Journal of Visualized Experiments

A Finite Element approach for locating the Center of Resistance of Maxillary Teeth

--Manuscript Draft--

Article Type:	Invited Methods Article - JoVE Produced Video
Manuscript Number:	JoVE60746R2
Full Title:	A Finite Element approach for locating the Center of Resistance of Maxillary Teeth
Section/Category:	JoVE Biology
Keywords:	Orthodontics, Center of Resistance, Maxillary teeth, Three-Dimensional, Cone-Beam Computed Tomography, Mimics, 3Matic, Finite Element Analysis.
Corresponding Author:	Madhur Upadhyay University of Connecticut Health Center Farmington, Connecticut UNITED STATES
Corresponding Author's Institution:	University of Connecticut Health Center
Corresponding Author E-Mail:	maupadhyay@uchc.edu
Order of Authors:	Bill Luu
	Edward Anthony Cronauer
	Vaibhav Gandhi
	Jonathan Kaplan
	David M Pierce
	Madhur Upadhyay
Additional Information:	
Question	**Response**
Please indicate whether this article will be Standard Access or Open Access.	Standard Access (US$2,400)
Please indicate the **city, state/province, and country** where this article will be **filmed**. Please do not use abbreviations.	Farmington, Connecticut, USA.

TITLE:

A Finite Element Approach for Locating the Center of Resistance of Maxillary Teeth

AUTHORS AND AFFILIATIONS:

Bill Luu[1], Edward Anthony Cronauer[2], Vaibhav Gandhi[1], Jonathan Kaplan[3], David M. Pierce[3,4], Madhur Upadhyay[1]

[1]Division of Orthodontics, University of Connecticut Health, Farmington, CT, USA
[2]Private Practice, Miami, Florida, USA.
[3]Departments of Biomedical Engineering, University of Connecticut, Storrs, CT, USA
[4]Department of Mechanical Engineering, University of Connecticut, Storrs, CT, USA

Corresponding Author:

Madhur Upadhyay (madhurup@yahoo.com, maupadhyay@uchc.edu)

Email Addresses of Co-authors:

Bill Luu (bill.luu@gmail.com)
Edward Cronauer (edward.cronauer@gmail.com)
Vaibhav Gandhi (gandhi@uchc.edu)
Jonathan Kaplan (JKaplan@gmail.com)
David Pierce (david.pierce@uconn.edu)

KEYWORDS:

orthodontics, center of resistance, maxillary teeth, three-dimensional, cone beam computed tomography, mimics, 3Matic, finite element analysis

SUMMARY:

This study outlines the necessary tools for utilizing low-dose three-dimensional cone beam-based patient images of the maxilla and maxillary teeth to obtain finite element models. These patient models are then used to accurately locate the C_{RES} of all the maxillary teeth.

ABSTRACT:

The center of resistance (C_{RES}) is regarded as the fundamental reference point for predictable tooth movement. The methods used to estimate the C_{RES} of teeth range from traditional radiographic and physical measurements to in vitro analysis on models or cadaver specimens. Techniques involving finite element analysis of high-dose micro-CT scans of models and single teeth have shown a lot of promise, but little has been done with newer, low-dose, and low resolution cone beam computed tomography (CBCT) images. Also, the C_{RES} for only a few select teeth (i.e., maxillary central incisor, canine, and first molar) have been described; the rest have been largely ignored. There is also a need to describe the methodology of determining the C_{RES} in detail, so that it becomes easy to replicate and build upon.

This study used routine CBCT patient images for developing tools and a workflow to obtain finite element models for locating the C_{RES} of maxillary teeth. The CBCT volume images were

manipulated to extract three-dimensional (3D) biological structures relevant in determining the C_{RES} of the maxillary teeth by segmentation. The segmented objects were cleaned and converted into a virtual mesh made up tetrahedral (tet4) triangles having a maximum edge length of 1 mm with 3matic software. The models were further converted into a solid volumetric mesh of tetrahedrons with a maximum edge length of 1 mm for use in finite element analysis. The engineering software, Abaqus, was used to preprocess the models to create an assembly and set material properties, interaction conditions, boundary conditions, and load applications. The loads, when analyzed, simulated the stresses and strains on the system, aiding in locating the C_{RES}. This study is the first step in accurate prediction of tooth movement.

INTRODUCTION:

The center of resistance (C_{RES}) of a tooth or segment of teeth is analogous to the center of mass of a free body. It is a term borrowed from the field of mechanics of rigid bodies. When a single force is applied at the C_{RES}, translation of the tooth in the direction of the line of action of the force occurs[1,2]. The position of the C_{RES} depends not only on the tooth's anatomy and properties but also on its environment (e.g., periodontal ligament, surrounding bone, adjacent teeth). The tooth is a restrained body, making its C_{RES} similar to the center of mass of a free body. In the manipulation of appliances, most orthodontists consider the relationship of the force vector to the C_{RES} of a tooth or a group of teeth. Indeed, whether an object will display tipping or bodily movement when submitted to a single force is mainly determined by the location of the C_{RES} of the object and the distance between the force vector and the C_{RES}. If this can be accurately predicted, treatment results will be greatly improved. Thus, an accurate estimation of C_{RES} can greatly enhance the efficiency of orthodontic tooth movement.

For decades, the orthodontic field has been revisiting research regarding the location of the C_{RES} of a given tooth, segment, or arch[1-12]. However, these studies have been limited in their approach in many ways. Most studies have determined the C_{RES} for only a few teeth, leaving out the majority. For example, the maxillary central incisor and the maxillary incisor segment have been evaluated quite extensively. On the other hand, there are only a few studies on the maxillary canine and first molar and none for the remaining teeth. Also, many of these studies have determined the location of the C_{RES} based on generic anatomical data for teeth, measurements from two-dimensional (2D) radiographs, and calculations on 2D drawings[8]. In addition, some of the current literature uses generic models or three-dimensional (3D) scans of dentiform models rather than human data[4,8]. As orthodontics shifts into 3D technology for planning tooth movement, it is crucial to revisit this concept to develop a 3D, scientific understanding of tooth movement.

With technological advancements resulting in increased computational power and modeling capabilities, the ability to create and study more complex models has increased. The introduction of computed tomography scanning and cone-beam computed tomography (CBCT) scanning has thrust models and calculations from the 2D world into 3D. Simultaneous increases in computing power and software complexity have allowed researchers to use 3D radiographs to extract accurate anatomical models for use in advanced software to segment the teeth, bone, periodontal ligament (PDL), and various other structures[7,8-10,13-15]. These segmented structures

can be converted into a virtual mesh for use in engineering software to calculate the response of a system when a given force or displacement is applied to it.

This study proposes a specific, replicable methodology that can be utilized to examine hypothetical orthodontic force systems applied on models derived from CBCT images of live patients. In utilizing this methodology, investigators can then estimate the C_{RES} of various teeth and take into consideration the biological morphology of dental structures, such as tooth anatomy, number of roots and their orientation in 3D space, mass distribution, and structure of periodontal attachments. A general outline of this process is shown in **Figure 1.** This is to orient the reader to the logical process involved in generation of 3D tooth models for locating the C_{RES}.

PROTOCOL:

An institutional review board exemption was obtained for evaluating CBCT volumes archived in the Division of Oral and Maxillofacial Radiology (IRB No. 17-071S-2).

1. Volume selection and criteria

1.1. Acquire a CBCT image of the head and face[16].

1.2. Examine the image for tooth alignment, missing teeth, voxel size, field of view, and overall quality of the image.

1.3. Make sure the voxel size is not larger than 350 µm (0.35 mm).

2. Segmentation of the teeth and bone

2.1. Load the raw DICOM files of the CBCT image into Mimics software for segmentation (**Figure 2**). Click **Image > Crop Project**. Crop the image to include only the maxilla and maxillary teeth.

NOTE: The field of view should be large enough to capture the maxilla and maxillary teeth. Make sure the image includes the tooth crowns, the hard palate up to the nasal floor, maxillary sinuses, facial surfaces of the maxillary teeth, and the posterior extent of the hard palate and maxillary tuberosity.

2.2. Right click on the tab for **Mask** and create a **New Mask** for the image. Rename the mask as UL1, UL2, …, UL7 for the left side and UR1, UR2, …, UR7 for the right side, based on the tooth of interest.

2.3. Identify the tooth of interest on the masked CBCT image (see views). Use the **Clear Mask** tool to erase the mask. The software might be unable to distinguish between the teeth and bone because the gray values of the two are similar.

NOTE: The thresholding tool in Mimics is unable to segment the teeth and bone separately. Therefore, a different method for segmentation is required.

2.4. Click on the **Multiple Slice Edit** tool (Ctrl + M). Select the view (**Axial**, **Coronal**, or **Sagittal**). Manually highlight (i.e., draw) some of the slices as deemed necessary.

NOTE: Highlighting more slices adds more detail to the structure.

2.5. Click the **Interpolate** tool to fill up the volume for the skipped slices and apply.

2.6. Generate the 3D volume for the tooth by right-clicking on the mask and selecting the option to calculate the 3D volume.

2.7. Repeat steps 2.2–2.6 for each tooth of the maxillary arch.

2.8. Select all the 3D maxillary teeth, UL7–UR7. Right-click to select **Smoothing**. Set the smoothing factor to 0.4 and iterations to 4.

2.9. To segment the maxillary bones right-click on the tab for **Mask**. Create a new mask for the image.

2.10. From the drop-down menu for predefined threshold sets, select **Custom**. Adjust the threshold value to include the complete maxillary bone. Be sure to check the **Fill Holes** box before applying the threshold.

NOTE: Small holes of ≤1 mm in the cortical bone are acceptable, because they can be removed easily in later stages.

2.11. Click on the **Dynamic Region Growing Tool** to fill the large holes visible in the mask. Select the maxillary bone mask as the target for the tool in addition to selecting the **Multiple Layer** box. Use 50 for the Min and 150 for Max values. Hold down the Control key while clicking on the areas of cortical bone that were not highlighted in the mask.

2.12. Right-click on the maxillary bone mask for the **Smooth Mask** function. Repeat this step 3x for best results.

2.13. Generate the 3D volume for the maxilla by right-clicking on the mask and selecting the option to calculate the 3D volume.

2.14. Select the 3D maxillary bone. Right-click to select smoothing. Set the smoothing factor to ~0.4 and iterations to 4.

2.15. Select the 3D maxillary bone and right-click to select **Wrap**. Set 0.2 mm for the smallest detail and 1 mm for the gap closing distance. Check the **Protect Thin Walls** option. Press **Ok**.

2.16. Rename the 3D maxillary bone "Maxilla".

3. Cleaning and meshing

3.1. Select the 3D objects and copy (**Ctrl + C**).

3.2. Open the 3matic software, and paste (**Ctrl + V**) the selected 3D objects. They will appear in the object tree and work area of 3matic as a 3D structure (**Figure 3**).

3.3. Click the **Fix** tab from the tool bar and use the **Smooth** option. Under the **Operations** box select the desired 3D object(s) or entities and **Apply** the default parameters.

3.4. Click the **Finish** tab from the tool bar and use the **Local Smoothing** option. Under the **Operations Box** select the desired 3D object(s) or entities. Use the cursor to manually smoothen out the desired regions.

3.5. Duplicate the teeth. On the object tree select all teeth, right-click, and select **Duplicate**.

3.6. Select **All Duplicated Teeth**, group, and name the folder "group 1". The original set will serve as the final teeth for analysis.

3.7. For the duplicated teeth in group 1, click the **Curve Module** and the **Create Curve** option. Manually draw a curve around the cementoenamel junction (CEJ) for all duplicated teeth.

3.8. Select the **Curve**, **Contour**, and **Border** entities under the **Smooth Curve** option.

3.9. Separate the crown and root surfaces into their own parts by selecting the **Split Surfaces by Curves** option and left-clicking on the 3D object to select.

3.10. Generate the PDL from the root structure of the tooth by splitting the tooth into root and crown at the CEJ.

3.10.1. Duplicate the 3D objects from group 1 (generated in step 3.6) as group 2. For group 2, in the object tree box, click on the **Object**. From the surface list delete the crown surface. Perform this step for all objects in group 2.

3.10.2. For group 2, click on **Design Module > Hollow**. Apply the desired parameters (**Table 1**).

3.10.3. Click on the **Fix Module > Fix Wizard**. Click on the individual parts, update, and follow the given directions.

3.10.4. Repeat step 3.10.3 for all parts. Rename all parts in the group 2 as "UL1_PDL" to "UL7_PDL" and "UR1_PDL" to "UR7_PDL".

3.11. In group 1, from the object tree box, click on **Object**. From the surface list delete the root surface.

3.12. Select **Fill Hole Normal** option and select the contour. Click on **Bad Contour** and **Apply**. The entire space will be filled.

3.13. Select the **Design Module > Local Offset** and select the entire crown surface. Check the following options: **Direction** (select external), **Offset Distance** (select 0.5), and **Diminishing Distance** (select 2.0). Apply.

3.14. Repeat step 3.13.

3.15. Repeat steps 3.11–3.14 for each tooth of the maxillary arch.

3.16. Remesh (**Figure 3**)

3.16.1. Click the **Remesh Module > Create Non-Manifold Assembly > Main Entity > Maxilla** from the object tree. Select intersecting entity for all objects from 3.4 (original teeth) and select **Apply**.

3.16.2. Click the **Remesh Module**. Split the non-manifold assembly.

3.16.3. Repeat steps 3.16.1–3.16.2 using an intersecting entity as all objects from group 1 and **Apply**.

3.16.4. As an optional step, only if required, select the **Finish Module > Trim > Entity > Maxilla**. Select the excess structure (i.e., noise) and **Apply**.

3.16.5. Click the **Fix Module > Fix Wizard > Maxilla > Update**. Follow the directions given.

3.16.6. Repeat step 3.16.1 using an intersecting entity as all objects from group 2 and **Apply**.

3.16.7. Click the **Remesh Module > Adaptive Remesh**. Select all intersecting entities from 3.16.6 and **Apply**.

3.16.8. Click the **Remesh Module > Split Non-manifold Assembly**.

3.16.9. Click the **Remesh Module > Create Non-manifold Assembly > Main Entity > Individual Object** (PDL) from group 2 from the object tree. Select **Intersecting Entity > Select Respective Object** from step 3.4 (corresponding to that tooth type) and **Apply**.

3.16.10. Click **Remesh Module > Adaptive Remesh**. **Select** the intersecting entity from 3.16.9 and **Apply**.

3.16.11. Click t**Remesh Module > Split Non-manifold Assembly**.

3.16.12. Repeat steps 3.16.9–3.16.11 for each tooth.

3.17. Click the **Remesh Module > Quality Preserving Reduce Triangles**. In the object tree select all entities (i.e., teeth, PDLs, and Maxilla) and **Apply**.

3.18. Click **Remesh Module > Create Volume Mesh > Select Entity**. Choose **Mesh Parameters**.

3.19. Repeat step 3.18 for all entities (i.e., teeth, PDLs, and Maxilla).

3.20. Manually export the input(.inp) files from 3Matic to Abaqus (**Figure 4**).

4. Finite element analysis

NOTE: All custom Python scripts can be found in the supplemental attachments. They have been generated using the macro manager function in Abaqus.

4.1. Preprocessing setup

4.1.1. Open Abaqus and select **Standard Model**. Click **File > Set the Work Directory > Select Location for File Storage**.

4.1.2. Click **File > Run Script** and select **Model_setup_Part1.py**

4.1.3. In the Model Directory specify the file path to load .inp files on Abaqus.

4.1.4. Click on **Models > Simulation > Parts > Maxilla > Surfaces**.

4.1.5. Name the surface in the dialog box "UL1 _socket".

4.1.6. Under **Select the Region of the Surface** choose **By Angle**. Add "15" as the angle.

4.1.7. Make sure all areas of the socket are selected. Press **Done** when completed.

4.1.8. Repeat steps 4.1.4–4.1.7 for the individual sockets.

4.1.9. Click on **Models > Simulation > Parts**. Then select **UL1 > Surfaces**. Name the surface "UL1".

4.1.10. Under **Select the Region of the Surface** opt for "Individually". Select the tooth on the screen and press **Done**.

4.1.11. Repeat steps 4.1.9–4.1.10 for all teeth.

4.1.12. Click on **Models > Simulation > Parts**. Then select **UL1_PDL > Surfaces**. Name the surface "UL1_PDL_inner".

4.1.13. Under **Select the Region of the Surface** choose **By Angle**. Add "15" as the angle.

NOTE: If an error is found during the final simulation, reduce the angle and reselect the surface.

4.1.14. Make sure the entire inner surface area of the PDL is selected. Press **Done** when completed.

4.1.15. **Select UL1_PDL > Surfaces**. Name the surface "UL1_PDL_outer".

4.1.16. Under **Select the Region of the Surface** choose **By Angle**. Add "15" as the angle.

NOTE: If an error is found during the final simulation, reduce the angle and reselect the surface.

4.1.17. Make sure the entire outer surface area of the PDL is selected. Press **Done** when completed.

4.1.18. Repeat steps 4.1.13–4.1.19 for all PDLs.

4.1.19. Click on **File > Run Script** and select **Model_setup_Part2.py**

4.1.20. Click on **Models > Simulation > BCs**. Name **BC_all**, then select **Step** as **Initial**. Under category, select "Mechanical", and under "Types of Selected Step" select "Displacement/Rotation". Press **Continue**.

4.1.21. Under **Select Regions for the Boundary Condition** select **By Angle**. Add "15" as the angle. Check **Create Set**. Select individual sockets for the 14 teeth. Press **Done**.

NOTE: This helped simulate instantaneous tooth movement.

4.1.22. Click on **Models > Simulation > Assembly > Sets > Create Set**. Name the set "U1_y_force".

4.1.23. In **Select the Nodes for the Set** choose **Individually**.

NOTE: A one Newton concentrated force was applied on a randomly selected tooth node in either the positive Y direction (simulating a distalization force) or the positive Z direction (simulating an intrusive force).

4.1.24. Select a node at the center of the crown on the buccal surface of the upper central incisor (U1) and press **Done**.

4.1.25. Click **Sets > Create Set**. Name the set "U1_z_force".

4.1.26. Repeat steps 4.1.23–4.1.24.

4.1.27. Repeat steps 4.1.22–4.1.26 for all teeth.

NOTE: Before a set is generated for a particular tooth as in 4.1.25, go to **Instance > Resume** for that tooth.

4.2. **Model set up**

4.2.1. Click on **Models > Simulation > Assembly > Instances**. Select **All Instances** and click **Resume**.

4.2.2. Click on **Tools > Query > Point/Node**. Select a node at the center of a randomly selected central incisor and press **Done**.

4.2.3. Under the command center at the bottom of the page, copy the X, Y, and Z coordinates of the node selected in step 4.2.2.

4.2.4. Under the vertical tool bar select **Translate Instance** and select the entire assembly (i.e., all instances) on the screen. Press **Done**.

4.2.5. In the **Select a Start Point for the Translation Vector** box, paste the copied coordinates in step 4.2.3 or enter the X, Y, Z values. Click **Enter**.

4.2.6. Under **Select an End Point for the Translation Vector or enter X,Y,Z:** enter the coordinates "0.0", "0.0", and "0.0". Click **Enter**.

4.2.7. For **Position of Instance**, press **Ok**.

4.2.8. Click on **Tools > Query > Point/Node** and select a node directly above the midline of the center incisors. Enter **Done**.

4.2.9. Under the command center at the bottom of the page, copy the X, Y, and Z coordinates of the node selected in step 4.2.8.

4.2.10. Under the vertical tool bar select **Translate Instance** and select the entire assembly (i.e., all instances) on the screen. Enter **Done**.

4.2.11. Paste the copied coordinates into the **Select a Start Point for the Translation Vector – or Enter X,Y,Z** box. Click **Enter**.

4.2.12. Under **Select an End Point for the Translation Vector – or enter X,Y,Z:** insert the coordinates as copied in step 4.2.9. Change the X coordinate to 0.0. Click **Enter**.

4.2.13. For **Position of Instance**, press **Ok**.

4.2.14. Click on **File > Run Script** and select **Model_setup_Part3.py**. Insert or change material properties.

4.2.15. Click on **Models > Simulation > Materials** and click **Bone/PDL/Tooth**. Insert tissue specific properties.

4.2.16. Click on **File > Run Script** and select **Functions.py**.

4.3. Processing the model

4.3.1. Click on **File > Run Script** and select **Job_submission.py**.

NOTE: The job module is where the user sets up one or more actions on the model, and job manager is where model analysis is started, progress is shown, and completion is noted.

4.3.2. In the dialog box titled **Suppress All**, enter the sides (L or R) of the teeth based on constraints (Under **Models> Simulation >Constraints**). Press **Ok**.

4.3.3. In the dialog box titled **Job Submission** enter "Y" to run the analysis for the specified tooth/teeth. Press **Ok**.

4.3.4. In the dialog box titled **Directions for Analysis** enter "Y" to specify force application. Press **Ok**.

4.4. Post processing for C_{RES} estimation

4.4.1. Choose **File > Run Script > Bulk_process.py**.

4.4.2. In the dialog box titled **Analyze Multiple Jobs** enter "Y" for the specified tooth/teeth. Press **Ok**.

4.4.3. In the dialog box titled **Directions for Analysis** enter "Y" for specifying force application. Press **Ok**.

4.4.4. In the dialog box titled **Get Input** enter **Specific Tooth Number** as outlined named Instances (e.g., UL1 or UL5, etc.). Press **Ok**.

4.4.5. Check coordinates for the **Force About Point** and **Estimated Location** in the command box. If they are not similar, then repeat steps 4.3.1–4.4.4.

NOTE: After the jobs for each step were run, a user-defined algorithm created in Python was run within the Abaqus interface to analyze the reaction force system and subsequent moments created as a result of load application. The algorithm automatically suggests a new node location to apply the load such that a moment of near-zero magnitude is created within the force system. This proceeds in an iterative process, until the node location that creates a moment closest to zero when a force is applied through it is found or estimated. The algorithm is described in detail in the Discussion section.

REPRESENTATIVE RESULTS:

In order to verify segmentation and manual outlining as described in the Procedures section (step 2), a maxillary first molar was extracted from a dry skull, and a CBCT image was taken. The image processing and editing software Mimics was used for manually outlining the tooth as described in step 2. Subsequently, meshing was performed, the segmented models were cleaned with 3matic software, and they were imported to Abaqus for analysis. We did not find any significant difference in the linear and volumetric measurements made on the FE model of the tooth and the actual tooth measured in the lab (**Supplemental Document 4**).

To verify the validity of the user-defined algorithm in determining the C_{RES} of an object, a simplified model of a beam encased within a sheath was used in the initial stages of script creation (**Figure 5A**). The steel encasement was constrained to three degrees of freedom of displacement, and the nodes at the beam/sheath interface were tied together. Nodes for force application were selected at random, and the subroutine was applied in an iterative fashion until the solution converged. In the simplified model, a length of 30 units and a width of 10 units were encased in the sheath. By following the defined algorithm and its calculations, the C_{RES} of the model beam was predicted (**Figure 5B**). This agreed with the theoretical calculations (see **Supplemental Document 3**). Thus, the validity of the user-defined algorithm was developed and verified in this simplified model and was subsequently implemented for the determination of the C_{RES} of maxillary teeth.

Table 2 shows the material properties assigned to the structures. Differences in the modelling of the material properties of the PDL and bone could affect the final location of the C_{RES} of a tooth. PDL anisotropy related to fiber orientation, differences in Poisson's ratio, loading patterns, and magnitude can also make a difference. PDL was assigned nonlinear, hyperelastic properties according to the Ogden model ($\mu1 = 0.07277$, $\alpha1 = 16.95703$, $D1 = 3 \times 10^{-7}$)[22,23]. Specific densities were also assigned = 1.85 g/cm^3 for bone; 2.02 g/cm^3 for teeth; and 1 g/cm^3 for PDL (i.e., the density of water, because PDL is mostly comprised of water)[24,25].

To standardize the force vectors and locate the position of the C_{RES}, a cartesian coordinate system was constructed (X-Y-Z) and defined by the following orientations: Y-axis (anteroposterior or labiolingual axis) oriented along the midpalatal suture with the posterior portion in the positive direction, Z-axis in the vertical direction (superio-inferior or occluso-gingival axis) with the superior or gingival portion of the model in the positive direction, and the X-axis in the transverse direction (buccolingual axis) with the buccal portion in the positive direction (**Figure 6**).

This coordinate system was applied in two ways: 1) A global coordinate system was established with its origin (O) located between the facial surfaces of the central incisors below the incisive papilla located on a line bisecting the inter-incisor and inter-molar widths in the X-Y plane; 2) Local coordinate systems were constructed with an origin 'R' for every tooth. The 'R' point specific for every tooth was defined as the geometric center on the buccal surface of the crown. This site was chosen to approximate the closest location where an operator might place a bracket to apply orthodontic forces. Representative results are shown in **Figure 7**.

The C_{RES} located relative to the global and local coordinate systems are shown in **Table 3** and **Table 4**. The locations of the C_{RES} obtained along the X coordinate when a force system was applied along the Y and Z coordinates were different from each other (**Table 5**). However, the average difference was small (0.88 ± 0.54 mm).

FIGURE AND TABLE LEGENDS:

Figure 1: Design flow chart. The three-step workflow for locating C_{RES}.

Figure 2: Layout of the Mimics software displaying maxillary teeth in all the three views (**X-Y-Z**) **and as a volumetric model.**

Figure 3: Steps involved for generating periodontal ligament (**PDL**) **utilizing non-manifold assembly of the 3matic software.** Remesh Module (**A**) Create non-manifold assembly, (**B**) Maxilla is set as the main entity, (**C**) PDL is set as the intersecting entity, (**D**) Adaptive Remesh, (**E**) Splitting the maxilla and the PDL, (**F**) Follow steps **B–F** for PDL as the main entity and the selected tooth as an intersecting entity, (**G**) Create volume mesh.

Figure 4: The Abaqus software layout.

Figure 5: Simplified model of the steel beam. (**A**) The beam encased in a steel sheath used to test the accuracy of the defined algorithm. (**B**) Location of C_{RES} of the encased beam as predicted by the defined algorithms.

Figure 6: The coordinate system for C_{RES} estimation relative to a global point of origin (O) and local point of origin (R) for each tooth. This is an illustration for the maxillary second premolar. This method was utilized for every tooth in the arch.

Figure 7: Three-dimensional representation of the C_{RES} of maxillary teeth. (**A**) Central incisor. (**B**) Lateral incisor. (**C**) Canine. (**D**) First premolar. (**E**) Second premolar. (**F**) First molar. (**G**) Second molar.

Table 1: Hollow tool parameters.

Table 2: Material properties of the finite element model.

Table 3: Three-dimensional (X-Y-Z) location of the C_{RES} of maxillary teeth in relation to the global point O.

Table 4: Three-dimensional (X-Y-Z) location of the C_{RES} of maxillary teeth in relation to a local point R for each tooth whose C_{RES} is being evaluated. Here, R is the geometric center of the buccal surface of the crown.

Table 5: Variation in the center of resistance position along the X-axis when force is applied along the Y-(Fy) and Z (Fz)-axes.

Supplemental Document 1: Python scripts of the algorithms utilized for the FEA.

Supplemental Document 2: An overview of the analysis of the force system.

Supplemental Document 3: Theoretical estimation of the center of mass of a simple beam encased in a sheath.

Supplemental Document 4: A finite element model of an extracted maxillary first molar.

DISCUSSION:

This study shows a set of tools to establish a consistent workflow for finite element analysis (FEA) of models of maxillary teeth derived from CBCT images of patients to determine their C_{RES}. For the clinician, a clear and straightforward map of the C_{RES} of the maxillary teeth would be an invaluable clinical tool to plan tooth movements and predict side effects. The finite element method (FEM) was introduced in dental biomechanical research in 1973[17], and since then has been applied to analyze the stress and strain fields in the alveolar support structures[6-12]. As evidenced by the number of steps outlined in the workflow (**Figure 1**), creating finite element models is a complex task. Therefore, certain aspects of the methodology had to be simplified.

First, tooth movement only in the alveolar socket was considered by assuming that resorption and apposition of the alveolar bone did not occur. This type of displacement is called primary[4] or instantaneous tooth movement[18]. It has been observed that the PDL is a critical entity in instantaneous tooth displacement. Bone and teeth could be reasonably assumed to be rigid to define PDL stresses for tooth movement[15]. Therefore, for this study the stress distribution was constrained within the tooth socket. The Create Boundary Condition tool allows the user to set boundary conditions for the model or apply constraints. Selected points are assigned zero degrees of freedom to ensure that the model stays rigid in that area. Consequently, the analysis time for calculating bone deformation and remeshing the solid elements of the deformed alveolar bone done in previous studies, was eliminated[19,20].

Second, an attempt to keep image resolution at moderate levels was made. CBCT image voxel size was 0.27 mm. This not only kept the radiation dosage at a minimum but also reduced the computational burden for assembling the global stiffness matrix for tetrahedral elements. However, the downside was that the CBCT resolution was insufficient to accurately and distinctly

capture the PDL on the scans. This was largely because the average PDL thickness is around 0.15 mm–0.38 mm (average: 0.2 mm)[21] and the image voxel size was 0.27 mm. This shortcoming with the CBCT scans created two issues: 1) The PDL could not be segmented on its own; and 2) Segmenting the bone and teeth using thresholding was not possible due to the lack of a distinct gray value change between the two. As a result, the software was unable to distinguish between the teeth and bone because the gray values were similar. In other words, Mimics was unable to segment the teeth and bone separately. Therefore, a different method of segmentation was developed. After attempting numerous tools, such as the region growing or split tool in Mimics, it was determined that the best way to segment the teeth was by manually highlighting the tooth structure on each slice of the CBCT. Here the Multiple Slice Edit Tool offered an efficiency advantage. Instead of having to manually highlight every slice, the user only has to highlight some of the slices. For this reason, it was the best method for segmenting the teeth, as it provided the greatest accuracy in getting good images of the anatomy of the teeth in a consistent manner.

Because Mimics was unable to segment the PDL in due to the low resolution of the CBCT images, it was necessary to grow the PDL from the root structure of the tooth. This required splitting the tooth into root and crown at the CEJ. Once grown, the constructed PDL was essentially two surfaces parallel to each other spaced 0.2 mm apart, where one surface was in intimate contact with the bone and the other with the root. It was critical that the surfaces were tied together in the finite element analysis so that a load added to a tooth was propagated through the PDL to the bone. The engineering software rejected models whose surfaces were too far apart or intersected too much, as this made connecting the surfaces impossible and invalidated the FEA model.

Third, all model surfaces were kept relatively smooth and free of small surface topography that is insignificant for the overall model analysis, such as a projection of extra bone off the buccal cortical surface. Fine elements on projections of anatomy add unnecessary complication to the mesh of the final model by decreasing the size of the elements in complicated areas of fine anatomy, thereby increasing the number of elements in the model. Smaller and more numerous elements increase the computing effort in the final finite element analysis.

The locations of the C_{RES} when the force was applied in the Y and Z directions were different, represented by the differences in their location along the X direction. However, the difference was small (**Table 5**) and was clinically as well as statistically insignificant. Therefore, the location of C_{RES} calculated in one direction may be used for the other. Previous work has also shown that when evaluated in 3D a single point for the C_{RES} is not observed[10,26,27]. Therefore, it has been suggested that instead of having a definite C_{RES} a better terminology could be "radius of resistance". This difference can be attributed to a number of factors, such as root morphology, boundary conditions, material properties, and point of load application.

Analysis of Force Systems using Custom Algorithms

The mathematical concepts, theoretical derivations, and computer simulations for locating the C_{RES} of a tooth have been previously described in detail[27-30]. In order to analyze the force systems created by the various loads applied and to predict the C_{RES} for the teeth, a custom algorithm was

written and run within Abaqus (see supplemental coding files). This algorithm was written using Python, accepts data from the FEA software output database (.odb file) as input, processes the data, and provides values for the moments created in the system by the applied load. Additionally, it estimates node locations that result in the generation of a lower moment within the system . This allows the user to run the simulation in an iterative fashion until the estimations converge onto a single location.

The algorithm accesses the nodal coordinates, the total displacement of each node, and the reaction forces at each node as a result of the applied load in each step. Reaction forces in the same direction as the original load application and reaction forces in the opposite direction are summed at each of the nodes in the system to determine the aggregate force vectors acting on the tooth during the simulation. Resulting moments are calculated in relation to the point of force application for each reaction force at each node and are also summed in the same fashion as the reaction forces. Thus, an aggregate force vector in the same direction as the original load application and the resulting moment created by that force vector about the point of force application is calculated, as well as the force vector in the opposite direction and its resulting moment. Because the system is in static equilibrium, the sum of all forces and moments equals zero. However, the breakdown of the reaction forces and moments in this fashion allows for the calculation of the effective locations where these aggregate forces act as pivot points in the system, and the center point between these pivot points provides an approximation of a point of force application that is closer to the C_{RES}.

In order to perform these calculations, the magnitude of the resulting moments is divided by the magnitude of their respective forces to give the magnitude of the distance (R vector) from the pivot points to the point of force application. The direction of the R vector is determined via a cross product of the moment and force vectors, where all must be orthogonal to each other, and the unit vector is determined by dividing by the magnitude of the cross product. The unit vector R is multiplied by the magnitude of the R vector previously calculated to yield the overall estimation in 3D space of the coordinates of each pivot point relative to the original point of force application. The midpoint between these two vectors provides the estimation for the location of the next point of force application in the following iteration. Additional information is attached in **Supplemental Document 2**.

The estimation of the C_{RES} is determined when the resulting moments in the system add to approximately zero. For the current study, this determination is made by finding the lowest positive and negative X-components of the calculated moments and averaging the two. Due to the randomly generated location of the nodes, and the inherent distance between any two nodes (0.5 mm), it is difficult to find a location where a precise zero moment is generated (**Table 5**).

Limitations

Despite our best efforts, there are some limitations to this study. First, because the PDL could not be visualized on the CBCT, it could not be segmented on its own and was generated from the root surface of the tooth at a uniform thickness of 0.2 mm. Finite element studies have shown that uniform versus nonuniform modeling affects the outcome of the FEA, and that nonuniform

modeling is superior[30,31]. Second, the number of steps to create an accurate model was lengthy. This is a limitation in terms of how quickly models can be made, which limits the possibility of using these tools for personal treatment plans for patients on a case by case basis. Additionally, the software required to generate these models is expensive and limited to the resources available at an educational institution or a large business. Further, once the models were made, very powerful computing was necessary to run the FEA. Thus, this method cannot be a viable treatment planning tool until the technology necessary is widely available.

Future research should focus on using these models to perform finite element analyses on the maxillary teeth to determine the C_{RES} for the arch and groups of teeth, especially those groups of teeth typically manipulated in orthodontics, such as the anterior segment in an extraction case or a posterior segment for intrusion in open bite patients. Once the C_{RES} is determined for these models, additional models should be developed from additional CBCT images to add to the existing data. With a sufficient data pool of C_{RES} locations, heat maps could be generated to indicate a general position of the C_{RES} that could serve as an invaluable reference for clinicians.

ACKNOWLEDGMENTS:
The authors would like to acknowledge the Charles Burstone Foundation Award for supporting the project.

DISCLOSURES:
The authors have nothing to disclose.

REFERENCES:
1. Smith, R. J., Burstone, C. J. Mechanics of tooth movement. *American Journal of Orthodontics*. **85** (4), 294–307 (1984).
2. Christiansen, R. L., Burstone, C. J. Centers of rotation within the periodontal space. *American Journal of Orthodontics*. **55** (4), 353–369 (1969).
3. Tanne, K., Nagataki, T., Inoue, Y., Sakuda, M., Burstone, C. J. Patterns of initial tooth displacements associated with various root lengths and alveolar bone heights. *American Journal of Orthodontics and Dentofacial Orthopedics*. **100** (1), 66–71 (1991).
4. Burstone, C. J., Pryputniewicz, R. J. Holographic determination of centers of rotation produced by orthodontic forces. *American Journal of Orthodontics*. **77** (4), 396–409 (1980).
5. Dermaut, L. R., Kleutghen, J. P., De, Clerck, H. J. Experimental determination of the Cres of the upper first molar in a macerated, dry human skull submitted to horizontal headgear traction. *American Journal of Orthodontics and Dentofacial Orthopedics*. **90** (1), 29–36 (1986).
6. Tanne, K., Sakuda, M., Burstone, C. J. Three-dimensional finite element analysis for stress in the periodontal tissue by orthodontic forces. *American Journal of Orthodontics and Dentofacial Orthopedics*. **92** (6), 499–505 (1987).
7. Meyer, B. N., Chen, J., Katona, T. R. Does the Cres depend on the direction of tooth movement? *American Journal of Orthodontics and Dentofacial Orthopedics*. **137** (3), 354–361 (2010).
8. Kojima, Y., Fukui, H. A finite element simulation of initial movement, orthodontic movement, and the centre of resistance of the maxillary teeth connected with an archwire. *European Journal of Orthodontics*. **36** (3), 255–261 (2014).

9. Reimann, S., Keilig, L., Jäger, A., Bourauel, C. Biomechanical finite-element investigation of the position of the centre of resistance of the upper incisors. *European Journal of Orthodontics*. **29** (3), 219–224 (2007).
10. Viecilli, R. F, Budiman, A., Burstone, C. J. Axes of resistance for tooth movement: Does the Cres exist in 3-dimensional space? *American Journal of Orthodontics and Dentofacial Orthopedics*. **143** (2), 163–172 (2013).
11. Ammar, H. H., Ngan, P., Crout, R. J., Mucino, V. H., Mukdadi, O. M. Three-dimensional modeling and finite element analysis in treatment planning for orthodontic tooth movement. *American Journal of Orthodontics and Dentofacial Orthopedics*. **139** (1), 59–71 (2011).
12. Sia, S., Koga, Y., Yoshida, N. Determining the center of resistance of maxillary anterior teeth subjected to retraction forces in sliding mechanics. An in vivo study. *Angle Orthodontics*. **77** (6), 999–1003 (2007).
13. Cattaneo, P. M., Dalstra, M., Melsen, B. Moment-to-force ratio, center of rotation, and force level: a finite element study predicting their interdependency for simulated orthodontic loading regimens. *American Journal of Orthodontics and Dentofacial Orthopedics*. **133** (5), 681–689 (2008).
14. Tominaga, J.Y. et al. Effect of play between bracket and archwire on anterior tooth movement in sliding mechanics: A three-dimensional finite element study. *Journal of Dental Biomechanics*. **3** (1758736012461269), (2012).
15. Cai, Y., Yang, X., He, B., Yao, J. Finite element method analysis of the periodontal ligament in mandibular canine movement with transparent tooth correction treatment. *BMC Oral Health*. **15** (106), (2015).
16. Pauwels, R., Araki, K., Siewerdsen, J. H., Thongvigitmanee, S. S. Technical aspects of dental CBCT: state of the art. *Dentomaxillofacial Radiology.* **44** (1), 20140224 (2015).
17. Farah, J. W., Craig, R. G., Sikarskie, D. L. Photoelastic and finite element stress analysis of a restored axisymmetric first molar. *Journal of Biomechanics*. **6** (5), 511–520 (1973).
18. van Driel, W. D., van Leeuwen, E. J., Von den Hoff, J. W., Maltha, J. C., Kuijpers-Jagtman, A. M. Time-dependent mechanical behavior of the periodontal ligament. *Proceedings of the Institution of Mechanical Engineers, Part H: Journal of Engineering in Medicine.* **214** (5), 497–504 (2000).
19. Bourauel, C. et al. Simulation of orthodontic tooth movements. A comparison of numerical models. *Journal of Orofacial Orthopedics.* **60** (2), 136–151 (1999).
20. Schneider, J., Geiger, M., Sander, F. G. Numerical experiments on longtime orthodontic tooth movement. *American Journal of Orthodontics and Dentofacial Orthopedics*. **121** (3), 257–265 (2002).
21. Ten Cate, A. R. *Oral histology, development, structure and function (5th ed)*. St. Louis Mosby (1998).
22. McCormack, S. W., Witzel, U., Watson, P. J., Fagan, M. J., Gröning, F. The Biomechanical Function of Periodontal Ligament Fibres in Orthodontic Tooth Movement. Agarwal S., ed. *PLoS One.* **9** (7), e102387 (2014).
23. Huang, H., Tang, W., Yan, B., Wu, B., Cao, D. Mechanical responses of the periodontal ligament based on an exponential hyperelastic model: a combined experimental and finite element method. *Computer Methods in Biomechanics and Biomedical Engineering*. **19** (2), 188–98 (2016).

24. Yang, J. A new device for measuring density of jaw bones. *Dentomaxillofacial Radiology*. **31** (5), 313–316 (2002).
25. Gradl, R. et al. Mass density measurement of mineralized tissue with grating-based X-ray phase tomography. *Plos One.* **11** (12), e01677979 (2016).
26. Jiang, F., Kula, K., Chen, J. Estimating the location of the center of resistance of canines. *Angle Orthodontics*. **86** (3), 365–371 (2016).
27. Nyashin Y. et al. Center of resistance and center of rotation of a tooth: experimental determination, computer simulation and the effect of tissue nonlinearity. *Computer Methods in Biomechanics and Biomedical Engineering*. **19**, 229–239 (2016).
28. Toms, S. R., Eberhardt, A. W. A nonlinear finite element analysis of the periodontal ligament under orthodontic tooth loading. *American Journal of Orthodontics and Dentofacial Orthopedics*. **123** (6), 657–665 (2003).
29. Osipenko MA, Nyashin MY, Nyashin YI. Centre of resistance and centre of rotation of a tooth: the definitions, conditions of existence, properties. *Russian Journal of Biomechanics*. **3** (1), 5–15 (1999).
30. Dathe, H., Nägerl, H., Dietmar, K. M. A caveat concerning center of resistance. *Journal of Dental Biomechanics.* **4** (1758736013499770), (2013).
31. Hohmann, A. et al. Influence of different modeling strategies for the periodontal ligament on finite element simulation results. *American Journal of Orthodontics and Dentofacial Orthopedics*. **139** (6), 775–83 (2011).

Figure 1

Click here to access/download;Figure;Figure 1.tiff

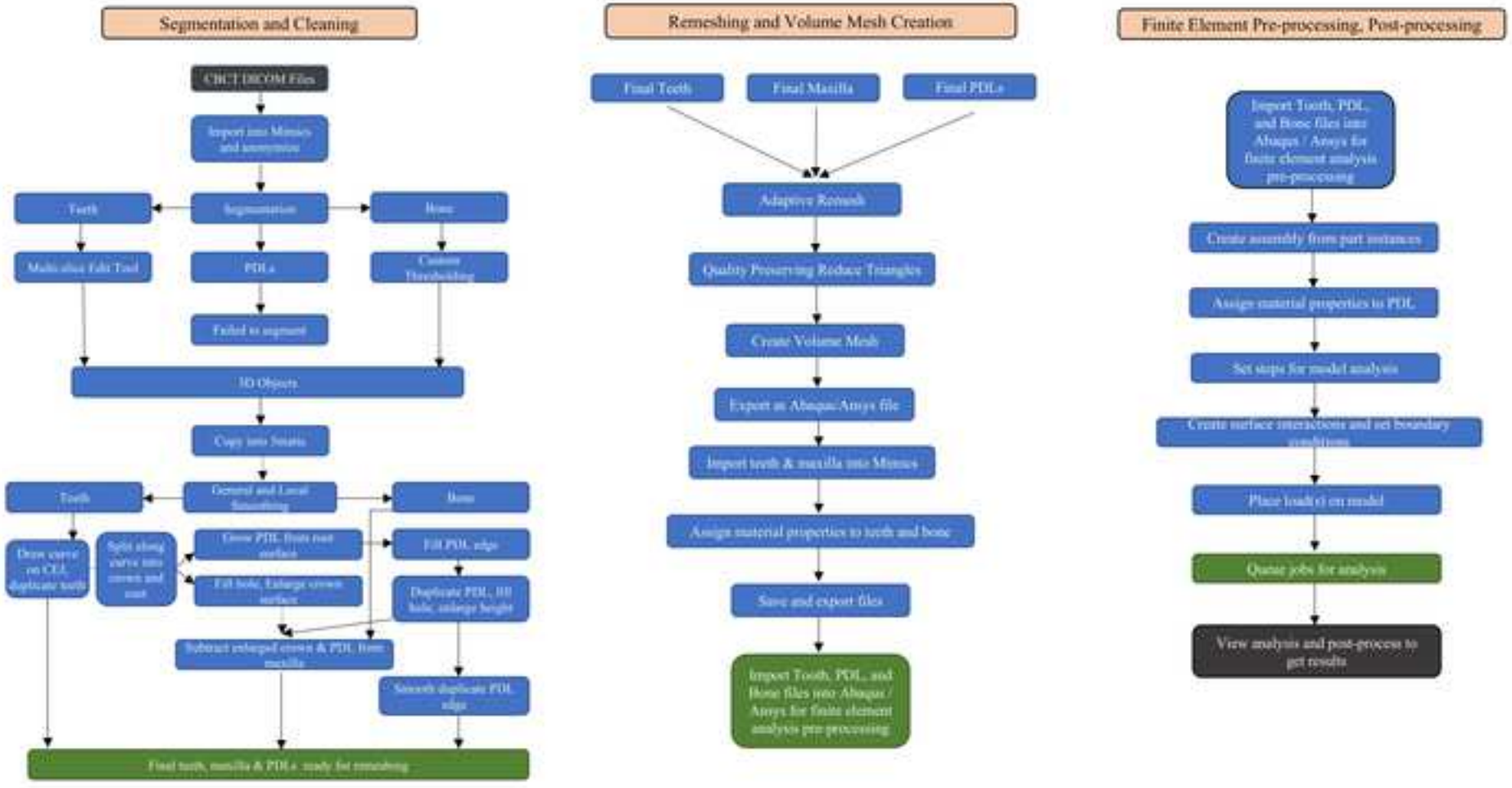


Figure 2

Click here to access/download;Figure;Figure 2.tiff

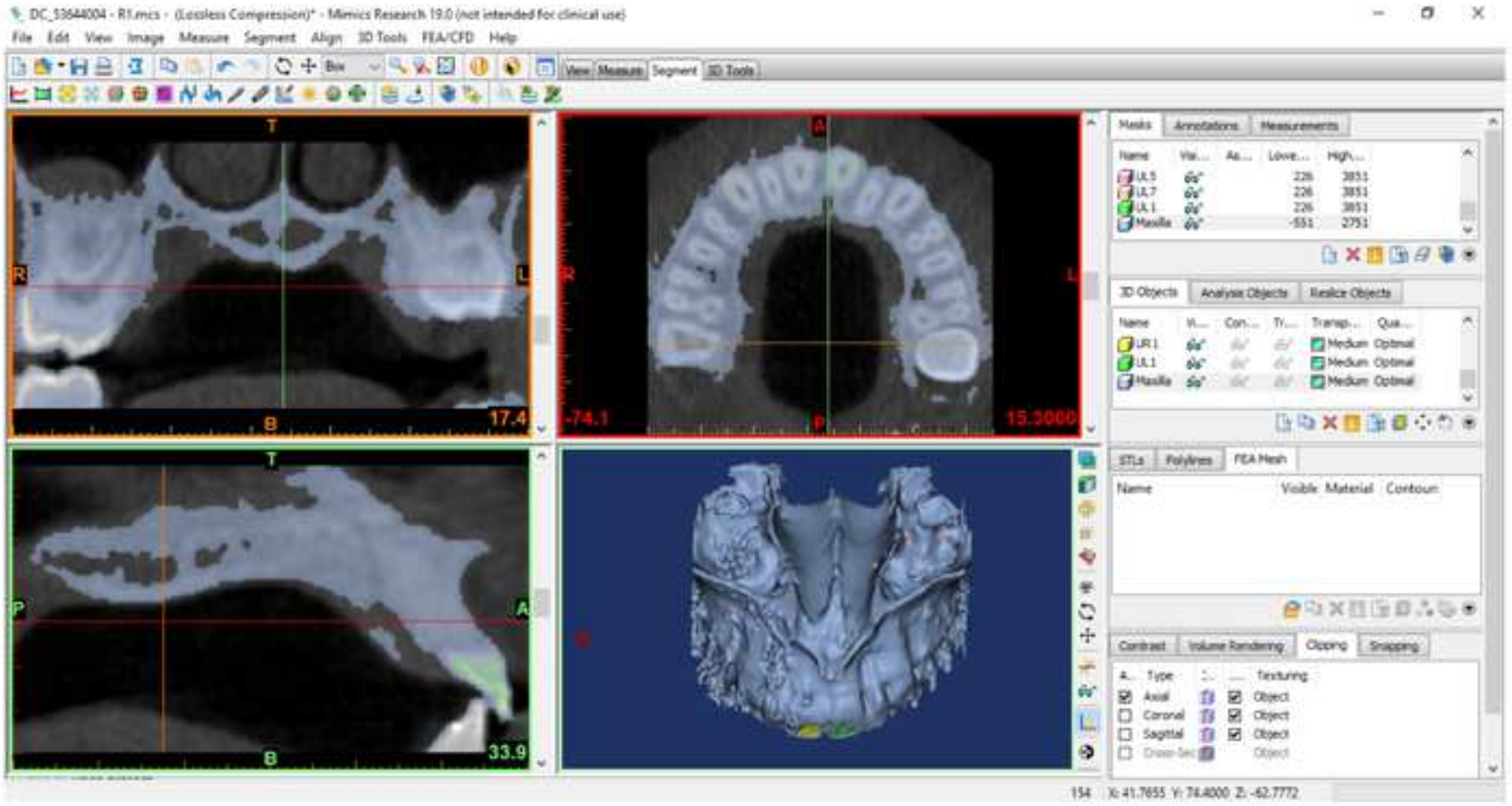


Figure 3

Click here to access/download;Figure;Figure 3.tiff

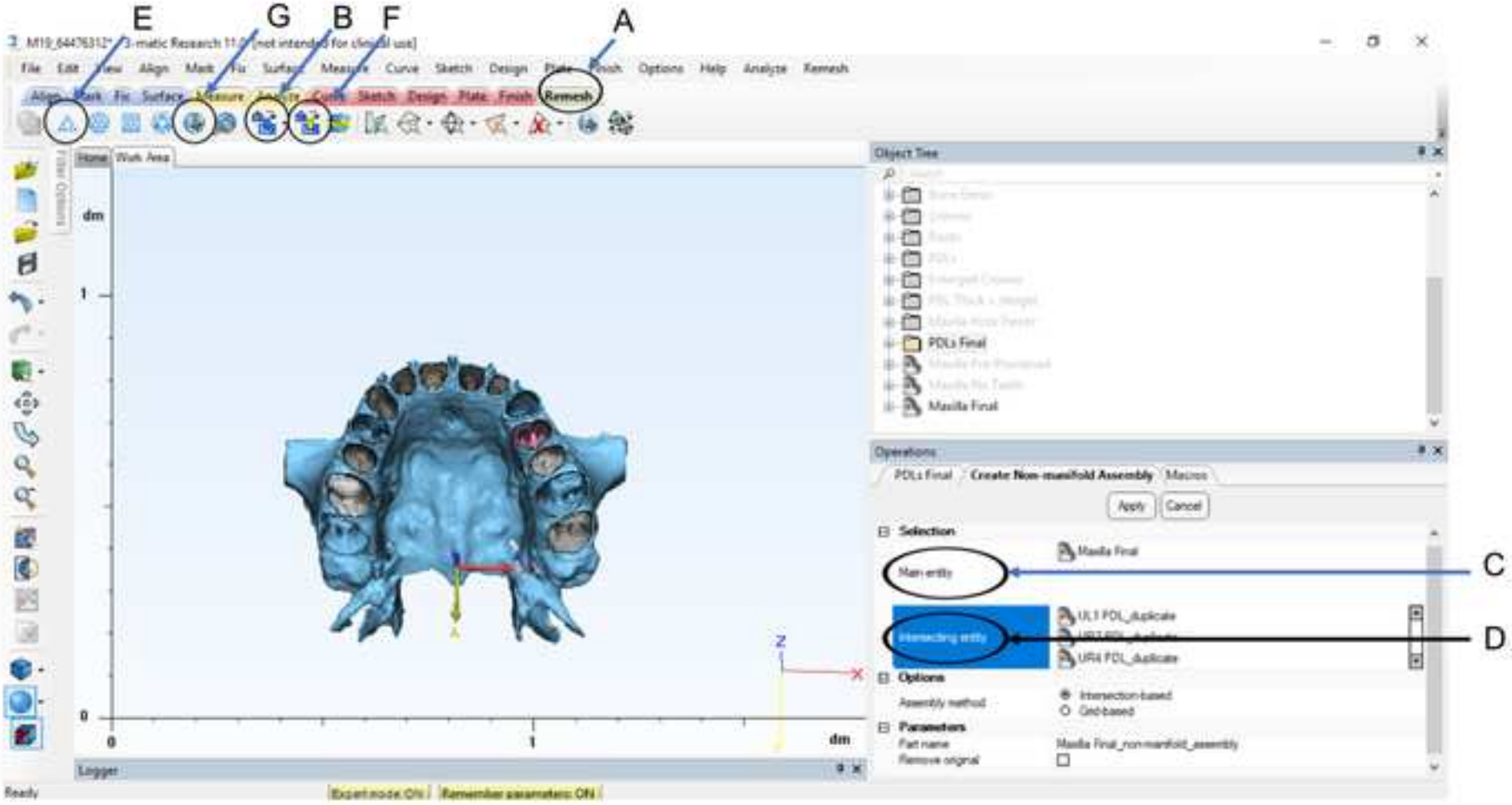


Figure 4

Click here to access/download;Figure;Figure 4.tiff

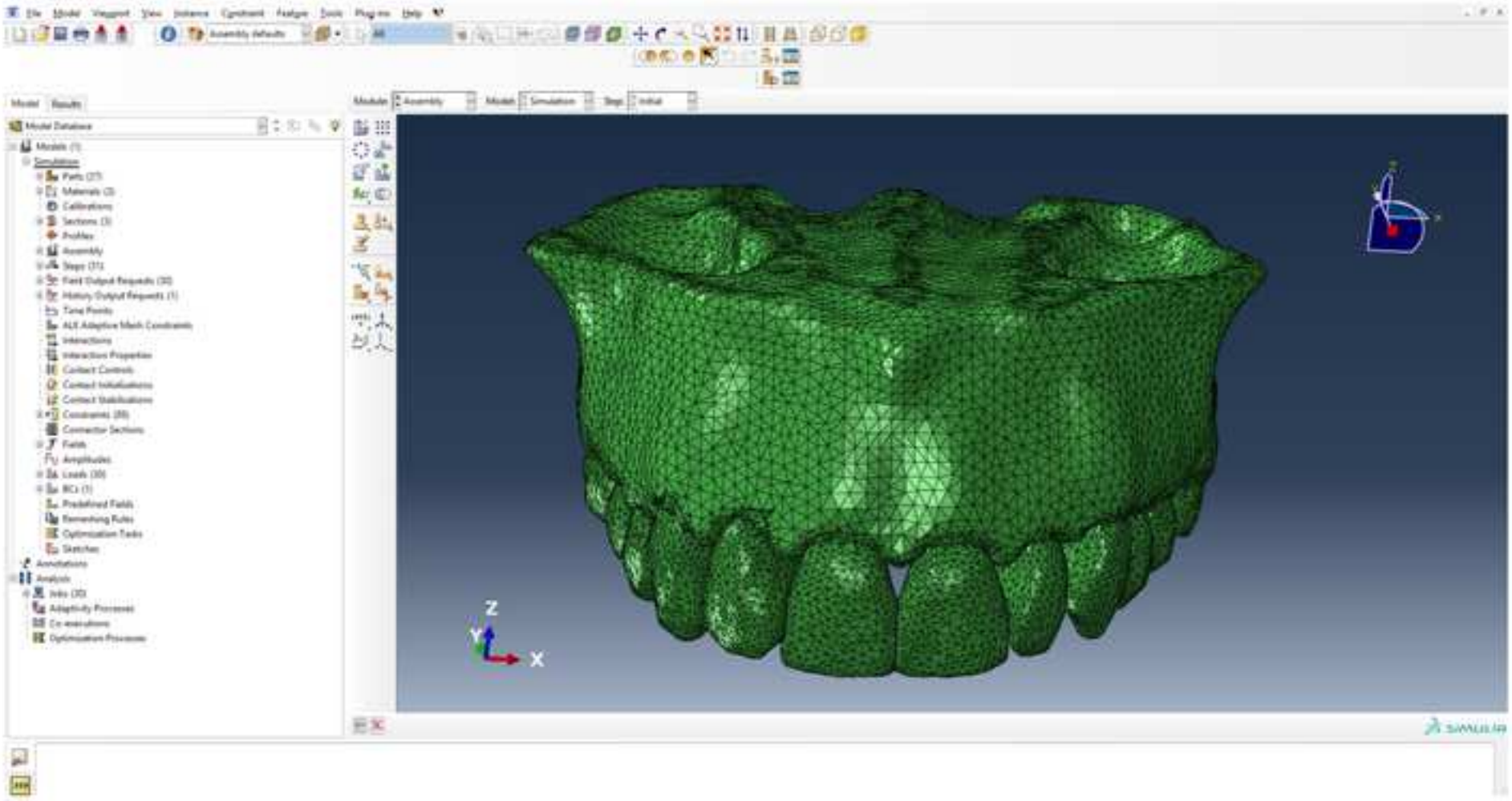


Figure 5

Click here to access/download;Figure;Figure 5[A,B].tiff

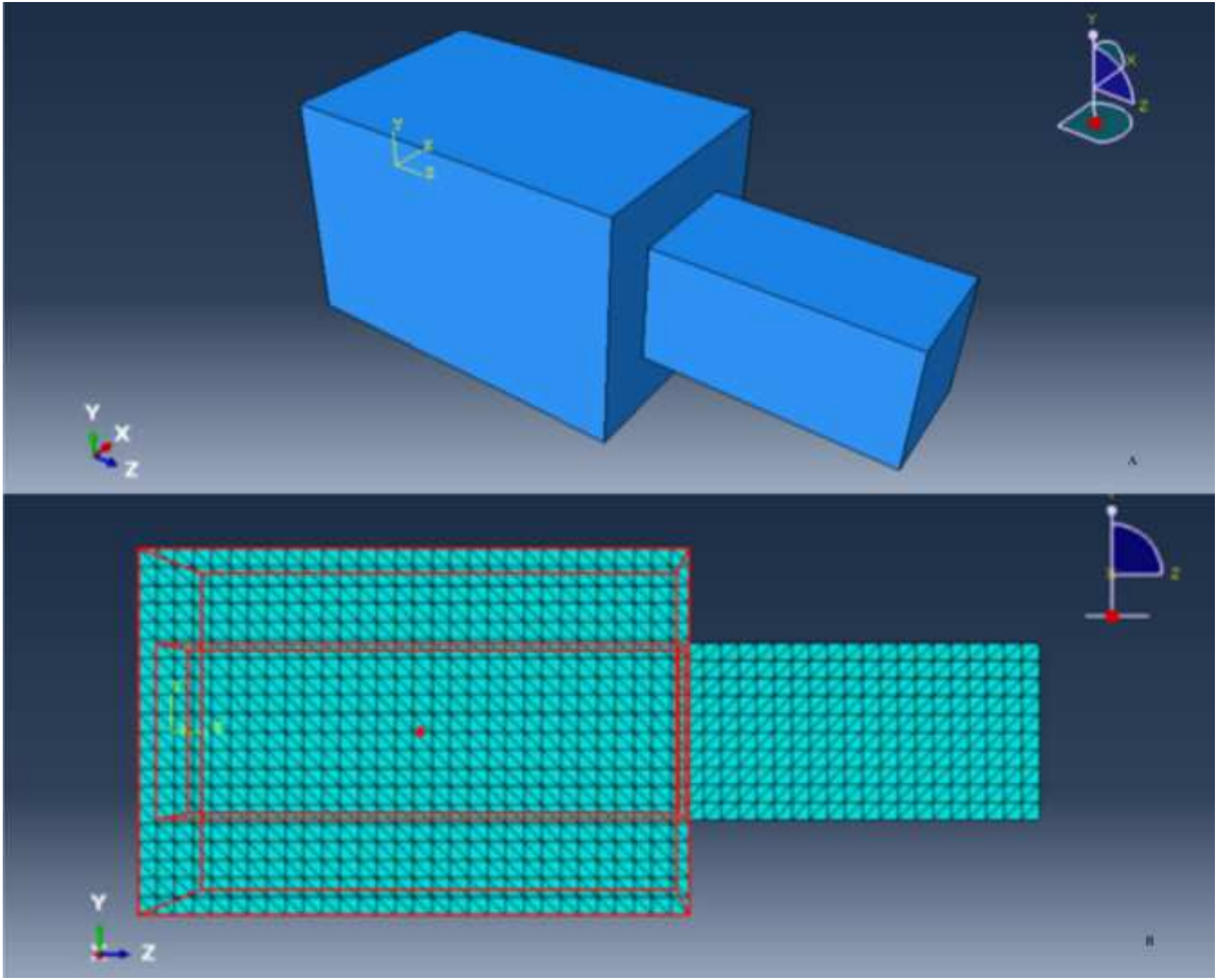
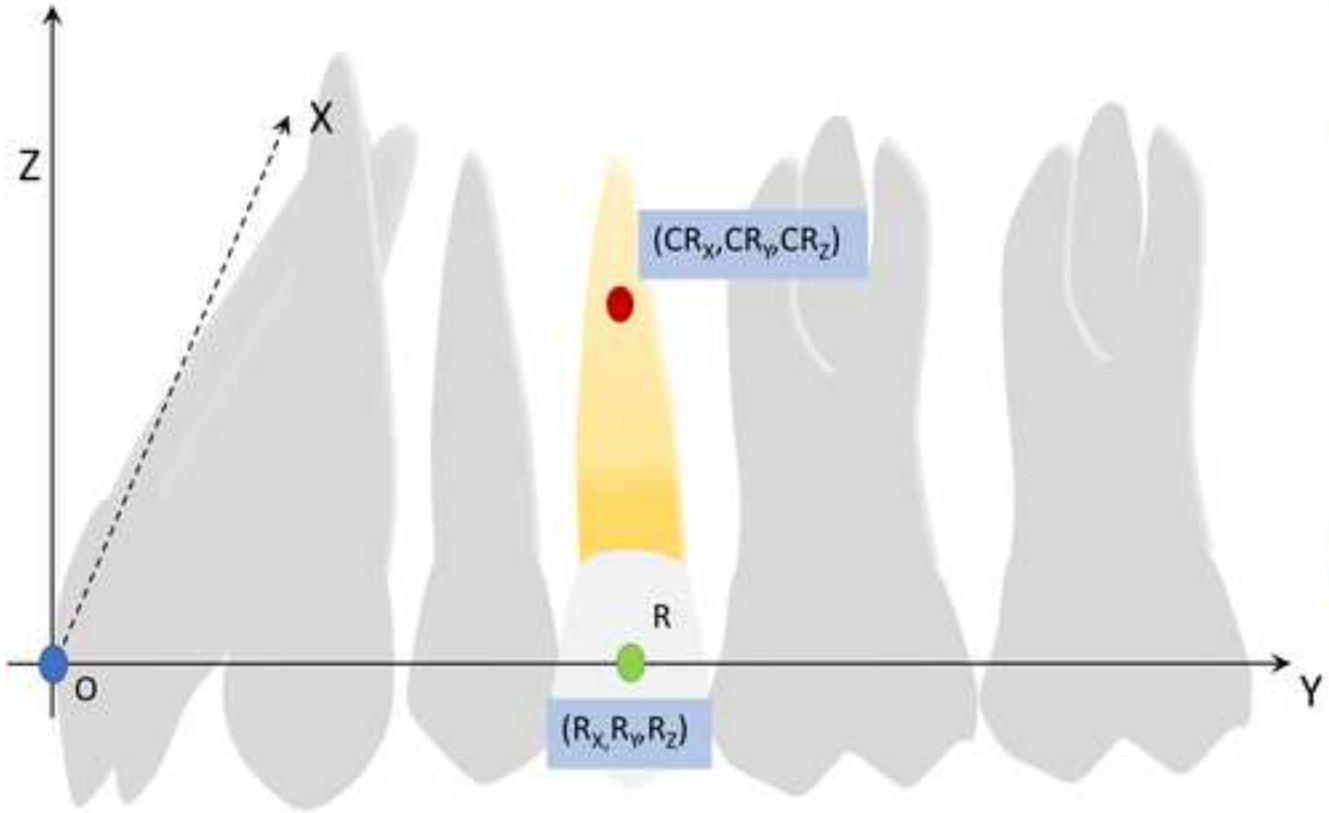


Figure 6



Location of C_{RES} Relative to 'O' :
(CR_X, CR_Y, CR_Z)

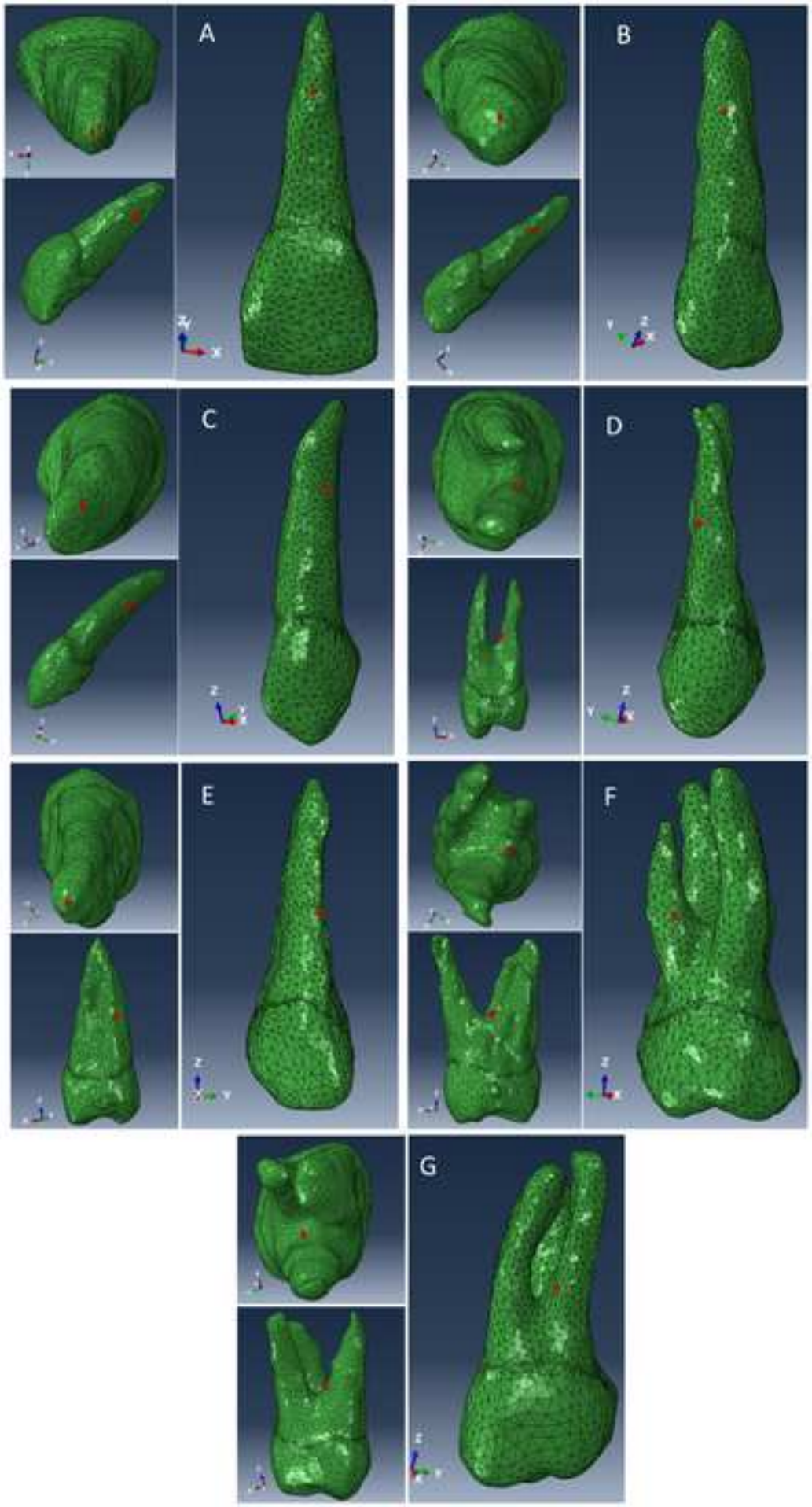
Location of 'R' Relative to 'O' :
(R_X, R_Y, R_Z) where,

R_X = Distance between 'O' point and 'R' in the transverse plane.
R_Y = Distance between 'O' point and 'R' in the anterio-posterior plane.
R_Z = Distance between 'O' point and 'R' in the anterio-posterior plane.

C_{RES} Relative to 'R' in the x-y-z plane:
$(CR_X\text{-}R_X)$, $(CR_Y\text{-}R_Y)$ & $(CR_Z\text{-}R_Z)$

Figure 7

Click here to access/download;Figure;Figure 7.tiff



Hollow Type:	**Both (outside & Inside)**
Distance	0.2
Smallest Detail:	0.05
Reduce:	Checked
Cleanup at border:	Checked
Cleanup factor:	1.1

Table 1: Hollow Tool Parameters

Structure	Elastic modulus (MPa)	Poisson's ratio	Specific density (g/cm^3)
Teeth	13000	0.3	2.02
Bone	17000	0.3	1.85
PDL	0.05	See text	1

Table 2 : Material Properties of the finite element model

Tooth Number	Tooth length	Root Length	**x**	**y**	**z**
UL1	25.2	15.1	3.4	11.0	12.9
UL2	26.0	16.8	8.8	13.2	14.3
UL3	29.1	19.5	15.1	18.0	15.6
UL4	23.8	15.7	18.4	21.5	10.6
UL5	24.8	18.2	20.9	28.2	10.1
UL6	22.0	16.4	25.8	38.7	11.6
UL7	21.4	15.0	27.4	43.2	11.4
UR1	24.9	14.6	-4.6	10.8	13.2
UR2	26.3	16.7	-9.9	13.0	13.6
UR3	30.9	21.1	-15.6	17.7	14.2
UR4	22.9	16.7	-19.0	21.9	9.2
UR5	23.4	16.7	-21.1	29.4	8.8
UR6	22.2	16.3	-23.9	39.6	9.8
UR7	20.8	15.9	-21.7	47.0	10.4

Table 3. Three-dimensional (x-y-z) location of the Center of Resistance of Maxillary teeth (in mm) in relation to the global point O.

Tooth Number	Tooth length	Root Length	x	y	z
UL1	25.2	15.1	-1.1	10.9	9.4
UL2	26.0	16.8	-5.5	9.4	10.4
UL3	29.1	19.5	-5.7	9.3	13.2
UL4	23.8	15.7	-6.4	5.7	9.0
UL5	24.8	18.2	-6.7	7.0	9.5
UL6	22.0	16.4	-6.9	8.3	10.4
UL7	21.4	15.0	-8.6	3.3	7.3
UR1	24.9	14.6	0.5	10.8	11.1
UR2	26.3	16.7	5.0	10.3	9.3
UR3	30.9	21.1	5.7	8.5	12.0
UR4	22.9	16.7	5.3	5.3	9.3
UR5	23.4	16.7	5.3	6.5	9.1
UR6	22.2	16.3	5.6	7.8	10.1
UR7	20.8	15.9	9.5	4.3	8.6

Table 4. Three dimensional (x-y-z) location of the Center of Resistance of Maxillary Teeth (in mm) in relation to a local point R for each tooth whose center of resistance is being evaluated.
Here, R is the geometric center of the buccal surface of the crown.

Tooth Number	Fy	FZ	Difference
UL1	-1.36	-0.80	0.56
UL2	-5.73	-5.23	0.5
UL3	-6.00	-5.45	0.55
UL4	-6.11	-6.65	0.54
UL5	-5.95	-7.40	1.46
UL6	-6.18	-7.67	1.49
UR1	0.36	0.67	0.31
UR2	5.23	4.77	0.46
UR3	5.93	5.38	0.55
UR4	4.57	6.01	1.44
UR5	5.88	4.69	1.91
UR6	5.19	5.98	0.79

Table 5: Variation in the Center of resistance position along the x axis (in mm) when force is applied along the y (Fy) and z (Fz

:) axis.

Name of Material/ Equipment	Company	Catalog Number	Comments/Description
3-matic software	Materialise, Leuven, Belgium.		Cleaning and meshing
Abaqus/CAE software version 2017	Dassault Systèmes Simulia Corp., Johnston, RI, USA.		Finite Element Analysis
Mimics software, version 17.0	Materialise, Leuven, Belgium.		Segmentation of teeth and bone

Revision notes (60746_R1_112519)

Editorial comments: In black
Author comments: In blue

1. The usage of '→' (right arrow) seems to vary in your protocol-sometimes it seems to refer to nested menus, sometimes actions done in succession. Please only use this symbol for menu items and describe the latter in words. Note that steps/substeps should always be written out sentences, largely in the imperative.

We have deleted a lot of the '→' (arrows) form the manuscript and converted them into full sentences. The '→' is now only being used for menu items.

2. The highlighted portion of the protocol is too long (~5 pages; see attached, formatted per JoVE guidelines); please reduce to 2.75 pages. This could potentially include combining substeps, as many are currently very short.

We have included in the revision a separate document titled 'video manuscript.' It has only the video manuscript. We have made sure that it is < 2.75 pages.

3. Step 1.1: Could you possibly include a reference here for this procedure?

We have added a reference for step 1.1:
16. Pauwels, R., Araki, K., Siewerdsen, J.H., Thongvigitmanee, S.S. Technical aspects of dental CBCT: state of the art. Dentomaxillofacial Radiology. ***44****(1), 20140224 (2015).*

4. Please combine Figures 5A and 5B into one file, with panel labels.

We have combined the images as one 'tiff' image.

Supplemental Contents: Python Scripts

Table of Contents

Please note: The highlighted text on page 1 requires custom modification according to the patient number and where the data is located in the directory of the computer.

1. Model_setup_Part1.py

```
#
#
# Part 1 imports the model from a .inp Abaqus file and sets up functions


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  *


from abaqus import getInput


# Import message box modules
import ctypes
MessageBox  =  ctypes.windll.user32.MessageBoxA


# Set up random list
import  random
random_list = ['right',  'left']


# Define directories
fields = (('Model directory:','C:/Users/gandhi/Desktop/Recent Model/Recent model/Pt6/'),
   ('Plugin directory', 'C:/SIMULIA/CAE/2017/win_b64/code/python2.7/lib/abaqus_plugins/findNearestNode'),
   ('Script          directory',          'C:/Users/gandhi/Desktop/Recent          Model/Recent          Scripts/'))
directory, plugin_dir, script_dir = getInputs(fields=fields, label='Specify file paths', dialogTitle='Define directories', )
```

```
#                Import                parts
model_name = getInput('Model file:', 'Pt2.inp')
model_file='{}{}'.format(directory,model_name)

mdb.models.changeKey(fromName='Model-1', toName='Simulation')
mdb.models['Simulation'].PartFromInputFile(
   inputFileName=model_file)

#    Set    model    variables
sim_model = mdb.models['Simulation']
max_part  = sim_model.parts['MAXILLA']

# Create element sets
max_elems =  max_part.elements[:]
max_elems_set = max_part.Set(elements=max_elems, name='MAXILLA_elem_set')

for        i        in        range(1,8):
   p = sim_model.parts['UL{}'.format(i)]
   elems = p.elements[:]
   p.Set(elements=elems,      name='UL{}_elem_set'.format(i))
   p        =        sim_model.parts['UL{}_PDL'.format(i)]
   elems            =            p.elements[:]
   p.Set(elements=elems,name='UL{}_PDL_elem_set'.format(i))
   p  = sim_model.parts['UR{}'.format(i)]
   elems            =            p.elements[:]
   p.Set(elements=elems,      name='UR{}_elem_set'.format(i))
   p        =        sim_model.parts['UR{}_PDL'.format(i)]
   elems            =            p.elements[:]
   p.Set(elements=elems,name='UR{}_PDL_elem_set'.format(i))

# User manually creates surfaces of interest (sockets, inner/outer PDL, teeth)
MessageBox(None,'Pleasecreatesocket,PDL,andtoothsurfaces.\nRunModel_setup_Part2whenfinished.','Model_setup_Part1Completed',
0)
```

2. Model_setup_Part2.py

```
#
#
# Part 2 assigns material definitions and creates all instances needed for  simulation

# Run function List

def fx_list():
   file_name  =  'Functions.py'
   file_path = script_dir + file_name
   execfile(file_path,  main  .__dict  )

fx_list()

# Create materials
sim_model.Material(name='Bone')
sim_model.materials['Bone'].Elastic(table=((17000.0, 0.3), ))
sim_model.materials['Bone'].Density(table=((1.85e-09, ), ))
sim_model.Material(name='Tooth')
sim_model.materials['Tooth'].Elastic(table=((17000.0, 0.3), ))
sim_model.materials['Tooth'].Density(table=((2.02e-09, ), ))
sim_model.Material(name='PDL')
sim_model.materials['PDL'].Hyperelastic(
      materialType=ISOTROPIC, testData=OFF, type=OGDEN,
      volumetricResponse=VOLUMETRIC_DATA, table=((0.07277, 16.95703, 3e-07), ))
sim_model.materials['PDL'].Density(table=((1.0e-09,  ),  ))

# Define sections
sim_model.HomogeneousSolidSection(name='Max_section',
   material='Bone', thickness=None)
sim_model.HomogeneousSolidSection(name='Tooth_section',
   material='Tooth', thickness=None)
sim_model.HomogeneousSolidSection(name='PDL_section',
   material='PDL', thickness=None)

#              Assign              sections
region  =  max_part.sets['MAXILLA_elem_set']
max_part.SectionAssignment(region=region, sectionName='Max_section', offset=0.0,
   offsetType=MIDDLE_SURFACE, offsetField='',
   thicknessAssignment=FROM_SECTION)

for          i          in          range(1,8):
   p = sim_model.parts['UL{}'.format(i)]
   region  =  p.sets['UL{}_elem_set'.format(i)]
   p.SectionAssignment(region=region, sectionName='Tooth_section'.format(i), offset=0.0,
   offsetField='',)
   p                    =                    sim_model.parts['UL{}_PDL'.format(i)]
   region  =  p.sets['UL{}_PDL_elem_set'.format(i)]
   p.SectionAssignment(region=region, sectionName='PDL_section'.format(i), offset=0.0,
   offsetField='',)
   p = sim_model.parts['UR{}'.format(i)]
   region = p.sets['UR{}_elem_set'.format(i)]
   p.SectionAssignment(region=region, sectionName='Tooth_section'.format(i), offset=0.0,
   offsetField='',)
   p                    =                    sim_model.parts['UR{}_PDL'.format(i)]
   region  =  p.sets['UR{}_PDL_elem_set'.format(i)]
   p.SectionAssignment(region=region, sectionName='PDL_section'.format(i), offset=0.0,
   offsetField='',)

# Assign element types
import mesh
```

```
hybrid_tet = mesh.ElemType(elemCode=C3D4H, elemLibrary=STANDARD,
   secondOrderAccuracy=OFF,  distortionControl=DEFAULT)

for                i                in                range(1,8):
   p = sim_model.parts['UL{}_PDL'.format(i)]
   region  =  p.sets['UL{}_PDL_elem_set'.format(i)]
   p.setElementType(regions=region, elemTypes=(hybrid_tet, ))
   p          =          sim_model.parts['UR{}_PDL'.format(i)]
   region = p.sets['UR{}_PDL_elem_set'.format(i)]
   p.setElementType(regions=region, elemTypes=(hybrid_tet, ))

# Set root assemblies variables
sim_root = sim_model.rootAssembly

#            Create             instances
sim_root.Instance(name='MAXILLA', part=max_part, dependent=ON)

for           i           in           range(1,8):
   p  = sim_model.parts['UL{}'.format(i)]
   sim_root.Instance(name='UL{}'.format(i),    part=p,    dependent=ON)
   p = sim_model.parts['UL{}_PDL'.format(i)]
   sim_root.Instance(name='UL{}_PDL'.format(i), part=p, dependent=ON)
   p = sim_model.parts['UR{}'.format(i)]
   sim_root.Instance(name='UR{}'.format(i),    part=p,    dependent=ON)
   p = sim_model.parts['UR{}_PDL'.format(i)]
   sim_root.Instance(name='UR{}_PDL'.format(i), part=p,dependent=ON)

# Create merged instances
instance_list = []

for i in range(1, 8):
   instance_list.append(sim_root.instances['UL{}'.format(i)])
   instance_list.append(sim_root.instances['UR{}'.format(i)])
   if i > 1:
      #        Merge        instances
      sim_root.InstanceFromBooleanMerge(name='U{}_{}'.format(i, i), instances=instance_list,
          originalInstances=SUPPRESS, mergeNodes=BOUNDARY_ONLY,
          nodeMergingTolerance=1e-06,   domain=BOTH)
      # Rename merged instance
      sim_root.features.changeKey(fromName='U{}_{}-1'.format(i,     i),
         toName='U{}_{}'.format(i, i))
      # Resume suppressed teeth
      for n in range(1, i + 1):
         sim_root.features['UL{}'.format(n)].resume()
         sim_root.features['UR{}'.format(n)].resume()

# Define constraints
for i in range(1,8):
   region1    =    sim_root.instances['MAXILLA'].surfaces['UL{}_socket'.format(i)]
   region2 = sim_root.instances['UL{}_PDL'.format(i)].surfaces['UL{}_PDL_outer'.format(i)]
   sim_model.Tie(name='UL{}_socket_PDL'.format(i),     master=region1,
      slave=region2, positionToleranceMethod=COMPUTED, adjust=OFF,
      tieRotations=ON, thickness=ON)
   region1    =    sim_root.instances['MAXILLA'].surfaces['UR{}_socket'.format(i)]
   region2 = sim_root.instances['UR{}_PDL'.format(i)].surfaces['UR{}_PDL_outer'.format(i)]
   sim_model.Tie(name='UR{}_socket_PDL'.format(i),     master=region1,
      slave=region2, positionToleranceMethod=COMPUTED, adjust=OFF,
      tieRotations=ON, thickness=ON)
   # Create single tooth constraints
   if i == 2 or 4 or 5 or 7:
```

```
    # Randomly pick between right/left
    side = random.choice(random_list)
    if side == 'left':
      region2 = sim_root.instances['UL{}_PDL'.format(i)].surfaces['UL{}_PDL_inner'.format(i)]
      region1 = sim_root.instances['UL{}'.format(i)].surfaces['UL{}'.format(i)]
      sim_model.Tie(name='UL{}_PDL'.format(i), master=region1,
        slave=region2, positionToleranceMethod=COMPUTED, adjust=OFF,
        tieRotations=ON, thickness=ON)
      if i == 2:
        U2 = 'left'
      elif i == 4:
        U4 = 'left'
      elif i == 5:
        U5 = 'left'
      else:
        U7 = 'left'
    else:
      region2 = sim_root.instances['UR{}_PDL'.format(i)].surfaces['UR{}_PDL_inner'.format(i)]
      region1 = sim_root.instances['UR{}'.format(i)].surfaces['UR{}'.format(i)]
      sim_model.Tie(name='UR{}_PDL'.format(i), master=region1,
        slave=region2, positionToleranceMethod=COMPUTED, adjust=OFF,
        tieRotations=ON, thickness=ON)
      if i == 2:
        U2 = 'right'
      elif i == 4:
        U4 = 'right'
      elif i == 5:
        U5 = 'right'
      else:
        U7 = 'right'
  # Create multi-tooth constraints
  if i > 1:
    for n in range(1, i + 1):
      region2 = sim_root.instances['UL{}_PDL'.format(n)].surfaces['UL{}_PDL_inner'.format(n)]
      region1 = sim_root.instances['U{}_{}'.format(i, i)].surfaces['UL{}'.format(n)]
      sim_model.Tie(name='UL{}_PDL_{}'.format(n, i), master=region1,
        slave=region2, positionToleranceMethod=COMPUTED, adjust=OFF,
        tieRotations=ON, thickness=ON)
      region2 = sim_root.instances['UR{}_PDL'.format(n)].surfaces['UR{}_PDL_inner'.format(n)]
      region1 = sim_root.instances['U{}_{}'.format(i, i)].surfaces['UR{}'.format(n)]
      sim_model.Tie(name='UR{}_PDL_{}'.format(n, i), master=region1,
        slave=region2, positionToleranceMethod=COMPUTED, adjust=OFF,
        tieRotations=ON, thickness=ON)

# Create posterior instances and constraints
# Randomly pick between right/left
# U4_7
instance_list = []
side = random.choice(random_list)
if side == 'left':
  # Create merged part
  # Generate instance list
  for i in range(4, 8):
    sim_root.features['UL{}'.format(i)].resume()
    instance_list.append(sim_root.instances['UL{}'.format(i)])
  # Merge instances
  sim_root.InstanceFromBooleanMerge(name='U4_7', instances=instance_list,
    originalInstances=SUPPRESS, mergeNodes=BOUNDARY_ONLY,
    nodeMergingTolerance=1e-06, domain=BOTH)
  # Rename merged instance
  sim_root.features.changeKey(fromName='U4_7-1',
```

```
        toName='U4_7')
    # Resume suppressed teeth
    for i in range(4, 8):
        sim_root.features['UL{}'.format(i)].resume()
    # Create constraints
    for i in range(4, 8):
        region2 = sim_root.instances['UL{}_PDL'.format(i)].surfaces['UL{}_PDL_inner'.format(i)]
        region1 = sim_root.instances['U4_7'].surfaces['UL{}'.format(i)]
        sim_model.Tie(name='UL{}_PDL_4_7'.format(i),    master=region1,
            slave=region2, positionToleranceMethod=COMPUTED, adjust=OFF,
            tieRotations=ON, thickness=ON)
    U4_7 = 'left'
else:
    # Create merged part
    # Generate instance list
    for i in range(4, 8):
        sim_root.features['UR{}'.format(i)].resume()
        instance_list.append(sim_root.instances['UR{}'.format(i)])
    #       Merge       instances
    sim_root.InstanceFromBooleanMerge(name='U4_7',   instances=instance_list,
         originalInstances=SUPPRESS, mergeNodes=BOUNDARY_ONLY,
         nodeMergingTolerance=1e-06,   domain=BOTH)
    # Rename merged instance
    sim_root.features.changeKey(fromName='U4_7-1',
        toName='U4_7')
    # Resume suppressed teeth
    for i in range(4, 8):
        sim_root.features['UR{}'.format(i)].resume()
    # Create constraints
    for i in range(4, 8):
        region2 = sim_root.instances['UR{}_PDL'.format(i)].surfaces['UR{}_PDL_inner'.format(i)]
        region1 = sim_root.instances['U4_7'].surfaces['UR{}'.format(i)]
        sim_model.Tie(name='UR{}_PDL_4_7'.format(i),    master=region1,
            slave=region2, positionToleranceMethod=COMPUTED, adjust=OFF,
            tieRotations=ON, thickness=ON)
    U4_7 = 'right'

#U5_7
instance_list           =           []
side = random.choice(random_list)
if side == 'left':
    # Create merged part
    # Generate instance list
    for i in range(5, 8):
        sim_root.features['UL{}'.format(i)].resume()
        instance_list.append(sim_root.instances['UL{}'.format(i)])
    #       Merge       instances
    sim_root.InstanceFromBooleanMerge(name='U5_7',   instances=instance_list,
         originalInstances=SUPPRESS, mergeNodes=BOUNDARY_ONLY,
         nodeMergingTolerance=1e-06,   domain=BOTH)
    # Rename merged instance
    sim_root.features.changeKey(fromName='U5_7-1',
        toName='U5_7')
    # Resume suppressed teeth
    for i in range(5, 8):
        sim_root.features['UL{}'.format(i)].resume()
    # Create constraints
    for i in range(5, 8):
        region2 = sim_root.instances['UL{}_PDL'.format(i)].surfaces['UL{}_PDL_inner'.format(i)]
        region1 = sim_root.instances['U5_7'].surfaces['UL{}'.format(i)]
        sim_model.Tie(name='UL{}_PDL_5_7'.format(i),    master=region1,
```

```
        slave=region2, positionToleranceMethod=COMPUTED, adjust=OFF,
        tieRotations=ON, thickness=ON)
    U5_7 = 'left'
else:
    # Create merged part
    # Generate instance list
    for i in range(5, 8):
        sim_root.features['UR{}'.format(i)].resume()
        instance_list.append(sim_root.instances['UR{}'.format(i)])
    #       Merge       instances
    sim_root.InstanceFromBooleanMerge(name='U5_7',   instances=instance_list,
        originalInstances=SUPPRESS, mergeNodes=BOUNDARY_ONLY,
        nodeMergingTolerance=1e-06,   domain=BOTH)
    # Rename merged instance
    sim_root.features.changeKey(fromName='U5_7-1',
        toName='U5_7')
    # Resume suppressed teeth
    for i in range(5, 8):
        sim_root.features['UR{}'.format(i)].resume()
    # Create constraints
    for i in range(5, 8):
        region2 = sim_root.instances['UR{}_PDL'.format(i)].surfaces['UR{}_PDL_inner'.format(i)]
        region1 = sim_root.instances['U5_7'].surfaces['UR{}'.format(i)]
        sim_model.Tie(name='UR{}_PDL_5_7'.format(i),    master=region1,
            slave=region2, positionToleranceMethod=COMPUTED, adjust=OFF,
            tieRotations=ON, thickness=ON)
    U5_7 = 'right'

#       User       manually       sets       BC,       creates       sets       for       force       points
MessageBox(None, 'Please set BC, create sets for force points, and orient model.\nRun Model_setup_Part3 when finished.',
'Model_setup_Part2 Completed', 0)
```

3. Model_setup_Part3.py

```
#
#
# Part 3 creates steps and jobs for each simulation

#       U1_y
if U1 == 'left':
   tooth = 'UL1'
else:
   tooth = 'UR1'

name = 'U1_y_force'
direction = 'y'

resume_tooth(tooth)
first_step(name, tooth, direction)
new_job(name)

#          UL1_z
name = 'U1_z_force'
direction = 'z'

next_step(name, tooth, direction)
new_job(name)

#       U3_y
if U3 == 'left':
   tooth = 'UL3'
else:
   tooth = 'UR3'

name = 'U3_y_force'
direction = 'y'

resume_tooth(tooth)
next_step(name, tooth, direction)
new_job(name)

#            U3_z
name = 'U3_z_force'
direction = 'z'

next_step(name, tooth, direction)
new_job(name)

#       U6_y
if U6 == 'left':
   tooth = 'UL6'
else:
   tooth = 'UR6'

name    =    'U6_y_force'
direction = 'y'

next_step(name, tooth, direction)
new_job(name)

#            U6_z
name = 'U6_z_force'
direction = 'z'
```

```
next_step(name, tooth, direction)
new_job(name)

# Multi tooth groups
for i in range(2,8):
   teeth = 'U{}_{}'.format(i, i)
   name = 'U{}_{}_y_force'.format(i, i)
   direction    =    'y'
   next_step(name, teeth, direction)
   new_job(name)
   name = 'U{}_{}_z_force'.format(i, i)
   direction    =    'z'
   next_step(name, teeth, direction)
   new_job(name)

# Posterior tooth groups
# U4_7
teeth          =          'U4_7'
name  =  'U4_7_y_force'
direction = 'y'

next_step(name, teeth, direction)
new_job(name)

name  =  'U4_7_z_force'
direction = 'z'

next_step(name, teeth, direction)
new_job(name)

# U5_7
teeth          =          'U5_7'
name  =  'U5_7_y_force'
direction = 'y'

next_step(name, teeth, direction)
new_job(name)

name  =  'U5_7_z_force'
direction = 'z'

next_step(name, teeth, direction)
new_job(name)

#          User          selects          which          jobs          to          run
MessageBox(None, 'Please proceed to Job_submission to select jobs to run.', 'Model setup completed',   0)
```

4. Functions.py

```
#
#
# List of all functions used in setting up and running analyses


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 *


from abaqus import getInput
import numpy as np
import math


# Import message box modules
import ctypes
MessageBox = ctypes.windll.user32.MessageBoxA


# Set up random list
import random
random_list = ['right', 'left']


# Define model variables
sim_model = mdb.models['Simulation']
sim_root = sim_model.rootAssembly
```

```
# Define directories
fields=(('Model directory:','C:/Users/gandhi/Desktop/Recent Model/Recent model/Pt2/'),
   ('Plugin directory', 'C:/SIMULIA/CAE/2017/win_b64/code/python2.7/lib/abaqus_plugins/findNearestNode'),
   ('Script directory', 'C:/Users/gandhi/Desktop/Recent Model/Recent Scripts/'))
directory, plugin_dir, script_dir = getInputs(fields=fields, label='Specify file paths', dialogTitle='Define directories', )

# Define functions
def PP3D(name):
   # Define post-processing script directory
   file_name = '3D_processing_fx.py'
   file_path = script_dir + file_name
   execfile(file_path, __main__.__dict__)

def suppress_tooth(tooth_number):
   # Suppress tooth_number materials
   sim_root.features[tooth_number].suppress()
   sim_root.features[tooth_number + '_PDL'].suppress()
   sim_model.constraints[tooth_number + '_socket_PDL'].suppress()
   sim_model.constraints[tooth_number + '_PDL'].suppress()

def suppress_merge(number):
   # Suppress merged part materials
   sim_root.features['U{}_{}'.format(number, number)].suppress()
   for i in range(1, int(number) + 1):
      sim_root.features['UL{}_PDL'.format(i)].suppress()
      sim_root.features['UR{}_PDL'.format(i)].suppress()
      sim_model.constraints['UL{}_PDL_{}'.format(i, number)].suppress()
      sim_model.constraints['UR{}_PDL_{}'.format(i, number)].suppress()
      sim_model.constraints['UL{}_socket_PDL'.format(i)].suppress()
      sim_model.constraints['UR{}_socket_PDL'.format(i)].suppress()

def resume_tooth(tooth_number):
   # Resume tooth_number materials
   sim_root.features[tooth_number].resume()
   sim_root.features[tooth_number + '_PDL'].resume()
```

```
    sim_model.constraints[tooth_number + '_socket_PDL'].resume()
    sim_model.constraints[tooth_number + '_PDL'].resume()

def  resume_merge(number):
    # Suppress merged part materials
    sim_root.features['U{}_{}'.format(number, number)].resume()
    for i in range(1, int(number) + 1):
        sim_root.features['UL{}_PDL'.format(i)].resume()
        sim_root.features['UR{}_PDL'.format(i)].resume()
        sim_model.constraints['UL{}_PDL_{}'.format(i, number)].resume()
        sim_model.constraints['UR{}_PDL_{}'.format(i, number)].resume()
        sim_model.constraints['UL{}_socket_PDL'.format(i)].resume()
        sim_model.constraints['UR{}_socket_PDL'.format(i)].resume()

def resume_step(name):
    sim_model.steps[name].resume()

def suppress_step(name):
    sim_model.steps[name].suppress()

def first_step(name, tooth, direction):
    #     Create     step
    sim_model.StaticStep(name=name,  previous='Initial',
        timePeriod=0.1, maxNumInc=10000, initialInc=0.001, minInc=1e-06,
        amplitude=RAMP, nlgeom=ON)
    if direction == 'y':
        # Field Output Request
        sim_model.fieldOutputRequests.changeKey(fromName='F-Output-1',
            toName='F-Output-{}_y'.format(tooth))
        sim_model.fieldOutputRequests['F-Output-{}_y'.format(tooth)].setValues(
            variables=('S','LE','U','RF','CF','COORD'))
        # Create load (y direction)
        region_name        =        name
        region =  sim_root.sets[region_name]
        sim_model.ConcentratedForce(name=region_name,
```

```
        createStepName=region_name, region=region, cf2=1.0,
        distributionType=UNIFORM,  field='', localCsys=None)
    else:
        # Field Output Request
        sim_model.fieldOutputRequests.changeKey(fromName='F-Output-1',
            toName='F-Output-{}_z'.format(tooth))
        sim_model.fieldOutputRequests['F-Output-{}_z'.format(tooth)].setValues(
            variables=('S','LE','U','RF','CF','COORD'))
        # Create load (z direction)
        region_name        =        name
        region =  sim_root.sets[region_name]
        sim_model.ConcentratedForce(name=region_name,
            createStepName=region_name, region=region, cf3=1.0,
            distributionType=UNIFORM,  field='', localCsys=None)


def next_step(name, tooth, direction):
    #     Create     step
    sim_model.StaticStep(name=name,  previous='Initial',
        timePeriod=0.1, maxNumInc=10000, initialInc=0.001, minInc=1e-06,
        amplitude=RAMP, nlgeom=ON)
    if direction == 'y':
        # Field Output Request
        sim_model.FieldOutputRequest(name='F-Output-{}_y'.format(tooth),
            createStepName=name,   variables=('S','LE','U','RF','CF','COORD'))
        # Create load (y direction)
        region_name        =        name
        region =  sim_root.sets[region_name]
        sim_model.ConcentratedForce(name=region_name,
            createStepName=region_name, region=region, cf2=1.0,
            distributionType=UNIFORM,  field='', localCsys=None)
    else:
        # Field Output Request
        sim_model.FieldOutputRequest(name='F-Output-{}_z'.format(tooth),
            createStepName=name,   variables=('S','LE','U','RF','CF','COORD'))
        # Create load (z direction)
```

```
    region_name = name
    region = sim_root.sets[region_name]
    sim_model.ConcentratedForce(name=region_name,
      createStepName=region_name, region=region, cf3=1.0,
      distributionType=UNIFORM, field='', localCsys=None)

def new_job(name):
  # Create job
  mdb.Job(name=name, model='Simulation', description='', type=ANALYSIS,
    atTime=None, waitMinutes=0, waitHours=0, queue=None, memory=90,
    memoryUnits=PERCENTAGE, getMemoryFromAnalysis=True,
    explicitPrecision=SINGLE, nodalOutputPrecision=SINGLE, echoPrint=OFF,
    modelPrint=OFF, contactPrint=OFF, historyPrint=OFF, userSubroutine='',
    scratch='', resultsFormat=ODB, multiprocessingMode=DEFAULT, numCpus=2,
    numDomains=2, numGPUs=1)

def submit_job(name):
  mdb.jobs[name].submit(consistencyChecking=OFF)

def write_input(name):
  mdb.jobs[name].writeInput(consistencyChecking=OFF)

def run_job(name):
  mdb.jobs[name].submit(consistencyChecking=OFF)
  mdb.jobs[name].waitForCompletion()

def iterate(name, dir):
  # Delete .lck file/.odb file
  import os
  file_name = '{}.lck'.format(name)
  file_path = dir + file_name
  os.remove(file_path)
  file_name = '{}.odb'.format(name)
  file_path = dir + file_name
  os.remove(file_path)
```

```
    # Re-run job
    run_job(name)
    # Re-run post-processing
    if direction == 'y':
        PPy(name)
    else:
        PPz(name)

def nearest_node(dir, instance, xcoord, ycoord, zcoord, set_name):
    import sys
    sys.path.insert(0, dir)
    import  nearestNodeModule
    session.viewports['Viewport: 1'].assemblyDisplay.setValues(mesh=ON)
    session.viewports['Viewport: 1'].assemblyDisplay.meshOptions.setValues(
    meshTechnique=ON)
    nearestNodeModule.hideTextAndArrow()
    n1 = sim_root.instances[instance].nodes
    pickedSelectedNodes = n1[:]
    n = nearestNodeModule.findNearestNode(xcoord = xcoord, ycoord = ycoord, zcoord = zcoord, name='',
    selectedNodes=pickedSelectedNodes,   instanceName=instance)
    label   =   n[0]
    coordinates   =   n[3]
    nodes1 = n1[label - 1:label]
    force_location =  sim_root.sets[name].nodes[0].coordinates
    if coordinates != force_location:
        sim_root.Set(nodes=nodes1, name=set_name)
    else:
        iterate_check = False

def create_set_from_node(node_number, instance, set_name):
    n1          =          sim_root.instances[instance].nodes
    CR_node = n1[node_number - 1:node_number]
    sim_root.Set(nodes=CR_node,  name=set_name)

def bool_set(set1_name, set2_name, set_name):
```

```
    set1 = sim_root.sets[set1_name]
    set2 = sim_root.sets[set2_name]
    sim_root.SetByBoolean(set_name, [set1, set2], DIFFERENCE)

def bool_add_set(set1_name, set2_name, set_name):
    set1 = sim_root.sets[set1_name]
    set2 = sim_root.sets[set2_name]
    sim_root.SetByBoolean(set_name, [set1, set2], UNION)

def hide_instances(number):
    instance_list = ['MAXILLA']
    for i in range(1, int(number) + 1):
        instance_list.append('UL{}_PDL'.format(i))
        instance_list.append('UR{}_PDL'.format(i))
    session.viewports['Viewport: 1'].assemblyDisplay.hideInstances(instances=instance_list)

def suppress_all():
    file_name = 'Suppress_all.py'
    file_path = script_dir + file_name
    execfile(file_path, __main__.__dict__)
```

5. Suppress_all.py

```
#
#
# This script suppresses all instances, constraints, and steps

# Get user input
fields=(('U1:','left'),('U2:','left'),('U3:','left'),('U4:','left'),('U5:','left'),('U6:','left'),('U7:','left'),('U4_7:','left'),('U5_7:','left'))
U1, U2, U3, U4, U5, U6, U7, U4_7, U5_7 = getInputs(fields=fields, label='Specify sides for unilateral groups', dialogTitle='Suppress all', )

# Suppress all
for i in range(1,8):
   sim_root.features['UL{}'.format(i)].suppress()
   sim_root.features['UL{}_PDL'.format(i)].suppress()
   sim_root.features['UR{}'.format(i)].suppress()
   sim_root.features['UR{}_PDL'.format(i)].suppress()
   sim_model.constraints['UL{}_socket_PDL'.format(i)].suppress()
   sim_model.constraints['UR{}_socket_PDL'.format(i)].suppress()

# Suppress merged parts
for i in range(2, 8):
   sim_root.features['U{}_{}'.format(i, i)].suppress()
   for n in range(1, i + 1):
      sim_model.constraints['UL{}_PDL_{}'.format(n, i)].suppress()
      sim_model.constraints['UR{}_PDL_{}'.format(n,  i)].suppress()

# Suppress unilateral parts
if U1 == 'left':
   sim_model.constraints['UL1_PDL'.format(i)].suppress()
else:
   sim_model.constraints['UR1_PDL'.format(i)].suppress()
if U2 == 'left':
   sim_model.constraints['UL2_PDL'.format(i)].suppress()
else:
   sim_model.constraints['UR2_PDL'.format(i)].suppress()
if U3 == 'left':
```

```
    sim_model.constraints['UL3_PDL'.format(i)].suppress()
else:
    sim_model.constraints['UR3_PDL'.format(i)].suppress()
if U4 == 'left':
    sim_model.constraints['UL4_PDL'.format(i)].suppress()
else:
    sim_model.constraints['UR4_PDL'.format(i)].suppress()
if U5 == 'left':
    sim_model.constraints['UL5_PDL'.format(i)].suppress()
else:
    sim_model.constraints['UR5_PDL'.format(i)].suppress()
if U6 == 'left':
    sim_model.constraints['UL6_PDL'.format(i)].suppress()
else:
    sim_model.constraints['UR6_PDL'.format(i)].suppress()
if U7 == 'left':
    sim_model.constraints['UL7_PDL'.format(i)].suppress()
else:
    sim_model.constraints['UR7_PDL'.format(i)].suppress()

# Posterior groups
sim_root.features['U4_7'].suppress()
for i in range(4, 8):
    if U4_7 == 'left':
        sim_model.constraints['UL{}_PDL_4_7'.format(i)].suppress()
    else:
        sim_model.constraints['UR{}_PDL_4_7'.format(i)].suppress()

sim_root.features['U5_7'].suppress()
for i in range(5, 8):
    if U5_7 == 'left':
        sim_model.constraints['UL{}_PDL_5_7'.format(i)].suppress()
    else:
        sim_model.constraints['UR{}_PDL_5_7'.format(i)].suppress()
```

```
# Suppress steps
name = 'U1_y_force'
suppress_step(name)
name = 'U1_z_force'
suppress_step(name)
name = 'U2_y_force'
suppress_step(name)
name = 'U2_z_force'
suppress_step(name)
name = 'U3_y_force'
suppress_step(name)
name = 'U3_z_force'
suppress_step(name)
name = 'U4_y_force'
suppress_step(name)
name = 'U4_z_force'
suppress_step(name)
name = 'U5_y_force'
suppress_step(name)
name = 'U5_z_force'
suppress_step(name)
name = 'U6_y_force'
suppress_step(name)
name = 'U6_z_force'
suppress_step(name)
name = 'U7_y_force'
suppress_step(name)
name = 'U7_z_force'
suppress_step(name)

# Suppress multi tooth groups
for i in range(2, 8):
    name = 'U{}_{}_y_force'.format(i, i)
    suppress_step(name)
    name = 'U{}_{}_z_force'.format(i, i)
```

```
    suppress_step(name)


# Suppress posterior tooth groups
name = 'U4_7_y_force'
suppress_step(name)
name = 'U4_7_z_force'
suppress_step(name)
name = 'U5_7_y_force'
suppress_step(name)
name = 'U5_7_z_force'
suppress_step(name)
```

6. Job_submission.py

```
#
#
# Submits multiple jobs to run in parallel


# Suppress all to start
suppress_all()


#                User                selects                which                jobs                to                run
fields=(('U1:','Y'),('U2:','Y'),('U3:','Y'),('U4:','Y'),('U5:','Y'),('U6:','Y'),('U7:','Y'),('U2_2:','Y'),('U3_3:','Y'),('U4_4:','Y'),
   ('U5_5:', 'Y'), ('U6_6:', 'Y'), ('U7_7:', 'Y'), ('U4_7:', 'Y'), ('U5_7:', 'Y'))
U1_job, U2_job, U3_job, U4_job, U5_job, U6_job, U7_job, U2_2_job, U3_3_job, U4_4_job, U5_5_job, U6_6_job, U7_7_job, U4_7_job,
U5_7_job=getInputs(fields=fields,
   label='Specify jobs to submit', dialogTitle='Job submission', )


job_list=[U1_job, U2_job, U3_job, U4_job, U5_job, U6_job, U7_job, U2_2_job, U3_3_job, U4_4_job, U5_5_job, U6_6_job, U7_7_job,
U4_7_job,  U5_7_job]
step_list = ['U1', 'U2', 'U3', 'U4', 'U5', 'U6', 'U7', 'U2_2', 'U3_3', 'U4_4', 'U5_5', 'U6_6', 'U7_7', 'U4_7', 'U5_7']


# U1
if U1_job == 'Y':
   if U1 == 'left':
      resume_tooth('UL1')
   else:
      resume_tooth('UR1')
   fields=(('Y force:','Y'),('Z force:','Y'))
   y_force, z_force = getInputs(fields=fields, label='Specify desired directions for U1', dialogTitle='Directions for analysis', )
   ify_force=='Y':
      resume_step('U1_y_force')
      submit_job('U1_y_force')
      suppress_step('U1_y_force')
   if z_force == 'Y':
      resume_step('U1_z_force')
      submit_job('U1_z_force')
      suppress_step('U1_z_force')
   if U1 == 'left':
```

```
        suppress_tooth('UL1')
    else:
        suppress_tooth('UR1')
# U2
if U2_job == 'Y':
    if U2 == 'left':
        resume_tooth('UL2')
    else:
        resume_tooth('UR2')
    fields=(('Y force:', 'Y'), ('Z force:', 'Y'))
    y_force, z_force = getInputs(fields=fields, label='Specify desired directions for U2', dialogTitle='Directions for analysis', )
    ify_force=='Y':
        resume_step('U2_y_force')
        submit_job('U2_y_force')
        suppress_step('U2_y_force')
    if z_force == 'Y':
        resume_step('U2_z_force')
        submit_job('U2_z_force')
        suppress_step('U2_z_force')
    if U2 == 'left':
        suppress_tooth('UL2')
    else:
        suppress_tooth('UR2')

# U3
if U3_job == 'Y':
    if U3 == 'left':
        resume_tooth('UL3')
    else:
        resume_tooth('UR3')
    fields=(('Y force:', 'Y'), ('Z force:', 'Y'))
    y_force, z_force = getInputs(fields=fields, label='Specify desired directions for U3', dialogTitle='Directions for analysis', )
    ify_force=='Y':
        resume_step('U3_y_force')
        submit_job('U3_y_force')
```

```
        suppress_step('U3_y_force')
    if z_force == 'Y':
        resume_step('U3_z_force')
        submit_job('U3_z_force')
        suppress_step('U3_z_force')
    if U3 == 'left':
        suppress_tooth('UL3')
    else:
        suppress_tooth('UR3')

# U4
if U4_job == 'Y':
    if U4 == 'left':
        resume_tooth('UL4')
    else:
        resume_tooth('UR4')
    fields=(('Y force:','Y'),('Z force:','Y'))
    y_force, z_force = getInputs(fields=fields, label='Specify desired directions for U4', dialogTitle='Directions for analysis', )
    ify_force=='Y':
        resume_step('U4_y_force')
        submit_job('U4_y_force')
        suppress_step('U4_y_force')
    if z_force == 'Y':
        resume_step('U4_z_force')
        submit_job('U4_z_force')
        suppress_step('U4_z_force')
    if U4 == 'left':
        suppress_tooth('UL4')
    else:
        suppress_tooth('UR4')

# U5
if U5_job == 'Y':
    if U5 == 'left':
        resume_tooth('UL5')
```

```
    else:
        resume_tooth('UR5')
    fields=(('Y force:','Y'),('Z force:','Y'))
    y_force, z_force = getInputs(fields=fields, label='Specify desired directions for U5', dialogTitle='Directions for analysis', )
    if y_force=='Y':
        resume_step('U5_y_force')
        submit_job('U5_y_force')
        suppress_step('U5_y_force')
    if z_force == 'Y':
        resume_step('U5_z_force')
        submit_job('U5_z_force')
        suppress_step('U5_z_force')
    if U5 == 'left':
        suppress_tooth('UL5')
    else:
        suppress_tooth('UR5')


# U6
if U6_job == 'Y':
    if U6 == 'left':
        resume_tooth('UL6')
    else:
        resume_tooth('UR6')
    fields=(('Y force:','Y'),('Z force:','Y'))
    y_force, z_force = getInputs(fields=fields, label='Specify desired directions for U6', dialogTitle='Directions for analysis', )
    if y_force=='Y':
        resume_step('U6_y_force')
        submit_job('U6_y_force')
        suppress_step('U6_y_force')
    if z_force == 'Y':
        resume_step('U6_z_force')
        submit_job('U6_z_force')
        suppress_step('U6_z_force')
    if U6 == 'left':
        suppress_tooth('UL6')
```

```
    else:
        suppress_tooth('UR6')
# U7
if U7_job == 'Y':
    if U7 == 'left':
        resume_tooth('UL7')
    else:
        resume_tooth('UR7')
    fields=(('Y force:','Y'),('Z force:','Y'))
    y_force, z_force = getInputs(fields=fields, label='Specify desired directions for U7', dialogTitle='Directions for analysis', )
    if y_force=='Y':
        resume_step('U7_y_force')
        submit_job('U7_y_force')
        suppress_step('U7_y_force')
    if z_force == 'Y':
        resume_step('U7_z_force')
        submit_job('U7_z_force')
        suppress_step('U7_z_force')
    if U7 == 'left':
        suppress_tooth('UL7')
    else:
        suppress_tooth('UR7')

# Multi tooth groups
for i in range(3, 9):
    if job_list[i] == 'Y':
        resume_merge('{}'.format(i - 1))
        fields=(('Y force:','Y'),('Z force:','Y'))
        y_force, z_force = getInputs(fields=fields, label='Specify desired directions for {}'.format(step_list[i]), dialogTitle='Directions for analysis', )
        if y_force=='Y':
            resume_step('U{}_{}_y_force'.format(i - 1, i - 1))
            submit_job('U{}_{}_y_force'.format(i - 1, i - 1))
            suppress_step('U{}_{}_y_force'.format(i-1,i-1))
        if z_force == 'Y':
            resume_step('U{}_{}_z_force'.format(i-1,i-1))
```

```
        submit_job('U{}_{}_z_force'.format(i - 1, i - 1))
        suppress_step('U{}_{}_z_force'.format(i - 1, i - 1))
    suppress_merge('{}'.format(i - 1))


# U4_7
if U4_7_job == 'Y':
    sim_root.features['U4_7'].resume()
    if U4_7 == 'left':
        for i in range(4, 8):
            sim_root.features['UL{}_PDL'.format(i)].resume()
            sim_model.constraints['UL{}_PDL_4_7'.format(i)].resume()
            sim_model.constraints['UL{}_socket_PDL'.format(i)].resume()
    else:
        for i in range(4, 8):
            sim_root.features['UR{}_PDL'.format(i)].resume()
            sim_model.constraints['UR{}_PDL_4_7'.format(i)].resume()
            sim_model.constraints['UR{}_socket_PDL'.format(i)].resume()
    fields = (('Y force:', 'Y'), ('Z force:', 'Y'))
    y_force, z_force = getInputs(fields=fields, label='Specify desired directions for U4_7', dialogTitle='Directions for analysis', )
    if y_force == 'Y':
        resume_step('U4_7_y_force')
        submit_job('U4_7_y_force')
        suppress_step('U4_7_y_force')
    if z_force == 'Y':
        resume_step('U4_7_z_force')
        submit_job('U4_7_z_force')
        suppress_step('U4_7_z_force')
    sim_root.features['U4_7'].suppress()
    if U4_7 == 'left':
        for i in range(4, 8):
            sim_root.features['UL{}_PDL'.format(i)].suppress()
            sim_model.constraints['UL{}_PDL_4_7'.format(i)].suppress()
            sim_model.constraints['UL{}_socket_PDL'.format(i)].suppress()
    else:
        for i in range(4, 8):
```

```
            sim_root.features['UR{}_PDL'.format(i)].suppress()
            sim_model.constraints['UR{}_PDL_4_7'.format(i)].suppress()
            sim_model.constraints['UR{}_socket_PDL'.format(i)].suppress()


# U5_7
if U5_7_job == 'Y':
    sim_root.features['U5_7'].resume()
    if U5_7 == 'left':
        for i in range(5, 8):
            sim_root.features['UL{}_PDL'.format(i)].resume()
            sim_model.constraints['UL{}_PDL_5_7'.format(i)].resume()
            sim_model.constraints['UL{}_socket_PDL'.format(i)].resume()
    else:
        for i in range(5, 8):
            sim_root.features['UR{}_PDL'.format(i)].resume()
            sim_model.constraints['UR{}_PDL_5_7'.format(i)].resume()
            sim_model.constraints['UR{}_socket_PDL'.format(i)].resume()
    fields = (('Y force:', 'Y'), ('Z force:', 'Y'))
    y_force, z_force = getInputs(fields=fields, label='Specify desired directions for U5_7', dialogTitle='Directions for analysis', )
    if y_force == 'Y':
        resume_step('U5_7_y_force')
        submit_job('U5_7_y_force')
        suppress_step('U5_7_y_force')
    if z_force == 'Y':
        resume_step('U5_7_z_force')
        submit_job('U5_7_z_force')
        suppress_step('U5_7_z_force')
    sim_root.features['U5_7'].suppress()
    if U5_7 == 'left':
        for i in range(5, 8):
            sim_root.features['UL{}_PDL'.format(i)].suppress()
            sim_model.constraints['UL{}_PDL_5_7'.format(i)].suppress()
            sim_model.constraints['UL{}_socket_PDL'.format(i)].suppress()
    else:
        for i in range(5, 8):
```

```
sim_root.features['UR{}_PDL'.format(i)].suppress()
sim_model.constraints['UR{}_PDL_5_7'.format(i)].suppress()
sim_model.constraints['UR{}_socket_PDL'.format(i)].suppress()
```

7. 3D_processing_fx.py

```
#
#
# After job is run, analyzes rxn forces from .odb and estimates CR in vertical axis
# (force with y-vector)


from  odbAccess  import  *
from abaqusConstants import *
from odbMaterial import *
from odbSection import *


import numpy as np
import math


# Access ODB
# Define job specific variables
job_name = name + '.odb'
step_name  = name
print("""
Job {}
""".format(job_name))
odb = openOdb(job_name)
print("""
Step {}
""".format(step_name))


#          Get          coord          from          set
sim_model          =          mdb.models['Simulation']
sim_root       =       sim_model.rootAssembly
force_location = sim_root.sets[name].nodes[0].coordinates
moment_ref =  np.array(force_location)


# Read field output from last frame (along Y-axis)


last_frame  = odb.steps[step_name].frames[-1]
```

```
rxn_forces = last_frame.fieldOutputs['RF']
orig_coord = last_frame.fieldOutputs['COORD']
displacement = last_frame.fieldOutputs['U']

rxn_field_values = rxn_forces.values
orig_node_coord = orig_coord.values
U = displacement.values

length = len(rxn_field_values)

# Define initial sum of forces and moments

pos_forces = np.zeros(3)
neg_forces = np.zeros(3)
pos_moments = np.zeros(3)
neg_moments = np.zeros(3)

for i in range(length):

        # Find deformed coordinates

        orig_coord_array = np.array(orig_node_coord[i].data)
        node_displacement  =  np.array(U[i].data)
        new_node_coord = orig_coord_array +  node_displacement

        # Create array of ([r_y, r_z], [F_y, F_z]) for moment about x-axis

        current_force_array = np.array(rxn_field_values[i].data)

        r = new_node_coord - moment_ref
        F = current_force_array

        # Determine moment at node

        current_moment = np.cross(r, F)
```

```
        # Sort forces/moments according to +/- direction
        # For force vectors in the same direction as original
        if np.dot(orig_force_vector, F) > 0:
                pos_forces += current_force_array
                pos_moments += current_moment
        elif np.dot(orig_force_vector, F) < 0:
                neg_forces += current_force_array
                neg_moments += current_moment

print ("""
The reaction forces are {} and {}, and the two moments created by those
forces about the point {} are {} and {}.
""".format(pos_forces, neg_forces, moment_ref,
pos_moments, neg_moments))

# Test for CR
# Determine magnitude of moment vectors in order to then divide by force and find magnitude of r vectors
a2 = pos_moments[0]**2 + pos_moments[1]**2 + pos_moments[2]**2
pos_moment_mag=math.sqrt(a2)
print('pos_moment_mag = ' + str(pos_moment_mag))
a2 = neg_moments[0]**2 + neg_moments[1]**2 + neg_moments[2]**2
neg_moment_mag = math.sqrt(a2)
print('neg_moment_mag = ' + str(neg_moment_mag))

if abs(pos_moment_mag) <= 0.01 and abs(neg_moment_mag) <= 0.01:
        print("The approximate location of CR is {}.".format(moment_ref))
        iterate_check = False
else:
        iterate_check = True
        # Determine magnitude of force vectors
        a2 = pos_forces[0]**2 + pos_forces[1]**2 + pos_forces[2]**2
        pos_force_mag = math.sqrt(a2)
        print('pos_force_mag = ' + str(pos_force_mag))
        a2 = neg_forces[0]**2 + neg_forces[1]**2 + neg_forces[2]**2
```

```
neg_force_mag = math.sqrt(a2)
print('neg_force_mag = ' + str(neg_force_mag))

# Determine magnitude of r vectors
r_mag_pos = pos_moment_mag / pos_force_mag
print('r_mag_pos = ' + str(r_mag_pos))
r_mag_neg = neg_moment_mag / neg_force_mag
print('r_mag_neg = ' + str(r_mag_neg))

# Determine unit vector for r (F X moment per right hand rule)
r_pos = np.cross((pos_forces + orig_force_vector), pos_moments)
r_neg = np.cross(neg_forces, neg_moments)
a2 = r_pos[0]**2 + r_pos[1]**2 + r_pos[2]**2
r_cross_pos_mag = math.sqrt(a2)
a2 = r_neg[0]**2 + r_neg[1]**2 + r_neg[2]**2
r_cross_neg_mag = math.sqrt(a2)

r_pos_unit = r_pos / r_cross_pos_mag
print('r_pos_unit = ' + str(r_pos_unit))
r_neg_unit = r_neg / r_cross_neg_mag
print('r_neg_unit='+str(r_neg_unit))

# Determine resultant r vectors
r_pos_forces = r_pos_unit * r_mag_pos
print('r_pos_forces='+str(r_pos_forces))
r_neg_forces = r_neg_unit * r_mag_neg
print('r_neg_forces = ' + str(r_neg_forces))

# Estimate new locations
pos_est = moment_ref + r_pos_forces
neg_est = moment_ref + r_neg_forces
print ("The estimated locations of the balancing forces from {} are {} and {}.".format(moment_ref, pos_est, neg_est))
avg = (pos_est + neg_est) / 2
new_force_location = avg
a2 = pos_est[0]**2 + pos_est[1]**2 + pos_est[2]**2
```

```
            pos_est_mag = math.sqrt(a2)
            a2 = neg_est[0]**2 + neg_est[1]**2 + neg_est[2]**2
            neg_est_mag = math.sqrt(a2)
            print("The estimated location is {}. Please verify force system at this location.".format(avg))
            xcoord = new_force_location[0]
            ycoord = new_force_location[1]
            zcoord = new_force_location[2]

closeOdb(odb)

file_name = '{}.lck'.format(name)
file_path='{}{}'.format(directory,file_name)
os.remove(file_path)
```

8. Bulk_process.py

```
#
#
# Run post-process function for several jobs at a time


# Get user input regarding which jobs to process
fields=(('U1:','Y'),('U2:','Y'),('U3:','Y'),('U4:','Y'),('U5:','Y'),('U6:','Y'),('U7:','Y'),('U2_2:','Y'),('U3_3:','Y'),('U4_4:','Y'),
  ('U5_5:', 'Y'), ('U6_6:', 'Y'), ('U7_7:', 'Y'), ('U4_7:', 'Y'), ('U5_7:', 'Y'))
U1_job, U2_job, U3_job, U4_job, U5_job, U6_job, U7_job, U2_2_job, U3_3_job, U4_4_job, U5_5_job, U6_6_job, U7_7_job, U4_7_job,
U5_7_job=getInputs(fields=fields,
  label='Specify jobs to analyze', dialogTitle='Analyze multiple jobs', )


job_list=[U1_job, U2_job, U3_job, U4_job, U5_job, U6_job, U7_job, U2_2_job, U3_3_job, U4_4_job, U5_5_job, U6_6_job, U7_7_job,
U4_7_job,  U5_7_job]
step_list = ['U1', 'U2', 'U3', 'U4', 'U5', 'U6', 'U7', 'U2_2', 'U3_3', 'U4_4', 'U5_5', 'U6_6', 'U7_7', 'U4_7', 'U5_7']


for n in range(0, 11):
  if job_list[n] == 'Y':
    fields = (('Y force:', 'Y'), ('Z force:', 'Y'))
    y_force, z_force = getInputs(fields=fields, label='Specify desired directions for {}'.format(step_list[n]), dialogTitle='Directions for analysis', )
    if n < 8:
      instance = getInput('Instance for nearest_node module:')
    else:
      instance = step_list[n]
    # Resume suppressed components for analysis
    sim_root.features[instance].resume()
    if y_force=='Y':
      name = '{}_y_force'.format(step_list[n])
      orig_force_vector = np.array([0, 1, 0])
      PP3D(name)
      if iterate_check == True:
        nearest_node(plugin_dir, instance, xcoord, ycoord, zcoord,  name)
    if z_force == 'Y':
      name = '{}_z_force'.format(step_list[n])
      orig_force_vector = np.array([0, 0, 1])
      PP3D(name)
```

```
    if          iterate_check          ==          True:
        nearest_node(plugin_dir, instance, xcoord, ycoord, zcoord,  name)
sim_root.features[instance].suppress()
```

1. Apply an initial load F_n at a random location x on the model

2. Read node coordinates : total displacement d and the reaction forces F_R at each node as a result of F_n

3. Group these forces into two resultant force vectors: one in the same direction F_{Ri} and one in the opposite direction F_{Ro} of the original applied load

4. For each node, use cross product $r \times F$ to calculate the resulting moments from the reaction force F_R at that node about the point of original force application and sum the moments to determine M_i and M_o

NB. $M_i = r_i \times F_{Ri}$ and $M_o = r_o \times F_{Ro}$

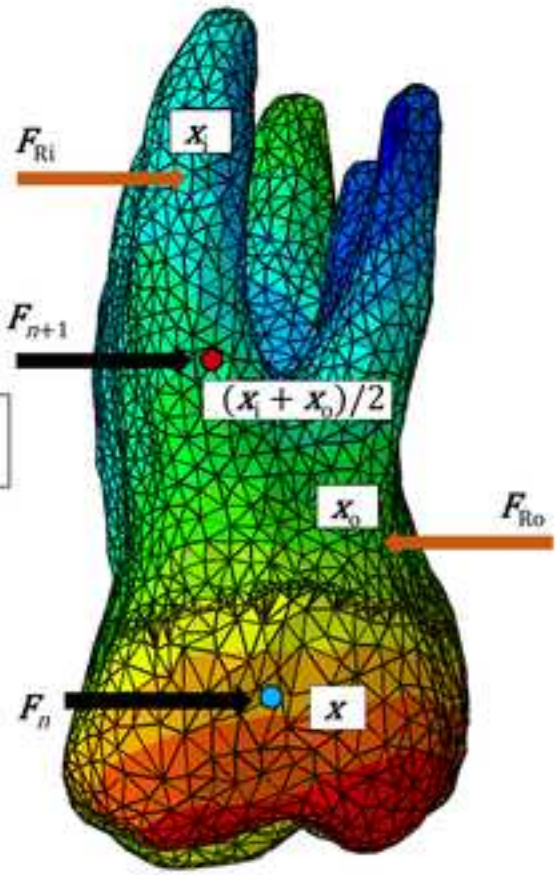
5. Determine the effective locations of the two force vectors via r_i and r_o as

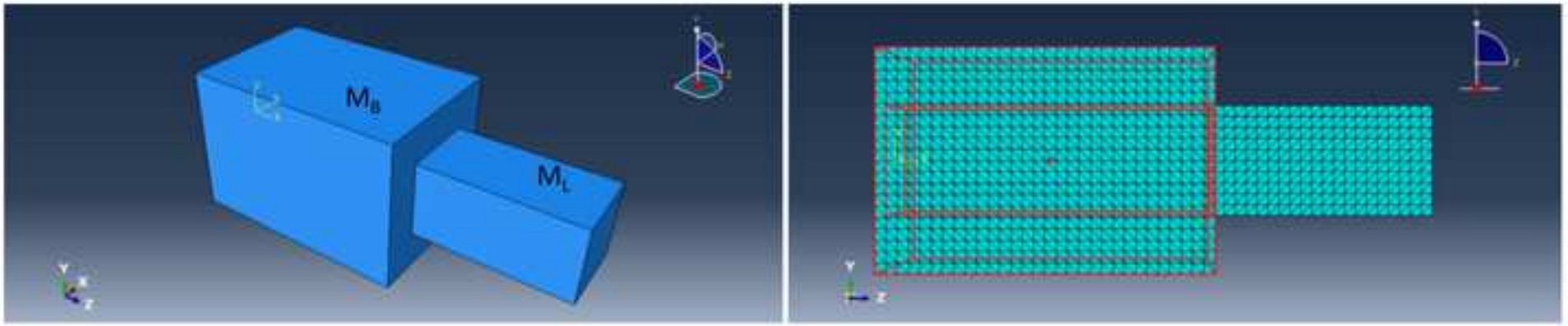
$r_{mag} = |M|/|F|$ with $r = r_{unit}\, r_{mag}$

The locations given as $x_i = x + r_i$ and $x_o = x + r_o$

6. Estimate the midpoint as a new location for force application F_{n+1} as $(x_i + x_o)/2$

7. Iterate until M_i or $M_o \cong 0$



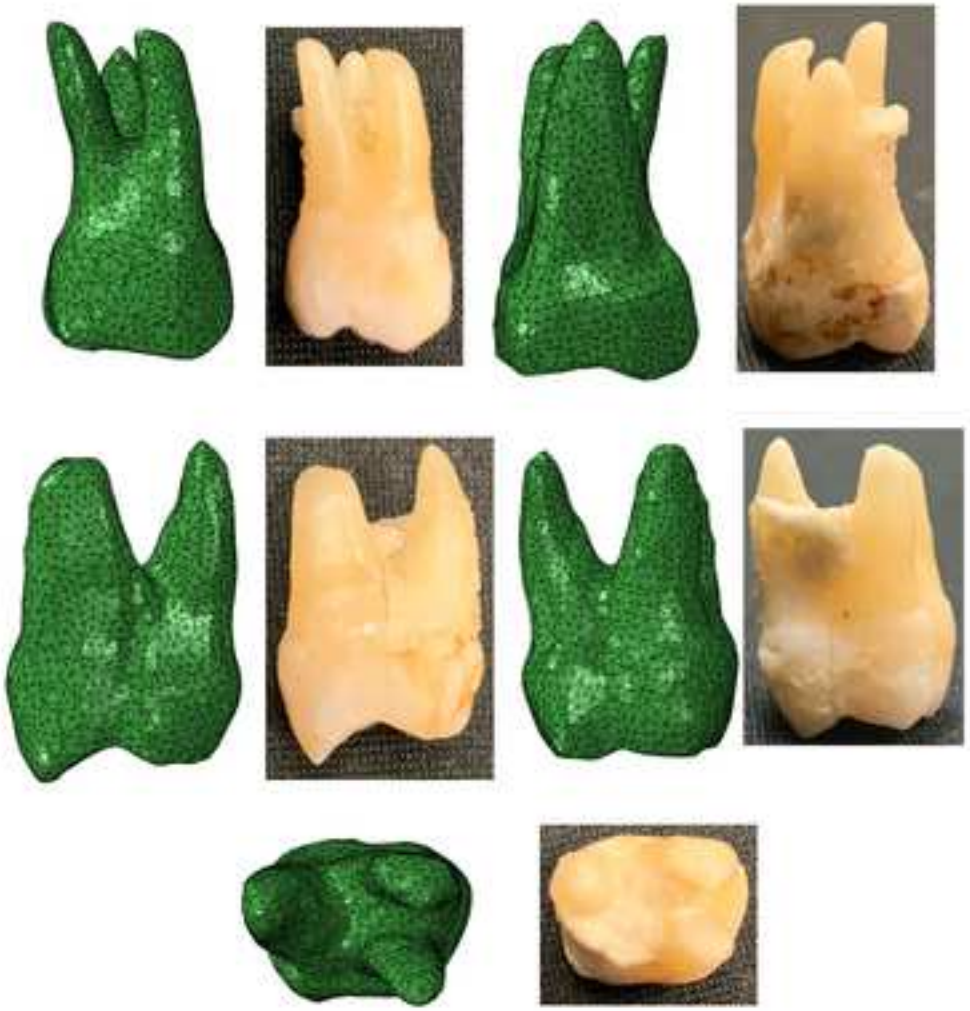


Let the longer (L) beam have mass M_L
And the bigger beam (B) have mass M_B
The total mass of the system = $M = M_B + M_L$

Assuming that the M of the entire system is located at G (red dot) and located by coordinates: x, y & z w.r.t a cartesian coordinate system O-XYZ, and each Center of mass : for M_B & M_L is located at : x_B, y_B , z_B & x_L, y_L , z_L respectively.

Summing the moments about the y –axis: $xM = x_B.M_B + x_L.M_L$
Similarly, $yM = y_B.M_B + y_L.M_L$ and $zM = z_B.M_B + z_L.M_L$

$$x = \frac{x_B.M_B + x_L.M_L}{M_B + M_L} \qquad y = \frac{y_B.M_B + y_L.M_L}{M_B + M_L} \qquad z = \frac{z_B.M_B + z_L.M_L}{M_B + M_L}$$



Parameter		Tooth (Actual)	Tooth(FEM)	Difference
	MB	17.49	17.23	0.26
Tooth length	DB	16.75	16.52	0.23
	P	16.72	16.59	0.13
	MB	10.95	10.21	0.74
Root length	DB	10.1	10.88	-0.78
	P	12.02	11.89	0.13
	MB-DB	4.07	4.01	0.06
Distance between roots	DB-P	8.16	8.12	0.04
	P-MB	7.43	7.38	0.05

**MB= Mesio-buccal; DB= Disto-buccal; P= Palatal*

***All measurements are in mm*