gmshModel Documentation

Release 1.0.3

Philipp Metsch

Nov 30, 2020

Getting Started

Gmsh is a powerful tool for the generation of meshes for numerical simulations but the built-in scripting language makes the meshing procedure and especially an automatization really hard. Luckily, Gmsh provides a Python-API with which all the capabilites of Gmsh can be used within Python.

GmshModel is intended to be an extendable tool that facilitates the mesh generation by interfacing the Gmsh-Python-API: it provides a basic framework for an automated mesh generation for self-defined model types and, with that, allows to automate the generation of complex models as, e.g., representative volume elements. To this end, GmshModel divides the mesh modeling procedure into basic steps:

1. Setting up a geometry using basic geometric entities and boolean operations.
2. Adding the geometric objects to Gmsh, performing the boolean operations and defining physical groups.
3. Creating a mesh with user-defined refinement fields.
4. Saving the mesh to various output formats.
5. Visualizing the resulting mesh.

So far, GmshModel is especially designed to automate the generation of representative volume elements that contain multiple inclusion objects. An extension of gmshModel is however possible by adding new geometric objects and model types to the framework.

It is not the purpose of GmshModel to replace the Gmsh scripting language or other great tools such as PyGmsh for the generation of meshes. GmshModel rather tries to function as an interface to Gmsh to facilitate the automation of recurring, complex meshing tasks that require the full functionality of Gmsh in a nice and easy to use programming environment such as Python.

CHAPTER 1

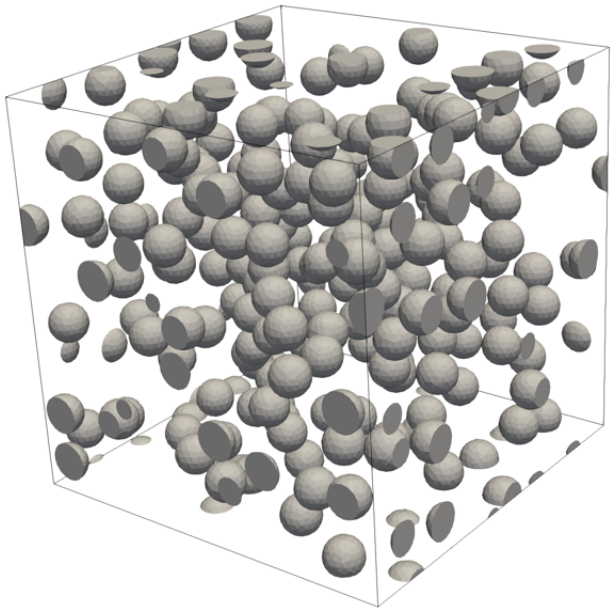
Getting Started

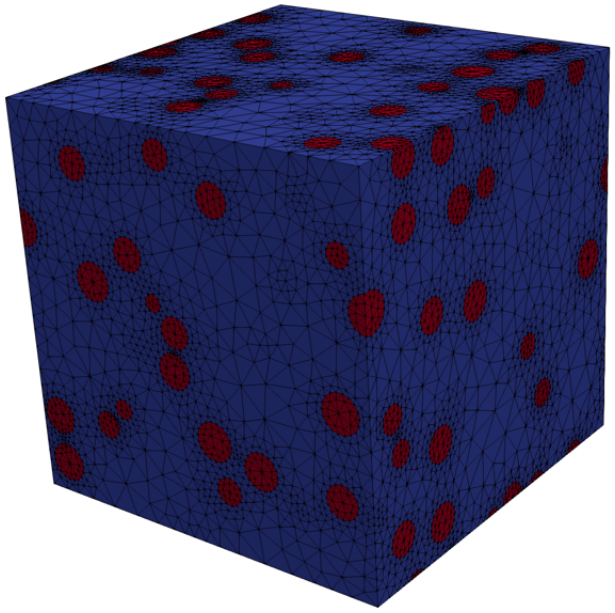
To get all information on how to install gmshModel, see *Installation*. If you are using `pip`, simply use the following command to install gmshModel and its features:

```
$ python3 -m pip install gmshModel
```

For `conda` users, the installation command is straightforward, too:

```
$ conda install -c conda-forge gmshModel
```





To check out what you can do with gmshModel and generate the above periodic mesh with `200` randomly placed spherical inclusions of radius `1` in a `[20x20x20]` domain, simply use the following code:

```
# import required model type
from gmshModel.Model import RandomInclusionRVE as RVE

# initialize new RVE
myRVE=RVE(size=[20,20,20], inclusionType="Sphere", inclusionSets=[1, 200])

# create Gmsh model
myRVE.createGmshModel()

# generate mesh
myRVE.createMesh()

# save resulting mesh to vtk
myRVE.saveMesh("myRVE.vtk")

# visualize result
myRVE.visualizeMesh()

# finalize Gmsh-Python-API
myRVE.close()
```

Go to *Examples* to check out more examples of meshes generated using gmshModel.

1.1 Installation

As most Python packages, gmshModel can be installed in more than one way. Here, the two common ways of the package installation will be pointed out: the installation via the Conda and PyPi package managers.

1.1.1 Dependencies

GmshModel is an interface tool and makes use of many great contributions of other people. To experience the full functionality of Gmsh model, the following (non-standard) software packages are required:

1. a dynamically built Gmsh to use the Gmsh-Python-API
2. meshio for the conversion of meshes to various output formats
3. pyvista for the visualization of meshes
4. pythonocc-core for the visualization of the model geometry

Using the supported PyPi and Conda package managers, all dependencies that are necessary to run gmshModel will be automatically installed. Since the `pythonocc-core` package does not provide an installation for PyPi, the geometry visualization feature will not be available for it.

1.1.2 Installation using Conda

The availability of gmshModel in the conda-forge channel allows a straightforward installation of the package using the following command:

```
$ conda install -c conda-forge gmshModel
```

1.1.3 Installation using PyPi

Since gmshModel is also available from the Python Package Index, PyPi users can simply install it using the following command:

```
$ python3 -m pip install gmshModel
```

If the package does not work after the installation due to an import error of the Gmsh-Python-API, your system probably cannot find the file `gmsh.py`. In order to fix this, a symbolic link from its installation location into the `site-packages` directory of your Python installation can be created:

```
# example for linux users:
$ ln -s <PATH TO GMSH>/lib/gmsh.py $HOME/.local/lib/<PYTHON VERSION>/site-packages/
↪gmsh.py
```

If you also want to have a working geometry visualization, you can compile pythonocc-core from source.

1.2 Using the visualization features

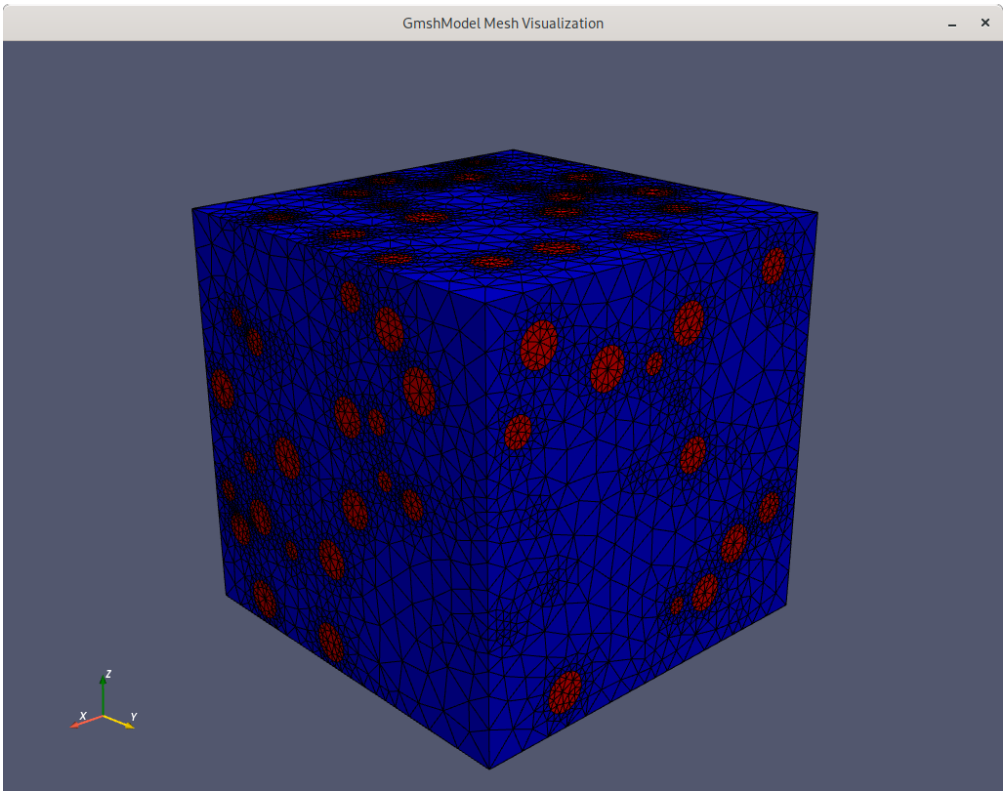
After the mesh generation, it is sometimes advantageous to have the possibility to visualize the resulting mesh in order to check if it matches the own requirements. In gmshModel, this can be accomplished by using the `visualizeMesh()` functionality of the `GenericModel`: since all available model types inherit the methods of `GenericModel`, the method is available for all models.

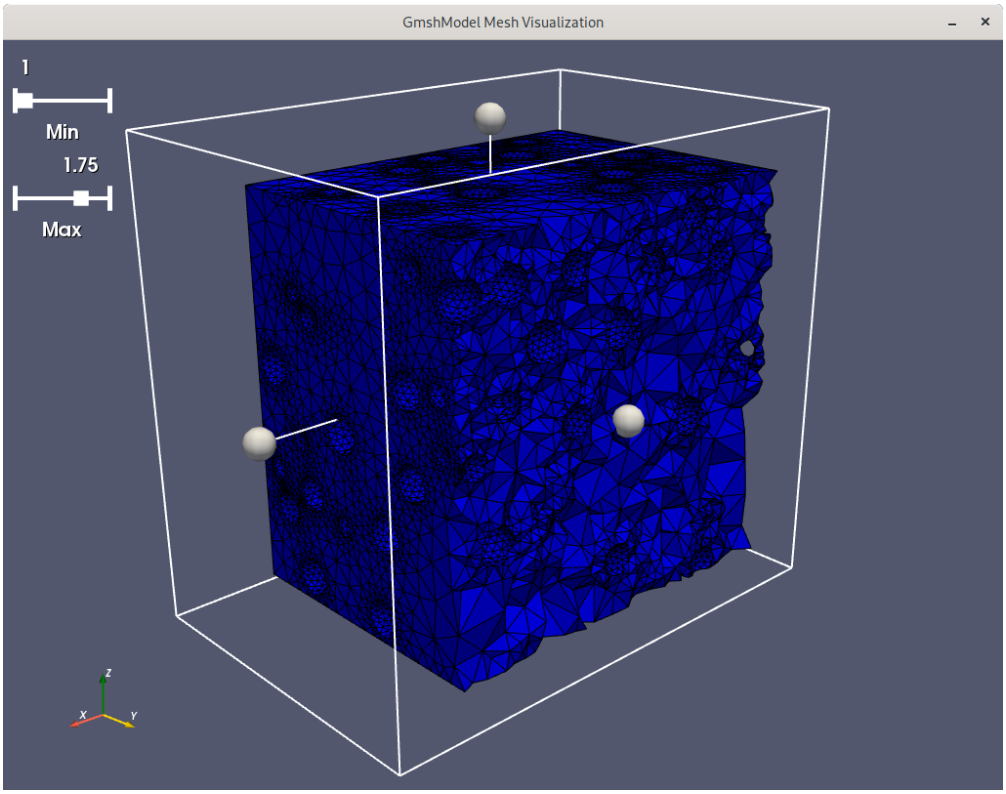
```
...

# visualize the mesh of myModel
myModel.visualizeMesh()

...
```

The mesh visualization is based on the pyvista library and uses its features. If the visualization method is called, the mesh is written to a temporary `.vtk`-file which is then visualized with pyvista. Within an active visualization window, several `key-events` allow for extended features:

`x`	set view to y-z-plane
`y`	set view to z-x-plane
`z`	set view to x-y-plane
`q`	close visualization window
`m`	toggle visualization menu
`space`	confirm settings and re-render
`d`	restore default settings

Since the *normal* way of generating meshes in Gmsh involves the definition of physical groups to, e.g., distinguish different materials, threshold sliders can be used if the visualization menu is activated. They allow to enable or disable different groups according to the defined physical groups in the gmshModel. Additionally, an extraction box widget can be used to extract regions of the mesh and have a closer look to them.





1.3 Examples

In the following examples, basic features of the mesh generation using GmshModel are shown: the basic toolchain is described and meshes with different output formats are exported.

1.3.1 Random distribution of circular inclusions in a rectangular domain

This example shows the generation of an RVE with randomly placed, circular inclusions. The basic procedure of the model and mesh generation are pointed out and the resulting mesh is visualized. For the example, only the standard configuration is used. However, in order to show all available options - user configurations are passed as dictionaries to the individual classes and methods - the dictionaries containing the default values are passed. This means that, if they were not passed, the resulting mesh would be the same.

Code

```
# Loading of the RandomInclusionRVE class
# Before the model and mesh generation can start, the required class has to be
# loaded. In this case it is the class RandomInclusionRVE
from gmshModel.Model import RandomInclusionRVE as RVE


# Initialization of the RVE
# In order to generate a mesh for RVEs with randomly placed inclusions, relevant
# data have to be passed for the initialization of a new object instance. For
# RVEs of the type under consideration, the following parameters are possible:
```

(continues on next page)

(continued from previous page)

```
#
# size: list/array (mandatory)
#   array defining the size of the RVE in the individual directions
#   -> size=[L_x, L_y, L_z]
#
# inclusionSets: list/array (mandatory)
#   array defining the relevant information (radius and amount) for the individual
#   groups of spherical inclusions to be placed
#   -> inclusionSets=[[r_1, n_1] [r_2, n_2], ..., [r_n, n_n]]
#
# inclusionType: string (mandatory)
#   string defining the type of inclusions within the RVE
#
# origin: list/array (optional)
#   array defining the origin of the RVE
#   -> origin=[O_x, O_y, O_z]
#
# periodicityFlags: list/array (optional)
#   array with flags (0/1) whether the current coordinate direction has to be
#   treated as periodic
#   periodicityFlags=[0/1, 0/1, 0/1]
#
# domainGroup: string (optional)
#   string defining which group the geometric objects defining the domain belong
#   to (to reference this group within boolean operations)
#
# inclusionGroup: string (optional)
#   string defining which group the geometric objects defining the inclusions
#   belong to (to reference this group within boolean operations)
#
# gmshConfigChanges: dict (optional)
#   dictionary for user updates of the default Gmsh configuration
#
initParameters={                                                                #␣
↪save all possible parameters in one dict to facilitate the method call
    "inclusionSets": [[1, 8], [0.5, 10]],                                       #␣
↪place 8 inclusions with radius 1 and 10 inclusions with radius 0.5
    "inclusionType": "Circle",                                                  #␣
↪define inclusionType as "Circle"
    "size": [10, 10, 10],                                                       # set␣
↪RVE size to [10,10,10]
    "origin": [0, 0, 0],                                                        # set␣
↪RVE origin to [0,0,0]
    "periodicityFlags": [1, 1, 1],                                              #␣
↪define all axis directions as periodic
    "domainGroup": "domain",                                                    # use
↪"domain" as name for the domainGroup
    "inclusionGroup": "inclusions",                                             # use
↪"inclusions" as name for the inclusionGroup
    "gmshConfigChanges": {"General.Terminal": 0,                                #␣
↪deactivate console output by default (only activated for mesh generation)
                          "Mesh.CharacteristicLengthExtendFromBoundary": 0,     # do␣
↪not calculate mesh sizes from the boundary by default (since mesh sizes are␣
↪specified by fields)
    }
}
testRVE=RandomInclusionRVE(**initParameters)
```

(continues on next page)

(continued from previous page)

```
# Gmsh model generation
# After all parameters for the RVE are set, the Gmsh model can be generated.
# This process involves the definition of geometric objects, their combination
# to more complex shapes using boolean operations and the definition of physical
# groups, i.e. groups of elements that belong to the same material or part of
# the boundary. For RVEs with randomly placed inclusions, only the placement
# options can be changed by the user. To this end, the possible parameters are:
#
# placementOptions: dict (optional)
#   user updates for the inclusion placement algorithm
#
modelingParameters={                                                          #␣
↪save all possible parameters in one dict to facilitate the method call
    "placementOptions": {"maxAttempts": 10000,                                #␣
↪maximum number of attempts to place one inclusion
                         "minRelDistBnd": 0.1,                                #␣
↪minimum relative (to inclusion radius) distance to the domain boundaries
                         "minRelDistInc": 0.1,                                #␣
↪minimum relative (to inclusion radius) distance to other inclusions}
    }
}
testRVE.createGmshModel(**modelingParameters)


# Gmsh mesh creation
# After the model has been created using the Gmsh-Python-API, the meshing
# can be performed. To this end, refinement fields defining the mesh sizes
# within the model have to be calculated and added to the Gmsh model. Once, the
# mesh sizes are specified,the mesh can be generated. Available parameters are:
#
# threads: int
#   number of threads to use for the meshing procedure
# refinementOptions: dict (optional)
#   dictionary containing user updates for the refinement field calculation
#
meshingParameters={                                                           #␣
↪save all possible parameters in one dict to facilitate the method call
    "threads": None,                                                          # do␣
↪not activate parallel meshing by default
    "refinementOptions": {"maxMeshSize": "auto",                              #␣
↪automatically calculate maximum mesh size with built-in method
                          "inclusionRefinement": True,                        #␣
↪flag to indicate active refinement of inclusions
                          "interInclusionRefinement": True,                   #␣
↪flag to indicate active refinement of space between inclusions (inter-inclusion␣
↪refinement)
                          "elementsPerCircumference": 18,                     # use␣
↪18 elements per inclusion circumference for inclusion refinement
                          "elementsBetweenInclusions": 3,                     #␣
↪ensure 3 elements between close inclusions for inter-inclusion refinement
                          "inclusionRefinementWidth": 3,                      # use␣
↪a relative (to inclusion radius) refinement width of 3 for inclusion refinement
                          "transitionElements": "auto",                       #␣
↪automatically calculate number of transitioning elements (elements in which tanh␣
↪function jumps from h_min to h_max) for inter-inclusion refinement
```

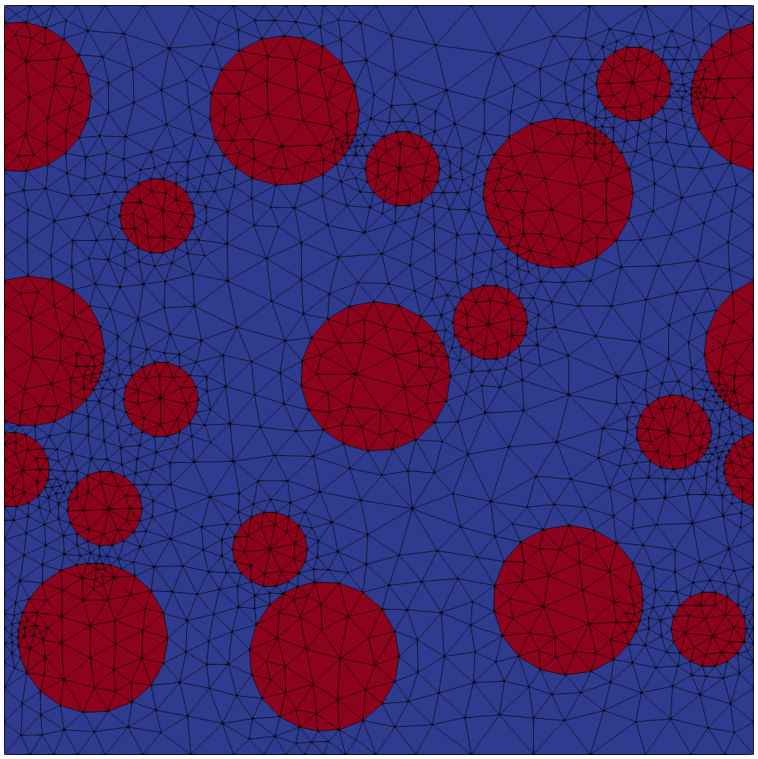
(continues on next page)

(continued from previous page)

```
                        "aspectRatio": 1.5                                    #␣
↪aspect ratio for inter-inclusion refinement: ratio of refinement in inclusion␣
↪distance and perpendicular directions
    }
}
testRVE.createMesh(**meshingParameters)


# Save resulting mesh to file
# The mesh is generated and can be saved to a file. To this end, only the file
# name - possibly containing a directory and the extension of the wanted mesh
# format - has to be passed. The package supports all mesh file formats that are
# supported by meshio. If no filename is passed, meshes are stored to the current
# directory using the unique model name and the default mesh file format (.msh)
#
testRVE.saveMesh("randomInclusions2DCirlce.vtu")


# Show resulting mesh
# To check the generated mesh, the result can also be visualized using built-in
# methods.
#
testRVE.visualizeMesh()


# Close Gmsh model
# For a proper closing of the Gmsh-Python-API, thAPI has to be finalized. This
# can be achieved by calling the close() method of the model
#
testRVE.close()
```

Result

If the mesh generation is successful, the result should look similar to the following:



Since the geometry involves a random placement of the circular inclusions, the mesh will slightly vary for each run of the example. However, in the end there should always be `18` circular inclusions with two different radii. The applied (default) refinement options try to ensure that there are about 3 elements between close inclusions and around `18` elements per inclusion circumference.

1.3.2 Random distribution of spherical inclusions in a box-shaped domain

This example shows the generation of an RVE with randomly placed, spherical inclusions. The basic procedure of the model and mesh generation are pointed out and the resulting mesh is visualized. For the example, only the standard configuration is used. However, in order to show the available options - all user configurations are passed as dictionaries to the individual classes and methods - the dictionaries containing the default values are passed. This means that, if they were not passed, the resulting mesh would be the same.

Code

```
# Loading of the RandomInclusionRVE class
# Before the model and mesh generation can start, the required class has to be
# loaded. In this case it is the class RandomInclusionRVE
from ..src.typeRandomInclusionRVE import RandomInclusionRVE


# Initialization of the RVE
# In order to generate a mesh for RVEs with randomly placed inclusions, relevant
# data have to be passed for the initialization of a new object instance. For
# RVEs of the type under consideration, the following parameters are possible:
#
# size: list/array (mandatory)
#   array defining the size of the RVE in the individual directions
```

(continues on next page)

(continued from previous page)

```
#   size=[L_x, L_y, L_z]
#
# inclusionSets: list/array (mandatory)
#   array defining the relevant information (radius and amount) for the individual
#   groups of spherical inclusions to be placed
#   inclusionSets=[[r_1, n_1] [r_2, n_2], ..., [r_n, n_n]]
#
# inclusionType: string (mandatory)
#   string defining the type of inclusions within the RVE
#
# origin: list/array (optional)
#   array defining the origin of the RVE
#   origin=[O_x, O_y, O_z]
#
# periodicityFlags: list/array (optional)
#   array with flags (0/1) whether the current coordinate direction has to be
#   treated as periodic
#   periodicityFlags=[0/1, 0/1, 0/1]
#
# domainGroup: string (optional)
#   string defining which group the geometric objects defining the domain belong
#   to (to reference this group within boolean operations)
#
# inclusionGroup: string (optional)
#   string defining which group the geometric objects defining the inclusions
#   belong to (to reference this group within boolean operations)
#
# gmshConfigChanges: dict (optional)
#   dictionary for user updates of the default Gmsh configuration
#
initParameters={                                                                #␣
↪save all possible parameters in one dict to facilitate the method call
    "inclusionSets": [1, 12],                                                   #␣
↪place 12 inclusions with radius 1
    "inclusionType": "Sphere",                                                  #␣
↪define inclusionType as "Sphere"
    "size": [6, 6, 6],                                                          # set␣
↪RVE size to [6,6,6]
    "origin": [0, 0, 0],                                                        # set␣
↪RVE origin to [0,0,0]
    "periodicityFlags": [1, 1, 1],                                              #␣
↪define all axis directions as periodic
    "domainGroup": "domain",                                                    # use
↪"domain" as name for the domainGroup
    "inclusionGroup": "inclusions",                                             # use
↪"inclusions" as name for the inclusionGroup
    "gmshConfigChanges": {"General.Terminal": 0,                                #␣
↪deactivate console output by default (only activated for mesh generation)
                          "Mesh.CharacteristicLengthExtendFromBoundary": 0,     # do␣
↪not calculate mesh sizes from the boundary by default (since mesh sizes are␣
↪specified by fields)
    }
}
testRVE=RandomInclusionRVE(**initParameters)


# Gmsh model generation
```

(continues on next page)

(continued from previous page)

```
# After all parameters for the RVE are set, the Gmsh model can be generated.
# This process involves the definition of geometric objects, their combination
# to more complex shapes using boolean operations and the definition of physical
# groups, i.e. groups of elements that belong to the same material or part of
# the boundary. For RVEs with randomly placed inclusions, only the placement
# options can be changed by the user. To this end, the possible parameters are:
#
# placementOptions: dict (optional)
#   user updates for the inclusion placement algorithm
modelingParameters={                                                          #␣
↪save all possible parameters in one dict to facilitate the method call
    "placementOptions": {"maxAttempts": 10000,                                #␣
↪maximum number of attempts to place one inclusion
                         "minRelDistBnd": 0.1,                                #␣
↪minimum relative (to inclusion radius) distance to the domain boundaries
                         "minRelDistInc": 0.1,                                #␣
↪minimum relative (to inclusion radius) distance to other inclusions}
    }
}
testRVE.createGmshModel(**modelingParameters)


# Gmsh mesh creation
# After the model has been created using the Gmsh-Python-API, the meshing
# can be performed. To this end, refinement fields defining the mesh sizes
# within the model have to be calculated and added to the Gmsh model. Once, the
# mesh sizes are specified,the mesh can be generated. Available parameters are:
#
# threads: int
#   number of threads to use for the meshing procedure
# refinementOptions: dict (optional)
#   dictionary containing user updates for the refinement field calculation
#
meshingParameters={                                                           #␣
↪save all possible parameters in one dict to facilitate the method call
    "threads": None,                                                          # do␣
↪not activate parallel meshing by default
    "refinementOptions": {"maxMeshSize": "auto",                              #␣
↪automatically calculate maximum mesh size with built-in method
                          "inclusionRefinement": True,                        #␣
↪flag to indicate active refinement of inclusions
                          "interInclusionRefinement": True,                   #␣
↪flag to indicate active refinement of space between inclusions (inter-inclusion␣
↪refinement)
                          "elementsPerCircumference": 18,                     # use␣
↪18 elements per inclusion circumference for inclusion refinement
                          "elementsBetweenInclusions": 3,                     #␣
↪ensure 3 elements between close inclusions for inter-inclusion refinement
                          "inclusionRefinementWidth": 3,                      # use␣
↪a relative (to inclusion radius) refinement width of 3 for inclusion refinement
                          "transitionElements": "auto",                       #␣
↪automatically calculate number of transitioning elements (elements in which tanh␣
↪function jumps from h_min to h_max) for inter-inclusion refinement
                          "aspectRatio": 1.5                                  #␣
↪aspect ratio for inter-inclusion refinement: ratio of refinement in inclusion␣
↪distance and perpendicular directions
    }
```

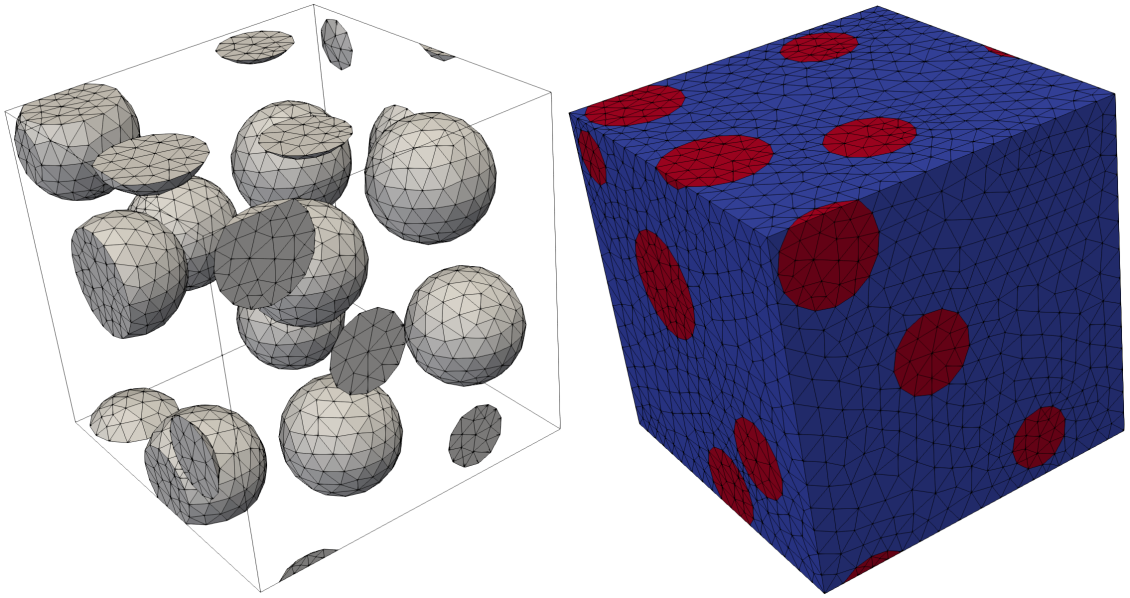
(continues on next page)

(continued from previous page)

```
}
testRVE.createMesh(**meshingParameters)


# Save resulting mesh to file
# The mesh is generated and can be saved to a file. To this end, only the file
# name - possibly containing a directory and the extension of the wanted mesh
# format - has to be passed. The package supports all mesh file formats that are
# supported by meshio. If no filename is passed, meshes are stored to the current
# directory using the unique model name and the default mesh file format (.msh)
#
testRVE.saveMesh("randomInclusions3DSphere.feap")


# Show resulting mesh
# To check the generated mesh, the result can also be visualized using built-in
# methods.
#
testRVE.visualizeMesh()


# Close Gmsh model
# For a proper closing of the Gmsh-Python-API, thAPI has to be finalized. This
# can be achieved by calling the close() method of the model
#
testRVE.close()
```

Result

If the mesh generation is successful, the result should look similar to the following:



Since the geometry involves a random placement of the spherical inclusions, the mesh will slightly vary for each run of the example. However, in the end there should always be `12` cylindrical inclusions that are periodically continued over all boundaries. The applied (default) refinement options try to ensure that there are about 3 elements between close inclusions and around `18` elements per inclusion circumference.

1.3.3 Random distribution of cylindrical inclusions in a box-shaped domain

This example shows the generation of an RVE with randomly placed, cylindrical inclusions. The basic procedure of the model and mesh generation are pointed out and the resulting mesh is visualized. For the example, only the standard configuration is used. However, in order to show all available options - user configurations are passed as dictionaries to the individual classes and methods - the dictionaries containing the default values are passed. This means that, if they were not passed, the resulting mesh would be the same.

Code

```
# Loading of the SimpleCubicCell class
# Before the model and mesh generation can start, the required class has to be
# loaded. In this case it is the class SimpleCubicCell
from gmshModel.Model import SimpleCubicCell as Cell


# Initialization of the unit cell
# In order to generate a mesh for simple cubic unit cells with spherical inclusions,
# relevant data have to be passed for the initialization of a new object instance. For
# unit cells of the type under consideration, the following parameters are possible:
#
# size: list/array (mandatory)
#   array defining the size of the RVE in the individual directions
#   -> size=[L_x, L_y, L_z]
#
# inclusionSets: list/array (mandatory)
#   array defining the relevant information (radius and amount) for the individual
#   groups of spherical inclusions to be placed
#   -> inclusionSets=[[r_1, n_1] [r_2, n_2], ..., [r_n, n_n]]
#
# inclusionType: string (mandatory)
#   string defining the type of inclusions within the RVE
#
# inclusionAxis: list/string (mandatory)
#   array defining the cylinder axis/direction
#   -> inclusionAxis=[A_x, A_y, A_z]
#
# origin: list/array (optional)
#   array defining the origin of the RVE
#   -> origin=[O_x, O_y, O_z]
#
# periodicityFlags: list/array (optional)
#   array with flags (0/1) whether the current coordinate direction has to be
#   treated as periodic
#   periodicityFlags=[0/1, 0/1, 0/1]
#
# domainGroup: string (optional)
#   string defining which group the geometric objects defining the domain belong
#   to (to reference this group within boolean operations)
#
# inclusionGroup: string (optional)
#   string defining which group the geometric objects defining the inclusions
#   belong to (to reference this group within boolean operations)
#
# gmshConfigChanges: dict (optional)
#   dictionary for user updates of the default Gmsh configuration
```

(continues on next page)

(continued from previous page)

```
#
initParameters={                                                                #␣
↪save all possible parameters in one dict to facilitate the method call
    "inclusionSets": [1, 13],                                                   #␣
↪place 13 inclusions with radius 1
    "inclusionType": "Cylinder",                                                #␣
↪define inclusionType as "Cylinder"
    "inclusionAxis": [0, 0, 2.5],                                               #␣
↪define inclusionAxis direction
    "size": [10, 10, 2.5],                                                      # set␣
↪RVE size to [10,10,2.5]
    "origin": [0, 0, 0],                                                        # set␣
↪RVE origin to [0,0,0]
    "periodicityFlags": [1, 1, 1],                                              #␣
↪define all axis directions as periodic
    "domainGroup": "domain",                                                    # use
↪"domain" as name for the domainGroup
    "inclusionGroup": "inclusions",                                             # use
↪"inclusions" as name for the inclusionGroup
    "gmshConfigChanges": {"General.Terminal": 0,                                #␣
↪deactivate console output by default (only activated for mesh generation)
                          "Mesh.CharacteristicLengthExtendFromBoundary": 0,     # do␣
↪not calculate mesh sizes from the boundary by default (since mesh sizes are␣
↪specified by fields)
    }
}
testRVE=RandomInclusionRVE(**initParameters)


# Gmsh model generation
# After all parameters for the RVE are set, the Gmsh model can be generated.
# This process involves the definition of geometric objects, their combination
# to more complex shapes using boolean operations and the definition of physical
# groups, i.e. groups of elements that belong to the same material or part of
# the boundary. For RVEs with randomly placed inclusions, only the placement
# options can be changed by the user. To this end, the possible parameters are:
#
# placementOptions: dict (optional)
#   user updates for the inclusion placement algorithm
#
modelingParameters={                                                            #␣
↪save all possible parameters in one dict to facilitate the method call
    "placementOptions": {"maxAttempts": 10000,                                  #␣
↪maximum number of attempts to place one inclusion
                         "minRelDistBnd": 0.1,                                  #␣
↪minimum relative (to inclusion radius) distance to the domain boundaries
                         "minRelDistInc": 0.1,                                  #␣
↪minimum relative (to inclusion radius) distance to other inclusions}
    }
}
testRVE.createGmshModel(**modelingParameters)


# Gmsh mesh creation
# After the model has been created using the Gmsh-Python-API, the meshing
# can be performed. To this end, refinement fields defining the mesh sizes
# within the model have to be calculated and added to the Gmsh model. Once, the
```

(continues on next page)

(continued from previous page)

```
# mesh sizes are specified,the mesh can be generated. Available parameters are:
#
# threads: int
#   number of threads to use for the meshing procedure
#
# refinementOptions: dict (optional)
#   dictionary containing user updates for the refinement field calculation
#
meshingParameters={                                                              #␣
↪save all possible parameters in one dict to facilitate the method call
    "threads": None,                                                             # do␣
↪not activate parallel meshing by default
    "refinementOptions": {"maxMeshSize": "auto",                                 #␣
↪automatically calculate maximum mesh size with built-in method
                          "inclusionRefinement": True,                           #␣
↪flag to indicate active refinement of inclusions
                          "interInclusionRefinement": True,                      #␣
↪flag to indicate active refinement of space between inclusions (inter-inclusion␣
↪refinement)
                          "elementsPerCircumference": 18,                        # use␣
↪18 elements per inclusion circumference for inclusion refinement
                          "elementsBetweenInclusions": 3,                        #␣
↪ensure 3 elements between close inclusions for inter-inclusion refinement
                          "inclusionRefinementWidth": 3,                         # use␣
↪a relative (to inclusion radius) refinement width of 3 for inclusion refinement
                          "transitionElements": "auto",                          #␣
↪automatically calculate number of transitioning elements (elements in which tanh␣
↪function jumps from h_min to h_max) for inter-inclusion refinement
                          "aspectRatio": 1.5                                     #␣
↪aspect ratio for inter-inclusion refinement: ratio of refinement in inclusion␣
↪distance and perpendicular directions
    }
}
testRVE.createMesh(**meshingParameters)


# Save resulting mesh to file
# The mesh is generated and can be saved to a file. To this end, only the file
# name - possibly containing a directory and the extension of the wanted mesh
# format - has to be passed. The package supports all mesh file formats that are
# supported by meshio. If no filename is passed, meshes are stored to the current
# directory using the unique model name and the default mesh file format (.msh)
#
testRVE.saveMesh("randomInclusions3DCylinder.xdmf")


# Show resulting mesh
# To check the generated mesh, the result can also be visualized using built-in
# methods.
#
testRVE.visualizeMesh()


# Close Gmsh model
# For a proper closing of the Gmsh-Python-API, thAPI has to be finalized. This
# can be achieved by calling the close() method of the model
#
```

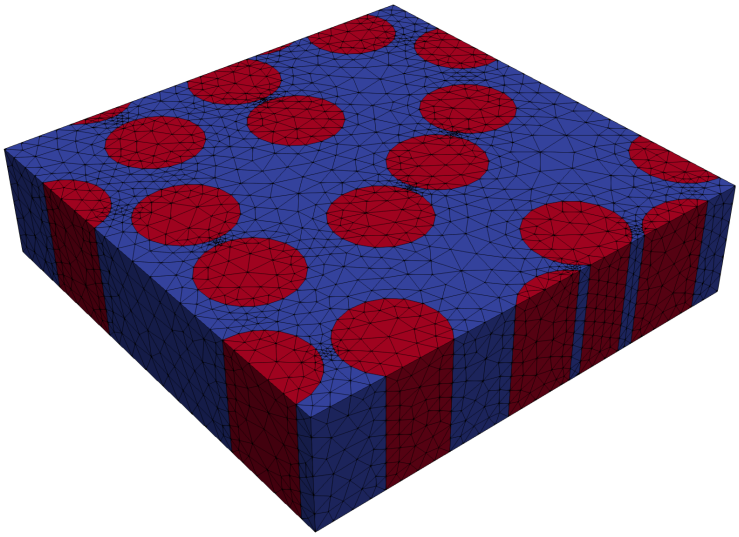
(continues on next page)

(continued from previous page)

```
testRVE.close()
```

Result

If the mesh generation is successful, the result should look similar to the following:



Since the geometry involves a random placement of the cylindrical inclusions, the mesh will slightly vary for each run of the example. However, in the end there should always be `13` cylindrical inclusions that are periodically continued over all boundaries. The applied (default) refinement options try to ensure that there are about 3 elements between close inclusions and around `18` elements per inclusion circumference.

1.3.4 Simple cubic unit cell with spherical inclusions

This example shows the generation of a unit cell with a simple cubic distribution of spherical inclusions. The basic procedures of the model and mesh generation are pointed out and the resulting mesh is visualized. For the example, only the standard configuration is used. However, in order to show all available options - user configurations are passed as dictionaries to the individual classes and methods - the dictionaries containing the default values are passed. This means that, if they were not passed, the resulting mesh would be the same.

Code

```
# Loading of the SimpleCubicUnitCell class
# Before the model and mesh generation can start, the required class has to be
# loaded. In this case it is the class SimpleCubicCell
from gmshModel.Model import SimpleCubicCell

# Initialization of the unit cell
# In order to generate a mesh for unit cells with a simple cubic distribution of
# spherical inclusions, relevant data have to be passed for the initialization of
# a new object instance. For unit cells of the type under consideration, the
# following parameters are possible:
#
# radius: float (mandatory)
```

(continues on next page)

(continued from previous page)

```
#   radius of the inclusions within the unit cell
#
# distance: float (defining either distance or size is mandatory)
#   distance of the inclusions within the unit cell
#   -> if the distance is given, the cells size is calculated automatically
#
# size: list/array (defining either distance or size is mandatory)
#   array defining the size of the RVE in the individual directions
#   -> size=[L_x, L_y, (L_z)]
#   -> if the size is given, the inclusion distances are calculated automatically
#      (this allows more flexibility and unit cells with inclusion distributions
#       that are similar to the physical unit cell under consideration)
#
# numberCells: list/array (optional)
#   array defining the number of cells in the 3 spatial axis directions
#   -> numberCells=[n_x, n_y, n_z]
#
# inclusionType: string (mandatory)
#   string defining the type of inclusions within the RVE
#
# origin: list/array (optional)
#   array defining the origin of the RVE
#   -> origin=[O_x, O_y, (O_z)]
#
# periodicityFlags: list/array (optional)
#   array with flags (0/1) whether the current coordinate direction has to be
#   treated as periodic
#   periodicityFlags=[0/1, 0/1, 0/1]
#
# domainGroup: string (optional)
#   string defining which group the geometric objects defining the domain belong
#   to (to reference this group within boolean operations)
#
# inclusionGroup: string (optional)
#   string defining which group the geometric objects defining the inclusions
#   belong to (to reference this group within boolean operations)
#
# gmshConfigChanges: dict (optional)
#   dictionary for user updates of the default Gmsh configuration
#
initParameters={                                                                #␣
↪save all possible parameters in one dict to facilitate the method call
    "numberCells": [1,1,1],                                                     #␣
↪generate 1 unit cell in every spatial direction
    "radius": 2,                                                                # set␣
↪the inclusion radius to 2
    "inclusionType": "Sphere",                                                  #␣
↪define inclusionType as "Sphere"
    "distance": 8,                                                              # set␣
↪the inclusion distance to 8 and calculate the correspondig cell size
    "origin": [0, 0, 0],                                                        # set␣
↪cell origin to [0,0,0]
    "periodicityFlags": [1, 1, 1],                                              #␣
↪define all axis directions as periodic
    "domainGroup": "domain",                                                    # use
↪"domain" as name for the domainGroup
    "inclusionGroup": "inclusions",                                             # use
↪"inclusions" as name for the inclusionGroup
```

(continues on next page)

(continued from previous page)

```
    "gmshConfigChanges": {"General.Terminal": 0,                                    #␣
↪deactivate console output by default (only activated for mesh generation)
                          "Mesh.CharacteristicLengthExtendFromBoundary": 0,      # do␣
↪not calculate mesh sizes from the boundary by default (since mesh sizes are␣
↪specified by fields)
    }
}
testCell=SimpleCubicCell(**initParameters)


# Gmsh model generation
# After all parameters for the unit cell are set, the Gmsh model can be generated.
# This process involves the definition of geometric objects, their combination
# to more complex shapes using boolean operations and the definition of physical
# groups, i.e. groups of elements that belong to the same material or part of
# the boundary. For unit cells no additional options are required for the
# inclusion placement. To this end, the command is simply:
#
testCell.createGmshModel()

# Gmsh mesh creation
# After the model has been created using the Gmsh-Python-API, the meshing
# can be performed. To this end, refinement fields defining the mesh sizes
# within the model have to be calculated and added to the Gmsh model. Once, the
# mesh sizes are specified,the mesh can be generated. Available parameters are:
#
# threads: int
#   number of threads to use for the meshing procedure
# refinementOptions: dict (optional)
#   dictionary containing user updates for the refinement field calculation
#
meshingParameters={                                                                 #␣
↪save all possible parameters in one dict to facilitate the method call
    "threads": None,                                                                # do␣
↪not activate parallel meshing by default
    "refinementOptions": {"maxMeshSize": "auto",                                    #␣
↪automatically calculate maximum mesh size with built-in method
                          "inclusionRefinement": True,                              #␣
↪flag to indicate active refinement of inclusions
                          "interInclusionRefinement": True,                         #␣
↪flag to indicate active refinement of space between inclusions (inter-inclusion␣
↪refinement)
                          "elementsPerCircumference": 18,                           # use␣
↪18 elements per inclusion circumference for inclusion refinement
                          "elementsBetweenInclusions": 3,                           #␣
↪ensure 3 elements between close inclusions for inter-inclusion refinement
                          "inclusionRefinementWidth": 3,                            # use␣
↪a relative (to inclusion radius) refinement width of 3 for inclusion refinement
                          "transitionElements": "auto",                             #␣
↪automatically calculate number of transitioning elements (elements in which tanh␣
↪function jumps from h_min to h_max) for inter-inclusion refinement
                          "aspectRatio": 1.5                                        #␣
↪aspect ratio for inter-inclusion refinement: ratio of refinement in inclusion␣
↪distance and perpendicular directions
    }
}
testCell.createMesh(**meshingParameters)
```

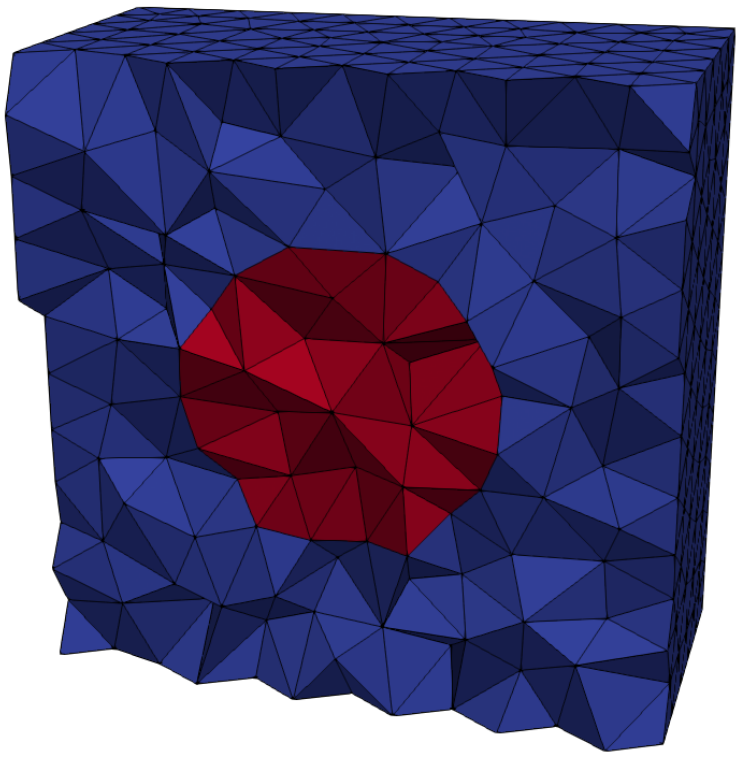
(continues on next page)

(continued from previous page)

```
# Save resulting mesh to file
# The mesh is generated and can be saved to a file. To this end, only the file
# name - possibly containing a directory and the extension of the wanted mesh
# format - has to be passed. The package supports all mesh file formats that are
# supported by meshio. If no filename is passed, meshes are stored to the current
# directory using the unique model name and the default mesh file format (.msh)
testCell.saveMesh("simpleCubicCell3DSphere.vtu")


# Show resulting mesh
# To check the generated mesh, the result can also be visualized using built-in
# methods.
testCell.visualizeMesh()


# Close Gmsh model
# For a proper closing of the Gmsh-Python-API, the API has to be finalized. This
# can be achieved by calling the close() method of the model
testCell.close()
```

Result

If the mesh generation is successful, the result should look like this:



Since the inclusion is fully embedded in the surrounding matrix material, the BoxWidget of the PyVista-based visualization tool is used to extract the part of the mesh that can be seen in the image above.

1.3.5 Body-centered cubic unit cell with circular inclusions

This example shows the generation of a unit cell with a body-centered cubic distribution of circular inclusions. The basic procedures of the model and mesh generation are pointed out and the resulting mesh is visualized. For the example, only the standard configuration is used. However, in order to show all available options - user configurations are passed as dictionaries to the individual classes and methods - the dictionaries containing the default values are passed. This means that, if they were not passed, the resulting mesh would be the same.

Code

```
# Loading of the BodyCenteredCubicUnitCell class
# Before the model and mesh generation can start, the required class has to be
# loaded. In this case it is the class BodyCenteredCubicCell
from gmshModel.Model import BodyCenteredCubicCell

# Initialization of the unit cell
# In order to generate a mesh for unit cells with a body-centered cubic distribution
# of circular inclusions, relevant data have to be passed for the initialization
# of a new object instance. For unit cells of the type under consideration, the
# following parameters are possible:
#
# radius: float (mandatory)
#   radius of the inclusions within the unit cell
#
# distance: float (defining either distance or size is mandatory)
#   distance of the inclusions within the unit cell
#   -> if the distance is given, the cells size is calculated automatically
#
# size: list/array (defining either distance or size is mandatory)
#   array defining the size of the RVE in the individual directions
#   -> size=[L_x, L_y, (L_z)]
#   -> if the size is given, the inclusion distances are calculated automatically
#      (this allows more flexibility and unit cells with inclusion distributions
#       that are similar to the physical unit cell under consideration)
#
# numberCells: list/array (optional)
#   array defining the number of cells in the 3 spatial axis directions
#   -> numberCells=[n_x, n_y, n_z]
#
# inclusionType: string (mandatory)
#   string defining the type of inclusions within the RVE
#
# origin: list/array (optional)
#   array defining the origin of the RVE
#   -> origin=[O_x, O_y, O_z]
#
# periodicityFlags: list/array (optional)
#   array with flags (0/1) whether the current coordinate direction has to be
#   treated as periodic
#   periodicityFlags=[0/1, 0/1, 0/1]
#
# domainGroup: string (optional)
#   string defining which group the geometric objects defining the domain belong
#   to (to reference this group within boolean operations)
#
# inclusionGroup: string (optional)
```

(continues on next page)

(continued from previous page)

```
#   string defining which group the geometric objects defining the inclusions
#   belong to (to reference this group within boolean operations)
#
# gmshConfigChanges: dict (optional)
#   dictionary for user updates of the default Gmsh configuration
#
initParameters={                                                                #␣
↪save all possible parameters in one dict to facilitate the method call
    "numberCells": [3,3,1],                                                     #␣
↪generate 3 unit cells in the in-plane direction and one along the third axis␣
↪direction
    "radius": 2,                                                                # set␣
↪the inclusion radius to 2
    "distance": 6,                                                              # set␣
↪the inclusion distance to 6 and calculate the correspondig cell size
    "inclusionType": "Circle",                                                  #␣
↪define inclusionType as "Circle"
    "origin": [10, 10, 0],                                                      # set␣
↪cell origin to [10,10,0]
    "periodicityFlags": [1, 1, 1],                                              #␣
↪define all axis directions as periodic
    "domainGroup": "domain",                                                    # use
↪"domain" as name for the domainGroup
    "inclusionGroup": "inclusions",                                             # use
↪"inclusions" as name for the inclusionGroup
    "gmshConfigChanges": {"General.Terminal": 0,                                #␣
↪deactivate console output by default (only activated for mesh generation)
                          "Mesh.CharacteristicLengthExtendFromBoundary": 0,     # do␣
↪not calculate mesh sizes from the boundary by default (since mesh sizes are␣
↪specified by fields)
    }
}
testCell=BodyCenteredCubicCell(**initParameters)


# Gmsh model generation
# After all parameters for the unit cell are set, the Gmsh model can be generated.
# This process involves the definition of geometric objects, their combination
# to more complex shapes using boolean operations and the definition of physical
# groups, i.e. groups of elements that belong to the same material or part of
# the boundary. For unit cells no additional options are required for the
# inclusion placement. To this end, the command is simply:
#
testCell.createGmshModel()

# Gmsh mesh creation
# After the model has been created using the Gmsh-Python-API, the meshing
# can be performed. To this end, refinement fields defining the mesh sizes
# within the model have to be calculated and added to the Gmsh model. Once, the
# mesh sizes are specified,the mesh can be generated. Available parameters are:
#
# threads: int
#   number of threads to use for the meshing procedure
# refinementOptions: dict (optional)
#   dictionary containing user updates for the refinement field calculation
#
meshingParameters={                                                             #␣
↪save all possible parameters in one dict to facilitate the method call
```

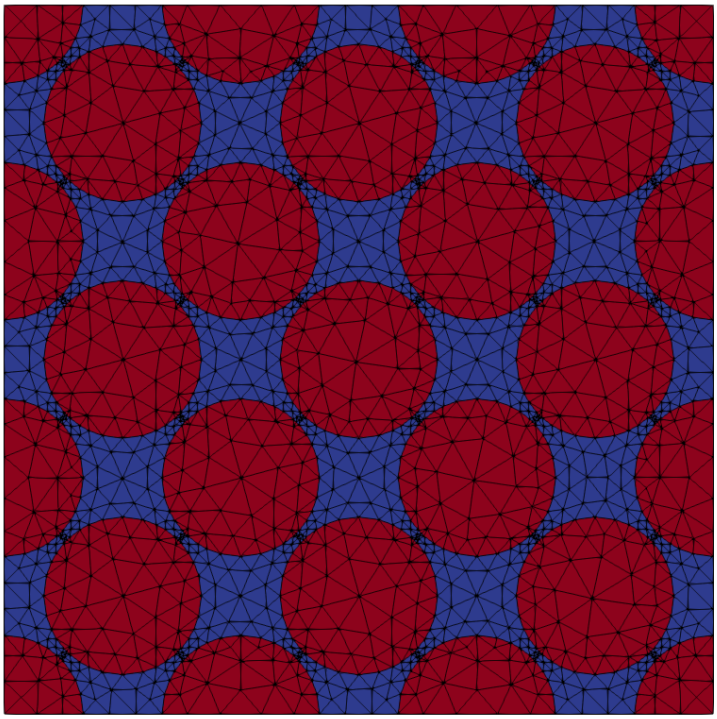
(continues on next page)

(continued from previous page)

```
    "threads": None,                                                    # do␣
↪not activate parallel meshing by default
    "refinementOptions": {"maxMeshSize": "auto",                        #␣
↪automatically calculate maximum mesh size with built-in method
                          "inclusionRefinement": True,                  #␣
↪flag to indicate active refinement of inclusions
                          "interInclusionRefinement": True,             #␣
↪flag to indicate active refinement of space between inclusions (inter-inclusion␣
↪refinement)
                          "elementsPerCircumference": 18,               # use␣
↪18 elements per inclusion circumference for inclusion refinement
                          "elementsBetweenInclusions": 3,               #␣
↪ensure 3 elements between close inclusions for inter-inclusion refinement
                          "inclusionRefinementWidth": 3,                # use␣
↪a relative (to inclusion radius) refinement width of 3 for inclusion refinement
                          "transitionElements": "auto",                 #␣
↪automatically calculate number of transitioning elements (elements in which tanh␣
↪function jumps from h_min to h_max) for inter-inclusion refinement
                          "aspectRatio": 1.5                            #␣
↪aspect ratio for inter-inclusion refinement: ratio of refinement in inclusion␣
↪distance and perpendicular directions
    }
}
testCell.createMesh(**meshingParameters)


# Save resulting mesh to file
# The mesh is generated and can be saved to a file. To this end, only the file
# name - possibly containing a directory and the extension of the wanted mesh
# format - has to be passed. The package supports all mesh file formats that are
# supported by meshio. If no filename is passed, meshes are stored to the current
# directory using the unique model name and the default mesh file format (.msh)
testCell.saveMesh("bodyCenteredCubicCell2DCircle.msh")


# Show resulting mesh
# To check the generated mesh, the result can also be visualized using built-in
# methods.
testCell.visualizeMesh()


# Close Gmsh model
# For a proper closing of the Gmsh-Python-API, the API has to be finalized. This
# can be achieved by calling the close() method of the model
testCell.close()
```

Result

If the mesh generation is successful, the result should look like this:



The mesh refinement between close inclusions is visible: a closer look reveals that indeed at least 3 elements between the inclusions are ensured.

1.3.6 Face-centered cubic unit cell with cylindrical inclusions

This example shows the generation of a unit cell with a face-centered cubic distribution of cylindrical inclusions. The basic procedures of the model and mesh generation are pointed out and the resulting mesh is visualized. For the example, only the standard configuration is used. However, in order to show all available options - user configurations are passed as dictionaries to the individual classes and methods - the dictionaries containing the default values are passed. This means that, if they were not passed, the resulting mesh would be the same.

Code

```
# Loading of the FaceCenteredCubicUnitCell class
# Before the model and mesh generation can start, the required class has to be
# loaded. In this case it is the class FaceCenteredCubicCell
from gmshModel.Model import FaceCenteredCubicCell

# Initialization of the unit cell
# In order to generate a mesh for unit cells with a face-centered cubic distribution
# of cylindrical inclusions, relevant data have to be passed for the initialization
# of a new object instance. For unit cells of the type under consideration, the
# following parameters are possible:
#
# radius: float (mandatory)
#   radius of the inclusions within the unit cell
#
# distance: float (defining either distance or size is mandatory)
#   distance of the inclusions within the unit cell
```

(continues on next page)

(continued from previous page)

```
#   -> if the distance is given, the cells size is calculated automatically
#
# size: list/array (defining either distance or size is mandatory)
#   array defining the size of the RVE in the individual directions
#   -> size=[L_x, L_y, (L_z)]
#   -> if the size is given, the inclusion distances are calculated automatically
#      (this allows more flexibility and unit cells with inclusion distributions
#       that are similar to the physical unit cell under consideration)
#
# numberCells: list/array (optional)
#   array defining the number of cells in the 3 spatial axis directions
#   -> numberCells=[n_x, n_y, n_z]
#
# inclusionType: string (mandatory)
#   string defining the type of inclusions within the RVE
#
# origin: list/array (optional)
#   array defining the origin of the RVE
#   -> origin=[O_x, O_y, O_z]
#
# periodicityFlags: list/array (optional)
#   array with flags (0/1) whether the current coordinate direction has to be
#   treated as periodic
#   periodicityFlags=[0/1, 0/1, 0/1]
#
# domainGroup: string (optional)
#   string defining which group the geometric objects defining the domain belong
#   to (to reference this group within boolean operations)
#
# inclusionGroup: string (optional)
#   string defining which group the geometric objects defining the inclusions
#   belong to (to reference this group within boolean operations)
#
# gmshConfigChanges: dict (optional)
#   dictionary for user updates of the default Gmsh configuration
#
initParameters={                                                                #␣
↪save all possible parameters in one dict to facilitate the method call
    "numberCells": [2,2,1],                                                     #␣
↪generate 2 unit cells in the in-plane direction and one along the cylinder axis␣
↪direction
    "radius": 2,                                                                # set␣
↪the inclusion radius to 2
    "distance": 6,                                                              # set␣
↪the inclusion distance to 8 and calculate the correspondig cell size
    "inclusionType": "Cylinder",                                                #␣
↪define inclusionType as "Cylinder"
    "inclusionAxis": [0,0,2],                                                   #␣
↪define cylinders to be aligned with the z-axis and have a length of 2
    "origin": [0, 0, 0],                                                        # set␣
↪cell origin to [0,0,0]
    "periodicityFlags": [1, 1, 1],                                              #␣
↪define all axis directions as periodic
    "domainGroup": "domain",                                                    # use
↪"domain" as name for the domainGroup
    "inclusionGroup": "inclusions",                                             # use
↪"inclusions" as name for the inclusionGroup
```

(continues on next page)

(continued from previous page)

```
    "gmshConfigChanges": {"General.Terminal": 0,                                    #␣
↪deactivate console output by default (only activated for mesh generation)
                          "Mesh.CharacteristicLengthExtendFromBoundary": 0,      # do␣
↪not calculate mesh sizes from the boundary by default (since mesh sizes are␣
↪specified by fields)
    }
}
testCell=FaceCenteredCubicCell(**initParameters)


# Gmsh model generation
# After all parameters for the unit cell are set, the Gmsh model can be generated.
# This process involves the definition of geometric objects, their combination
# to more complex shapes using boolean operations and the definition of physical
# groups, i.e. groups of elements that belong to the same material or part of
# the boundary. For unit cells no additional options are required for the
# inclusion placement. To this end, the command is simply:
#
testCell.createGmshModel()

# Gmsh mesh creation
# After the model has been created using the Gmsh-Python-API, the meshing
# can be performed. To this end, refinement fields defining the mesh sizes
# within the model have to be calculated and added to the Gmsh model. Once, the
# mesh sizes are specified,the mesh can be generated. Available parameters are:
#
# threads: int
#   number of threads to use for the meshing procedure
# refinementOptions: dict (optional)
#   dictionary containing user updates for the refinement field calculation
#
meshingParameters={                                                                 #␣
↪save all possible parameters in one dict to facilitate the method call
    "threads": None,                                                                # do␣
↪not activate parallel meshing by default
    "refinementOptions": {"maxMeshSize": "auto",                                    #␣
↪automatically calculate maximum mesh size with built-in method
                          "inclusionRefinement": True,                              #␣
↪flag to indicate active refinement of inclusions
                          "interInclusionRefinement": True,                         #␣
↪flag to indicate active refinement of space between inclusions (inter-inclusion␣
↪refinement)
                          "elementsPerCircumference": 18,                           # use␣
↪18 elements per inclusion circumference for inclusion refinement
                          "elementsBetweenInclusions": 3,                           #␣
↪ensure 3 elements between close inclusions for inter-inclusion refinement
                          "inclusionRefinementWidth": 3,                            # use␣
↪a relative (to inclusion radius) refinement width of 3 for inclusion refinement
                          "transitionElements": "auto",                             #␣
↪automatically calculate number of transitioning elements (elements in which tanh␣
↪function jumps from h_min to h_max) for inter-inclusion refinement
                          "aspectRatio": 1.5                                        #␣
↪aspect ratio for inter-inclusion refinement: ratio of refinement in inclusion␣
↪distance and perpendicular directions
    }
}
testCell.createMesh(**meshingParameters)
```

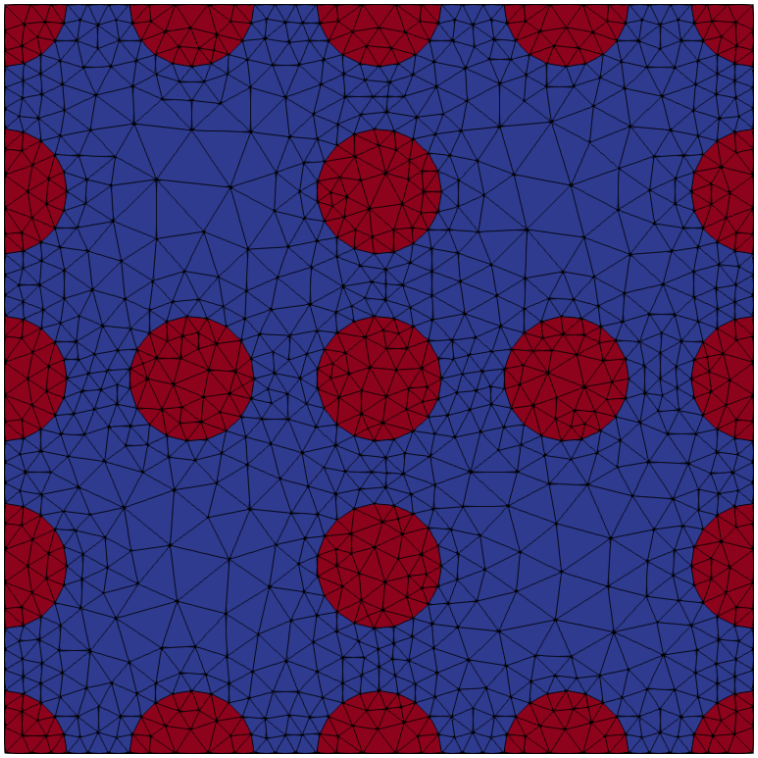
(continues on next page)

(continued from previous page)

```
# Save resulting mesh to file
# The mesh is generated and can be saved to a file. To this end, only the file
# name - possibly containing a directory and the extension of the wanted mesh
# format - has to be passed. The package supports all mesh file formats that are
# supported by meshio. If no filename is passed, meshes are stored to the current
# directory using the unique model name and the default mesh file format (.msh)
testCell.saveMesh("faceCenteredCubicCell3DCylinder.xdmf")


# Show resulting mesh
# To check the generated mesh, the result can also be visualized using built-in
# methods.
testCell.visualizeMesh()


# Close Gmsh model
# For a proper closing of the Gmsh-Python-API, the API has to be finalized. This
# can be achieved by calling the close() method of the model
testCell.close()
```

Result

If the mesh generation is successful, the result should look like this:



1.3.7 Hexagonal unit cell with spherical inclusions

This example shows the generation of a unit cell with a hexagonal distribution of spherical inclusions. The basic procedures of the model and mesh generation are pointed out and the resulting mesh is visualized. For the example, only the standard configuration is used. However, in order to show all available options - user configurations are passed as dictionaries to the individual classes and methods - the dictionaries containing the default values are passed. This means that, if they were not passed, the resulting mesh would be the same.

Code

```
# Loading of the HexagonalUnitCell class
# Before the model and mesh generation can start, the required class has to be
# loaded. In this case it is the class HexagonalCell
from gmshModel.Model import HexagonalCell

# Initialization of the unit cell
# In order to generate a mesh for unit cells with a hexagonal distribution of
# spherical inclusions, relevant data have to be passed for the initialization of
# a new object instance. For unit cells of the type under consideration, the
# following parameters are possible:
#
# radius: float (mandatory)
#   radius of the inclusions within the unit cell
#
# distance: float (defining either distance or size is mandatory)
#   distance of the inclusions within the unit cell
#   -> if the distance is given, the cells size is calculated automatically
#
# size: list/array (defining either distance or size is mandatory)
#   array defining the size of the RVE in the individual directions
#   -> size=[L_x, L_y, (L_z)]
#   -> if the size is given, the inclusion distances are calculated automatically
#      (this allows more flexibility and unit cells with inclusion distributions
#       that are similar to the physical unit cell under consideration)
#
# numberCells: list/array (optional)
#   array defining the number of cells in the 3 spatial axis directions
#   -> numberCells=[n_x, n_y, n_z]
#
# inclusionType: string (mandatory)
#   string defining the type of inclusions within the RVE
#
# origin: list/array (optional)
#   array defining the origin of the RVE
#   -> origin=[O_x, O_y, (O_z)]
#
# periodicityFlags: list/array (optional)
#   array with flags (0/1) whether the current coordinate direction has to be
#   treated as periodic
#   periodicityFlags=[0/1, 0/1, 0/1]
#
# domainGroup: string (optional)
#   string defining which group the geometric objects defining the domain belong
#   to (to reference this group within boolean operations)
#
# inclusionGroup: string (optional)
```

(continues on next page)

(continued from previous page)

```
#   string defining which group the geometric objects defining the inclusions
#   belong to (to reference this group within boolean operations)
#
# gmshConfigChanges: dict (optional)
#   dictionary for user updates of the default Gmsh configuration
#
initParameters={                                                                #␣
↪save all possible parameters in one dict to facilitate the method call
    "numberCells": [1,1,1],                                                     #␣
↪generate 1 unit cell in every spatial direction
    "radius": 2.5,                                                              # set␣
↪the inclusion radius to 2.5
    "inclusionType": "Sphere",                                                  #␣
↪define inclusionType as "Sphere"
    "size": [6, 6*3**(0.5), 6*(8/3)**(0.5)],                                    # set␣
↪cell size instead of distance
    "origin": [0, 0, 0],                                                        # set␣
↪cell origin to [0,0,0]
    "periodicityFlags": [1, 1, 1],                                              #␣
↪define all axis directions as periodic
    "domainGroup": "domain",                                                    # use
↪"domain" as name for the domainGroup
    "inclusionGroup": "inclusions",                                             # use
↪"inclusions" as name for the inclusionGroup
    "gmshConfigChanges": {"General.Terminal": 0,                                #␣
↪deactivate console output by default (only activated for mesh generation)
                          "Mesh.CharacteristicLengthExtendFromBoundary": 0,     # do␣
↪not calculate mesh sizes from the boundary by default (since mesh sizes are␣
↪specified by fields)
    }
}
testCell=HexagonalCell(**initParameters)


# Gmsh model generation
# After all parameters for the unit cell are set, the Gmsh model can be generated.
# This process involves the definition of geometric objects, their combination
# to more complex shapes using boolean operations and the definition of physical
# groups, i.e. groups of elements that belong to the same material or part of
# the boundary. For unit cells no additional options are required for the
# inclusion placement. To this end, the command is simply:
#
testCell.createGmshModel()

# Gmsh mesh creation
# After the model has been created using the Gmsh-Python-API, the meshing
# can be performed. To this end, refinement fields defining the mesh sizes
# within the model have to be calculated and added to the Gmsh model. Once, the
# mesh sizes are specified,the mesh can be generated. Available parameters are:
#
# threads: int
#   number of threads to use for the meshing procedure
# refinementOptions: dict (optional)
#   dictionary containing user updates for the refinement field calculation
#
meshingParameters={                                                             #␣
↪save all possible parameters in one dict to facilitate the method call
```

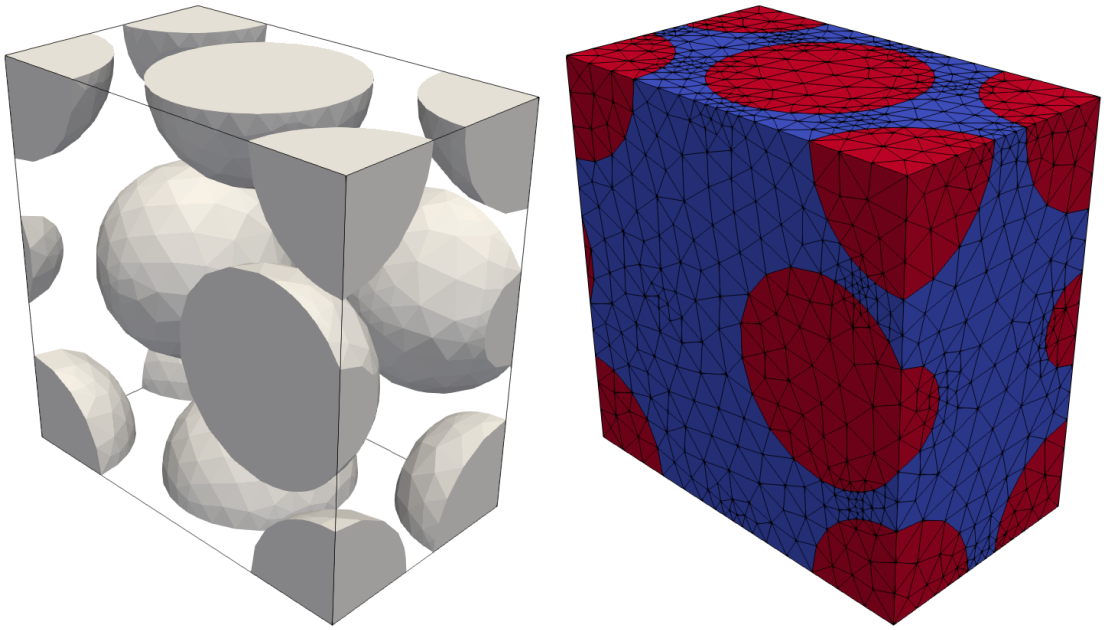
(continues on next page)

(continued from previous page)

```
    "threads": None,                                                    # do␣
↪not activate parallel meshing by default
    "refinementOptions": {"maxMeshSize": "auto",                        #␣
↪automatically calculate maximum mesh size with built-in method
                          "inclusionRefinement": True,                  #␣
↪flag to indicate active refinement of inclusions
                          "interInclusionRefinement": True,             #␣
↪flag to indicate active refinement of space between inclusions (inter-inclusion␣
↪refinement)
                          "elementsPerCircumference": 18,               # use␣
↪18 elements per inclusion circumference for inclusion refinement
                          "elementsBetweenInclusions": 3,               #␣
↪ensure 3 elements between close inclusions for inter-inclusion refinement
                          "inclusionRefinementWidth": 3,                # use␣
↪a relative (to inclusion radius) refinement width of 3 for inclusion refinement
                          "transitionElements": "auto",                 #␣
↪automatically calculate number of transitioning elements (elements in which tanh␣
↪function jumps from h_min to h_max) for inter-inclusion refinement
                          "aspectRatio": 1.5                            #␣
↪aspect ratio for inter-inclusion refinement: ratio of refinement in inclusion␣
↪distance and perpendicular directions
    }
}
testCell.createMesh(**meshingParameters)


# Save resulting mesh to file
# The mesh is generated and can be saved to a file. To this end, only the file
# name - possibly containing a directory and the extension of the wanted mesh
# format - has to be passed. The package supports all mesh file formats that are
# supported by meshio. If no filename is passed, meshes are stored to the current
# directory using the unique model name and the default mesh file format (.msh)
testCell.saveMesh("hexagonalCell3DSphere.feap")


# Show resulting mesh
# To check the generated mesh, the result can also be visualized using built-in
# methods.
testCell.visualizeMesh()


# Close Gmsh model
# For a proper closing of the Gmsh-Python-API, the API has to be finalized. This
# can be achieved by calling the close() method of the model
testCell.close()
```

Result

If the mesh generation is successful, the result should look like this:



The left image shows that the distances between the individual inclusions are equal. The right image shows the corresponding mesh with a slight refinement between close inclusions.

1.3.8 Helical chain with circular inclusions

This example shows the mesh generation for a unit cell of a helical chain with circular inclusions. The basic procedures of the model and mesh generation are pointed out and the resulting mesh is visualized. For the example, only the standard configuration is used. However, in order to show all available options - user configurations are passed as dictionaries to the individual classes and methods - the dictionaries containing the default values are passed. This means that, if they were not passed, the resulting mesh would be the same.

Code

```
# load numpy
import numpy as np

# Loading of the HelicalChain class
# Before the model and mesh generation can start, the required class has to be
# loaded. In this case it is the class HelicalChain
from gmshModel.Model import HelicalChain

# Initialization of the unit cell
# In order to generate a mesh for unit cells of a helical chain with circular
# inclusions, relevant data have to be passed for the initialization of
# a new object instance. For unit cells of the type under consideration, the
# following parameters are possible:
#
# inclusionRadius: float (mandatory)
#   radius of the inclusions within the unit cell
#
# chainRadius: float (mandatory)
#   radius of the helical chain
```

(continues on next page)

(continued from previous page)

```
#
# theta: float (mandatory)
#   angle (radian) between neighboring inclusions of the helical chain
#
# size: list/array (mandatory)
#   array defining the size of the RVE in the individual directions
#   -> size=[L_x, L_y, (L_z)]
#
# numberCells: list/array (optional)
#   array defining the number of cells in the 3 spatial axis directions
#   -> for two-dimensional problems, n_z is automatically set to 1
#   -> numberCells=[n_x, n_y, n_z]
#
# inclusionType: string (mandatory)
#   string defining the type of inclusions within the RVE
#   -> currently, only circular (2D) and spherical (3D) inclusions are supported
#
# origin: list/array (optional)
#   array defining the origin of the RVE
#   -> origin=[O_x, O_y, (O_z)]
#
# periodicityFlags: list/array (optional)
#   array with flags (0/1) whether the current coordinate direction has to be
#   treated as periodic
#   periodicityFlags=[0/1, 0/1, 0/1]
#
# domainGroup: string (optional)
#   string defining which group the geometric objects defining the domain belong
#   to (to reference this group within boolean operations)
#
# inclusionGroup: string (optional)
#   string defining which group the geometric objects defining the inclusions
#   belong to (to reference this group within boolean operations)
#
# gmshConfigChanges: dict (optional)
#   dictionary for user updates of the default Gmsh configuration
#
initParameters={                                                                #␣
↪save all possible parameters in one dict to facilitate the method call
    "numberCells": [3,1,1],                                                     #␣
↪generate 3 unit cells in the 1-direction
    "inclusionRadius": 1,                                                       # set␣
↪the inclusion radius to 1
    "chainRadius": 0.8,                                                         # set␣
↪the chain radius to 1.2
    "inclusionType": "Circle",                                                  #␣
↪define inclusionType as "Sphere"
    "chainDirection": [0,1,0],                                                  #␣
↪generate chain in 3-direction
    "theta": np.pi,                                                             # set␣
↪angle between neighboring inclusions to 180 degrees (only plausible choice)
    "size": [12, 4, 0],                                                         # set␣
↪cell size  (resulting layer distance of 4/3)
    "origin": [0, 0, 0],                                                        # set␣
↪cell origin to [0,0,0]
    "periodicityFlags": [1, 1, 1],                                              #␣
↪define all axis directions as periodic
```

(continues on next page)

(continued from previous page)

```
    "domainGroup": "domain",                                                    # use
↪"domain" as name for the domainGroup
    "inclusionGroup": "inclusions",                                             # use
↪"inclusions" as name for the inclusionGroup
    "gmshConfigChanges": {"General.Terminal": 0,                                #␣
↪deactivate console output by default (only activated for mesh generation)
                          "Mesh.CharacteristicLengthExtendFromBoundary": 0,     # do␣
↪not calculate mesh sizes from the boundary by default (since mesh sizes are␣
↪specified by fields)
    }
}
testChain=HelicalChain(**initParameters)


# Gmsh model generation
# After all parameters for the chain are set, the Gmsh model can be generated.
# This process involves the definition of geometric objects, their combination
# to more complex shapes using boolean operations and the definition of physical
# groups, i.e. groups of elements that belong to the same material or part of
# the boundary. For helical chains, no additional options are required for the
# inclusion placement. To this end, the command is simply:
#
testChain.createGmshModel()

# Gmsh mesh creation
# After the model has been created using the Gmsh-Python-API, the meshing
# can be performed. To this end, refinement fields defining the mesh sizes
# within the model have to be calculated and added to the Gmsh model. Once, the
# mesh sizes are specified,the mesh can be generated. Available parameters are:
#
# threads: int
#   number of threads to use for the meshing procedure
# refinementOptions: dict (optional)
#   dictionary containing user updates for the refinement field calculation
#
meshingParameters={                                                             #␣
↪save all possible parameters in one dict to facilitate the method call
    "threads": None,                                                            # do␣
↪not activate parallel meshing by default
    "refinementOptions": {"maxMeshSize": "auto",                                #␣
↪automatically calculate maximum mesh size with built-in method
                          "inclusionRefinement": True,                          #␣
↪flag to indicate active refinement of inclusions
                          "interInclusionRefinement": True,                     #␣
↪flag to indicate active refinement of space between inclusions (inter-inclusion␣
↪refinement)
                          "elementsPerCircumference": 18,                       # use␣
↪18 elements per inclusion circumference for inclusion refinement
                          "elementsBetweenInclusions": 3,                       #␣
↪ensure 3 elements between close inclusions for inter-inclusion refinement
                          "inclusionRefinementWidth": 3,                        # use␣
↪a relative (to inclusion radius) refinement width of 3 for inclusion refinement
                          "transitionElements": "auto",                         #␣
↪automatically calculate number of transitioning elements (elements in which tanh␣
↪function jumps from h_min to h_max) for inter-inclusion refinement
                          "aspectRatio": 1.5                                    #␣
↪aspect ratio for inter-inclusion refinement: ratio of refinement in inclusion␣
↪distance and perpendicular directions
```

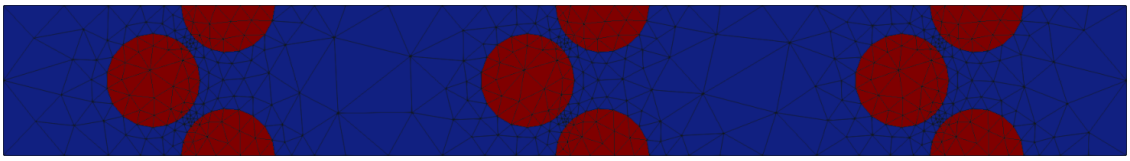
(continues on next page)

(continued from previous page)

```
    }
}
testChain.createMesh(**meshingParameters)


# Save resulting mesh to file
# The mesh is generated and can be saved to a file. To this end, only the file
# name - possibly containing a directory and the extension of the wanted mesh
# format - has to be passed. The package supports all mesh file formats that are
# supported by meshio. If no filename is passed, meshes are stored to the current
# directory using the unique model name and the default mesh file format (.msh)
testChain.saveMesh("helicalChain2DCircle.vtu")


# Show resulting mesh
# To check the generated mesh, the result can also be visualized using built-in
# methods.
testChain.visualizeMesh()


# Close Gmsh model
# For a proper closing of the Gmsh-Python-API, the API has to be finalized. This
# can be achieved by calling the close() method of the model
testChain.close()
```

Result

If the mesh generation is successful, the result should look like this:



In the image, 3 chains can be seen. The refinement between close inclusions ensures at least 3 elements between them.

1.3.9 Helical chain with spherical inclusions

This example shows the mesh generation for a unit cell of a helical chain with spherical inclusions. The basic procedures of the model and mesh generation are pointed out and the resulting mesh is visualized. For the example, only the standard configuration is used. However, in order to show all available options - user configurations are passed as dictionaries to the individual classes and methods - the dictionaries containing the default values are passed. This means that, if they were not passed, the resulting mesh would be the same.

Code

```
# load numpy
import numpy as np

# Loading of the HelicalChain class
# Before the model and mesh generation can start, the required class has to be
```

(continues on next page)

(continued from previous page)

```
# loaded. In this case it is the class HelicalChain
from gmshModel.Model import HelicalChain

# Initialization of the unit cell
# In order to generate a mesh for unit cells of a helical chain with spherical
# inclusions, relevant data have to be passed for the initialization of
# a new object instance. For unit cells of the type under consideration, the
# following parameters are possible:
#
# inclusionRadius: float (mandatory)
#   radius of the inclusions within the unit cell
#
# chainRadius: float (mandatory)
#   radius of the helical chain
#
# theta: float (mandatory)
#   angle (radian) between neighboring inclusions of the helical chain
#
# size: list/array (mandatory)
#   array defining the size of the RVE in the individual directions
#   -> size=[L_x, L_y, (L_z)]
#
# numberCells: list/array (optional)
#   array defining the number of cells in the 3 spatial axis directions
#   -> for two-dimensional problems, n_z is automatically set to 1
#   -> numberCells=[n_x, n_y, n_z]
#
# inclusionType: string (mandatory)
#   string defining the type of inclusions within the RVE
#   -> currently, only circular (2D) and spherical (3D) inclusions are supported
#
# origin: list/array (optional)
#   array defining the origin of the RVE
#   -> origin=[O_x, O_y, (O_z)]
#
# periodicityFlags: list/array (optional)
#   array with flags (0/1) whether the current coordinate direction has to be
#   treated as periodic
#   periodicityFlags=[0/1, 0/1, 0/1]
#
# domainGroup: string (optional)
#   string defining which group the geometric objects defining the domain belong
#   to (to reference this group within boolean operations)
#
# inclusionGroup: string (optional)
#   string defining which group the geometric objects defining the inclusions
#   belong to (to reference this group within boolean operations)
#
# gmshConfigChanges: dict (optional)
#   dictionary for user updates of the default Gmsh configuration
#
initParameters={                                                                #␣
↪save all possible parameters in one dict to facilitate the method call
    "numberCells": [1,1,2],                                                     #␣
↪generate 1 unit cell in axis directions perpendicular to the chain, generate 2␣
↪chains in chain direction
    "inclusionRadius": 1,                                                       # set␣
↪the inclusion radius to 1
```

(continues on next page)

(continued from previous page)

```
    "chainRadius": 1.25,                                                    # set␣
↪the chain radius to 1.25
    "inclusionType": "Sphere",                                              #␣
↪define inclusionType as "Sphere"
    "chainDirection": [0,0,1],                                              #␣
↪generate chain in 3-direction
    "theta": np.pi/3,                                                       # set␣
↪angle between neighboring inclusions to 60 degrees
    "size": [10, 10, 21],                                                   # set␣
↪cell size  (resulting layer distance of 1.75)
    "origin": [0, 0, 0],                                                    # set␣
↪cell origin to [0,0,0]
    "periodicityFlags": [1, 1, 1],                                          #␣
↪define all axis directions as periodic
    "domainGroup": "domain",                                                # use
↪"domain" as name for the domainGroup
    "inclusionGroup": "inclusions",                                         # use
↪"inclusions" as name for the inclusionGroup
    "gmshConfigChanges": {"General.Terminal": 0,                            #␣
↪deactivate console output by default (only activated for mesh generation)
                          "Mesh.CharacteristicLengthExtendFromBoundary": 0,     # do␣
↪not calculate mesh sizes from the boundary by default (since mesh sizes are␣
↪specified by fields)
    }
}
testChain=HelicalChain(**initParameters)


# Gmsh model generation
# After all parameters for the chain are set, the Gmsh model can be generated.
# This process involves the definition of geometric objects, their combination
# to more complex shapes using boolean operations and the definition of physical
# groups, i.e. groups of elements that belong to the same material or part of
# the boundary. For helical chains, no additional options are required for the
# inclusion placement. To this end, the command is simply:
#
testChain.createGmshModel()

# Gmsh mesh creation
# After the model has been created using the Gmsh-Python-API, the meshing
# can be performed. To this end, refinement fields defining the mesh sizes
# within the model have to be calculated and added to the Gmsh model. Once, the
# mesh sizes are specified,the mesh can be generated. Available parameters are:
#
# threads: int
#   number of threads to use for the meshing procedure
# refinementOptions: dict (optional)
#   dictionary containing user updates for the refinement field calculation
#
meshingParameters={                                                         #␣
↪save all possible parameters in one dict to facilitate the method call
    "threads": None,                                                        # do␣
↪not activate parallel meshing by default
    "refinementOptions": {"maxMeshSize": "auto",                            #␣
↪automatically calculate maximum mesh size with built-in method
                          "inclusionRefinement": True,                      #␣
↪flag to indicate active refinement of inclusions
```

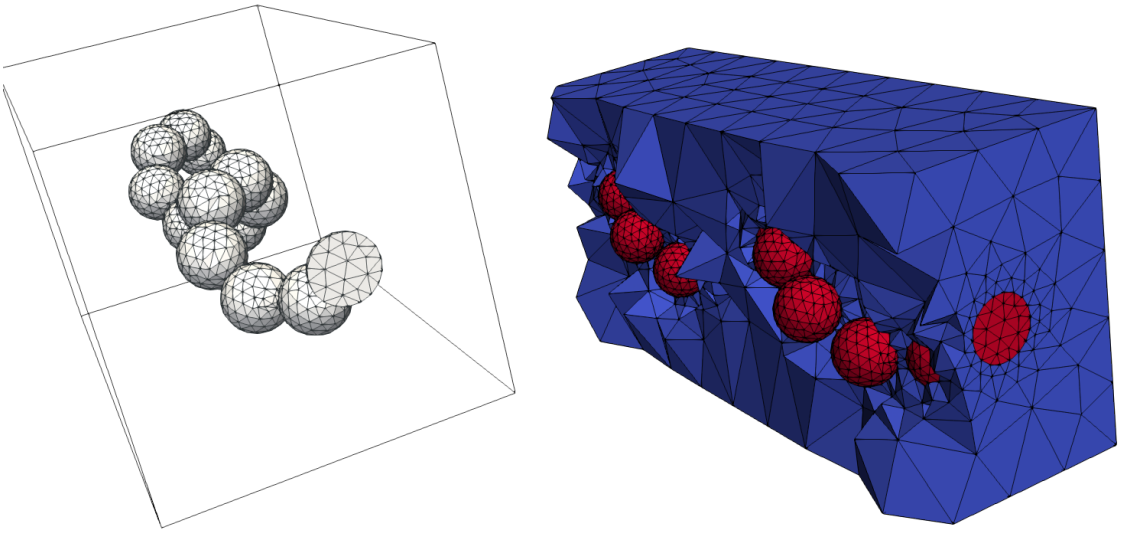
(continues on next page)

(continued from previous page)

```
                        "interInclusionRefinement": True,                    #␣
↪flag to indicate active refinement of space between inclusions (inter-inclusion␣
↪refinement)
                        "elementsPerCircumference": 18,                      # use␣
↪18 elements per inclusion circumference for inclusion refinement
                        "elementsBetweenInclusions": 3,                      #␣
↪ensure 3 elements between close inclusions for inter-inclusion refinement
                        "inclusionRefinementWidth": 3,                       # use␣
↪a relative (to inclusion radius) refinement width of 3 for inclusion refinement
                        "transitionElements": "auto",                        #␣
↪automatically calculate number of transitioning elements (elements in which tanh␣
↪function jumps from h_min to h_max) for inter-inclusion refinement
                        "aspectRatio": 1.5                                   #␣
↪aspect ratio for inter-inclusion refinement: ratio of refinement in inclusion␣
↪distance and perpendicular directions
    }
}
testChain.createMesh(**meshingParameters)


# Save resulting mesh to file
# The mesh is generated and can be saved to a file. To this end, only the file
# name - possibly containing a directory and the extension of the wanted mesh
# format - has to be passed. The package supports all mesh file formats that are
# supported by meshio. If no filename is passed, meshes are stored to the current
# directory using the unique model name and the default mesh file format (.msh)
testChain.saveMesh("helicalChain3DSphere.xdmf")


# Show resulting mesh
# To check the generated mesh, the result can also be visualized using built-in
# methods.
testChain.visualizeMesh()


# Close Gmsh model
# For a proper closing of the Gmsh-Python-API, the API has to be finalized. This
# can be achieved by calling the close() method of the model
testChain.close()
```

Result

If the mesh generation is successful, the result should look like this:



The left image shows the structure of the helical chain. In the right image, an extraction of the mesh is shown to illustrate the mesh quality resulting from the default refinement options.

CHAPTER 2

Documentation

Here, you can find out how gmshModel works, which classes and methods are involved and how you can use them to generate your own model:

- *Geometry* gives information on available geometric objects
- *Model* explains all available models
- *Visualization* gives information on the visualization capabilities of gmshModel
- *MeshExport* comments on additional mesh output formats (extending meshio)

2.1 API Reference

The core functionality of GmshModel is the mesh generation for complex models using Gmsh and the Gmsh-Python-API. The creation of such complex models, often requires methods for the geometry generation. To this end, basic geometric objects and helper methods for, e.g., distance calculations are provided within the `Geometry` module of GmshModel: using boolean operations for groups of basic geometric objects, complex models can be defined step by step. An extension of the modules will help to broaden the range of available models.

After the geometry is defined, it has to be transferred into a Gmsh model: all geometric objects are translated to their Gmsh representations, boolean operations are performed and physical groups are added to the model. Within the Model module of `Model`, predefined models can be found. Since, so far, the focus of GmshModel was on mesh models for representative volume elements with multiple, randomly placed inclusions, especially those models are already defined in GmshModel. However, since the GenericModel defines all required methods for the model generation, the basic tasks for the development of a new model are the definitions of required geometric objects and their arrangement within the model, of boolean operations and physical groups to be performed/added in Gmsh and of refinement information for an auomated mesh size computation.

Finally, basic GUIs for the geometry and mesh visualization are defined within the `Visualization` module while an extension of the mesh conversion capabilities of meshio for simulations using FEAP is defined within the `MeshExport` module.

2.1.1 Model

The Model module provides defined model class definitions for GmshModel: starting from the `GenericModel` class, where all basic attributes and method for every GmshModel are defined, more specialized models are defined as children of GenericModel

GenericModel

The GenericModel is the base class for other, more specific classes which aim to mesh models using the Gmsh-Python-API. In addition to the methods defined within the Gmsh-Python-API, this class provides methods for all basic steps of a model generation using Gmsh: some of these methods are only placeholders here and - if required - have to be specified/overwritten for the more specialized models.

Class Definition

GenericRVE

The GenericRVE provides a class definition for an RVE generation using Python and Gmsh. The class inherits from the GenericModel class and extends it in order order to handle the problems that are connected with the generation of models with periodicity constraints.

Currently, the class is restricted to RVEs with rectangular (2D)/ box-shaped (3D) domains (explicitly assumed within the setupPeriodicity() method).

Class Definition

InclusionRVE

The InclusionRVE provides a class definition for a generation of RVEs with inclusions using Python and Gmsh. The class inherits from the GenericRVE class and extends it in order to handle distance and refinement calculations

Currently, the class is restricted to RVEs with rectangular (2D)/ box-shaped (3D) domains (explicitly assumed within the setupPeriodicity() method) which comprise inclusions that are all of the same type (explicitly assumed by using one inclusionInformation array and one inclusionAxis variable).

Class Definition

RandomInclusionRVE

The RandomInclusionRVE class provides a class definition for a generation of RVEs with randomly placed inclusions. The class inherits from the InclusionRVE class and extends it in order to specify the remaining placeholder methods of the GenericModel. Methods to create the geometry, define refinement information and additional information for required boolean operations and physical groups are part of the class.

Class Definition

GenericUnitCell

The GenericUnitCell class provides required information for inclusion-based unit cells. It inherits from the InclusionRVE class and extends its attributes and methods to handle the boolean operations and the definition of physical groups.

All unit cell allow to create "real" unit cells by passing the inclusion distance to the classes initialization method. If the cells size is specified instead, the distance is calculated automatically: this allows for unit cells with an inclusion distribution that is close to physical unit cells but gives more flexibility in their generation.

Class Definition

SimpleCubicCell

The SimpleCubicCell provides a class definition for a generation of unit cells with a simple cubic distribution of the inclusions. The class inherits from the GenericUnitCell class and extends it in order to specify the remaining placeholder methods of the GenericModel. Especially, methods to determine the cells size and place the inclusions are provided.

Class Definition

BodyCenteredCubicCell

The BodyCenteredCubicCell provides a class definition for a generation of unit cells with a body-centered cubic distribution of the inclusions. The class inherits from the GenericUnitCell class and extends it in order to specify the remaining placeholder methods of the GenericModel. Especially, methods to determine the cells size and place the inclusions are provided.

Class Definition

FaceCenteredCubicCell

The FaceCenteredCubicCell provides a class definition for a generation of unit cells with a face-centered cubic distribution of the inclusions. The class inherits from the GenericUnitCell class and extends it in order to specify the remaining placeholder methods of the GenericModel. Especially, methods to determine the cells size and place the inclusions are provided.

Class Definition

HexagonalCell

The HexagonalCell provides a class definition for a generation of unit cells with a hexagonal cubic distribution of the inclusions. The class inherits from the GenericUnitCell class and extends it in order to specify the remaining placeholder methods of the GenericModel. Especially, methods to determine the cells size and place the inclusions are provided.

Class Definition

HelicalChain

The HelicalChain class provides a class definition for a generation of unit cells with inclusions distributed in a helical chain. The class inherits from the InclusionRVE class and extends it in order to specify the remaining placeholder methods of the GenericModel.

Currently, the class is restricted to circular and spherical inclusions which are arranged in a helical chain that is parallel to one of the coordinate axes.

Class Definition

2.1.2 Geometry

The Geometry model provides basic geometric objects and helper methods for the model geometry generation. An extension of geometric objects will help to extend the class of geometries that can be generated with GmshModel.

GeometricObjects

Within GeometricObjects classes for different geometric objects used in the GmshModel are defined. The geometric objects are used within the geometry generation to create Gmsh model from basic geometry entities.

Class Definitions

GeometricObject

Box

Rectangle

Sphere

Cylinder

Circle

DistanceCalculations

This file provides methods for distance calculations

Methods

2.1.3 Visualization

To see, if the generated mesh matches the own requirements, a basic visualization tool using pyvista has been defined in the `MeshVisualization` class. If a mesh generation does not work and the pythonocc package is available, the geometry of the Gmsh model can also be visualized using the `GeometryVisualization` class.

GeometryVisualization

The GeometryVisualization provides the class definition for the geometry visualization of the model. It is based on the pythonocc library and provides simple methods for the visualization.

Class Definition

MeshVisualization

The MeshVisualization provides the class definition for the mesh visualization of a Gmsh model. It is based on the pyvista and vtk libraries und implements additional features for the mesh visualization.

Class Definition

2.1.4 MeshExport

This module provides additional methods for mesh export formats that cannot be handled using meshio. At the moment, only a special export method for FEAP is implemented - other export formats can be added here, if required.

MeshExport

The MeshExport module provides a simple mesh export feature for FEAP by directly using the information available from the model without involving meshio.

Module Functions

CHAPTER 3

Index

- genindex