QSA Q-Bat - Tutorial 1

QuickerSim Automotive Ltd

Contents

1. Introduction

This tutorial presents how to create and simulate the heating of a simple battery geometry, consisting of one large cell, a cooling plate and surrounding air. The geometry is presented in Figure 1.

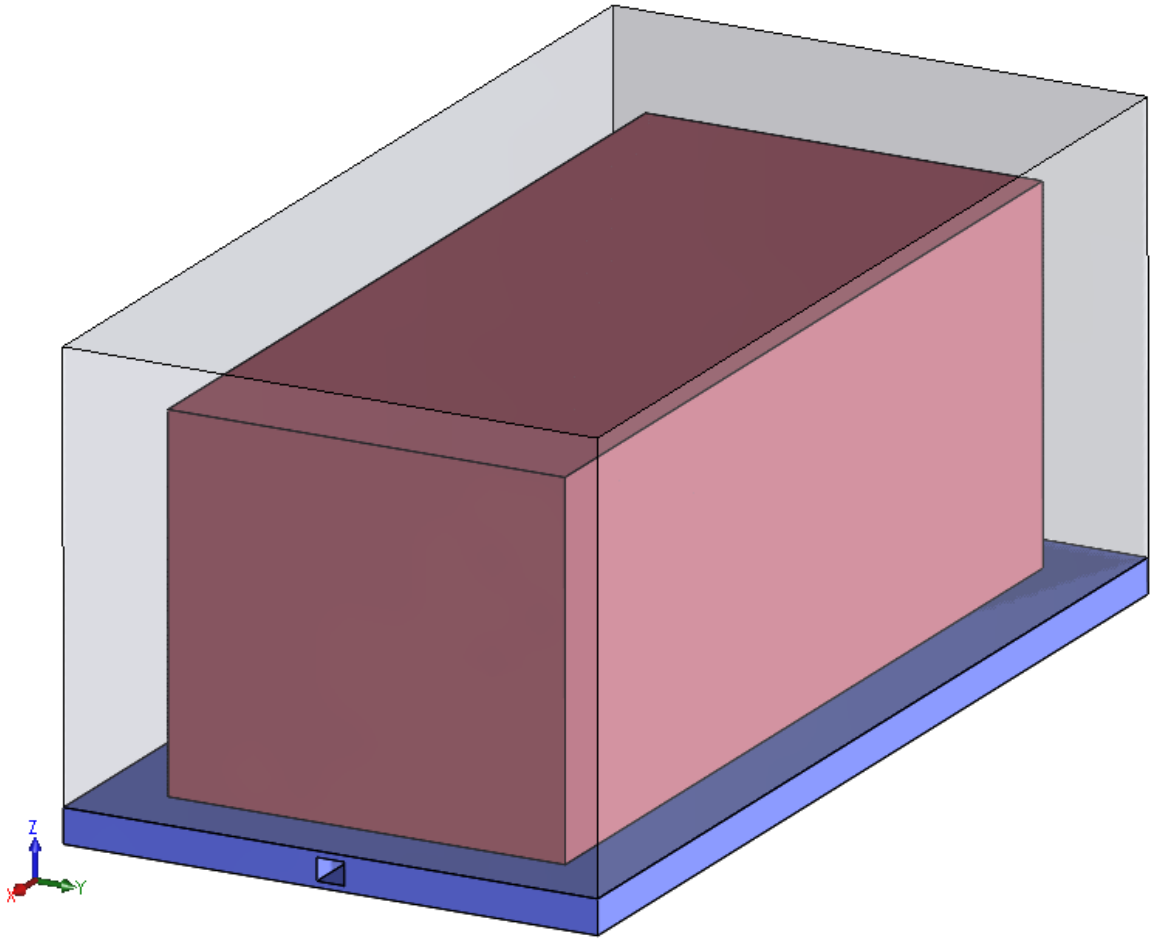


Figure 1: Battery assembly

Excel spreadsheets and msh files used in this tutorial have been prepared beforehand and are available as an attachment.

2. Preparing the model

Let's start by enforcing a good habit - clear your workspace and command window in Matlab and make sure all other windows are closed.

```
clc
clear
close all
```

The case described in this tutorial is not complicated and therefore does not require parallel toolbox.

2.1 Cell

We can now begin creating components of our assembly. First we will create the cell, which is represented by a simple prism, depicted on Figure 2.

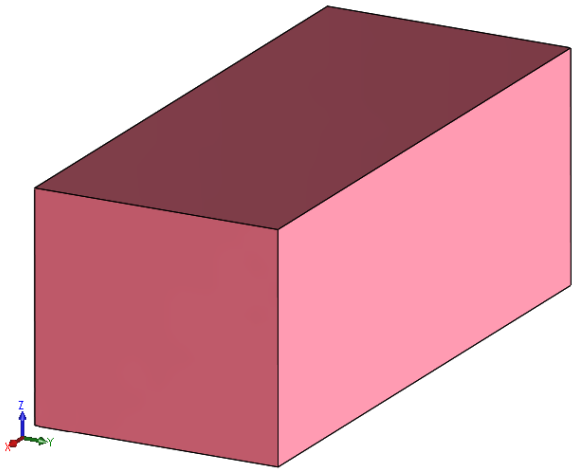


Figure 2: Cell

We have to create the cell prototype, which will be a virtual cell (not included anywhere in the model), with set properties and dimensions. We specify the type of our cell (prismatic or cylindrical) and its dimensions - the first three indices are the coordinates of the lower left corner of the prism and the other three - are its dimensions (x,y,z) in meters. It is also required to specify the material and electric properties of the cell, by appointing an Excel spreadsheet in which the data is collected.

```
cell_prototype = defineCellPrototype('type','prismatic','dimensions', ...
        [0, 0, 0, 0.1155, 0.264, 0.1055], 'data', ...
        'tutorials/tutorial1/data/cell_data.xlsx', 'heat_source', 10000);
```

```
Reading mesh from Gmsh, version = 4.1
Reading section: Entites...  Done!
Reading section: Nodes... Done!
Reading section: Elements...  Done!
Mesh read succesfully.
Material data rho depends on .
```

```
Material data cp depends on .
Material data lambda depends on .
Material data capacity depends on .
Warning: First resistance (R_0) data is missing in data structure
Warning: Second resistance (R_1) data is missing in data structure
Warning: Capacitance (C) data is missing in data structure
```

As we can see some the electric properties of the cell are missing. These warnings can be ignored (only in this tutorial), as in the presented case the heat source value is given by the User (it does not have to be calculated from the electric parameters and current).

The data in the spreadsheet must be compliant with a certain model. Parameters are specified by writing the parameter name in a cell and its value below. It is also possible to enter non-constant parameters, depending for example on temperature, what is presented in Tutorial 2.

	A	B	C	D	E	F	G	H	I	J	K	L
1												
2												
3				rho			cp			lambda		capacity
4				2000			733			2.7		20000.00
5												
6												
7												

Figure 3: Cell properties

Now we can display the created component and check the generated mesh.

```
cell_prototype.plot
```

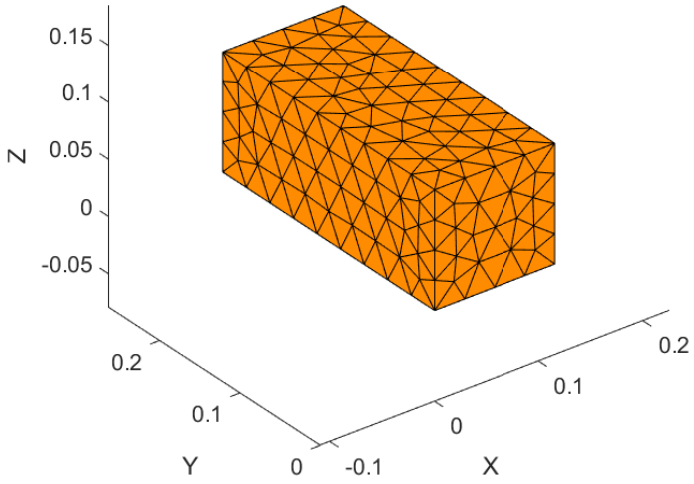


Figure 4: Cell mesh

Remember to save the cell prototype, in case you would like to use it in different models.

```
cell_prototype.saveComponent('tutorials\tutorial1\cell_proto')
```

In the next step we will have to put the cell in our simulation domain. We use the *instantiateInLocation* function and specify the location matrix of our cell. We must define the prototype which we want to use (in this case - the cell prototype).

```
cells = instantiateInLocation('prototype', cell_prototype, ...
    'location_matrix', [0.02, 0, 0]);
```

In order to be able to append the cells to our battery module we have to define them inside a cell casket.

```
cell_casket = defineCellCasket('cells', cells);
```

2.2 Cooling plate

After finishing the cell, we can move on to other components of the assembly. Next we will create the cooling plate, on which the cell stands (shown on Figure 5).

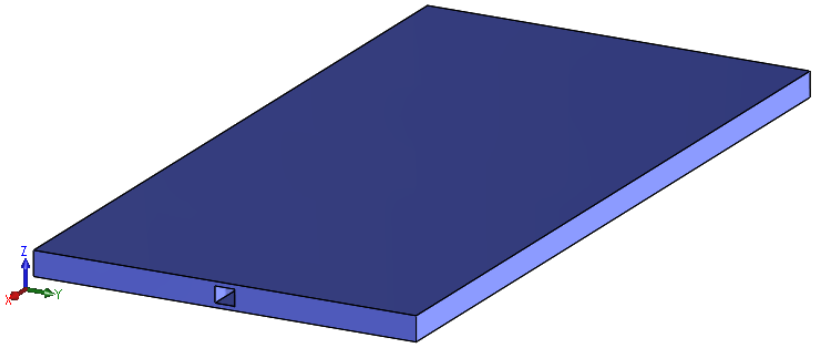


Figure 5: Cooling plate

In this tutorial the cooling plate geometry will be made using commands implemented in QSA Q-Bat. The geometry could be also loaded as a step or msh file (what is shown in Tutorial 2).

```
cp_box = qscad.makebox([0, 0, 0], [0.1555, 0.304, 0.02]);
pipe_box = qscad.makebox([0.07375, 0, 0.00675], [0.008, 0.304, 0.0065]);
cp_shape = qscad.booleandifference(cp_box, pipe_box);
```

While defining the prototype, we have to define the shape we just created, the pipe radius, mass flow rate, inlet temperature and material properties of the cooling plate and cooling fluid. Additionally, we define the size of the mesh and number of points on curvature to create a dense mesh. In this case the value of alpha (heat transfer coefficient) is known and does not need to be calculated in the software.

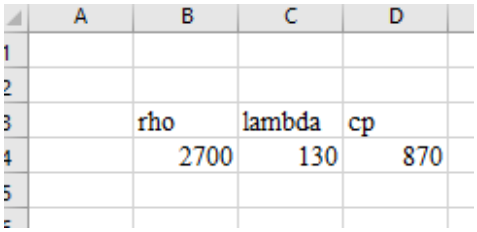
The pipe wall, inlet and outlet ids are know, so they can be defined by the User (if they are not known, the pipe walls and inlet/outlet can be chosen interactively, what is shown in Tutorial 2).

```
cooling_plate_prototype = defineCoolingPlatePrototype('qscad_shape', ...
    cp_shape, 'pipe_radius', 0.003586, 'pipe_mass_flow_rate', 0.02, ...
    'pipe_inlet_temperature', 15, 'predefined_alpha', 2000, ...
    'cooling_plate_data','tutorials/tutorial1/data/cp_data.xlsx', ...
    'coolant_data', 'tutorials/tutorial1/data/coolant_data.xlsx',...
    'mesh_size', 0.15, 'curv_points', 5,'pipe_wall_ids', [7, 10, 9, 8], ...
    'pipe_inlet_ids', [9, 8, 11, 10], 'pipe_outlet_ids', [17, 16, 19, 18]);
```

```
Material data lambda depends on .
Material data rho depends on .
Material data cp depends on .
Material data rho depends on T.
Material data cp depends on T.
Material data lambda depends on T.
```

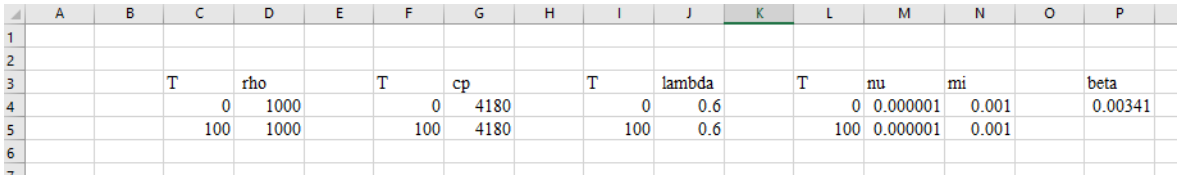
```
Material data nu depends on T.
Material data mi depends on T.
```

The Excel spreadsheets used in this step are shown below. To simplify the model, the cooling fluid's properties are assumed as constant and independent of the temperature.



	A	B	C	D
1				
2				
3		rho	lambda	cp
4		2700	130	870
5				

Figure 6: Cooling plate - material properties



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1																
2																
3			T	rho		T	cp		T	lambda		T	nu	mi		beta
4			0	1000		0	4180		0	0.6		0	0.000001	0.001		0.00341
5			100	1000		100	4180		100	0.6		100	0.000001	0.001		
6																

Figure 7: Cooling plate - cooling fluid properties

At this point it is very important to check if the pipe walls, inlet and outlet have been specified correctly.

```
cooling_plate_prototype.showPipeWall
```

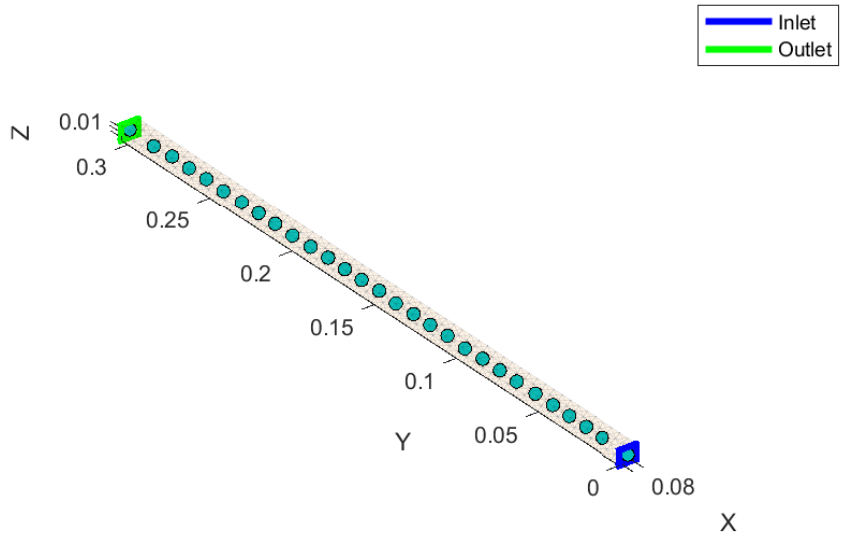


Figure 8: Cooling plate - pipe wall

Again, remember to save the cooling plate prototype:

```
cooling_plate_prototype.saveComponent('tutorials\tutorial1\cp_proto')
```

Once the prototype is created, we have to insert it into the simulation domain.

```
cooling_plate = instantiateInLocation('prototype', ...
    cooling_plate_prototype, 'location_matrix', [0, -0.02, -0.02]);
```

The component can now be displayed to make sure it has been created and placed correctly.

```
ui.utils.ComponentDisplayer.displayComponent(cooling_plate)
```

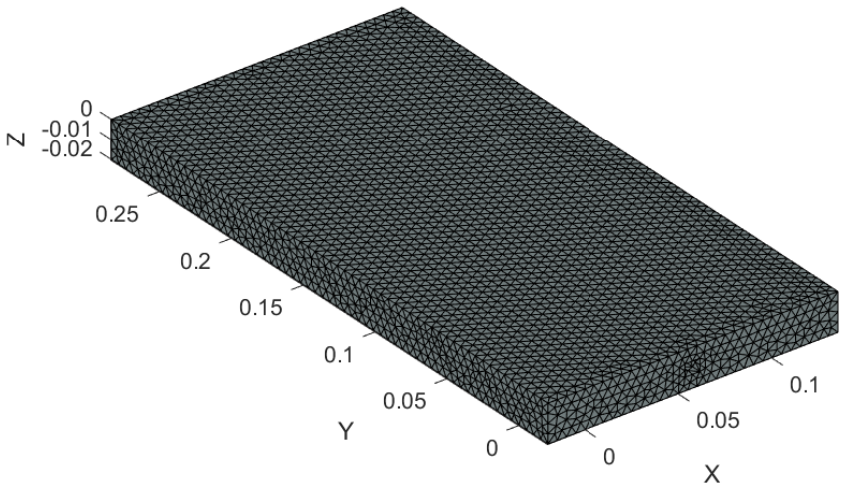


Figure 9: Cooling plate mesh

2.3 Heat component

The last component we need to create is the air surrounding the cell, which will be modeled as a passive heat component. A mesh has been created beforehand and its path will be given directly to the software.

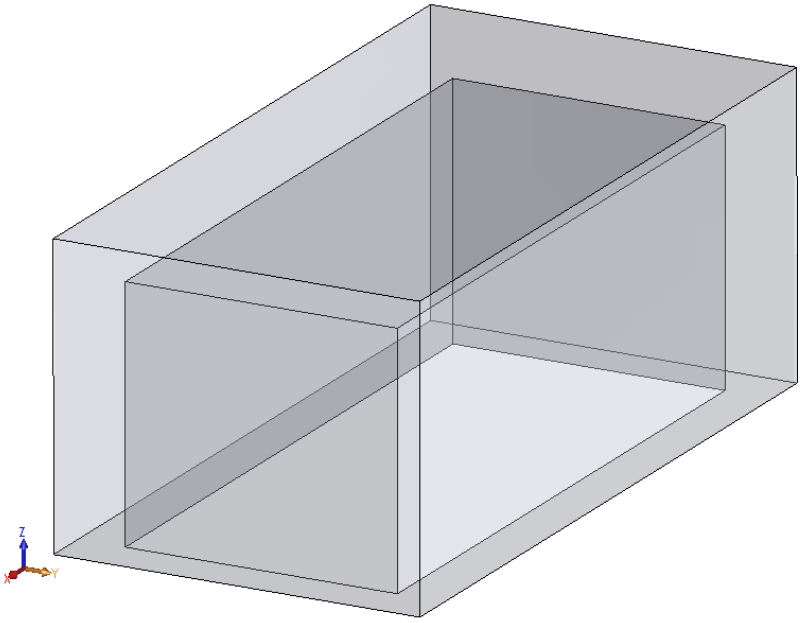


Figure 10: Air

Create the component in the same manner as before. The properties of the air are defined in an Excel spreadsheet (displayed on Figure 11).

	A	B	C	D	E	F	G	H	I	J
1										
2										
3				rho			cp			lambda
4				1.22			1000			0.024
5										
6										

Figure 11: Air - material properties

```
air_prototype = defineHeatComponentPrototype('mesh_path', ...
    'tutorials/tutorial1/msh/air.msh', 'heat_component_data', ...
    'tutorials/tutorial1/data/air_data.xlsx');
```

Remember to save the created prototype.

```
air_prototype.saveComponent('tutorials/tutorial1/air_proto');
```

The same way as before, we have to place the object into our simulation domain:

```
air = instantiateInLocation('prototype', air_prototype, ...
    'location_matrix',[0, 0, 0]);
```

The component must be translated, so it will be located in a proper place in the assembly.

```
air.translate([0, -0.02, 0.1255]);
```

The element is now placed correctly and can be displayed.

```
ui.utils.ComponentDisplayer.displayComponent(air)
```

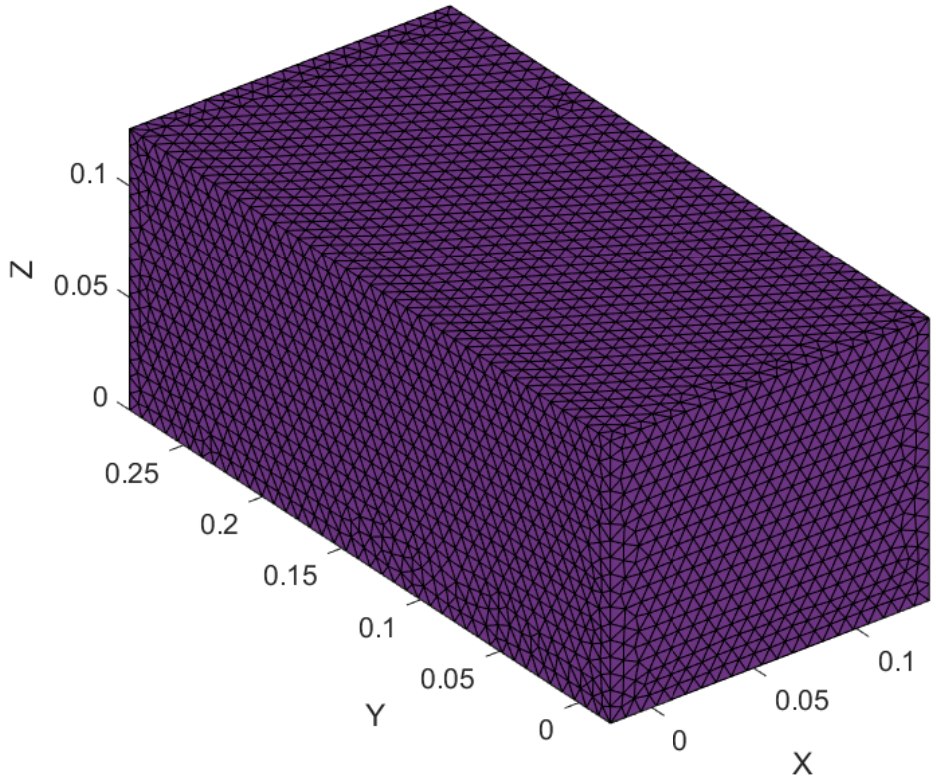


Figure 12: Air mesh

In this tutorial we do not specify and boundary conditions on the walls of our heat component. This way the outer walls will remain adiabatic. Adding boundary conditions to simulate forced convection around the model are presented in Tutorial 3.

2.4 Battery module

All of the components have now been created. We can combine them, to obtain the full battery module.

Specify the cell caskets, cooling plates and heat components we want to use. Contacts inside the module will be added separately, as their values are not all equal.

```
battery_module = defineBatteryModule('cell_caskets', cell_casket, ...
    'cooling_plates', cooling_plate, 'heat_components', air);
```

Contacts are specified by stating the bodies between which the contact occurs, and the thermal conductivity and thickness. We want to simulate a thermal pad between the cells and cooling plate, with conductivity equal to $3\frac{\mathrm{W}}{\mathrm{m}\cdot\mathrm{K}}$ and 0.001 m thickness. The parameters of contact between the cell and air (heat component) have been adjusted to describe natural convection. To simulate contact between the cooling plate and air (no thermal pad) a high value of summary conductance was chosen (ratio of thermal conductivity and contact thickness equal to 10^{12}).

```
battery_module.setCouplingBetweenCellCasketsAndCoolingPlate(3,1e-3);
battery_module.setCouplingBetweenCellCasketsAndHeatComponents(5,1);
battery_module.setCouplingBetweenCoolingPlatesAndHeatComponents(1e6,1e-6);
```

Using the *displayComponent* function we can display our newly created model.

```
ui.utils.ComponentDisplayer.displayComponent(battery_module)
```

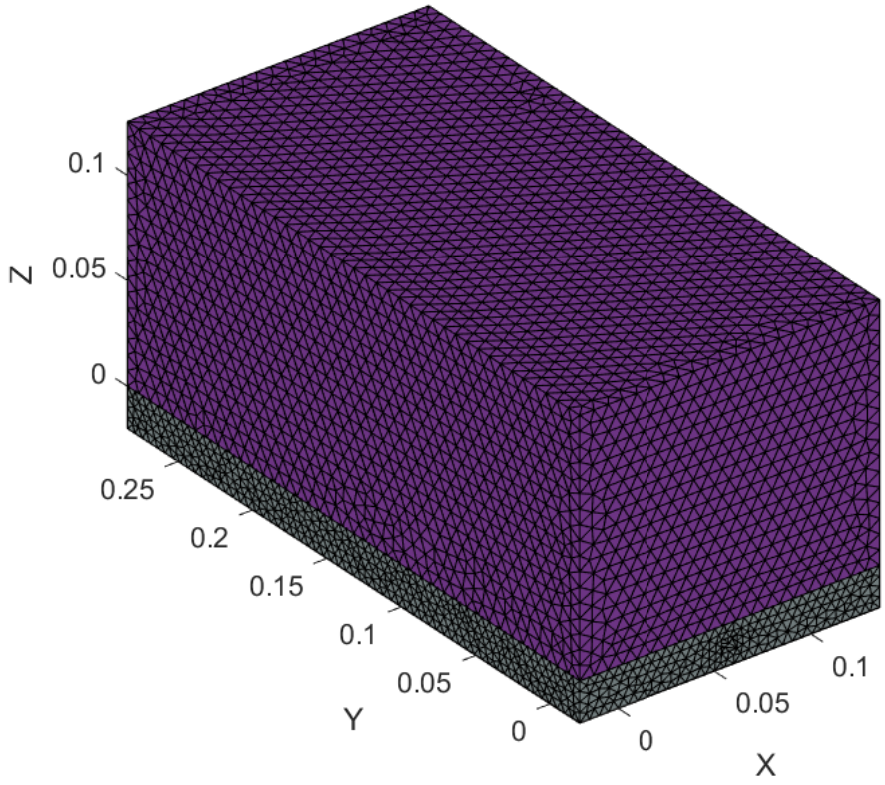


Figure 13: Battery module

Lastly, we set an initial temperature, that will be applied to the whole battery module.

```
battery_module.setInitialCondition(20);
```

Our battery module is done! We can now compile the model and run our simulation.

3. Calculations

3.1 Model assembly and reduction

The model is reduced and assembled using the *assembleModel* function. Adding *tic* and *toc* will allow to measure the assembly time.

```
t1 = tic;
m = assembleModel(battery_module);
t2 = toc(t1);
```

During the process various messages will be displayed in the command window. It is advised to read them, to assure the accuracy of the created model. It is specifically important to check if the number of contacts found by the software corresponds to the actual number of contacts in your model.

```
Found 1 thermal couples in BatteryModule: between CellCasket and
CoolingPlate.
Found 5 thermal couples in BatteryModule: between CellCasket and
HeatComponent.
Found 1 thermal couples in BatteryModule: between CoolingPlate and
HeatComponent.
Assembling domain 1. Number of nodes: 58744. Number of elements: 35221 ...
Elapsed time: 0.98319 s.
Assembling domain 2. Number of nodes: 53000. Number of elements: 30829 ...
Elapsed time: 0.85646 s.
Assembling domain 3. Number of nodes: 1807. Number of elements: 987 ...
Elapsed time: 0.042403 s.
Assembling couple 1
Elapsed time: 9.1382 s.
Assembling couple 2
Elapsed time: 3.5057 s.
Assembling couple 3
Elapsed time: 3.5539 s.
Assembling couple 4
Elapsed time: 1.6606 s.
Assembling couple 5
Elapsed time: 1.4923 s.
Assembling couple 6
Elapsed time: 4.0121 s.
Assembling couple 7
Elapsed time: 24.3793 s.
Assembling thpipe couple 1
Elapsed time: 0.21104 s.
Reducing domain 1. Number of nodes: 58744. Number of elements: 35221 ...
Elapsed time: 12.5668 s.
Reducing domain 2. Number of nodes: 53000. Number of elements: 30829 ...
Elapsed time: 21.4534 s.
Reducing domain 3. Number of nodes: 1807. Number of elements: 987 ...
Elapsed time: 0.20228 s.
```

```
Assembling and reducing nonlinear domain 1 out of 3
Elapsed time: 0.0004127 s.
Assembling and reducing nonlinear domain 2 out of 3
Elapsed time: 9.13e-05 s.
Assembling and reducing nonlinear domain 3 out of 3
Elapsed time: 8.83e-05 s.
```

3.2 Running the calculations

Once the model is compiled we can run the simulation. We specify the time step in seconds and the number of time steps we want to simulate. In this tutorial we want to simulate a steady state (the heat source is constant), therefore we choose a very large time step.

```
m.runReducedMerged(1000, 100);
```

Congratulations, your simulation is done!

3.3 Post-processing and export

To have a clear space for post-processing we can close all the figures and define our axes ratio.

```
close all
cla
ax = gca;
```

By using different post processing functions we can plot out the solution. Let's plot the coolant temperature profile throughout the pipe for the last time step.

```
plotCoolantTempProfile(cooling_plate, m, 100, ax)
```

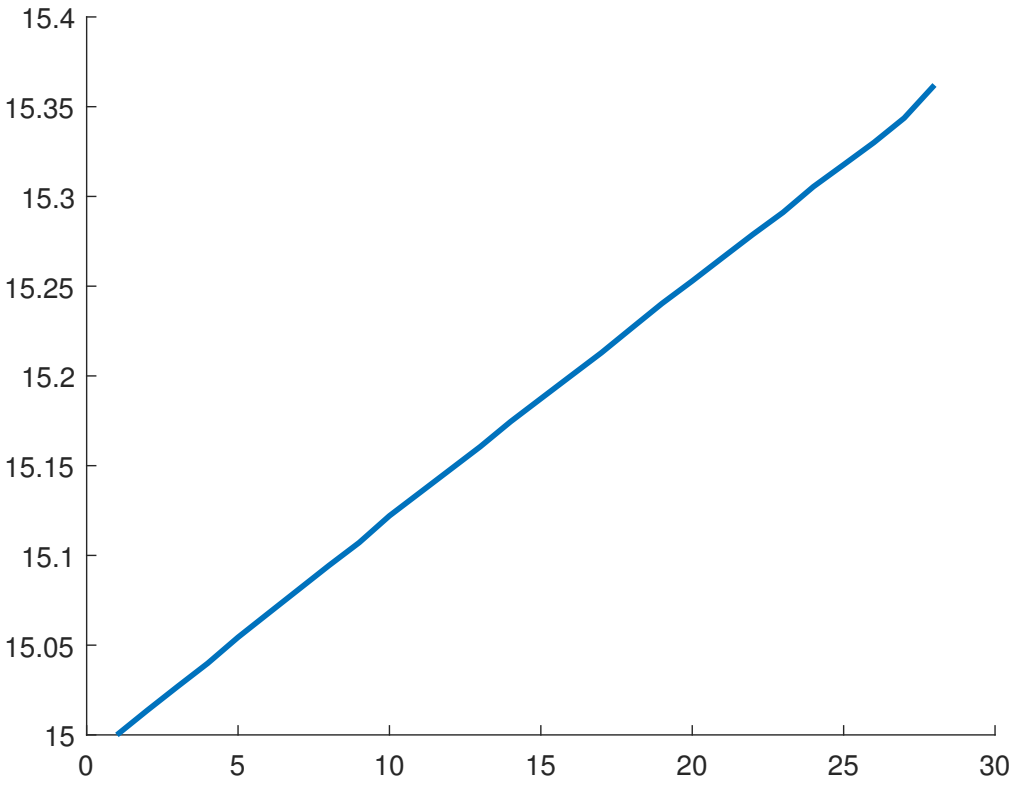


Figure 14: Cooling fluid temperature at 100th time step

We would also like to display the solution on the whole model in the last time step (when it has already reached a steady state).

```
plotSolution(battery_module, m, 100)
```

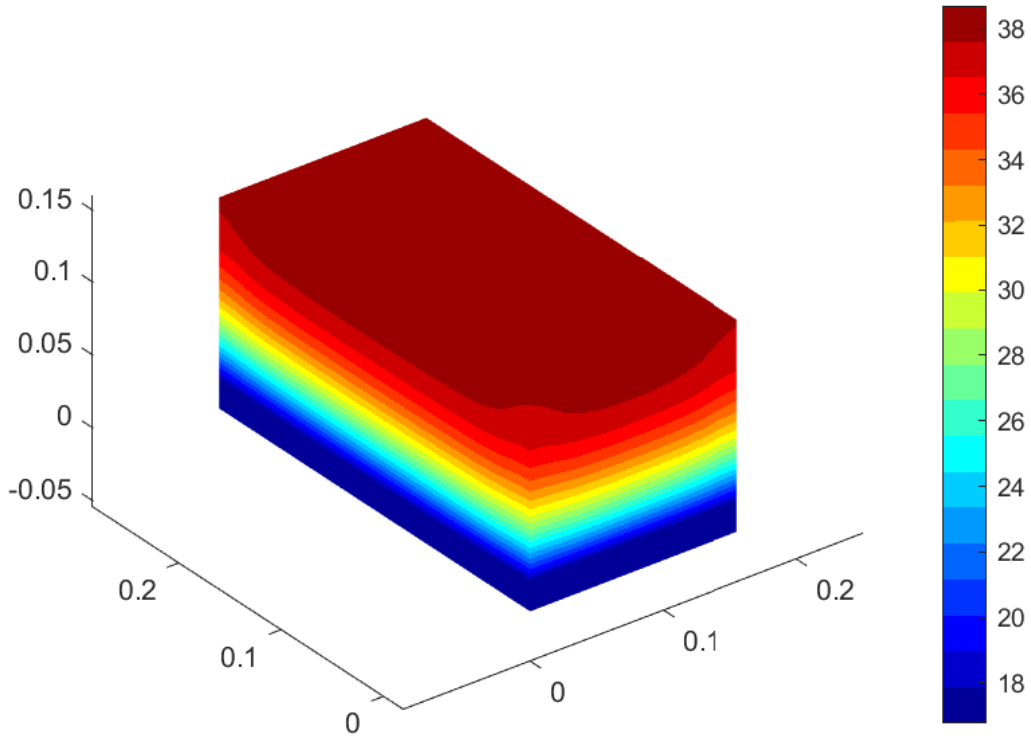


Figure 15: Solution at 100th time step

Lastly, we will export our solution to extensions which are compatible with other software, such as Paraview or Excel.

```
exportSolutionToVTK(battery_module, m, ...
    'tutorials/tutorial1/tutorial1_vtk', 101);
exportSolutionToCSV(battery_module, m, ...
    'tutorials/tutorial1/tutorial1_solution.xls');
```