Chapter 19
Using Abaqus with Python to Perform QSMA on the TMD Structure

Brennan Bahr, Drithi Shetty, and Matthew S. Allen

Abstract Automotive and aerospace structures are increasingly making use of thin panels to reduce weight while seeking to maintain durability and minimize noise transmission. These panels can exhibit geometrically nonlinear behavior due to bending-stretching coupling. Additionally, the use of mechanical fasteners results in nonlinear hysteretic behavior due to friction between the contact surfaces. The Tribomechadynamics benchmark structure, consisting of a thin panel clamped at the ends using bolted joints, was developed as part of a research challenge to test the ability of the nonlinear dynamics community to predict the dynamic behavior of a structure with both friction and geometric nonlinearity. Simulating the dynamic response of a high-fidelity nonlinear FE model is highly computationally expensive, even for such a small-scale structure. Therefore, quasi-static methods have been gaining popularity. This paper builds on our previous efforts to predict the amplitude-dependent frequency and damping of the first bending mode of this structure using quasi-static modal analysis (QSMA). A 3D FE model of the TMD structure was analyzed. The paper shows how Python, an open-source programming language, can be integrated with a commercial finite element package to perform QSMA. This minimizes file input/output compared to our previous approach and speeds up the process. We also investigate using the pseudo-inverse of the mode shape matrix, rather than the mass matrix times the mode shape matrix, to further accelerate the computations. The QSMA results are used to fit a reduced-order model to the structure, which comprises a single DOF implicit condensation and expansion (or SICE) ROM for geometric nonlinearity and an Iwan model to characterize friction nonlinearity. This model is able to reproduce the nonlinear modal behavior with high fidelity while significantly reducing the computational cost.

Keywords Friction · Geometric Nonlinearity · Reduced-order modeling · Contact · Hysteresis

19.1 Introduction

Thin panels are commonly used in the design of lightweight, high-speed structures that are assembled together using mechanical fasteners. These panels exhibit nonlinear behavior due to bending-stretching coupling at large deformations [1, 2]. Additionally, friction at the interfaces that are fastened together results in energy dissipation which has a nonlinear effect on the system dynamics [3]. The industry standard is often to make linear approximations to create a computationally efficient finite element model. Such simplifications lead to conservative designs that need to be iterated on, resulting in greater cost of prototyping and dynamic testing. However, simulating the dynamic response of a high-fidelity nonlinear finite element model can be highly computationally expensive, with the cost increasing with complexity [4]. Therefore, reduced-order modeling approaches have been developed as a more efficient alternative [5, 6].

One such approach that has been gaining traction in the structural dynamics community is the method of quasi-static modal analysis [7, 8], or QSMA. In this method, the nonlinear FE model under consideration is statically excited in the shape of the mode of interest. The corresponding displacement can then be calculated using any finite element package. This is done over a range of load amplitudes to obtain the force-displacement backbone curve. The results can then be used to quantify the modal dynamic response of the structure over the amplitude range of interest, typically by estimating the

B. Bahr (✉) · M. S. Allen
Department of Mechanical Engineering, Brigham Young University, Provo, UT, USA
e-mail: matt.allen@byu.edu

D. Shetty
Department of Mechanical Engineering, UW-Madison, Madison, WI, USA
e-mail: ddshetty@wisc.edu

M. R. W. Brake et al. (eds.), *Nonlinear Structures & Systems, Volume 1*, Conference Proceedings of the Society for Experimental Mechanics Series, https://doi.org/10.1007/978-3-031-36999-5_19

amplitude-dependent frequency and damping behavior. Lacayo and Allen [8] showed that QSMA can be used to calculate the force-displacement relation for a nonlinear mode of a jointed structure consisting only of friction nonlinearity. Park and Allen [9] derived a similar relation for a single mode of a geometrically nonlinear structure. QSMA has been tested on different benchmark systems [10, 11], producing results that are in fairly good agreement with dynamic response predictions at a fraction of the computational cost. Furthermore, the method has been successfully applied to real-world structures [12].

Although QSMA offers a clear computational advantage over nonlinear dynamic analyses, some bottlenecks in its implementation still exist, especially in the case of larger 3D FE models. Efforts have been made to address these challenges. For instance, Jewell et al. [10] found that the contact and solver settings need to be iterated on to improve solution convergence and reduce solve time. Zare and Allen [13] proposed a contact algorithm that speeds up the quasi-static simulations, especially in the case of 3D models that comprise a two-dimensional friction problem. Another bottleneck, which is the main focus of this paper, is the extraction of the structural mass and mode shape matrices from the commercial FE package. The implementation of QSMA requires the mass and mode shape matrices, obtained by a linear eigenvalue analysis, in order to calculate the distributed static load to be applied. Since QSMA is not currently a standard procedure in commercial FE software, the static load must be externally calculated and fed to the FE package. Thus, the structural mass matrix and/or the mode shape matrix must be extracted from the FE software, which can grow quite large as the complexity of the model increases. In the past, Matlab scripts were created that would call on an FEA software, such as Abaqus, to perform a linear or static analysis. Due to the differences in syntax between the two programs, an additional step of translating the collected data into the appropriate language is needed if QSMA is to be performed. Skipping this additional step is desirable as the translation process takes up a significant amount of time while performing QSMA.

In our prior works, the steps above were performed using Matlab scripts that called upon a set of Python scripts to interface with Abaqus. However, Python has much of the same capability as Matlab so it was suggested to eliminate Matlab and perform all of the necessary analysis within Python. This paper presents a procedure to directly interface with the Python scripting language that is built into the commercial FEA software Abaqus in order to reduce the time that is needed to complete an analysis using QSMA. This is partly enabled by recent upgrades to Abaqus that incorporate newer and more complete versions of Python. While eliminating Matlab speeds up the file input-output, it was still necessary to write the mass matrix to a text file and read that into Python, and it became clear that doing so was an additional major bottleneck. Hence, a study was performed to see if the pseudo-inverse of the first m columns of the mode shape matrix could be used to obtain an adequate approximation of the distributed loading needed for QSMA.

The proposed improvements have been tested on a simplified 2D FE model of the TMD benchmark structure [14]. The TMD benchmark structure [15] consists of a thin, curved panel that is clamped at the ends with the help of bolts, thus potentially consisting of both geometric and frictional nonlinearity. Additionally, a 3D, high-fidelity FE model of the TMD structure has been considered. While loading the mass matrix of this model using the previous approach would take days, this paper shows how the new approach results in significant computational savings. In both case studies, the reduced-order modeling approach presented by Shetty et al. [14] has been used to estimate the overall changes in damping and frequency.

The following section reviews the major steps required to implement QSMA on a finite element model and discusses the Python scripts that are needed to implement this. The proposed approach is then applied to a simple two-dimensional model of two cantilevered beams that are bolted at their free ends, to verify the method. The scripts for this example are found in the Appendix and should enable others to test this approach and implement it on their systems. Then the new approach is applied to the TMD benchmark structure and its relative merits are investigated.

19.2 Theory

The QSMA process that was first presented in [8] is detailed below; see [8, 10] for additional details. The FE equations of motion for a N-degree of freedom (MDOF) system are given below, including the pre-stress in the joints $\mathbf{F}_{pre}$ and the joint force, $\mathbf{F}_J(\mathbf{x}, \boldsymbol{\theta})$, where $\boldsymbol{\theta}$ captures the stuck/slip state of each pair of contact nodes in the FEM.

$$\mathbf{M}\ddot{\mathbf{x}} + \mathbf{K}\mathbf{x} + \mathbf{F}_J(\mathbf{x}, \boldsymbol{\theta}) = \mathbf{F}_{ext} + \mathbf{F}_{pre} \tag{19.1}$$

The nonlinear term is approximated as $\mathbf{K}_0\mathbf{x}$ for small displacements about the preloaded state and the following eigenvalue problem is solved to find the linearized modes:

$$\left(\mathbf{K} + \mathbf{K}_0 - \omega_r^2\mathbf{M}\right)\boldsymbol{\phi}_r = \mathbf{0} \tag{19.2}$$

where ω_r and $\boldsymbol{\phi}_r$ are the natural frequency and eigenvector for the rth mode.

In prior works, the quasi-static force $\mathbf{M}\boldsymbol{\phi}_r\alpha$ was applied to excite only the rth mode, so that the quasi-static problem solved by the FEA software was the following:

$$\mathbf{Kx} + \mathbf{F}_J(\mathbf{x}, \boldsymbol{\theta}) = \mathbf{M}\boldsymbol{\phi}_r\alpha \tag{19.3}$$

Once the solution $\mathbf{x}$ was obtained, the modal response $q_r(\alpha)$ was obtained using $q_r(\alpha) = \boldsymbol{\phi}_r^{\mathrm{T}}\mathbf{Mx}(\alpha)$. While the above is theoretically exact, for large models it can be quite cumbersome to extract the mass matrix from the software so that the products $\boldsymbol{\phi}_r^{\mathrm{T}}\mathbf{M}$ and $\mathbf{M}\boldsymbol{\phi}_r$ can be evaluated. (For the 3D model discussed later, it could take more than a day to write the mass matrix to a text file and then to read that into Matlab.) Hence, this paper explores the following alternative.

The product $\boldsymbol{\phi}^{\mathrm{T}}\mathbf{M}$ is the inverse of the mode shape matrix, and so $\boldsymbol{\phi}_r^{\mathrm{T}}\mathbf{M}$ is simply the rth column of that inverse. Hence, it can be estimated by taking the pseudo-inverse of the mode shape matrix $\boldsymbol{\phi}$. In practice, one typically only computes a finite number of columns of the mode shape matrix, so the pseudo-inverse is denoted

$$\boldsymbol{\phi}^{L,m} = pinv\left(\boldsymbol{\phi}(:, 1:m)\right) \tag{19.4}$$

where $(:, 1:m)$ denotes the first m columns of the matrix. Once the pseudo-inverse has been computed, one can extract the rth column for the mode of interest, which is denoted $\boldsymbol{\phi}_r^{L,m}$ and is a $1 \times N$ vector. The matrix $\boldsymbol{\phi}$ is easily extracted from the Abaqus *.odb files into Python, and so one can easily compute the loading as $\mathbf{M}\boldsymbol{\phi}_r\alpha \approx \left(\boldsymbol{\phi}_r^{L,m}\right)^{\mathrm{T}}\alpha$ in Python. Then, Abaqus can be called to solve the following quasi-static problem:

$$\mathbf{Kx} + \mathbf{F}_J(\mathbf{x}, \boldsymbol{\theta}) = \left(\boldsymbol{\phi}_r^{L,m}\right)^{\mathrm{T}}\alpha \tag{19.5}$$

Once the nonlinear quasi-static solution $\mathbf{x}$ has been obtained, this can be converted into the rth modal displacement using $q_r(\alpha) \approx \left(\boldsymbol{\phi}_r^{L,m}\right)^{\mathrm{T}}\mathbf{x}(\alpha)$. Hence, the variables that need to be saved or exported are much smaller than the length N vectors and $N \times N$ matrices that are saved and imported into Matlab using our prior approach [10]. A validation of this new method is described in Sect. 19.3.2 of this work.

19.2.1 Using Python to Perform QSMA

This subsection outlines the proposed Python approach and contrasts it to our prior approach. As is stated in the previous section, the natural frequency and eigenvalues of various modes are needed in order to perform QSMA, and these should be obtained about the preloaded state. Hence, the procedure below assumes that one has already performed a nonlinear static analysis to preload the joints and has then performed linear modal analysis about that state, and that the output *.odb file from that analysis is available.

Once that is complete, one can run the first Python script, designated “InputWriter.py,” as this script will use the data from the linear analysis to calculate and apply a quasi-static load to be used in the QSMA nonlinear static analysis, as is outlined in Fig. 19.1. This script creates two additional input files. The first file, designated “model_force,” will contain the information needed for the QSMA static analysis step. This includes the typical parameter settings used to control the convergence of the model, as well as the forces and moments to be applied, which are given in the right-hand side of Eq. (19.5). The second file, designated “model_staticforce,” is a simple script that combines the original input file, which contains the model definition, with the “model_force” input file that was just created.

The “model_staticforce” input file can also be used to specify that a restart analysis is to be run. When this capability is used, Abaqus simply resumes the model that was run earlier (i.e., when doing preload and linear modal analysis), and so the model does not need to be assembled again and the preload step does not need to be repeated. The use of restarts in Abaqus is discussed later in this work as it was found to significantly decrease the time that is spent performing QSMA.

The second Python script, “PostProcessing.py,” is used to extract the static response $\mathbf{x}$ in Eq. (19.5) that was found during the QSMA static analysis and then to convert it to modal coordinates q_r. Then, a small number of modal coordinates are written to a *.mat file for storage. The authors currently bring this data into Matlab for any further analysis but this step is fast now because the data to be written is small as one is typically not interested in more than a few tens of modes. One could compute the amplitude-dependent frequency and damping from only one mode, the mode that is directly excited in

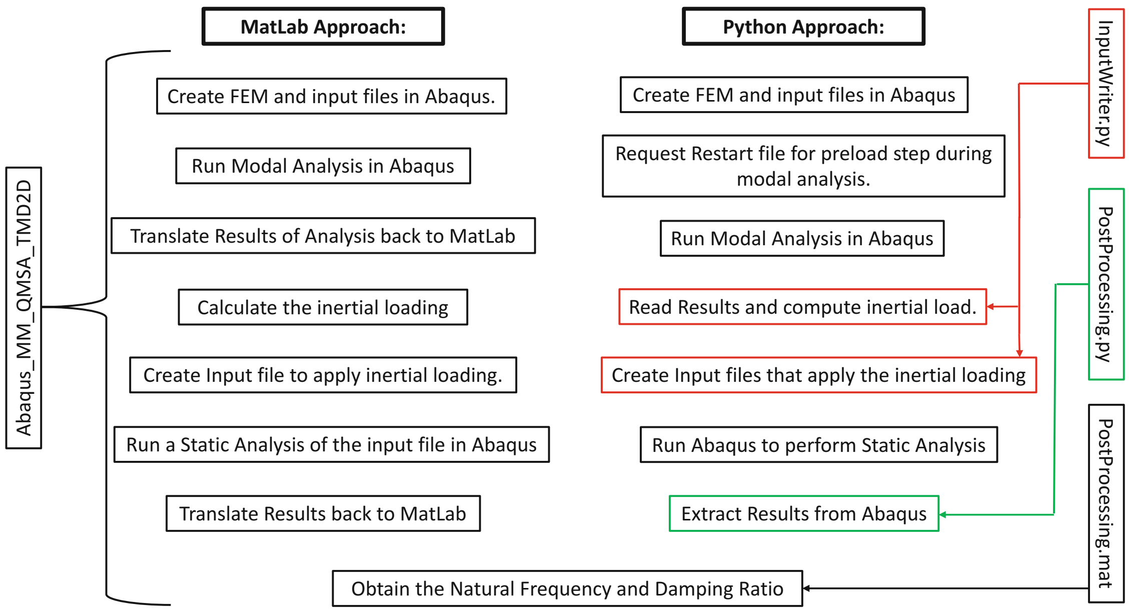
MatLab Approach:
Create FEM and input files in Abaqus.
Run Modal Analysis in Abaqus
Translate Results of Analysis back to MatLab
Calculate the inertial loading
Create Input file to apply inertial loading.
Run a Static Analysis of the input file in Abaqus
Translate Results back to MatLab
Abaqus_MM_QMSA_TMD2D
Python Approach:
Create FEM and input files in Abaqus
Request Restart file for preload step during modal analysis.
Run Modal Analysis in Abaqus
Read Results and compute inertial load.
Create Input files that apply the inertial loading
Run Abaqus to perform Static Analysis
Extract Results from Abaqus
InputWriter.py
PostProcessing.py
PostProcessing.mat
Obtain the Natural Frequency and Damping Ratio

Fig. 19.1 Flowchart representation of the two methods that are discussed in this work. The names of the scripts are given on the far sides of the flowchart, with Matlab on the left and Python on the right

Eq. (19.5), but other modes are also typically imported in order to see how much they were statically coupling to the mode in question.

19.2.1.1 Using Abaqus's Restarts

Restarts were used frequently while analyzing various models to reduce the time needed. Restarts allow the user to resume an analysis at a specified step if the requisite files for the previous steps have been created. This was very useful in the context of QSMA because any QSMA analysis starts with a preload. Using restarts one can run preload only once and then perform QSMA for as many modes as needed or as many load levels as needed. The disadvantage is that the files needed for the restart consume quite a bit of disk space. A description of how to create and use restart files can be found in [16].

19.2.1.2 Rough Friction vs. Lagrange Friction

As is discussed later in this work, the TMD benchmark structure is particularly difficult to solve due to the fact that it consists of both geometric and frictional nonlinearity. The nonlinearity that arises due to friction occurs due to micro-slip between the bolts and the assembly. Geometric nonlinearity comes about as the panel bends due to bending-stretching coupling. As is explained in [14], one way to distinguish the effects of the two types of nonlinearity is to run an analysis with both types of nonlinearity enabled and then to run a separate analysis in which the nonlinearity due to friction is eliminated. This can be done by setting the coefficient of friction to infinity so as to ensure that no micro-slip will arise during the analysis. The data from these different runs are then collected and processed as outlined in [14].

To implement this, one only needs to edit the portion of the Abaqus input file where the interaction property is defined and to set that to what Abaqus designates as "rough" friction [16]. This effectively sets the coefficient of friction to infinity to ensure that nothing will slip in the analysis. An example of this type of analysis is shown in Fig. 19.2, where the modal load-displacement curves are shown for both types of analyses.

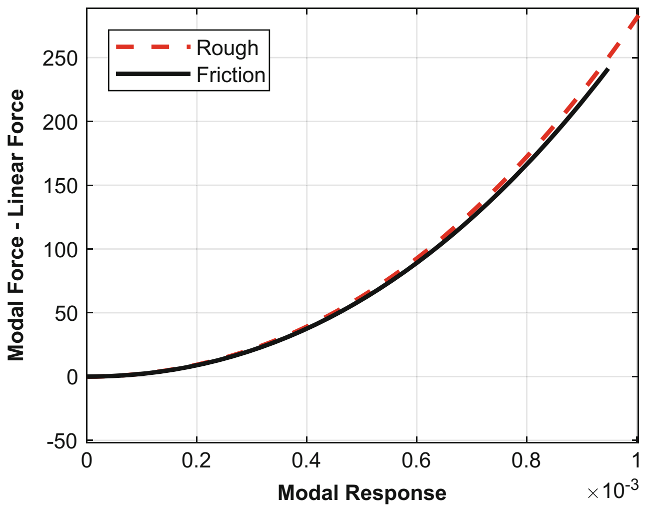

Fig. 19.2 Load-displacement curves for 2D TMD benchmark model with an infinite coefficient of friction (i.e., "rough" contact in Abaqus) and Lagrange friction with $\mu = 0.6$

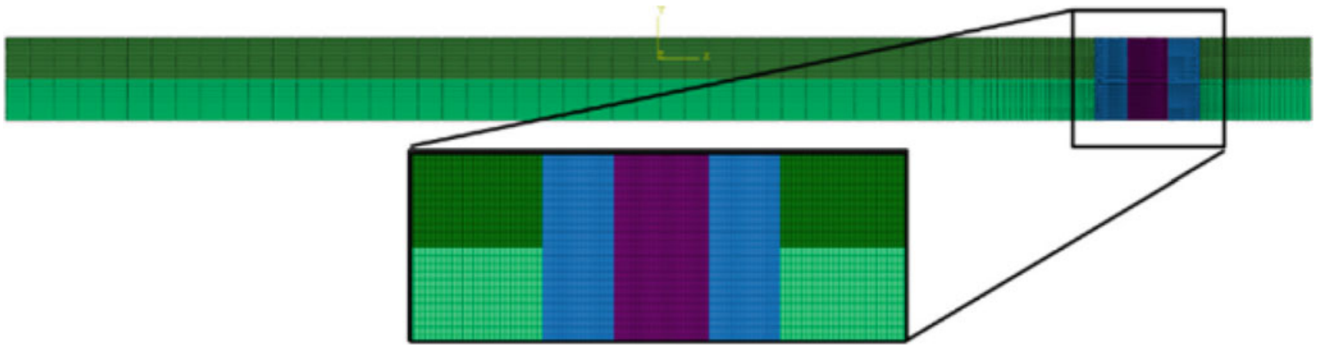

Fig. 19.3 Visual representation of the stacked beam model that is used in Sect. 19.3. The magnified portion of the figure represents the pressure load that is implemented to represent a bolt that holds the two plates together

19.3 Case Study 1: Stacked Beam

To test the efficacy of using Python to perform QSMA, the abovementioned scripts were used to analyze the same simple two-dimensional model that was used in [10]. QSMA was performed using Python to extract the amplitude-dependent variables of the first mode of vibration, and the results were compared to those obtained by using the previous method.

The model consists of two cantilever beams that are bolted together at one end as is seen in Fig. 19.3. Each beam is 203 mm long, 6.35 mm thick, and infinite in width due to the plane strain elements that were used during analysis. A fixed boundary condition was applied to the left side of the two beams, and the bolt that fastened the two beams together was modeled as a pressure load on the top and bottom of the beam 25.44 mm from the right side of the structure. The pressure load applied to the system is 288 Newton meters in magnitude distributed across 6.35 mm to mimic the effect of a bolt with a 6.35 mm diameter fastened with 4450 N of preload. For a more detailed description of the meshing and materials that were used, see [10]. It is worth mentioning that the material that was used to construct the beam is linearly elastic, which means that the only damping in the system is due to Coulomb friction between the bolt and the beams.

19.3.1 Application of Quasi-static Modal Analysis

The first step in the analysis was to solve for the preload in the bolt and to perform linear modal analysis. The input file for this step is available from the authors upon request. This analysis results in the computation of the linearized mode shapes

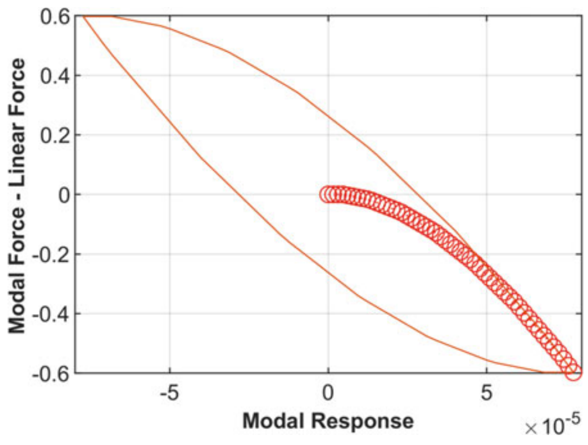

Fig. 19.4 Example of a modal force-displacement result for the 2D beam, as well as the resulting hysteresis loop that is formed by using Masing's rules. As is mentioned in [8], the slope of the secant line of the force-displacement backbone is used to calculate the natural frequency of the system, and the area enclosed in the hysteresis loop is used to find the damping ratio of the system

in Eq. (19.2). The first 20 modes of the assembly are considered, as a greater number of eigenvalues calculated result in a more orthogonal matrix when computing the pseudo-inverse in Eq. (19.4), as is discussed in Sect. 19.3.2 of this work. These modes are also used after the QSMA analysis to determine if modal coupling is present in the results.

After the modal analysis was completed, the first Python script is run. This script extracts the mode shapes that were evaluated during the analysis and calculates the inertial load that is to be applied to each node using Eq. (19.5). An input file is then created that specifies the magnitude and direction of the load on each node.

If various tests were to be performed, then restart files were created before submitting the new input files for analysis. The restart functionality does little to speed up this small model, but this model is a good test bench for debugging the restart procedure.

19.3.1.1 Post-Processing Results from the Static Analysis

The second Python script is initiated, either from the command line or from the Abaqus GUI after the static analysis has concluded to extract the modal displacements of each mode due to the preload and inertial load. Note that even though it is a Python script, it must be run within Abaqus because it makes use of libraries that are only available within the Python installation that exists within Abaqus. As mentioned previously, that second script saves modal load-displacement data, such as that shown in Fig. 19.2. This force-displacement backbone curve can then be used to form a hysteresis loop using Masing's rules. This practice was utilized in [8] in connection with quasi-static modal analysis to calculate the natural frequency and damping of the structure. The same procedure is used in this work to find the natural frequency and damping of the nonlinear structure being examined. An example of the force-displacement backbone curve, as well as the resultant hysteresis loop, is shown in Fig. 19.4.

The calculated values for the natural frequency and damping are then graphed with respect to peak modal velocity. The results are then compared to the graphs that were created using the previous method in Fig. 19.5

19.3.2 Validation of Pseudo-inverse Approach

This section explores the feasibility of using the pseudo-inverse method to perform QSMA. As mentioned previously, this method was desired over using $\mathbf{M}\boldsymbol{\phi}$ due to the time that is needed to pull the mass matrix into either Matlab or Python in order to compute the product. In order to test the viability of the pseudo-inverse method, QSMA was performed on the stacked beam structure [10] using both approaches. A summary of the comparison is provided below, as well as recommendations for using the $pinv$ function in the future.

The accuracy of the pseudo-inverse $\boldsymbol{\phi}^{L,m}$ was compared with that of the traditional approach by multiplying the matrix by $\boldsymbol{\phi}$. The result can then be compared with that of the $\boldsymbol{\phi}_r^T \boldsymbol{M}$ matrix to assess the accuracy. It is worth noting that the Python script that was used to calculate $\boldsymbol{\phi}^{L,m}$ used Numpy's $pinv$ function. The Python library Scipy also has a pseudo-inverse function, but Numpy's pseudo-inverse approximation uses less memory than Scipy's as Scipy's $pinv$ function uses a least squares approximation, while Numpy's $pinv$ function uses an SVD to approximate the inverse. This made Numpy's $pinv$ function more desirable as it takes less time to compute while still giving reliable results.

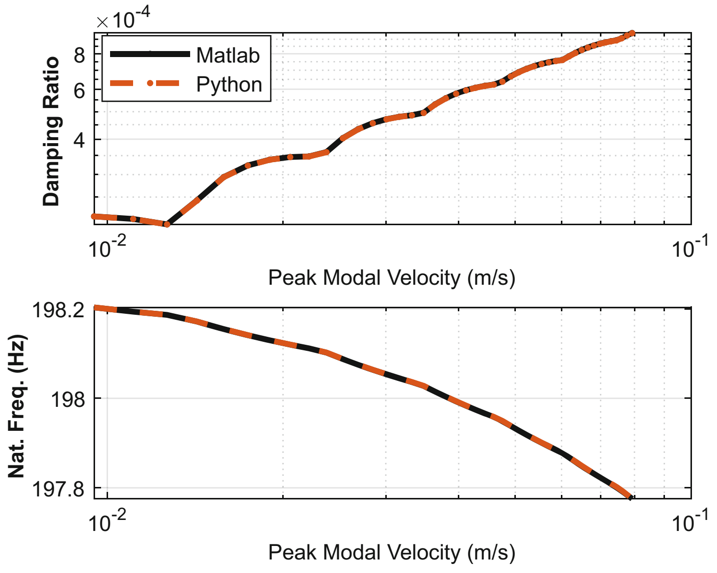

Fig. 19.5 Comparison of the natural frequency and damping ratio obtained using the Python scripts that are described in this work and the previous Matlab script for the stacked beam model. The results differ only in the third decimal place

Table 19.1 The product of $\boldsymbol{\phi}^{L,m}$ and $\boldsymbol{\phi}$ for the 2D stacked beam structure [10]. The pseudo-inverse was computed using 20 modes, but the table only shows the first five columns and rows of the product

	Mode 1	Mode 2	Mode 3	Mode 4	Mode 5
Mode 1	1	6.14e-16	−3.95e-16	−1.49e-15	1.94e-16
Mode 2	8.34e-16	1	8.70e-16	2.15e-15	−3.07e-16
Mode 3	1.17e-15	−9.86e-16	1	−7.14e-16	5.10e-16
Mode 4	7.43e-17	−3.91e-16	−6.31e-18	1	−4.68e-17
Mode 5	−4.17e-16	−1.19e-16	−2.70e-15	−2.57e-16	1

As seen in Table 19.1, the pseudo-inverse provides an excellent approximation to the inverse $\boldsymbol{\phi}$. In fact, the off-diagonal values were lower using this approach than using the more rigorous $\mathbf{M}\boldsymbol{\phi}$. Hence, it seems that this is a viable approach for calculating the inertial load that is to be applied in QSMA.

When testing this approach, it was found that the method lost accuracy if too small a number of mode shapes was used, yet using 20 mode shapes as done above gave excellent results and it was not very expensive to compute that many modes even for the larger models considered later.

19.4 Case Study 2: 2D TMD Benchmark Structure

The advantages of using the python approach become much clearer when studying a larger model. The model that is studied in this section is known as the Tribomechadynamics benchmark structure [14]. The benchmark structure is comprised of a 1.5 mm thick panel that is attached to two blocks, known as blades. The panel is fastened to both blades by two rows of three bolts. The surface of the support that is in contact with the panel has an inclination of 1^o on both sides, which results in slight curvature of the panel, as seen in Fig. 19.6. Due to the assembly of the panel, the structure will experience geometric nonlinearity, as well as nonlinearity due to friction. The structure will also behave differently depending on the direction in which the inertial load is applied, which is to be accounted for if a full analysis of the structure is needed.

Because this system has both geometric and friction/slip nonlinearity, the standard QSMA approach must be augmented. Shetty et al. [14] proposed an extension that was found to give excellent results for this structure, but to implement their method one must perform at least three analyses: two analyses with the friction disabled (i.e., the surfaces are glued together) in which the structure is deformed in positive and negative directions, and a third with both friction and geometric nonlinearity present. Hence, the use of restart files was quite beneficial for this particular model as it allowed the first two steps of the analysis to be run only once. Those first two steps required about an hour for this model. It is also worth noting that the static was performed using a Riks analysis (rather than Abaqus's static, general method). The Riks analysis was found to provide

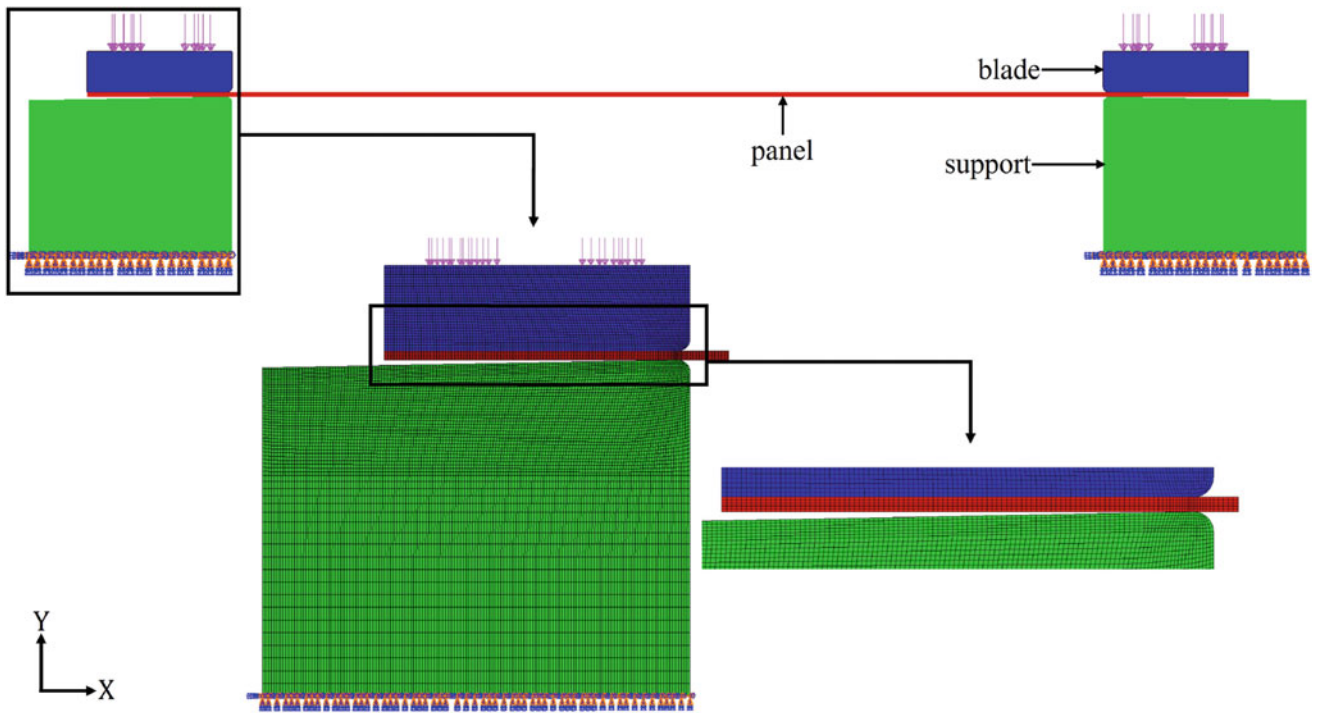

Fig. 19.6 2D FE model of the TMD structure, with a magnified version of the left-hand side interface showing the mesh density at contact, from [14]

greater accuracy over the range of loads in the study in [14], whereas the static general analysis only meets the specified tolerances for the largest loads in each analysis.

Once the linear analysis had been completed, one can proceed to calculate the inertial load. In the Python script in the Appendix, the load scale factor determines both the magnitude and the sign of the load applied to the beam. Because the mode can have an arbitrary sign, some care is needed. In the script shown some checks are included to ensure that a load scale factor of positive one applies the inertial load in the positive y direction, while a load scale factor of negative one applies the inertial load in the $-y$ direction. The maximum displacement of the panel is also specified by indicating that a node located at the middle of the panel should not be displaced more than 4.2 mm, or double the thickness of the panel, in either direction. The maximum load proportionality factor could also be specified in place of the maximum displacement of a chosen node. Once these variables have been defined, the first Python script can then be run in Abaqus to calculate the forces and moments that are to be applied at each node and to write those to an input file.

Abaqus is then called to run the QSMA static analysis. Once the static analysis has finished running, the results can be extracted from the *.odb file for post-processing using the second Python script. Two more analyses are then run in which slip at the joint is eliminated by setting the contact to “rough” in Abaqus. This analysis is repeated with a positive and negative load scale factor.

19.4.1 Results

The results obtained using the new Python based approach were once again indistinguishable from those presented in [14], and so they are not repeated here. Table 19.2 compares the time required for each step when using each method.

The three QSMA static analyses mentioned previously are denoted A through C in Table 19.2. Analyses A and B had “rough” contact (no slip), with the load applied in the $-y$ and $+y$ directions, respectively. Analysis C was run with Lagrangian friction with $\mu = 0.6$ and with the quasi-static load applied in the $-y$ direction. The preload analysis, linear analysis, and frequency and damping computation all follow the same process and therefore should have similar times as is shown. The main differences between the two methods are the time needed to load the mass and stiffness matrices from the Abaqus output file and the time needed to process the QSMA output. It takes less time to process the QSMA output using

Table 19.2 Time taken to estimate the frequency and damping of the TMD 2D FE model using the Matlab [14] compared to the Python approach proposed in this work

Analysis	Matlab scripts	Python scripts with restarts
Preload analysis	2911 s (≈49 min.)	3301 s (≈55 min.)
Linear modal analysis	4 s	4 s
Calculation and application of inertial load	84.2 s	47.5 s
QSMA—analysis A	530 s (≈9 min.)	486 s (≈ 8 min.)
QSMA—analysis B	530 s (≈9 min.)	532 s (≈9 min.)
QSMA—analysis C	673 s (≈11 min.)	632 s (≈11 min.)
Post-processing QSMA output	(3 × 2123) = 6369 s (≈1.77 h)	(3 × 1894) = 5682 s (≈1.6 h)
Frequency and damping computation	63 s	16 s
Total time taken	3.12 h	3.06 h

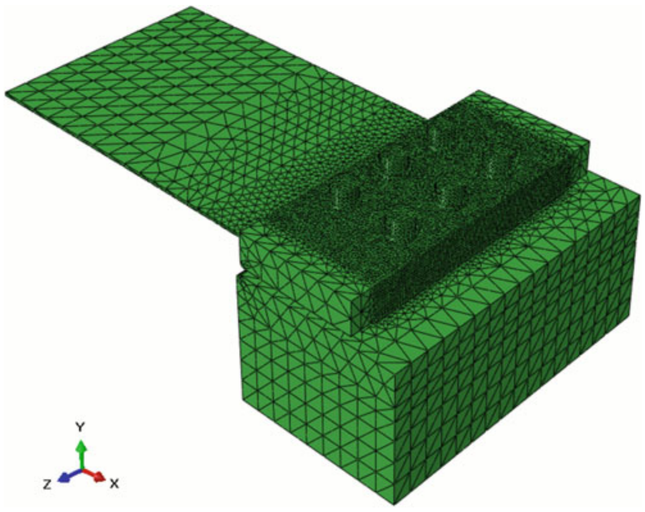

Fig. 19.7 3D FE model of the support, blades, and panel, dubbed here the partial assembly because half-symmetry was employed

the Python scripts as all of the information is processed in Python and only a small set of results are written to disk. While the difference in time taken may not be huge in this case study, it grows significantly when larger FE models are considered. The frequency and damping estimated by this method are identical to the results presented in [14] and have therefore not been shown here for brevity.

19.5 Case Study 3: 3D TMD Benchmark Structure

A 3D version of the TMD benchmark structure [14] has also been created, and the proposed approach has been instrumental in running analyses on this model. The same process that was used for the 2D version of the model is currently being used, but the analyses take much longer to solve due to the complexity of the model. It is hoped that the new methods described in this paper will significantly decrease the amount of time needed for analysis. To date, the preload analysis has been completed using restart, and several static analyses have been completed. As an example, it takes 2–3 days to extract the mass matrix from the 3D model and to load it into Abaqus, but this step is no longer needed when the $pinv$ method described in Sect. 19.4 is implemented. Each static analysis takes 5–7 days to complete and they frequently crash, so the Python workflow proposed here is optimal as it is easy to restart if interrupted and doesn't require Matlab to be open (Fig. 19.7).

19.6 Conclusions

This paper has presented a new approach for performing QSMA that makes use of Python to minimize file input/output and the associated computational costs. While the changes to the workflow are quite elementary, it is hoped that the exposition can serve as a tutorial for any researchers interested in performing QSMA. The new workflow appears to be quite promising when large models need to be studied, and is relatively easy to implement.

The interested reader can copy and paste the Python scripts provided in the Appendix and apply this approach to a wide range of finite element models with minimal modification. These scripts are provided without warranty, and they will likely be improved in the coming months and those improvements can also be made available upon request. We also welcome any improvements that any readers may make.

The conference presentation will cover the results obtained on the 3D TMD structure, which have been enabled by this new and more efficient workflow.

Appendix

InputWriter.py

```
import timeit
tic = timeit.default_timer()
from odbAccess import *
from abaqusConstants import *
import copy
import sys
import numpy
import scipy
from scipy.io import savemat
import section
import regionToolset
import displayGroupMdbToolset as dgm
import part
import material
import assembly
import step
import interaction
import load
import mesh
import optimization
import job
import sketch
import visualization
import xyPlot
import displayGroupOdbToolset as dgo
import connectorBehavior
import math

#Values that need to be changed based on the model, jobname, and
#mode number to be analyzed.
modelname = 'TMDbenchmark-2D-riks'
requestedMode = 1
jobName = "TMDbenchmark-2D-refine_matmodes"
part = 'PART-1-1'          #'PART-1-1' is default
loadSclFactor = 1.0        #constant taken from previous code
thick = 1.5                #constant taken from previous code
```

```
requested_disp = loadSclFactor*thick
lastStep = 'BoltPreload'  #Last step before adding quasi-static load
Modal = False
#Does the input file have a modal analysis step that is not needed?

odbFileName= jobName + ".odb"
db = openOdb(odbFileName)


setName='mode'
step_names=db.steps.keys()
step_name=step_names[-1]

myStep=db.steps[step_name]
numFrames=len(myStep.frames)-1
numValues=len(myStep.frames[0].fieldOutputs['U'].values)

#   print numFrames
#   print numValues

# frames[0] is the base state, really have numFrames-1 numFrames
freq=numpy.zeros((numFrames,1),dtype=float);
disp=numpy.zeros((6*numValues,numFrames),dtype=float);
dof =numpy.zeros((6*numValues, 1), dtype=float); # JDS Modification
descript=[]

v=0
while v < numFrames:
    myMode=myStep.frames[v+1]
    freq[v]=myMode.frequency
    descript.append(myMode.description)
    n=0
    while n < numValues:
        #
        try:
            data=myMode.fieldOutputs['U'].values[n].data
        except OdbError:
             data=myMode.fieldOutputs['U'].values[n].dataDouble
        #
        # Add displacements
        disp[6*n+0][v]=data[0]
        disp[6*n+1][v]=data[1]
        if len(data) > 2:
            disp[6 * n + 2][v] = data[2]
        else:
            disp[6 * n + 2][v] = 0.
        #
        #
        # Add DOF to DOF vector using [node.1, node.2, node.3, ...] convention
        if v == 0:
            dof[6*n + 0] = myMode.fieldOutputs['U'].values[n].nodeLabel + 0.1
            dof[6*n + 1] = myMode.fieldOutputs['U'].values[n].nodeLabel + 0.2
            dof[6*n + 2] = myMode.fieldOutputs['U'].values[n].nodeLabel + 0.3
        #
        if myMode.fieldOutputs.has_key('UR'):
            #
            try:
                data=myMode.fieldOutputs['UR'].values[n].data
            except OdbError:
                data=myMode.fieldOutputs['UR'].values[n].dataDouble
            #
            # Add displacements
            disp[6*n + 3][v] = data[0]
            disp[6*n + 4][v] = data[1]
            if len(data) > 2:
                disp[6*n + 5][v] = data[2]
            else:
                disp[6*n + 5][v] = 0.
            #
            # Add DOF
            if v == 0:
              dof[6*n + 3] = myMode.fieldOutputs['UR'].values[n].nodeLabel + 0.4
              dof[6*n + 4] = myMode.fieldOutputs['UR'].values[n].nodeLabel + 0.5
              dof[6*n + 5] = myMode.fieldOutputs['UR'].values[n].nodeLabel + 0.6
                #
            elif myMode.fieldOutputs.has_key('UR3'): # ? need to add else?
              else: #
                #
                try:
                    data=myMode.fieldOutputs['UR3'].values[n].data
                except OdbError:
                    data=myMode.fieldOutputs['UR3'].values[n].dataDouble
                #
                disp[6*n + 5][v]=data
                if v == 0:
                    dof[6*n + 5] = myMode.fieldOutputs['UR3'].values[n].
                      nodeLabel + 0.6
        n+=1
    v+=1

"""
Creating the Scale Factor to multiply with disp to get the correct force to add
to the model before analysis
"""

"""
modesOfInterest = [1]

if (modeNum > 1) :
for i in range(1,modeNum) :
modesOfInterest.append(i + 1)
"""

modeNum = requestedMode
N = numpy.size(dof)

phi = copy.deepcopy(disp)

P = numpy.size(phi[:,modeNum - 1])
```

```
#taking the pseudoinverse of phi
mPhi = numpy.linalg.pinv(phi)

#find translational displacements using dof
tempDof = copy.deepcopy(dof)
for i in range(0,N):
        if ((float(dof[i]) - floor(dof[i])) < 0.35):
            tempDof[i] = 1.0
        else:
            tempDof[i] = 0.0

#need to make a fake array as to not change the values of phi
#we need to find the max of the translation values in phi
fakePhi = copy.deepcopy(phi)

#find values of phi that match translational displacements
for i in range(0,P):
        fakePhi[i,modeNum-1] = abs((float(phi[i, modeNum-1])) * (float
          (tempDof[i])))
        #print(tempDof[i],phi[i,modeNum-1],fakePhi[i,modeNum-1])
        # ^^^ Use to check values if needed

#find max value of translational displacements for scale factor
phi_max = float(max(fakePhi[:,modeNum-1]))

#find index of phi_max
phi_max_ind = numpy.argmax(fakePhi[:,modeNum - 1])

#Ensures the sign of loadSclFactor is independent of an arbitrary modeShape sign
loadSclFactor = ((loadSclFactor) * (float(numpy.sign(phi[phi_max_ind,
  modeNum-1]))))

#Calculating Scale Factor to apply to mPhi
alpha = ((float(freq[modeNum-1])*2*math.pi)**2)*loadSclFactor*thick/phi_max

fshape = alpha * mPhi[modeNum-1,:]

"""
sign command in matlab is the same as numpy.sign() command in python
** is the same as ^ for python
numpy.linalg.pinv(phi) - finds the pseudoinverse of phi
numpy.where - finds the index of the argument in ()
copy.deepcopy - creates a copy of the array without using a pointer to the data
like phi = disp
"""

with open(modelname + '_force.inp','w') as f:
    f.write('*STEP, inc=10000000, nlgeom=YES, NAME=STATIC\n')
    f.write('*STATIC, riks, stabilize, factor=0.0002, allsdtol=0.00,
      continue=NO\n')
    f.write('**InitArcLenInc, EstTotArcLen, MinArcLenInc, MaxArcLenInc, Max LPF,
        NodeRegion, DOF, MaxDisp\n')
    f.write('1e-3, 12., 1e-8, 1e-1, , MaxDispNode, 2, 4.2\n')
    f.write('*CLOAD, OP=new\n')
    for i in range(0,N):
        dofi = float(dof[i])
        nodeNum = int(dofi)
        dirval = int(round((dofi-nodeNum)*10))
        force = fshape[i]
        if (nodeNum == 0 or abs(force) < 1.0e-15):
            continue
        elif (dirval == 1):
            f.write(str(nodeNum)+', 1, ')
            f.write(str(force)+'\n')

        elif (dirval == 2):
            f.write(str(nodeNum)+', 2, ')
            f.write(str(force)+'\n')

        elif (dirval == 3):
            f.write(str(nodeNum)+', 3, ')
            f.write(str(force)+'\n')

        elif (dirval == 4):
            f.write(str(nodeNum)+', 4, ')
            f.write(str(force)+'\n')

        elif (dirval == 5):
            f.write(str(nodeNum)+', 5, ')
            f.write(str(force)+'\n')

        elif (dirval == 6):
            f.write(str(nodeNum)+', 6, ')
            f.write(str(force)+'\n')

        else:
            continue
    f.write('*END STEP\n')

with open(modelname + '_staticforce.inp','w') as f:
    f.write('** Static Force\n')
    f.write('**\n**\n')
    f.write('*Restart, read, step=1\n')
    f.write('** -----------------------------------------------------------
      -----\n')
    f.write('**\n')
    f.write('** ----------------------------------------------------\n')
    f.write('** ------ STEP data for SUBCASE 1\n')
    f.write('** ----------------------------------------------------\n')
    f.write('*INCLUDE, INPUT=' + modelname + '_force.inp')

mdic = {'alpha':alpha, 'Mphi': mPhi, 'fn':freq, 'phi_max_ind':phi_max_ind,
    'phi_max':phi_max}
savemat("VariablesFromModal", mdic, True)

numpy.save("Mphi",mPhi)

toc = timeit.default_timer()

print(toc - tic)
```

PostProcessing.py

```
import timeit
tic = timeit.default_timer()
from odbAccess import *
from abaqusConstants import *
import copy
import sys
import numpy
import scipy
from scipy.io import savemat
import section
import regionToolset
import displayGroupMdbToolset as dgm
import part
import material
import assembly
import step
import interaction
import load
import mesh
import optimization
import job
import sketch
import visualization
import xyPlot
import displayGroupOdbToolset as dgo
import connectorBehavior
import math

jobName = "TMDbenchmark-2D-riks_staticforce"
odbFileName= jobName + ".odb"
matFileName = jobName + "_FromAbaqus"
db = openOdb(odbFileName)


setName='mode'
step_names=db.steps.keys()
step_name=step_names[-1]

myStep=db.steps[step_name]
numFrames=len(myStep.frames)
numValues=len(myStep.frames[0].fieldOutputs['U'].values)

#   print numFrames
#   print numValues

# frames[0] is the base state, really have numFrames-1 numFrames
staticDisp=numpy.zeros((6*numValues,numFrames),dtype=float);
dof =numpy.zeros((6*numValues, 1), dtype=float); # JDS Modification
descript=[]

"""
Getting disp, dof, and freq arrays
"""
```

```
v=0
while v < numFrames:
    myMode=myStep.frames[v]
    descript.append(myMode.description)
    n=0
        while n < numValues:
        #
        try:
            data=myMode.fieldOutputs['U'].values[n].data
        except OdbError:
            data=myMode.fieldOutputs['U'].values[n].dataDouble
        #
        # Add displacements
        staticDisp[6*n+0][v]=data[0]
        staticDisp[6*n+1][v]=data[1]
        if len(data) > 2:
            staticDisp[6 * n + 2][v] = data[2]
        else:
            staticDisp[6 * n + 2][v] = 0.
        #
        #
        # Add DOF to DOF vector using [node.1, node.2, node.3, ...] convention
        if v == 0:
            dof[6*n + 0] = myMode.fieldOutputs['U'].values[n].nodeLabel + 0.1
            dof[6*n + 1] = myMode.fieldOutputs['U'].values[n].nodeLabel + 0.2
            dof[6*n + 2] = myMode.fieldOutputs['U'].values[n].nodeLabel + 0.3
        #
        if myMode.fieldOutputs.has_key('UR'):
            #
            try:
                data=myMode.fieldOutputs['UR'].values[n].data
            except OdbError:
                data=myMode.fieldOutputs['UR'].values[n].dataDouble
            #
            # Add displacements
            staticDisp[6*n + 3][v] = data[0]
            staticDisp[6*n + 4][v] = data[1]
            if len(data) > 2:
                staticDisp[6*n + 5][v] = data[2]
            else:
                staticDisp[6*n + 5][v] = 0.
            #
            # Add DOF
            if v == 0:
              dof[6*n + 3] = myMode.fieldOutputs['UR'].values[n].nodeLabel + 0.4
              dof[6*n + 4] = myMode.fieldOutputs['UR'].values[n].nodeLabel + 0.5
              dof[6*n + 5] = myMode.fieldOutputs['UR'].values[n].nodeLabel + 0.6
                #
            elif myMode.fieldOutputs.has_key('UR3'): #? need to add else? else:#
                #
                try:
                    data=myMode.fieldOutputs['UR3'].values[n].data
                except OdbError:
                    data=myMode.fieldOutputs['UR3'].values[n].dataDouble
                #
```

```
                    staticDisp[6*n + 5][v]=data
                    if v == 0:
                        dof[6*n + 5] = myMode.fieldOutputs['UR3'].values[n].
                         nodeLabel + 0.6
            n+=1
        v+=1

mPhi = numpy.load("Mphi.npy")
#loads in mPhi from the previous script.

LPF = numpy.array(db.steps['STATIC'].historyRegions['Assembly Assembly-1']
    .historyOutputs['LPF'].data)
#Extracts the Load Proportionality Factor.

#Extracts the modal displacement calculated from the Static Analysis.
u_nls = numpy.squeeze(staticDisp)
R = numpy.size(u_nls[:,0])
u_prel= numpy.zeros((R,1),dtype = float)
#Creates an array with the displacements calculated from the Preload Step.
for i in range (0,R):
    u_prel[i,0] = u_nls[i,0]

C = numpy.size(u_nls[0,:])
tempArray = numpy.ones((1,C), dtype = float)
m = numpy.matmul(u_prel,tempArray)
#Separates the displacements from the Preload Step from the displacements
#from the Static Step.
u_nls = u_nls - m

tempDisp = copy.copy(u_nls)
u_nls = numpy.zeros((R,(C-1)),dtype = float)
for i in range (0,C-1):
    for j in range (0,R):
            u_nls[j,i] = tempDisp[j,i+1]

q_nls = numpy.matmul(mPhi,u_nls)

mdic = {"u_nls":u_nls,"u_prel":u_prel,"q_nls":q_nls,"LPF":LPF}
savemat("VariablesFromStatic",mdic,True)

toc = timeit.default_timer()
print(toc-tic)
```

References

1. Woinowsky-Krieger, S.: The effect of an axial force on the vibration of hinged bars. J. Appl. Mech. **17**(1), 35–36 (2021)
2. Raju, I.S., Venkateswara Rao, G., Kanaka Raju, K.: Effect of longitudinal or inplane deformation and inertia on the large amplitude flexural vibrations of slender beams and thin plates. J. Sound Vib. **49**(3), 415–422 (1976)
3. Ungar, E.E.: The status of engineering knowledge concerning the damping of built-up structures. J. Sound Vib. **26**(1), 141–154 (1973)
4. Kuether, R.J., Najera, D.A., Ortiz, J., Khan, M.Y., Miles, P.R.: 2021 tribomechadynamics research challenge: Sandia National Laboratories high-fidelity FEA approach. Presented at 40th International Modal Analysis Conference (IMAC-XL) (2022)
5. Mignolet, M.P., Przekop, A., Rizzi, S.A., Michael Spottswood, S.: A review of indirect/non-intrusive reduced order modeling of nonlinear geometric structures. J. Sound Vib. **332**(10), 2437–2460 (2013)
6. Segalman, D.J., Gregory, D.L., Starr, M.J., Resor, B.R., Jew, M.D., Lauffer, J.P., Ames, N.M.: Handbook on Dynamics of Jointed Structures. Technical report, Sandia National Laboratories, Albuquerque, 2009

7. Festjens, H., Chevallier, G., Dion,J.-L.: A numerical tool for the design of assembled structures under dynamic loads. Int. J. Mech. Sci. **75**, 170–177 (2013)
8. Lacayo, R.M., Allen, M.S.: Updating structural models containing nonlinear Iwan joints using quasi-static modal analysis. Mech. Syst. Signal Process. **118**, 133–157 (2019)
9. Park, K., Allen, M.S.: Quasi-static modal analysis for reduced order modeling of geometrically nonlinear structures. J. Sound Vib. **502**, 116076 (2021)
10. Jewell, E., Allen, M.S., Zare, I., Wall, M.: Application of quasi-static modal analysis to a finite element model and experimental correlation. J. Sound Vib. **479**, 115376 (2020)
11. Wall, M., Zare, I., Allen, M.S.: Predicting S4 Beam Joint Nonlinearity Using Quasi-Static Modal Analysis. Springer, Berlin (2020)
12. Allen, M.S., Schoneman, J.D., Scott, W., Sills, J.W.: Application of Quasi-Static Modal Analysis to an Orion Multi-Purpose Crew Vehicle Test. Houston (2020)
13. Zare, I., Allen, M.S.: Adapting a contact-mechanics algorithm to predict damping in bolted joints using quasi-static modal analysis. Int. J. Mech. Sci. **189**, 105982 (2021)
14. Shetty, D., Allen, M.S., Park, K.: A new approach to model a system with both friction and geometric nonlinearity. J. Sound Vib. **552**, 117631 (2023)
15. Krack, M., Schwingshackl, C., Brake, M.R.: The tribomechadynamics research challenge. In 40th International Modal Analysis Conference (IMAC-XL), p. 3 (2022)
16. Abaqus Analysis User's Guide (2014)