Automatic 3D Quantitative Analysis of Intra-Articular Calcaneus Fractures in the Posterior Talocalcaneal Joint: Development and Validation of a Segmentation and Measurement Method

Cile van Holthe

MSc Thesis Technical Medicine

November 2024

Automatic 3D Quantitative Analysis of Intra-Articular Calcaneus Fractures in the Posterior Talocalcaneal Joint: Development and Validation of a Segmentation and Measurement Method

Cile van Holthe

4644557

02-11-2024

Thesis in partial fulfilment of the requirements for the joint degree of Master of Science in

Technical Medicine

Leiden University ; Delft University of Technology ; Erasmus University Rotterdam

Master thesis project (TM30004),
Erasmus MC,
Image Guided Interventions and Therapy
Department of Radiology & Department of Trauma Surgery

Chair/Technical Supervisor: Dr. ir. T. (Theo) van Walsum, Erasmus MC
Medical Supervisor: Dr. M.G. (Mark) van Vledder, Erasmus MC
Daily Supervisor: A. (Alexander) Wakker, Erasmus MC
Independent Thesis Committee member: Dr. M. (Monique) Reijnierse, Leiden University Medical Center

Universiteit Leiden

TU Delft Delft University of Technology

Erasmus Universiteit Rotterdam

Contents

Abstract

Introduction

Calcaneus fractures represent 60% of tarsal bone fractures and are particularly challenging to assess when they involve the posterior talocalcaneal (PTC) facet. Accurate evaluation of intra-articular fractures, including metrics such as gap area and step-off, is crucial for treatment planning but remains difficult due to limitations of using 2D cross-sections from 3D CT imaging. Deciding between surgical or conservative management from these images often leads to significant inter- and intra-observer variability, complicating clinical decisions and affecting outcomes. This study aims to develop and validate an AI-based method for automatic segmentation and 3D quantitative analysis of PTC fractures, improving assessment and providing more objective data for decision-making.

Methods

A retrospective study was conducted on CT scans from 44 patients for training using 5-fold cross-validation and 33 patients for external validation. The nnU-Net framework was trained to segment the PTC fragments and posterior talar facet (PTF). Automatic 3D measurements, including gap area, inter-articular distances, maximal step-off, and maximal gap, were computed from the segmented models. Results were validated against manual 2D measurements performed by two observers, as well as through comparison with the external validation set.

Results

The nnU-Net achieved a Dice score of 0.78 for PTC segmentation in the training set and 0.75 in the external validation set. Moderate positive correlations were observed between the 3D automatic measurements and manual 2D measurements. Specifically, the correlation between 3D gap area and 2D maximal gap measurement was Spearman's rho = 0.62, while the correlation between 3D and 2D maximal step-off measurements was rho = 0.52. Additional correlations were found between the 3D fracture area and the 2D maximal gap and step-off measurements, with rho values of 0.65 and 0.64, respectively, indicating that the 3D analysis is consistent with corresponding 2D measurements.

Conclusion

This study introduces an AI-based method for automatic 3D analysis of calcaneus fractures, offering faster and more detailed fracture metrics, which may improve treatment planning over traditional 2D slice evaluations.

1. | Introduction

Calcaneus fractures account for 60% of tarsal bone fractures (1). These fractures typically result from concentrated axial loading forces, such as those occurring after a fall or jump from a height, or as a result of a road traffic incident, where the impact drives the talus bone distally into the calcaneus. Calcaneus fractures can be classified into extra- and intra-articular types. Intra-articular fractures involve the calcaneocuboid joint or any of the three subtalar joint surfaces. Among the subtalar joint surfaces, the posterior talocalcaneal (PTC) joint facet is the largest and serves as the primary weight-bearing surface of the calcaneus, forming a joint with the posterior talar facet (PTF) (Figure 1). The PTC is the most frequently fractured surface in displaced intra-articular calcaneal fractures (2). Achieving anatomical reduction of the PTC facet during calcaneus fracture surgery is essential, as it plays a pivotal role in determining the overall outcome and functionality of the foot (3,4).

The primary goal of treating calcaneus fractures is to restore or maintain the congruent shape of the calcaneus, particularly the joint surfaces, maintain the height of the calcaneus, and the length and width of the heel. Treatment options considered vary between conservative, open reduction and internal fixation (ORIF), minimally invasive approaches, and primary joint arthrodesis. If calcaneus fractures are not properly managed, they can lead to deformities of the hindfoot and arthritis of the subtalar joint and calcaneocuboid joint. The optimal management approach, whether conservative or surgical, is determined by a range of factors, including fracture type (intra-articular or extra-articular), the extent of comminution, the degree of displacement, as well as additional radiological findings and patient-specific characteristics. These fractures are typically assessed and classified using computed tomography (CT) scans.

Although the calcaneus is the most frequently fractured tarsal bone, there are numerous controversies surrounding the management of these fractures, particularly intra-articular fractures (5). The complexity is compounded by the inherent challenges observers face when interpreting CT scans, even with high-resolution techniques. While CT scans provide detailed information, observers are limited by two-dimensional (2D) slices or 3D reconstructions of entire bones, making it difficult to accurately evaluate joint surface areas. This limitation can hinder the evaluation of fracture fragment displacement, including critical metrics such as gap and step-off measurements. These measurements are particularly challenging, as they often span multiple slices and involve assessing the complex 3D relationships between fracture fragments, making it difficult to determine the true extent of displacement in intra-articular fractures.

To address these challenges, a broader effort in evaluating morphological aspects of fractures using 3D reconstructions has been initiated, as highlighted in the study by Wakker et al. (6), which introduced a novel method for performing detailed morphological measurements on three-dimensional models. Building on this, the present study focuses on the application of AI-based automatic analysis specifically for intra-articular fractures of the PTC of the calcaneus.

Moreover, there is currently no universally established protocol that unequivocally prescribes open reduction for intra-articular calcaneus fractures. The Sanders classification, which categorizes fractures based on the number of fracture lines and their involvement in the PTC, is widely used but remains a subject of ongoing debate and does not fully resolve this issue (7,8). These complexities often result in significant inter- and intra-observer variability in Sanders classification(9–11) and the assessment of fracture displacement, including gap and step-off (12). While few studies have focused specifically on calcaneus fractures, similar issues have been observed with other types of fractures (13). Additionally,

due to the difficulties in consistently classifying fractures and assessing displacement, it is challenging to determine whether a patient would benefit from a particular treatment in terms of achieving a better clinical outcome (8).

Lastly, the comprehensive analysis required for accurate interpretation of CT scans can be time-consuming. Due to these limitations, quantitative measurements on CT scans for calcaneus fractures are not frequently performed in clinical practice. The inherent variability in observer assessments, coupled with the time-intensive nature of reviewing multiple imaging planes, makes these measurements challenging to implement consistently. Consequently, the choice of treatment often relies on the surgeon's clinical judgment, specific case findings, and personal preferences, rather than on standardized, objective parameters(14,15).

Recent advancements in artificial intelligence (AI) tools have already shown promise in automating objective parameter evaluation of fractures (16–18). These AI-based quantitative measurements leverage machine learning algorithms to analyze medical imaging data for fracture detection and to extract relevant fracture parameters automatically (19,20). However, these studies have primarily focused on fracture detection and classification based on 2D information, such as X-ray images. Relying solely on 2D images to assess fractures poses challenges in capturing the complete extent of injuries, especially for intra-articular fractures, where three-dimensional (3D) displacement, such as gaps and step-offs, may occur across multiple image slices. Furthermore, certain studies have investigated quantitative measurements on 3D models for fractures other than calcaneal fractures. However, these methodologies typically require multiple manual interventions, preventing the measurements from being fully automated (13,21).

To address the challenges inherent in the diagnostic process and ensure objective fracture evaluation of intra-articular fractures of the PTC of the calcaneus, the implementation of AI-based automatic analysis using 3D models derived from CT imaging, and resulting objective parameters offers a promising solution.

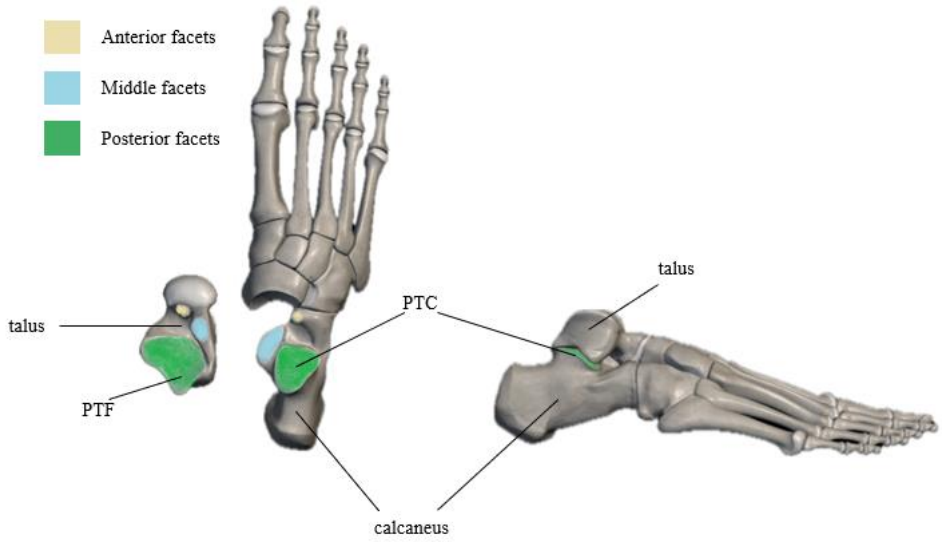


Figure 1: Anatomical overview of the calcaneus, talus, posterior talocalcaneal facet (PTC) and posterior talar facet (PTF). Axial view with flipped talus (left) and lateral view with talus in correct position (right) (22).

2. | Methods

This section delineates our approach in the following order: (1) patient selection and data acquisition, (2) automatic segmentation, (3) automatic 3D measurements, (4) 2D measurements, and (5) statistical analysis.

2.1 | Patient selection

In this retrospective multicenter study, the dataset consisted of patients with one or multiple calcaneus fractures who underwent diagnostic imaging studies between 2006 and 2021 at ErasmusMC or between 2019 and 2023 at Maasstad Hospital. All data were anonymized before use. The inclusion criteria for the dataset were: (1) both the calcaneus and talus bones were fully visible on the CT scan, (2) the patient age was 16 years or older, (3) the talus was not fractured and (4) slice thickness was less or equal to 1 mm. An external validation set (2014–2024) adhered to the same inclusion criteria. The local Medical Research Ethics Committee (No. MEC-2024-0380) reviewed and exempted the study protocol. Considering the study's design, the committee waived the requirement for obtaining informed consent from the participants.

2.2 | Automatic segmentation

In order to perform automatic 3D measurements, the generation of accurate 3D models through automated segmentation is essential. The subsequent sections outline the methods for creating these models, as well as the training and application of the automated segmentation algorithms utilized in this study. See Figure 3 for all steps in the automatic segmentation method.

2.2.1 | 3D model creation

For the training of the automated segmentation model, 3D models were manually segmented by a single observer for all patients in the training dataset. These 3D models were generated from the original CT data using the Mimics Research software package (Version 26.0, Materialise, Leuven, Belgium). First, the CT data (DICOM files) were imported, and bony tissue was extracted by creating a mask using a threshold (Hounsfield units > 226). The mask was then split to ensure separate masks were created for the talus and calcaneus. The region-growing tool was applied to remove noise and exclude adjacent bony structures. The PTC surface of the calcaneus mask and talus mask was refined using smart fill to enhance mask accuracy.

The talus and calcaneus masks were then transformed into objects to allow import into the Materialise 3-matic software package (Version 18.0, Materialise, Leuven, Belgium), where a smoothing algorithm was applied to refine the model surfaces by removing sharp edges and smoothing rough areas. The fragmented PTC and PTF surfaces were manually marked to extract these regions. From these surfaces, 3D segments were created by moving the surface 2 mm inward, generating 3D segments with a thickness of 2 mm for each marked surface.

These 3D segments were re-imported into Mimics and transformed back into masks to verify correspondence with the original CT data, and adjustments were made as necessary. See Figure 2 for an overview of the 3D model creation steps. Along with the PTF and PTC fragment masks, the complete talus mask was also retained for subsequent steps. The images, including the masks of the 3D segments and the entire talus, each represented by a uniform intensity value, were then exported as DICOM files.

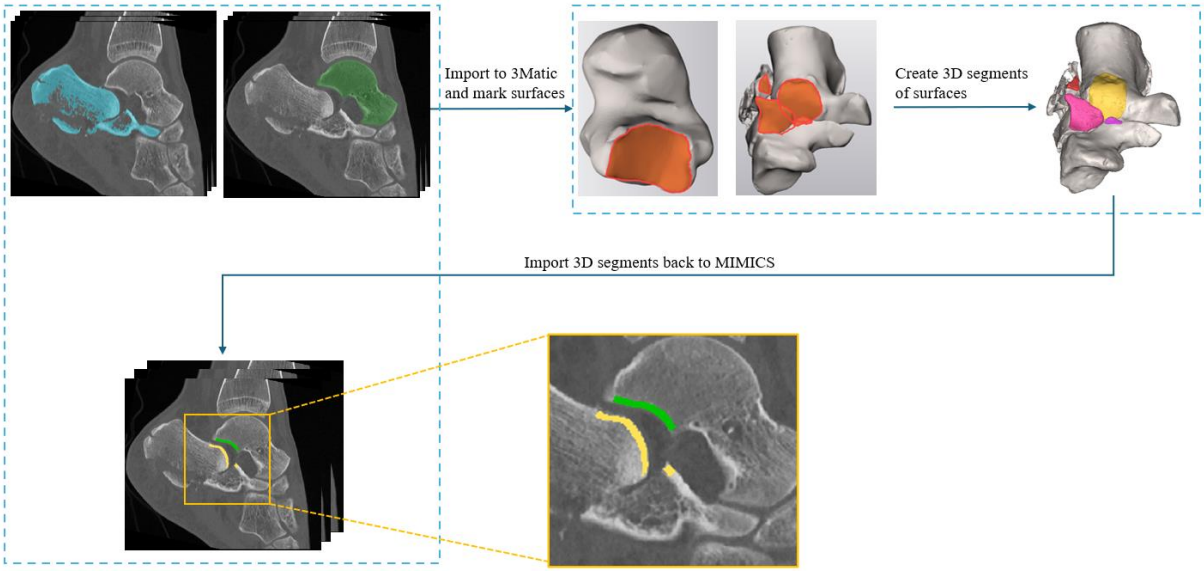


Figure 2: Workflow for 3D model creation. CT data is imported into Mimics, transferred to 3-matic for surface marking, and 3D segments are created. The segments are then imported back into Mimics for verification.

2.2.2 | Preprocessing

All DICOM files containing segmentations were converted into label masks in the Neuroimaging Informatics Technology Initiative (NIfTI) format using Python, as required for the subsequent processing steps. Additionally, the raw images, without segmentations, were also converted to NIfTI format. After conversion, label values were assigned to the segmentations: the background was assigned a label value of 0, the PTF a label value of 1, and the PTC fragments a label value of 2. By assigning these distinct label values, the automated segmentation method was able to correctly identify and segment the different structures.

2.2.3 | Automatic segmentation

The NIfTI files containing the PTC and PTF label masks, along with the unsegmented NIfTI images, were used to train the automatic segmentation framework. The framework utilized nnU-Net, a deep convolutional neural network (CNN) developed by Isensee et al. The nnU-Net was trained using a 5-fold cross-validation with the 3D low-resolution configuration, splitting the data 80:20 between training and validation sets. This approach enabled the model to automatically segment the PTC fragments and the PTF. Additionally, another nnU-Net model was trained for the segmentation of the entire talus, using the same configuration and cross-validation process as for the PTC and PTF. The training, inference, and postprocessing of both models were executed on a GPU cluster using a SLURM script, with details provided in Appendix A. Pre- and post-processing steps, such as resampling, normalization, and connected component analysis, were handled automatically by nnU-Net.

Once the automated configuration and training were completed, an ensemble of the fully trained models from the 5 folds was used to make predictions on unseen images.

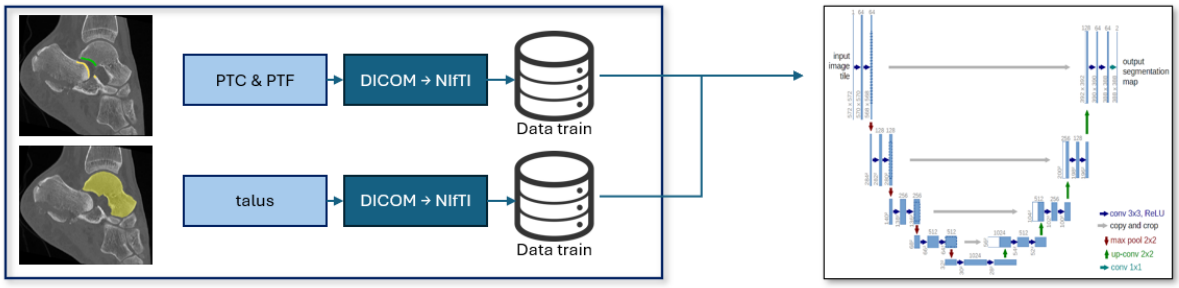


Figure 3: Overview of the automatic segmentation process. The left panel shows the raw DICOM files overlayed with the label masks of the PTC, PTF and talus which are then converted into NIfTI format, which are then used to train the nnU-Net model (right panel). The right panel illustrates the U-Net architecture used by nnU-Net for segmentation, where the input image tile is processed through convolutional layers, max pooling, and upsampling to produce the final output segmentation map(23)

2.3 | Automatic 3D measurements

To perform quantitative analyses of the fractures, 3D measurements were required based on the automatically segmented models. The following sections describe the methodology for creating and processing 3D models, including the calculation of gap areas, inter-articular distances, surface and fracture areas, and maximal step-off and maximal gap measurements, using a series of custom scripts and established algorithms (Figure 4).

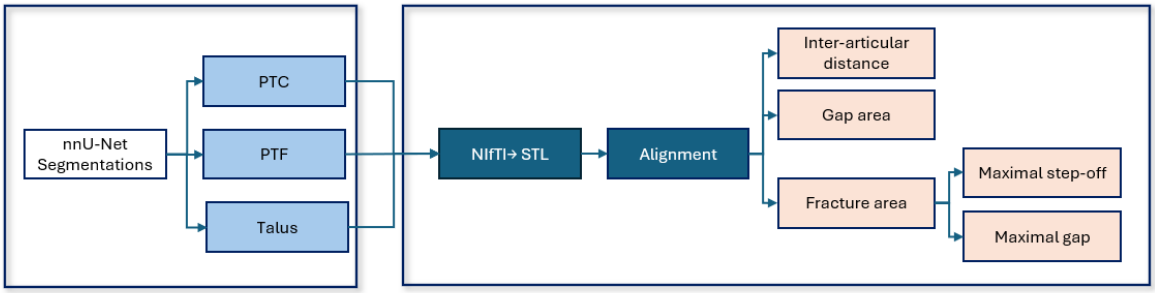


Figure 4: Overview of the steps for automatic 3D measurements. nnU-Net segmentations of the PTC, PTF, and talus are converted from NIfTI to STL format. The generated STL meshes are then aligned using mean shape models. Subsequently, gap area and inter-articular distance measurements are computed. Fracture area measurements follow, along with maximal step-off and maximal gap calculations.

2.3.1 | Creation of STL Models

A custom Python script was developed to convert the voxel-based NIfTI label masks into STL files. Segmented regions corresponding to the talus, the PTF, and the PTC fragments were identified from the NIfTI files. The Marching Cubes algorithm was employed with a threshold of 0,5 to generate 3D surface meshes from these segmented regions.

For the PTC, where all fragments initially shared the same label, a connectivity-based region-growing algorithm was applied as post-processing to separate and label each fragment individually. This allowed for the generation of separate STL files for each PTC fragment.

Additionally, Laplacian smoothing to the STL files of the PTC fragments, PTF, and talus was applied to reduce noise and improve surface quality. The smoothing process iteratively adjusted the mesh vertices based on the Laplacian of the vertex adjacency graph, with a smoothing factor ($\lambda = 0.05$) and 10 iterations. The values for λ and the number of iterations were chosen iteratively after visual inspection to ensure that the surfaces were smoothed without creating gaps or losing other important structural information. All resulting smoothed STL files were saved for subsequent analysis.

2.3.2 | Alignment of STL Models

The generated STL meshes of the talus were aligned using a mean shape model of the right talus, and a mirrored version of the mean shape model for the left talus, ensuring consistent positioning within the same world coordinate system, as described by Wakker et al. (24). The PTF and PTC fragments were translated and rotated using the same transformation matrix as the talus, preserving their relative positions. The aligned 3D models were saved for further analysis. Additionally, a mean shape model of the calcaneus, positioned at the average anatomical location relative to the mean talus shape model, was available and used for subsequent analysis. (See Appendix C for the Python script corresponding to sections 2.3.1 and 2.3.2)

2.3.3 | Gap Area Calculation

Following the alignment of the talus, PTC fragments, and PTF, several steps were performed to quantify potential gaps in articulation. First, a convex hull was generated to enclose all points of the PTC fragments. This convex hull was created by projecting the 3D points onto the first two principal components derived from the Principal Components Analysis (PCA), creating a 2D convex hull as the smallest convex polyhedron enclosing all PTC fragments. The convex hull was then translated by a factor of 3 along the third principal component to position it both anterocranial and posterocaudal to the PTC fragments. The value of 3 was chosen iteratively, ensuring that the PTC fragments of the training cases were positioned correctly between the two convex hulls.

The points of the convex hulls were scaled inward by a factor of 0.8 toward their centroid to ensure that no gap areas were erroneously detected along the outer boundary of the PTC. The value of 0.8 was chosen iteratively by adjusting the scaling factor and visually inspecting the results for all training cases to confirm that the gap area did not extend beyond the actual fragments.

An interpolation method was applied to generate evenly distributed points over the entire surfaces of the convex hulls for further analysis. Then rays were traced from the evenly distributed points on the convex hull at the anterocranial position of the PTC fragments towards the convex hull positioned posterocaudal to the PTC fragments. If a ray did not intersect with the PTC fragments but only with the posterocaudal convex hull, the presence of a gap was indicated. Refer to Figure 5 for the visual representation of the gap area calculation steps.
The gap area percentage G was calculated as the ratio of the number of rays that intersected only the convex hull to the total number of rays traced:

$$G = \frac{N_{gap}}{N_{total}} X\ 100\% \qquad (2)$$

Where N_{gap} is the number of rays that intersected only the posterocaudal convex hull, and N_{total} is the total number of rays traced.

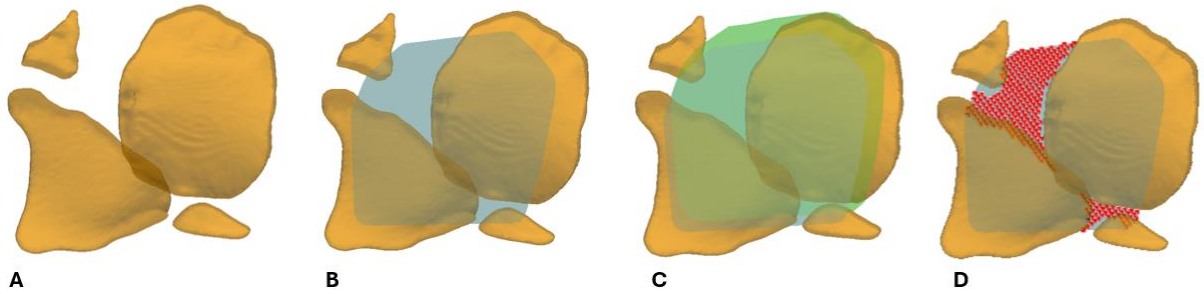


Figure 5: (a) Smoothed STL meshes of PTC fragments, (b) convex hull (blue) of PTC fragments, (c) convex hull at anterocranial position of PTC fragments (green) and convex hull at posterocaudal position of PTC fragments (green), (d) intersection points (red points) of the traced rays on the posterocaudal convex hull.

2.3.4 | Interarticular distance calculations

Several steps were conducted to calculate the distance between the PTC fragments and the PTF. First, the normal vectors $\overrightarrow{n_{PTC_i}}$ and $\overrightarrow{n_{PTF_i}}$ were calculated for each of the triangular faces on the surface mesh of the PTC fragments and the PTF. Thereafter, normal vectors were excluded based on their angular similarity to a reference direction vector $\vec{d}$. The reference direction vector of the PTC fragments $\overrightarrow{d_{PTC}}$ was determined as the vector pointing from the center of mass (COM) of the mean shape model of the calcaneus towards the COM of the PTC fragments. For the PTF, the reference direction vector $\overrightarrow{d_{PTF}}$ was the vector pointing from the COM of the talus to the COM of the PTF. Then, the dot product was calculated and was used to calculate the cosine of the angles between each normal vectors $\overrightarrow{n_{PTC_i}}$ and $\overrightarrow{n_{PTF_i}}$ and the reference direction vectors $\overrightarrow{d_{PTC}}$ and $\overrightarrow{d_{PTF}}$, separately for both the PTC and PTF.
A normal vector was retained if the cosine of the dot product indicated an angle smaller than the predefined threshold of 50 degrees:

$$\cos(\theta_{treshold}) \geq \overrightarrow{n_i} \cdot \vec{d} \qquad (3)$$

The threshold of 50 degrees was determined iteratively on the training dataset by testing different values and visually inspecting the results to ensure that the correct points directly opposite each other were captured without filtering out too many important vectors. After this filtering step, the points p_{PTF_i} and p_{PTC_i}corresponding to the remaining normal vectors $\overrightarrow{n_{PTF_i}}$ and $\overrightarrow{n_{PTC_i}}$ of the PTF and PTC meshes were saved. From the points p_{PTF_i}, rays were traced towards the PTC fragments in the direction of the corresponding normal vectors $\overrightarrow{n_{PTF_i}}$. The points where the rays intersected the PTC fragments surfaces, q_{PTC_i}, were saved.

After this filtering step, the remaining points q_{PTC_i} were systematically redistributed to achieve an even spatial distribution across the entire region previously covered by the set of points q_{PTC_i}.
For every point q_{PTC_i}, a 1-nearest neighbor search was conducted, using KD-tree search algorithm, to find the closest corresponding point p_{PTC_i}.
Then from the corresponding p_{PTC_i} points rays are traced towards the PTF the Euclidean distance between the points p_{PTC_i} on the PTC fragment meshes and the intersection points r_{PTF_i} on the PTF mesh was calculated using the following formula:

$$d_i = \left\| p_{PTC_i} - r_{PTF_i} \right\| \qquad (4)$$

The double ray tracing process ensures that the points where the distances are measured lie directly opposite to each other on both the PTF and PTC, capturing the true articulation between the two surfaces. The results for each patient were visualized accordingly, as illustrated in Figure 6. Additionally, the standard deviation of the inter-articular distances d_i for each patient was computed. Where the standard deviation σ_d provides information on the consistency of the articulation across the joint surface, and therefore indicates if fragments are displaced. (See Appendix D for the Python script corresponding to sections 2.3.3 and 2.3.4).

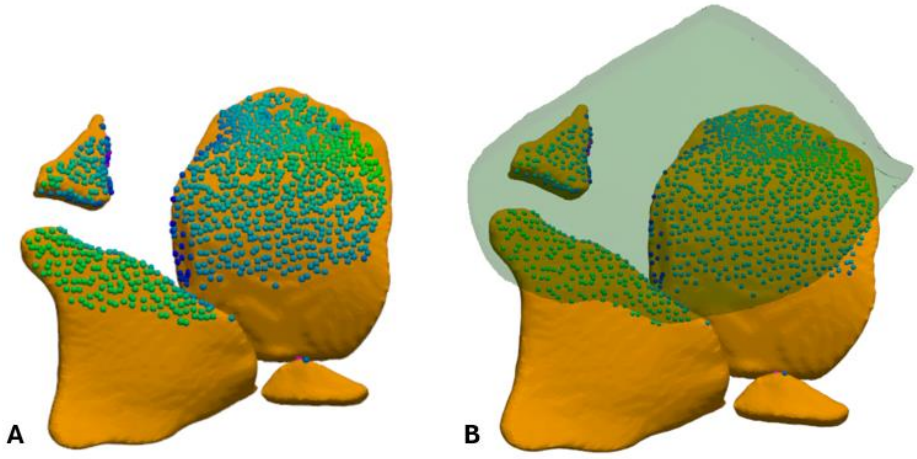


Figure 6: (a) The points p_{PTC_i} on the PTC mesh (orange) are color-coded according to their Euclidean distance d_i from the intersection points r_{PTF_i} on the PTF, transitioning smoothly from green for the points with the smallest distances, d_i, to purple for the points with the longest distances d_i. The minimum and maximum distance values are calculated per patient, ensuring that the color coding reflects the individual variation in distances for each case. (b) Visualized together with transparent corresponding PTF mesh.

2.3.5 | Surface- and fracture area analysis

To accurately assess the surface areas of the PTC fragments and identify the total fracture area, which includes both the total gap and total step within the PTC, a comprehensive series of mesh processing steps were undertaken.

While the STL meshes generated by the marching cubes algorithm typically represent both upper and lower surfaces of 3D objects, only a single surface per fragment was required for surface area measurement. The first step in this process involved detecting sharp edges in the mesh, indicative of fragment boundaries, by evaluating the dihedral angle between adjacent triangular faces of the smoothed STL files. The dihedral angle θ between two faces with normals n_1 and n_2 is calculated as:

$$\cos(\theta) = \frac{\boldsymbol{n_1} \cdot \boldsymbol{n_2}}{\|\boldsymbol{n_1}\| \|\boldsymbol{n_2}\|} \quad (5)$$

If θ exceeds a threshold of 10 degrees, the edge between these faces is classified as sharp, indicating a fragment boundary. The threshold of 10 degrees was determined iteratively by evaluating the training set, ensuring that the boundary edges were fully extracted without including edges inside the fragments. The sharp edges, where the dihedral angle exceeded the threshold, along with their corresponding triangle surfaces, were retained if at least five connected sharp edges were identified through connected component analysis. These connected sharp edges represented the contiguous boundaries of the fragments. The threshold of five connected edges was iteratively determined using the training set to

ensure that boundary edges were accurately extracted, while filtering out small, erroneously detected edges within the fragments. Once identified, these connected components, along with their corresponding triangles, were used to generate individual mesh files for each fragment, precisely capturing the detected boundaries. Excluding fragments with fewer than five connected edges may omit very small fragments, which is clinically appropriate, as these small fragments are unlikely to impact treatment decisions. From this boundary STL mesh, all points were extracted and used to generate surface meshes of each fragment through Delaunay triangulation. Delaunay triangulation ensures that no points lie inside the circumcircle of any triangle in the mesh, providing a stable and accurate surface representation. The surface area A of each fragment was computed by summing the areas of the individual triangles in the surface mesh. The area of a single triangle T_i with vertices $\overrightarrow{v_1}, \overrightarrow{v_2}$ and, $\overrightarrow{v_3}$ is given by:

$$A_i = \frac{1}{2} \| (\overrightarrow{v_2} - \overrightarrow{v_1}) \times (\overrightarrow{v_3} - \overrightarrow{v_1}) \| \quad (6)$$

The total surface area of the fragment is then:

$$A_{fragment} = \sum_{i=1}^{N} A_i \quad (7)$$

Where N is the number of triangles in the fragment's surface mesh. For each patient, the individual surface meshes were combined to form a unified surface mesh representing the entire PTC. The total surface area $A_{combined}$ of this unified mesh was calculated using the same approach as for individual fragments. The fracture area $A_{fracture}$ was determined by subtracting the sum of the areas of all fragments from the combined surface area:

$$A_{fracture} = A_{combined} - \sum_{i=1}^{M} A_{fragment_i} \quad (8)$$

Where M is the number of fragments. The magnitude of $A_{fracture}$ provided insight into the degree of fragment displacement or misalignment. A larger fracture area indicates a more significant translation of the fragments relative to their original positions, involving either vertical displacement, horizontal separation, or both. See Figure 7 for a visual representation of the measurements. The surface areas of the fragments and the fracture area were reported for each patient, providing quantitative measures of the fractures present. Refer to Appendix E for the Python script associated with Section 2.3.5, and to Appendix J for the visualization of an example patient from the training dataset for Section 2.3.5.

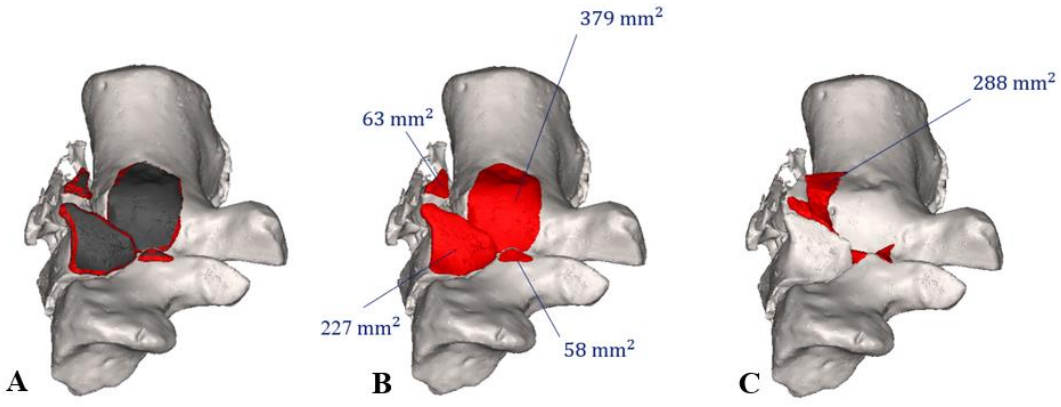


Figure 7: Visual representation of boundary extraction (a), fragment surface calculations (b), and fracture area calculations (c) for PTC fragments.

2.3.6 | Maximal Step-off and maximal gap analysis

To assess the maximal step-off and gap in the articular surface of the PTC, the distances were measured between vertically displaced fracture fragments for step-off and horizontally separated fragments for gap. The step-off and gap represent the discontinuity or misalignment in the joint surface caused by the vertical and horizontal displacement of fragments respectively.

The boundary points of each fragment were first extracted from the edge STL files generated during the fracture area calculations. These points, representing the fragment boundaries, were captured as point clouds.

A convex hull of the combined calcaneus fragments for each patient was generated and used as the reference plane for the articular surface. This convex hull was computed in two dimensions by projecting the point cloud of the boundaries of all calcaneus fragments onto a plane spanned by the first two principal components derived from PCA. The third principal component, representing the normal vector to the articular surface, was then used to measure the displacement of the fragments perpendicular to this reference plane.

Next, the boundary points of each fragment were projected perpendicularly onto the combined convex hull along the direction of the third principal component. For each pair of fragments, the closest points between the projected boundary points were identified based on their distances within the two-dimensional convex hull plane. Mutual closest-point pairs were confirmed by ensuring that each point in the pair was the closest to the other in both directions. Additionally, a check was performed to ensure no other projected points intersected the line connecting the mutually closest pair, as such an intersection would indicate that another fragment intercepted the line along the third principal component direction. See Figure 8A for a visual representation.

The maximal step-off was calculated as the maximum absolute value of the perpendicular distances between the original boundary points of the fragments and their corresponding projections onto the convex hull plane for the mutually closest point pairs. In contrast, the maximal gap was determined as the maximum distance between the projected boundary points of the mutually closest point pairs within the 2D convex hull (Figure 8B). (See Appendix F for the Python script corresponding to section 2.3.6)

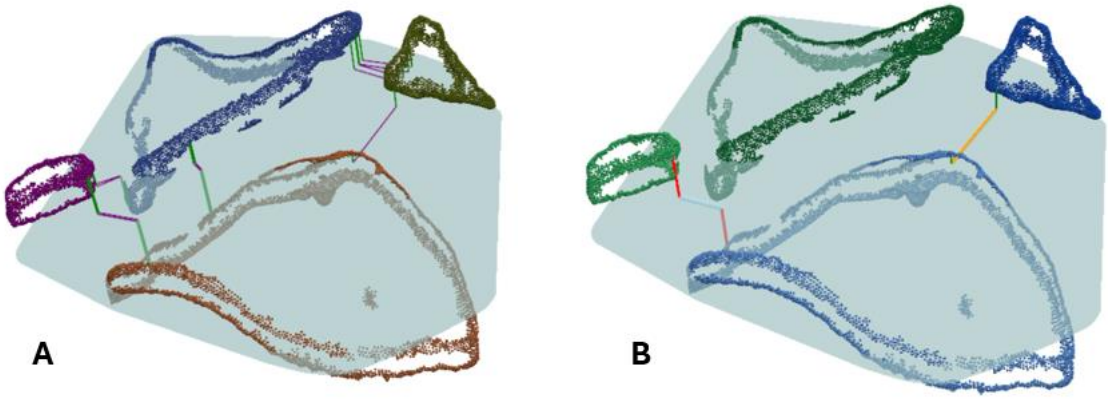


Figure 8: (A) The 2D convex hull of the combined fragments (light blue) with the boundary point clouds of each fragment. Mutual closest point pairs are shown with their projection lines (green) onto the convex hull. The purple lines represent distances between these projected pairs on the convex hull. (B) The red line indicates the maximal perpendicular distance between mutual closest points, representing the maximal step-off. The orange line shows the maximal distance between projected pairs on the 2D convex hull, representing the maximal gap.

2.4 | Manual 2D measurements

To compare automatic 3D measurement methods with current clinical practice, conventional 2D measurements were performed on each patient in the training dataset by two observers, an expert trauma surgeon and a general trauma surgeon (MV, VvW). A Graphical User Interface (GUI) was developed in MevisLab™ alongside a standardized measurement protocol to ensure consistent and uniform measurements across cases. See Appendix G for the MeVisLab tool.
The CT scans were first resliced so that the subtalar joint surface was parallel to the axial plane and perpendicular to the other planes. Using the resliced coronal images, the observers applied the Sanders classification system, categorizing fractures as type 0 (no intra-articular fracture on the PTC surface) or as types 1, 2, 3, or 4 (7). Observers then independently measured the maximal 2D gap (the distance between fracture fragments along the articular surface) and the maximal 2D step-off (the largest displacement perpendicular to the articular surface) on the coronal or sagittal plane, depending on where the measurements were most clearly identified.

3. | Experiments and Results

3.1 | Data

The dataset used for training and quantitative evaluation of the automatic segmentation framework, through 5-fold cross-validation, and for the development of the automatic 3D measurement method, included 44 patients: 8 from ErasmusMC and 36 from Maasstad Hospital. The patients had a mean age of 43.8 years, with 61% being male and 39% female, slice thickness ranged between 0.4 mm and 1 mm. 33 Patients were included in the anonymized external validation set; no patient demographics were available for this dataset due to anonymization procedures. The external dataset was divided into two groups: 10 patients with manual segmentations performed by one observer and 23 patients without manual segmentations. The slice thicknesses for the external validation set ranged from 0.2 mm to 1 mm. In cases where multiple filters or reconstructions were available per patient, the dataset used the reconstruction with the bone filter and the smallest slice thickness.

3.2 | Automatic segmentation

3.2.1 | Quantitative evaluation

The Dice similarity coefficient was used as a quantitative evaluation metric to assess the nnU-Net's ability to accurately segment the fractured PTC, PTF, and talus. The Dice coefficient measures the overlap between the predicted and ground truth segmentation masks. It provides a quantitative measure of segmentation accuracy by comparing how closely the predicted mask matches the actual fracture region. The Dice coefficient is calculated using the formula:

$$Dice = \frac{2|A \cap B|}{|A|+|B|} \quad (1)$$

Where A is the set of pixels in the predicted mask, B is the set of pixels in the ground truth mask, $|A \cap B|$ represents the intersection of the predicted and ground truth regions. The Dice coefficient ranges from 0 to 1, where 0 indicates no overlap between the predicted and ground truth segmentations, and 1 indicates perfect overlap, with higher values reflecting more accurate segmentation.
Table 1 presents the quantitative results of both trained nnU-Net models. The model trained for talus segmentation achieved the highest Dice score, with a value of 0.98 for both the training set after 5-fold cross-validation and the first group of the dataset used for external validation. The PTC fragments

segmentation achieved a Dice score of 0.78 for the training set during cross-validation and 0.75 for the external validation set. For the PTF segmentations, the Dice scores were 0.85 for the training set and 0.83 for the external validation set.

Table 1: Quantitative results of nnU-Net segmentations for PTC, PTF, and talus, including Dice similarity coefficients for the internal 5-fold cross-validation (ErasmusMC and Maasstad datasets) and external validation.

Type validation set		Dice		
		Avg	PTC	PTF
5-fold cross validation (44)	PTC & PTF	0.81	0.78	0.85
	Talus	0.98	-	-
External validation (10)	PTC & PTF	0.79	0.75	0.83
	Talus	0.98	-	-

Avg = average, PTC = posterior calcaneal facet, PTF = posterior talar facet

When examining the individual Dice scores for each patient, six cases in the training set and three cases in the external validation set had a Dice score below 0.7 for either the PTF or PTC segmentations.

3.2.2 | Qualitative evaluation

For qualitative assessment, two independent observers with expertise in CT segmentation and fracture assessment evaluated the segmentations independently by visual inspection using a five-point Likert scale. A score of 1 indicated strong disagreement, while 5 indicated strong agreement, with higher scores reflecting better performance. Each observer assessed the segmentations for each patient in the external validation set based on the following criteria: (1) the accuracy of the PTF segmentation, (2) the accuracy of the PTC segmentation, (3) the ability of the model to correctly distinguish fracture fragments, ensuring that segments meant to be separated by fracture lines, and (4) whether the generated segmentation was adequate for subsequent automatic 3D measurements. The full Likert scale questionnaire used for this assessment is provided in Appendix B.

Mean Likert scores and standard deviations were calculated for each aspect of the segmentation. Table 2 presents the qualitative evaluation results for 10 patients of the external validation set.

Table 2: Qualitative evaluation of nnU-Net segmentations for 10 patients of the external validation set, assessing PTF and PTC segmentation accuracy, fragment separation quality, and suitability for further analysis (Mean ± SD).

Evaluation criteria	**Mean ± SD**
PTF segmentation accuracy	4.9 ± 0.32
PTC segmentation accuracy	4.8 ± 0.42
Fragment separation quality	4.7 ± 0.48
Suitable for further analysis	5 ± 0

SD = standard deviation, PTC = posterior calcaneal facet, PTF = posterior talar facet

The remaining 23 patients in the external validation set underwent qualitative evaluation by visual inspection only. All segmentations were considered sufficient for further analysis. See Table 3 for details.

Table 3: Qualitative evaluation of nnU-Net segmentations for the 23 patients of the external validation set, assessing PTF and PTC segmentation accuracy, fragment separation quality, and suitability for further analysis (Mean ± SD).

Evaluation criteria	Mean ± SD
PTF segmentation accuracy	5 ± 0
PTC segmentation accuracy	4.94 ± 0.25
Fragment separation quality	4.61 ± 0.49
Suitable for further analysis	4.96 ± 0.21

SD = standard deviation, PTC = posterior calcaneal facet, PTF = posterior talar facet

3.3 | Automatic 3D measurements

Automatic 3D measurements were performed for all patients in the training dataset, and all patients in the external validation set. The distances, gap area, surface areas of each fragment, the combined surface area, the fracture area, the maximal step-off and maximal gap were compiled for each patient and saved to an Excel spreadsheet, providing a comprehensive overview of the surface areas and fracture characteristics.

Table 4 presents the results of the automatic 3D measurements for all patients in the training data. Refer to Appendix H for the inter-articular distance plots, and Appendix I for the gap area plots, of all patients from the training set and external validation set.

Table 4: Automatic 3D measurements of PTC fragments for all patients in the training dataset, including mean distance, gap area, number of fragments, total fragment area, fracture area, and maximal step-off (Mean, Median, Standard Deviation, Minimum, Maximum).

Variable	Mean	SD	Minimum	Maximum
Gap area (%)	16.05	15.03	0	55.26
Mean distance (mm)	3.59	1.28	1.89	6.67
SD of the mean distance	2.20	1.01	0.32	4.45
Number of fragments	2.32	0.98	1	5
Area of all fragments (mm^2)	1546.51	424.30	822.24	2660.93
Fracture area (mm^2)	174.04	159.45	0	568.79
Maximal step-off (mm)	3.98	3.67	0	15.34
Maximal gap (mm)	4.14	3.38	0	14.30

SD = standard deviation

For the 10 patients included in both the quantitative and qualitative analysis of the external validation set, 3D measurements were conducted on both the manually created and nnU-Net-generated segmentations. Figure 9 provides a patient-level comparison between the two segmentation methods, while the results of the mean statistics for each segmentation method are presented in Table 5.

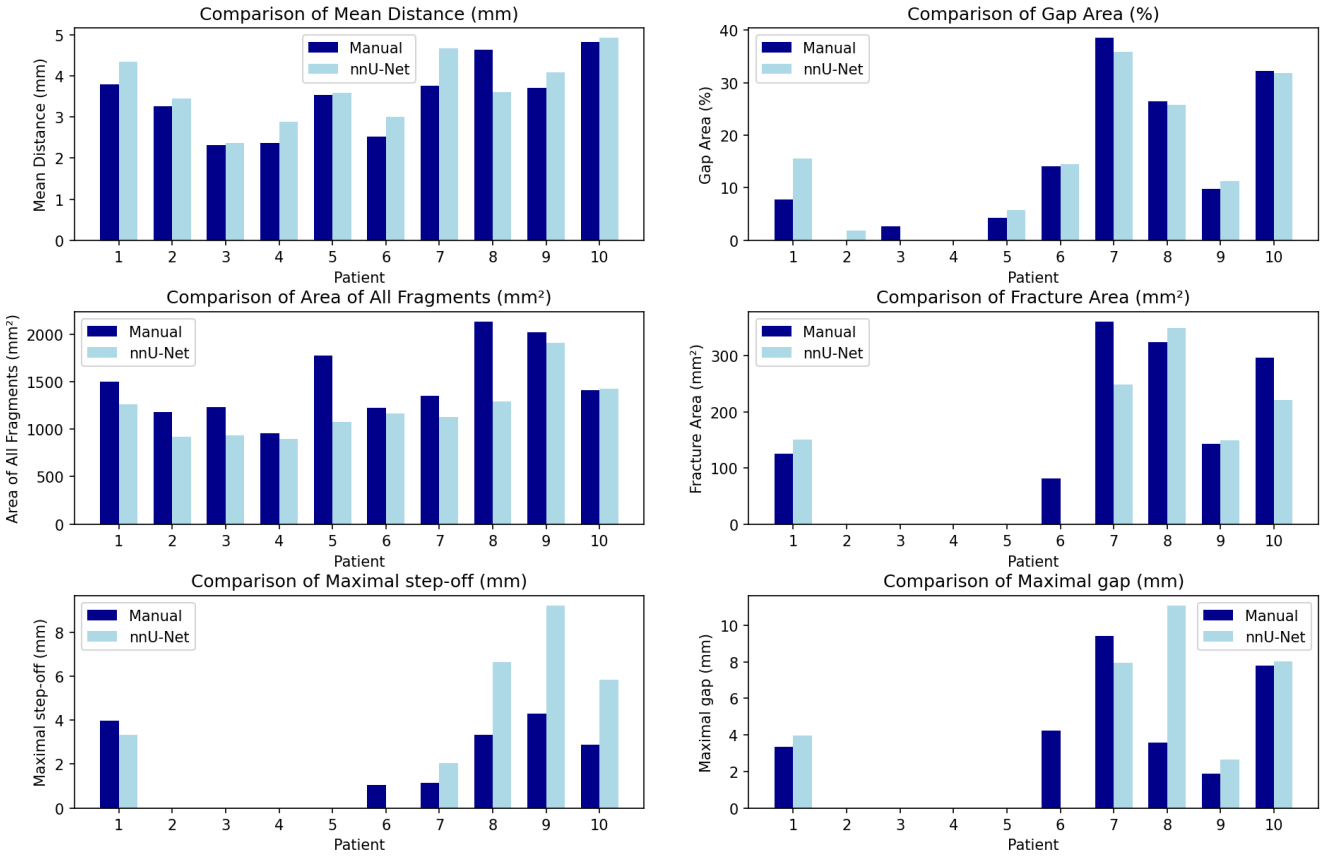


Figure 9: Patient level comparison of 10 patients in the external validation set where 3D measurements are based on manually segmented 3D models and 3D models based on nnU-Net segmentations.

Table 5: Full comparison of means and standard deviations for each 3D measurement of the 10 patients in the external validation set where both qualitative and quantitative evaluation was conducted.

Metric	Mean (manual)	SD (manual)	Mean (nnU-Net)	SD (nnU-Net)
Percentage of gap area (%)	14.28	13.46	14.22	13.13
Mean distance (mm)	3.83	1.45	3.69	0.82
SD of the mean distance	2.78	2.09	2.45	1.25
Number of fragments	2	1	2.2	1.32
Area of all fragments (mm^2)	2558.87	3607.33	1199.84	303.58
Fracture area (mm^2)	269.88	474.39	111.83	130.10
Maximal step-off (mm)	2.14	2.30	2.71	3.41
Maximal gap (mm)	5.26	8.07	3.37	4.21

SD = standard deviation

For the remaining 23 patients, 3D measurements based on the nnU-Net segmentations are shown in Table 6.

Table 6*:* Automatic 3D measurements on 3D models based on nnU-Net segmentations for 23 patients in the external validation dataset, including mean distance, gap area, number of fragments, total fragment area, fracture area, and maximal step-off (Mean, Median, Standard Deviation, Minimum, Maximum).

Variable	Mean	SD	Minimum	Maximum
Gap area (%)	14.81	11.67	0	39.77
Mean distance (mm)	3.59	1.12	1.77	6.34
SD of the mean distance	2.01	0.80	0.53	3.2
Number of fragments	1.96	0.71	1	3
Area of all fragments (mm^2)	1223.09	340.08	629.50	1736.18
Fracture area (mm^2)	103.62	82.60	0	228.77
Maximal step-off (mm)	3.54	3.03	0	11.22
Maximal gap (mm)	3.38	3.012	0	9.73

SD = standard deviation

All 3D measurements were visualized per patient to facilitate a thorough visual inspection. This process allowed for the assessment of the quality of the mesh processing steps, the representativity of the computed measurements, and the overall extent of the fracture.

The average total time required for the entire method, from automatic segmentation to obtaining the 3D measurement results, was 4 minutes and 37 seconds per patient. All steps were executed sequentially. The segmentation process was performed on a single GPU, utilizing 128 GB of RAM and 10 CPU cores per job. The subsequent 3D measurements and calculations were carried out using a single CPU.

3.4 | Manual 2D measurements

Manual 2D measurements were performed for all patients in the training set by two independent observers. To assess the interobserver agreement for the Sanders classification, Cohen's kappa was calculated, while the Interclass Correlation Coefficient (ICC) was used for the maximal gap and maximal step-off measurements. IBM SPSS Statistics for Windows, version 25 (IBM Corp., Armonk, NY, USA) was used for both analyses. A two-way mixed-effects model with absolute agreement for multiple raters was applied for the ICC measurements (25). The required sample size for assessing interobserver reliability was estimated using an online ICC hypothesis testing calculator, based on the method developed by Walter et al. (26,27). A minimum of 41 participants was determined to be necessary for each group. This calculation was made under the assumption of 2 observers, a significance level (alpha) of 0.05, a power of 80% (beta of 0.20), a minimum acceptable reliability of 0.4, an expected reliability of 0.7, and no anticipated drop-outs.

The Cohen's kappa for the Sanders classification was 0.26 (95% CI: 0.08-0.47), indicating fair agreement (28). The ICC (95% CI) between both observers for the maximal gap measurement and maximal step-off measurement resulted in 0.60 (0.07-0.82) and 0.59 (0.10-0.81), respectively, indicating moderate agreement between the two observers (25). An overview of the measurements can be found in Table 7.

Table 7: Results of manual 2D measurements for all 44 patients in the training dataset by two independent observers, including the Interclass Correlation Coefficient between the observer and the 95% confidence interval.

Variable	Mean (observer 1)	SD (observer 1)	Mean (observer 2)	SD (observer 2)	Agreement (95% CI)
Sanders classification	2.66	1.01	2.18	1.13	Cohen's kappa = 0.26 (0.08-0.47)
Maximal step-off (mm^2)	8.14	5.02	4.99	4.47	ICC = 0.60 (0.07-0.82)
Maximal gap (mm^2)	5.55	3.11	3.59	2.65	ICC = 0.59 (0.10-0.81)

SD=standard deviation, ICC = Interclass Correlation Coefficient

3.5 | Correlation calculations

Correlations between the average 2D manual measurements from both observers and the 3D automatic measurements for all patients in the training dataset were evaluated using Spearman's rank correlation for continuous variables and Kendall's Tau for ordinal variables, such as the Sanders classification. A p-value of <0.05 was considered statistically significant. The correlation analyses between 2D and 3D measurements were performed using Python libraries, including pandas and pingouin.

A moderate positive correlation (29) was found between the maximal 2D gap measurement and the 3D gap area (%) (Spearman's rho = 0.62, p<0.001) (Figure 10A). Also, moderate positive correlations were observed between the maximal 2D step-off measurement and the maximal 3D step-off measurement, as well as between the maximal 2D gap measurement and the maximal 3D gap measurement, with Spearman's rho values of 0.52 (p<0.001) and 0.65 (p<0.001), respectively (Figures 10B and 10C).

Since both 2D gap and step-off lengths may relate to overall fracture size, these measurements were correlated with the 3D fracture area. The correlation between the maximal 2D gap and the 3D fracture area resulted in a Spearman's rho of 0.65 (p<0.001). Likewise, the correlation between the maximal 2D step-off and the 3D fracture area was Spearman's rho = 0.64 (p<0.001), indicating moderate positive correlations between these variables (Figures 10D and 10E). The Sanders classification also demonstrated a moderate positive correlation with the number of fracture fragments (Kendall's Tau = 0.57, p<0.001), suggesting that higher Sanders classifications are associated with a greater number of fragments (Figure 10F).

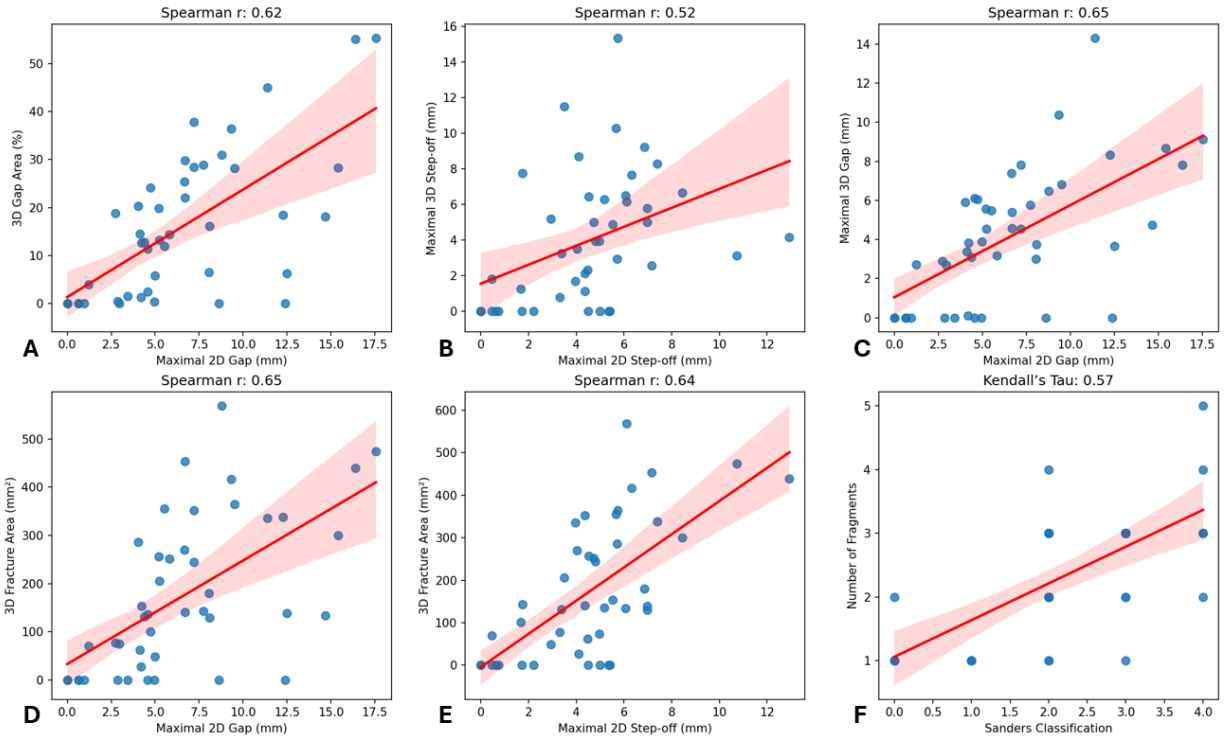


Figure 10: Correlation plots between manual 2D and automatic 3D measurements. (A) Maximal 2D gap measurement vs. 3D gap area. (B) Maximal 2D step-off vs. maximal 3D step-off. (C) Maximal 3D gap vs. maximal 2D gap. (D) Maximal 2D gap vs. 3D fracture area. (E) Maximal 2D step-off vs. 3D fracture area. (F) Sanders classification vs. number of fracture fragments. Red lines indicate linear regression, with pink shaded areas showing the 95% confidence intervals.

4. | Discussion

This study focuses on developing an automatic segmentation and 3D measurement method tailored for intra-articular calcaneus fractures, particularly targeting the PTC joint surface. The method uses deep learning, specifically the nnU-Net framework, to create accurate 3D segmentations of key anatomical structures from CT scans. These segmentations are then used to generate precise 3D measurements of fracture characteristics, such as gap area and step-off, providing a more comprehensive evaluation of calcaneal fractures compared to traditional 2D methods.

A key finding of the study is that moderate positive correlations were identified between manual 2D gap measurements and 3D gap area (Spearman's rho = 0.62), as well as between 2D and 3D step-off measurements (rho = 0.52). Furthermore, the Sanders classification correlated moderately with the number of fracture fragments found in 3D (Kendall's Tau = 0.57). These results suggest that the proposed automatic method aligns with current state-of-the-art approaches in terms of evaluating fracture characteristics and may serve as a robust and objective tool for assessing intra-articular calcaneus fractures.

The method enhances clinical assessment by providing structured, quantitative 3D measurements, which hold potential to improve treatment planning for complex intra-articular calcaneus fractures, offering a promising complement to existing manual techniques.

When assessing interobserver variability for the Sanders classification, the Cohen's kappa value of 0.26 found in this study was lower than those reported by Bhattacharya et al. (9) (Cohen's kappa = 0.32) and Humphrey et al. (10) (Cohen's kappa = 0.41). However, for the maximal 2D gap and step-off

measurements, the ICC for the gap (0.59) was comparable to the value reported by Roelofs et al.(13) for distal radius fractures (ICC = 0.54). In contrast, the ICC for the step-off (0.60) was notably higher than the value reported by Roelofs et al. for distal radius fractures (ICC = 0.21).
The mean combined area of the PTC fragments in this study is similar to the mean PTC area of 14.5 cm^2 reported by Qiang et al. (30). However, differences in sex, ethnicity, and population characteristics between the two studies should be considered when interpreting this comparison. For the other 3D measurements, we were unable to identify comparable analyses in the available literature.

In 6 cases from the training dataset and 3 cases from the first group of the external validation set, a Dice score below 0.7 was observed. However, upon visual inspection, the segmentations of these cases all received a Likert score of 5 for being sufficient for further analysis. The primary reason for the lower Dice scores was the presence of parts of the calcaneus and talus from the contralateral side in the CT images. While nnU-Net included these contralateral structures in its segmentation, the manual segmentation masks did not, leading to a discrepancy that negatively affected the Dice score.
Aside from such discrepancies, another important factor influencing Dice scores is the size of the segmentation mask. A larger overall mask tends to result in more overlap, leading to a higher Dice score. This explains why the Dice score for the entire talus segmentation is higher compared to that of the PTC and PTF surface segmentations. The higher Dice score for the full talus does not necessarily reflect better segmentation quality but rather the greater overlap due to the larger mask. Nevertheless, the Dice score for the surface segmentation remains relatively high, especially considering that it is calculated from small masks that are only 2 mm thick.

The automatic segmentation framework occasionally produced minor connections between PTC fragments, particularly in cases of non-displaced fractures, as observed during qualitative analysis. Although these connections are unlikely to have clinical implications, since non-displaced fractures are generally treated conservatively and do not influence treatment decisions (31), they can still affect metrics such as the number of fragments, area per fragment, fracture area, and maximal step-off. This occurs because non-displaced fractures, which should delineate separate fragments, may instead result in the merging of fragments, potentially obscuring larger fracture areas or significant step-offs elsewhere in the structure. Expanding the training dataset to include more cases of non-displaced fractures could enhance the model's ability to differentiate these fine details in future implementations.
Additionally, no definitive ground truth is available for parameters such as gap area, inter-articular distances, fragment and fracture area, or maximal step-off. These 3D measurements were compared with manual 2D measurements, where the observed ICC values ranged from 0.59 to 0.60, indicating moderate agreement between the two observers. Although the absence of an absolute standard is common in this area of research, the manual assessments still provide a valuable reference point for comparison. While they cannot be considered definitive, they offer important insights that help guide the evaluation of the 3D measurements.

A limitation of this study is that the ground truth used for model training was derived from manual segmentations performed by a single observer. These segmentations were based on CT scans in which the cartilage surfaces were not visible, making it difficult to accurately delineate the PTC area. As a result, the segmentation boundaries in regions involving cartilage may be uncertain, potentially impacting the accuracy of the model's predictions in these areas.

For future research, in addition to expanding the training dataset for nnU-Net to improve segmentation accuracy, the automatically calculated 3D measurements hold promise as decision-support tools in clinical treatment planning. To validate the utility of these parameters, a retrospective study should be

conducted, incorporating follow-up data and building a prediction model. This model could help determine whether 3D metrics such as gap size, inter-articular distances, maximal step-off, and fracture area provide meaningful insights for selecting optimal treatment options and improving outcomes in calcaneal fracture management.

5.| Conclusion

This study successfully demonstrates the development and validation of a complete method for the automatic segmentation and automatic 3D quantitative analysis of intra-articular calcaneus fractures in the PTC surface. The ability to automatically generate accurate 3D measurements within minutes has the potential to streamline clinical workflows and enhance decision-making in the management of calcaneus fractures.

References

1. Galluzzo M, Greco F, Pietragalla M, De Renzis A, Carbone M, Zappia M, et al. Calcaneal fractures: Radiological and CT evaluation and classification systems. Vol. 89, Acta Biomedica. Mattioli 1885; 2018. p. 138–50.
2. 77/ ACTA CHIRURGIAE ORTHOPAEDICAE ET TRAUMATOLOGIAE ČECHOSL CURREnT COnCEPTS REvIEw SOUbORný REfERáT.
3. Zhao Z, Li J. Calcaneus Fractures. In: Orthopaedic Trauma Surgery [Internet]. Singapore: Springer Nature Singapore; 2023. p. 397–432. Available from: https://link.springer.com/10.1007/978-981-16-0215-3_12
4. Weisman G, Araujo MGS. Calcaneus Fractures. In: Orthopaedics and Trauma [Internet]. Cham: Springer International Publishing; 2024. p. 613–21. Available from: https://link.springer.com/10.1007/978-3-031-30518-4_48
5. Dhillon MS, Bali K, Prabhakar S. Controversies in calcaneus fracture management: A systematic review of the literature. Vol. 95, Musculoskeletal Surgery. Springer-Verlag Italia s.r.l.; 2011. p. 171–81.
6. Wakker AM, Van Lieshout EMM, De Boer AS, Cornelissen BMW, Verhofstad MHJ, Van Walsum T, et al. A novel method to perform morphological measurements on three-dimensional (3D) models of the calcaneus based on computed tomography (CT)-imaging. Quant Imaging Med Surg. 2024 Jun 1;14(6):3778–88.
7. Jiménez-Almonte JH, King JD, Luo TD, Aneja A, Moghadamian E. Classifications in brief: Sanders classification of intraarticular fractures of the Calcaneus. Vol. 477, Clinical Orthopaedics and Related Research. Lippincott Williams and Wilkins; 2019. p. 467–71.
8. Leigheb M, Codori F, Samaila EM, Mazzotti A, Villafañe JH, Bosetti M, et al. Current Concepts about Calcaneal Fracture Management: A Review of Metanalysis and Systematic Reviews. Vol. 13, Applied Sciences (Switzerland). Multidisciplinary Digital Publishing Institute (MDPI); 2023.
9. Bhattacharya R, Vassan UT, Finn P, Port A. Sanders classification of fractures of the os calcis AN ANALYSIS OF INTER-AND INTRA-OBSERVER VARIABILITY. 2005;87(2):205.
10. Humphrey CA, Dirschl DR, Ellis TJ. Interobserver Reliability of a CT-Based Fracture Classification System [Internet]. Available from: http://journals.lww.com/jorthotrauma
11. Rahmaniar W, Wang WJ. Real-time automated segmentation and classification of calcaneal fractures in CT images. Applied Sciences (Switzerland). 2019 Aug 1;9(15).
12. Ogawa BK, Charlton TP, Thordarson DB. Radiography versus computed tomography for displacement assessment in calcaneal fractures. Foot Ankle Int. 2009 Oct;30(10):1005–10.
13. Roelofs LJM, Meesters AML, Assink N, Kraeima J, Van der Meulen TD, Doornberg JN, et al. A new quantitative 3D gap area measurement of fracture displacement of intra-articular distal radius fractures: Reliability and clinical applicability. PLoS One. 2022 Sep 1;17(9 September).
14. Griffin D, Parsons N, Shaw E, Kulikov Y, Hutchinson C, Thorogood M, et al. Operative versus non-operative treatment for closed, displaced, intra-articular fractures of the calcaneus: Randomised controlled trial. BMJ (Online). 2014 Jul 24;349.
15. Rodemund C, Krenn R, Kihm C, Leister I, Ortmaier R, Litzlbauer W, et al. Minimally invasive surgery for intra-articular calcaneus fractures: a 9-year, single-center, retrospective study of a standardized technique using a 2-point distractor. BMC Musculoskelet Disord. 2020 Dec 1;21(1).
16. Cha Y, Kim JT, Park CH, Kim JW, Lee SY, Yoo J Il. Artificial intelligence and machine learning on diagnosis and classification of hip fracture: systematic review. Vol. 17, Journal of Orthopaedic Surgery and Research. BioMed Central Ltd; 2022.
17. Kraus M, Anteby R, Konen E, Eshed I, Klang E. Artificial intelligence for X-ray scaphoid fracture detection: a systematic review and diagnostic test accuracy meta-analysis. European Radiology. Springer Science and Business Media Deutschland GmbH; 2023.
18. Langerhuizen DWG, Janssen SJ, Mallee WH, Van Den Bekerom MPJ, Ring D, Kerkhoffs GMMJ, et al. What Are the Applications and Limitations of Artificial Intelligence for Fracture Detection and

Classification in Orthopaedic Trauma Imaging? A Systematic Review. Vol. 477, Clinical Orthopaedics and Related Research. Lippincott Williams and Wilkins; 2019. p. 2482–91.
19. Lex JR, Di Michele J, Koucheki R, Pincus D, Whyne C, Ravi B. Artificial Intelligence for Hip Fracture Detection and Outcome Prediction: A Systematic Review and Meta-analysis. JAMA Netw Open. 2023 Mar 17;6(3):E233391.
20. Ashkani-Esfahani S, Mojahed Yazdi R, Bhimani R, Kerkhoffs GM, Maas M, DiGiovanni CW, et al. Detection of ankle fractures using deep learning algorithms. Foot and Ankle Surgery. 2022 Dec 1;28(8):1259–65.
21. Assink N, Kraeima J, Slump CH, ten Duis K, de Vries JPPM, Meesters AML, et al. Quantitative 3D measurements of tibial plateau fractures. Sci Rep. 2019 Dec 1;9(1).
22. Anatomy3DAtlas [Internet]. [cited 2024 Oct 21]. Available from: https://anatomy3datlas.com/
23. Ronneberger O, Fischer P, Brox T. U-Net: Convolutional Networks for Biomedical Image Segmentation. 2015 May 18; Available from: http://arxiv.org/abs/1505.04597
24. Wakker AM, Verhofstad MHJ, Visser JJ, Van Vledder MG, Van Walsum T. Talus-derived reference coordinate system for 3D calcaneal assessment: A novel approach to improve morphological measurements. Journal of Orthopaedic Research. 2024;
25. Koo TK, Li MY. A Guideline of Selecting and Reporting Intraclass Correlation Coefficients for Reliability Research. J Chiropr Med. 2016 Jun 1;15(2):155–63.
26. Arifin WN. Sample size calculator [Internet]. [cited 2024 Oct 4]. Available from: https://wnarifin.github.io/ssc/ssicc.html
27. Walter SD, Eliasziw M, Donner A, The JP. SAMPLE SIZE AND OPTIMAL DESIGNS FOR RELIABILITY STUDIES. Vol. 17, STATISTICS IN MEDICINE. 1998.
28. Sim J, Wright CC. Number 3 [Internet]. Vol. 85, Physical Therapy. 2005. Available from: https://academic.oup.com/ptj/article/85/3/257/2805022
29. Mukaka MM. Statistics Corner: A guide to appropriate use of Correlation coefficient in medical research [Internet]. Vol. 24, Malawi Medical Journal. 2012. Available from: www.mmj.medcol.mw
30. Qiang M, Chen Y, Zhang K, Li H, Dai H. Measurement of three-dimensional morphological characteristics of the calcaneus using CT image post-processing. J Foot Ankle Res. 2014 Mar 14;7(1).
31. Jiménez-Almonte JH, King JD, Luo TD, Aneja A, Moghadamian E. Classifications in brief: Sanders classification of intraarticular fractures of the Calcaneus. Vol. 477, Clinical Orthopaedics and Related Research. Lippincott Williams and Wilkins; 2019. p. 467–71.

Appendices

Appendix A: nnU-Net SLURM script

```
#!/bin/bash
#SBATCH --ntasks=10           ### How many CPU cores do you need?
#SBATCH --mem=128G            ### How much RAM memory do you need?
#SBATCH -p long               ### The queue to submit to: express, short, long, interactive
#SBATCH --gres=gpu:1          ### How many GPUs do you need? #
#SBATCH -t 2-00:00:00         ### The time limit in D-hh:mm:ss format
#SBATCH -o out_%j.log         ### Where to store the console output (%j is the job number)
#SBATCH -e error_%j.log       ### Where to store the error output
#SBATCH --job-name=PTC_pred   ### Name your job so you can distinguish between jobs

# Load the modules
module purge
module load Python/3.9.5-GCCcore-10.3.0
source /mnt/trtm0001/data/Alex/venvs/nnUnet_env/bin/activate


# To set up the correct environment for nnUNet, follow the following steps from the command line
before running the slurm script:
# module load Python/3.9.5-GCCcore-10.3.0
# python3 -m venv nnunet_env
# pip install --upgrade pip wheel
# pip install numpy
# pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu117
# pip install nnunetv2
# pip install --upgrade git+https://github.com/FabianIsensee/hiddenlayer.git
# pip install ipython

export nnUNet_preprocessed="/mnt/trtm0001/data/Alex/nnUnet/nnUNet_preprocessed"
export nnUNet_results="/mnt/trtm0001/data/Alex/nnUnet/nnUNet_results"
export nnUNet_raw="/mnt/trtm0001/data/Alex/nnUnet/nnUNet_raw"


## Follow the following steps to run the entire pipeline of nnU-Net:

#***See if all data is in correct order and have the correct names:***
 nnUNetv2_plan_and_preprocess -d 602 --verify_dataset_integrity

#**Train nnUNet:***
 nnUNetv2_train 602 3d_lowres 4 --npz

#***Run best configuration :***
 nnUNetv2_find_best_configuration 602 -c 3d_lowres # Output will tell what interference + post-
processing should be run

#***Run inference like this:***
 nnUNetv2_predict -d Dataset602_fractured_calcaneusPTC_talusPTC -i imagesTs -o
output_predictions -f  0 1 2 3 4 -tr nnUNetTrainer -c 3d_lowres -p nnUNetPlans

#***Once inference is completed, run postprocessing like this:***
nnUNetv2_apply_postprocessing -i output_predictions -o output_predictions_and_postprocessing -
pp_pkl_file
/mnt/trtm0001/data/Alex/nnUnet/nnUNet_results/Dataset602_fractured_calcaneusPTC_talusPTC/nnUNetT
rainer__nnUNetPlans__3d_lowres/crossval_results_folds_0_1_2_3_4/postprocessing.pkl -np 8 -
plans_json
/mnt/trtm0001/data/Alex/nnUnet/nnUNet_results/Dataset602_fractured_calcaneusPTC_talusPTC/nnUNetT
rainer__nnUNetPlans__3d_lowres/crossval_results_folds_0_1_2_3_4/plans.json

#***Evaluate the predictions from imagesTs compared to labelsTs:***
nnUNetv2_evaluate_folder -djfile
/mnt/trtm0001/data/Alex/nnUnet/nnUNet_raw/Dataset602_fractured_calcaneusPTC_talusPTC/dataset.jso
n -pfile
/mnt/trtm0001/data/Alex/nnUnet/nnUNet_raw/Dataset602_fractured_calcaneusPTC_talusPTC/output_pred
ictions/plans.json
/mnt/trtm0001/data/Alex/nnUnet/nnUNet_raw/Dataset602_fractured_calcaneusPTC_talusPTC/labelsTs
```

```
/mnt/trtm0001/data/Alex/nnUnet/nnUNet_raw/Dataset602_fractured_calcaneusPTC_talusPTC/output_predictions_and_postprocessing


echo 'The slurm script has ended'
```

Appendix B – Likert questionnaire

Q1. I found the segmentations of the PTC surface to be visually accurate and representative of the joint anatomy.

○ Strongly disagree	○ Disagree	○ Neither agree nor disagree	○ Agree	○ Strongly agree

Q2. I found the segmentations of the PTF surface to be visually accurate and representative of the joint anatomy.

○ Strongly disagree	○ Disagree	○ Neither agree nor disagree	○ Agree	○ Strongly agree

Q3. I found that the individual fragments identified during the segmentation process were correctly separated and represented the appropriate anatomical structures.

○ Strongly disagree	○ Disagree	○ Neither agree nor disagree	○ Agree	○ Strongly agree

Q4. I found the segmentations produced to be sufficient for further 3D analysis and measurement.

○ Strongly disagree	○ Disagree	○ Neither agree nor disagree	○ Agree	○ Strongly agree

Appendix C- STL creation and alignment with talus

```
import os
import re
import nibabel as nib
import numpy as np
from stl import mesh
from skimage import measure
import tkinter as tk
from tkinter import filedialog
import open3d as o3d
import copy
import trimesh
import pyvista as pv
import open3d as o3d
import copy
import matplotlib.pyplot as plt
from skimage import measure, morphology
import time
from scipy.ndimage import binary_closing
import networkx as nx
import scipy.sparse
from scipy.ndimage import binary_closing

# Define Laplacian smoothing function
def laplacian_smoothing(vertices, faces, iterations=10, lambda_=0.05):
    """Smooth the mesh using Laplacian smoothing."""
    mesh = trimesh.Trimesh(vertices=vertices, faces=faces)
    adjacency_graph = mesh.vertex_adjacency_graph
    adjacency_matrix = nx.to_scipy_sparse_array(adjacency_graph)  # Updated to use
to_scipy_sparse_array
    adjacency_matrix = scipy.sparse.csr_matrix(adjacency_matrix)  # Ensure it's a sparse
matrix
    laplacian = scipy.sparse.csgraph.laplacian(adjacency_matrix, normed=False)
    identity = scipy.sparse.identity(laplacian.shape[0])
    smoothing_matrix = identity - lambda_ * laplacian

    smoothed_vertices = vertices.copy()
    for _ in range(iterations):
        smoothed_vertices = smoothing_matrix.dot(smoothed_vertices)
        displacement = smoothed_vertices - vertices
        max_displacement = np.percentile(np.linalg.norm(displacement, axis=1), 95)
        displacement = np.clip(displacement, -max_displacement, max_displacement)
        smoothed_vertices = vertices + displacement

    return smoothed_vertices

# Function to check if normals are correctly oriented
def check_normals(mesh_obj):
    face_normals = mesh_obj.face_normals
    face_centroids = mesh_obj.triangles_center
    mesh_centroid = mesh_obj.centroid
    vectors_from_centroid = face_centroids - mesh_centroid
```

```
    dot_products = (face_normals * vectors_from_centroid).sum(axis=1)
    if (dot_products < 0).mean() > 0.5:
        print("Normals are pointing inward, inverting normals...")
        return False
    else:
        print("Normals are correctly pointing outward.")
        return True

# Function to save STL with smoothing
def save_stl(mask, output_path, affine, apply_closing=False, apply_smoothing=True,
iterations=10, lambda_=0.05):
    if np.any(mask):
        if apply_closing:
            mask = binary_closing(mask, structure=np.ones((3, 3, 3)))

        # Marching cubes to extract surface
        verts, faces, _, _ = measure.marching_cubes(mask, level=0.5)

        # Apply the affine transformation to the vertices
        verts_homogeneous = np.hstack([verts, np.ones((verts.shape[0], 1))])
        verts_transformed = verts_homogeneous.dot(affine.T)[:, :3]

        # Optionally apply Laplacian smoothing to the vertices
        if apply_smoothing:
            verts_transformed = laplacian_smoothing(verts_transformed, faces,
iterations=iterations, lambda_=lambda_)

        # Create the STL mesh
        mesh_data = mesh.Mesh(np.zeros(faces.shape[0], dtype=mesh.Mesh.dtype))
        for i, f in enumerate(faces):
            for j in range(3):
                mesh_data.vectors[i][j] = verts_transformed[f[j], :]

        # Save the STL file
        mesh_data.save(output_path)
        print(f"Saved STL file: {output_path}")

        # Load mesh using trimesh and check normals
        mesh_obj = trimesh.load(output_path)
        if not check_normals(mesh_obj):
            # If normals are inward, invert and save
            mesh_obj.invert()
            mesh_obj.export(output_path)
            print(f"Normals inverted and saved for {output_path}")

# Function to process NIFTI to STL with smoothing
def nifti_to_stl(nifti_file, base_output_dir, label_name, subdir):
    print(f"Loading NIFTI file: {nifti_file}")
    img = nib.load(nifti_file)
    data = img.get_fdata()
    affine = img.affine

    # Extract numeric part and side indicator from file name
```

```
    file_name = os.path.basename(nifti_file)
    match = re.search(r'(\d+)([RL])', file_name)
    if match:
        numeric_part = match.group(1).zfill(3)  # Format the numeric part
        side_indicator = match.group(2)
    else:
        print(f"No side indicator found in file name: {file_name}")
        return

    # Create output directories based on the formatted numeric part and side indicator
    patient_dir = os.path.join(base_output_dir, f"{numeric_part}{side_indicator}")
    output_dir = os.path.join(patient_dir, subdir)
    os.makedirs(output_dir, exist_ok=True)

    if subdir == 'talus_PTC':
        mask = (data == label_name)
        stl_path = os.path.join(output_dir,
f'{numeric_part}{side_indicator}_talus_PTC.stl')
        save_stl(mask, stl_path, affine, apply_closing=False, apply_smoothing=True)  # No
closing for talus_PTC

    elif subdir == 'talus':
        mask = (data == label_name)
        stl_path = os.path.join(output_dir, f'{numeric_part}{side_indicator}_talus.stl')
        save_stl(mask, stl_path, affine, apply_closing=True, apply_smoothing=True)  #
Apply closing for talus

    elif subdir == 'calcaneus_PTC_fragments':
        calcaneus_mask = (data == label_name)
        labeled_calcaneus, num_labels = measure.label(calcaneus_mask, return_num=True,
connectivity=3)
        print(f"Found {num_labels} fragments in calcaneus.")

        for label in range(1, num_labels + 1):
            fragment_mask = (labeled_calcaneus == label)
            calcaneus_stl_path = os.path.join(output_dir,
f'{numeric_part}{side_indicator}_calcaneus_{label}.stl')
            save_stl(fragment_mask, calcaneus_stl_path, affine, apply_closing=False,
apply_smoothing=True)


def select_folders(title):
    root = tk.Tk()
    root.withdraw()  # Hide the main window
    folder = filedialog.askdirectory(title=title)
    if not folder:
        print(f"No folder selected for {title}.")
        return None
    return folder

# Folder selection for NIFTI to STL processing
nifti_folder_fragments = select_folders("Select input folder where the NIFTI masks of the
PTC fragments are stored")
```

```
nifti_folder_talus = select_folders("Select input folder where the NIFTI masks of the
complete talus are stored")
base_output_dir = select_folders("Select Output Folder for the STL files")

# Process the NIFTI files into STL with optional Laplacian smoothing
if nifti_folder_fragments and nifti_folder_talus and base_output_dir:
    # Process all NIFTI files for talus_PTC
    for nifti_file in os.listdir(nifti_folder_fragments):
        if nifti_file.endswith('.nii') or nifti_file.endswith('.nii.gz'):
            nifti_path = os.path.join(nifti_folder_fragments, nifti_file)
            nifti_to_stl(nifti_path, base_output_dir, 1, 'talus_PTC')
            print(f"Processing of {nifti_file} from NIFTI to STL (talus_PTC) is
complete.")

    # Process all NIFTI files for talus
    for nifti_file in os.listdir(nifti_folder_talus):
        if nifti_file.endswith('.nii') or nifti_file.endswith('.nii.gz'):
            nifti_path = os.path.join(nifti_folder_talus, nifti_file)
            nifti_to_stl(nifti_path, base_output_dir, 1, 'talus')
            print(f"Processing of {nifti_file} from NIFTI to STL (talus) is complete.")

    # Process all NIFTI files for calcaneus_PTC_fragments
    for nifti_file in os.listdir(nifti_folder_fragments):
        if nifti_file.endswith('.nii') or nifti_file.endswith('.nii.gz'):
            nifti_path = os.path.join(nifti_folder_fragments, nifti_file)
            nifti_to_stl(nifti_path, base_output_dir, 2, 'calcaneus_PTC_fragments')
            print(f"Processing of {nifti_file} from NIFTI to STL (calcaneus_PTC_fragments)
is complete.")
else:
    print("Folder selection was not completed.")



######  Align_parts_with_talus #######

def calculate_mass_properties(root, file):
    your_mesh = trimesh.load_mesh(os.path.join(root, file))
    mass_properties = your_mesh.mass_properties
    volume = mass_properties['volume']
    cog = mass_properties['center_mass']
    inertia = mass_properties['inertia']
    return volume, cog, inertia

## Comment from here if the alignemtn is already done
def draw_registration_result_PTC_joint(moving, fixed, talus_PTC, calc_PTCs,
transformation):
    moving_temp = copy.deepcopy(moving)
    fixed_temp = copy.deepcopy(fixed)
    talus_PTC_temp = copy.deepcopy(talus_PTC)
    calc_PTCs_temp = [copy.deepcopy(calc) for calc in calc_PTCs]

    moving_temp.paint_uniform_color([1, 0.706, 0])  # Yellow for moving
    fixed_temp.paint_uniform_color([0, 0.651, 0.929])  # Blue for fixed
```

```
    talus_PTC_temp.paint_uniform_color([0, 1, 0])  # Green for talus_PTC

    # Generate a unique color for each calcaneus fragment using HSV colormap
    n = len(calc_PTCs_temp)
    colors = plt.cm.get_cmap("hsv", n)(np.arange(n))[:, :3]  # Ensuring unique colors

    for calc_temp, color in zip(calc_PTCs_temp, colors):
        calc_temp.paint_uniform_color(color.tolist())  # Apply color
        calc_temp.transform(transformation)  # Apply transformation

    moving_temp.transform(transformation)
    talus_PTC_temp.transform(transformation)

    o3d.visualization.draw_geometries([moving_temp, fixed_temp, talus_PTC_temp] +
calc_PTCs_temp)
    return [moving_temp, talus_PTC_temp] + calc_PTCs_temp


def align_calc_based_on_talus_PTC_joint(filename_talus, filename_talus_PTC,
filenames_calc_PTC,
                                        root_folder_talus, root_folder_talus_PTC,
root_folder_calc_PTC,
                                        fixed_talus_mesh, fixed_talus_cog):
    # Start time tracking (processing only, excluding visualization)
    start_time = time.time()
    moving_mesh_talus = o3d.io.read_triangle_mesh(os.path.join(root_folder_talus,
filename_talus))
    moving_mesh_talus_PTC = o3d.io.read_triangle_mesh(os.path.join(root_folder_talus_PTC,
filename_talus_PTC))
    moving_mesh_calc_PTCs = [o3d.io.read_triangle_mesh(os.path.join(root_folder_calc_PTC,
filename))
                             for filename in filenames_calc_PTC]

    volume_moving, cog_moving, inertia_moving =
calculate_mass_properties(root_folder_talus, filename_talus)
    difference = fixed_talus_cog - cog_moving

    moving_mesh_talus.translate(difference)
    moving_mesh_talus_PTC.translate(difference)
    for moving_mesh_calc_PTC in moving_mesh_calc_PTCs:
        moving_mesh_calc_PTC.translate(difference)

    fixed_talus = fixed_talus_mesh.sample_points_uniformly(20000)
    moving_talus = moving_mesh_talus.sample_points_uniformly(20000)

    voxel_size = 1
    moving_down, moving_fpfh = preprocess_point_cloud(moving_talus, voxel_size)
    fixed_down, fixed_fpfh = preprocess_point_cloud(fixed_talus, voxel_size)

    result_ransac = execute_global_registration(moving_down, fixed_down, moving_fpfh,
fixed_fpfh, voxel_size)
    result_icp = refine_registration(moving_talus, fixed_talus, moving_fpfh, fixed_fpfh,
voxel_size, result_ransac)
```

```
    # Stop the time tracking before visualization
    end_time = time.time()

    # Calculate and print the total elapsed time (processing only, excluding
visualization)
    elapsed_time = end_time - start_time
    print(f"Alignment processing time (excluding visualization): {elapsed_time:.2f}
seconds.")

    all_new_locations = draw_registration_result_PTC_joint(
        moving_mesh_talus,
        fixed_talus_mesh,
        moving_mesh_talus_PTC,
        moving_mesh_calc_PTCs,
        result_icp.transformation)

    poisson_meshes = [o3d.geometry.TriangleMesh.compute_triangle_normals(mesh) for mesh in
all_new_locations]
    o3d.visualization.draw_geometries(all_new_locations)

    return poisson_meshes

def preprocess_point_cloud(pcd, voxel_size):
    pcd_down = pcd.voxel_down_sample(voxel_size)
    radius_normal = voxel_size * 2
    pcd_down.estimate_normals(o3d.geometry.KDTreeSearchParamHybrid(radius=radius_normal,
max_nn=30))
    radius_feature = voxel_size * 50
    pcd_fpfh = o3d.pipelines.registration.compute_fpfh_feature(
        pcd_down, o3d.geometry.KDTreeSearchParamHybrid(radius=radius_feature, max_nn=100))
    return pcd_down, pcd_fpfh

def execute_global_registration(moving_down, fixed_down, moving_fpfh, fixed_fpfh,
voxel_size):
    distance_threshold = voxel_size * 20000
    result = o3d.pipelines.registration.registration_ransac_based_on_feature_matching(
        moving_down, fixed_down, moving_fpfh, fixed_fpfh, True,
        distance_threshold,
        o3d.pipelines.registration.TransformationEstimationPointToPoint(False),
        3, [
            o3d.pipelines.registration.CorrespondenceCheckerBasedOnEdgeLength(0.9),
            o3d.pipelines.registration.CorrespondenceCheckerBasedOnDistance(distance_thres
hold)
        ], o3d.pipelines.registration.RANSACConvergenceCriteria(1000000, 50000))
    return result

def refine_registration(moving, fixed, moving_fpfh, fixed_fpfh, voxel_size,
result_ransac):
    distance_threshold = voxel_size * 5000
    result = o3d.pipelines.registration.registration_icp(
        moving, fixed, distance_threshold, result_ransac.transformation,
        o3d.pipelines.registration.TransformationEstimationPointToPoint())
```

```
    print(result)
    return result

# Stop the commenting here to remain with the output folders
# Select template STL folders and files
root_folder_fixed_talus_left = select_folders("Select the folder where the left foot talus
template STL file is stored")
filename_fixed_talus_left = filedialog.askopenfilename(title="Select the left foot talus
template STL file", filetypes=[("STL files", "*.stl")])

root_folder_fixed_talus_right = select_folders("Select the folder where the right foot
talus template STL file is stored")
filename_fixed_talus_right = filedialog.askopenfilename(title="Select the right foot talus
template STL file", filetypes=[("STL files", "*.stl")])

# Load the left and right talus template meshes
fixed_talus_mesh_left = o3d.io.read_triangle_mesh(filename_fixed_talus_left)
volume_fixed_left, cog_fixed_left, inertia_fixed_left =
calculate_mass_properties(root_folder_fixed_talus_left,
os.path.basename(filename_fixed_talus_left))

fixed_talus_mesh_right = o3d.io.read_triangle_mesh(filename_fixed_talus_right)
volume_fixed_right, cog_fixed_right, inertia_fixed_right =
calculate_mass_properties(root_folder_fixed_talus_right,
os.path.basename(filename_fixed_talus_right))

# Loop over all patient directories within the selected base output directory
for patient_dir in os.listdir(base_output_dir):
    patient_path = os.path.join(base_output_dir, patient_dir)
    if os.path.isdir(patient_path):
        # Determine side from the patient directory name
        match = re.search(r'([RL])$', patient_dir)
        if match:
            side_indicator = match.group(1)
        else:
            print(f"No valid side indicator found in patient directory name:
{patient_dir}")
            continue

        # Select the appropriate template based on the side indicator
        if side_indicator == 'L':
            fixed_talus_mesh = fixed_talus_mesh_left
            fixed_talus_cog = cog_fixed_left
        elif side_indicator == 'R':
            fixed_talus_mesh = fixed_talus_mesh_right
            fixed_talus_cog = cog_fixed_right
        else:
            print(f"Unexpected issue with side indicator extraction for directory:
{patient_dir}")
            continue

        input_root_talus_PTC = os.path.join(patient_path, 'talus_PTC')
        input_root_talus = os.path.join(patient_path, 'talus')
```

```
        input_root_calc_PTC = os.path.join(patient_path, 'calcaneus_PTC_fragments')

        # Define new output folders for aligned files
        output_root_talus_PTC = os.path.join(patient_path, 'Aligned_with_talus',
'talus_PTC')
        output_root_talus = os.path.join(patient_path, 'Aligned_with_talus', 'talus')
        output_root_calc_PTC = os.path.join(patient_path, 'Aligned_with_talus',
'calcaneus')

        os.makedirs(output_root_talus_PTC, exist_ok=True)
        os.makedirs(output_root_talus, exist_ok=True)
        os.makedirs(output_root_calc_PTC, exist_ok=True)

        # Process and align STL files for the current patient directory
        for filename_talus, filename_talus_PTC in zip(os.listdir(input_root_talus),
os.listdir(input_root_talus_PTC)):
            filenames_calc_PTC = os.listdir(input_root_calc_PTC)

            new_locations = align_calc_based_on_talus_PTC_joint(
                filename_talus, filename_talus_PTC, filenames_calc_PTC,
                input_root_talus, input_root_talus_PTC, input_root_calc_PTC,
                fixed_talus_mesh, fixed_talus_cog)

            # Save individual aligned parts
            o3d.io.write_triangle_mesh(os.path.join(output_root_talus, filename_talus),
new_locations[0])
            o3d.io.write_triangle_mesh(os.path.join(output_root_talus_PTC,
filename_talus_PTC), new_locations[1])

            for idx, (filename, mesh) in enumerate(zip(filenames_calc_PTC,
new_locations[2:])):
                o3d.io.write_triangle_mesh(os.path.join(output_root_calc_PTC, filename),
mesh)

print("Alignment and saving of STL files completed.")
```

Appendix D – Calculations gap area and inter-articular distances

```
import os
import pyvista as pv
import numpy as np
import csv
from scipy.spatial import cKDTree
from scipy.stats import shapiro
from scipy.spatial import ConvexHull, Delaunay
from scipy.interpolate import interp1d
from sklearn.decomposition import PCA
import trimesh
import re
from tkinter import filedialog, Tk
from matplotlib.colors import LinearSegmentedColormap
import time
from scipy.interpolate import griddata


# Define function to select file and directories
def select_file(title):
    root = Tk()
    root.withdraw()  # Hide the main window
    file_path = filedialog.askopenfilename(title=title)
    return file_path

def select_directory(title):
    root = Tk()
    root.withdraw()  # Hide the main window
    folder_path = filedialog.askdirectory(title=title)
    return folder_path

# Select template files and base output directory
input_talus_template_left = select_file("Select the left talus template STL file")
input_talus_template_right = select_file("Select the right talus template STL file")
input_calcaneus_template_left = select_file("Select the left calcaneus template STL file")
input_calcaneus_template_right = select_file("Select the right calcaneus template STL
file")
base_output_dir = select_directory("Select the base output directory where the output of
the alignments are stored e.g. D:\Afstuderen Technical Medicine\data\STL")

# Start timing after file selection
start_time = time.time()  # Record the start time

# Define base name for combined STL outputs
output_file = 'combined_calcaneus'

combined_distances = []
percentage_holes = []
filenames = []
std_devs = []
results = []
```

```
def filter_normals_by_angle(mesh, direction_vector, angle_threshold=50):
    """Filter normals of a mesh by an angle threshold with a direction vector."""
    # Compute cosine of the threshold angle
    cos_threshold = np.cos(np.radians(angle_threshold))

    # Normalize the direction vector
    norm = np.linalg.norm(direction_vector)
    if norm == 0:
        return [], [], []  # Handle case where the direction vector is zero
    direction_norm = direction_vector / norm

    # Get the normals from the mesh
    normals = mesh.point_normals
    points = mesh.points

    # Calculate the dot products of the normals with the direction vector
    dot_products = np.dot(normals, direction_norm)

    # Find normals within the specified angle threshold
    indices = np.where(dot_products <= cos_threshold)[0]

    # Extract the corresponding normals
    filtered_normals = normals[indices]
    filtered_points = points[indices]

    return filtered_normals, indices, filtered_points

def create_glyphs_from_normals(mesh, indices, normals):
    """Create glyphs from the normals of a mesh."""
    # Extract points corresponding to the normals
    points = mesh.points[indices]

    # Create a PolyData object from these points and normals
    point_cloud = pv.PolyData(points)
    point_cloud['normals'] = normals

    # Create glyphs from the normals
    arrows = point_cloud.glyph(orient='normals', scale=True, factor=1)
    return arrows

def distance_PTC_joint(normal_array, normals_points, mesh_tal):
    """
    Compute distances between points on a surface (represented by normals) and another
mesh
    (represented by mesh_tal) along given directions (normal_array).
    """
    # Lists to store results
    intersection = []
    distances = []
    new_points = []
    original_points = []
    original_vectors = []
```

```
    for point, vector in zip(normals_points, normal_array):
        scalar = 20000
        new_point = point + scalar * vector

        vector_flipped = -1 * vector
        new_point_flipped = point + scalar * vector_flipped

        # Trace the ray from the point along the direction of vector to see if it
intersects with mesh_tal
        result, ind = mesh_tal.ray_trace(point, new_point)
        result_flipped, ind_flipped = mesh_tal.ray_trace(point, new_point_flipped)

        # Check if result or result_flipped is empty by evaluating their length
        if len(result) == 0 and len(result_flipped) == 0:
            # No intersection found for both
            intersection.append(0)
        else:
            # Intersection found
            intersection.append(1)
            # Calculate distance for non-empty result
            distance = np.linalg.norm(point - result[0]) if len(result) > 0 else
float('inf')
            # Calculate distance for non-empty result_flipped
            distance_flipped = np.linalg.norm(point - result_flipped[0]) if
len(result_flipped) > 0 else float('inf')
            # Choose the smaller distance
            original_points.append(point)
            original_vectors.append(vector)
            if distance_flipped < distance:
                distances.append(distance_flipped)
                new_points.append(result_flipped[0])
            elif distance != float('inf'):
                distances.append(distance)
                new_points.append(result[0])

    return distances, new_points, original_points, original_vectors

def distance_PTC_joint_convex_hull(normal_array, normals_points, mesh_hull,
convex_hull_mesh):
    """
    Compute distances between points on a surface (convex_hull_PTC, so in the middle of
the PTC joint) and another mesh
    (represented by mesh_tal) along given directions (normal_array). Additionally, check
for
    intersections with a convex hull mesh, distal from the PTC joint.
    """
    # Lists to store results
    distances = []
    new_points = []
    original_points = []
    original_vectors = []
    hull_intersections = []
```

```
    for point, vector in zip(normals_points, normal_array):
        scalar = 20000
        new_point = point + scalar * vector

        vector_flipped = -1 * vector
        new_point_flipped = point + scalar * vector_flipped

        # Trace the ray from the point along the direction of vector to see if it
intersects with mesh_tal
        result, ind = mesh_hull.ray_trace(point, new_point)
        result_flipped, ind_flipped = mesh_hull.ray_trace(point, new_point_flipped)

        # Trace the ray from the point along the direction of vector to see if it
intersects with convex_hull_mesh
        hull_result, hull_ind = convex_hull_mesh.ray_trace(point, new_point)
        hull_result_flipped, hull_ind_flipped = convex_hull_mesh.ray_trace(point,
new_point_flipped)

        # Check if result or result_flipped is empty by evaluating their length
        if len(result) == 0 and len(result_flipped) == 0:
            # Check if the ray intersects with the convex hull mesh
            if len(hull_result) > 0:
                hull_intersections.append(hull_result[0])
            elif len(hull_result_flipped) > 0:
                hull_intersections.append(hull_result_flipped[0])
        else:
            # Intersection found
            original_points.append(point)
            original_vectors.append(vector)
            if len(result) > 0:
                distance = np.linalg.norm(point - result[0])
                distances.append(distance)
                new_points.append(result[0])
            elif len(result_flipped) > 0:
                distance_flipped = np.linalg.norm(point - result_flipped[0])
                distances.append(distance_flipped)
                new_points.append(result_flipped[0])

    return distances, new_points, original_points, original_vectors, hull_intersections

def combine_stl_files(input_directory, output_directory, base_output_file, patient_number,
side_indicator):
    os.makedirs(output_directory, exist_ok=True)
    stl_files = [f for f in os.listdir(input_directory) if f.endswith('.stl')]

    # List to store all meshes for the current patient
    patient_meshes = []

    for stl_file in stl_files:
        mesh = trimesh.load_mesh(os.path.join(input_directory, stl_file))
        patient_meshes.append(mesh)

    if patient_meshes:
```

```
        combined_mesh = trimesh.util.concatenate(patient_meshes)
        combined_output_file =
f"{base_output_file}_patient{patient_number}_{side_indicator}.stl"
        output_path = os.path.join(output_directory, combined_output_file)
        combined_mesh.export(output_path)
        print(f"Combined STL file for patient {patient_number} ({side_indicator}) saved as
{output_path}")

# Function to load and decimate the mesh
def load_and_decimate_mesh(filepath, point_threshold, target_reduction=0.7):
    """
    Load a mesh and decimate it if the number of points exceeds the threshold.

    Parameters:
    - filepath: str, path to the mesh file
    - point_threshold: int, threshold for the number of points
    - reduction_factor: float, fraction to reduce the mesh by (0 < reduction_factor < 1)

    Returns:
    - mesh: PyVista mesh object, possibly decimated
    """
    mesh = pv.read(filepath)
    num_points = mesh.n_points
    if num_points > point_threshold:
        mesh = mesh.decimate(target_reduction=target_reduction)
    return mesh

# Function to compute the center of a point cloud
def compute_center_of_point_cloud(cloud_data, com_talus_ptc, com_calcaneus_ptc,
scale_factor=0.8, translation_factor=3.0):
    """
    Determines the center of 3D points in 3D space using PCA and convex hull methods.
    Shrinks the convex hull by moving points closer to the centroid and translates
    the hull in the direction of the 3rd principal component (longitudinal axis).
    """
    # Compute the mean of the point cloud
    cloud_mean = np.mean(cloud_data, axis=0)

    # Calculate the direction vector from cloud mean to com_calcaneus
    direction_vector = com_calcaneus - cloud_mean
    direction_vector /= np.linalg.norm(direction_vector)  # Normalize the direction vector

    # Perform PCA to get principal components
    _, _, principal_components = np.linalg.svd(cloud_data - cloud_mean)

    # Ensure the 3rd principal component points in the positive X or Y direction
    if principal_components[2][1] < 0:  # Check the Y component if X is zero
        principal_components[2] = -principal_components[2]

    # Use the corrected 3rd principal component for translation
    translation_vector = translation_factor * principal_components[2]
    translation_vector_ptc = 15 * principal_components[2]
```

```
    # Project points onto plane defined by the first two principal components
    projected_points_2d = np.dot(cloud_data - cloud_mean, principal_components[:2].T)

    # Add small noise to avoid degenerate geometry
    noise = np.random.normal(scale=1e-6, size=projected_points_2d.shape)
    projected_points_2d += noise

    # Compute convex hull in 2D space
    hull = ConvexHull(projected_points_2d)
    hull_points = projected_points_2d[hull.vertices]

    # Compute the centroid of the convex hull
    hull_centroid = np.mean(hull_points, axis=0)

    # Shrink the convex hull points towards the centroid
    shrunk_hull_points = hull_centroid + scale_factor * (hull_points - hull_centroid)

    spacing = 0.5  # Adjust spacing as needed
    filled_points_2d = generate_points_in_hull(shrunk_hull_points, spacing)

    # Project the shrunk hull points back to 3D space
    shrunk_hull_points_3d = shrunk_hull_points @ principal_components[:2] + cloud_mean
    shrunk_hull_points_3d_ptc_joint = filled_points_2d @ principal_components[:2] +
cloud_mean

    # Translate the convex hull points in the direction of the direction vector
    translated_hull_points_3d = shrunk_hull_points_3d + translation_vector
    translated_hull_points_3d_ptc_joint = shrunk_hull_points_3d_ptc_joint -
translation_vector

    # Close the convex hull loop
    closed_hull_points = np.vstack([translated_hull_points_3d,
translated_hull_points_3d[0, :]])
    closed_hull_points_ptc_joint = np.vstack([translated_hull_points_3d_ptc_joint,
translated_hull_points_3d_ptc_joint[0, :]])

    # Calculate cumulative distance along the hull edges
    cumulative_distances = np.cumsum(np.sqrt(np.sum(np.diff(closed_hull_points, axis=0) **
2, axis=1)))
    cumulative_distances = np.insert(cumulative_distances, 0, 0)

    # Interpolate to get even distribution of points on the translated convex hull
    num_points = 100
    fx = interp1d(cumulative_distances, closed_hull_points[:, 0], kind='linear')
    fy = interp1d(cumulative_distances, closed_hull_points[:, 1], kind='linear')
    fz = interp1d(cumulative_distances, closed_hull_points[:, 2], kind='linear')
    even_distances = np.linspace(0, cumulative_distances[-1], num_points)
    evenly_distributed_points = np.column_stack([fx(even_distances), fy(even_distances),
fz(even_distances)])

    # Compute the mean of the evenly distributed points
    mean_evenly_distributed_points = np.mean(evenly_distributed_points, axis=0)
```

```
    # Project the center point back to 3D space
    center_point_3d = mean_evenly_distributed_points

    return center_point_3d, evenly_distributed_points, principal_components,
closed_hull_points_ptc_joint

# Helper function to generate points within a convex hull
def generate_points_in_hull(hull_points, spacing):
    """Generate points within the convex hull of a given set of points."""
    # Get the bounding box of the convex hull
    min_x, min_y = np.min(hull_points, axis=0)
    max_x, max_y = np.max(hull_points, axis=0)

    # Generate a grid of points over the bounding box
    x = np.arange(min_x, max_x, spacing)
    y = np.arange(min_y, max_y, spacing)
    xx, yy = np.meshgrid(x, y)
    grid_points = np.c_[xx.ravel(), yy.ravel()]

    # Filter points inside the convex hull
    hull_path = Delaunay(hull_points)
    mask = hull_path.find_simplex(grid_points) >= 0
    points_in_hull = grid_points[mask]

    return points_in_hull

# Function to map distance to a fixed color based on defined ranges
def get_colored_distance(distance, min_distance, max_distance, cmap):
    """
    Maps distance to a color using a custom colormap.

    Parameters:
    - distance: The distance value.
    - min_distance: The minimum distance in the dataset.
    - max_distance: The maximum distance in the dataset.
    - cmap: The colormap to use for mapping distances to colors.

    Returns:
    - A numpy array representing the color.
    """
    # Normalize the distance to be within [0, 1]
    normalized_distance = (distance - min_distance) / (max_distance - min_distance)

    # Get the color from the colormap
    color = cmap(normalized_distance)

    # Matplotlib returns colors in RGBA, we only need RGB
    return np.array(color[:3])

def evenly_distribute_points_on_surface(points_3d, spacing=0.5):
    """
    Distribute points evenly over the surface area covered by points_3d using a grid.
```

```
    Parameters:
    - points_3d: np.array, original 3D points on the surface.
    - spacing: float, the spacing between points in the grid.

    Returns:
    - grid_points_3d: np.array, evenly distributed points on the surface.
    """
    # Delaunay triangulation of the original points
    tri = Delaunay(points_3d[:, :2])  # Using only the first two components for 2D
triangulation

    # Generate a grid over the bounding box of the original points
    min_bounds = np.min(points_3d, axis=0)
    max_bounds = np.max(points_3d, axis=0)

    # Create a uniform grid within the bounds
    x = np.arange(min_bounds[0], max_bounds[0], spacing)
    y = np.arange(min_bounds[1], max_bounds[1], spacing)
    xx, yy = np.meshgrid(x, y)
    grid_points_2d = np.c_[xx.ravel(), yy.ravel()]

    # Filter grid points that are inside the convex hull of the original points
    mask = tri.find_simplex(grid_points_2d) >= 0
    grid_points_2d = grid_points_2d[mask]

    # Interpolate the z values for the grid points based on the original points
    grid_z = griddata(points_3d[:, :2], points_3d[:, 2], grid_points_2d, method='linear')

    # Combine x, y, and interpolated z values into new 3D points
    grid_points_3d = np.column_stack([grid_points_2d, grid_z])

    return grid_points_3d

# Define the number of patients per row and initialize counters
patients_per_row = 10
num_patients = len([d for d in os.listdir(base_output_dir) if
os.path.isdir(os.path.join(base_output_dir, d))])
num_rows = (num_patients + patients_per_row - 1) // patients_per_row
distance_plotter = pv.Plotter(shape=(num_rows, patients_per_row), window_size=[1600, 400 *
num_rows])
gap_area_plotter = pv.Plotter(shape=(num_rows, patients_per_row), window_size=[1600, 400 *
num_rows])
results = []

def write_results_to_csv(results, output_dir):
    csv_file_path = os.path.join(output_dir, "distance_and_gap.csv")
    with open(csv_file_path, "w", newline="") as file:
        writer = csv.writer(file, delimiter=';')
        writer.writerow(["Patient", "Mean Distance (mm)", "Standard Deviation (mm)", "Gap
Area (%)"])
        for result in results:
            mean_distance = f"{result[1]:.2f}".replace('.', ',')
            std_dev = f"{result[2]:.2f}".replace('.', ',')
```

```
            gap_area = f"{result[3]:.2f}".replace('.', ',')
            writer.writerow([result[0], mean_distance, std_dev, gap_area])

for patient_index, patient_dir in enumerate(os.listdir(base_output_dir)):
    patient_path = os.path.join(base_output_dir, patient_dir)
    if os.path.isdir(patient_path):
        match = re.search(r'(\d+)([RL])$', patient_dir)
        if match:
            patient_number = match.group(1)
            side_indicator = match.group(2)
            patient_name = f"{patient_number}{side_indicator}"
        else:
            print(f"No valid side indicator found in patient directory name:
{patient_dir}")
            continue

        input_calcaneus = os.path.join(patient_path, 'Aligned_with_talus', 'calcaneus')
        input_calcaneus_ptc_combined = os.path.join(patient_path, 'Aligned_with_talus',
'combined_calc_stl')
        input_talus = os.path.join(patient_path, 'Aligned_with_talus', 'talus')
        input_talus_ptc = os.path.join(patient_path, 'Aligned_with_talus', 'talus_PTC')

        # Create the combined calcaneus file per patient with patient number and side
indicator
        combine_stl_files(input_calcaneus, input_calcaneus_ptc_combined, output_file,
patient_number, side_indicator)

        for filename_calcaneus, filename_calcaneus_ptc_combined, filename_talus,
filename_talus_ptc in zip(
                os.listdir(input_calcaneus), os.listdir(input_calcaneus_ptc_combined),
                os.listdir(input_talus), os.listdir(input_talus_ptc)):

            filenames.append(f"{patient_name}_{filename_calcaneus}")

            calcaneus_mesh = load_and_decimate_mesh(os.path.join(input_calcaneus,
filename_calcaneus), point_threshold=5000)
            calcaneus_ptc_mesh =
load_and_decimate_mesh(os.path.join(input_calcaneus_ptc_combined,
filename_calcaneus_ptc_combined), point_threshold=5000)

            if side_indicator == 'L':
                talus_template_path = input_talus_template_left
                calcaneus_template_path = input_calcaneus_template_left
            elif side_indicator == 'R':
                talus_template_path = input_talus_template_right
                calcaneus_template_path = input_calcaneus_template_right
            else:
                print(f"Unknown side indicator {side_indicator} for patient
{patient_number}. Skipping.")
                continue

            talus_mesh = load_and_decimate_mesh(os.path.join(input_talus, filename_talus),
point_threshold=5000)
```

```
            talus_mesh_template = load_and_decimate_mesh(talus_template_path,
point_threshold=5000)
            talus_ptc_mesh = load_and_decimate_mesh(os.path.join(input_talus_ptc,
filename_talus_ptc), point_threshold=5000)

            calcaneus_template_mesh = load_and_decimate_mesh(calcaneus_template_path,
point_threshold=5000)

            com_calcaneus = calcaneus_mesh.center_of_mass()
            com_calcaneus_ptc = calcaneus_ptc_mesh.center_of_mass()

            direction_calc_to_ptc_joint = com_calcaneus_ptc - com_calcaneus

            com_talus = talus_mesh.center_of_mass()
            com_talus_ptc = talus_ptc_mesh.center_of_mass()

            direction_talus_to_ptc_joint = com_talus_ptc - com_talus

            center_tub, cloud_tub, principal_components, cloud_ptc =
compute_center_of_point_cloud(
                calcaneus_ptc_mesh.points, com_talus_ptc, com_calcaneus_ptc)

            normals_ptc_calcaneus = calcaneus_ptc_mesh.compute_normals(cell_normals=False)
            normals_ptc_talus = talus_ptc_mesh.compute_normals(cell_normals=False)

            filtered_normals_calcaneus, indices_calcaneus, filtered_points_calcaneus =
filter_normals_by_angle(
                normals_ptc_calcaneus, direction_calc_to_ptc_joint)

            if len(filtered_points_calcaneus) == 0:  # Fallback: Use all points and
normals if filtering fails
                filtered_points_calcaneus = normals_ptc_calcaneus.points
                filtered_normals_calcaneus = normals_ptc_calcaneus.point_normals

            filtered_normals_talus, indices_talus, filtered_points_talus =
filter_normals_by_angle(
                normals_ptc_talus, direction_talus_to_ptc_joint)

            print(f"Processing patient {patient_number} ({side_indicator})")
            if filtered_points_calcaneus.size == 0:
                print(f"No filtered points for calcaneus in patient {patient_number}
({side_indicator})")

            # Ensure there are filtered points and normals to process
            if len(filtered_points_calcaneus) > 0:
                convex_hull_mesh = pv.PolyData(cloud_tub)
                convex_hull_mesh_ptc = pv.PolyData(cloud_ptc)
                filled_convex_full_mesh = convex_hull_mesh.delaunay_2d()
                filled_convex_full_mesh_ptc = convex_hull_mesh_ptc.delaunay_2d()
                normal_filled_convex_full_mesh_ptc =
filled_convex_full_mesh_ptc.compute_normals(cell_normals=False)
                normals_filled_convex_full_mesh_ptc =
normal_filled_convex_full_mesh_ptc.point_normals
```

```
                distances_to_calc, corresponding_point_calc, original_points,
original_vectors, _ = distance_PTC_joint_convex_hull(
                    filtered_normals_talus, filtered_points_talus, calcaneus_ptc_mesh,
filled_convex_full_mesh)

                _, _, _, _, hull_intersections = distance_PTC_joint_convex_hull(
                    normals_filled_convex_full_mesh_ptc,
filled_convex_full_mesh_ptc.points,
                    calcaneus_ptc_mesh, filled_convex_full_mesh)

                hull_intersections_array = np.array(hull_intersections)

                mean_distance_PTC_joint = np.mean(distances_to_calc)
                # percentage_threshold = 0.50
                # distance_threshold = mean_distance_PTC_joint * (1 +
percentage_threshold)

                filtered_distances = [dist for dist in distances_to_calc] #if dist <=
distance_threshold]
                filtered_points = [point for dist, point in zip(distances_to_calc,
corresponding_point_calc)]# if dist <= distance_threshold]


                if len(filtered_points) == 0:
                    print(f"No filtered points after distance thresholding in patient
{patient_number} ({side_indicator})")

                # Ensure there are filtered points to work with
                if len(filtered_points) > 0:
                    filtered_points_3d = np.array(filtered_points)

                    # Apply the evenly distribute points function to filtered points
                    evenly_distribute_points_3d =
evenly_distribute_points_on_surface(filtered_points_3d)

                    # Perform distance calculations using the evenly distributed points
                    tree = cKDTree(filtered_points_calcaneus)
                    closest_points_indices = tree.query(evenly_distribute_points_3d,
k=1)[1]
                    closest_points_calcaneus =
np.array(filtered_points_calcaneus)[closest_points_indices]
                    closest_normals_calcaneus =
np.array(filtered_normals_calcaneus)[closest_points_indices]

                    distances_to_talus, corresponding_point_talus, original_point_calc,
original_vector_calc = distance_PTC_joint(
                        closest_normals_calcaneus, closest_points_calcaneus,
talus_ptc_mesh)

                    # Filter distances and points again based on the threshold
                    filtered_points_calc = [(dist, point) for dist, point in
zip(distances_to_talus, original_point_calc)]# if dist <= distance_threshold]
```

```
                    # Extract distances and points for further analysis
                    distances = [dist for dist, point in filtered_points_calc]
                    std_dev = np.std(distances)
                    std_devs.append(std_dev)
                    mean_distance_between_bones = np.mean(distances)
                    combined_distances.append(mean_distance_between_bones)

                    # Update points with filtered values
                    points = [point for dist, point in filtered_points_calc]
                    points_3d = np.array(points)

                    # Calculate percentage of holes
                    if cloud_ptc.size > 0:
                        percentage_hole = (len(hull_intersections_array) * 100) /
len(cloud_ptc)
                    else:
                        percentage_hole = 0

                    # Store patient results
                    results.append([patient_name, mean_distance_between_bones, std_dev,
percentage_hole])
                else:
                    print(f"Skipping patient {patient_number} ({side_indicator}) due to
insufficient data.")
                    results.append([patient_name, float('nan'), float('nan'), 0])
            else:
                print(f"Skipping patient {patient_number} ({side_indicator}) due to
insufficient data.")
                results.append([patient_name, float('nan'), float('nan'), 0])

            # Prepare for visualization
            if len(filtered_points) > 0:
                colors = [
                    (0.0, 1.0, 0.0),  # Green
                    (0.0, 0.8, 0.8),  # Blue-Green
                    (0.0, 0.0, 1.0),  # Blue
                    (0.5, 0.0, 1.0),  # Purple-Blue
                    (1.0, 0.0, 1.0),  # Purple
                ]

                # Create a custom colormap
                custom_cmap = LinearSegmentedColormap.from_list('custom_cmap', colors,
N=256)

                pca = PCA(n_components=2)
                projected_points_2d = pca.fit_transform(points_3d)
                mean_3d = np.mean(points_3d, axis=0)
                projected_points_3d = projected_points_2d @ pca.components_[:2, :] +
mean_3d
                min_dist = min(distances)
                max_dist = max(distances)
```

```
                colors_1 = np.array([get_colored_distance(d, min_dist, max_dist,
custom_cmap) for d in distances])
                row_idx = patient_index // patients_per_row
                col_idx = patient_index % patients_per_row

                # Add each patient's results to the distance plotter's subplot
                distance_plotter.subplot(row_idx, col_idx)
                distance_plotter.add_points(points_3d, scalars=colors_1[:, :3], rgb=True,
point_size=3, render_points_as_spheres=True)
                #distance_plotter.add_mesh(talus_ptc_mesh, color="cyan", opacity=0.5)
                distance_plotter.add_mesh(calcaneus_ptc_mesh, color="orange")
                distance_plotter.add_text(f"{patient_name}", font_size=12)

                center_PTC_joint_calcaneus = pv.PolyData(com_calcaneus_ptc)
                convex_hull_ptc_calcaneus = pv.PolyData(cloud_tub)
                center_ptc_calcaneus = pv.PolyData(center_tub)
                start_point = center_tub
                end_point = center_tub + principal_components[2] * 10

                gap_area_plotter.subplot(row_idx, col_idx)
                #gap_area_plotter.add_mesh(center_ptc_calcaneus, color='pink',
point_size=7.0, render_points_as_spheres=True)
                gap_area_plotter.add_mesh(calcaneus_ptc_mesh, color="orange", opacity=0.5)
                gap_area_plotter.add_mesh(pv.Line(start_point, end_point), color='green',
line_width=5, label='3rd Principal Component')

                if hull_intersections:
                    gap_area_plotter.add_points(hull_intersections_array, color='red',
point_size=5, label='Hull Intersections')

                gap_area_plotter.add_mesh(filled_convex_full_mesh, color='lightblue',
opacity=0.5, label='Convex Hull')
                gap_area_plotter.add_mesh(filled_convex_full_mesh_ptc, color='lightgreen',
opacity=0.5, label='Convex Hull')
                gap_area_plotter.add_axes()
                gap_area_plotter.add_text(f"{patient_name}", font_size=12)

# Display all patients' distance plots together
distance_plotter.show()

# Display all patients' gap area plots together
gap_area_plotter.show()

# Calculate the mean
mean_distance = np.mean(combined_distances)


# Save results to a CSV file in the base output directory
write_results_to_csv(results, base_output_dir)

# Stop timing before the script ends
end_time = time.time()  # Record the end time
elapsed_time = end_time - start_time
```

```
minutes, seconds = divmod(elapsed_time, 60)
hours, minutes = divmod(minutes, 60)

print(f"Total runtime (excluding folder selection): {int(hours)}h {int(minutes)}m
{int(seconds)}s")
```

Appendix E – Number of fragments, fragment area, and fracture area calculations

```
import os
import numpy as np
from stl import mesh
import pyvista as pv
from collections import defaultdict
import tkinter as tk
from tkinter import filedialog
import trimesh
import scipy.sparse
import networkx as nx
from scipy.spatial import Delaunay
import pandas as pd
import time

def calculate_normal(v0, v1, v2):
    """Calculate the normal vector of the triangle."""
    return np.cross(v1 - v0, v2 - v0)

def dihedral_angle(normal1, normal2):
    """Calculate the dihedral angle between two normals."""
    cosine_angle = np.dot(normal1, normal2) / (np.linalg.norm(normal1) *
np.linalg.norm(normal2))
    cosine_angle = np.clip(cosine_angle, -1.0, 1.0)
    return np.arccos(cosine_angle)

def find_sharp_edges(stl_mesh, threshold_angle):
    """Find sharp edges in the mesh based on the dihedral angle threshold."""
    edges = {}
    for i, face in enumerate(stl_mesh.vectors):
        normal = calculate_normal(face[0], face[1], face[2])
        for j in range(3):
            edge = tuple(sorted((tuple(face[j]), tuple(face[(j + 1) % 3]))))
            if edge not in edges:
                edges[edge] = []
            edges[edge].append((i, normal))

    sharp_edges = []
    threshold_radians = np.radians(threshold_angle)
    for edge, normals in edges.items():
        if len(normals) == 2:
            angle = dihedral_angle(normals[0][1], normals[1][1])
            if angle > threshold_radians:
                sharp_edges.append((edge, [n[0] for n in normals]))

    return sharp_edges

def build_edge_graph(sharp_edges):
    """Build a graph of sharp edges where vertices are connected edges."""
    edge_graph = defaultdict(set)
```

```
    for edge, _ in sharp_edges:
        v0, v1 = edge
        edge_graph[v0].add(edge)
        edge_graph[v1].add(edge)
    return edge_graph

def find_connected_components(edge_graph):
    """Find connected components in the edge graph."""
    visited = set()
    components = []

    def dfs(node, component):
        stack = [node]
        while stack:
            v = stack.pop()
            if v not in visited:
                visited.add(v)
                component.add(v)
                for neighbor_edge in edge_graph[v]:
                    for neighbor in neighbor_edge:
                        if neighbor not in visited:
                            stack.append(neighbor)

    for node in edge_graph:
        if node not in visited:
            component = set()
            dfs(node, component)
            components.append(component)

    return components

def filter_connected_sharp_edges(sharp_edges, min_size=5):
    """Filter sharp edges to keep only those in large connected components."""
    edge_graph = build_edge_graph(sharp_edges)
    components = find_connected_components(edge_graph)

    # Filter out small components, keeping only large connected components
    large_components = [comp for comp in components if len(comp) > min_size]

    connected_sharp_edges = []
    for edge, triangles in sharp_edges:
        v0, v1 = edge
        for comp in large_components:
            if v0 in comp or v1 in comp:
                connected_sharp_edges.append((edge, triangles))
                break

    return connected_sharp_edges

def create_region_mesh(stl_mesh, sharp_edges):
    """Create a mesh of all triangles encaptured by sharp edges and return the triangles
array."""
    edge_triangles = set()
```

```
    for edge, triangle_indices in sharp_edges:
        for triangle_index in triangle_indices:
            edge_triangles.add(triangle_index)

    triangles = []
    for triangle_index in edge_triangles:
        triangles.append(stl_mesh.vectors[triangle_index])

    return np.array(triangles)

def save_mesh_as_stl(triangles, file_path):
    """Save the triangles as an STL file."""
    if triangles.size == 0:
        print(f"No triangles to save for {file_path}")
        return

    region_mesh = mesh.Mesh(np.zeros(triangles.shape[0], dtype=mesh.Mesh.dtype))
    for i, triangle in enumerate(triangles):
        region_mesh.vectors[i] = triangle

    region_mesh.save(file_path)

def extract_boundary_points(sharp_edges):
    """Extract boundary points from sharp edges."""
    points = set()
    for edge, _ in sharp_edges:
        points.add(edge[0])
        points.add(edge[1])
    return np.array(list(points))

def create_surface_mesh(boundary_points):
    """Create a surface mesh using Delaunay triangulation."""
    if len(boundary_points) < 3:
        print("Not enough points to perform triangulation.")
        return None

    tri = Delaunay(boundary_points[:, :2])  # Perform 2D Delaunay triangulation on the XY
plane
    triangles = boundary_points[tri.simplices]
    return triangles

def save_surface_mesh(triangles, file_path):
    """Save the surface mesh as an STL file."""
    if triangles is None or triangles.size == 0:
        print(f"No triangles to save for {file_path}")
        return

    surface_mesh = mesh.Mesh(np.zeros(triangles.shape[0], dtype=mesh.Mesh.dtype))
    for i, triangle in enumerate(triangles):
        surface_mesh.vectors[i] = triangle

    surface_mesh.save(file_path)
    print(f"Saved surface mesh to {file_path}")
```

```
def calculate_surface_area(triangles):
    """Calculate the surface area of the mesh."""
    if triangles.size == 0:
        return 0.0

    def triangle_area(triangle):
        a = np.linalg.norm(triangle[1] - triangle[0])
        b = np.linalg.norm(triangle[2] - triangle[0])
        c = np.linalg.norm(triangle[2] - triangle[1])
        s = (a + b + c) / 2
        return np.sqrt(s * (s - a) * (s - b) * (s - c))

    return sum(triangle_area(triangle) for triangle in triangles)


def create_and_save_surface_mesh(sharp_edges, output_path):
    """Extract boundary points, create a surface mesh, and save it as an STL file."""
    boundary_points = extract_boundary_points(sharp_edges)
    surface_triangles = create_surface_mesh(boundary_points)
    save_surface_mesh(surface_triangles, output_path)
    return surface_triangles


def process_individual_file(file_path, output_path, surface_output_path,
threshold_angle=10, min_size=20):
    """Process a single STL file for edge detection and save the region mesh and surface
mesh."""
    stl_mesh = mesh.Mesh.from_file(file_path)
    sharp_edges = find_sharp_edges(stl_mesh, threshold_angle)
    connected_sharp_edges = filter_connected_sharp_edges(sharp_edges, min_size)

    region_triangles = create_region_mesh(stl_mesh, connected_sharp_edges)
    save_mesh_as_stl(region_triangles, output_path)

    surface_triangles = create_and_save_surface_mesh(connected_sharp_edges,
surface_output_path)
    return surface_triangles

def create_combined_surface_mesh(surface_folder, combined_output_path):
    """Create a combined surface mesh from all surface meshes in the surface folder."""
    boundary_points = []

    for filename in os.listdir(surface_folder):
        if filename.endswith(".stl"):
            surface_mesh = mesh.Mesh.from_file(os.path.join(surface_folder, filename))
            for v in surface_mesh.vectors:
                boundary_points.extend(v)

    boundary_points = np.array(boundary_points)
    if len(boundary_points) < 3:
        print("Not enough points to perform triangulation.")
        return
```

```
    combined_surface_triangles = create_surface_mesh(boundary_points)
    save_surface_mesh(combined_surface_triangles, combined_output_path)
    return combined_surface_triangles


def visualize_all_stl(input_folder, edge_folder, surface_folder, combined_surface_path,
areas):
    """Visualize original STL files, edges, and surface meshes."""
    input_files = [f for f in os.listdir(input_folder) if f.endswith('.stl')]

    # Filter out fragments with no valid surface area
    valid_areas = [i for i, area in enumerate(areas[:-1]) if "fragment_area" in area]
    num_valid_fragments = len(valid_areas)

    plotter = pv.Plotter(shape=(num_valid_fragments + 1, 3), window_size=[1200, 400 *
(num_valid_fragments + 1)])

    for i, valid_index in enumerate(valid_areas):
        filename = input_files[valid_index]
        input_file = os.path.join(input_folder, filename)
        edge_file = os.path.join(edge_folder,
f"{os.path.splitext(filename)[0]}_region.stl")
        surface_file = os.path.join(surface_folder,
f"{os.path.splitext(filename)[0]}_surface.stl")

        original_mesh = pv.read(input_file)

        # Check if the edge file exists
        if os.path.exists(edge_file):
            edge_mesh = pv.read(edge_file)
        else:
            edge_mesh = None
            print(f"Edge file not found: {edge_file}")

        # Check if the surface file exists
        if os.path.exists(surface_file):
            surface_mesh = pv.read(surface_file)
        else:
            surface_mesh = None
            print(f"Surface file not found: {surface_file}")

        plotter.subplot(i, 0)
        plotter.add_mesh(original_mesh, color='white')
        plotter.add_title(f'Original {filename}\nArea:
{areas[valid_index]["fragment_area"]:.2f} mm²', font_size=10)

        plotter.subplot(i, 1)
        plotter.add_mesh(original_mesh, color='white', opacity=0.7)
        if edge_mesh:
            plotter.add_mesh(edge_mesh, color='red')
        plotter.add_title(f'Edges {filename}', font_size=10)
```

```
        plotter.subplot(i, 2)
        if surface_mesh:
            plotter.add_mesh(surface_mesh, color='blue')
            plotter.add_title(f'Surface {filename}\nArea:
{areas[valid_index]["fragment_area"]:.2f} mm²', font_size=10)

    # Visualize combined surface mesh
    combined_surface_mesh = pv.read(combined_surface_path)
    combined_area = areas[-1]["combined_area"]
    fracture_area = combined_area - sum(area["fragment_area"] for area in areas[:-1] if
"fragment_area" in area)
    plotter.subplot(num_valid_fragments, 2)
    plotter.add_mesh(combined_surface_mesh, color='green')
    plotter.add_title(f'Combined Surface Mesh\nCombined Area: {combined_area:.2f}
mm²\nFracture Area: {max(fracture_area, 0):.2f} mm²', font_size=10)

    plotter.show()


if __name__ == "__main__":
    root = tk.Tk()
    root.withdraw()

    base_input_folder = filedialog.askdirectory(title="Select the base directory
containing patient folders")
    if not base_input_folder:
        print("No directory selected. Exiting.")
        exit()

    base_output_folder = filedialog.askdirectory(title="Select the base directory for
output files")
    if not base_output_folder:
        print("No output directory selected. Exiting.")
        exit()

    # Start timing after folder selection
    start_time = time.time()  # Record the start time

    threshold_angle = 10
    min_size = 20

    results = []  # List to store results for each patient and fragment

    for patient_dir in os.listdir(base_input_folder):
        patient_input_folder = os.path.join(base_input_folder, patient_dir,
"Aligned_with_talus", "calcaneus")
        edge_output_folder = os.path.join(base_output_folder, patient_dir,
"Aligned_with_talus", "edges")
        combined_output_folder = os.path.join(base_output_folder, patient_dir,
"Aligned_with_talus", "combined_meshes_calc")
        surface_output_folder = os.path.join(base_output_folder, patient_dir,
"Aligned_with_talus", "surface_meshes")
```

```
        if not os.path.exists(patient_input_folder):
            print(f"Input folder not found: {patient_input_folder}")
            continue

        if not os.path.exists(edge_output_folder):
            os.makedirs(edge_output_folder)
        if not os.path.exists(surface_output_folder):
            os.makedirs(surface_output_folder)

        stl_files = [f for f in os.listdir(patient_input_folder) if f.endswith(".stl")]

        patient_areas = []
        fragment_triangles = []

        print(f"Processing patient: {patient_dir}")

        # Count the number of fragments for this patient
        num_fragments = len(stl_files)

        for filename in stl_files:
            input_path = os.path.join(patient_input_folder, filename)
            edge_output_path = os.path.join(edge_output_folder,
f"{os.path.splitext(filename)[0]}_region.stl")
            surface_output_path = os.path.join(surface_output_folder,
f"{os.path.splitext(filename)[0]}_surface.stl")

            surface_triangles = process_individual_file(input_path, edge_output_path,
surface_output_path, threshold_angle, min_size)

            # Check if surface_triangles is None or has no size (skip fragment if so)
            if surface_triangles is None or surface_triangles.size == 0:
                print(f"Skipping fragment {filename} due to insufficient points for
triangulation.")
                continue  # Skip to the next fragment

            # Calculate area of the fragment
            fragment_area = calculate_surface_area(surface_triangles)
            patient_areas.append({"filename": filename, "fragment_area": fragment_area})

            # Store fragment triangles for comparison
            fragment_triangles.append(surface_triangles)


        # Create combined surface mesh and calculate its area
        combined_surface_path = os.path.join(surface_output_folder,
"combined_surface.stl")
        combined_surface_triangles = create_combined_surface_mesh(surface_output_folder,
combined_surface_path)
        combined_area = calculate_surface_area(combined_surface_triangles)
        total_fragment_area = sum(area["fragment_area"] for area in patient_areas)
        fracture_area = combined_area - total_fragment_area

        # Determine the appropriate fracture area description
```

```
        if len(stl_files) == 1:
            fracture_area_description = "No intra-articular fracture in the PTC joint
surface"
            fracture_percentage = "N/A"
        elif fracture_area < 0:
            fracture_area_description = "Fracture area is negligible"
            fracture_area = 0.0  # Ensure fracture_area is zero for the percentage
calculation
            fracture_percentage = "N/A"
        else:
            fracture_area_description = f"{fracture_area}"
            fracture_percentage = (fracture_area / combined_area) * 100

        patient_areas.append({"filename": "combined_surface", "combined_area":
combined_area, "fracture_area": fracture_area_description})

        # Add results for the patient including the number of fragments
        results.append({
            "Patient": patient_dir,
            "Number of Fragments": num_fragments,  # New field for the number of fragments
            **{f"Area Fragment {i+1} (mm²)": area["fragment_area"] for i, area in
enumerate(patient_areas[:-1])},
            "Area of All Fragments (mm²)": total_fragment_area,
            "Combined Surface Area (mm²)": combined_area,
            "Fracture Area (mm²)": fracture_area_description,
            "Fracture Area (%)": fracture_percentage
        })
        # Stop timing before visualizations
        end_time = time.time()  # Record the end time

        visualize_all_stl(patient_input_folder, edge_output_folder, surface_output_folder,
combined_surface_path, patient_areas)



    # Calculate the total elapsed time (in seconds) and convert to a readable format
    elapsed_time = end_time - start_time
    minutes, seconds = divmod(elapsed_time, 60)
    hours, minutes = divmod(minutes, 60)

    # Print the elapsed time
    print(f"Total runtime (excluding folder selection and visualization): {int(hours)}h
{int(minutes)}m {int(seconds)}s")


    # Save results to Excel
    df = pd.DataFrame(results)
    column_order = ['Patient', 'Number of Fragments'] + sorted([col for col in df.columns
if col.startswith("Area Fragment")], key=lambda x: int(x.split()[2])) + ['Area of All
Fragments (mm²)', 'Combined Surface Area (mm²)', 'Fracture Area (mm²)', 'Fracture Area
(%)']
    df = df[column_order]
```

```
# Save the updated DataFrame to an Excel file
output_excel_path = os.path.join(base_output_folder, "surface_and_fracture_area.xlsx")
df.to_excel(output_excel_path, index=False)
print(f"Saved surface areas with percentage to {output_excel_path}")
```

Appendix F – Maximal step-off and maximal gap measurements

```
import os
import pyvista as pv
import numpy as np
import pandas as pd
from tkinter import filedialog, Tk
import time
from scipy.spatial import ConvexHull, Delaunay
import re
import random

# Define function to select directories
def select_directory(title):
    root = Tk()
    root.withdraw()  # Hide the main window
    folder_path = filedialog.askdirectory(title=title)
    return folder_path

# Select the base output directory
base_output_dir = select_directory("Select the base output directory where the STL files
are stored")

# Start timing after file selection
start_time = time.time()  # Record the start time

combined_distances = []  # List to store distances for each patient
results = []

# Function to shrink the convex hull by scaling points inward towards the centroid
def shrink_convex_hull(points, scale_factor=1):
    centroid = np.mean(points, axis=0)
    shrunk_points = centroid + scale_factor * (points - centroid)
    return shrunk_points

# Function to project a point onto a plane defined by the combined convex hull and third
component
def project_onto_plane(point, plane_point, normal_vector):
    vector = point - plane_point
    distance_to_plane = np.dot(vector, normal_vector)
    projected_point = point - distance_to_plane * normal_vector
    return projected_point

# Function to filter points inside the 2D convex hull
def filter_points_in_hull(points_2d, hull_2d):
    delaunay = Delaunay(hull_2d)
    mask = delaunay.find_simplex(points_2d) >= 0  # Check if the points are inside the
Delaunay triangulation
    return points_2d[mask]

# Function to check if any fragment intervenes within a specific radius of the mutual
closest point line
```

```
def is_fragment_in_between_2d_radius(proj_point1, proj_point2, projected_fragments,
exclude_indices, tolerance=0.1):
    line_vector = proj_point2[:2] - proj_point1[:2]
    line_length = np.linalg.norm(line_vector)
    normalized_line_vector = line_vector / line_length

    for idx, fragment in enumerate(projected_fragments):
        if idx in exclude_indices:  # Skip the fragments that are being evaluated
            continue

        for point in fragment:
            projection_length = np.dot((point[:2] - proj_point1[:2]),
normalized_line_vector)
            if 0 <= projection_length <= line_length:
                closest_point_on_line = proj_point1[:2] + projection_length *
normalized_line_vector
                distance_to_line = np.linalg.norm(point[:2] - closest_point_on_line)

                if distance_to_line <= tolerance:
                    return True
    return False

# Function to find mutual closest pairs based on 2D distances on the convex hull after
projection
def find_mutual_closest_combined_line(fragment_hulls, combined_hull, third_component,
tolerance=0.1):
    longest_total_line = None
    max_total_length = -np.inf
    max_gap_pair = None
    max_gap_length = -np.inf
    mutual_closest_pairs = []

    projected_fragments = []
    original_fragments = []

    for fragment in fragment_hulls:
        projected_points = []
        original_points = []
        for point in fragment.points:
            projected_point = project_onto_plane(point, np.mean(combined_hull.points,
axis=0), third_component)
            projected_points.append(projected_point)
            original_points.append(point)

        projected_fragments.append(np.array(projected_points))
        original_fragments.append(np.array(original_points))

    for i in range(len(fragment_hulls)):
        for j in range(i + 1, len(fragment_hulls)):
            fragment1_projected = projected_fragments[i]
            fragment2_projected = projected_fragments[j]
            fragment1_original = original_fragments[i]
            fragment2_original = original_fragments[j]
```

```
            for idx1, proj_point1 in enumerate(fragment1_projected):
                min_distance = np.inf
                closest_idx2 = None
                for idx2, proj_point2 in enumerate(fragment2_projected):
                    distance = np.linalg.norm(proj_point1[:2] - proj_point2[:2])
                    if distance < min_distance:
                        min_distance = distance
                        closest_idx2 = idx2

                closest_back_idx1 = None
                min_back_distance = np.inf
                for idx_back, proj_point_back in enumerate(fragment1_projected):
                    distance_back = np.linalg.norm(proj_point_back[:2] -
fragment2_projected[closest_idx2][:2])
                    if distance_back < min_back_distance:
                        min_back_distance = distance_back
                        closest_back_idx1 = idx_back

                if closest_back_idx1 == idx1:
                    point1 = fragment1_original[idx1]
                    point2 = fragment2_original[closest_idx2]
                    projected1 = fragment1_projected[idx1]
                    projected2 = fragment2_projected[closest_idx2]

                    if not is_fragment_in_between_2d_radius(projected1, projected2,
projected_fragments, exclude_indices=[i, j], tolerance=tolerance):
                        mutual_closest_pairs.append((point1, point2, projected1,
projected2))

                        gap_length = np.linalg.norm(projected1[:2] - projected2[:2])
                        if gap_length > max_gap_length:
                            max_gap_length = gap_length
                            max_gap_pair = (projected1, projected2)

    for point1, point2, projected1, projected2 in mutual_closest_pairs:
        length1 = np.linalg.norm(point1 - projected1)
        length2 = np.linalg.norm(point2 - projected2)

        side1 = np.sign(np.dot(point1 - np.mean(combined_hull.points, axis=0),
third_component))
        side2 = np.sign(np.dot(point2 - np.mean(combined_hull.points, axis=0),
third_component))

        if side1 == side2:
            total_length = abs(length1 - length2)
        else:
            total_length = abs(length1 + length2)

        if total_length > max_total_length:
            max_total_length = total_length
            longest_total_line = (point1, point2, projected1, projected2)
```

```
    return mutual_closest_pairs, longest_total_line, max_total_length, max_gap_length,
max_gap_pair

# Function to generate random colors
def generate_random_color():
    return [random.uniform(0, 1) for _ in range(3)]

# Function to create a 3D convex hull mesh and return it for plotting
def create_transparent_convex_hull_mesh_3d(mesh, scale_factor=1):
    points = mesh.points
    if points.shape[0] > 0:
        if points.shape[0] < 4:
            print("Not enough points to create a 3D convex hull.")
            return None, None, None

        mean = np.mean(points, axis=0)
        centered_points = points - mean
        U, S, Vt = np.linalg.svd(centered_points)
        principal_components = Vt[:2].T
        third_component = Vt[2]
        points_2d = centered_points @ principal_components

        hull = ConvexHull(points_2d)
        hull_points_2d = points_2d[hull.vertices]
        shrunk_hull_points_2d = shrink_convex_hull(hull_points_2d, scale_factor)
        shrunk_hull_points_3d = shrunk_hull_points_2d @ principal_components.T + mean
        convex_hull_3d = ConvexHull(shrunk_hull_points_3d)
        faces = convex_hull_3d.simplices
        face_array = np.column_stack((np.full(len(faces), 3), faces))
        mesh = pv.PolyData(shrunk_hull_points_3d, face_array)
        return mesh, principal_components, third_component
    return None, None, None

# Function to plot patient results with transparent convex hull
def plot_patient(filled_hull, fragment_hulls, fragment_colors, patient_name,
mutual_closest_pairs, longest_total_line=None, max_gap_pair=None):
    plotter = pv.Plotter()

    plotter.add_mesh(filled_hull, color='purple', opacity=0.2, line_width=1)

    for hull, color in zip(fragment_hulls, fragment_colors):
        plotter.add_mesh(hull, color=color, line_width=2, point_size=3.0,
render_points_as_spheres=True)

    for (point1, point2, projected1, projected2) in mutual_closest_pairs:
        line1 = pv.Line(point1, projected1)
        line2 = pv.Line(point2, projected2)
        plotter.add_mesh(line1, color='green', line_width=2)
        plotter.add_mesh(line2, color='green', line_width=2)
        connection_line = pv.Line(projected1, projected2)
        plotter.add_mesh(connection_line, color='purple', line_width=2)

    if longest_total_line is not None:
```

```
        point1, point2, projected1, projected2 = longest_total_line
        line1 = pv.Line(point1, projected1)
        line2 = pv.Line(point2, projected2)
        plotter.add_mesh(line1, color='red', line_width=3)
        plotter.add_mesh(line2, color='red', line_width=3)
        connection_line = pv.Line(projected1, projected2)
        plotter.add_mesh(connection_line, color='lightblue', line_width=5)

        length1 = np.linalg.norm(point1 - projected1)
        length2 = np.linalg.norm(point2 - projected2)
        side1 = np.sign(np.dot(point1 - np.mean(filled_hull.points, axis=0),
third_component))
        side2 = np.sign(np.dot(point2 - np.mean(filled_hull.points, axis=0),
third_component))

        if side1 == side2:
            total_length = abs(length1 - length2)
            print(f"Patient {patient_name} - Maximal Step: {length1:.2f} mm -
{length2:.2f} mm = {total_length:.2f} mm (same side)")
        else:
            total_length = abs(length1 + length2)
            print(f"Patient {patient_name} - Maximal Step: {length1:.2f} mm +
{length2:.2f} mm = {total_length:.2f} mm (opposite sides)")

    if max_gap_pair is not None:
        projected1, projected2 = max_gap_pair
        gap_line = pv.Line(projected1, projected2)
        plotter.add_mesh(gap_line, color='orange', line_width=4)
        gap_length = np.linalg.norm(projected1[:2] - projected2[:2])
        print(f"Patient {patient_name} - Maximal Gap: {gap_length:.2f} mm")

    plotter.add_text(f"Patient {patient_name}: maximal step", font_size=12)
    plotter.show()

# Main processing loop
for patient_dir in os.listdir(base_output_dir):
    patient_path = os.path.join(base_output_dir, patient_dir)
    if os.path.isdir(patient_path):
        match = re.search(r'(\d+)([RL])$', patient_dir)
        if match:
            patient_number = match.group(1)
            side_indicator = match.group(2)
            patient_name = f"{patient_number}{side_indicator}"

            combined_stl_file = os.path.join(patient_path, 'Aligned_with_talus',
'combined_calc_stl', f"combined_calcaneus_patient{patient_number}_{side_indicator}.stl")
            fragments_dir = os.path.join(patient_path, 'Aligned_with_talus', 'edges')

            if os.path.exists(combined_stl_file) and os.path.exists(fragments_dir):
                calcaneus_ptc_mesh = pv.read(combined_stl_file)
                transparent_hull_mesh, principal_components, third_component =
create_transparent_convex_hull_mesh_3d(calcaneus_ptc_mesh)
                fragment_hulls = []
```

```
                fragment_colors = []

                for stl_file in os.listdir(fragments_dir):
                    if stl_file.endswith(".stl"):
                        fragment_mesh = pv.read(os.path.join(fragments_dir, stl_file))
                        fragment_hull = shrink_convex_hull(fragment_mesh.points,
scale_factor=0.9)
                        if fragment_hull is not None:
                            fragment_hulls.append(pv.PolyData(fragment_hull))
                            fragment_colors.append(generate_random_color())

                if len(fragment_hulls) > 1:
                    mutual_closest_pairs, longest_total_line, max_total_length,
max_gap_length, max_gap_pair = find_mutual_closest_combined_line(
                        fragment_hulls, transparent_hull_mesh, third_component,
tolerance=0.1
                    )

                    combined_distances.append({
                        "Patient": patient_name,
                        "Maximal step-off (mm)": max_total_length,
                        "Maximal gap (mm)": max_gap_length
                    })

                    plot_patient(transparent_hull_mesh, fragment_hulls, fragment_colors,
patient_name, mutual_closest_pairs, longest_total_line, max_gap_pair)
                else:
                    print(f"Skipping patient {patient_name}: No valid convex hulls or only
one fragment found.")
                    combined_distances.append({
                        "Patient": patient_name,
                        "Maximal step-off (mm)": 0,
                        "Maximal gap (mm)": 0
                    })
            else:
                print(f"Combined STL file or edges folder not found for {patient_name}.
Skipping this patient.")
                combined_distances.append({
                    "Patient": patient_name,
                    "Maximal step-off (mm)": 0,
                    "Maximal gap (mm)": 0
                })
        else:
            print(f"Skipping patient {patient_dir}: no valid side indicator found.")
            combined_distances.append({
                "Patient": patient_dir,
                "Maximal step-off (mm)": 0,
                "Maximal gap (mm)": 0
            })

# Stop timing before the script ends
end_time = time.time()
elapsed_time = end_time - start_time
```

```
minutes, seconds = divmod(elapsed_time, 60)
hours, minutes = divmod(minutes, 60)
print(f"Total runtime: {int(hours)}h {int(minutes)}m {int(seconds)}s")

# Save combined distances to Excel file with correct column names
df = pd.DataFrame(combined_distances)
output_file = os.path.join(base_output_dir, "maximal_step_and_maximal_gap_3d.xlsx")
df.to_excel(output_file, index=False)
print(f"Results saved to {output_file}")
```

Appendix G – Manual measurements tool in MEVISLAB

Overview of the model connections:

The tool was designed by connecting various MeVisLab modules to enable seamless interaction between the visualization and analysis of patient data, supporting both manual measurement and classification workflows.

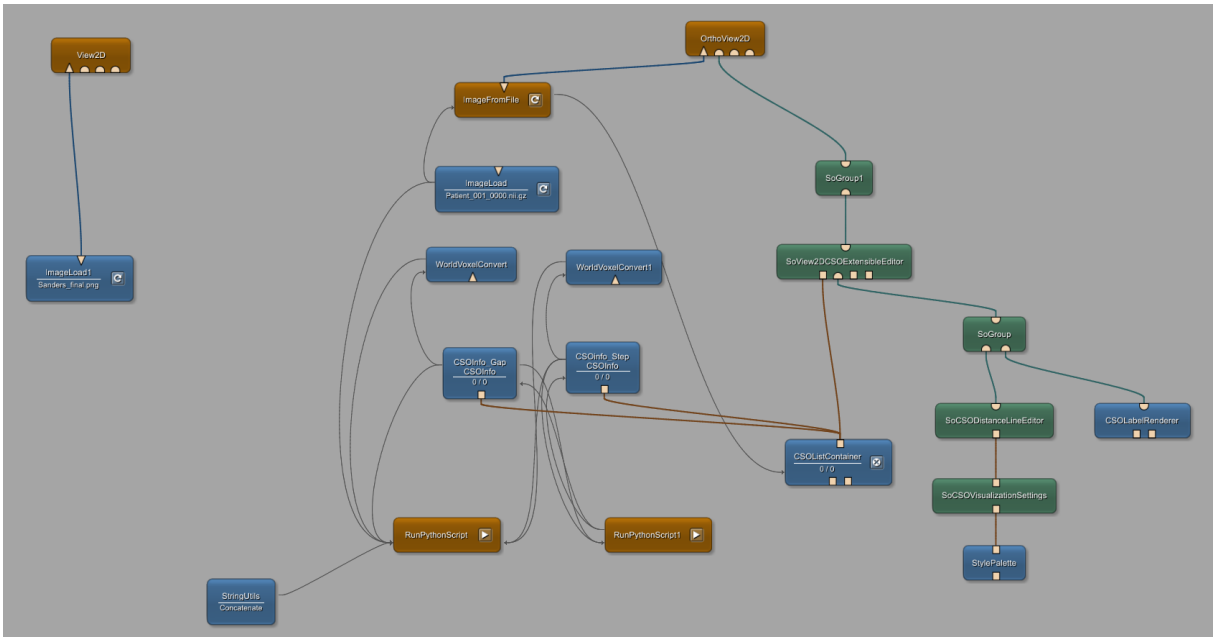


Figure 11: Overview of the model connections

MEVISLAB script

Custom scripting was integrated to automate data processing steps and enhance user interaction, allowing efficient calculation of maximal gap and step-off distances directly from the patient's CT scans.

```
Interface {
 Parameters {
  Field Sanders_classification {
   type = String
   value = 0,1,2,3,4
  }
  Field targetFileName { type = string isFilePath = Yes}
  Field targetSavePath { type = string isFilePath = Yes}

 }
}

Window {
 title = "Sander Classification"
 Horizontal "Sanders classification and gap and step measurements"{
  Splitter {
   Category {
    Field ImageLoad.filename {
     title      = "Browse for CT scans"
     browseMode = open
     expandX    = Yes
     expandY    = Yes
    }
    Box  {
     Button ImageFromFile.openInputFile {
      title   = "Load File"
```

```
        expandX = Yes
        expandY = Yes
       }
      }
      viewer OrthoView2D.self {
       type    = SoRenderArea
       ph      = 1024
       pw      = 1024
       expandX = true
       expandY = true
      }

      Button {
       title   = "Save DICOM"
       expandX = True
       expandY = True
      }
     }
    }

    Category  {
     Box "User guided interface Sander Classification"  {
      HyperText {
       text    = "
       <h2>Adjust Image:</h2>
       <ul>
       <li>Dragging: <b>Shift</b> + click mousewheel</li>
       <li>Zooming: <b>Ctrl</b> + click mousewheel</li>
       <li>Brightness: Right click</li>
       </ul>

       <hr>

       <h2>Sanders Classification:</h2>
       <p>Choose the correct Sanders classification type from the drop-down menu(Sanders type 1, 2, 3, or 4).
       <hr>

       <h2>Gap and Step Measurements:</h2>
       <p>Measure the largest gap and step present in the PTC
       <h3>Gap Measurement</h3>

       <ul>
       <li>Press <b>Alt</b> and left-click for the first point of the measurement.</li>
       <li>Move the line to the desired second point.</li>
       <li>Press <b>Alt</b> and set the second point of the gap measurement.</li>
       </ul>

       <h3>Step Measurement</h3>
       <ul>
       <li>Press <b>Alt</b> and left-click for the first point of the measurement.</li>
       <li>Move the line to the desired second point.</li>
       <li>Press <b>Alt</b> and set the second point of the step measurement.</li>
       </ul>

       <p>If the measurements are incorrect, press the 'Delete all lines' button. Note: both lines will be removed.</p>

       <p>When you are satisfied with the measurements, save the data by clicking the 'Save data' button. After that, you can load in a new
patient.</p>"
       ph = 400
       pw = 700
      }
      viewer View2D.self {
       type    = SoRenderArea
```

```
      ph     = 300
      pw     = 700
      expandX = true
      expandY = true
    }

    Box {
      expandY = True
      expandX = True
       title  = "Sanders Classification"
       alignX = Center
      ComboBox StringUtils.string1 {
      comboField = Sanders_classification
      editable = False
       width = 512
      }}
    Box  {
      title = "Measure largest step and Gap"
      alignX = Center
      width  = 512

      expandX = True
      expandY = True
      Field CSOInfo_Gap.csoLength {
       title  = "Largest Gap in mm"
       width  = 512
       alignX = Center
      }
      Field CSOinfo_Step.csoLength {
       title  = "Largest Step in mm"
       width  = 512
       alignX = Center
      }
      Button CSOListContainer.clear {
       title   = "Delete all lines"
       expandX = True
       expandY = True
      }
    }
   Box {
     Button RunPythonScript.execute {
      title   = "Save Data"
      expandX = True
      expandY = True
     }
   }
  }
 }
}
}
```

MEVISLAB Graphical User Interface (GUI)

A user-friendly GUI was developed, providing surgeons with an intuitive interface for performing manual measurements and assigning a Sanders classification to each patient. The interface also included an instructions section to guide users through the measurement process, further streamlining the evaluation procedure. The files uploaded in the GUI were the NIfTI files of the CT scans, which were resliced parallel

to the joint surfaces of the subtalar joint.

Figure 12: GUI used for manual measurements

Appendix H – distance plots

Plots for all 44 patients from the training dataset

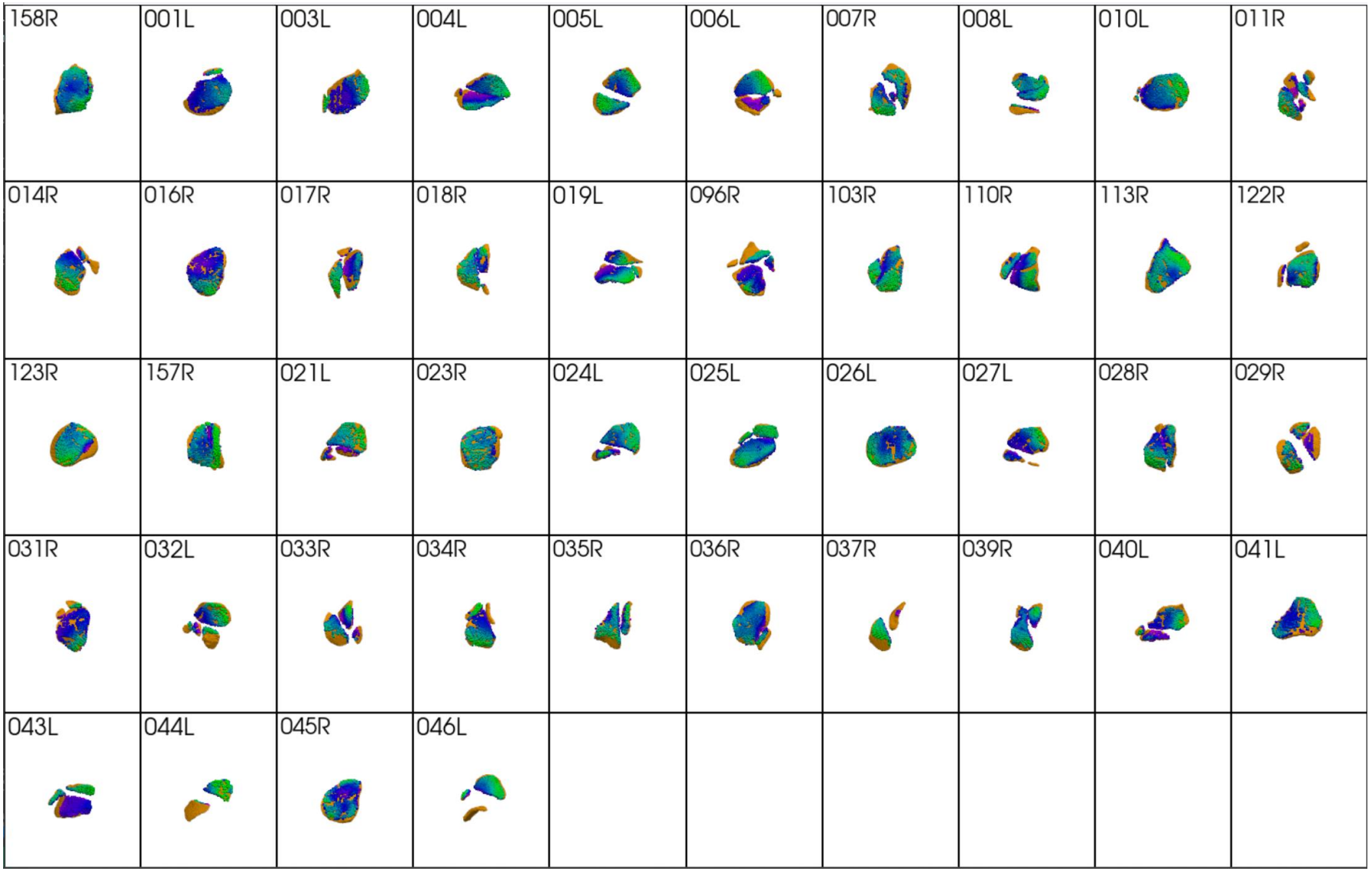


Figure 13: distance plots of the 44 patients from the training dataset. The points are color-coded according to their distance from the PTF, transitioning smoothly from green for the closest points, to purple for the most distant points. The PTC surface mesh is visualized in orange.

Plots for 10 patients from the external validation dataset – segmentations by nnU-Net

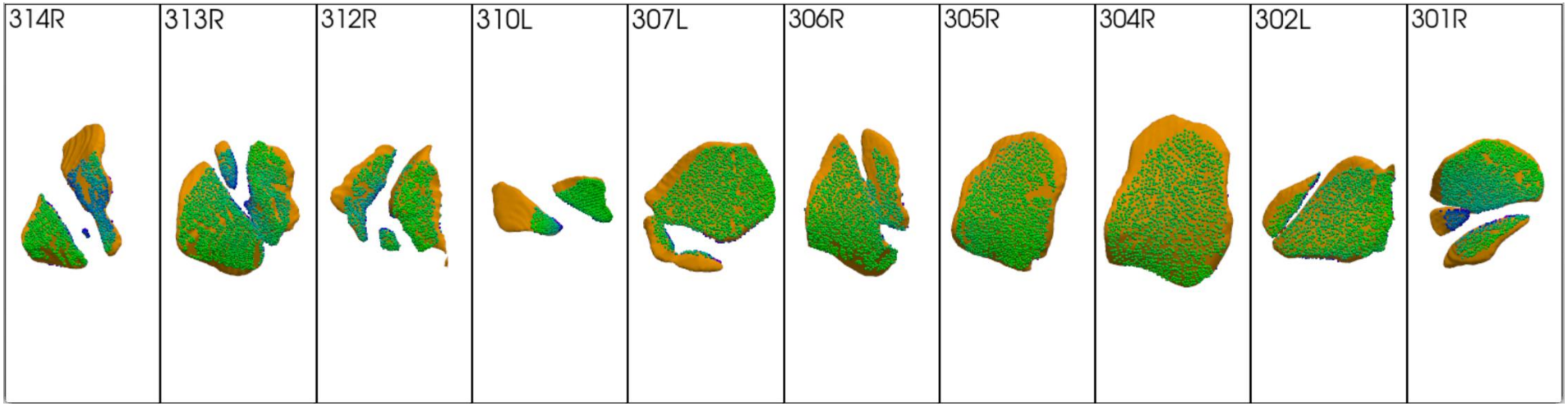


Figure 14: distance plots of the 10 patients from the external validation dataset, where the segmentations are made by nnU-Net. The points are color-coded according to their distance from the PTF, transitioning smoothly from green for the closest points, to purple for the most distant points. The PTC surface mesh is visualized in orange.

Plots for 10 patients from the external validation dataset – segmentations manual

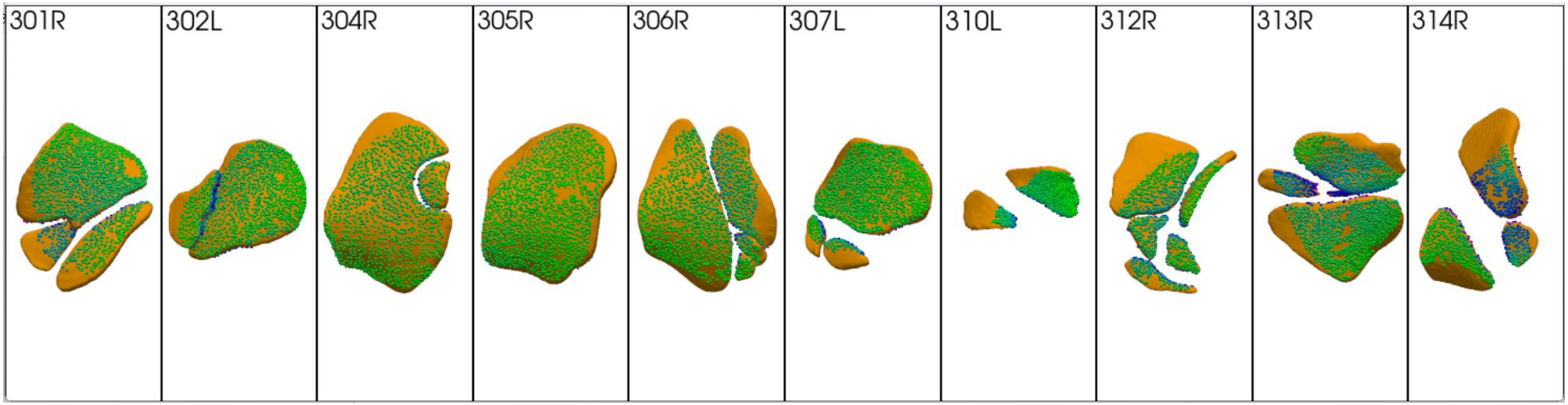


Figure 15: distance plots of the 10 patients from the external validation dataset, where the segmentations are made manually. The points are color-coded according to their distance from the PTF, transitioning smoothly from green for the closest points, to purple for the most distant points. The PTC surface mesh is visualized in orange.

Plots for 23 patients from the external validation dataset – segmentations by nnU-Net

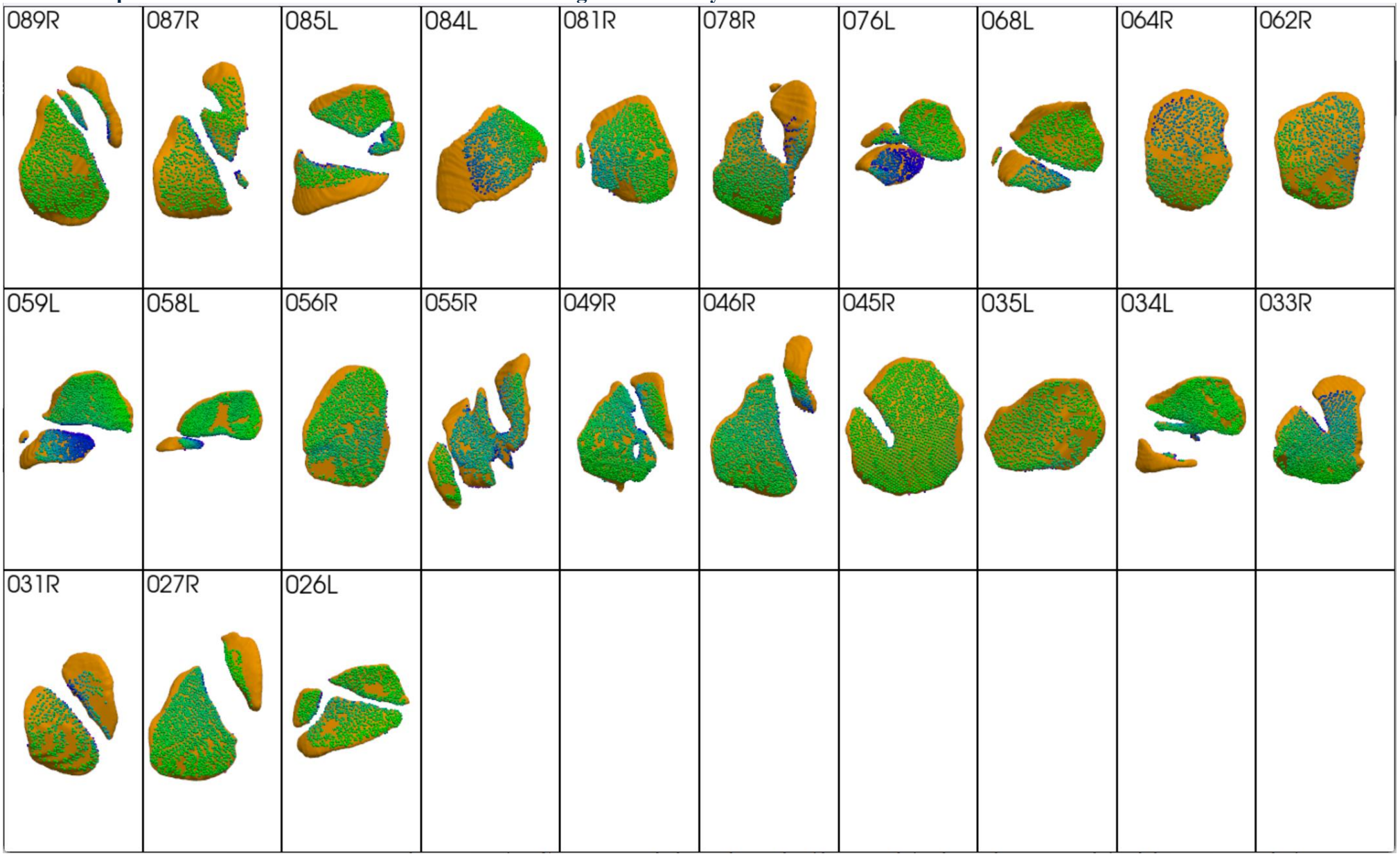


Figure 16: distance plots of the 23 patients from the external validation dataset, where the segmentations are made by nnU-Net. The points are color-coded according to their distance from the PTF, transitioning smoothly from green for the closest points, to purple for the most distant points. The PTC surface mesh is visualized in orange.

Appendix I – Gap area plots

Gap area plots for all 44 patients from the training dataset

Figure 17: The intersection points (red points) of the rays from the anterocranial convex hull (green) on the posterocaudal convex hull (light blue) and the PTC fragment surfaces (yellow) were plotted for all 44 patients from the training data set.

Gap area plots for 10 patients from the external validation dataset – segmentations by nnU-Net

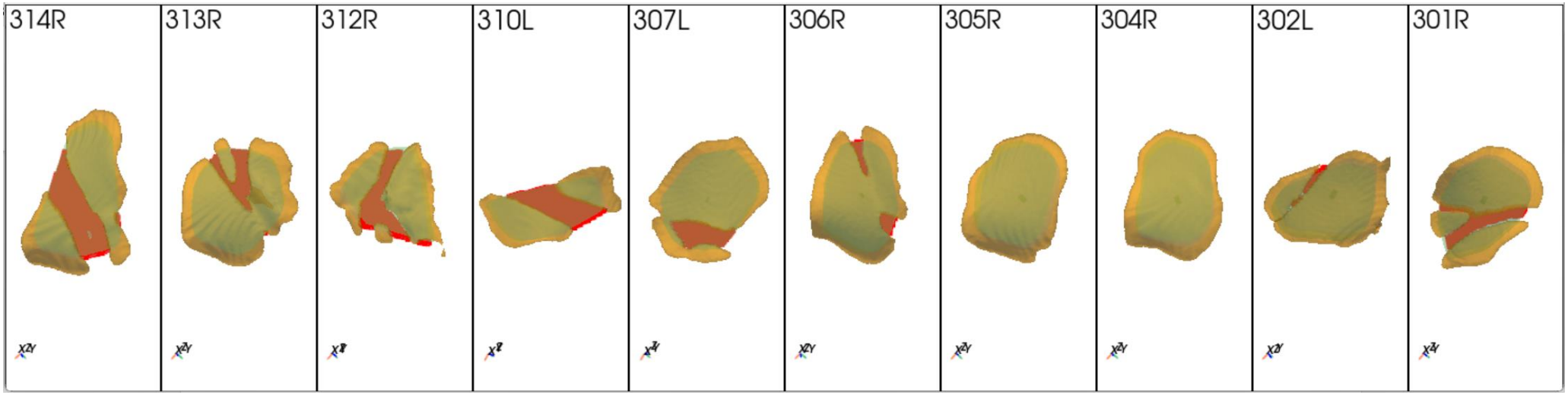


Figure 18: The intersection points (red points) of the rays from the anterocranial convex hull (green) on the posterocaudal convex hull (light blue) and the PTC fragment surfaces (yellow) were plotted for 10 patients from the external validation dataset, where segmentations were created by nnU-Net.

Gap area plots for 10 patients from the external validation dataset – segmented manually

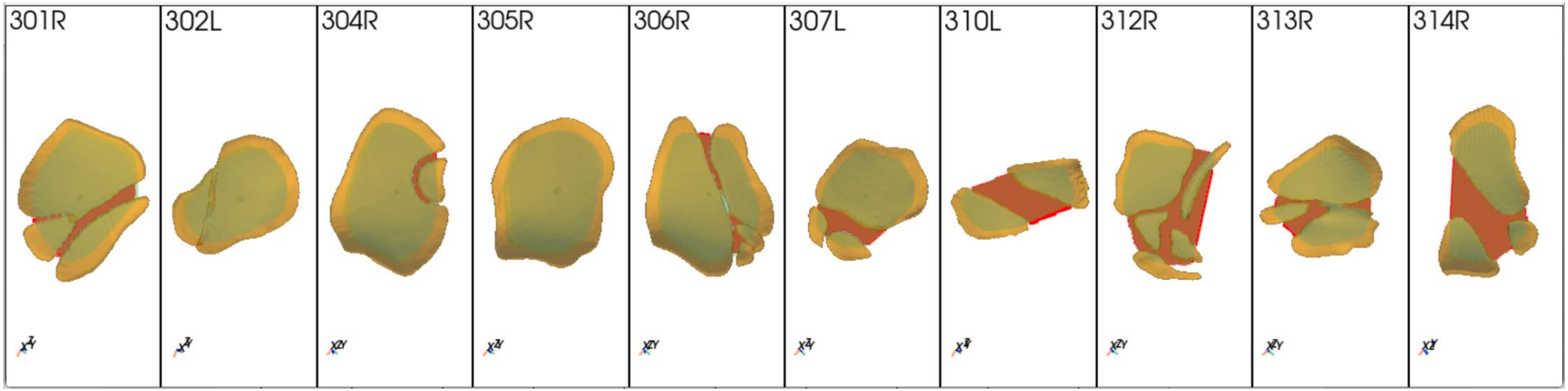


Figure 19: The intersection points (red points) of the rays from the anterocranial convex hull (green) on the posterocaudal convex hull (light blue) and the PTC fragment surfaces (yellow) were plotted for 10 patients from the external validation dataset, where segmentations were created manually.

Gap area plots for 23 patients from the external validation dataset – segmentations by nnU-Net

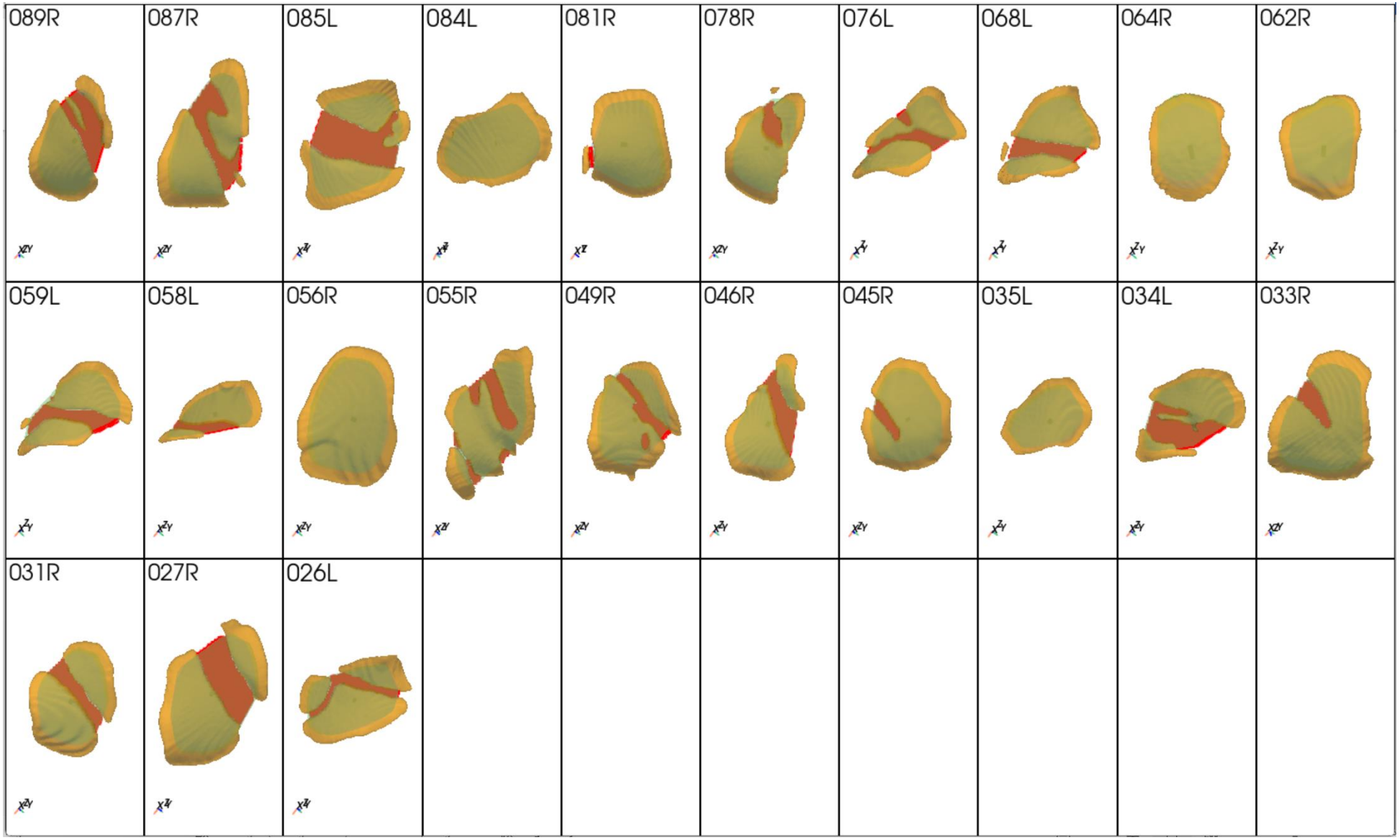
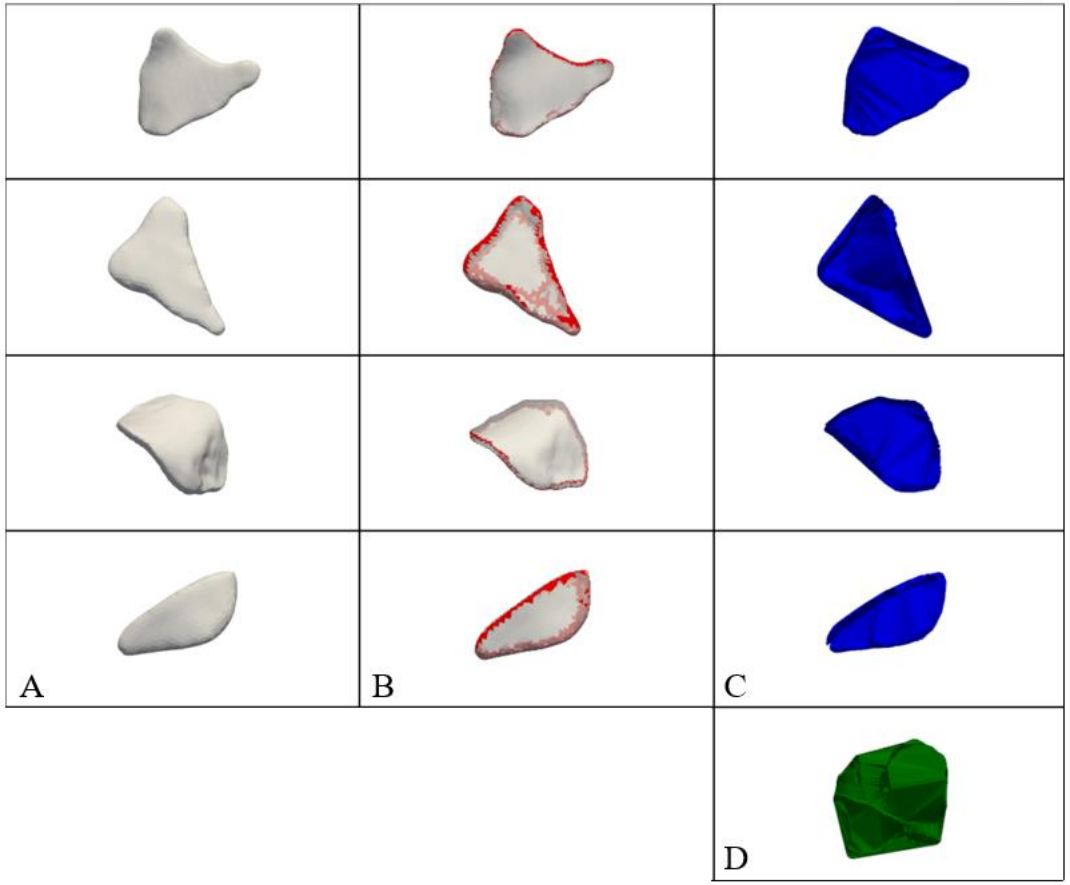


Figure 20: The intersection points (red points) of the rays from the anterocranial convex hull (green) on the posterocaudal convex hull (light blue) and the PTC fragment surfaces (yellow) were plotted for 23 patients from the external validation dataset, where segmentations were created by nnU-Net

Appendix J – Example patient: fragment and fracture area visualization



A = created smoothed STL files per fragment of example patient from training set, B = fragments including boundary mesh visualized in red, C = fragment surface area after Delaunay triangulation, D = surface area of all fragments combined after Delaunay triangulation

Appendix K – Maximal step-off and maximal gap visualization

Maximal step-off and maximal gap visualization of all 44 patients from training dataset

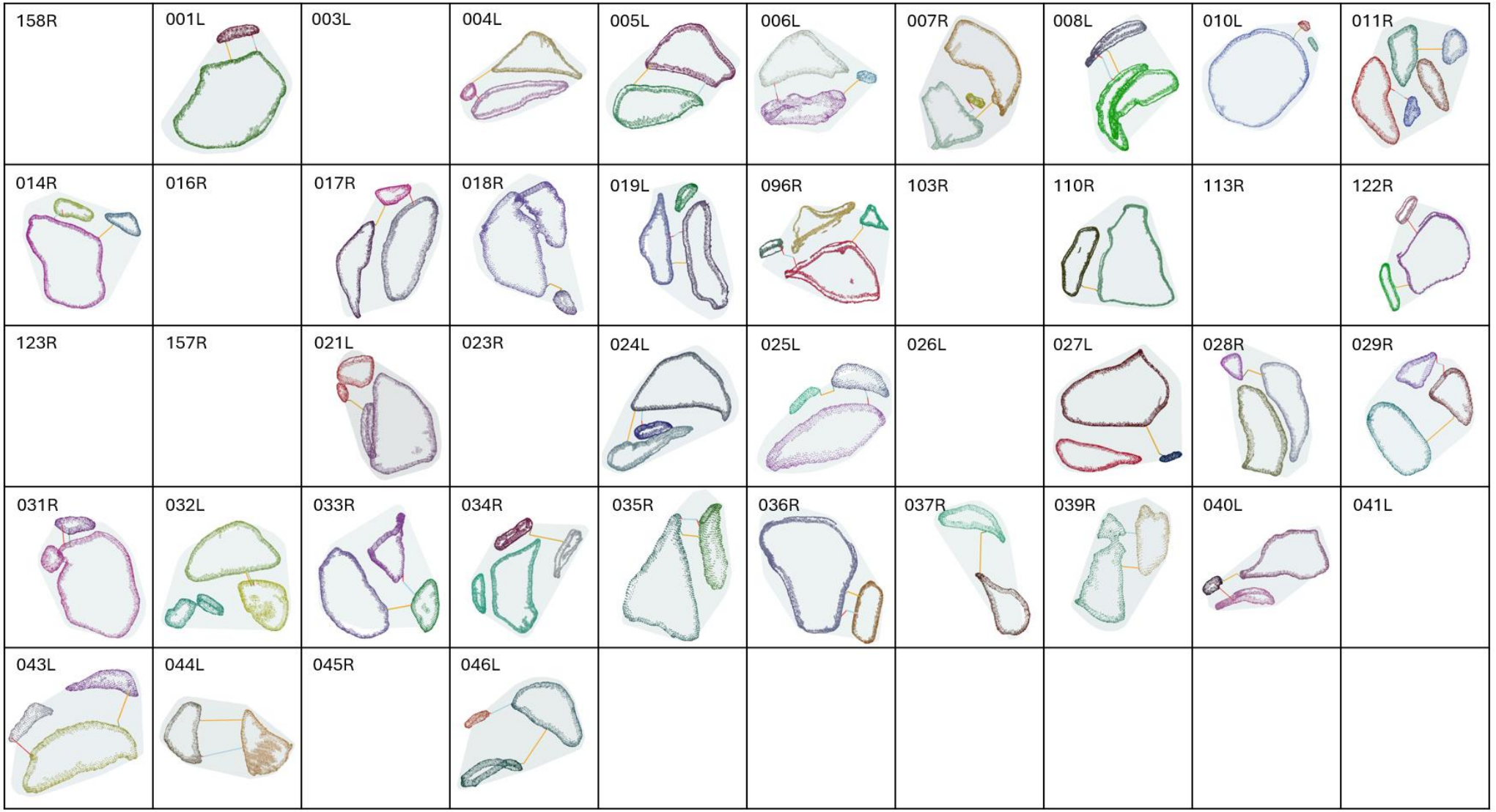


Figure 21: Maximal step-off (red lines) and maximal gap (orange line) for all 44 patients in the training dataset. When only one line is shown, a single mutual closest point pair represents both the maximal step-off and maximal gap. For patients without measurements, only one fragment was present, preventing maximal step-off or gap calculations.

Maximal step-off and maximal gap visualization of 10 patients from external validation dataset – segmented manually

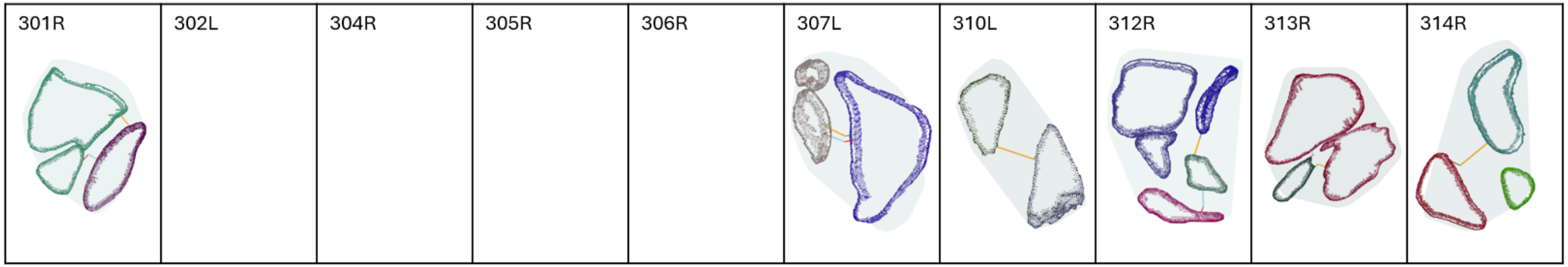


Figure 22: Maximal step-off (red lines) and maximal gap (orange line) for 10 patients in the external validation dataset where segmentation were created manually. When only one line is shown, a single mutual closest point pair represents both the maximal step-off and maximal gap. For patients without measurements, only one fragment was present, preventing maximal step-off or gap calculations.

Maximal step-off and maximal gap visualization of 10 patients from external validation dataset – segmented by nnU-Net

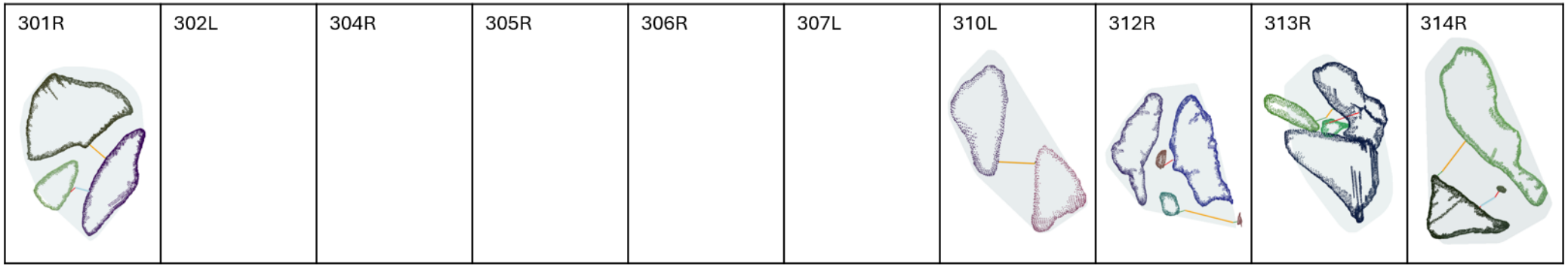


Figure 23: Maximal step-off (red lines) and maximal gap (orange line) for 10 patients in the external validation dataset where segmentation were created by nnU-Net. When only one line is shown, a single mutual closest point pair represents both the maximal step-off and maximal gap. For patients without measurements, only one fragment was present, preventing maximal step-off or gap calculations.

Maximal step-off and maximal gap visualization of 23 patients from external validation dataset – segmented by nnU-Net

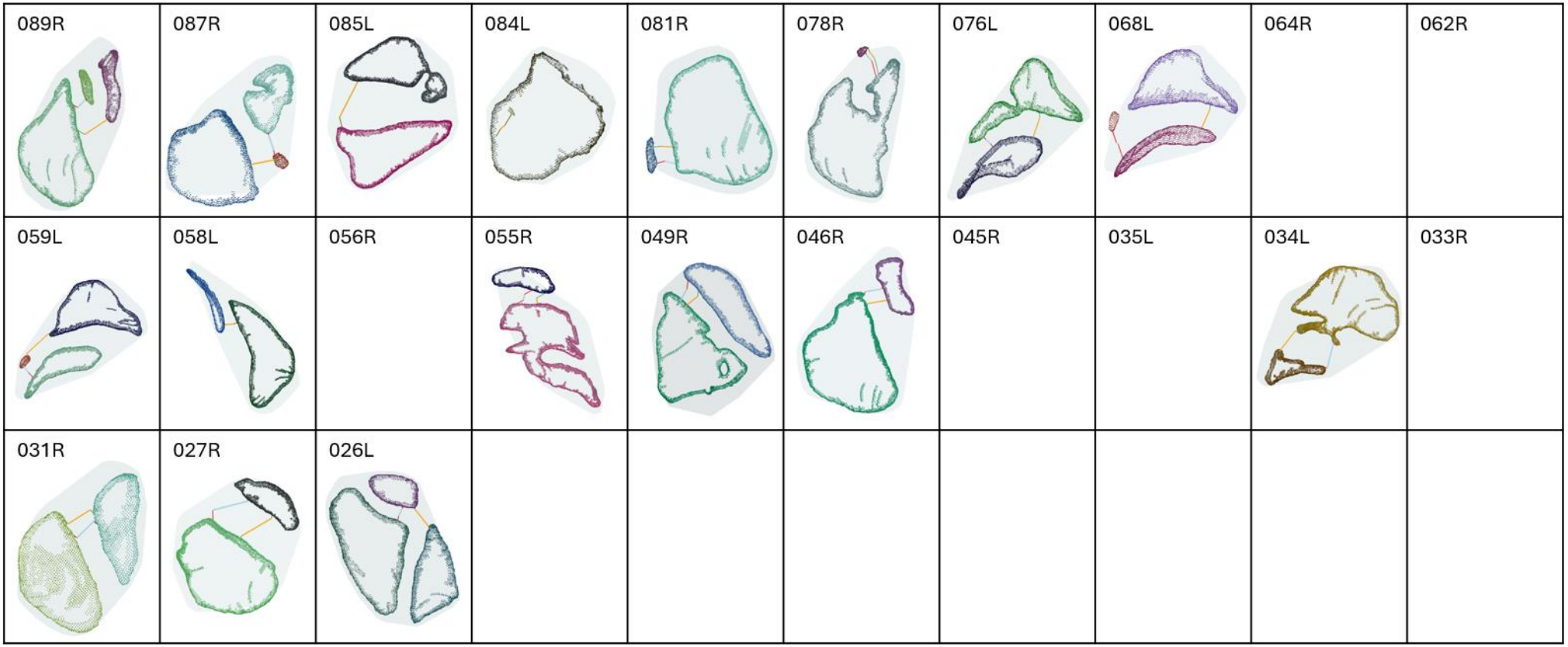


Figure 24: Maximal step-off (red lines) and maximal gap (orange line) for 23 patients in the external validation dataset where segmentation were created by nnU-Net. When only one line is shown, a single mutual closest point pair represents both the maximal step-off and maximal gap. For patients without measurements, only one fragment was present, preventing maximal step-off or gap calculations