An Abaqus-MATLAB tutorial for Jointed Systems

Nidish Narayanaa Balaji

September 20, 2023

Contents

1 Preprocessing

The Brake-Reuß Beam (BRB) is a 3-bolted lap-joint beam which will serve as our tutorial joint structure (read more about the benchmark here). To make the tutorial self-contained, the tutorial will begin with CAD-modeling of the BRB in Abaqus and take you all the way to matrix extraction and nonlinear dynamics analysis on MATLAB/OCTAVE. Buckle up! :)

A picture of the parts of the BRB is shown in Fig. 1 and this will be used to guide the modeling here. Standard 5/16 bolt-nut-washers will be used for the assembly. Since Bolt thread interactions won't be modeled here, the bolt shank is modeled as a smooth cylinder (see 2.2 for more details on bolt prestress modeling).

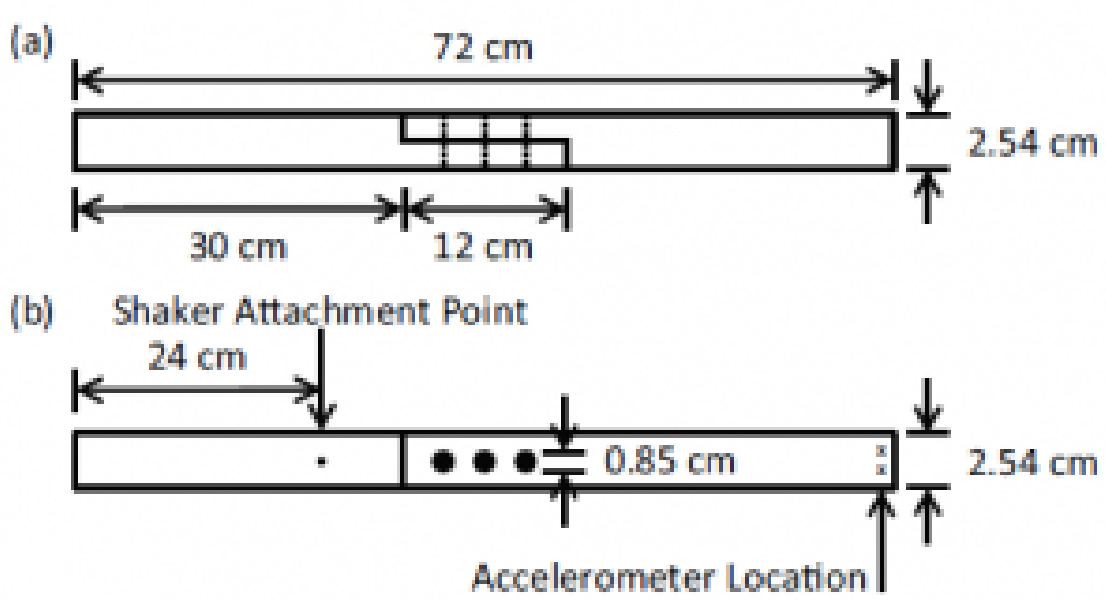


Figure 1: Geometry of the BRB Benchmark

Note: If you already have a model of your structure and want to just apply the node-selection/matrix extraction, you can safely ignore the sections particular to the modeling of the BRB and just go through the remaining, whose instructions are quite general. The general principles in the modeling section are, however, recommended for all jointed structure modeling.

1.1 A Note on Python Scripting

Abaqus Python scripting is quite powerful and will be used quite liberally throughout this tutorial. The following are some tips/tricks to get started for beginners:

- For beginners, the easiest way to start scripting is to open up CAE (the GUI) and do the necessary tasks. Abaqus would have saved the actions in a Python file called `abaqus.rpy`. This is just python code that can be imported into abaqus to repeat the same tasks. All standard python commands work so this makes it very helpful.
- On Windows the **work directory** may be an unfamiliar concept (on Linux it is just the directory from which Abaqus is launched). This directory can be manually set by `File->Set Work Directory`, following which all the files (including the `abaqus.rpy` file) will be put in the specified folder.
- Fig 2 shows a pictorial overview of the different options available in Abaqus for scripting support. The Abaqus PDE can be used for Abaqus python script development.

1.2 Modeling Parts SCRIPT:GUI

We will first model the two "halves" of the BRB. The following steps enumerate the process (can be skipped if you already have a model).

We will do the construction through Abaqus Python code that can be imported through `File->Run Script` or just typed into the command line.

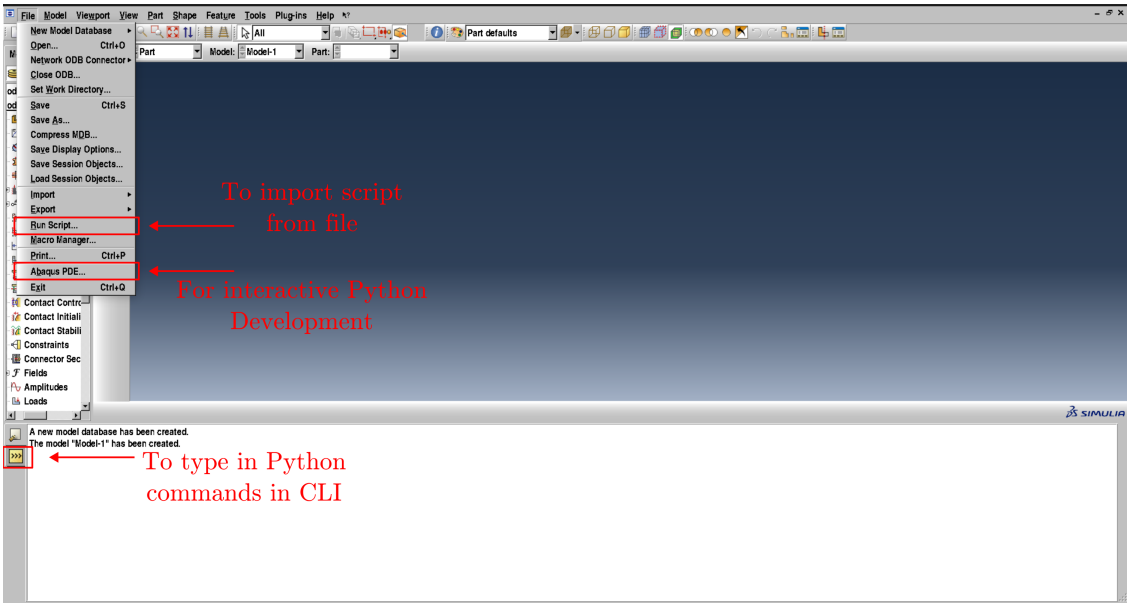


Figure 2: Options in Abaqus for Scripting

1. First import the following code to import all the necessary objects and then the material properties (steel, here).

```
# -*- coding: utf-8 -*-
import sys

from part import *
from material import *
from section import *
from assembly import *
from step import *
from interaction import *
from load import *
from mesh import *
from optimization import *
from job import *
from sketch import *
from visualization import *
from connectorBehavior import *

mdl = mdb.models['Model-1']
##############################
# MATERIAL PROPERTIES: STEEL #
##############################
mdl.Material(name='STEEL')

steel = mdl.materials['STEEL']
steel.Density(table=((7800.0, ), ))
```

```
steel.Elastic(table=((2e11, 0.29),))
mdl.HomogeneousSolidSection(material='STEEL', name=
                            'Section-1', thickness=None)
```

The above code creates a new material called "STEEL", with Young's Modulus 200GPa, Poisson's ratio 0.29, and density 7800 kg/m^3.

2. Now use the following to model the half beam (this can be done on CAE with the GUI quite easily).

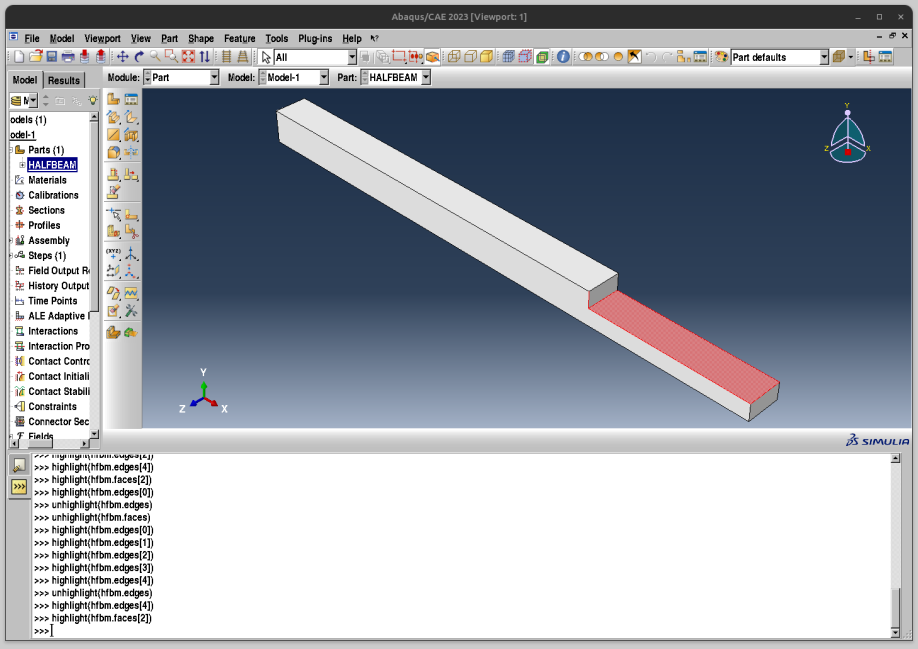
```
mdl = mdb.models['Model-1']

####################
# PART : HALFBEAM #
####################
# 1. Sketch and Extrude
mdl.ConstrainedSketch(name='__profile__', sheetSize=2.0)
sktch = mdl.sketches['__profile__']
sktch.Line(point1=(-36e-2, 1.27e-2), point2=(-6e-2, 1.27e-2))
sktch.Line(point1=(-6e-2, 1.27e-2), point2=(-6e-2, 0))
sktch.Line(point1=(-6e-2, 0), point2=(6e-2,0))
sktch.Line(point1=(6e-2,0), point2=(6e-2,-1.27e-2))
sktch.Line(point1=(6e-2,-1.27e-2), point2=(-36e-2,-1.27e-2))
sktch.Line(point1=(-36e-2,-1.27e-2), point2=(-36e-2,1.27e-2))

mdl.Part(dimensionality=THREE_D, name='HALFBEAM', type=DEFORMABLE_BODY)
hfbm = mdl.parts['HALFBEAM']
hfbm.BaseSolidExtrude(depth=25.4e-3, sketch=sktch)
del sktch

# 2. Cut out Holes
mdl.ConstrainedSketch(name='__profile__', sheetSize=2.0,
                      transform=
                      hfbm.MakeSketchTransform(
                          sketchPlane=hfbm.faces[2],
                          sketchPlaneSide=SIDE1,
                          sketchUpEdge=hfbm.edges[8],
                          sketchOrientation=RIGHT,
                          origin=(0.0, 0.0, 1.27e-2)))
sktch = mdl.sketches['__profile__']
cs = [-3e-2, 0.0, 3e-2];
gs = []
for i in range(3):
    gs.append(sktch.CircleByCenterPerimeter(center=(cs[i], 0),
                                            point1=(cs[i]+0.85e-2/2, 0)))

hfbm.CutExtrude(sketchPlane=hfbm.faces[2], sketchPlaneSide=SIDE1,
                sketchUpEdge=hfbm.edges[8],
```

```
                sketchOrientation=RIGHT, sketch=sktch)


# 3. Partition object
hfbm.PartitionCellByExtendFace(cells=hfbm.cells,
                               extendFace=hfbm.faces[6])
hfbm.PartitionCellByExtendFace(cells=hfbm.cells,
                               extendFace=hfbm.faces[7])

mdl.ConstrainedSketch(name='__profile__', sheetSize=2.0,
                      gridSpacing=30e-3, transform=
                      hfbm.MakeSketchTransform(
                          sketchPlane=hfbm.faces[11],
                          sketchPlaneSide=SIDE1,
                          sketchUpEdge=hfbm.edges[27],
                          sketchOrientation=RIGHT,
                          origin=(0.0, 0.0, 1.27e-2)))
sktch = mdl.sketches['__profile__']
cs = [-3e-2, 0.0, 3e-2];
wor = i2m*0.34375
for i in range(3):
    sktch.CircleByCenterPerimeter(center=(cs[i], 0),
                                  point1=(cs[i]+wor, 0))
hfbm.PartitionCellBySketch(cells=hfbm.cells, sketch=sktch,
                           sketchUpEdge=hfbm.edges[27],
                           sketchPlane=hfbm.faces[11])
hfbm.PartitionCellByExtrudeEdge(cells=hfbm.cells, edges=hfbm.edges[0],
                                line=hfbm.edges[-1], sense=FORWARD)
hfbm.PartitionCellByExtrudeEdge(cells=hfbm.cells, edges=hfbm.edges[3],
                                line=hfbm.edges[-1], sense=FORWARD)
hfbm.PartitionCellByExtrudeEdge(cells=hfbm.cells, edges=hfbm.edges[6],
                                line=hfbm.edges[-1], sense=FORWARD)

pt = hfbm.InterestingPoint(edge=hfbm.edges[-2], rule=MIDDLE)
hfbm.PartitionCellByPlanePointNormal(cells=hfbm.cells, point=pt, normal=hfbm.edges[-2]

pt = hfbm.InterestingPoint(edge=hfbm.edges[75], rule=MIDDLE)
hfbm.PartitionCellByPlanePointNormal(cells=hfbm.cells, point=pt, normal=hfbm.edges[75]

pt = hfbm.InterestingPoint(edge=hfbm.edges[39], rule=MIDDLE)
hfbm.PartitionCellByPlanePointNormal(cells=hfbm.cells, point=pt, normal=hfbm.edges[39]

pt = hfbm.InterestingPoint(edge=hfbm.edges[85], rule=MIDDLE)
hfbm.PartitionCellByPlanePointNormal(cells=hfbm.cells, point=pt, normal=hfbm.edges[85]

pt = hfbm.InterestingPoint(edge=hfbm.edges[39], rule=MIDDLE)
hfbm.PartitionCellByPlanePointNormal(cells=hfbm.cells, point=pt, normal=hfbm.edges[39]

```

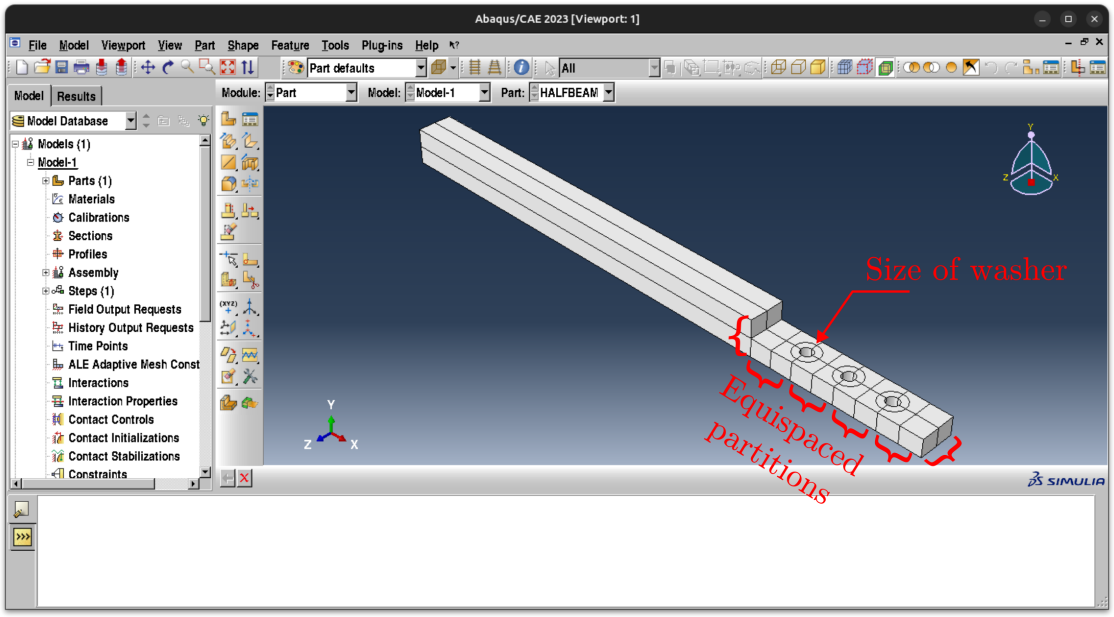
```
115  pt = hfbm.InterestingPoint(edge=hfbm.edges[41], rule=MIDDLE)
116  hfbm.PartitionCellByPlanePointNormal(cells=hfbm.cells, point=pt, normal=hfbm.edges[41]
117
118  pt = hfbm.InterestingPoint(edge=hfbm.edges[111], rule=MIDDLE)
119  hfbm.PartitionCellByPlanePointNormal(cells=hfbm.cells, point=pt, normal=hfbm.edges[111
120
121  pt = hfbm.InterestingPoint(edge=hfbm.edges[135], rule=MIDDLE)
122  hfbm.PartitionCellByPlanePointNormal(cells=hfbm.cells, point=pt, normal=hfbm.edges[135
123
124  # 4. Assign Material
125  regn = hfbm.Set(cells=hfbm.cells, name='Set-1')
126  hfbm.SectionAssignment(region=regn, sectionName='Section-1')
```

Scripting note

You can see that certain faces and edges were used in the above. A quick way to check which face is what will be to use the `highlight` command on the Abaqus python console. Here is an example:

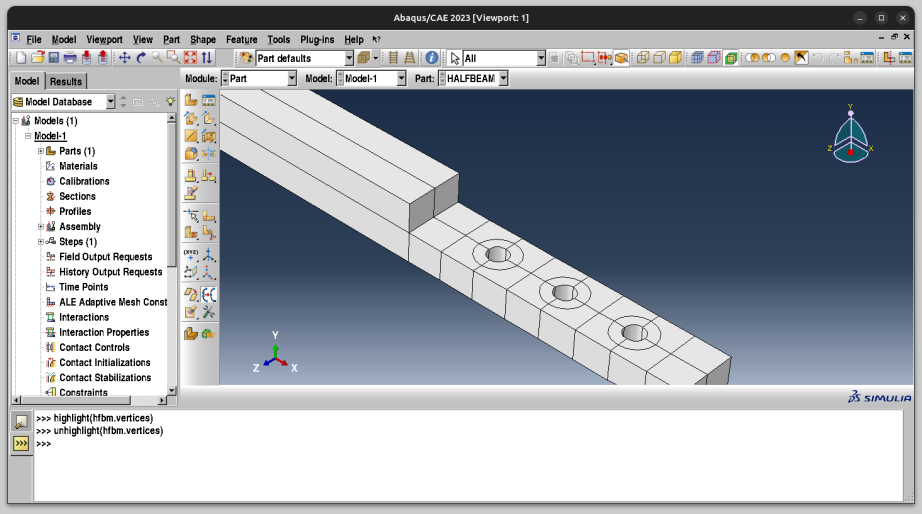


At the end of this step, you should have a partitioned part that looks like this. The partitioning is done in this way to help with the seeded meshing, constraint enforcement, etc.



IMPORTANT! Geometry Correction Note

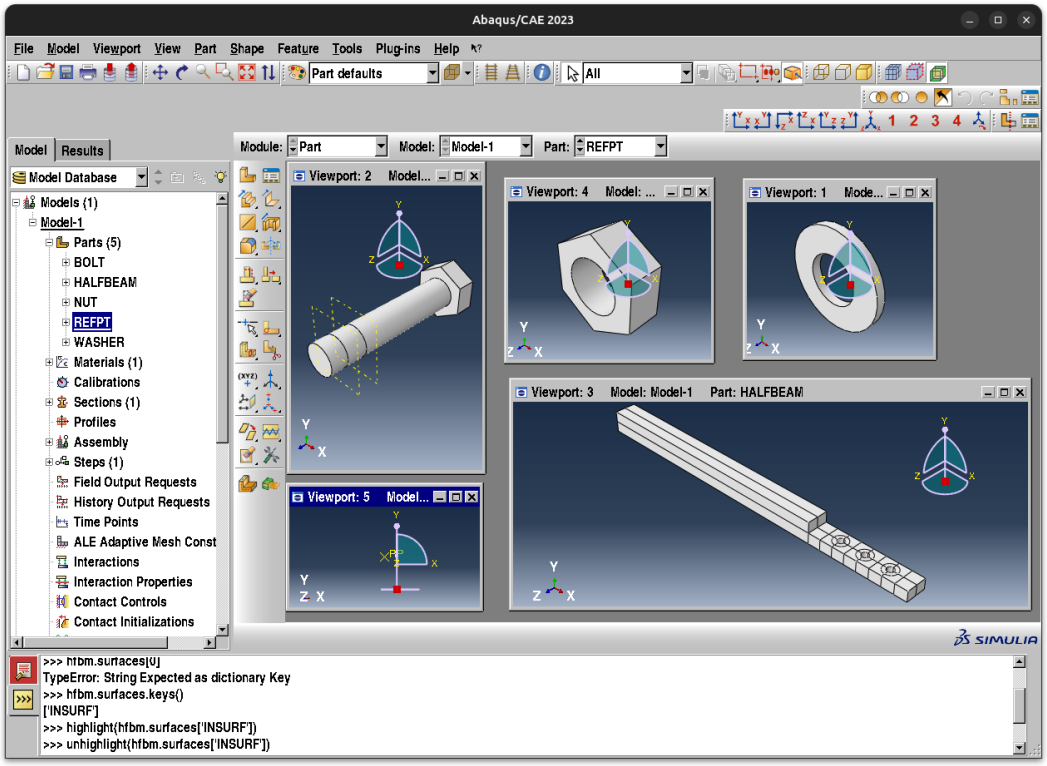
You have to ensure that the curved edges on the above are, indeed, single edges. You will run into meshing issues if this is not the case. If not, you will have to use the "Merge Edges" tool in the Part module and individually select each edge and merge them.



1. Create a Reference Point Part called `REFPT`. This will be necessary for the application of bolt prestress (see 2.2 below).

```
127  rpt = mdl.Part(name='REFPT', dimensionality=THREE_D,
128                 type=DEFORMABLE_BODY)
129  rpt.ReferencePoint(point=(0.0, 0.0, 0.0))
130
131  rpt.Set(name='Set-1', referencePoints=rpt.referencePoints.values())
```

2. Now import the nuts, washers and bolts by importing the file `https://github.com/Nidish96/Abaqus4Joints/blob/main/scripts/c_nutwasherbolt_516.py`. You should be able to see the nut, washer and bolt, along with the half beam and reference point created before, as follows after importing:



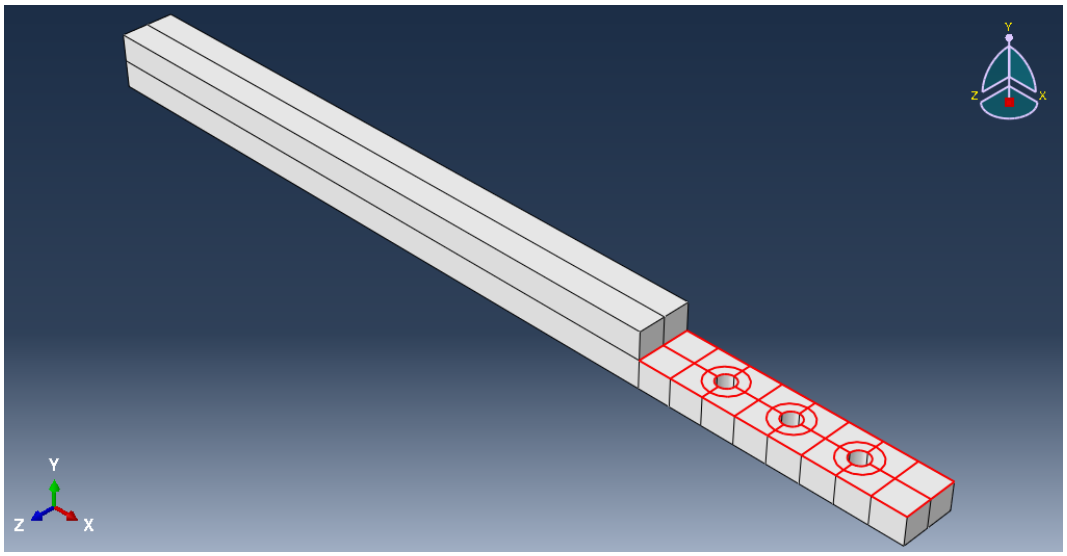
The Python script also assigns the material "STEEL" (see above) to the parts.

The file $\text{model}_{\text{step0.cae}}$ stores the CAE file at the end of the above steps (building parts, and partitioning).

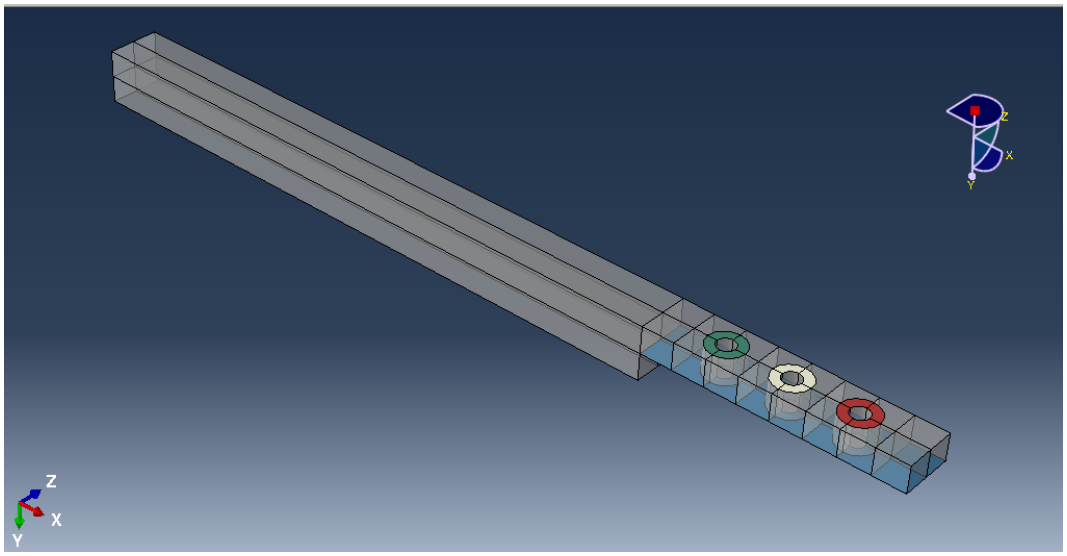
1.3 Creating appropriate surface sets GUI

It is necessary to choose appropriate surface sets for the constraint enforcement and, eventually, the interfacial mesh extraction. Doing this with surfaces (before meshing) is advantageous since the same scripts can be reused even if the model is remeshed.

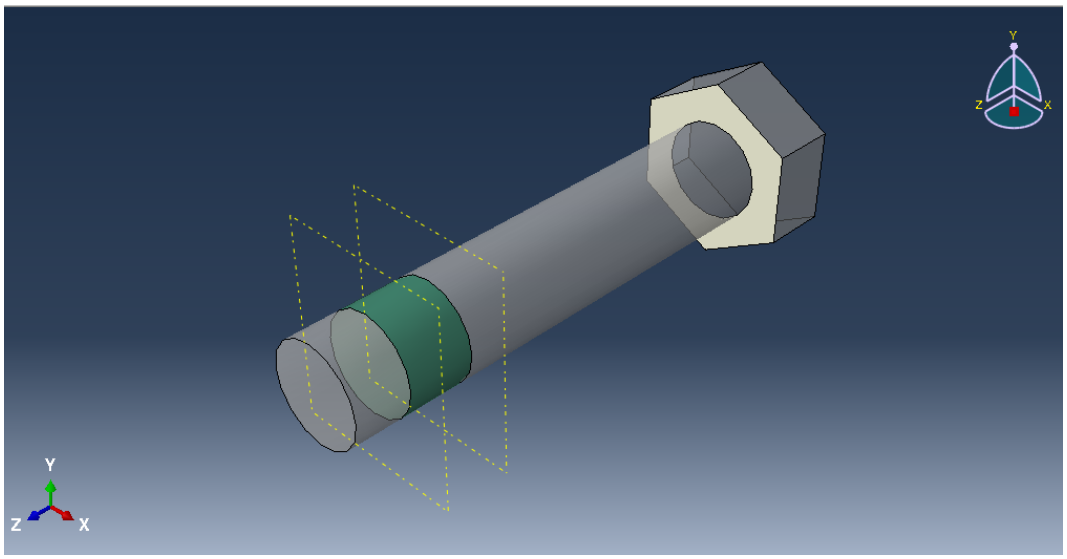
1. First select the **interfacial faces** on the half beam model and assign the name `INSURF` to it. You can do this through `Tools->Surface->Create` and then selecting the appropriate faces through the GUI. Select the faces by holding `Shift` and deselect by holding `Ctrl`. Here is a picture of the surface highlighted in the viewport.



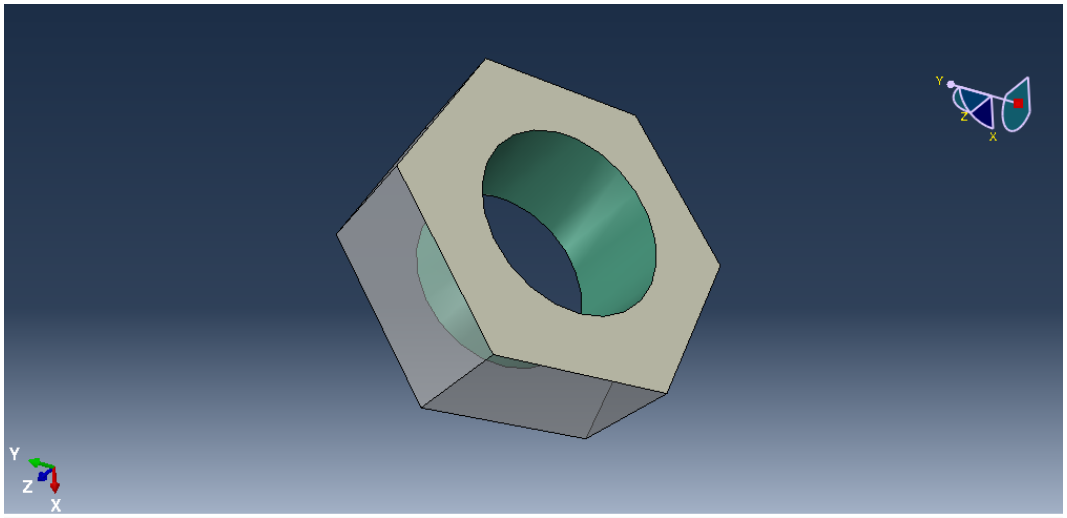
2. Turn to the *outside* of the beam and select the faces around the bolts individually and name them `BmW1`, `BmW2`, and `BmW3` respectively. These will be tied to the washer for the analysis. (`BmWi` denotes the i$^{\text{th}}$ Beam-Washer contact). Here is a picture of the relevant surfaces highlighted (using different colors for each).



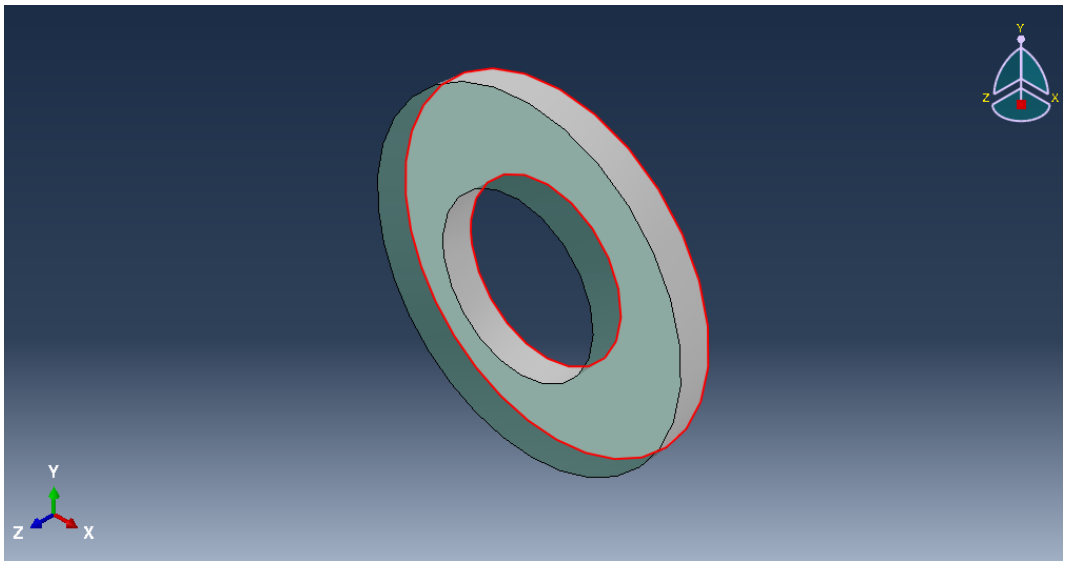
3. Create two surfaces on the `BOLT` model as shown in the figure below. Name the surface marked white as `BlW` and the surface marked green as `BN`. (`BlW` denotes the Bolt-Washer contact surface; `BN` denotes the Bolt-Nut contact surface)



4. Create two surfaces on the `NUT` model as shown in the figure below. Name the white surface as `NW` and the green surface as `NB`. (`NW` denotes the Nut-Washer contact surface; `NB` denotes the Nut-Bolt contact surface) **Note the direction axes carefully.**



5. Create two surface on the `WASHER` model and label them as `WSTOP` and `WSBOT` (short for Washer-Surface Top and Bottom). Here is a graphical depiction of the model with the surfaces.

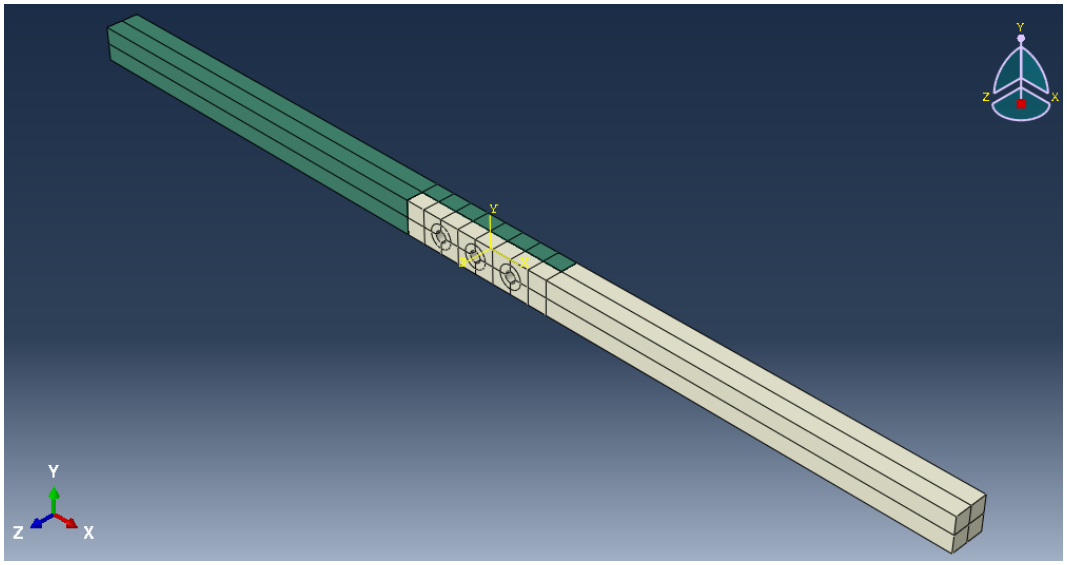


Now all the parts have been created and relevant surfaces have been identified. **Note**: It is important to have traceable but short names so that a lot of the repetitive tasks can be sped up considerably using scripting.

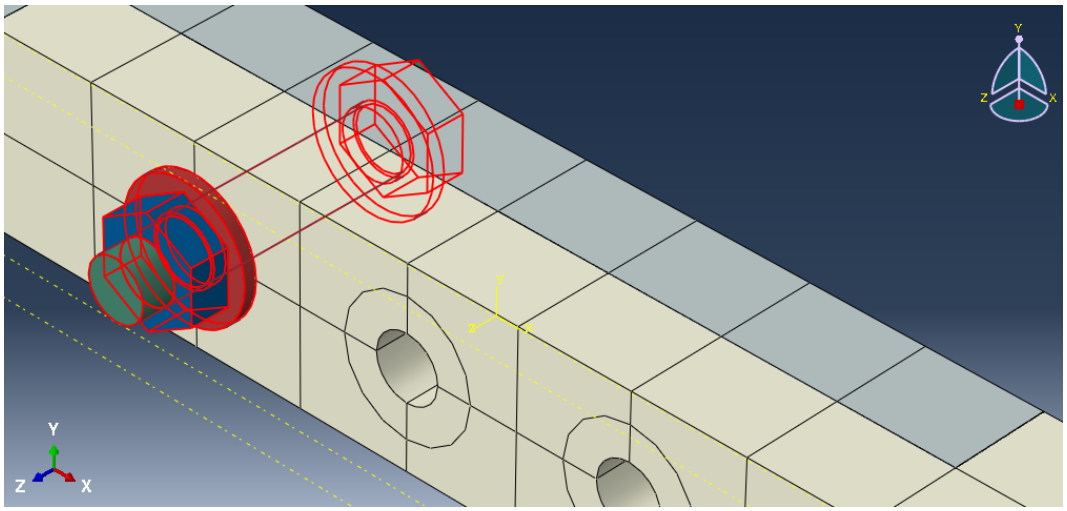
1.4 Create Assembly GUI

Note: While importing the parts, choose Instance Type as "Independent (Mesh on Instance)". This will be advantageous if we want to modify the meshes just at the interface for mirror symmetry. We will do independent meshing since it is good practice, although independent meshes are not a requirement for this example (dependent meshes can be used due to symmetry).

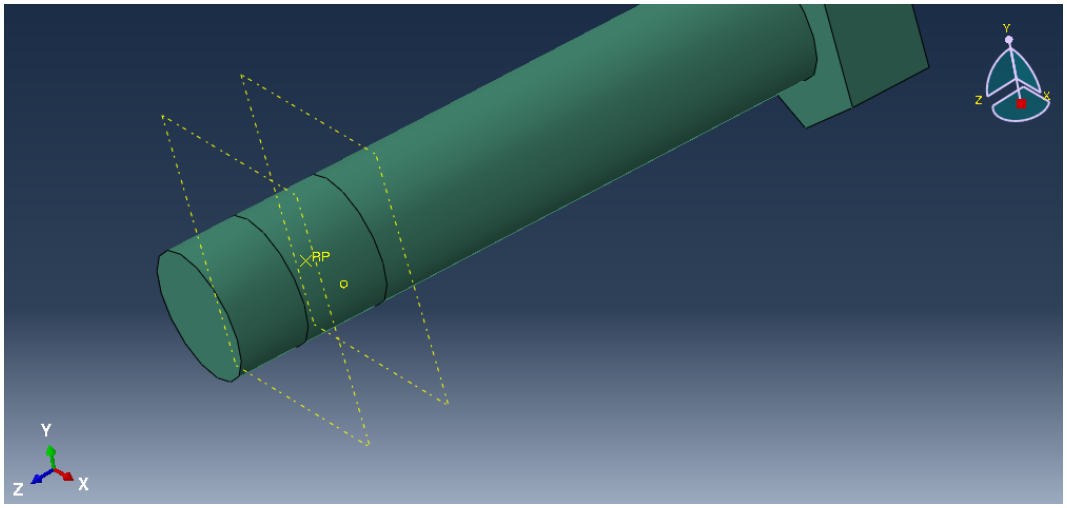
1. Import the two beams and re-orient/move them as follows. **Note that the bolt-axis has to be pointed in the +z direction.** This will be the convention followed throughout this tutorial. The green beam in the following is renamed as `TOPBEAM` and the white beam in the following is renamed as `BOTBEAM`.



2. Import one instance each of the `BOLT` and `NUT` and 2 instances of the `WASHER` and assemble them as shown. Ensure that the washer is oriented (on each side) with the `WSTOP` oriented in the `-z` direction and the `WSBOT` is oriented in the `+z` direction. Rename the washer instances in the top and bottom as `TOPWASHER-1` and `BOTWASHER-1` respectively. Rename the bolt and nut as `NUT-1` and `BOLT-1` respectively.

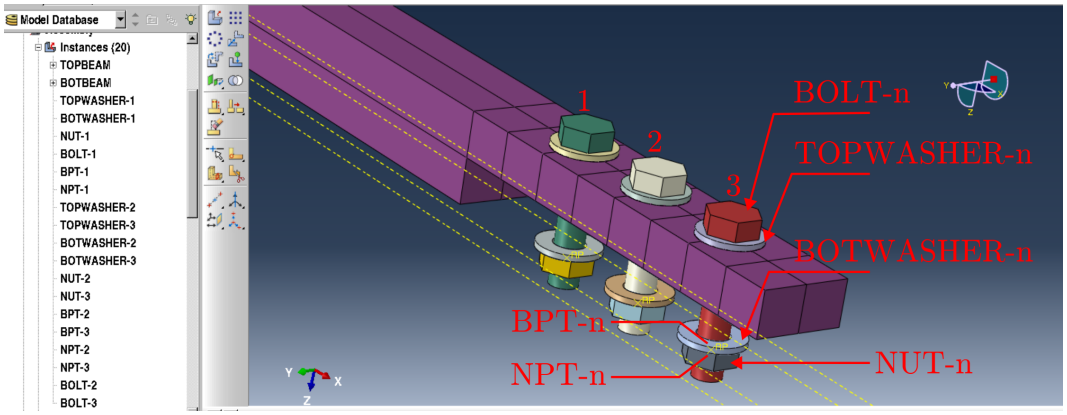


3. Import one instance of the reference point `REFPT`. As mentioned before, this will be used to enforce bolt prestress. This will be achieved by first placing it at the centroid of the intersection of the `BOLT` and `NUT` (surfaces `BN` and `NB`). It is important that the reference point is placed at the centroid. This can be done in the GUI by first moving it to an external point and then translating it along the axis. The figure below shows an example (a datum point was used for this here). Rename this to `BPT-1` (standing for Bolt Point 1).



4. Now import another instance of the reference point `REFPT` and translate it to be coincident with `BPT-1`. Rename this `NPT-1` (standing for Nut Point 1). These two points will have to be on the same geometrical point but equal and opposite forces will be applied for the force application (see 2.2 below for those steps). The point `BPT-1` will be coupled to the bolt surface `BN` and `NPT-1` will be tied to the nut surface `NB` through RBE3 elements.

5. Use the "Linear Pattern" dialog and copy the bolt-washer-washer-nut assembly thrice along the beam such that the assembly is complete. The figure below shows the final assembly along with a representation of the names of the different parts. Recall that the beam shown in the figure below is `TOPBEAM` and the hidden one is `BOTBEAM`.



Now that the assembly is complete, the relevant constraints will have to be enforced, followed by a realization of the bolt preload.

2 Constraints

2.1 Tie-Constraint Enforcement SCRIPT:GUI

All the following operations can be conducted in the `Interaction` module in CAE. But since selecting each surface can be a time-consuming process, we use the following for-loop in Abaqus-python (either call it as a script or just copy paste it into the CLI) to make the required tie constraints. Specifically, it ties the Bolt-Washer, Nut-Washer, and Washer-Beam surfaces. Note that in the last set of constraints, we recall the fact that the bottom beam is flipped. So `WASHER-1` is tied to `BmW3` of the bottom beam (and so on for 2,3).

```
mdl = mdb.models['Model-1']
ras = mdl.rootAssembly

for i in range(1, 4):
    # Bolt-Washer Constraints
    mdl.Tie(name='BW-%d' %(i), main=ras.instances['TOPWASHER-%d' %(i)].surfaces['WSTOP'],
            secondary=ras.instances['BOLT-%d' %(i)].surfaces['BlW'],
            positionToleranceMethod=COMPUTED, adjust=ON, tieRotations=ON, thickness=ON)

    # Nut-Washer Constraints
    mdl.Tie(name='NW-%d' %(i), main=ras.instances['BOTWASHER-%d' %(i)].surfaces['WSBOT'],
            secondary=ras.instances['NUT-%d' %(i)].surfaces['NW'],
            positionToleranceMethod=COMPUTED, adjust=ON, tieRotations=ON, thickness=ON)

    # TopWasher-Beam Constraints
    mdl.Tie(name='BmTW-%d' %(i), main=ras.instances['TOPBEAM'].surfaces['BmW%d' %(i)],
            secondary=ras.instances['TOPWASHER-%d' %(i)].surfaces['WSBOT'],
```

```
18              positionToleranceMethod=COMPUTED, adjust=ON, tieRotations=ON, thickness=ON)
19
20      # BotWasher-Beam Constraints
21      mdl.Tie(name='BmBW-%d' %(i), main=ras.instances['BOTBEAM'].surfaces['BmW%d' %((3-i)%3+1)
22              secondary=ras.instances['BOTWASHER-%d' %(i)].surfaces['WSTOP'],
23              positionToleranceMethod=COMPUTED, adjust=ON, tieRotations=ON, thickness=ON)
```

2.2 Bolt Preload Realization SCRIPT:GUI

You might already have encountered the different parts of the model above that were carefully constructed for the bolt preload application (bolt-shank partitioning, reference points, etc.).

What 's wrong with the inbuilt Bolt Pretension in ABAQUS?

The Abaqus Approach

- You can find the Abaqus documentation for the bolt load here.
- ABAQUS/CAE applies bolt load by specifying a **bolt cross-section**, a **bolt axis**, and the **bolt load**.
- The documentation for the Abaqus implementation can be found here.
- The load is applied in the context of a constraint:
  - It can be a load constraint wherein the displacements/strains at the cross-section are adjusted to match the load.
  - It can be a deformation constraint, wherein the loads are adjusted.
- In either case, the bolt load **can not be written down as a constant load vector** that can be exported/used elsewhere.
- It is therefore not possible to use the inbuilt bolt pretension in a substructured analysis, for example.
- Here are the documented limitations:
  - An assembly load cannot be specified within a substructure.
  - If a submodeling analysis is performed, any pre-tension section should not cross regions where driven nodes are specified.
  - Nodes of a pre-tension section should not be connected to other parts of the body through multi-point constraints.

Our Approach

- Our method of bolt pretension application addresses all the above issues pertaining to substructuring/submodeling by using **Distributed Coupling** elements (aka RBE3/Spider elements in other FE software).
- We will first couple the bolt-shank area that is in contact with the nut (the thread area) to a 6DOF point (3 translations + 3 rotations), and do the same for the nut inner surface, with another point, using **Distributed coupling** elements.
- The bolt and nut will be "fastened" by arresting every DOF in these two nodes except the axial DOF. This will ensure that the only relative displacement between the bolt and nut are axial, which may result from tightening/loosening of the bolt.
- A **"pulling force" is applied on the bolt-coupling node**, which acts as the tension on the bolt, and a **"pushing force" is applied on the nut-coupling node**, which acts to maintain the system's equilibrium, i.e., fastening.
- It is understood that there is an interface that the assembly is tightening, which should provide the required reaction forces to balance everything out.
- Now, the bolt load is merely a constant force vector which can be manipulated as one desires.

We will now go through the process of specifying this.

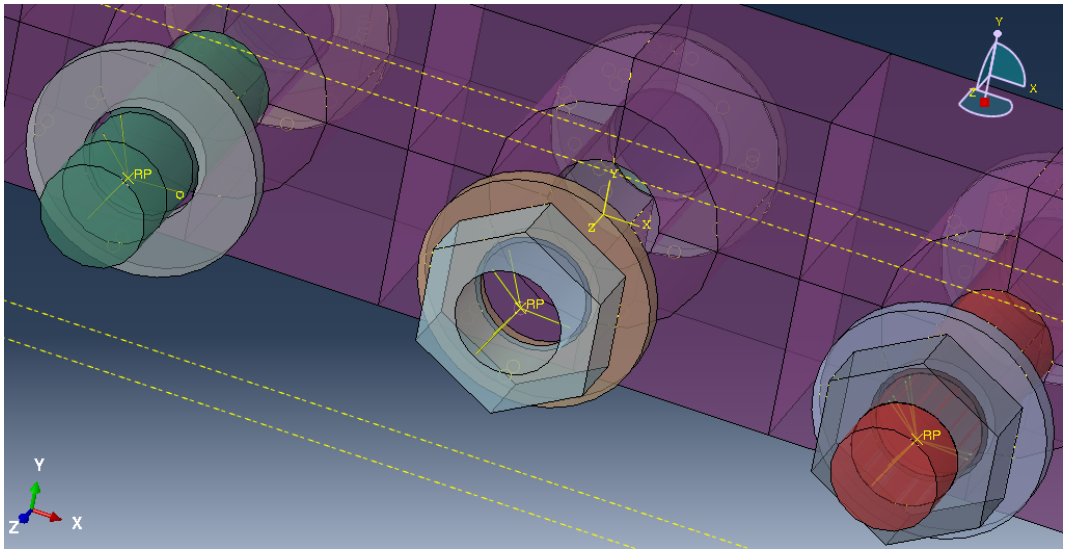
1. First couple the bolt-shank surface `BN` with the appropriate reference point (`BPT-n`). This can be done in CAE through `Interaction->Create Constraint->Tie`. The following is python code that will do this in a loop.

```
24  # Bolt and Nut Point Coupling
25  for i in range(1, 4):
26      mdl.Coupling(name='BPC-%d' %(i), controlPoint=ras.instances['BPT-%d' %(i)].sets['Se
27                   surface=ras.instances['BOLT-%d' %(i)].surfaces['BN'],
28                   influenceRadius=WHOLE_SURFACE, couplingType=STRUCTURAL,
29                   weightingMethod=UNIFORM)
30
31      mdl.Coupling(name='NPC-%d' %(i), controlPoint=ras.instances['NPT-%d' %(i)].sets['Se
32                   surface=ras.instances['NUT-%d' %(i)].surfaces['NB'],
33                   influenceRadius=WHOLE_SURFACE, couplingType=STRUCTURAL,
34                   weightingMethod=UNIFORM)
```

2. Now we constrain the X, Y, Rx, Ry, and Rz DOFs (all 5 DOFs other than the Z DOF, which is the bolt-axis) using equation constraints. If the bolt axis is not a principal direction in the model, then the constraint equations must be modified appropriately. Note that this can also be used in the large deformation context through the use of a local coordinate system (the global CS is used here, so **applicability is restricted to small deformations**). This can be done in CAE through `Interaction->Create Constraint->Coupling`. The following code does this through a nested loop such that `BNEQn-m` **represents the m^{th} DOF constraint at the n^{th} location**.

```
35  # Equation Constraints constraining bolt and nut ref-points to each other
36  for i in range(1, 4):
37      for j in [1, 2, 4, 5, 6]:
38          mdl.Equation(name='BNEQ%d-%d' %(i, j),
39                       terms=((1.0, 'BPT-%d.Set-1' %(i), j),
40                              (-1.0,'NPT-%d.Set-1' %(i), j)))
```

Here is an image showing the constraints applied graphically.



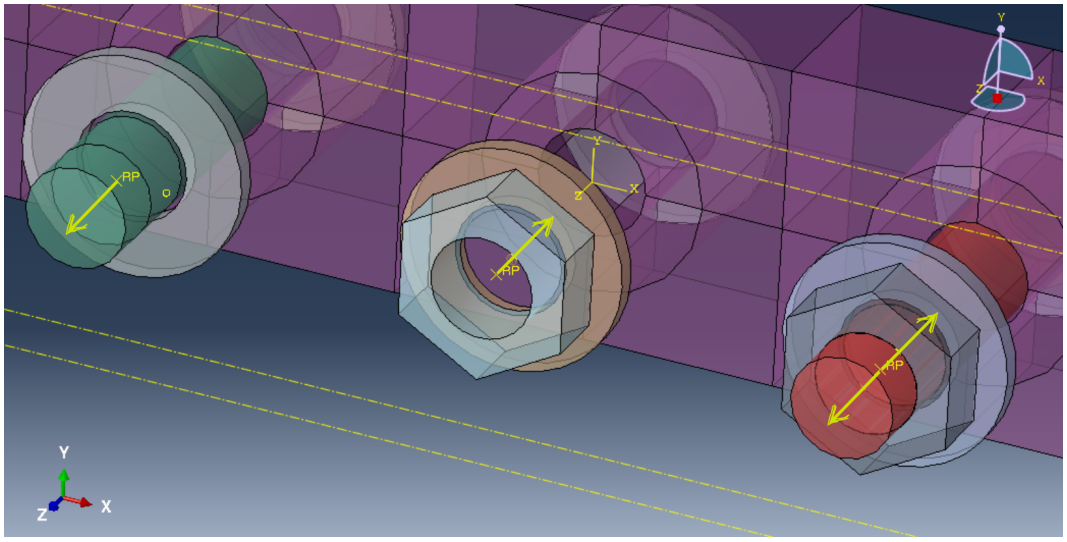
3. We next create a static analysis step (`Step->Create Step->Static, General`) named as `PRESTRESS`. This can be done in CAE, but here is the Python code.

```
41  mdl.StaticStep(name='PRESTRESS', previous='Initial', nlgeom=ON)
```

Note: Nonlinear Geometric effects (nlgeom) is set to ON since this was found to be helpful for convergence of prestress analysis.

4. We finally apply bolt forces as concentrated forces at the `BPT-n` and `NPT-n` reference points. Switch to the `Load` module to do this from CAE. The following python code applies a tightening load of 12kN to each bolt.

```
42  # Apply forces
43  bpmag = 12e3  # 12kN bolt-load
44  for i in range(1, 4):
45      # Force in +z on bolt-points
46      mdl.ConcentratedForce(name='BoltLoad-%d' %(i), createStepName='PRESTRESS',
47                            region=ras.instances['BPT-%d' %(i)].sets['Set-1'],
48                            cf3=bpmag, distributionType=UNIFORM)
49
50      # Force in -z on bolt-points
51      mdl.ConcentratedForce(name='NutLoad-%d' %(i), createStepName='PRESTRESS',
52                            region=ras.instances['NPT-%d' %(i)].sets['Set-1'],
53                            cf3=-bpmag, distributionType=UNIFORM)
```



5. **Note** that the bolt load constructed in the above manner is a "linear" load - i.e., the load can be increased/decreased by scaling the resulting force vector. It is assumed here that all 3 bolts are loaded equally. If this is not the case, the different loads can be exported separately and scaled appropriately (for external analysis). It is, however, a physical requirement for equilibrium that `BoltLoad-n` and `NutLoad-n` have to be of equal magnitude and opposite signs.

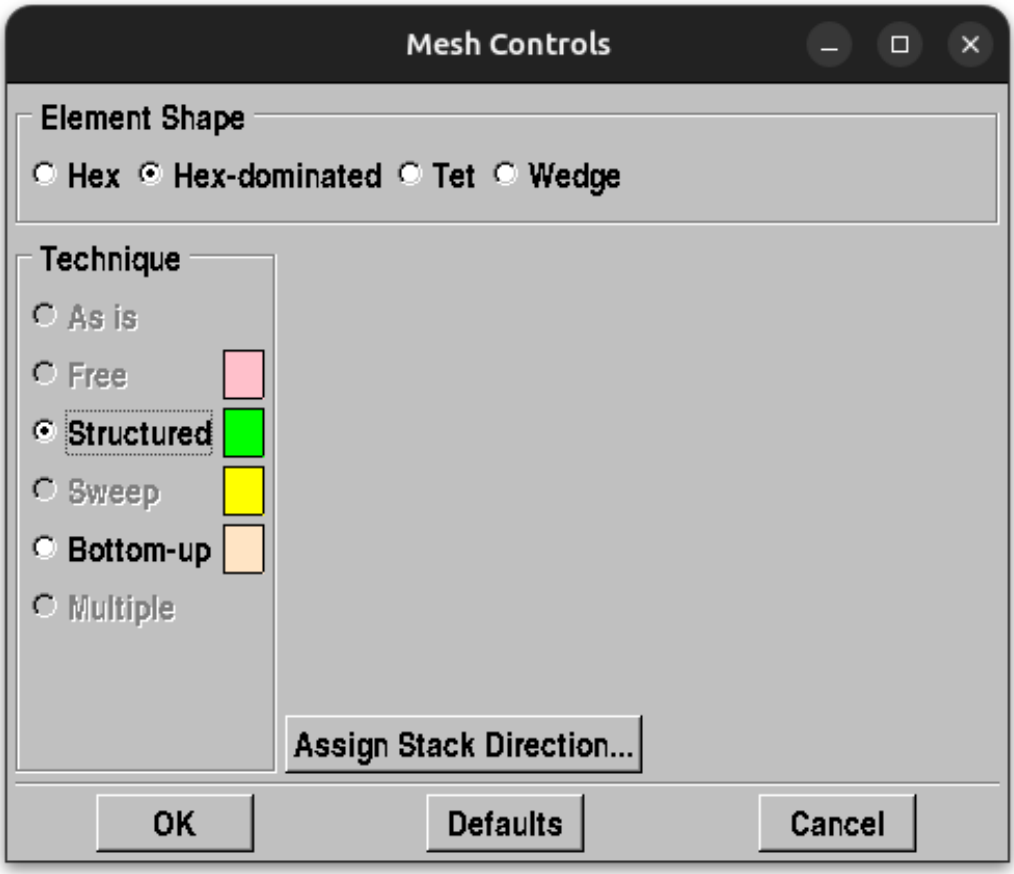
The file $\text{model}_{\text{step1.cae}}$ stores the CAE-file generated after the end of the above steps. We will now proceed with meshing.

3 Meshing

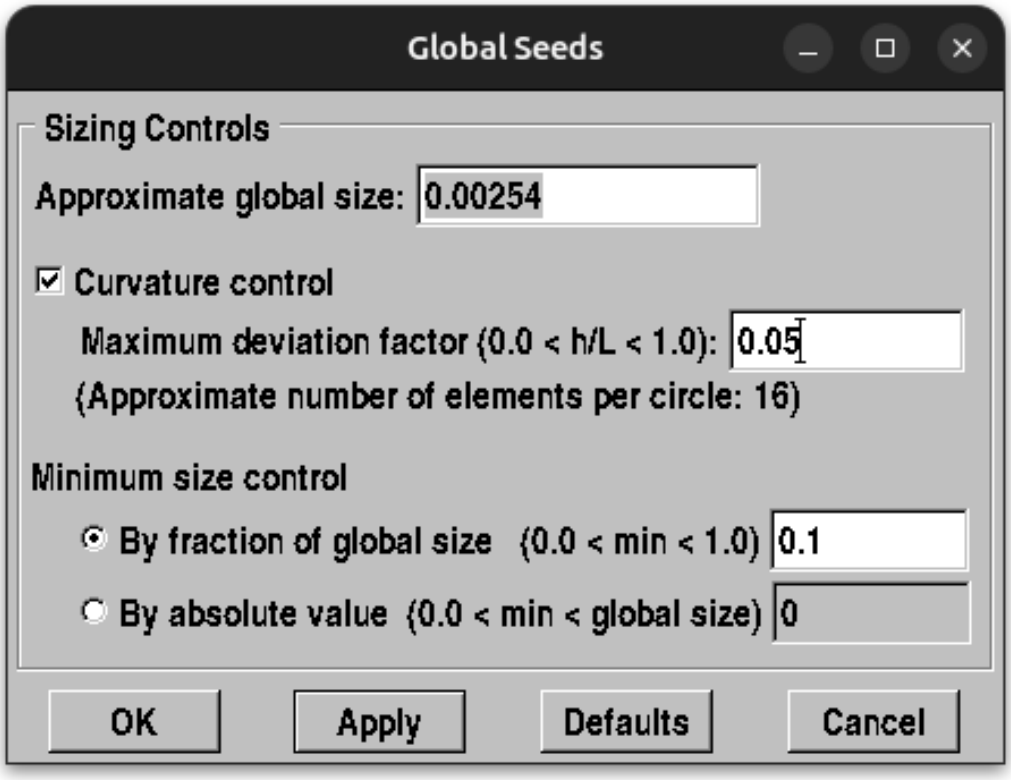
Since the model is perfectly symmetric, it is sufficient to mesh the model with a global seed, after seeding the interface area locally. Switch to the `Mesh` module for this section.

3.1 Meshing Instructions

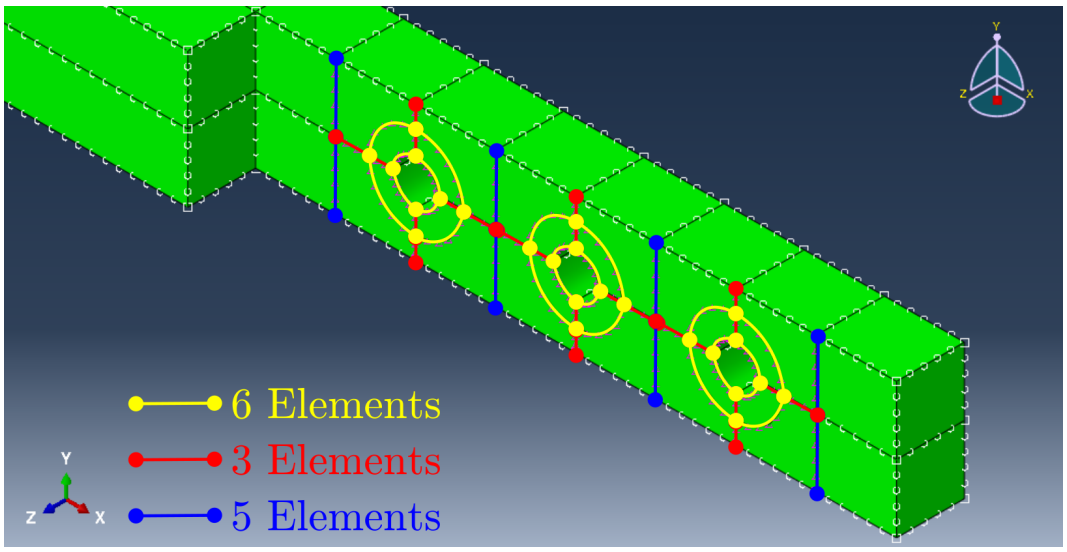
1. Choose `Mesh->Assign Mesh Controls` and select the whole assembly (all the instances). We will use a structured hex-dominated element type for the full model (it will give a warning that this is not possible for certain regions. This is okay).



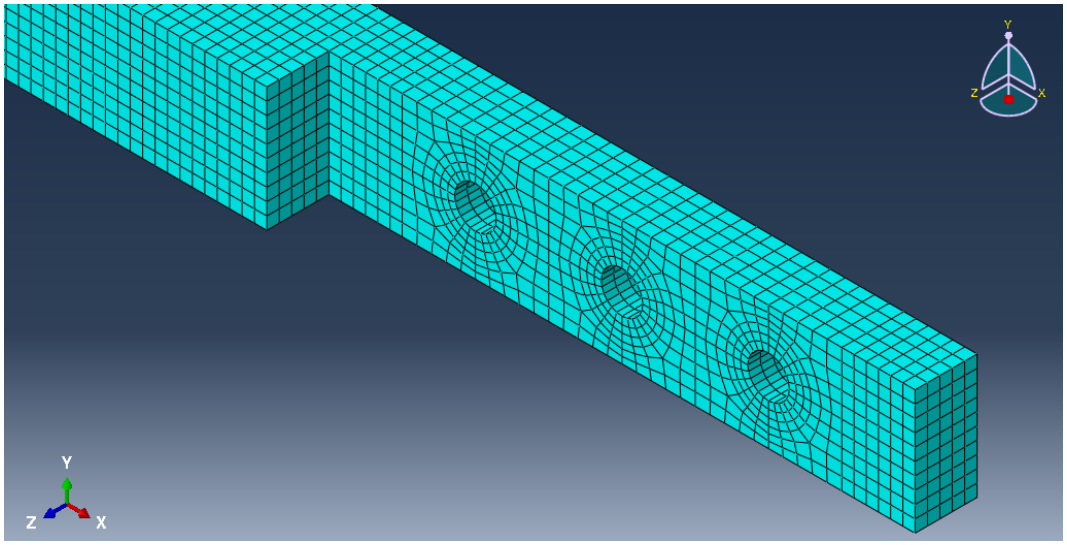
2. Now choose `Mesh->Seed Part Instance` and select all the instances again. Set 0.00254 as the global mesh seed (to ensure 10 elements across thickness) and 0.05 for curvature control.



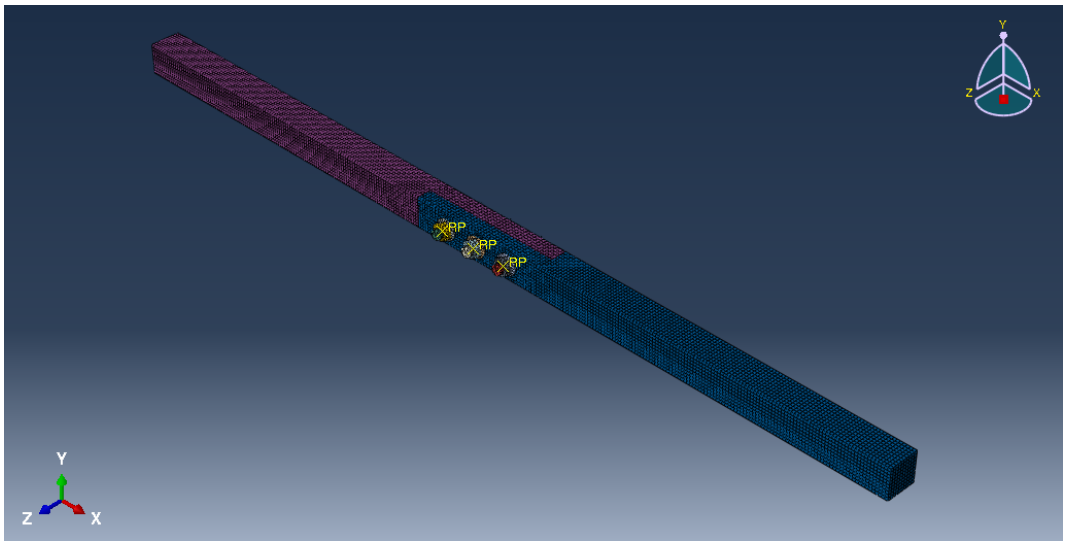
3. Apply local seeds at the interface as shown here (on both interfaces). Use the `Mesh->Seed Edges` tool by choosing the Method as "By Number" and specify the number of elements.



4. Now mesh the assembly using `Mesh->Mesh Part Instance` and selecting all the part instances. Here is a view of the interfacial mesh you should get after the above seeding.



5. Here is an image of the total assembly with the mesh.



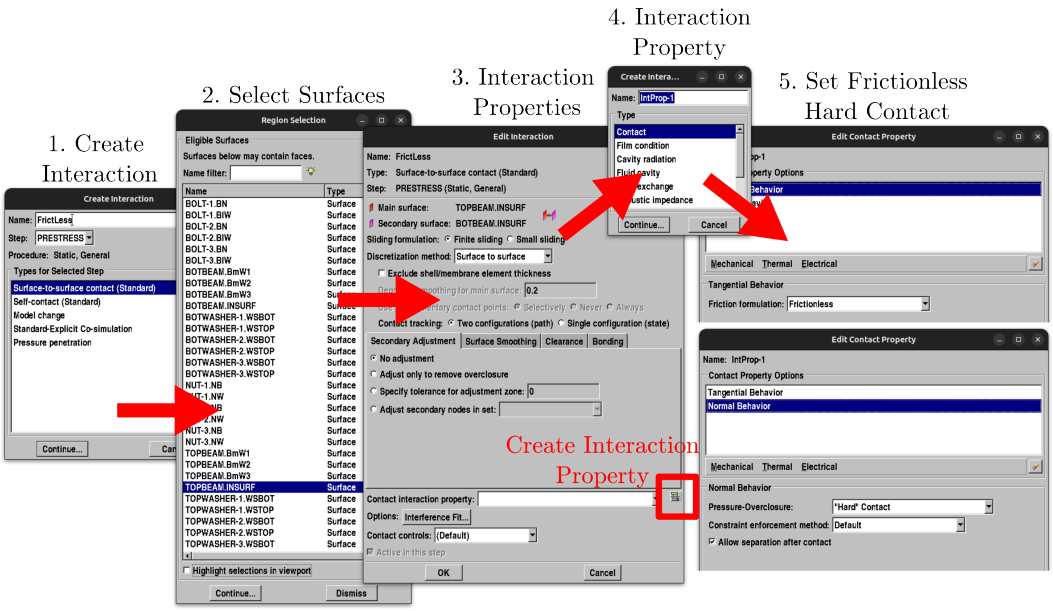
The file $\text{model}_{\text{step1.cae}}$ stores the CAE-file along with the mesh.

3.2 (Optional, recommended) Verify correctness of model through frictionless prestress

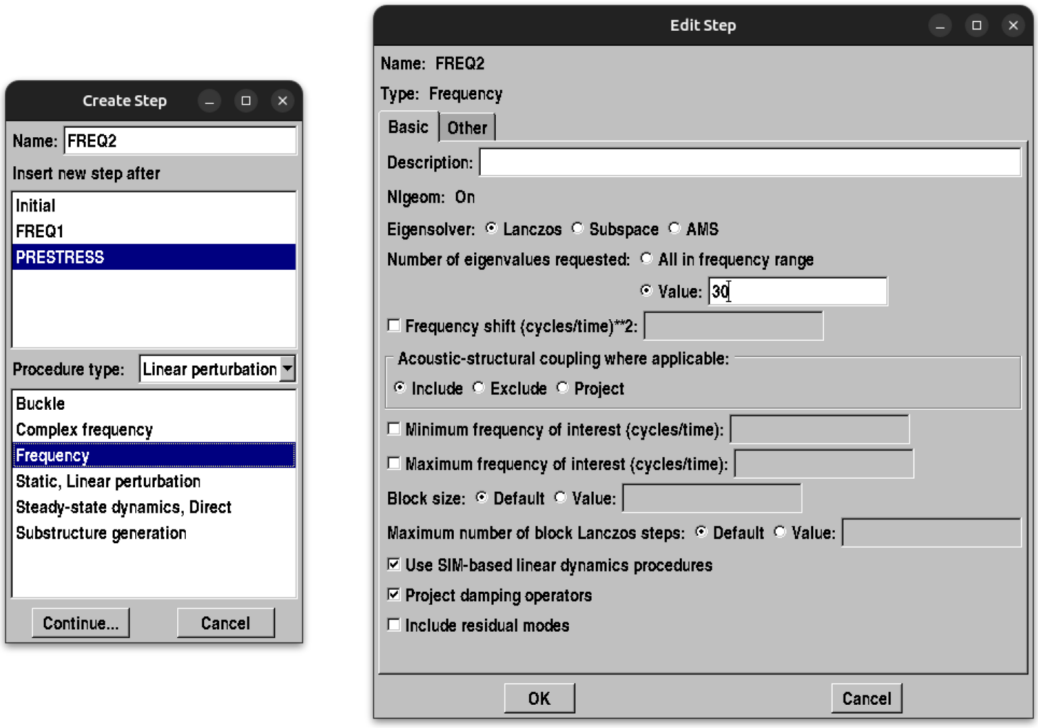
We will now conduct a simple frictionless contact analysis to verify the correctness of the model.

Setup

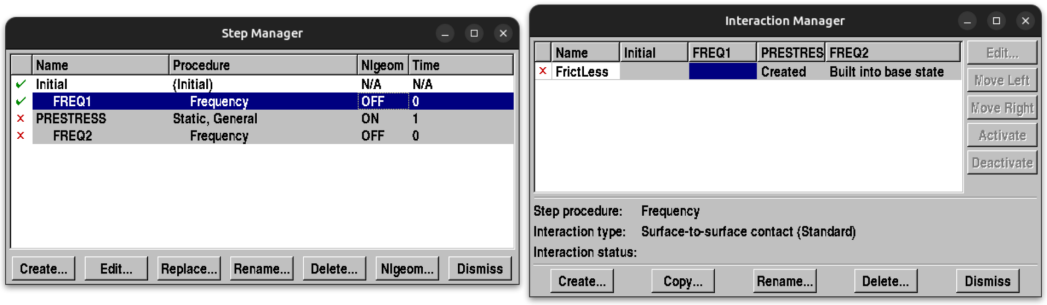
1. For the static prestress analysis, first a surface-to-surface contact interaction property has to be created and assigned. You can do this from CAE starting from `Interaction->Create Interaction` and following the steps in the figure below.



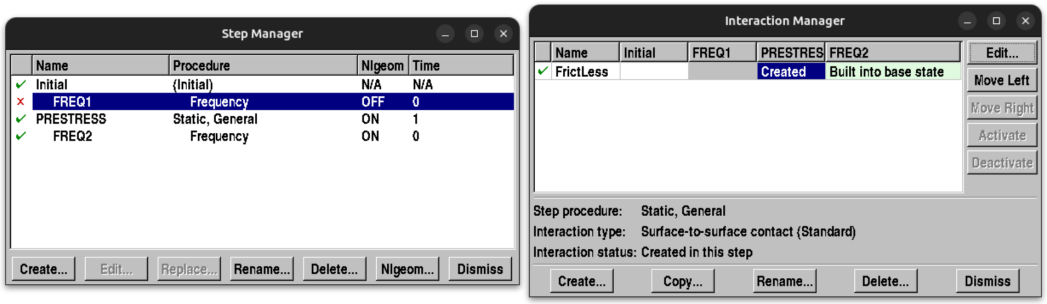
2. Now create two linear frequency steps, one before and one after the static prestress step, as shown in the figure below. Request 30 eigenvalues in each case.



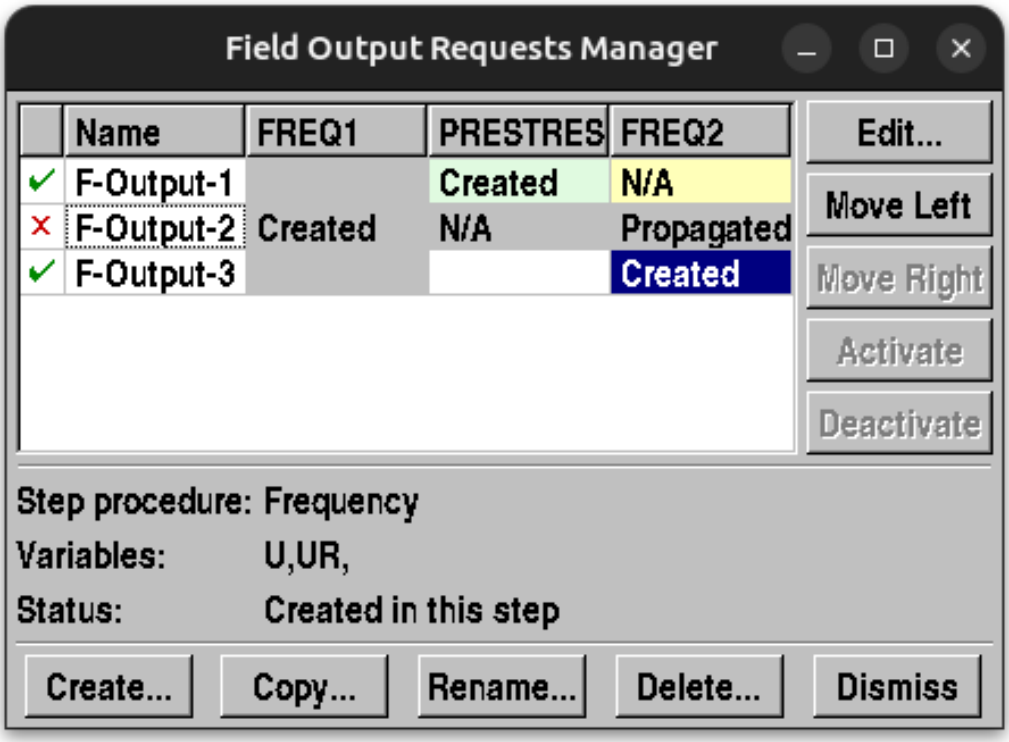
3. Create 2 jobs. Suppress the interaction property in the first one and have just `FREQ1` active (see figure below).



For the second job, suppress `FREQ1` and resume the other steps. Resume the surface interaction properties (see figure below).



Optionally, field outputs for `FREQ2` (by default it will be set to none if `FREQ1` is suppressed). You can do this in `Step->Create Field Outputs`.

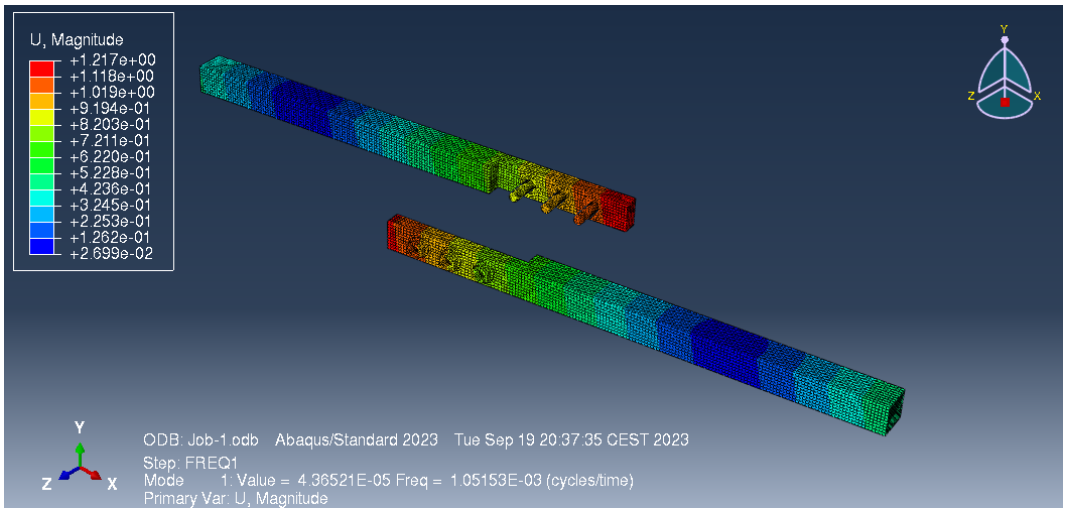


Results

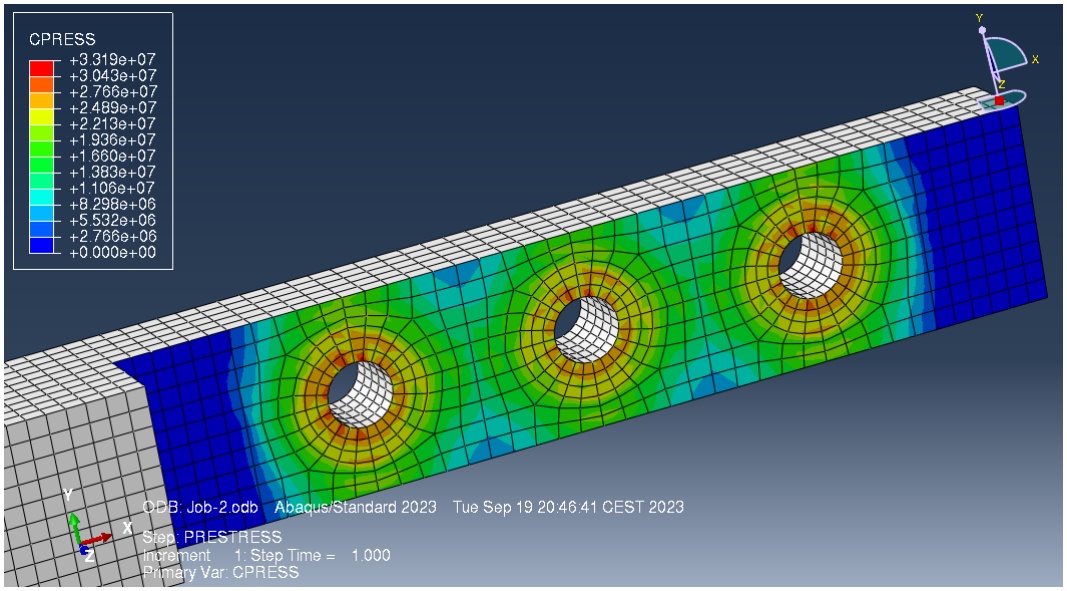
1. The first analysis should reveal that the system has **7 Rigid Body Modes** (RBMs). This is because the two beams are constrained together in all directions except the axial (where tightening has to happen). So the number of RBMs has to be $2 \times 6 - 5 = 7$. The first 10 modal frequencies have to be (frequencies in cycles/time, or Hz):

Index	Frequency
1	1.0515e-3
2	1.2851e-3
3	1.4602e-3
4	1.6246e-3
5	1.6592e-3
6	1.8082e-3
7	1.9025e-3
8	70.323
9	151.26
10	609.25

   In the above, modes 1-7 are RBMs and modes 8 onwards are the elastic modes. Looking at the mode-shapes should make it clear that the two beams are free to move axially (the following is mode 1, for eg).



2. The second analysis should converge within a few iterations. If not, the "Initial Increment Size" in `Step->PRESTRESS->Edit->Increment->Initial Increment Size` has to be reduced. If it doesn't converge, try to apply a boundary condition to make the problem well-posed, and try again. If it still doesn't converge, check your model again. Here is a picture of the contact pressures at the interface at the end of the static prestress step.



3. By default, ABAQUS **fuses the nodes** that are in contact at the end of the hard contact step, for the eigenvalue analysis that follows it (Linear Perturbation step). Since we are using frictionless tangential here, the tangential DOFs are not fused. Here are the first 10 frequencies from the `FREQ2` step (frequencies in cycles/time, or Hz):

Index	Frequency
1	0.00
2	0.00
3	0.00
4	0.00
5	0.00
6	2.2068e-3
7	141.10
8	152.42
9	570.33
10	643.49

   It must be observed that the model, under the prestressed state, **has only 6 RBMs**. In other words, the bolt-axial direction has now been fixed due to the fact that the contact constraints are active on at least one spot on the interface.

4. If your model passes all the above, then you are ready to proceed. **INCLUDE NOTE ABOUT MESH CONVERGENCE?**

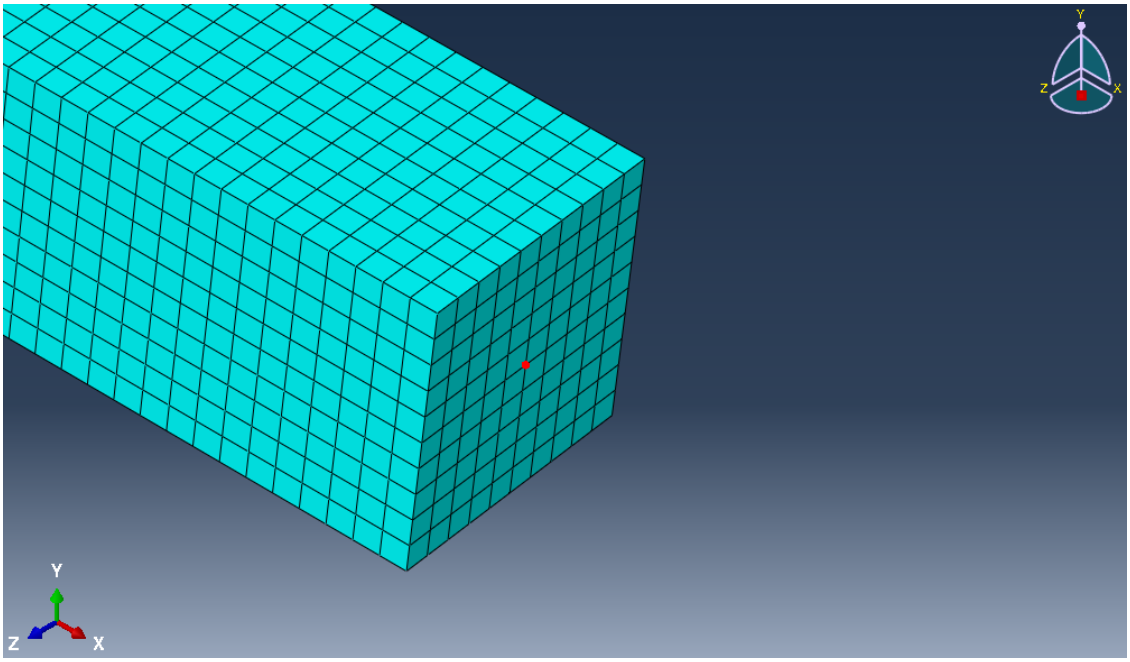
The file $\text{model}_{\text{step2.cae}}$ is the cae file containing the model with the above tests included.

4 Postprocessing

Now that the model is verified to be correct/consistent, we will proceed with the steps necessary for substructured Matrix Extraction.

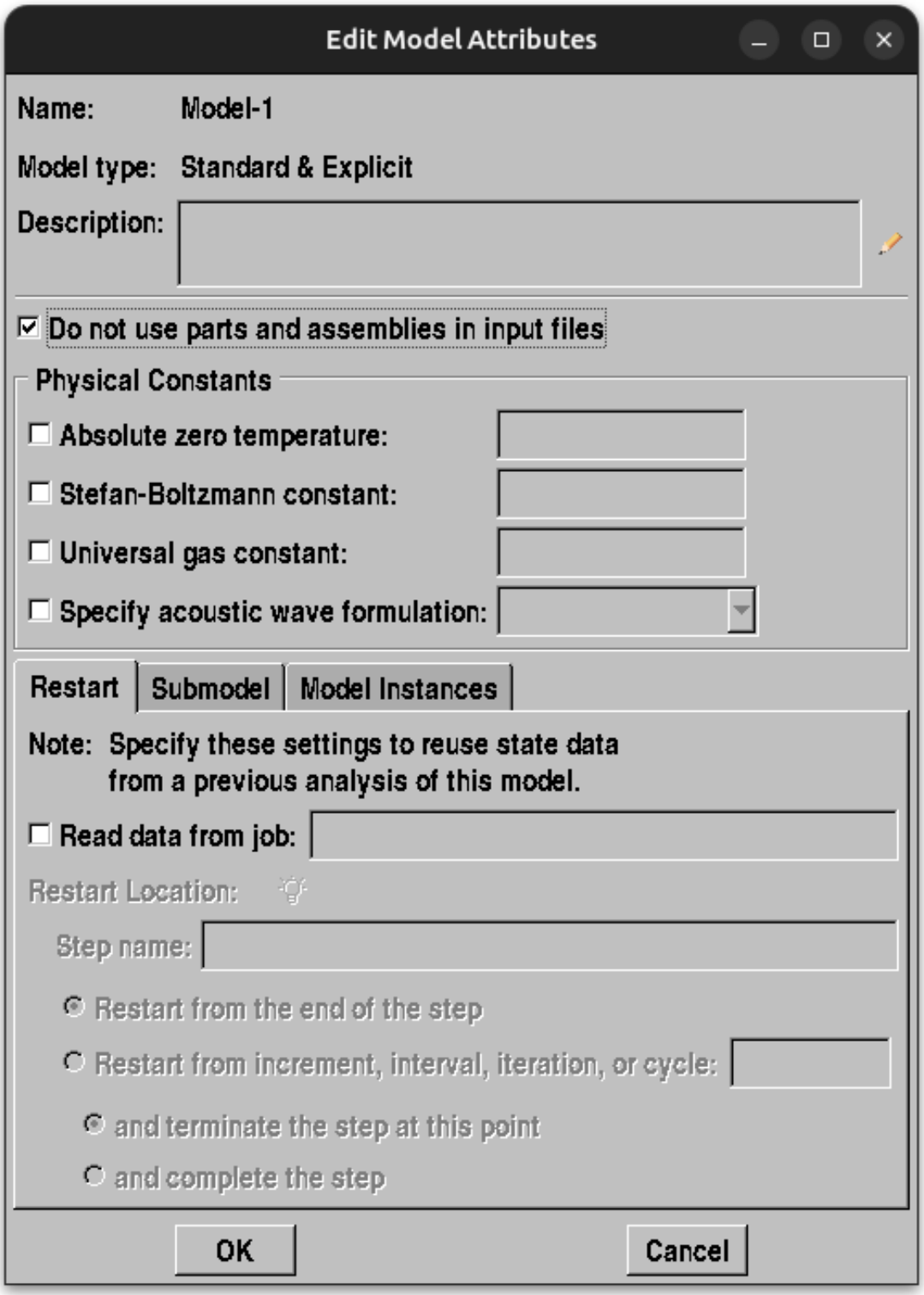
Before proceeding it is necessary to first identify nodes on the meshed model that correspond to the input and output locations. Choose the mid point on the right end as the output node for this

tutorial. Create a node set called `OutNodes` by selecting `Mesh->Tools->Set->Create->Node` and choosing the node. Select "unsorted node set" if available.



It is possible to choose multiple nodes here if you have a MIMO case.

It is also helpful to use global node and element indexing henceforth. This can be specified in `Model->Edit Attributes->Model-1` as follows:



4.1 Reorganize Interfacial Node Sets (readjust if necessary) SCRIPT

Although we have taken care to ensure that the mesh of the `TOPBEAM` and `BOTBEAM` are conformal at the interface, minor imperfections in the nodal locations may exist. Furthermore, the ordering

of the nodes on the top interface will be different from that on the bottom interface. Through some scripting, we can create node sets in such a way that the top and bottom interface nodesets are ordered in a convenient fashion.

1. We use the following slightly modified header for this script:

```
 1  # -*- coding: utf-8 -*-
 2  # 1. Preamble
 3  import sys
 4  import numpy as np
 5
 6  from part import *
 7  from material import *
 8  from section import *
 9  from assembly import *
10  from step import *
11  from interaction import *
12  from load import *
13  from mesh import *
14  from optimization import *
15  from job import *
16  from sketch import *
17  from visualization import *
18  from connectorBehavior import *
19
20  from abaqus import *
21  from abaqusConstants import *
22  from caeModules import *
23  import regionToolset
24  import job
25  import step
26  import sets
27
28  mdl = mdb.models['Model-1']
29  ras = mdl.rootAssembly
30
31  mdl.setValues(noPartsInputFile=ON)
```

2. Now we identify the top and bottom surface nodes, and store the top nodes and elements into separate variables. The node and element ordering of the top nodes will be preserved, and the nodes in the bottom will be resorted according to this in #3 below.

```
32  # 2. Get top and bottom surfaces and nodes
33  topsurf = ras.instances['TOPBEAM'].surfaces['INSURF']
34  botsurf = ras.instances['BOTBEAM'].surfaces['INSURF']
35
36  topnodes = topsurf.nodes
37  botnodes = botsurf.nodes
```

```
38 N = len(topnodes)  # Number of nodes
39
40 # Top Nodes and Coordinates
41 Topnd_dict = dict(zip([topnodes[i].label for i in range(N)], range(N)))
42 # maps original node ID (in FE model) to node ID in interface node set
43 TopNdCds = np.array([topnodes[i].coordinates for i in range(N)])
44
45 # Top Elements
46 TopEls = np.array([topsurf.elements[i].connectivity for i in range(len(topsurf.elements
47 ELS = np.zeros((TopEls.shape[0], 5), dtype=int)
48 for ne in range(TopEls.shape[0]):
49     elefac = topsurf.elements[ne].getElemFaces()
50
51     # Gives you the list of faces on the interface (we only expect a single face)
52     fe = np.argwhere([all([Topnd_dict.has_key(x) for x in
53                            [elefac[k].getNodes()[i].label for i in range(4)]])
54                       for k in range(6)])[0,0]
55     ELS[ne, 0] = ne
56     # Searches for the face where all the nodes are in the interface and returns those
57     ELS[ne, 1:] = [Topnd_dict[x] for x in [elefac[fe].getNodes()[k].label
58                                            for k in range(4)]]
59     ELS[ne, :] += 1
60
61 # Save interfacial nodes and elements to txt files
62 np.savetxt('Nodes.dat', TopNdCds) # Save to dat file
63 np.savetxt('Elements.dat', ELS, fmt='%d')
```

3. Now we extract the bottom nodes and sort them.

```
64 # 3. Node Pairing. We assume len(botnodes)=len(topnodes).
65 botleft = range(N)
66 bts = []
67 tmi = 0
68 for i in range(N):
69     # Calculates deviation of selected node coordinate on bottom to each
70     # node coordinate on top and "assigns" the closest one to the index.
71     bts.append(
72         botleft.pop(
73             np.argmin(
74                 np.linalg.norm(
75                     topnodes[i].coordinates-np.array([botnodes[j].coordinates for j in
76                     axis=1)
77             )
78         )
79     )
```

4. We now adjust the nodes on the bottom (this affects the FE mesh directly!)

```
80 # 4. Adjust Nodes on Bottom Beam Interface to Match Top Beam Exactly
```

```
for i in range(N):
    ras.editNode(nodes=botnodes[bts[i]:bts[i]+1],
                 coordinates=(topnodes[i].coordinates,))
```

5. We now create nodesets for the top (`TOPS_NDS`) and bottom (`BOTS_NDS`).

```
# 5. Create Node Sets
botpairednds = botnodes.sequenceFromLabels(tuple([botnodes[i].label for i in bts]))
# Reordering from the sorting above

ras.SetFromNodeLabels(name="TOPS_NDS",
                      nodeLabels=((topnodes[0].instanceName,
                                   tuple([topnodes[i].label for i in range(N)])),),
                      unsorted=True)
ras.SetFromNodeLabels(name="BOTS_NDS",
                      nodeLabels=((botpairednds[0].instanceName,
                                   [botpairednds[i].label
                                    for i in range(len(botpairednds))]),),
                      unsorted=True)
```

Note that we've used the unsorted keyword to ensure that ABAQUS does not reorder the nodesets (default behavior).

6. We now simplify the model by removing all interaction properties and steps (except initial). One side-effect of doing this is that this removes the bolt loads also, since loads can only be saved when there is a corresponding step! So we will reintroduce the bolt loads in the substructuring step in #8 below.

```
# 6. Simplify model (remove interactions, all steps, etc.)
tmp = mdl.interactions
while len(tmp) > 0:
    del tmp[tmp.keys()[-1]]

tmp = mdl.interactionProperties
while len(tmp) > 0:
    del tmp[tmp.keys()[-1]]

# Remove all steps except initial
tmp = mdl.steps
while len(tmp) > 1:
    del tmp[tmp.keys()[-1]]
```

4.2 Setup Fixed interface CMS (HCB-CMS) with linear frequency and sub-structure steps SCRIPT

1. We are interested in **Fixed-Interface Component Mode Synthesis** with 20 retained modes. So we first do a fixed interface modal analysis and request $20 \times 3 = 60$ modes (for ensuring accuracy of the first 20 modes). We fix all the nodes in the nodesets `TOPS_NDS` and `BOTS_NDS`.

```
# 7. Create a Frequency Step for fixed interface modal analysis
mdl.FrequencyStep(name="Fixed-Int-Modal", previous="Initial",
                  normalization=MASS, eigensolver=LANCZOS,
                  numEigen=60)
mdl.EncastreBC(name="TOPFIX", createStepName="Fixed-Int-Modal",
               region=ras.sets['TOPS_NDS'])
mdl.EncastreBC(name="BOTFIX", createStepName="Fixed-Int-Modal",
               region=ras.sets['BOTS_NDS'])
```

2. Next, we create the substructuring step and re-specify the bolt loads (these were removed in #6 above). The bolt loads are specified of unit magnitude and can be scaled during analysis. **This by default uses the mode shapes from the previous step and calculates static constraint modes automatically.**

```
# 8. Create a substructuring step, specify the modes and retained DOFs
mdl.SubstructureGenerateStep(name="HCBCMS", previous="Fixed-Int-Modal",
                             substructureIdentifier=1,
                             retainedEigenmodesMethod=MODE_RANGE, modeRange=((1, 20, 1
                             recoveryMatrix=REGION, recoveryRegion=ras.sets['OutNodes'
                             computeReducedMassMatrix=True)

mdl.RetainedNodalDofsBC(name="TOPRET", createStepName="HCBCMS",
                        region=ras.sets['TOPS_NDS'],
                        u1=ON, u2=ON, u3=ON)
mdl.RetainedNodalDofsBC(name="BOTRET", createStepName="HCBCMS",
                        region=ras.sets['BOTS_NDS'],
                        u1=ON, u2=ON, u3=ON)

# Apply Bolt Loads (1N magnitude)
for i in range(1, 4):
    mdl.ConcentratedForce(name='BoltLoad-%d' %(i), createStepName="HCBCMS",
                          cf3=1.0, region=ras.instances['BPT-%d' %(i)].sets['Set-1'])
    mdl.ConcentratedForce(name='NutLoad-%d' %(i), createStepName="HCBCMS",
                          cf3=-1.0, region=ras.instances['NPT-%d' %(i)].sets['Set-1'])

sbs = mdl.steps['HCBCMS']
sbs.LoadCase(name="LCASE", loads=tuple(('BoltLoad-%d' %(i), 1.0) for i in range(1, 4))
             tuple(('NutLoad-%d' %(i), 1.0) for i in range(1, 4)))
```

3. We finally need to request matrix output. Here is the ABAQUS documentation for the `*Substructure Matrix Output` card. Thus far (until version 2023) ABAQUS CAE doesn't support requesting matrix output from Substructure steps. We have to request this manually in the .inp files. Thankfully we can write to the keywords directly from CAE. Go to `Model->Edit Keywords->Model-1` to do this in the GUI. In scripting, this is known as a "synch" operation, which is done as follows:

```
# 9. Request substructure matrix outputs
# ABAQUS CAE doesn't support this yet (GUI or scripting),
```

```
144  # so the keywords need to be manually modified.
145  mdl.keywordBlock.synchVersions(storeNodesAndElements=False)
146
147  li = np.argwhere([mdl.keywordBlock.sieBlocks[i][0:20] == "*Retained Nodal Dofs"
148                    for i in range(len(mdl.keywordBlock.sieBlocks))])[0][0]
149  txi = mdl.keywordBlock.sieBlocks[li]
150  mdl.keywordBlock.replace(li, "*Retained Nodal Dofs, sorted=NO"+txi[20:])
151
152  mdl.keywordBlock.insert(len(mdl.keywordBlock.sieBlocks)-2,
153                          "*Substructure Matrix Output, FILE NAME=Modelmats, MASS=YES, S
```

4. We then create a job and write it to an "inp" file that can be run from ABAQUS.

```
154  # 10. Create a job and write an inp file
155  mdb.Job(name="Job", model='Model-1')
156  mdb.jobs['Job'].writeInput()
```

5. Here is what the final HCBCMS step looks like in this case. The most important part is the `*Substructure Matrix Output` card in the bottom, which we introduced through the "synch" step in #3 above. Here is the ABAQUS documentation for the `*Substructure Matrix Output` card.

```
** ----------------------------------------------------------------
**
** STEP: HCBCMS
**
*Step, name=HCBCMS, nlgeom=NO
*Substructure Generate, overwrite, type=Z1, recovery matrix=YES, nset=OutNodes, mass matrix
*Select Eigenmodes, generate
1, 20, 1
*Damping Controls, structural=COMBINED, viscous=COMBINED
*Retained Nodal Dofs, sorted=NO
BOTS_NDS, 1, 3
TOPS_NDS, 1, 3
**
** LOAD CASES
**
*Substructure Load Case, name=LCASE
** Name: BoltLoad-1   Type: Concentrated force   Scale factor: 1
*Cload
BPT-1_Set-1, 3, 1.
** Name: BoltLoad-2   Type: Concentrated force   Scale factor: 1
*Cload
BPT-2_Set-1, 3, 1.
** Name: BoltLoad-3   Type: Concentrated force   Scale factor: 1
*Cload
BPT-3_Set-1, 3, 1.
** Name: NutLoad-1   Type: Concentrated force   Scale factor: 1
```

```
*Cload
NPT-1_Set-1, 3, -1.
** Name: NutLoad-2   Type: Concentrated force   Scale factor: 1
*Cload
NPT-2_Set-1, 3, -1.
** Name: NutLoad-3   Type: Concentrated force   Scale factor: 1
*Cload
NPT-3_Set-1, 3, -1.
*Substructure Matrix Output, FILE NAME=Modelmats, MASS=YES, STIFFNESS=YES, SLOAD=YES, RECOV
*End Step
```

4.3 Run Job and Generate the mtx file. SCRIPT:GUI

You can now run the Job named "Job" in the GUI, or directly run the "Job.inp" file from the command line using

```
abaqus job=Job inp=Job.inp cpus=2 interactive
```

Once the job is done, it will output the matrix file "Modelmats.mtx" into the working directory. This, along with the "Nodes.dat" and "Elements.dat" files that were written out earlier, are all we need for external analyses.

5 Matrix Extraction

By default, ABAQUS uses the matrix market format for the outputted matrices. Note the following, in terms of format:

- Each line that starts with an asterix ("*") is a comment.
- The linear matrices of the current model are fully symmetric.
- So only the upper triangular parts of the matrices are exported.
- The load vector is exported as a vector.
- Columns of the recovery matrix R, defined by

$$x_{out} = Rx_{cms}$$

  where x_{cms} is the vector of DOFs of the CMS model (substructure) and x_{out} are the output DOFs. Each column of R is $N_{out} \times 1$, and R is $N_{out} \times N_{cms}$.

5.1 Postprocessing Exported Matrices SCRIPT

1. The first few lines provide the generalized coordinates of the model. The nodes are listed as positive integers and the "modal" DOFs are listed as negative integers. This is followed by a list of DOFs active in each set.
2. The STIFFNESS and MASS matrices are respectively prepended by

```
*     MATRIX,TYPE=STIFFNESS
      <Stiffness matrix entries in upper triangular form>

*     MATRIX,TYPE=MASS
      <Mass matrix entries in upper triangular form>
```

3. The load vector (the bolt prestress load, here) is written as,

```
** SUBSTRUCTURE LOAD CASE VECTOR. SLOAD CASE <name>
***CLOAD
** 1, 1, <entry>
** 1, 2, <entry>
** 1, 3, <entry>
      .
      .
      .
```

4. Finally, the recovery matrix is provided row-by-row as

```
*     SUBSTRUCTURE RECOVERY VECTOR CORRESPONDING TO RETAINED DOFS NUMBER 1
      <entries>
*     SUBSTRUCTURE RECOVERY VECTOR CORRESPONDING TO RETAINED DOFS NUMBER 2
      <entries>
      .
      .
      .
```

5. In the actual file, the order is STIFFNESS, LOAD, MASS, then SUBSTRUCTURE.

6. The following Bash script processes the output (a stiffness, a mass, a load case, and recovery entries are expected)

```
#!/bin/sh

if [ $# = 2 ]
then
    echo "Correct call!"
    OUT=$2
elif [ $# = 1 ]
then
    echo "Acceptable call!"
    a="$1"
    OUT="${a%.*}.mat"
else
    echo "Wrong call - quitting!"
fi
echo "Preprocessing mtx files"
gawk 'BEGIN{mstart=0;} ($1~/^\*M/){mstart++; next} (mstart==1){if($1!~/^\*/){print}else{exi
```

```
gawk 'BEGIN{mstart=0;} ($1~/^\*M/){mstart++; next} (mstart==2){if($1!~/^\*/){print}else{exi
gawk 'BEGIN{vstart=0;} ($0~/^\*\*\*C/){vstart++; next} (vstart>0){print $2,$3,$4} (vstart>0
gawk 'BEGIN{rstart=0} ($0~/^\*\* SUBSTRUCTURE REC/){rstart++; if(rstart>1){printf("\n")}nex
echo "Preprocessing mtx files done"

python <<EOF
import numpy as np
import scipy.io as io

print("Reading Mass Matrix from mtx file.");
Mv = np.loadtxt('.MASS.mtx');
print("Done.");

print("Reading Stiffness Matrix from mtx file.");
Kv = np.loadtxt('.STIFFNESS.mtx');
print("Done.");

print("Reading Recovery Matrix from mtx file.");
R = np.loadtxt('.RECOV.mtx');
print("Done.");

print("Processing Matrices.")

Nelm = len(Mv);
Nelk = len(Kv);
if (Nelm!=Nelk):
        sys.exit("GIGO - Mass & Stiffness not of same length.");
Nel = Nelm;

Nd = ((np.sqrt(1+8*Nel)-1)/2).astype(int); # Solution of Nd(Nd+1)/2-Nel = 0

M = np.zeros((Nd,Nd));
K = np.zeros((Nd,Nd));

(xi,yi) = np.tril_indices(Nd);
M[xi,yi] = Mv;
M[yi,xi] = Mv;
K[xi,yi] = Kv;
K[yi,xi] = Kv;

print("Done.")

print("Reading Forcing Vector from mtx file.");
Fvdat = np.loadtxt('.FVEC.mtx');
print("Done.");

print("Processing Force Vector.");
Fv = np.zeros(M.shape[0]);
```

```
ids = range(np.where(np.diff(Fvdat[:, 1])==0)[0][0], Fvdat.shape[0])
n1dofnds = Fvdat[ids, 0].astype(int)
n3dofnds = Fvdat[list(set(range(Fvdat.shape[0]))-set(ids)), 0].astype(int)
Fv[((n3dofnds-1)*3+np.kron(np.ones(int(len(n3dofnds)/3)),[0, 1, 2])).astype(int)] = Fvdat[l
print("Whew.")
Fv[range(-len(n1dofnds), 0)] = Fvdat[ids, 2]
print("Done.")

print("Matrix extraction complete - writing mat file")
dict = {"M": M, "K": K, "R": R, "Fv": Fv};
io.savemat(".out.mat",dict);
print("Processing Over")
EOF
mv .out.mat $OUT
rm .STIFFNESS.mtx .MASS.mtx .RECOV.mtx .FVEC.mtx
```

- This bash script first uses GNU Awk, a simple but powerful utility that allows line-by-line parsing of files.
- The script also uses the cut utility from GNU coreutils for manipulations.
- Finally, the script uses Python (compatible with 2/3), involving numpy and scipy.io, for converting the quantities into a MATLAB mat-file that can be loaded on MATLAB.
- This script can be called as follows:

  ```
  ./readwritematvec.sh Modelmats.mtx
  ```

- In windows, this can be done either through Cygwin or Windows Subsystem for Linux.

7. It may be possible to do this natively in MATLAB, but since this requires a line-parser, awk is better suited for the job than MATLAB. A MATLAB implementation might require loading the whole file into memory for speed. Line-parsing on MATLAB was extremely slow for me.

The file $\text{model}_{\text{step3.cae}}$ is the final cae file that contains all of the above.

5.2 Simple Analysis on MATLAB/OCTAVE Script

We will now do our first nonlinear analysis on MATLAB/OCTAVE with the exported matrices. We will conduct a **Nonlinear Prestress Analysis** for a frictionless contact with a unilateral spring on the normal direction. **TO BE ADDED SOON**

6 Outro/Contact Details

In summary, all the scripts used in the tutorial are provided in

1. $\text{a}_{\text{steelmat.py}}$: Loading material
2. $\text{b}_{\text{halfbm.py}}$: Building the half-beam model
3. $\text{c}_{\text{nutwasherbolt516.py}}$: Building the bolt, nut, and washer models.
4. $\text{d}_{\text{applyconstraints.py}}$: Applying the relevant constraints and introducing bolt prestress.

5. $e_{\text{nodeproc.py}}$: The postprocessing script create surface node sets and setting up substructure analysis.

6. readwritematvec.sh: Contains the mtx reading script written with `bash` and `python`.

Don't forget to check out the repository in github at https://github.com/Nidish96/Abaqus4Joints. It has the above scripts as well as all the cae files.

You can contact me at nidish.balaji@ila.uni-stuttgart.de or nidbid@gmail.com for any questions/suggestions.