()
            cfv.draw_element_values(data, coords, edof, dofsPerNode, ...
                elType, disp, draw_elements=False, ...
                draw_undisplaced_mesh=True, title="Displacements", ...
                cmap='turbo', magnfac=25)
            cfv.draw_element_values(data, left_coords, edof, ...
                dofsPerNode, elType, disp, draw_elements=False, ...
                draw_undisplaced_mesh=True, title="Displacements", ...
                cmap='turbo', magnfac=25)
            cfv.colorbar('horizontal')
            plt.xlabel('x [m]')
            plt.ylabel('y [m]')

    def draw_mesh(self, mesh):
        coords, edof, dofs, bdofs, elementmarkers, boundary_elements = ...
            mesh.create()
        left_coords = np.array([[-c[0], c[1]] for c in coords])
        cfv.figure()
        cfv.drawMesh(coords=coords, edof=edof, ...
            dofs_per_node=mesh.dofsPerNode, el_type=mesh.elType, ...
            filled=False, title="Mesh")
        cfv.drawMesh(coords=left_coords, edof=edof, ...
            dofs_per_node=mesh.dofsPerNode, el_type=mesh.elType, ...
            filled=False, title="Mesh")

    def show(self):
        cfv.show_and_wait()

    def plotminmax(self, data, tgrid, N):
        a = np.zeros(N-1)
```

```
        for i in range(N- 1):
            a[i] = np.max(data[i])
        plt.figure()
        plt.plot(tgrid,a)
        plt.title('Maximum temperature')
        plt.xlabel('Time [s]')
        plt.ylabel('Temperature [K]')
        plt.show()

        for i in range(N-1):
            a[i] = np.min(data[i])
        plt.figure()
        plt.plot(tgrid,a)
        plt.title('Minimum temperature')
        plt.xlabel('Time [s]')
        plt.ylabel('Temperature [K]')
        plt.show()

        for i in range(N - 1):
            a[i] = np.max(np.abs(data[i]-T_initial))
        plt.figure()
        plt.plot(tgrid,a)
        plt.title('Maximum temperature deviation')
        plt.xlabel('Time [s]')
        plt.ylabel('Temperature deviation [K]')
        plt.show()
```

References

[1] Division of Solid Mechanics, LTH. Finite element formulation of transient heat flow.

[2] Division of Solid Mechanics, LTH. Assignment in the finite element method, 2024. 2024.

[3] H. P. Niels Ottosen. *Introduction to Finite Element Method*. Pearson, Lund, 1992.