RSG Dialog Builder in Abaqus/CAE – Generation of screws

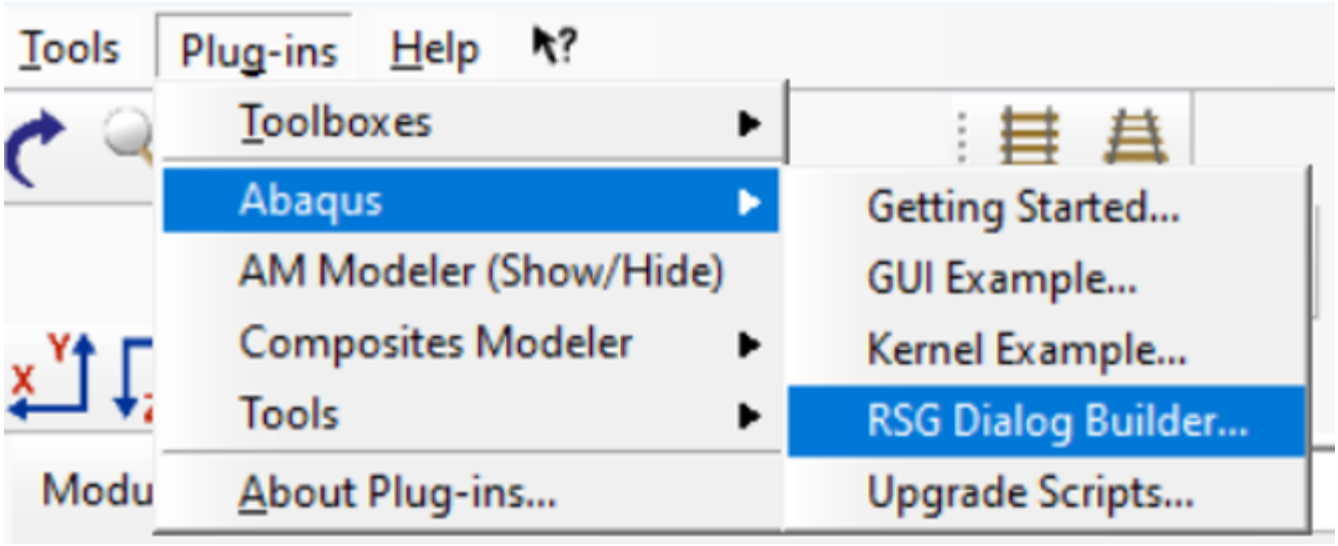
Simulation · https://plm.systemworkx.de/en/docs/rsg-dialog-builder-in-abaqus-cae-generation-of-screws/

Introduction

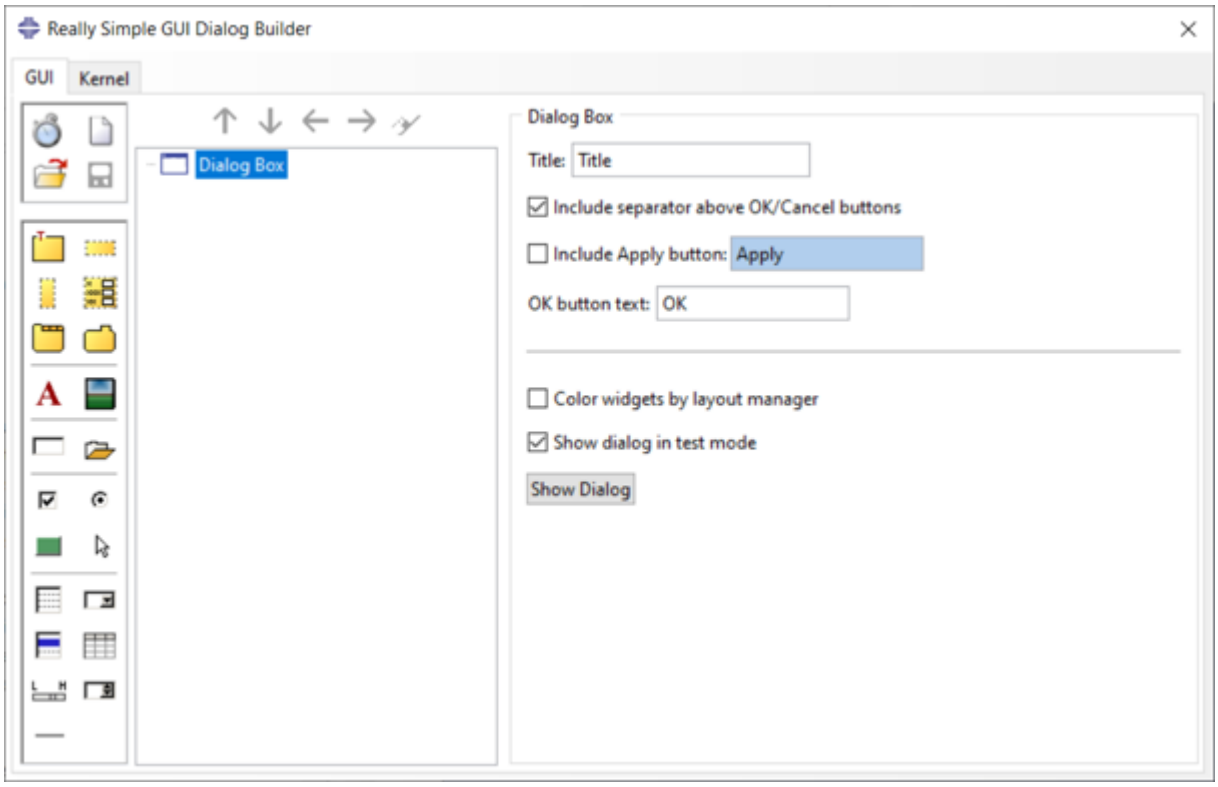
The RSG Dialog Builder is a simple tool for creating plug-ins in Abaqus/CAE. This tool is delivered by default with every version. Here, the RSG Builder consists of two parts, the GUI and the executing script in the kernel of Abaqus/CAE. Such plugins can be used to automate the work process. We have already used RSG Builder in Abaqus/CAE: QuickStart Plugin with RSG Dialog Builder In this article we want to use RSG Builder to create standard components, in this case screws. Here we describe in detail the creation of the GUI and a script to run it. Screws are used to connect components together via contact and pretension. Especially for large assemblies, in which many components are connected by different bolt connections, such a plugin offers a valuable contribution to the efficient modeling of different bolt sizes. Even if this example seems elaborate at first glance: The creation of standard components is feasible even for users with little or no experience in scripting. The improvements to the script described in section 4 can also be implemented with little knowledge of Python.

Procedure

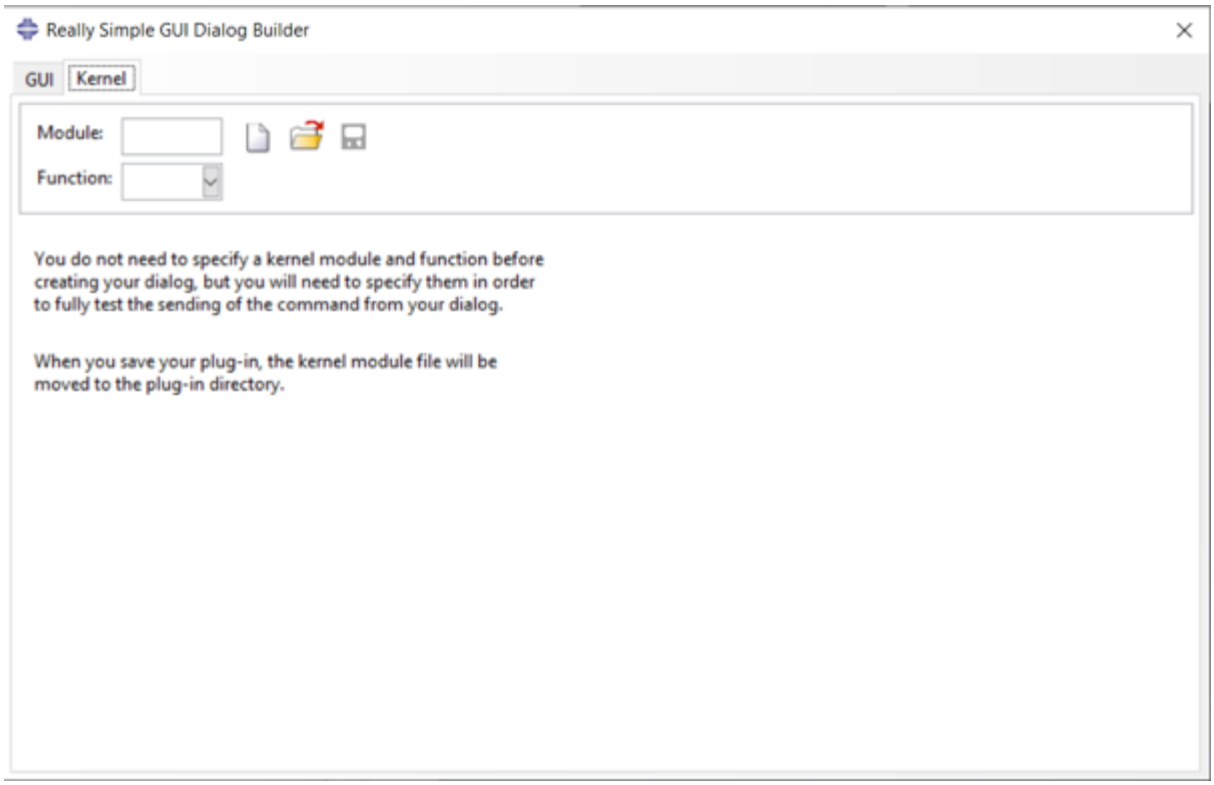
The RSG Dialog Builder is located in the menu item Abaqus Plugins, see the following image. RSG stands for "**Really Simple GUI**".



The RSG Dialog Builder is divided into 2 windows:

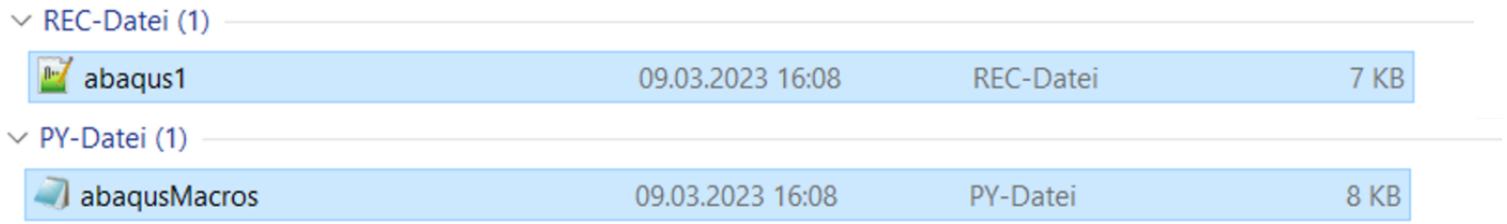


The GUI window: The necessary parameters (variables) for the kernel module are defined here. There is a possibility to create pull-down menus, buttons, tables, etc. By activating "Show dialog in test mode", you can see in real time how your GUI has been set up.



The kernel window: This is where the Python code is required for execution. As explained below, this Python code can be generated using the Macro Manager and, after a little revision, inserted.

The procedure used here to create the Python script via the Macro Manager can also be used for other projects. Step 1: Starting the macro recorder Step 2: Creating the screw geometry, partitioning the screw and defining the material and section Step 3: Defining the element type and meshing the screw with a global element size Step 4: Closing the macro recorder The macro recorder creates two files with the names ABAQUS1.rec and AbaqusMacros.py. The content is accumulated, so it is recommended to delete an already existing file of this name. The file only contains the Python code required for the script.



Generation Of A Standard Component Using The Example Of A Screw

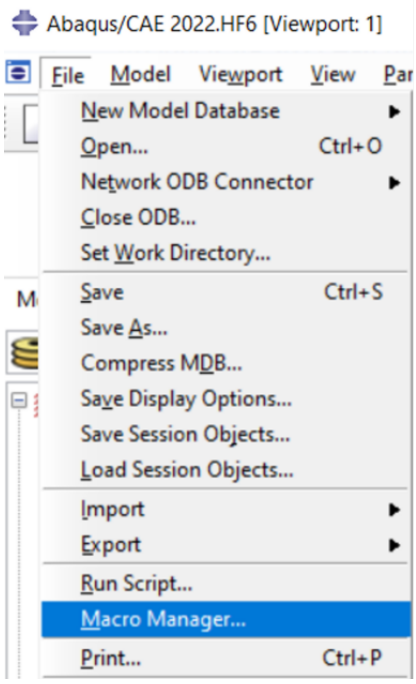
Before creating the geometry, it is advisable to make a sketch of the screw and its parameters, see image below. This sketch can be further used in the PlugIn below. Likewise, a list of all parameters used should

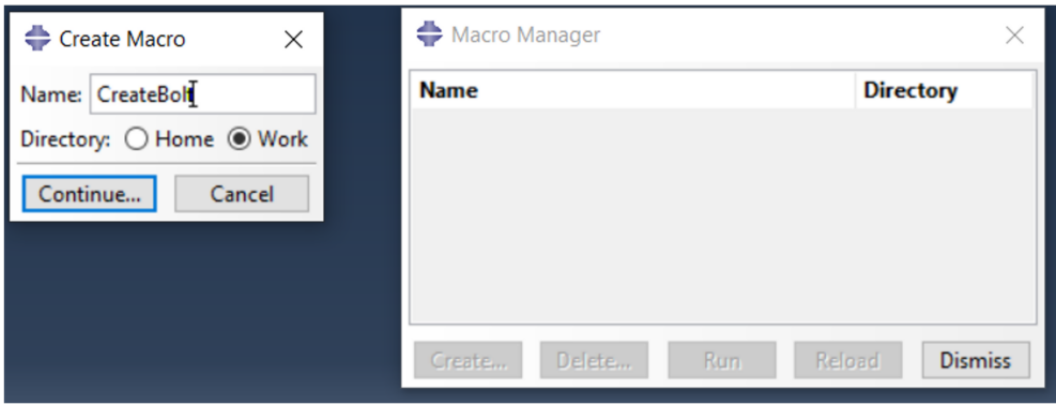
be created in advance. In this example they are:

DK = head diameter KH = head height LS = shank length LG = thread length DS = shank diameter Pname = name of the part Matname = name of the material Secname = name of the section Module = modulus of elasticity Dens = density of the material NUE= transverse contraction number msize = element size (% of DS) Standard values: DK~1.5*DS ; KH~DS

Creating The Script

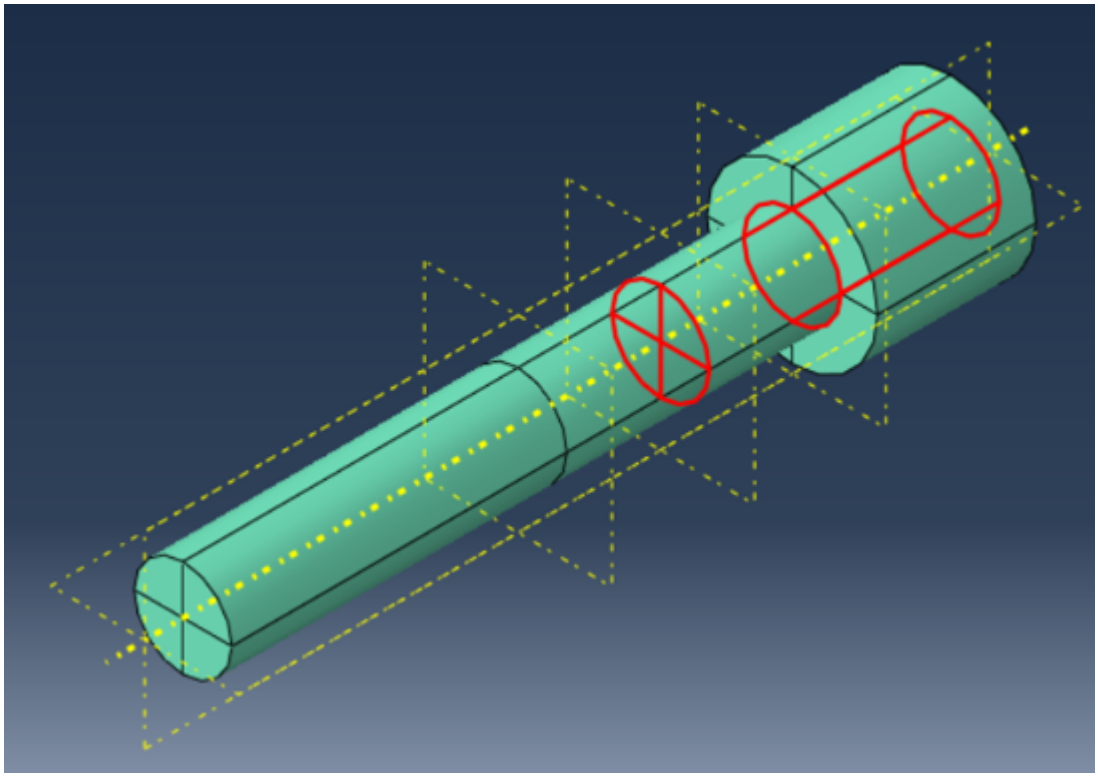
Step 1: Starting the macro recorder As shown in the picture below, start the Macro Manager via the Files tab. A new macro is generated there. It is recommended that you first save a new macro only in the working directory (Work).



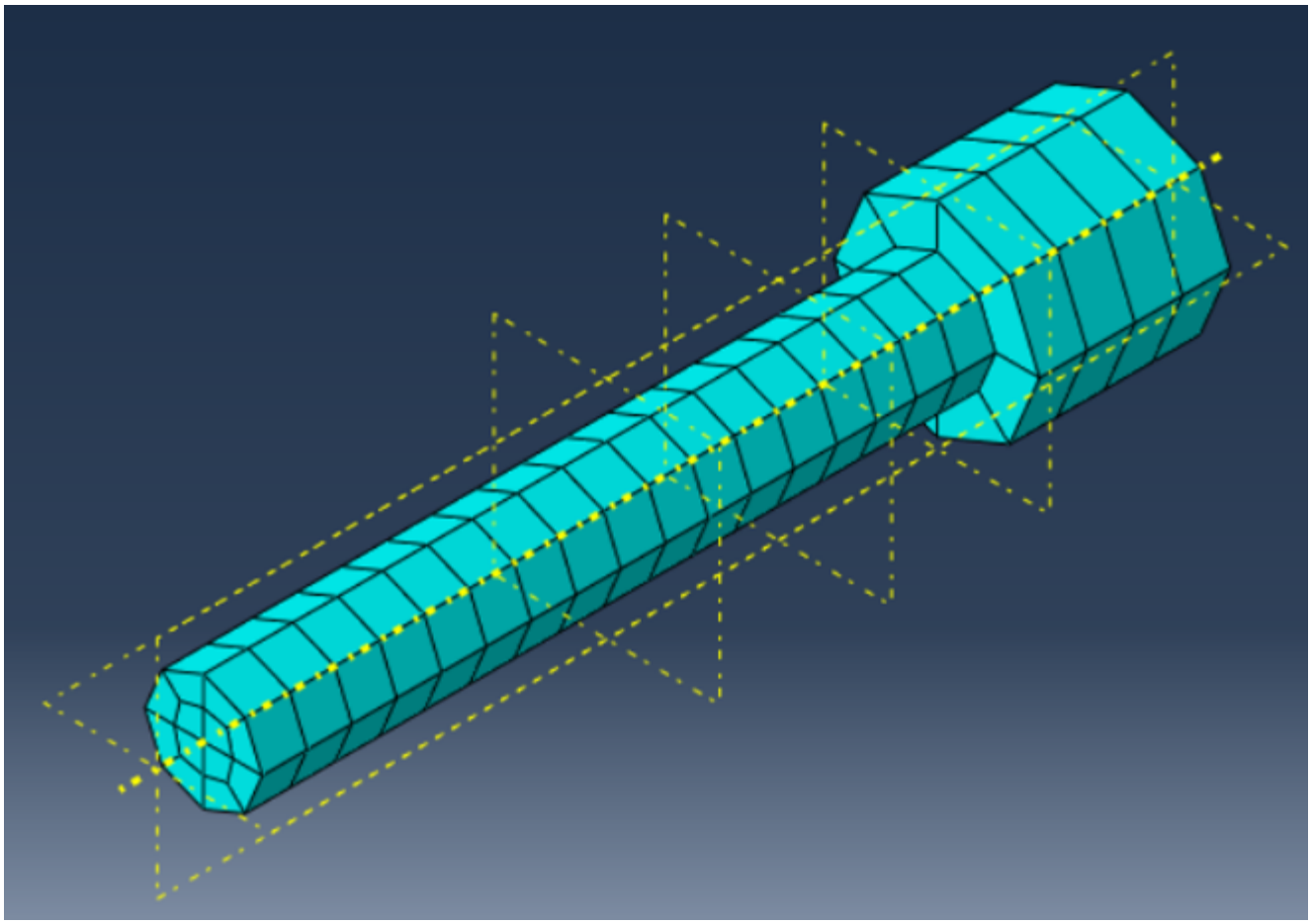


Step 2: Creating the screw geometry and partitioning the screw When creating the geometry and datum planes, it is useful and helpful to use unique values, e.g. 7.777 for the parameter KH (head height of the screw). This way the value can be found quickly in the macro file and is then replaced by the corresponding parameter (see further down in the text). The part is called Bolt_Script. This name is freely selectable. Then a material and a section are defined. The section is assigned to the component.

The partitioned component with the assigned section, the 5 datum levels and the "Bolt_Axis" axis, which is not absolutely necessary but is helpful for further scripts, is shown on the right. Further partitions are marked in red and will be explained in more detail later.



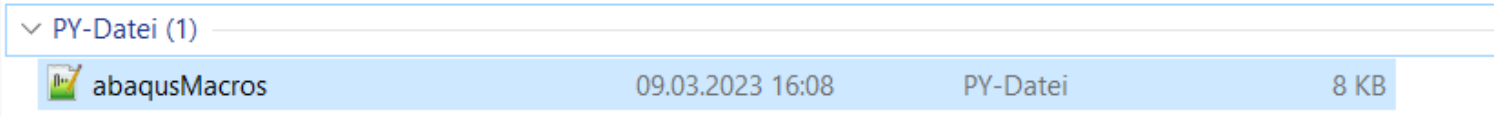
Step 3 : Definition of the element type (here C3D8) and meshing of the screw with a global element size. The element size is selected as 3,333. This value can therefore be easily identified in the script and replaced by a parameter.



Step 4 : Close the macro recorder.

Editing The Script

Now the file abaqusMacros.py can be opened with the editor. Here the editor Notepad++ ([5]) is used for editing and processing.



The macro is saved as a function ('def') in a Python script. The function does not yet contain any parameters and must now be filled. The import definitions are taken over, even if not all modules are needed. The name of the function may be changed.

```
# -*- coding: mbcs -*-
# Do not delete the following import lines
from abaqus import *
from abaqusConstants import *
import   __main__

def Bolt():
    import section
    import regionToolset
    import displayGroupMdbToolset as dgm
    import part
    import material
    import assembly
    import optimization
    import step
    import interaction
    import load
    import mesh
    import job
    import sketch
    import visualization
    import xyPlot
    import displayGroupOdbToolset as dgo
    import connectorBehavior
```

The function is now extended by the required parameters by entering the parameters as transfer values in the brackets after the function name. This is where the previously created list proves helpful. The order is arbitrary, but the list must be complete.

DK = head diameter KH = head height LS = shank length LG = thread length DS = shank diameter Pname = name of the part Matname = name of the material Secname = name of the section Module = modulus of elasticity Dens = density of the material NUE= transverse contraction number msize = element size (% of DS)

```
# -*- coding: mbcs -*-
# Do not delete the following import lines
from abaqus import *
from abaqusConstants import *
import __main__
VN = session.currentViewportName
MN = session.sessionState[VN]['modelName']
def CreateBolt(Pname,DS,LS,LG,DK,KH,Matname,Secname,Module,NUE,Dens,msize):
    import section
    import regionToolset
    import displayGroupMdbToolset as dgm
    import part
    import material
    import assembly
    import optimization
    import step
    import interaction
    import load
    import mesh
    import job
    import sketch
    import visualization
    import xyPlot
    import displayGroupOdbToolset as dgo
    import connectorBehavior
```

The placeholders VN and MN are generated. In the following, all entries of 'Viewport: 1' are replaced by VN and all entries of 'Model-1' are replaced by MN in the script. It is assumed that the part was created in Viewport 1 and the model was not renamed. Otherwise, the designations must be adapted accordingly.

```
# -*- coding: mbcs -*-
# Do not delete the following import lines
from abaqus import *
from abaqusConstants import *
import   main
VN = session.currentViewportName
MN = session.sessionState[VN]['modelName']
def CreateBolt(Pname,DS,LS,LG,DK,KH,Matname,Secname,Module,NUE,Dens,msize):
    import section
    import regionToolset
    import displayGroupMdbToolset as dgm
    import part
    import material
    import assembly
    import optimization
    import step
    import interaction
    import load
    import mesh
    import job
    import sketch
    import visualization
    import xyPlot
    import displayGroupOdbToolset as dgo
    import connectorBehavior
```

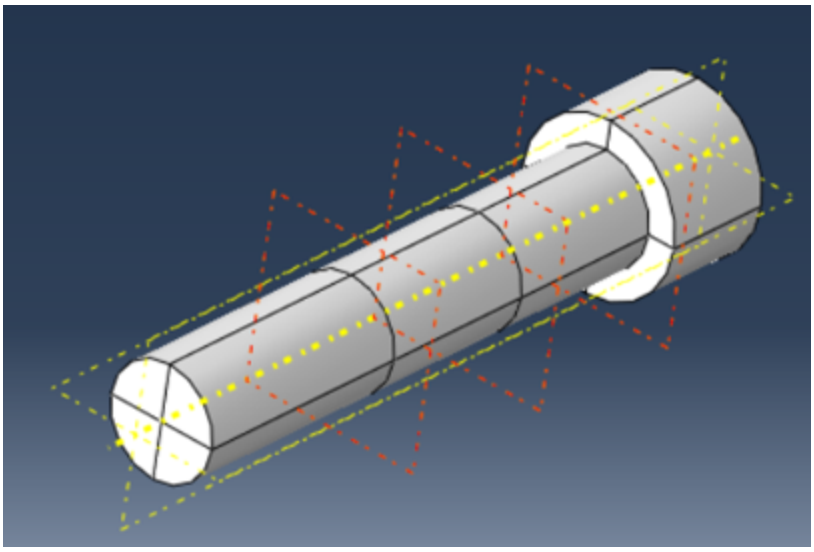
All dimensions and names used when creating the part are now replaced by their parameters in the script file. This shows that it previously made sense to assign recognizable values to the dimensions.

```
s.RadialDimension(curve=g[2], textPoint=(0.0, 0.0), radius=7.777)
```



```
s1.RadialDimension(curve=g[2], textPoint=(0.0, 0.0), radius=0.5*DK)
```

All 5 dimensions are replaced by the corresponding parameters (KH,DK,LS,LG,DS). The same applies to the date planes that were created with an offset from the xy plane (see image).



Finally, the remaining parameters used in the script are assigned, here the part Bolt_Skript is replaced by the parameter Pname (the quotation marks of Bolt_Skript must not be copied). Once all parameters in the script have been assigned, it is saved with the extension .py (here: Bolt_without_Nut.py) and can now be used in the RSG-Builder. It is recommended to use a powerful editor up to this point, as the editor in the kernel module only offers limited functions.

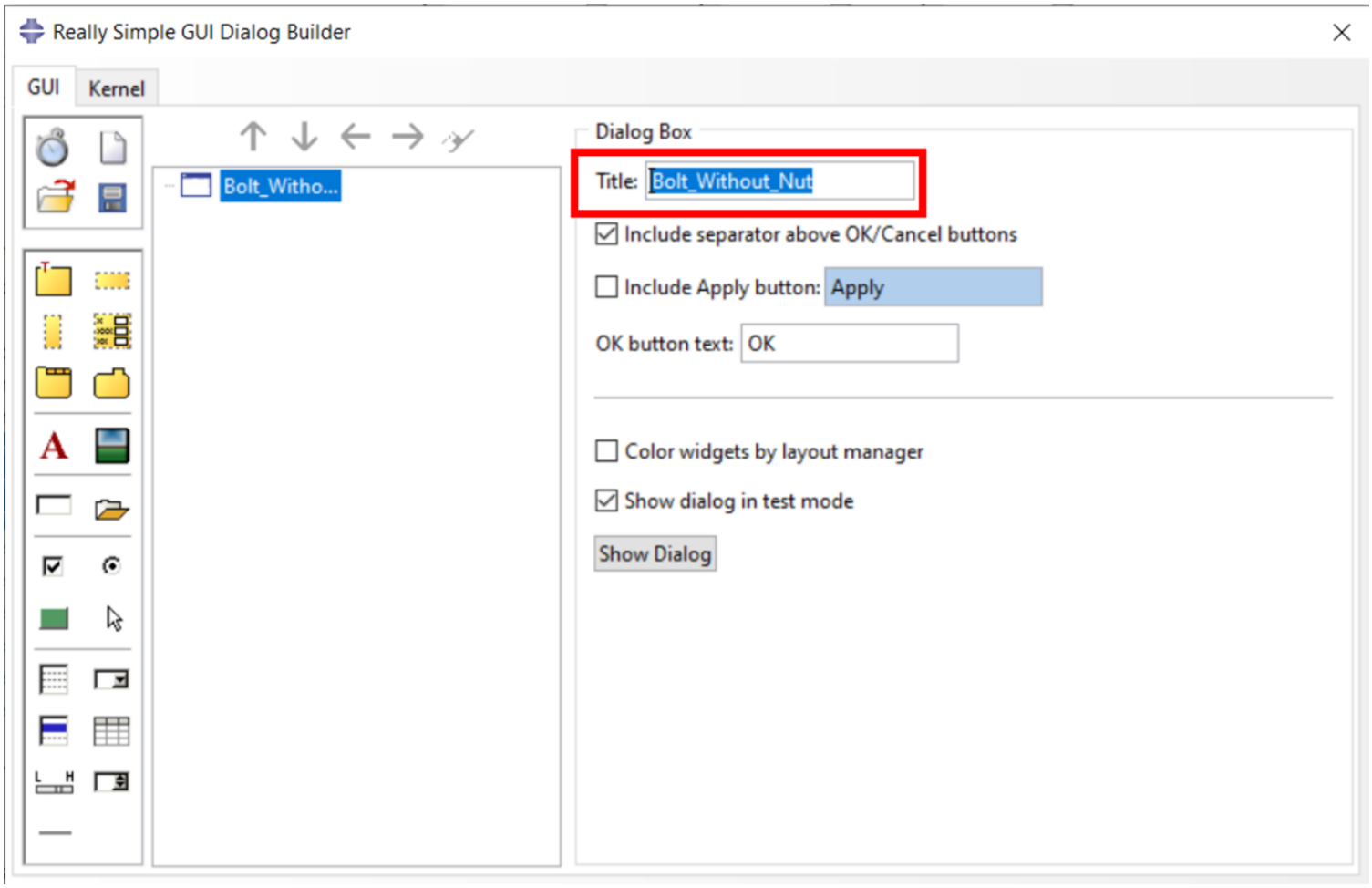
```
p = mdb.models['Model-1'].Part(name='Bolt_Skript', dimensionality=THREE_D,
    type=DEFORMABLE_BODY)
p = mdb.models['Model-1'].parts['Bolt_Skript']
p.BaseSolidExtrude(sketch=s, depth=14.444)
s.unsetPrimaryObject()
p = mdb.models['Model-1'].parts['Bolt_Skript']
session.viewports['Viewport: 1'].setValues(displayedObject=p)
del mdb.models['Model-1'].sketches['__profile__']
p = mdb.models['Model-1'].parts['Bolt_Skript']
```



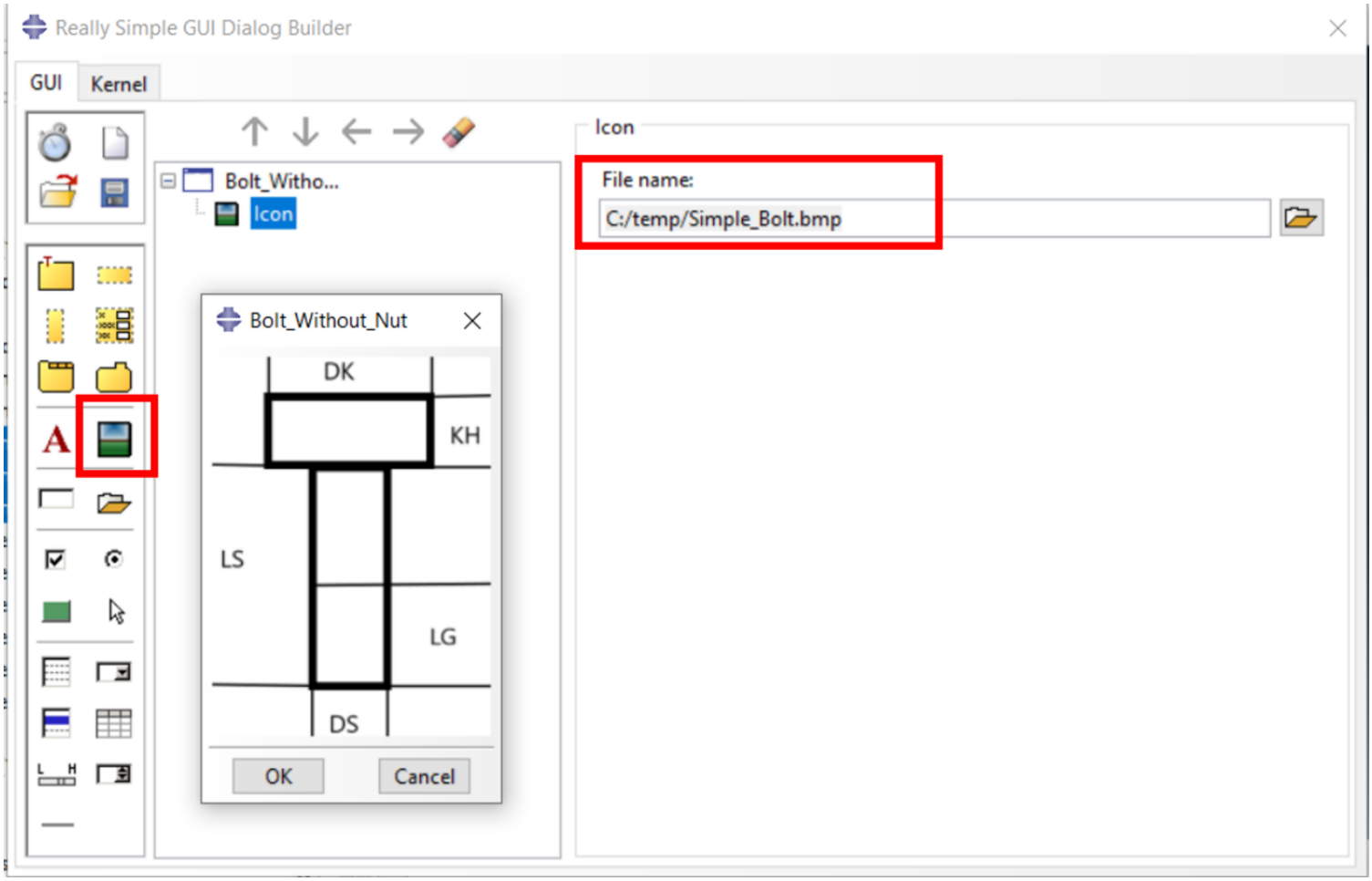
```
p = mdb.models[MN].Part(name=Pname, dimensionality=THREE_D,
    type=DEFORMABLE_BODY)
p = mdb.models[MN].parts[Pname]
p.BaseSolidExtrude(sketch=s1, depth=KH)
s1.unsetPrimaryObject()
p = mdb.models[MN].parts[Pname]
session.viewports[VN].setValues(displayedObject=p)
del mdb.models[MN].sketches['__profile__']
session.viewports[VN].partDisplay.setValues(renderStyle=WIREFRAME)
p = mdb.models[MN].parts[Pname]
```

Use Of The RSG Builder

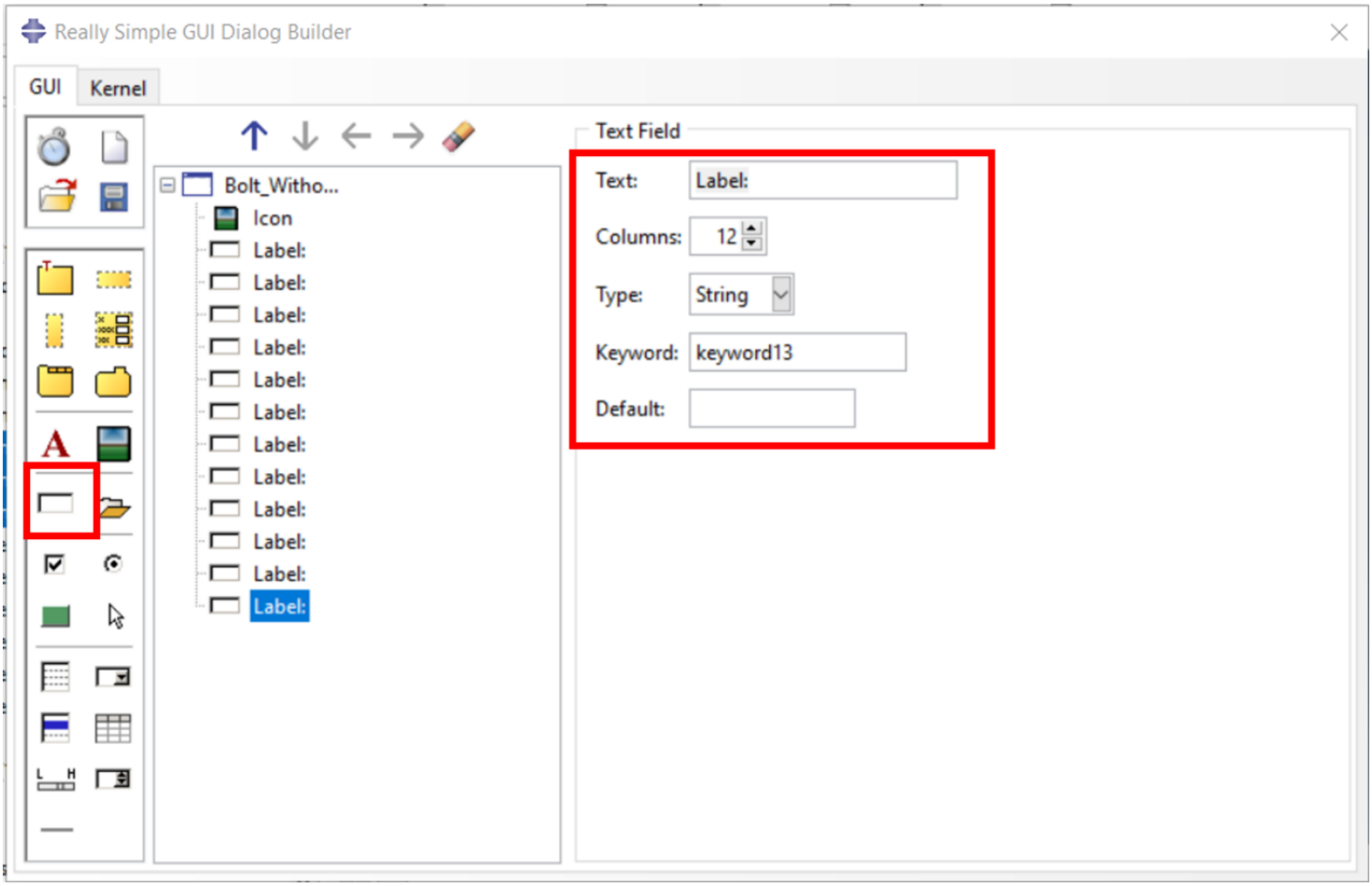
Now open the RSG Dialog Builder and assign a title first



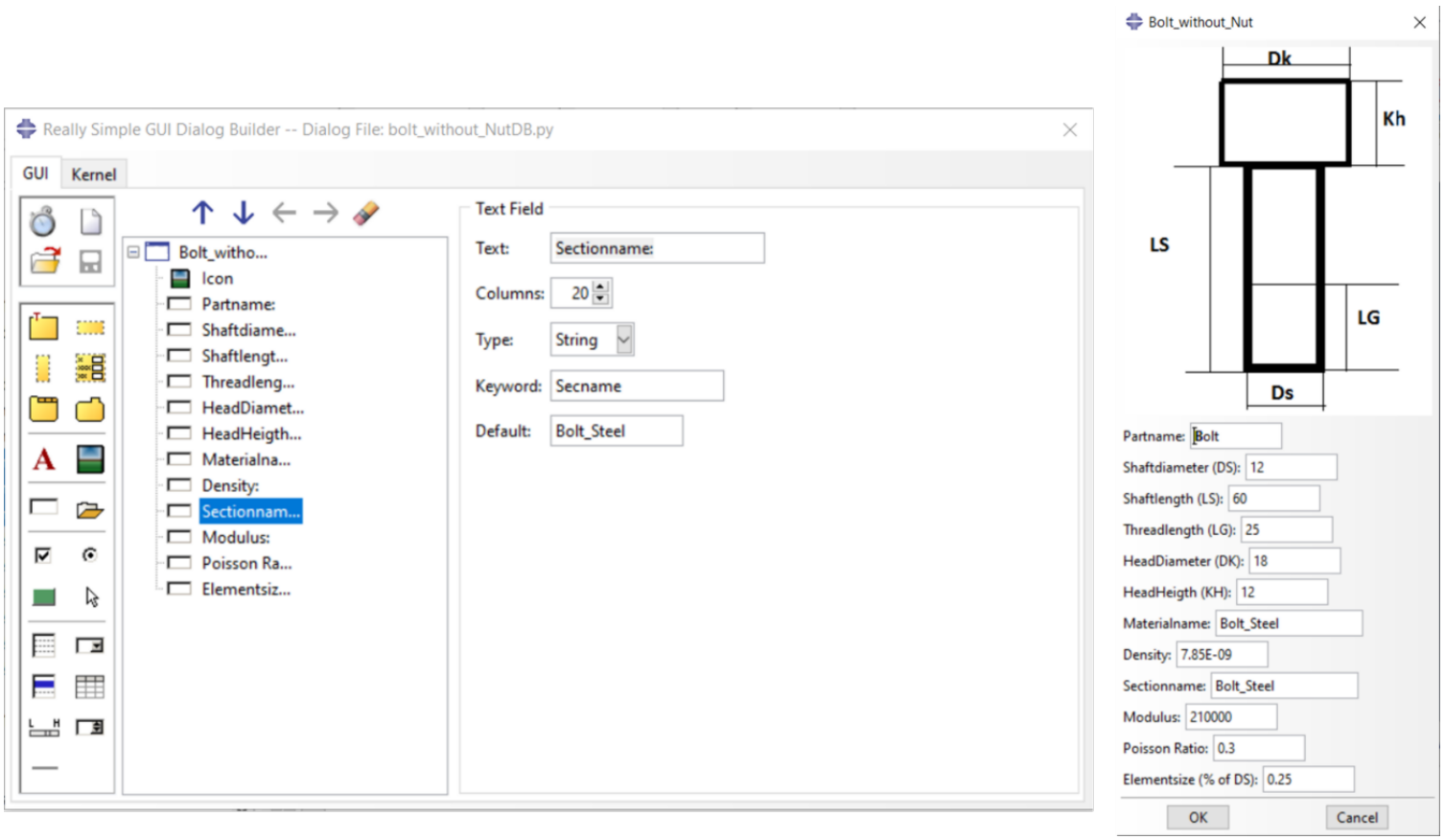
In this example, it makes sense to include the previously created sketch



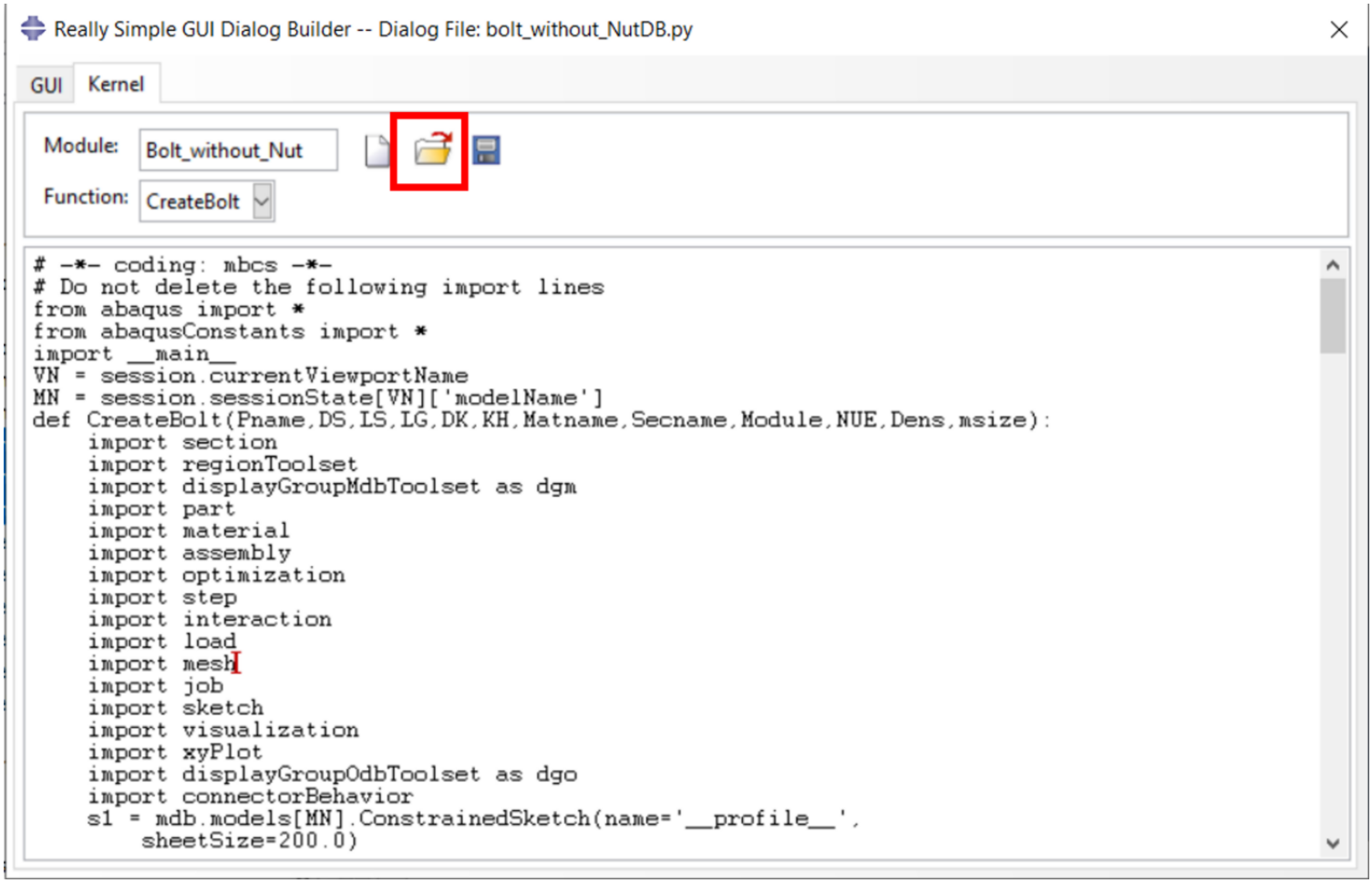
Now 12 text fields are created for the 12 parameters **Text**: Enter the text that describes the parameter in more detail here **Type**: A distinction is made between String (character string), Float (numerical value) and Integer (integer value) **Keyword**: Name of the parameter for the kernel **Default**: Preset value



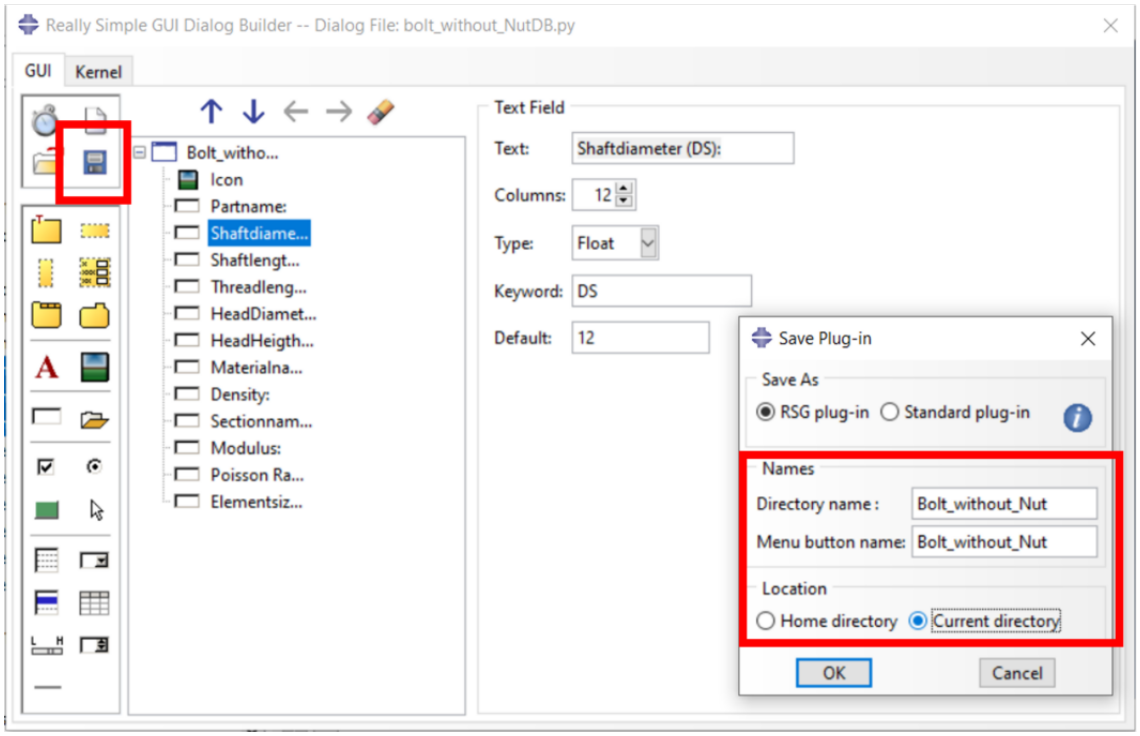
The finished GUI on the right in the picture below.

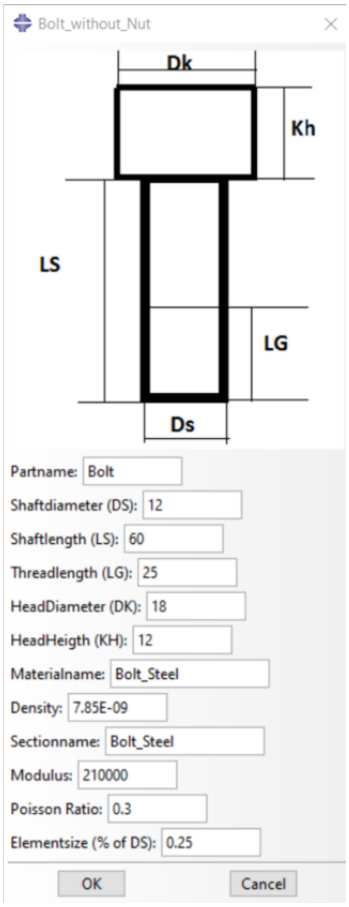


Loading the script: The script is loaded into the Abaqus/CAE kernel via the 'Kernel' tab and the 'Open file' icon. The script can also be edited further in this dialog.



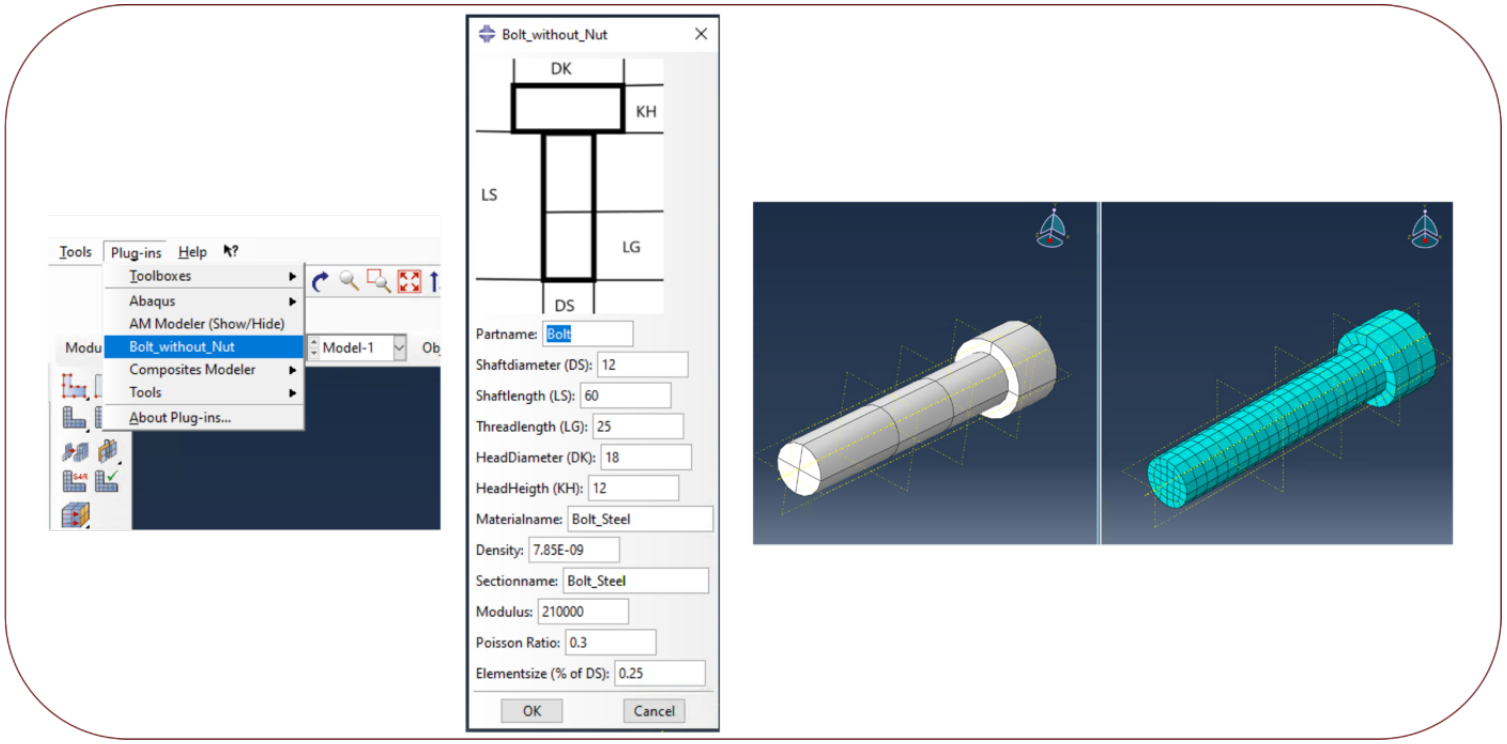
The 'Save file' icon now saves the script under a name to be assigned, preferably in the working directory. If the script is to be loaded with every call, it is recommended to move the plugin from the working directory to the plugin directory of the installation afterwards. For Windows versions, this is e.g. : C:\Simulia\CAE\plugins23 Alternatively, a central plugin directory can also be used. This is defined via plugin_central_dir in the environment file, see [4]. The PlugIn should only be stored in one directory, because Abaqus/CAE searches numerous locations for PlugIns and in case a PlugIn is found multiple times, a warning is issued when CAE is started. Saving it as an RSG PlugIn ensures that it can be subsequently edited and modified again.



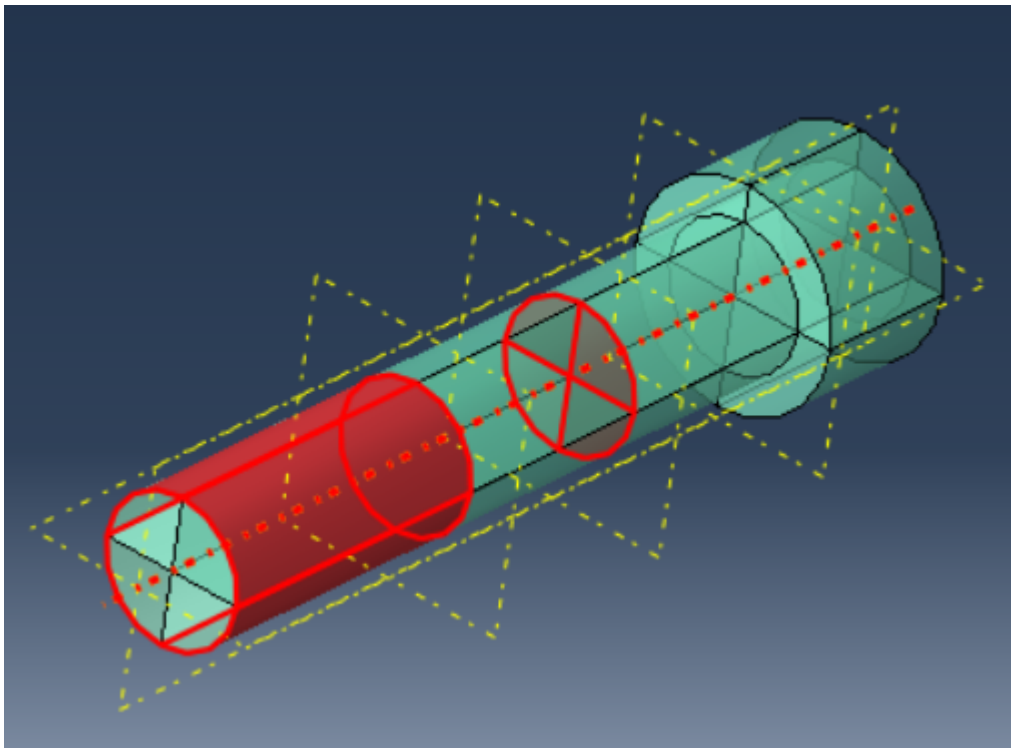


Result

In this example, the 'Bolt_without_Nut' directory for the plugin contains the following files: bolt_without_Nut_plugin.py, bolt_without_NutDB.py, bolt_without_nut.py and Simple_Bolt.png. We simply copy the entire directory to C:\Simulia\CAE\plugins23. If we start Abaqus/CAE now, we can call the plugin (left in the picture below). Note that in the plugin the names for the new Part, a Material and a Section are agreed, in the image below center. A separate part name should be used for each screw. If the materials of different screws differ, new names should be defined for the material and the section. Otherwise, previous definitions will be overwritten without warning. The screw has now been created and meshed (bottom right in the image) and can now be imported into the assembly.



The screw created after calling up the plug-in. Shown in red are: Datum Bolt_Axis , surface BL_Surf for the bolt force and surface "Pname_+_Tie" for the connection of the threaded part to the corresponding component.



Summary

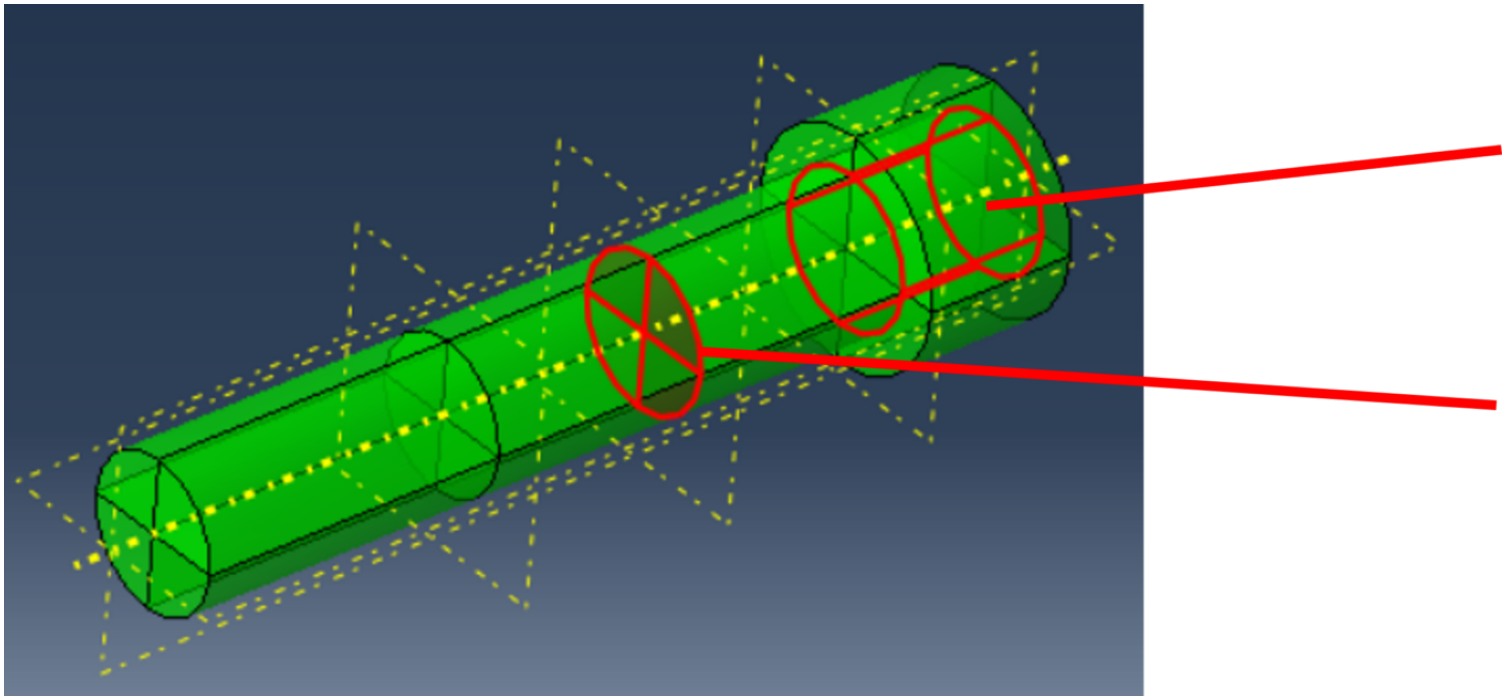
Once the plug-in has been successfully created, valuable tools are available to the user:

- Quick creation of any number of different screws
- Easy application of screw preload over predefined area without interactive instance selection
- Easy selection of surfaces for connection with the components of the assembly

Such a plug-in is indispensable, especially for very large assemblies. Even if this article seems complex, creating a plug-in with the RSG Builder is, with a little practice, a powerful and simple means of efficiently reducing the effort required for recurring processes.

Improvement Of The Script

It is not guaranteed that the plug-in will always work across all versions. This is because the partitions shown here in red were created with lines selected interactively. These lines appear in the script as a sequence of numbered lines whose numbering is not necessarily identical from version to version. The same applies to the inner surfaces for the bolt preload force and the surfaces of the threaded part. In this chapter, a more general scripting procedure is presented, which is intended to prevent the occurrence of such potential problems.



Partition of the screw head

Surfaces for bolt force

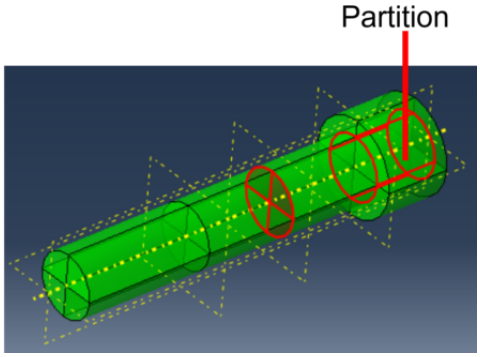
Alternative Definition Of The Partition Of The Bolt Head

In the above procedure with the Macro Recorder, the lines (edges) are selected directly and the script uses the IDs of these edges, as shown in the image below.

```
    pickedEdges =(e[9], e[20], e[33], e[42])
    p.PartitionCellByExtrudeEdge(line=d2[3], cells=pickedCells, edges=pickedEdges,
        sense=REVERSE)
```

We replace this definition with a search for the 4 arcs required for the partition. We benefit from the fact that we build up the screw ourselves within the part and thus the values are available for the search. The script block shown in the image below replaces the code from the macro recorder.

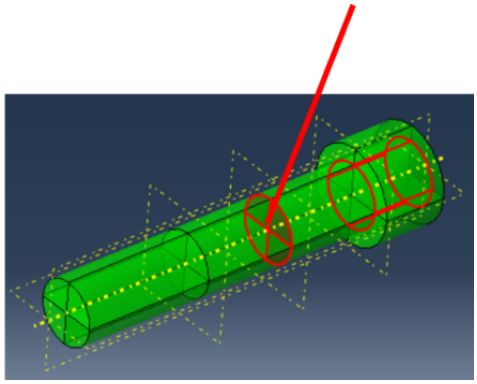
```
    RS=0.5*DS
    e = p.edges
    found_edges1 = e.getClosest(coordinates = ((RS*2/3, RS*2/3, KH),))
    found_edges2 = e.getClosest(coordinates = ((-RS*2/3, RS*2/3, KH),))
    found_edges3 = e.getClosest(coordinates = ((-RS*2/3, -RS*2/3, KH),))
    found_edges4 = e.getClosest(coordinates = ((RS*2/3, -RS*2/3, KH),))
    edges2=e.findAt(((found_edges1.values()[0][1]), ),
                    ((found_edges2.values()[0][1]), ),
                    ((found_edges3.values()[0][1]), ),
                    ((found_edges4.values()[0][1]), ))
    p.PartitionCellByExtrudeEdge(line=d1[2], cells=pickedCells, edges=edges2,
        sense=REVERSE)
```



Alternative Selection Of Inner Surfaces For Bolt Force

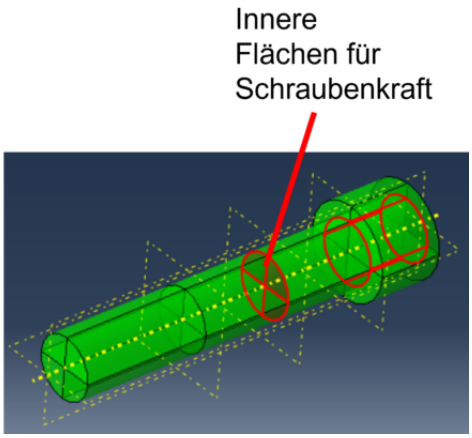
By using the Macro Recorder, IDs are used for the surfaces, see image below. These IDs could change.

```
s = p.faces
side1Faces = s.getSequenceFromMask(mask=('[#20000000 #10100008 ]', ), )
p.Surface(side1Faces=side1Faces, name='BL_Surf')
```



We now use a script to find the 4 quarter circles required for the bolt force. The following block replaces the code from the macro recorder.

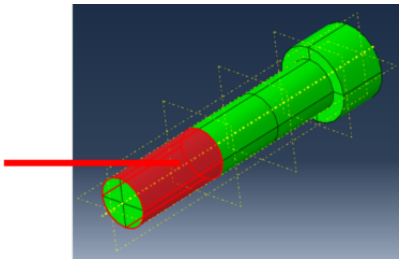
```
zpos=KH+0.5*(LS-LG)
    for i in range(1,5):
        a=p.faces.findAt(((0.25*DS,0.25*DS,zpos), ),
        ((-0.25*DS,-0.25*DS,zpos), ),
    ((0.25*DS,-0.25*DS,zpos), ),
    ((-0.25*DS,0.25*DS,zpos), ))
        if i==1:
            x=a
        else:
            x=x+a
    p.Surface(side1Faces=x, name='BL Surf')
```



Selection Of Surfaces For Connection To The Component

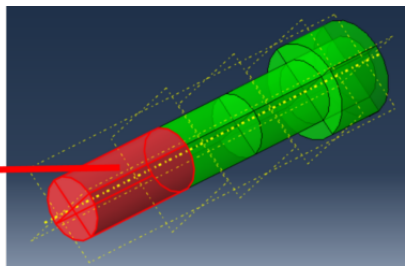
The same applies to the surfaces for the connection to the component via a TIE constraint. We want to replace the getSequenceFromMask created by the Macro Recorder, see image below, with a more general formulation.

```
p = mdb.models['Model-1'].parts['Bolt_Skript']
s = p.faces
side1Faces = s.getSequenceFromMask(mask=('[#a0000000 #200002 ]', ), )
p.Surface(side1Faces=side1Faces, name='Bolt_Tie')
#: The surface 'Bolt_Tie' has been created (4 faces).
```



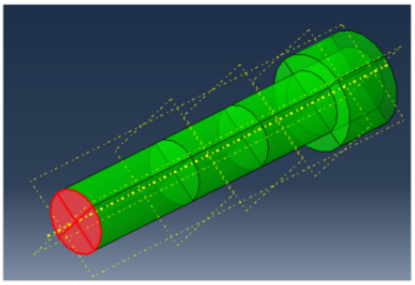
To define the areas independently of their numbering, a few more steps are required, which, as before, must be added to the script in the kernel and replace the code from the macro recorder shown above. First, the following command selects all faces of the front partition. ATTENTION: Due to a bug in V2024, the search range of the cylinder must be increased by a small amount (0.00001) Additional step #1

```
#   Get all Surfaces of cell
    a=p.faces.getByBoundingCylinder((0.,0.,KH+LS-LG),(0.,0.,KH+LS),RS)
    p.Surface(side1Faces=a, name='CYL Surf')
```

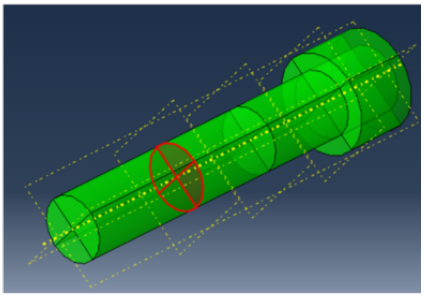


Selection of the surfaces on the two end faces, additional steps #2 and #3

```
RS=0.5*DS
    zpos=KH+LS
#   Get front Surfaces
    for i in range(1,5):
        a=p.faces.findAt(((0.25*DS,0.25*DS,zpos), ),
        ((-0.25*DS,-0.25*DS,zpos), ),
    ((0.25*DS,-0.25*DS,zpos), ),
    ((-0.25*DS,0.25*DS,zpos), ))
        if i==1:
            x=a
        else:
            x=x+a
    p.Surface(side1Faces=x, name='endsurf')
```

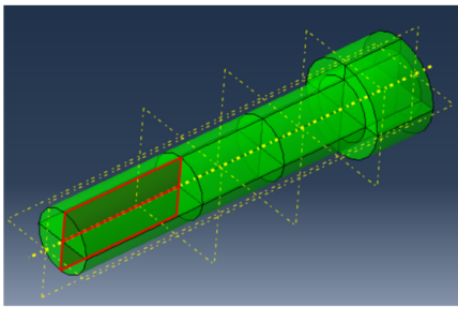


```
    zpos=KH+LS-LG
#   Get back Surfaces
    for i in range(1,5):
        a=p.faces.findAt(((0.25*DS,0.25*DS,zpos), ),
        ((-0.25*DS,-0.25*DS,zpos), ),
    ((0.25*DS,-0.25*DS,zpos), ),
    ((-0.25*DS,0.25*DS,zpos), ))
        if i==1:
            x=a
        else:
            x=x+a
    p.Surface(side1Faces=x, name='intsurf')
```

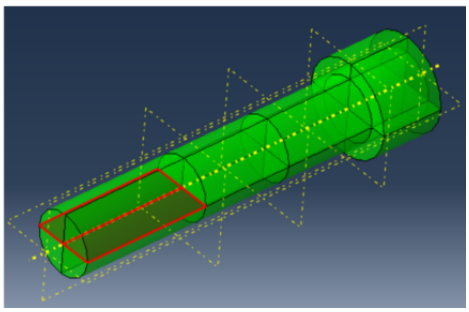


Selection of inner vertical and horizontal flats, additional steps #4 and #5

```
#       Get y Surfaces
    for i in range(1,2):
        a=p.faces.findAt(((0.,0.25*DS,zpos), ),
        ((0.,-0.25*DS,zpos), ))
        if i==1:
            x=a
        else:
            x=x+a
    p.Surface(side1Faces=x, name='ysurf')
```



```
#       Get x Surfaces
    for i in range(1,2):
        a=p.faces.findAt(((0.25*DS,0.,zpos), ),
        ((-0.25*DS,-0.,zpos), ))
        if i==1:
            x=a
        else:
            x=x+a
    p.Surface(side1Faces=x, name='xsurf')
```

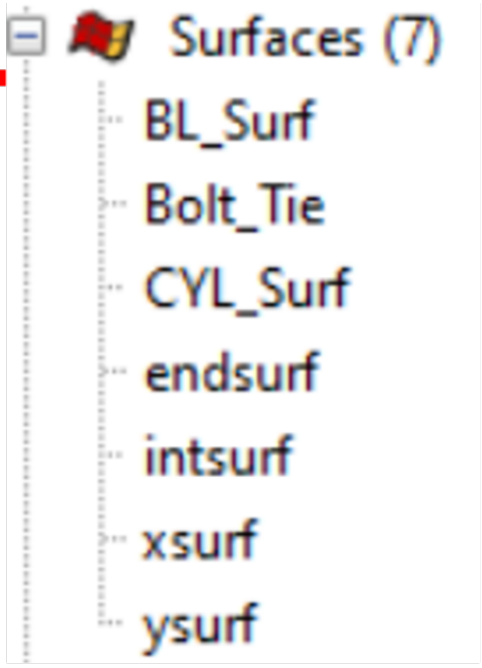


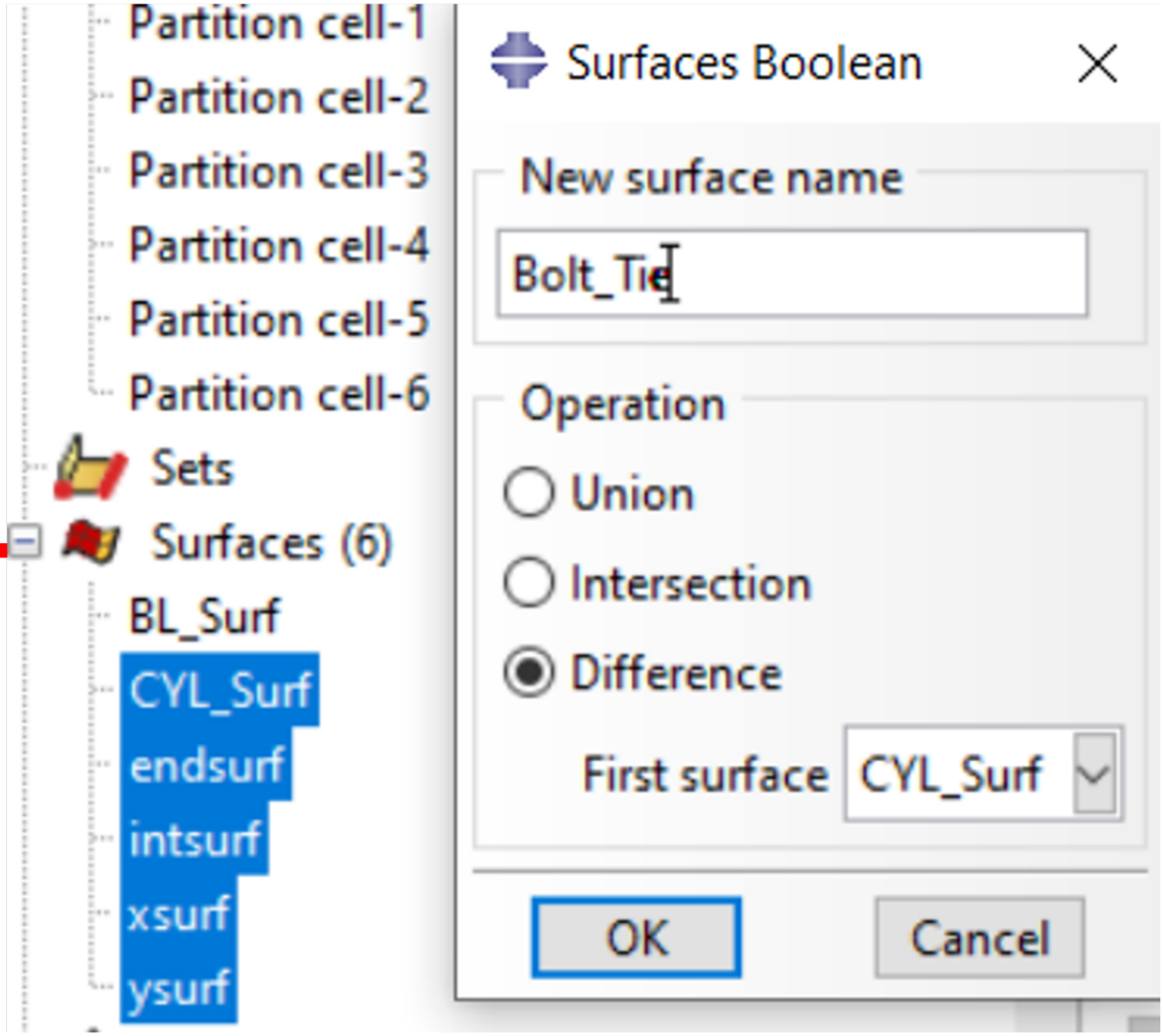
The areas created can now be found in the part tree.

```
a=mdb.models[MN].parts[Pname]
    a.SurfaceByBoolean(name=Pname+'_Tie', operation=DIFFERENCE, surfaces=(
        a.surfaces['CYL_Surf'], a.surfaces['ysurf'], a.surfaces['xsurf'],
        a.surfaces['intsurf'], a.surfaces['endsurf'], ))
```

Additional step #6: Create the surface Bolt_Tie by Boolean operation: Subtract the surfaces endsurf, intsurf, xsurf and ysurf from the surface CYL_Surf (right mouse click on the selected surfaces). In the

script, Bolt_Tie becomes "Pname+Tie".


Surfaces (7)
BL_Surf
Bolt_Tie
CYL_Surf
endsurf
intsurf
xsurf
ysurf

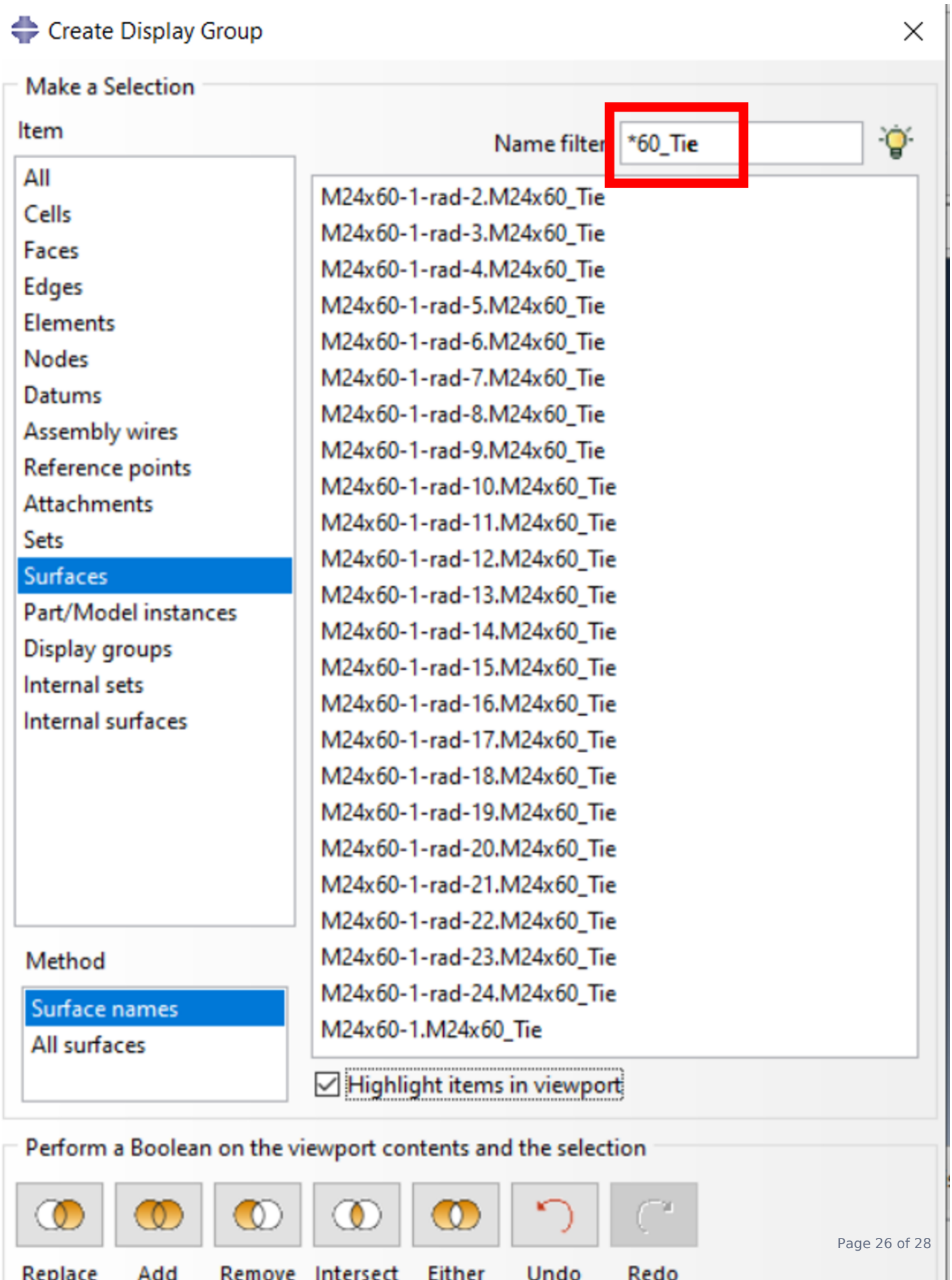

Partition cell-1
Partition cell-2
Partition cell-3
Partition cell-4
Partition cell-5
Partition cell-6
Sets
Surfaces (6)
BL_Surf
CYL_Surf
endsurf
intsurf
xsurf
ysurf
Surfaces Boolean
New surface name
Bolt_Tie
Operation
Union
Intersection
Difference
First surface
CYL_Surf
OK
Cancel

Additional and final step #7: The now redundant areas are deleted

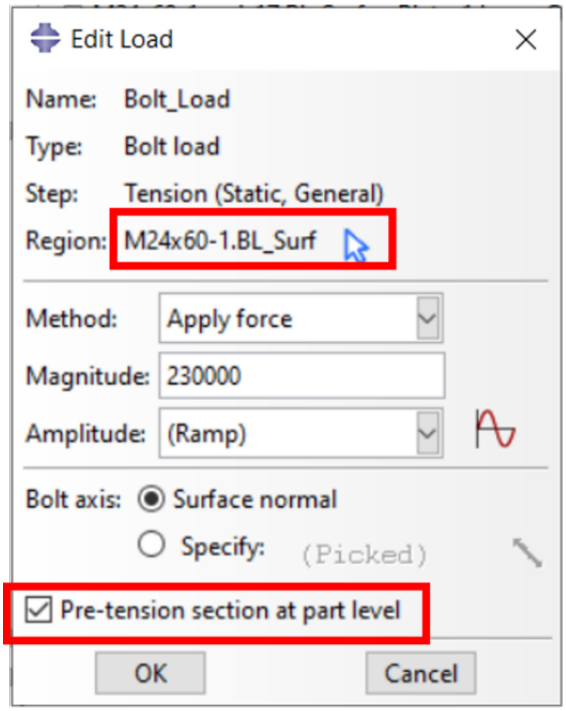
```
mdb.models[MN].parts[Pname].deleteSurfaces(surfaceNames=('CYL_Surf',
    'endsurf', 'intsurf', 'xsurf', 'ysurf', ))
```

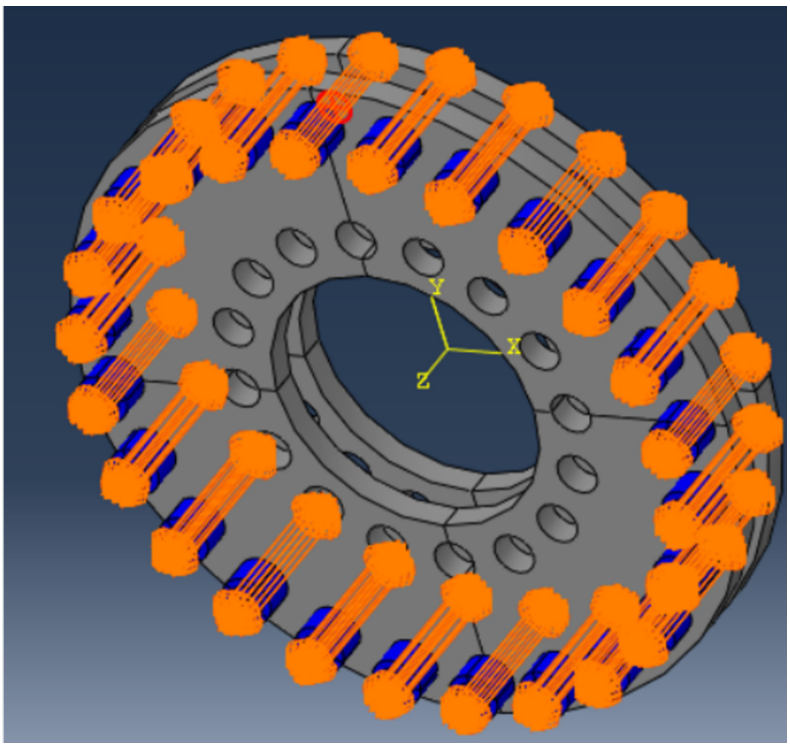
Simplified Definition Of Many Instances Of Screws

If many instances of one or more screws are required, the connection to the components can often be combined. In this example, the screw has the name "M24x60". The surfaces for the connection from the threaded surface of the screw to the part, with standardized name "Pname+_Tie", can now be easily combined into a single surface without the need to select the instances. Select these surfaces via the filter in "Create Display Group" (*-filter is helpful) and save individual surfaces, e.g. "M24x60_Tie_Secondary".


Create Display Group
Make a Selection
Item
All
Cells
Faces
Edges
Elements
Nodes
Datums
Assembly wires
Reference points
Attachments
Sets
Surfaces
Part/Model instances
Display groups
Internal sets
Internal surfaces
Name filter
*60_Tie
M24x60-1-rad-2.M24x60_Tie
M24x60-1-rad-3.M24x60_Tie
M24x60-1-rad-4.M24x60_Tie
M24x60-1-rad-5.M24x60_Tie
M24x60-1-rad-6.M24x60_Tie
M24x60-1-rad-7.M24x60_Tie
M24x60-1-rad-8.M24x60_Tie
M24x60-1-rad-9.M24x60_Tie
M24x60-1-rad-10.M24x60_Tie
M24x60-1-rad-11.M24x60_Tie
M24x60-1-rad-12.M24x60_Tie
M24x60-1-rad-13.M24x60_Tie
M24x60-1-rad-14.M24x60_Tie
M24x60-1-rad-15.M24x60_Tie
M24x60-1-rad-16.M24x60_Tie
M24x60-1-rad-17.M24x60_Tie
M24x60-1-rad-18.M24x60_Tie
M24x60-1-rad-19.M24x60_Tie
M24x60-1-rad-20.M24x60_Tie
M24x60-1-rad-21.M24x60_Tie
M24x60-1-rad-22.M24x60_Tie
M24x60-1-rad-23.M24x60_Tie
M24x60-1-rad-24.M24x60_Tie
M24x60-1.M24x60_Tie
Highlight items in viewport
Method
Surface names
All surfaces
Perform a Boolean on the viewport contents and the selection
Replace
Add
Remove
Intersect
Either
Undo
Redo

The screw load can now be applied just as easily without having to select the instance. To do this, just select the surface "'Pname'+BL_Surf" and apply the force or displacement (depending on the method). By activating " Pre-tension section at part level" the force is applied to ALL screws. .





Literature

[1] Type Hints for Abaqus/Python Scripting(formerly Hailin Wang) [2] Abaqus Scripting Users Guide in the DS User Manuals [3] Abaqus Scripting Reference Guide in the DS User Manuals [4] Abaqus/CAE Central Plugin Directory in the DS User Manuals [5] Notepad++ Website

Plugin Download

We are happy to provide you with the plugin. We also have a variant with mother. Unfortunately, we cannot provide a download link yet. Please use the contact option below. A request does not commit you to anything, just as we cannot take responsibility for the use of the plugin. Bolt_without_Nut (bolt without nut, as described above) (download link/authorization required) Bolt_with_Nut Bolt with nut (download link/ /authorization required)

