QSA Q-Bat Explorer — Code Documentation

QuickerSim Automotive Ltd

Contents

1. User Interface - creating the model

The functions presented in this section describe all of the tools needed to create the desired model. From creating component prototypes, through placing and copying them inside the simulation domain, creating aggregating components to hydraulically connecting cooling plates. The functions are arranged in alphabetical order.

Caution! *When generating a msh file and loading it into QSA Q-Bat Explorer it is critical to assign unique physical IDs to all of the surfaces in the mesh generation software.*

1.1 connectCoolingPlatesPipes.m

```
function connectCoolingPlatesPipes(args)
```

This function creates hydraulic connections between cooling plates. Cooling plates are always connected from the outlet of the first specified cooling plate to the inlet of the second cooling plate etc.

Input arguments:

- Required:
  - *'cooling_plates'* - vector of cooling plates, arranged in order from the first to last cooling plates we want to couple.

Example:

```
connectCoolingPlatesPipes('cooling_plates', [cooling_plate1, ...
    cooling_plate2, cooling_plate3]);
```

1.2 copyInstanceInLocation.m

```
function instances = copyInstanceInLocation(args)
```

This function copies a component (not a prototype) in accordance with a specified location matrix.

Input arguments:

- Required:
  - *'instance'* - component (not prototype) which we want to replicate.
  - *'location_matrix'* - matrix (n, 3) double, specifying the location of n repetitions of the prototype. Each row of the matrix corresponds to the [x,y,z] coordinates of the repetition (in meters).

Output:

- *'instances'* - array of specified components placed in accordance with the *'location_matrix'*.

Example:

```
battery_islands = copyInstanceInLocation('instance', battery_island, ...
    'location_matrix', [-0.4, 0, 0; -0.4, 0.5, 0]);
```

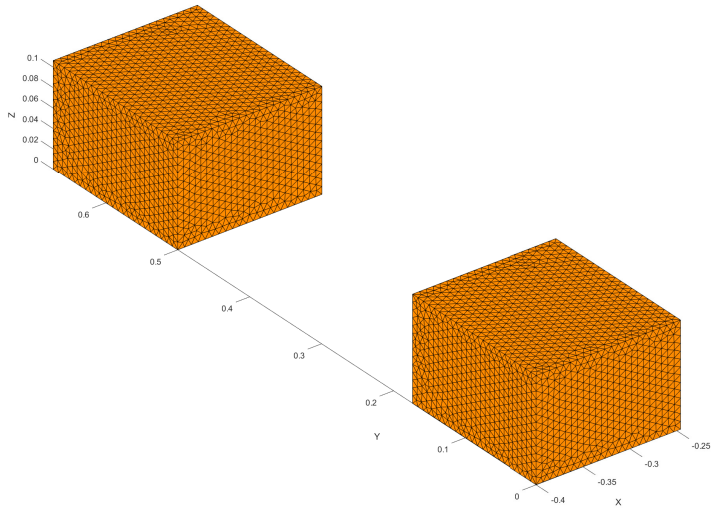

Figure 1: Copy instance in location

1.3 copyInstanceInPattern.m

```
function instances = copyInstanceInPattern(args)
```

This function copies a component (not a prototype) in accordance with a specified pattern.

Input arguments:

- Required:
  - *'instance'* - component (not prototype) which we want to replicate.
  - *'pattern_type'* - string specifying the pattern type (*'grid'* or *'triangle'*).
  - *'pattern_dimensions'* - vector (1, 5) or (1, 6) double specifying the dimensions of the pattern (in meters):
    - for *'grid'*
      [x, y, nx, ny, dx, dy] - creates grid pattern with left lower corner located at point [x, y]. Distance between repetitions along X and Y axis are defined by vector [dx, dy]. Number of repetitions along X and Y axis are defined by [nx, ny].
    - for *'triangle'*
      [x, y, nx, ny, dx] - creates an equilateral triangle pattern with left lower corner located at point [x, y]. The distance between repetitions is defined by dx. Number of repetitions along X and Y axis are defined by [nx, ny] (nx >= ny).

Output:

- *'instances'* - array of specified components placed in accordance with the specified pattern dimensions.

Examples:

```
battery_modules = copyInstanceInPattern('instance', battery_module, ...
    'pattern_type', 'grid', 'pattern_dimensions', [0, 0, 4 , 3, 0.2, 0.5]);

battery_modules = copyInstanceInPattern('instance', battery_module, ...
    'pattern_type', 'triangle', 'pattern_dimensions', [0, 0, 3 , 3, 0.5]);
```

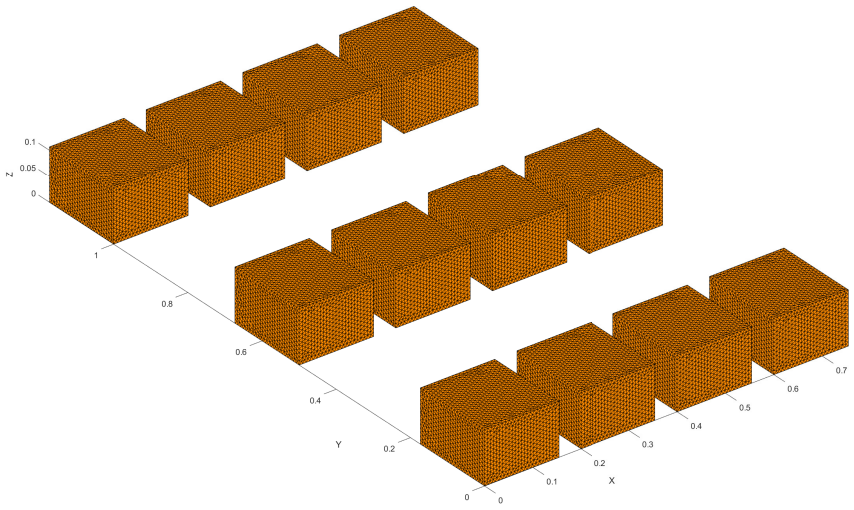

Figure 2: Copy instance in pattern

1.4 defineBattery.m

```
function battery = defineBattery(args)
```

This function creates a battery, which can (but does not have to) consist of one or more battery islands and one or more heat components.

Input arguments:

- Optional:
  - *'battery_islands'* - vector of battery islands to be added to the battery.
  - *'heat_components'* - vector of heat components to be added to the battery.
  - *'all_contacts'* - Boolean variable (*true* or *false*), creates thermal contacts basing on all contacts in the battery.

- Other / dependent:
  - *'conductivity'* - scalar double, defines thermal conductivity of all contacts (in $\frac{W}{m*K}$).
  - *'contacts_thickness'* - scalar double, defines thickness of all contacts (in meters).

Output:

- *'battery'* - battery object with specified battery islands, heat components and contacts between them.

Example:

```
battery1 = defineBattery('battery_islands', [battery_islands1, ...
    battery_islands2], 'all_contacts', true, 'conductivity', 1, ...
    'contacts_thickness', 1e-3);
```

1.5 defineBatteryIsland.m

```
function battery_island = defineBatteryIsland(args)
```

This function creates a battery island, which can (but does not have to) consist of one or more battery modules, one or more heat components and one or more cooling plates.

Input arguments:

- Optional:
  - *'battery_modules'* - vector of battery modules to be added to the battery island.
  - *'heat_components'* - vector of heat components to be added to the battery island.
  - *'cooling_plates'* - vector of cooling plates to be added to the battery island.
  - *'all_contacts'* - Boolean value (*true* or *false*), creates thermal contacts basing on all contacts in the battery island.
- Other / dependent:
  - *'conductivity'* - scalar double, defines thermal conductivity of all contacts (in $\frac{W}{mK}$).
  - *'contacts_thickness'* - scalar double, defines thickness of all contacts (in meters).

Output:

- *'battery_island'* - battery island object with specified battery modules, heat components, cooling plates and contacts between them.

Examples:

```
battery_island1 = defineBatteryIsland('battery_modules', bm1, ...
    'heat_components', heat_component_array1);

battery_island2 = defineBatteryIsland('battery_modules', [bm1, bm2], ...
    'all_contacts', 'cooling_plates', cp1, true, 'conductivity', 1, ...
    'contacts_thickness', 1e-3);
```

1.6 defineBatteryModule.m

```
function battery_module = defineBatteryModule(args)
```

This function creates a battery module, which can (but does not have to) consist of one or more cell caskets, one or more heat components and one or more cooling plates.

Input arguments:

- Optional:
  - *'cell_caskets'* - vector of cell caskets to be added to the battery module.
  - *'heat_components'* - vector of heat components to be added to the battery module.
  - *'cooling_plates'* - vector of cooling plates to be added to the battery module.
  - *'all_contacts'* - Boolean variable (*true* or *false*), creates thermal contacts basing on all contacts in the battery module.
- Other / dependent:
  - *'conductivity'* -scalar double, defines thermal conductivity of all contacts (in $\frac{W}{m*K}$).
  - *'contacts_thickness'* - scalar double, defines thickness of all contacts (in meters).

Output:

- *'battery_module'* - battery module object with specified cell caskets, heat components, cooling plates and contacts between them.

Examples:

```
bm1 = defineBatteryModule('cell_caskets', cell_caskets1, ...
    'heat_components', heat_component_array1);

bm2 = defineBatteryModule('cell_caskets', ...
    [cell_caskets1, cell_caskets1], 'cooling_plates', cooing_plate11, ...
    'all_contacts', true, 'conductivity', 1, 'contacts_thickness', 1e-3);
```

1.7 defineCellCasket.m

```
function cell_casket = defineCellCasket(args)
```

This function creates a cell casket, which can (but does not have to) consist of one or more cells and one or more heat components.

Input arguments:

- Optional:
  - *'cells'* - vector of cells to be added to the cell casket.
  - *'heat_components'* - vector of heat components to be added to the cell casket.
  - *'all_contacts'* - Boolean variable (*true* or *false*), creates thermal contacts basing on all contacts in the cell casket.
- Other / dependent:
  - *'conductivity'* - scalar double, defines thermal conductivity of all contacts (in $\frac{W}{m*K}$).
  - *contacts_thickness'* - scalar double, defines thickness of all contacts (in meters).

Output:

- *'cell_casket'* - cell casket object with specified cells, heat components and contacts between them.

Examples:

```
cell_casket1 = defineCellCasket('cells', cell_array1, ...
    'heat_components', heat_component_array1);

cell_casket2 = defineCellCasket('cells', [cell_array1, cell_array2], ...
    'all_contacts', true, 'conductivity', 1, 'contacts_thickness', 1e-3);
```

1.8 defineCellPrototype.m

```
function cell_prototype = defineCellPrototype(args)
```

This function creates a cell prototype (virtual cell with a specified geometry, material properties and mesh).

Input arguments:

- Required:
  - *'data'* - string, path to .xlsx or .xls file containing the material properties of the cell.
- One of the following required:
  - *'load_component'* - string, path to folder with previously saved cell prototype.
  - *'type'* - string specifying the type of the cell (*'prismatic'* or *'cylindric'*).
  - *'qscad_shape'* - qs shape containing the geometry of the cell (created using qscad).

  - *'mesh_path'* - string, path to .msh file containing previously created component mesh (does not require additionally specifying the geometry).

- Optional:
  - *'heat_source'* - value of volumetric heat generated in the cell (in $\frac{W}{m^3}$), can be defined as a constant heat source value (scalar double) or as a time variable heat source (matrix (n, 2) double).
  - *'mesh_size'* - scalar double (0 - inf), density of mesh generated inside the software (larger numbers correspond to small mesh, worse quality, 0 - dense mesh, good quality).
  - *'curv_points'* - integer, number of points on curvature (per 2π radians) used while creating mesh inside the software (3-50).
  - *'number_of_modes'* - integer, number of modes generated on the cell when creating the solution vector (the higher the number of mode the more accurate the solution but the longer the computational time).

- Other / dependent:
  - *'mass'* - scalar double, mass of the cell (required as an argument if the density of the cell is not specified in the *'data'* file).
  - *'capacity'* - scalar double, cell capacity (in A*s), required as an argument if the capacity of the cell is not specified in the *'data'* file.
  - *'dimensions'* - vector (1, 6) or (1, 5) double, dimensions of the cell, required if defining geometry by specifying the cell *'type'*:
    - for *'prismatic'*
      [x, y, z, dx, dy, dz] - creates cuboid with left lower corner located at point [x, y, z]. Dimensions along X, Y and Z axis are defined by vector [dx, dy, dz].
    - for *'cylindric'*
      [x, y, z, r, h] - creates cylinder with center point located at point [x, y, z] and base on the XY plane. Parameter r - specifies the base radius and h - the height of the cylinder (along Z axis).

Output:

- *'cell_prototype'* - cell prototype (virtual object) with a specified geometry, material properties and mesh.

Examples:

```
cell_prototype1 = defineCellPrototype('type','prismatic','dimensions', ...
    [0, 0, 0, 0.4, 0.2, 0.15], 'mesh_size', 0.6, 'curv_points', 12, ...
    'data', 'data/cell_data.xlsx');

cell_prototype2 = defineCellPrototype('mesh_path', ...
    'meshes/cell_mesh.msh', 'data', 'data/cell_data.xlsx', ...
    'heat_source', 10000);
```

1.9 defineCoolingPlatePrototype.m

```
function cooling_plate_prototype = defineCoolingPlatePrototype(args)
```

This function creates a cooling plate prototype (virtual cooling plate with a specified geometry, material properties and mesh).

Input arguments:

- Required:
  - *'pipe_radius'* - scalar double, specifies the cooling pipe radius (in meters).
  - *'pipe_mass_flow_rate'* - scalar or vector (n, 2) double, specifies the cooling fluid mass flow rate in time (in $\frac{kg}{s}$). If a scalar value is entered, the mass flow rate is defined as constant in time. Otherwise the first column of the entered vector corresponds to time and the second - to the value of mass flow rate. Between the specified points the values are interpolated linearly.
  - *'pipe_inlet_temperature'* - scalar or vector (n, 2) double, specifies the cooling fluid temperature at the inlet (in $°C$). If a scalar value is entered, the temperature is defined as constant in time. Otherwise the first column of the entered vector corresponds to time and the second - to the value of fluid inlet temperature. Between the specified points the values are interpolated linearly.
  - *'cooling_plate_data'* - string, path to .xlsx or .xls file containing the material properties of the cooling plate.
  - *'coolant_data'* - string, path to .xlsx or .xls file containing the material properties of the cooling fluid.
- One of the following required:
  - *'load_component'* - string, path to folder with saved previously created cooling plate prototype.
  - *'qscad_shape'* - qs shape containing the geometry of the cooling plate (created using qscad).
  - *'mesh_path'* - string, path to .msh file containing previously created component mesh (does not require additionally specifying the geometry).
- Optional:
  - *'pipe_discretization'* - integer, number of control points created in the cooling fluid pipe.
  - *'mesh_size'* - scalar double (0 - inf), density of mesh generated inside the software (larger numbers correspond to small mesh, worse quality, 0 - dense mesh, good quality).
  - *'curv_points'* - integer, number of points on curvature (per 2π radians) used while creating mesh inside the software (3-50).
  - *'number_of_modes'* - integer, number of modes generated on the cell when creating the solution vector (the higher the number of modes the more accurate the solution but the longer the computational time).

  - *'predefined_alpha'* - scalar double, value of heat transfer coefficient (in $\frac{W}{m^2*K}$) in forced convection (through cooling fluid walls). If it is not known it will be calculated internally.
  - *'pipe_wall_ids'* - integer vector of wall ids corresponding to the pipe walls (if not given as an argument, walls will have to be chosen interactively).
  - *'pipe_inlet_ids'* - integer vector of curve ids corresponding to the pipe inlet (if not given as an argument, curves will have to be chosen interactively).
  - *'pipe_outlet_ids'* - integer vector of curve ids corresponding to the pipe outlet (if not given as an argument, curves will have to be chosen interactively).
- Other / dependent:
  - *'mass'* - scalar double, mass of the cooling plate in kg (required as an argument if the density is not specified in the *'cooling_plate_data'* file).

Output:

- *'cooling_plate'* - cooling plate prototype (virtual object) with a specified geometry, material properties and mesh.

Examples:

```
cp1 = defineCoolingPlatePrototype('mesh_path', ...
    'meshes/cooling_plate1.msh', 'cooling_plate_data', ...
    'data/aluminium.xlsx', 'coolant_data','data/water.xlsx', ...
    'pipe_radius', 0.03, 'pipe_mass_flow_rate', 0.05, ...
    'pipe_inlet_temperature', 15, 'predefined_alpha', 2500);

cp2 = defineCoolingPlatePrototype('mesh_path', ...
    'meshes/cooling_plate2.msh', 'cooling_plate_data', ...
    'data/aluminium.xlsx','coolant_data','data/water.xlsx', ...
    'pipe_wall_ids', [1,3,4,7], 'pipe_inlet_ids', [3,17], ...
    'pipe_outlet_ids', [13,4], 'pipe_radius', 0.02, ...
    'pipe_mass_flow_rate', 0.05, 'pipe_inlet_temperature', 15);
```

1.10 defineHeatComponentPrototype.m

```
function heat_component_prototype = defineHeatComponentPrototype(args)
```

This function creates a heat component prototype (virtual heat component with a specified geometry, material properties and mesh).

Input arguments:

- Required:
  - *'heat_component_data'* - string, path to .xlsx or .xls file containing the material properties of the heat component.
- One of the following required:
  - *'load_component'* - string, path to folder with saved previously created heat component prototype.

- *'shape'* - string specifying the shape of the heat component (*'box'* or *'cylinder'*).
- *'qscad_shape'* - qs shape containing the geometry of the heat component (created using qscad).
- *'mesh_path'* - string, path to .msh file containing previously created component mesh.

- Optional:
  - *'mesh_size'* - scalar (0 - inf), density of mesh generated inside the software (larger numbers correspond to small mesh, worse quality, 0 - dense mesh, good quality).
  - *'curv_points'* - integer, number of points on curvature (per 2π radians) used while creating mesh inside the software (3-50).
  - *'number_of_modes'* - integer, number of modes generated on the heat component when creating the solution vector (the higher the number of modes the more accurate the solution but the longer the computational time).

- Other / dependent:
  - *'mass'* - scalar double, mass of the heat component in kg (required as an argument if the density of the component is not specified in the *'heat_component_data'* file).
  - *'dimensions'* - vector (1, 6) or (1, 5) double, dimensions of heat component, required if using *'shape'* for creating the heat component geometry:
    - for *'box'*
      [x, y, z, dx, dy, dz] - creates cuboid with left lower corner located at point [x, y, z]. Dimensions along X, Y and Z axis are defined by vector [dx, dy, dz].
    - for *'cylinder'*
      [x, y, z, r, h] - creates cylinder with center point located at point [x, y, z] and base on the XY plane. Parameter r - specifies the base radius and h - the height (along Z axis).

Output:

- *heat_component'* - heat component prototype (virtual object) with a specified geometry, material properties and mesh.

Examples:

```
heat_component_prototype1 = defineHeatComponentPrototype('shape',...
    'cylinder', 'dimensions',[0,0,0,0.1,0.1], 'heat_component_data', ...
    '/data/plastic_data.xlsx');

heat_component_prototype2 = defineHeatComponentPrototype('mesh_path', ...
    '/meshes/hc2.msh', 'heat_component_data', ...
    '/data/plastic_data.xlsx', 'number_of_modes', 30);
```

1.11 instantiateInLocation.m

```
function instances = instantiateInLocation(args)
```

This function places the virtual prototype inside the simulation domain in accordance with a specified location matrix.

Input arguments:

- Required:
  - *'prototype'* - name of prototype which we want to replicate and place inside the simulation domain.
  - *'location_matrix'* - matrix (n, 3) double specifying the location of n repetitions of the prototype. Each row of the matrix corresponds to the [x, y, z] coordinates of the repetition (in meters).

Output:

- *'instances'* - array of specified components (non-virtual) placed in the simulation domain in accordance with the *'location_matrix'*.

Example:

```
cells = instantiateInLocation('prototype', cell_prototype, ...
    'location_matrix', [-0.42, 0.02, 0; -0.42, 0.235, 0]);
```

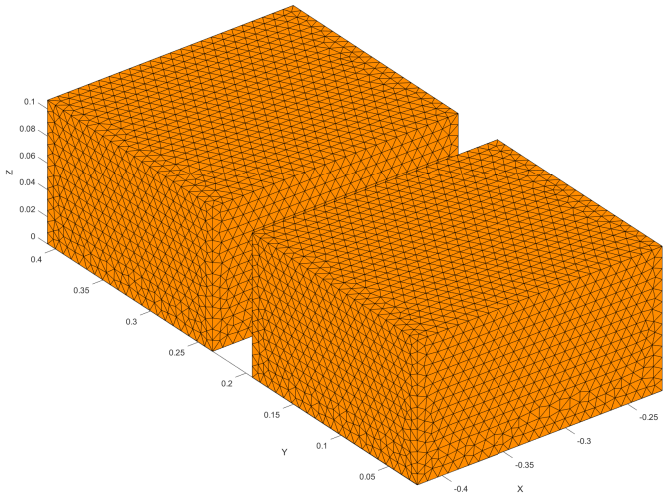

Figure 3: Instantiate in location

1.12 instantiteInPattern.m

```
function instances = instantiateInPattern(args)
```

This function places the virtual prototype inside the simulation domain in accordance with a specified pattern.

Input arguments:

- Required:
  - *'prototype'* - name of prototype which we want to replicate and place inside the simulation domain.
  - *'pattern_type'* - string specifying the pattern type (*'grid'* or *'triangle'*).
  - *'pattern_dimensions'* - vector (1, 6) or (1, 5) double specifying the dimensions of the pattern (in meters):
    - for *'grid'*
      [x, y, nx, ny, dx, dy] - creates grid pattern with left lower corner located at point [x, y]. Distance between repetitions along X and Y axis are defined by vector [dx, dy]. Number of repetitions along X and Y axis are defined by [nx, ny].
    - for *'triangle'*
      [x, y, nx, ny, dx] - creates an equilateral triangle pattern with left lower corner located at point [x, y]. The distance between repetitions is defined by dx. Number of repetitions along X and Y axis are defined by [nx, ny] (nx >= ny).

Output:

- *'instances'* - array of specified components (non-virtual) placed in accordance with the pattern dimensions.

Examples:

```
cells1 = instantiateInPattern('prototype', cell_prototype, ...
    'pattern_type', 'grid', 'pattern_dimensions', [0, 0, 4 , 3, 0.2, 0.5]);

cells2 = instantiateInPattern('prototype', cell_prototype, ...
    'pattern_type', 'triangle', 'pattern_dimensions', [0, 0, 3 , 2, 0.5]);
```

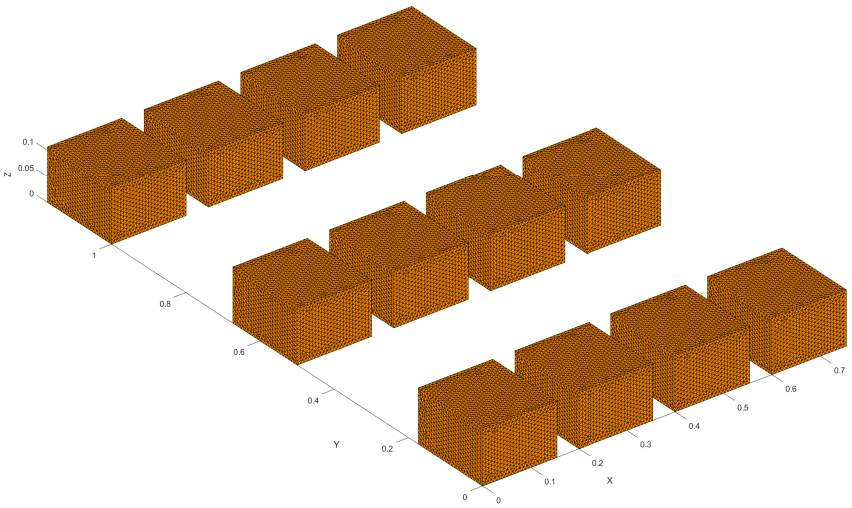

Figure 4: Instantiate in pattern

2. Translating and rotating objects

The functions presented in this section can be used on all non-virtual objects (placed in the simulation domain) and are use for transformations inside the domain.

2.1 ui.qsModuleKernel.internal.TransformationKernel.m

2.1.1 translate

```
function self = translate(self, translation_vector)
```

Function translates the given object by a specified vector.

Input arguments:

- *translation_vector* - vector (3, 1) Double, defining translation in each direction (in meters).

Example:

```
cell.translate([1, 0.2, 0]);
```

2.1.2 rotateAboutOrigin

```
function self = rotateAboutOrigin(self, rotation_angles)
```

Function rotates the given object by specified angles around each axis.

Input arguments:

- *rotation_angles* - vector (3, 1) double containing rotation angles in radians. The center of this rotation is the coordinate center. Order of rotation: ZYX.

Example:

```
cell.rotateAboutOrigin([0, pi/2, 0]);
```

3. Boundary conditions

Functions described in this section describe the methods for assigning boundary conditions to component prototypes. Boundary conditions cannot be assigned to non-virtual components.

3.1 ui.qsModuleKernel.BCs.m

3.1.1 addDirichletBC

```
function self = addDirichletBC(self, varargin)
```

Function adds Dirichlet boundary condition on component prototype.

Input arguments:

- Required:
  - *'ids'* - scalar vector of surface ids on which the boundary condition will be applied.
  - *'value'* - scalar double, value of boundary condition (temperature in $°C$).

Example:

```
cell_prototype.bcs.addDirichletBC('ids', [1,3,4], 'value', 25);
```

3.1.2 addNeumannBC

```
function addNeumannBC(self, varargin)
```

Function adds Neumann boundary condition on component prototype.

Input arguments:

- Required:
  - *'ids'* - integer vector of surface ids on which the boundary condition will be applied.
- One of the following required:
  - *'value'* - scalar double, value of boundary condition (heat flux in $\frac{W}{m^2}$) if it is constant.
  - *'time_variable_value'* - vector (n, 2) double specifying the value of boundary condition in time. The first column corresponds to time and the second - to the value of heat flux (in $\frac{W}{m^2}$). Between the specified points the values are interpolated linearly.

Example:

```
cell_prototype.bcs.addNeumannBC('ids', [1,3,4], 'time_variable_value', ...
    heat_flux_table);
```

3.1.3 addSource

```
function addSource(self, varargin)
```

Function adds heat source on component prototype.

Input parameters:

- One of the following required:
  - *'heat_source'* - scalar double, value of volumetric heat source (in $\frac{W}{m^3}$) if it is constant.
  - *'time_variable_heat_source'* - vector (n, 2) double, with the first column corresponding to time and the second column - to the value of heat source (in $\frac{W}{m^3}$) at given time points. Between the specified time points the values are interpolated linearly.
  - *'time_variable_current'* - vector (n, 2) double, specifying the value of electric current in time. The first column corresponds to time and the second - to the value of current (in A). Between the specified points the values are interpolated linearly.

Example:

```
cell_prototype.bcs.addSource('time_variable_current', current_profile);
```

3.1.4 addRobinBC

```
function addRobinBC(self, varargin)
```

Function adds Robin boundary condition on component prototype.

Input arguments:

- Required:
  - *'ids'* - integer vector of surface ids on which the boundary condition will be applied.
  - *'alpha'* - scalar double, value of heat transfer coefficient (in $\frac{W}{m^2*K}$) if it is constant.
- One of the following required:
  - *'t_inf'* - scalar double, specifying the fixed temperature (in $°C$) on the surfaces.
  - *'time_variable_temperature'* -vector (n, 2) double, specifying the value of ambient temperature in time. The first column corresponds to time and the second - to the value of temperature (in $°C$). Between the specified points the values are interpolated linearly.

Example:

```
cell_prototype.bcs.addRobinBC('ids', [1,3,4], 't_inf', 20, 'alpha', 35);
```

3.1.5 addRobinBCWithConductiveSheet

```
function addRobinBCWithConductiveSheet(self,  varargin)
```

Function adds Robin boundary condition on component prototype (with a conductive sheet in between). At the moment this function does not support time variable parameters.

Input arguments:

- Required:
  - *'ids'* - integer vector of surface ids on which the boundary condition will be applied.
  - *'lambda'* - scalar double, specifying the thermal conductivity of the conductive sheet (in $\frac{W}{m*K}$).
  - *'thickness'* - scalar double, specifying the thickness of the conductive sheet (in meters).
  - *'alpha'* - scalar double, value of heat transfer coefficient (in $\frac{W}{m^2*K}$) if it is constant.
  - *'t_inf'* - scalar double, specifying the fixed temperature (in $°C$) on the surfaces.

Example:

```
cell_prototype.bcs.addRobinBCWithConductiveSheet('ids', [1,3,4], ...
    't_inf', 20, 'alpha', 35, 'lambda', 600, 'thickness', 0.0015);
```

4. Contacts

The functions presented in this section can be used to add contacts between different bodies of the assembly. Contacts can be defined in various different ways. By default no contacts are created while defining the model.

4.1 defineCellCasket.m / defineBatteryModule.m / defineBatteryIsland.m/ defineBattery.m

By defining contacts inside one of the functions presented above, the User can add contacts of the same value (thickness and conductivity) to all adhering bodies in the assembly (where applicable).

Input arguments:

- *'all_contacts'* - Boolean variable (*true* or *false*), defines all contacts in the cell casket.
- *'conductivity'* - scalar double, defines thermal conductivity of all contacts (in $\frac{W}{m*K}$).
- *'contacts_thickness'* - scalar double, defines thickness of all contacts (in meters).

Example:

```
battery_module = defineBatteryModule('cell_caskets', cell_casket1, ...
    'heat_components', [hc1, hc2], 'cooling_plates', cp1, ...
    'all_contacts', true, 'conductivity', 1, 'thickness', 1e-3);
```

4.2 Defining contacts between component of given types

Caution! *Using these functions does not overwrite existing contacts, but creates new ones.*

4.2.1 setCouplingBetweenCellsAndHeatComponents

```
function setCouplingBetweenCellsAndHeatComponents(self, lambda, ...
    thickness, varargin)
```

This function adds contacts between cells and heat components within a cell casket.

Input arguments:

- *'lambda'* - scalar double, defines thermal conductivity of contacts (in $\frac{W}{m*K}$).
- *'thickness'* - scalar double, defines thickness of contacts (in meters).

Example:

```
cell_casket1.setCouplingBetweenCellsAndHeatComponents(3,2e-3);
```

4.2.2 setCouplingBetweenCellCasketsAndHeatComponents

```
function setCouplingBetweenCellCasketsAndHeatComponents(self, lambda, ...
    thickness, varargin)
```

This function adds contacts between cell caskets and heat components within a battery module.

Input arguments:

- *'lambda'* - scalar double, defines thermal conductivity of contacts (in $\frac{W}{m*K}$).
- *'thickness'* - scalar double, defines thickness of contacts (in meters).

Example:

```
my_battery_module.setCouplingBetweenCellCasketsAndHeatComponents(3, 1e-3);
```

4.2.3 setCouplingBetweenCellCasketsAndCoolingPlate

```
function setCouplingBetweenCellCasketsAndCoolingPlate(self, lambda, ...
    thickness, varargin)
```

This function adds contacts between cell caskets and cooling plates within a battery module.

Input arguments:

- *'lambda'* - scalar double, defines thermal conductivity of contacts (in $\frac{W}{m*K}$).
- *'thickness'* - scalar double, defines thickness of contacts (in meters).

Example:

```
my_battery_module.setCouplingBetweenCellCasketsAndCoolingPlate(3, 2e-3);
```

4.2.4 setCouplingBetweenCoolingPlatesAndHeatComponents

```
function setCouplingBetweenCoolingPlatesAndHeatComponents(self, ...
    lambda, thickness, varargin)
```

This function adds contacts between heat components and cooling plates within a specified aggregating object (battery module, battery island or battery).

Input arguments:

- *'lambda'* - scalar double, defines thermal conductivity of all contacts (in $\frac{W}{m*K}$).
- *'thickness'* - scalar double, defines thickness of all contacts (in meters).

Example:

```
my_battery.setCouplingBetweenCoolingPlatesAndHeatComponents(3, 1e-3);
```

4.2.5 setCouplingBetweenBatteryModulesAndHeatComponents

```
function setCouplingBetweenBatteryModulesAndHeatComponents(self, ...
    lambda, thickness, varargin)
```

This function adds contacts between battery modules and heat component within a battery island.

Input arguments:

- *'lambda'* - scalar double, defines thermal conductivity of contacts (in $\frac{W}{m*K}$).
- *'thickness'* - scalar double, defines thickness of contacts (in meters).

Example:

```
battery_island1.setCouplingBetweenBatteryModulesAndHeatComponents(5, 1);
```

4.2.6 setCouplingBetweenBatteryModulesAndCoolingPlate

```
function setCouplingBetweenBatteryModulesAndCoolingPlate(self, ...
    lambda, thickness, varargin)
```

This function adds contacts between battery modules and cooling plates within a battery island.

Input arguments:

- *'lambda'* - scalar double, defines thermal conductivity of contacts (in $\frac{W}{m*K}$).
- *'thickness'* - scalar double, defines thickness of contacts (in meters).

Example:

```
battery_island1.setCouplingBetweenBatteryModulesAndCoolingPlate(3, 1e-3);
```

4.2.7 setCouplingBetweenBatteryIslandsAndHeatComponents

```
function setCouplingBetweenBatteryIslandsAndHeatComponents(self, ...
    lambda, thickness, varargin)
```

This function adds contacts between battery islands and heat components within a battery.

Input arguments:

- *'lambda'* - scalar double, defines thermal conductivity of contacts (in $\frac{W}{m*K}$).
- *'thickness'* - scalar double, defines thickness of contacts (in meters).

Example:

```
my_battery.setCouplingBetweenBatteryIslandsAndHeatComponents(3, 1e-3);
```

4.3 Custom couplings

Custom contacts can be set using one of the following functions.

Caution! *Using these functions does not overwrite existing contacts, but creates new ones.*

4.3.1 setCustomCoupling

```
function setCustomCoupling(self, lambda, thickness, obj1, obj2)
```

This function adds contacts between specific bodies (both must be a part of the given aggregating component).

Input arguments:

- *'lambda'* - scalar double, defines thermal conductivity of contact (in $\frac{W}{m*K}$).
- *'thickness'* - scalar double, defines thickness of contact (in meters).
- *'obj1'* - first object of the contact.
- *'obj2'* - second object of the contact.

Example:

```
battery_mod.setCustomCoupling(10, 2e-3, cell_casket1, ...
    [heat_component2, heat_component4]);
```

4.3.2 setAllCouplings

```
function setAllCouplings(self, lambda, thickness)
```

This function adds contacts between all of the adhering bodies contained in an aggregating body. This function is equivalent to setting all the contacts while defining an aggregating object (Section 4.1).

Input arguments:

- *'lambda'* - scalar double, defines thermal conductivity of all contacts (in $\frac{W}{m*K}$).
- *'thickness'* - scalar double, defines thickness of all contacts (in meters).

Example:

```
my_battery_module.setAllCouplings(10, 3e-3);
```

5. Assembly and simulation

Functions presented in this section are used to reduce the prepared model and perform calculations.

5.1 assembleModel.m

```
function engine_model = assembleModel(ui_model)
```

Function assembles full model and reduces it.

Input argument:

- *'ui_model'* - model created using the User Interface functions.

Output:

- *'engine_model'* - reduced model on which the calculation will be performed.

Example:

```
m = assembleModel(battery);
```

5.2 runReducedMerged.m

```
function res = runReducedMerged(this, dt, nTSteps, electroSubsteps, ...
    varargin)
```

Function performs calculations on the reduced model.

Input parameters:

- Required:
  - *'dt'* - integer, length of time step for simulation (in seconds).
  - *'nTSteps'* - integer, number of times steps.
- Dependent:
  - *'electroSubsteps'* - integer, number of substeps for calculating the electric current value and electrical cell properties (within the solution calculation time step). This parameter is not required if heat source is specified as a volumetric heat source value (and not electric current profile).
- Optional:
  - *'startStep'* - integer, index of time step at which the calculations are started.
  - *'endStep'* - integer, index of time step at which the calculations are ended.
  - *'maxSubIter'* - integer, maximum number of subiterations within one time step.
  - *'maxRes'* - scalar double, maximum residuum within one time step.

Examples:

```
% for steady state simulation
m.runReducedMerged(10000, 100, 'maxSubIter', 20)

% for unsteady simulation
m.runReducedMerged (6, 100, 2, 'maxRes', 1e-6)
```

6. Post-processing

Functions described in this section are used for post processing of the obtained solution.

6.1 exportSolutionToCSV.m

```
function exportSolutionToCSV(ui_model, engine_model, file_name)
```

This function exports the solution to a csv file.

Input arguments:

- Required:
  - *ui_model* - model created using the User Interface functions.
  - *engine_model* - reduced model on which the calculation were performed.
  - *file_name* - string, name of exported file.

Example:

```
exportSolutionToCSV(battery, m, '\solutions\battery_calculations.csv');
```

6.2 exportSolutionToVTK.m

```
function exportSolutionToVTK(ui_model, engine_model, dir_name, time_step)
```

This function exports solution to a vtk file.

Input arguments:

- Required:
  - *ui_model* - model created using the User Interface functions.
  - *engine_model* - reduced model on which the calculation were performed.
  - *dir_name* - string, name of exported file directory.
- Optional:
  - *time_step* - integer vector, time steps from which you wish to export the solution. If *time_step* is not specified, the solution for all performed time step will be exported.

Example:

```
exportSolutionToVTK(battery, m, '\solutions\unsteady_calculations', 1:20)
```

6.3 plotComponentsMaxTempOverTime.m

```
function plotComponentsMaxTempOverTime(ui_model, engine_model, ...
    ui_component_type, ax)
```

This function plots the maximum temperature in a specific type of component over time.

Input arguments:

- Required:
  - *ui_model* - model created using the User Interface functions.
  - *engine_model* - reduced model on which the calculation were performed.
  - *ui_component_type* - type of components for which we want to plot the solution.
- Optional:
  - *ax* - MATLAB axes object.

Example:

```
plotComponentsMaxTempOverTime(battery, m, 'cell', gca)
```

6.4 plotComponentsMeanTempOverTime.m

```
function plotComponentsMeanTempOverTime(ui_model, engine_model, ...
    ui_component_type, ax)
```

This function plots the mean temperature in a specific type of component over time.

Input arguments:

- Required:
  - *ui_model* - model created using the User Interface functions.
  - *engine_model* - reduced model on which the calculation were performed.
  - *ui_component_type* - type of components for which we want to plot the solution.
- Optional:
  - ax - MATLAB axes object.

Example:

```
plotComponentsMeanTempOverTime(battery, m, 'cell', gca)
```

6.5 plotComponentsMinTempOverTime.m

```
function plotComponentsMinTempOverTime(ui_model, engine_model, ...
    ui_component_type, ax)
```

This function plots the minimum temperature in a specific type of component over time.

Input arguments:

- Required:
  - *ui_model* - model created using the User Interface functions.
  - *engine_model* - reduced model on which the calculation were performed.
  - *ui_component_type* - type of components for which we want to plot the solution.
- Optional:
  - *ax* - MATLAB axes object.

Example:

```
plotComponentsMinTempOverTime(battery, m, 'cell', gca)
```

6.6 plotCoolantOutletTempOverTime.m

```
function plotCoolantOutletTempOverTime(cooling_plate, engine_model, ax)
```

This function plots the outlet temperature of the cooling fluid in a specified cooling plate over time.

Input arguments:

- Required:
  - *cooling_plate* - cooling plate for which we want to plot the outlet temperature.
  - *engine_model* - reduced model on which the calculation were performed.
- Optional:
  - *ax* - MATLAB axes object.

Example:

```
plotCoolantOutletTempOverTime(cp1, m, gca)
```

6.7 plotCoolantTempProfile.m

```
function plotCoolantTempProfile(cooling_plate, engine_model, time_step, ax)
```

This function plots the temperature of the cooling fluid along the cooling pipe of a specified cooling plate in a given time step.

Input arguments:

- Required:
  - *cooling_plate* - cooling plate for which we want to plot the temperature profile.
  - *engine_model* - reduced model on which the calculation were performed.
  - *time_step* - integer vector, time steps for which you wish to plot the solution.
- Optional:
  - *ax* - MATLAB axes object.

Example:

```
plotCoolantOutletTempOverTime(cp1, m, 60, gca)
```

6.8 plotCurrentOverTime.m

```
function plotCurrentOverTime(ui_model, engine_model, ax)
```

This function plots the electric current profile in the model over time.

Input arguments:

- Required:
  - *ui_model* - cells created using the User Interface functions, from which we want to plot the current profile.
  - *engine_model* - reduced model on which the calculation were performed.
- Optional:
  - *ax* - MATLAB axes object.

Example:

```
plotCurrentOverTime([cell_array1, cell_array2], m, gca)
```

6.9 plotMaxTempOverTime.m

```
function plotMaxTempOverTime(ui_model, engine_model, ax)
```

This function plots the maximum temperature in the model over time.

Input arguments:

- Required:
  - *ui_model* - model created using the User Interface functions.
  - *engine_model* - reduced model on which the calculation were performed.
- Optional:
  - *ax* - MATLAB axes object.

Example:

```
plotMaxTempOverTime(battery, m, gca)
```

6.10 plotMeanTempOverTime.m

```
function plotMeanTempOverTime(ui_model, engine_model, ax)
```

This function plots the mean temperature in the model over time.

Input arguments:

- Required:
  - *ui_model* - model created using the User Interface functions.
  - *engine_model* - reduced model on which the calculation were performed.
- Optional:
  - *ax* - MATLAB axes object.

Example:

```
plotMeanTempOverTime(battery, m, gca)
```

6.11 plotMinTempOverTime.m

```
function plotMinTempOverTime(ui_model, engine_model, ax)
```

This function plots the minimum temperature in the model over time.

Input arguments:

- Required:
  - *ui_model* - model created using the User Interface functions.
  - *engine_model* - reduced model on which the calculation were performed.
- Optional:
  - *ax* - MATLAB axes object.

Example:

```
plotMinTempOverTime(battery, m, gca)
```

6.12 plotSolution.m

```
function plotSolution(ui_model, engine_model, timestep, ax)
```

This function plots the solution in a given time step.

Input arguments:

- Required:
  - *ui_model* - model created using the User Interface functions.
  - *engine_model* - reduced model on which the calculation were performed.
  - *time_step* - integer vector, time steps for which you wish to plot the solution.
- Optional:
- *ax* - MATLAB axes object.

Example:

```
plotSolution(battery, m, 100, gca)
```

7. Saving and loading the components and model

Functions described in this section are used for saving and loading the model and its components.

7.1 Saving & loading prototypes

Component prototypes can be saved and loaded using functions included in *BCs.m*.

7.1.1 saveComponent

```
function saveComponent(self, path_to_write)
```

Function saves component prototype to folder in a specified path.

Input arguments:

- *'path_to_write'* - path in which a folder containing the saved component will be created.

Example:

```
cell_prototype.saveComponent('my_prototypes/cell')
```

7.1.2 loadComponent

```
function loadComponent(self, path_to_read)
```

Function loads previously saved component prototype.

Input arguments:

- *'path_to_read'* - path in which the component prototype is saved.

Examples:

```
cell_prototype.loadComponent('my_prototypes/cell')

% function can also used while defining the component prototype
cell_prototype = defineCellPrototype('load_component', ...
    'my_prototypes/cell')
```

7.2 Saving & loading the model

The reduced model can be save and loaded using standard Matlab functions.

7.2.1 Saving the model

```
save(filename,variables,version)
```

Function saves workspace variables to file.

Input parameters:

- *'filename'* - string, path in which the reduced model will be saved.
- *'variables'* - variables which we want to save (the reduced model).
- *'version'* - specifies the MAT-file version. For correctly saving the model the *v7.3* version should be used.

Example:

```
save('models/reduced_battery_model1','m','-v7.3');
```

7.2.2 Loading the model

Loading a model can be carried out by double clicking on the saved model in a opened Matlab environment or by using the *load* function.

```
load(filename)
```

Function loads variables from file into workspace.

Input parameters:

- *'filename'* - string, path in which the reduced model is saved.

Example:

```
load('models/reduced_battery_model1');
```

8. Known issues & limitations

8.1 Direct contact

A direct contact between two bodies is simulated by a coupling with a very high conductance (in $\frac{W}{m^2*K}$). This approach might create artefacts and therefore problems with the solution. If this occurs, it is advised to decrease the value of conductance of the direct contacts, keeping them at least 3 orders of magnitude higher that the maximum value of conductance of other contacts.

8.2 Pipe discretization error

When discretizing the cooling plate pipe an error might appear, related to an excessive amount of discretization points. It is advised to decrease the number of dicretization points on the pipe and call the function once more.