3D MODELS OF BONE FROM CT IMAGES BASED ON 3D POINT CLOUDS

GENERATION AND SEGMENTATION OF 3D MODELS OF BONE FROM CT IMAGES BASED ON 3D POINT CLOUDS

By ELYSE RIER, B.ENG.

A Thesis Submitted to the School of Graduate Studies in Partial Fulfilment of the Requirements for the Degree Master of Applied Science

Descriptive note

McMaster University MASTER OF APPLIED SCIENCE (2021), Hamilton, Ontario

(School of Biomedical Engineering)

TITLE: Generation and segmentation of 3D models of bone from CT images based on 3D point clouds

AUTHOR: Elyse Rier, B.ENG. (McMaster University)

SUPERVISOR: Dr. Gregory Wohl

PAGES: xiv, 208

Lay abstract

The creation of 3D models of bone from CT images has become popular for education, surgical planning, and the design of implants. Software is available to convert CT images into 3D models but can be expensive and technically taxing. The purpose of this project was to develop a process to allow surgeons to create and interact with models from imaging data. This project applied a threshold to binarize a set of CT images, extracted the edges using a Canny Edge detector, and used the edge pixels to create a 3D point cloud. The 3D point cloud was then converted to a mesh object. A user interface was implemented that allowed the selection of portions of the model and a new 3D model to be created from the selection. The process can be improved by improving the quality of the mesh output and adding features to the user interface.

Abstract

The creation of 3D models of bone from CT images has become popular for surgical planning, the design of implants, and educational purposes. Software is available to convert CT images into 3D models of bone, however, these can be expensive and technically taxing. The goal of this project was to create an open-source and easy-to-use methodology to create 3D models of bone and allow the user to interact with the model to extract desired regions. The method was first created in MATLAB and ported to Python. The CT images were imported into Python and the images were then binarized using a desired threshold determined by the user and based on Hounsfield Units (HU). A Canny edge detector was applied to the binarized images, this extracted the inner and outer surfaces of the bone. Edge points were assigned x, y, and z coordinates based on their pixel location, and the location of the slice in the stack of CT images to create a 3D point cloud. The application of a Delaunay tetrahedralization created a mesh object, the surface was extracted and saved as an STL file. An add-on in Blender was created to allow the user to select the CT images to import, set a threshold, create a 3D mesh model, draw an ROI on the model, and extract that region based on the desired thickness and create a new 3D object. The method was fully open-sourced so was inexpensive and was able to create models of a skull and allow the segmentation of portions of that mesh to create new objects. Future work needs to be conducted to improve the quality of the mesh, implement sampling to reduce the time to create the mesh, and add features that would benefit the end-user.

Acknowledgements:

I would like to thank my supervisor, Dr. Wohl, for his continued support over the past two years through brainstorming, a continuous outlet for ideas and support, and motivation. I would also like to thank the rest of my committee, Dr. Noseworthy and Dr. Shirani. From the beginning of this project, Dr. Shirani has always made himself available for meetings to discuss the algorithms developed and was very involved in the fundamental beginning of the project. Throughout the whole project, Dr. Noseworthy has answered all of my questions about CT images to ensure the method of extracting the point cloud was done correctly. Lastly, I would like to acknowledge my support system as a whole, whether it be friends, family, or roommates, without these people, who always listened to me vent my frustration at difficult times throughout the project, this project would not have been successful.

Table of Contents:

Chapter 1: Introduction..1
Chapter 2: Background Information..4
2.1: Creation of 3D Models from Medical Images...4
2.1.1: Image acquisition...4
2.1.2: Image processing..7
2.1.3: Methods of Mesh Generation...12
2.2: Current Solutions to Create 3D models from CT Images....................................17
2.2.1: Licensed Software...17
2.2.2: License-free Software...18
2.3: 3D Point Clouds..20
2.3.1: 3D Point Cloud Definition..20
2.3.2: Obtaining 3D Point Clouds...20
2.3.3: Applications of 3D Point Clouds..21
2.4: Mesh File Format for Additive Manufacturing..22
2.5: Method for Nearest Neighbour Search...22
Chapter 3: Methods and Results...24
3.1 Chapter Overview..24
3.2 Sample Data Preparation...25
3.3 Creation of Point Cloud...26
3.3.1 Function: *open_CT*..27
3.3.2 Function: *thresh_edge_CT*...27
3.3.3 Function: *extract_PT_Cloud*...31
3.4 Down Sampling Point Cloud Algorithms...33
3.4.1 Pseudo-Uniform Sampling...33
3.4.2 Targeted Sampling...37
3.5 Mesh Creation from Point Cloud...44
3.5.1 Testing on Models...45
3.5.2 Testing on Point Clouds from CT Images of Head...60

3.6 Processing Time in Python 64
3.7 Comparison of Meshes 66
3.7.1 Comparison of Point Clouds 66
3.7.2 Comparison by Volume Measures 67
3.7.3 Residual Volume 72
3.7.4 Distance Measures Between Meshes 73
3.8 Comparison of Sampling Algorithms 78
3.8.1 Design of Experiments 78
3.6.2 DOE Results 82
3.8.3 Comparing Sampling using a Simplified Shape 95
3.8.4 Processing Time for Pseudo-Uniform Sampling and Full Point Cloud 106
3.9 User Interface 108
3.9.1 Blender Interface – Buttons 108
3.9.2 Demonstration of Process in Blender 130
3.9.3 Processing Time in Blender 136
Chapter 4: Discussion and Conclusion 138
4.1 Implications of Qualitative Data 138
4.2 Implications of Quantitative Data 139
4.3 Limitations 144
4.3.1 Basis for Comparison/Error Analysis 144
4.3.2 Mesh Quality 146
4.4 Areas of Improvement and Future Work 148
4.4.1 Sampling Improvement 148
4.4.2 Improvement of Mesh Quality 151
4.4.3 Thresholding of CT Images 152
4.4.4 Additions/Improvements to User Interface 153
4.4.5 Error Analysis 155
4.4.6 3D Printing Testing 155
4.4.7 Comparison to Other Methods 156
4.5 Conclusion 156
Bibliography 158

Appendix A: Python Package Versions........ 170
Appendix B: Histograms and Heat Maps for Comparison of Meshes for Sampling Algorithms 171
Appendix C: CT Acquisition Parameters........ 182
Appendix D: Python Code........ 183
Appendix E: Blender Add-on code........ 196

List of Figures:

Chapter 2:

Figure 2. 1: Example of some possible triangle configurations for Marching Cubes (Boris n.d.) 14
Figure 2. 2: 2D demonstration of the BPA. 15
Figure 2. 3: Demonstration of a 2D Delaunay triangulation 17

Chapter 3:

Figure 3. 1: Images from Skull1........ 26
Figure 3. 2: Images from Skull2........ 26
Figure 3. 3: Images from a dataset of lower limbs........ 26
Figure 3. 4: Depiction of the process of the Canny edge detector........ 30
Figure 3. 5: Demonstration of the directions chosen to create the point cloud........ 32
Figure 3. 6: Example of point cloud for Skull2........ 33
Figure 3. 7: Depiction of the effects of sampling a point set........ 36
Figure 3. 8: Demonstration of boundaries of the grid for the targeted sampling........ 39
Figure 3. 9: Depiction of two curves used of which the angular similarity was calculated........ 44
Figure 3. 10: Two models used to test mesh methods........ 46
Figure 3. 11: Output of BPA mesh function on line stack........ 49
Figure 3. 12: Output of BPA for line stack for larger radii list........ 50
Figure 3. 13: Results of Poisson method on the line test model........ 52
Figure 3. 14: Output from TetraMesh for line model........ 53
Figure 3. 15: Output of 2D Delaunay from PyVista applied to a line model........ 54
Figure 3. 16: Output of 3D Delaunay from PyVista applied to a line model........ 55
Figure 3. 17: Output of 3D Delaunay from PyVista applied to a cylindrical model........ 56
Figure 3. 18: Output of PyVista's Delaunay 3D for cylinder mesh with 1.0 mm spacing........ 58
Figure 3. 19: Output of PyVista's Delaunay 3D for cylinder mesh with 5.0 mm spacing........ 59
Figure 3. 20: Output of Delaunay 3D from PyVista with an alpha value of 20.0........ 60
Figure 3. 21: Output mesh using point cloud from Skull1........ 62
Figure 3. 22: Output mesh using point cloud from Skull2........ 62
Figure 3. 23: Resultant meshes created for images of the head and lower limbs........ 66
Figure 3. 24: Dense point cloud of a cube with 100 slices and slices spacing of 0.352mm........ 69
Figure 3. 25: Meshes for 6 different configurations of cubes for volume calculations. 70
Figure 3. 26: Edges chosen during the custom mesh comparison algorithm........ 77

Figure 3. 27: Representation of pixels that create a grid during the targeted sampling. 80
Figure 3. 28: Relationship between angular similarity range and percentage sampled. 81
Figure 3. 29: Results of mesh comparison between the full mesh and 10% pseudo-uniform. 84
Figure 3. 30: Results of mesh comparison between the full mesh and 90% pseudo-uniform. 85
Figure 3. 31: Results of comparison of meshes between the full mesh and tar1. 86
Figure 3. 32: Results of comparison of meshes between the full mesh and tar8. 87
Figure 3. 33: Summary of the linear model for the average of the average distance for the targeted sampling. 93
Figure 3. 34: Summary of the linear model for percentages sampled for the targeted sampling. 94
Figure 3. 35: Simplified shape to test the pseudo-uniform and targeted sampling methods. 96
Figure 3. 36: Simplified object with Canny edge detector applied. 97
Figure 3. 37: Resultant point cloud extracted from simplified images. 97
Figure 3. 38: Results for pseudo-uniform sampling on simplified shape. 99
Figure 3. 39: Results of sampling with targeted sampling on simplified shape. 100
Figure 3. 40: Point clouds for simplified shape, A. full point cloud, B. 10% pseudo-uniformly sampled, and C. 10% randomly sampled. 105
Figure 3. 41: Panel Created for Blender Interface. 109
Figure 3. 42: Choosing a file directory and saving it in the PropertyGroup. 111
Figure 3. 43: Flow chart depicting the process used to render a surface mesh in Blender. 114
Figure 3. 44: Depiction of creation of vectors a and b used to calculate the normal of selected vertices. 119
Figure 3. 45: Depiction of the first stage of selection of inner points. 123
Figure 3. 46: Refinement stage for selection of inner points when the largest component of normal was in the x-direction. 128
Figure 3. 47: Blender interface before opening CT images and rendering mesh. 131
Figure 3. 48: Demonstration of Blender interface. 132
Figure 3. 49: Examples of ROI selection made by the user and the resulting STL. ... 133
Figure 3. 50: Selection of portion of the frontal bone and resulting STL. 134
Figure 3. 51: Blender interface for a set of CT images of the lower limbs. 135
Figure 3. 52: Mesh used to measure the execution time in Blender. 136

Chapter 4:

Figure 4. 1: STL files imported into Meshmixer. 147

Appendix B:

Figure B 1: Heat map (A) and histogram (B) for the pseudo-uniform sampling of 20%.......... 171
Figure B 2: Heat map (A) and histogram (B) for the pseudo-uniform sampling of 30%.......... 172
Figure B 3: Heat map (A) and histogram (B) for the pseudo-uniform sampling of 50%.......... 173
Figure B 4: Heat map (A) and histogram (B) for the pseudo-uniform sampling of 70%.......... 174
Figure B 5: Heat map (A) and histogram (B) for targeted sampling, tar2. 175
Figure B 6: Heat map (A) and histogram (B) for targeted sampling, tar3. 176
Figure B 7: Heat map (A) and histogram (B) for targeted sampling, tar4. 177
Figure B 8: Heat map (A) and histogram (B) for targeted sampling, tar5. 178
Figure B 9: Heat map (A) and histogram (B) for targeted sampling, tar6. 179
Figure B 10: Heat map (A) and histogram (B) for targeted sampling, tar7. 180
Figure B 11: Heat map (A) and histogram (B) for targeted sampling, tar9. 181

List of Tables:

Chapter 3:

Table 3. 1: Parameters used to create the various meshes involving the cube.......... 69
Table 3. 2: Volume Measurements of five created meshes of cubes.. 70
Table 3. 3: Standard Order table for the experiments to test targeted sampling. 82
Table 3. 4: Average distances for uniform sampling.......... 88
Table 3. 5: Overall Average distances for targeted sampling.......... 88
Table 3. 6: Difference between averages for pseudo-uniform and targeted sampling. . 91
Table 3. 7: Percentage of grids in each angular similarity range for the targeted sampling experiments.......... 102
Table 3. 8: Time to sample, create the mesh, and total time for full and uniformly sampled. 107

Appendix A:

Table A. 1: Python package versions used throughout this project. 170

Appendix C:

Table C. 1: CT Acquisition Parameters for CT images used throughout this project. 182

List of all Abbreviations and Symbols

ACR: AMERICAN COLLEGE OF RADIOLOGY

API: APPLICATION PROGRAMMING INTERFACE

ASCII: AMERICAN STANDARD CODE FOR INFORMATION INTERCHANGE

BPA: BALL PIVOTING ALGORITHM

CAD: COMPUTER-AIDED DESIGN

CAE: COMPUTER-AIDED ENGINEERING

CAM: COMPUTER-AIDED MANUFACTURING

CSG: CONSTRUCTIVE SOLID GEOMETRY

CT: COMPUTED TOMOGRAPHY

DICOM: DIGITAL IMAGING AND COMMUNICATIONS IN MEDICINE

DOE: DESIGN OF EXPERIMENT

HU: HOUNSFIELD UNIT

JPG: JOINT PHOTOGRAPHIC EXPERTS GROUP

LIDAR: LIGHT DETECTION AND RANGING

MDCT: MULTIDETECTOR COMPUTED TOMOGRAPHY

MITK: MEDICAL IMAGING INTERACTION TOOLKIT

MM: MILLIMETERS

MRI: MAGNETIC RESONANCE IMAGING

NEMA: NATIONAL ELECTRICAL MANUFACTURER'S ASSOCIATION

RAM: RANDOM ACCESS MEMORY

RP: RAPID PROTOTYPING

ROI: REGION OF INTEREST

STL: STEREOLITHOGRAPHY

TCIA: THE CANCER IMAGING ARCHIVE

VTK: VISUALIZATION TOOLKIT

σ: SIGMA, STANDARD DEVIATION OF GAUSSIAN FILTER

α: ALPHA, RADIUS FOR ALPHA SHAPES AND RADIUS USED IN CREATING DELAUNAY MESH

Declaration of Academic Achievement

I, Elyse Rier, completed preliminary research, wrote, and tested the code used throughout this project, designed experimental procedures to test the different algorithms used, performed any statistical analysis described, and am solely responsible for the content in this thesis. My supervisor, Dr. Gregory Wohl provided expertise in 3D modelling, provided CT images to develop the work, supervised, and guided this work. Dr. Shahram Shirani guided the development of the algorithms used throughout this work. Dr. Michael Noseworthy provided guidance on the properties of CT images and provided CT images to attempt an error analysis.

Chapter 1: Introduction

Modelling of medical imaging data has widespread uses including presurgical planning, the design and production of implants, visualization of a specific anatomical region of interest (ROI), and for educational purposes (Bibb and Winder 2010; Noser et al. 2011; Parthasarathy 2014). The models extracted from medical images can be 3D printed providing the advantage of physical interaction with the model rather than viewing the object on a 2D screen (Rengier et al. 2010). These methods are commonly applied to head and neck reconstruction, and neurosurgery (Bibb and Winder 2010). An operation such as craniofacial surgery is complex so computer-aided design (CAD) or computer-aided manufacturing (CAM) help create precise custom implants during planning, and models can be included in intraoperative navigation (Parthasarathy 2014). Bone models used for surgical planning and implant design must accurately represent the patient's anatomy to ensure that any implant produced based on the model adequately fits the defect (Gelaude, Sloten, and Lauwers 2008; Noser et al. 2011). With advances in metal additive manufacturing and bioprinting, future applications of medical imaging data could also include the creation of biomimetic structures (Bishop et al. 2017; Seol et al. 2014) and implants (Javaid and Haleem 2018; Sing et al. 2016).

Physical models can be created by following three steps: 1) image acquisition, 2) image processing which includes segmentation and the creation of a triangular mesh, and 3) 3D printing of the model (Huotilainen et al. 2014; Marro, Bandukwala, and Mak 2016;

Rengier et al. 2010). These steps and the commonly used modalities will be described in more depth in the background information.

The use of 3D printing models from medical images for designing implants and planning is expected to grow. These models are limited based on training, available equipment, and guidelines for radiologists (Mitsouras et al. 2015). To create the models, the user needs technical knowledge and skill to create the 3D printable files. Clinically, there are a limited number of software applications designed specifically for 3D medical modelling (Mitsouras et al. 2015). Software often is complex and requires human, technical, and financial resources (Wallner et al. 2019).

The purpose of my project was to develop a methodology to create 3D models of bone from computed tomography (CT) images that is inexpensive and requires little technical knowledge. The functionality of the method aimed to aid surgical planning and design of implants by allowing the user to easily manipulate the 3D model to extract a desired region from the 3D model to create a model of the desired segment separate from the original model. To accomplish this there were goals that the methodology had to accomplish:

1) Allow the user to import CT images
2) Create a 3D model of the bone based on a user-defined threshold
3) Create a model of specified segments
4) Be accessible in terms of cost and ability to be used with little technical knowledge

The methodology centered around creating a 3D mesh model of the bone from the CT images. This was created by importing the CT images and creating a 3D point cloud to represent the inner and outer surfaces of the bone, the 3D point cloud was then converted into a mesh model.

This thesis describes the process from which the mesh model was created and the development of the user interface. A review of common techniques to create 3D models from medical images, the current software available, a review of the concept of 3D point clouds used for this project, and common file formats are introduced through the background information. The steps and resultant models produced by the methodology are discussed in the methodology and results chapter. The thesis ends with a discussion of the implications of the results, the limitations of the method, the improvements that can be made, and future work to be done on the project.

Chapter 2: Background Information

This chapter includes a review of the process to create 3D models from medical images including the image acquisition, and the image processing (segmentation and the generation of the 3D mesh objects). A discussion of existing software is done to highlight the need for inexpensive and easy-to-use techniques. As a 3D point cloud was initially used to define the inner and outer surface of the bone in this project, applications of point clouds are also covered.

2.1: Creation of 3D Models from Medical Images

2.1.1: Image acquisition

The first stage in creating a 3D model, image acquisition, can be done using CT or magnetic resonance imaging (MRI). When making models of bone, CT is a common choice and considered the gold standard because it has a high tissue to bone contrast creating images with a good depiction of the bone (Schmutz et al. 2014; Schmutz, Wullschleger, and Schuetz 2007; Stephen et al. 2020) and produces models with high geometric accuracy (Rathnayaka et al. 2012). In addition, CT imaging is fast, inexpensive, more readily available, and is less affected by motion than MRI (Stephen et al. 2020). One of the main disadvantages of using CT imaging is that it exposes the patient to high levels of radiation (Rathnayaka et al. 2012; Stephen et al. 2020). MRI is another technique that can be used, these images have a lower bone contrast and field distortions can cause image distortions, but the patient is not exposed to any radiation (Stephen et al. 2020).

CT images are composed of 2D cross-sectional images of the body. They are created by rotating an X-ray tube around a patient, the X-rays transmitted through the object are measured by a ring of detectors in the gantry that surrounds the patient (Das 2015). Multiple X-ray projections are used to reconstruct the 2D images. The intensity of the transmitted radiation is used to determine the density or attenuation of the tissue in each pixel, and attenuation values are associated with each voxel. Average attenuation of voxels allows a value to be assigned to each pixel with units of Hounsfield Units (HU) (Das 2015). The attenuation of pixels is replaced by a CT number which has units of HU, the CT number is calculated as:

$$CT\ number = K(u_{voxel} - u_{water})/u_{water},$$

where: u is attenuation and K is a constant that is standardized to 1000 (Goldman 2007).

Digital imaging and communications in medicine (DICOM) files are used by radiologists and for medical imaging including CT and MRI. The standard was introduced through a collaboration between the American College of Radiology (ACR) and the National Electrical Manufacturer's Association (NEMA) (Graham, Perriss, and Scarsbrook 2005). DICOM allows for the integration of devices from different manufacturers and allows for the transmission of images between systems. DICOM files contain two components: the header, and the image data (Graham, Perriss, and Scarsbrook 2005; Varma 2012). The header portion of the file can be accessed through tags; these tags are grouped based on the information they contain (Varma 2012). The header contains and allows the user to access information such as information on the patient, acquisition parameters, and image properties. The header is then followed by the pixel data contained in the image (Graham, Perriss, and Scarsbrook 2005; Varma 2012). Although the use of DICOM images is standard practice in the field of medical imaging, the files cannot be opened directly by certain operating systems so specific software is required to view them, the files are large (i.e., 35 MB for CT images of the head), and may need to be compressed before they are transmitted (Graham, Perriss, and Scarsbrook 2005; Varma 2012).

During image acquisition, multiple parameters can affect the outcome of a generated 3D model whether the scanning is done through a conventional CT or multidetector computed tomography (MDCT). Rengier (Rengier et al. 2010) notes that MDCT using isotropic voxels and a slice thickness less than 1 mm can be applied for rapid prototyping (RP) to help reduce partial volume effects. Slice thickness can affect the 3D model that is produced from the CT images as larger slice thicknesses can create a step effect in the models (Bibb and Winder 2010), and the required thickness is dependent on the anatomical complexity of the area being imaged (Marro, Bandukwala, and Mak 2016). In a technical note by Bibb and Winder (Bibb and Winder 2010) it is recommended that a slice thickness of 2 mm is sufficient when creating models of larger bones such as long bones, while areas that are more complex such as the maxillofacial area requires a smaller slice thickness between 0.5 and 1 mm. While maximizing the data obtained by the CT scanner can be achieved by decreasing the slice thickness, this also increases the radiation exposure of the patient (Bibb and Winder 2010). In a pilot study conducted by Schmutz et al. (Schmutz, Wullschleger, and Schuetz 2007), the impact of the slice spacing on the geometry of 3D models of intact ovine cadaver long bones of the hind limb was investigated. Five different slice spacings were used, 0.5, 1, 2, 3, and 4 mm, the model created from the 0.5 mm spacing images was used as a reference and was verified against a model created using a 3D laser scanner. From this pilot study, it was determined that the geometric accuracy of the model of the bone decreased as the slice spacing increased. Deviations between models and the reference were more apparent in areas where there were changes in surface geometry relative to the axis of the scan, for example in the epiphyseal regions (Schmutz, Wullschleger, and Schuetz 2007). Using a multidetector-row helical CT, Shin (Shin et al. 2002) wanted to determine parameters that would create the best 3D images by varying three parameters, the detector row collimator width (for this method this parameter is equivalent to axial resolution or slice thickness), pitch, and the overlap of the slices during the reconstruction. The detector row collimation used was 1.25, 2.5, and 5.0 mm, the pitch used was 3.0 and 6.0, and the overlap used was 0%, 25%, and 50%. 3D image quality was rated by two radiologists, and it was determined that image quality increased with decreasing row collimation, decreasing pitch, and an increased overlap (Shin et al. 2002). Noise within the CT images is another factor that can cause inaccuracies in the 3D models as it distorts tissue boundaries and can cause the model to be rough (pixelated). Noise can be reduced by increasing the tube current while scanning, however, this increases the radiation dose (Bibb and Winder 2010).

2.1.2: Image processing

Once images are obtained, image processing may be done to remove artifacts, noise, and select certain regions of interest from the images. For this project, filters to remove noise and artifacts were not used, the following description of image processing focuses on some of the processes available to segment bone from CT images.

Kernels are used to sharpen or smooth images, there is a tradeoff between edges and noise, sharpening enhances the edges while increasing the noise, and smoothing reduces the sharp edges while decreasing noise (Bibb and Winder 2010). Removing noise and maintaining the features of the images is difficult, this may lead to missing structures limiting the analysis of the CT images (Diwakar and Kumar 2018). In the review (Diwakar and Kumar 2018) advantages of some of the noise reduction algorithms reviewed included an improvement in image quality, smoothing and the maintenance of tissue contrast, disadvantages of the algorithms in some cases included loss of small structures and a high computational time. Some of the disadvantages associated with the processing can later affect the quality of image segmentation through features that are inadvertently removed and the potential addition of artifacts (Iassonov, Gebrenegus, and Tuller 2009).

A common artifact in CT images is streak artifacts, these occur due to objects such as implants and dental fillings (Fakhar et al. 2018; X. Zhang, Wang, and Xing 2011), as these objects cause high attenuation of the X-ray beam during scanning (X. Zhang, Wang, and Xing 2011). Individuals such as (X. Zhang, Wang, and Xing 2011) have introduced methods to effectively remove metal artifacts while reducing noise and preserving contrast. Fakhar et al. (Fakhar et al. 2018) compared anatomical landmarks and bone measurements after a low and medium level of artifact removal was applied to CT images containing objects such as dental implants and fillings. After artifact removal, it was determined that large anatomical structures were still visible but, smaller structures can be affected so they are not as visible. Algorithms to remove metal artifacts can be limited because detail may be lost around the tissue-metal interface (Barrett and Keat 2004).

2.1.2.1: Types of Segmentation:

Segmentation is the process in which pixels or voxels of an area of interest are delineated from one another. This is necessary to create 3D models of desired areas from imaging modalities (Bell and Moore 2020). Many methods may be used to segment images, these include manual segmentation, thresholding, edge detection, and region growing.

Manual segmentation

One of the simplest methods of image segmentation is manual segmentation, this involves the user manually selecting or tracing an ROI to be segmented. Although straightforward, this method is time-consuming and subjective such that there is large variability between users (Rathnayaka et al. 2011).

Thresholding

The two types of thresholding to segment images that will be discussed here are global or single threshold segmentation and multi-threshold segmentation. Thresholding determines pixels in the images that are either kept or removed based on a chosen value (Marro, Bandukwala, and Mak 2016).

Global thresholding is one of the most common and straightforward segmentation methods used for the creation of cranial base models (Friedli et al. 2020), medical additive manufacturing (van Eijnatten et al. 2018), and in commercial 3D reconstruction software (Rathnayaka et al. 2011). Thresholding is accomplished by having the user visually select a threshold (Rathnayaka et al. 2011), the gray value chosen is used to create a new image from pixel values that are greater than the threshold (van Eijnatten et al. 2018) to delineate an ROI. Although simple and common, the selection of the threshold by the user affects the accuracy and the repeatability (Rathnayaka et al. 2011), and the method requires manual post-processing (van Eijnatten et al. 2018). A single threshold struggles to capture an entire ROI when the structure has varying pixel intensities depending on the location in the structure, for example, this is true for a long bone (Rathnayaka et al. 2011). When voxels contain more than one tissue the partial volume effect occurs, this can cause the ROI to be over- or underestimated. Artifacts, noise, and variations between the gray values of different scanning procedures can also affect the segmentation, these are not considered by global thresholding algorithms (van Eijnatten et al. 2018). The chosen threshold will affect the segmentation of the desired ROI, a threshold that is too large leads to a surface that is over smoothed such that sharp bony processes may be removed, while too low of a threshold can cause the surface to bridge gaps where the bone is not present such as bridging across joints (Stock et al. 2020). The choice of threshold is user-dependent however, Stock et al. (Stock et al. 2020) found that small deviations in the threshold utilized, considered to be deviations of less than 50 HU, cause minimal surface deviations.

One of the other types of thresholding is a local or multi-level threshold method. This method works similarly to the global threshold but, uses multiple thresholds after the images are split into different regions (van Eijnatten et al. 2018). Improvement of accuracy could be accomplished with more advanced thresholding techniques however, some of these are not available in commercial software packages (van Eijnatten et al. 2018).

Edge detection

The process of edge detection analyzes pixel gradients within an image, pixels with a gradient above a certain value are classified as edges, Canny edge detector is commonly used as it is fast and accurate (van Eijnatten et al. 2018). This method takes into account local intensity values so is an acceptable option when values vary between regions (van Eijnatten et al. 2018; Rathnayaka et al. 2011) and the method has high repeatability (Rathnayaka et al. 2011). Noise can cause pixels to be misidentified as edges, so post-processing may be required to segment the entire desired portion of bone from the image. This method may be combined with region growing (van Eijnatten et al. 2018).

Region growing

Region growing is used to group voxels of the same intensity (implying the same tissue or structure). The user manually chooses a seed voxel, and surrounding voxels are compared to the seed voxel. A homogeneity criterion is used to determine if the surrounding voxels should be classified as the same tissue as the seed voxel and if so, they are grouped (van Eijnatten et al. 2018). This method can help to fill gaps within the segmentation however, structures may be misconnected within the presence of noise and partial volume effects (van Eijnatten et al. 2018).

2.1.3: Methods of Mesh Generation

The final step in the process of creating a 3D model is to generate a mesh object from the segmented volume information. There are many different algorithms available to create mesh objects, some of these algorithms include Marching Cubes, Delaunay triangulation and tetrahedralization, Poisson surface reconstruction, and Ball Pivoting Algorithm. The latter three were the algorithms tested when choosing which algorithm could be applied to the extracted point cloud in this work.

Marching Cube

The Marching Cube algorithm is one of the most common algorithms used to extract an isosurface from volume data (Y. Zhang, Bajaj, and Sohn 2003) and is highly used to create a mesh from segmented medical images (Lobos and Rojas Moraleda 2013). Volume data is divided into cubes (Lobos and Rojas Moraleda 2013); the cubes consist of 8 pixels, 4 pixels on each of 2 adjacent slices (Lorensen and Cline 1987; Pagès, Sermesant, and Frey 2005). The vertices of the cube are compared to a value, the vertex is labelled based on if it is less than (outside the surface) or greater than (inside the surface) the user-specified value, and the labelling of the vertices is used to decide if the voxel intersects the surface (Lobos and Rojas Moraleda 2013; Lorensen and Cline 1987; Pagès, Sermesant, and Frey 2005). Triangles are created based on the intersection of the voxel with the surface. Triangles are determined based on a lookup table (Lorensen and Cline 1987; Pagès, Sermesant, and Frey 2005; Y. Zhang, Bajaj, and Sohn 2003), an 8-bit number that is the combination of values assigned to the 8 vertices of the cube acts as a pointer to the lookup table which determines the configuration of the cube (Lorensen and

Cline 1987). Examples of some of the possible configurations of the cube are shown in Figure 2.1, this figure was taken from (Boris n.d.). The created surface triangulations by marching cubes can contain a large number of elements, which limits its use for being used directly for computational purposes (Pagès, Sermesant, and Frey 2005). Errors can occur with marching cubes; these include non-manifold triangulations (Pagès, Sermesant, and Frey 2005), holes, sharp or flat triangles, and a staircase effect can occur (Lobos and Rojas Moraleda 2013). Structures with non-manifold geometry cannot be 3D printed. When an object is non-manifold and is unfolded into a 2D shape, the surface normals do not point in the same direction (Sculpteo n.d.). Some examples of non-manifold geometry include more than two faces with the same edge, more than two surfaces connected to a vertex, objects that are open so have no volume, and meshes with internal features (Sculpteo n.d.). The holes and triangles may be fixed using other programs or algorithms but, the staircase effect is more difficult to resolve (Lobos and Rojas Moraleda 2013).

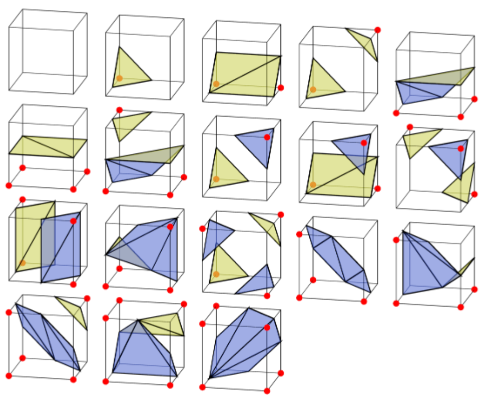


Figure 2. 1: *Example of some possible triangle configurations based on the locations of the vertices in the Marching Cubes algorithm, where the red points are inside the object. The image was taken from (Boris n.d.).*

Poisson

The Poisson algorithm may be used to overcome the staircase effect that may otherwise be produced by marching cubes and the algorithm considers all points so is resilient to noise (Lobos and Rojas Moraleda 2013). Poisson reconstruction attempts to fit a watertight surface to a set of points using a 3D indicator function with a gradient related to the integral of the surface normals. A Poisson equation is used to solve the least-squares approximation for the indicator function. An octree is used to define the implicit function and solve the Poisson equation and an isosurface is extracted to represent the surface (Kazhdan, Bolitho, and Hoppe 2006).

Ball Pivoting Algorithm (BPA)

The BPA reconstructs a surface by starting with a seed triangle where a ball with a certain radius touches the three points of the triangle. The ball rotates around each edge that defines the mesh boundary and stays in contact with two of the original points until it touches a new point without falling through the point cloud. The edge and new point create a triangle that is added to the mesh (Bernardini et al. 1999; Open3D.org n.d.; Poux n.d.). A 2D representation of the ball pivoting around points is shown in Figure 2.2 and was adapted based on the image in (Poux n.d.; Rapponotti, Snowden, and Zeng n.d.). The blue circle represents the starting position of the ball as it touches the two vertices of an edge, the ball then rotates to the position demonstrated in red where it intersects two other vertices. The edge connecting the two vertices intersected by the red ball (dotted line) is added to the mesh.

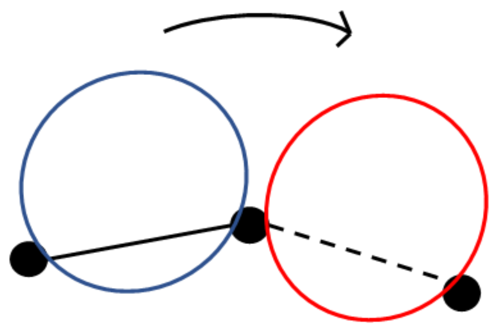


Figure 2. 2: *2D demonstration of the BPA, where the blue circle is the initial location of the ball which then moves to the location of the red circle. The solid line is an existing edge, as the red circle intersects points the dotted line is added to the mesh.*

Tetrahedral Mesh

Tetrahedral volume mesh generation is one of the most common choices for creating patient-specific biomechanical models and methods to generate these are standard in computer-aided engineering (CAE) (Wittek et al. 2016). Quality meshes are created from established techniques such as the Delaunay tetrahedralization and can be created automatically if the information on the closed surface of the desired structure is known

(Wittek et al. 2016). Another form of volume mesh that can be created is a hexahedral mesh, although accurate meshes can be created, the process can be time-consuming and require a large effort by the user and at the time of the review by Wiitek et al. (Wittek et al. 2016), there were no available algorithms to automatically generate a hexahedral mesh of more complicated structures.

Delaunay

Delaunay Triangulation creates a mesh where no vertices are within the open circumscribing disks of the triangles of the mesh. The circumscribing disk connects the three vertices of the corresponding triangle. In 3D, Delaunay tetrahedralizations are defined when no vertices are within the circumscribing spheres of the tetrahedra (Cheng et al. 2012). In 2D, Delaunay triangulations of a point set maximizes the minimum angle, minimizes the largest circumdisk, and minimizes the min-containment disk (Cheng et al. 2012). The min-containment disk of a triangle is the smallest disk that contains the vertices of the triangle (Cheng et al. 2012) in other words it is the smallest circle that connects the three vertices. In dimensions higher than two, the minimization of the largest min-containment-ball is the only property that holds, but 3D Delaunay is still popular because the interpolation error is minimized (Cheng et al. 2012). An example of a 2D Delaunay triangulation is shown in Figure 2.3, where the vertices and edges are in blue, and the circumscribing circles are represented in red. As demonstrated in this figure, the circumscribing circle of one triangle does not contain any of the vertices of the other triangles that compose the triangulation.

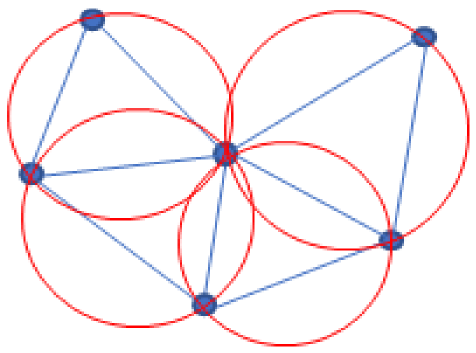


Figure 2. 3: *Demonstration of a 2D Delaunay triangulation where the mesh is represented by the blue vertices and edges, and the circumscribing circles or disks are shown in red. This mesh is a Delaunay mesh as the circumscribing circles do not contain any vertices of the mesh.*

2.2: Current Solutions to Create 3D models from CT Images

2.2.1: Licensed Software

The use of computing in the diagnosis and treatment of craniofacial conditions is based on 3D reconstruction or volume renderings of image segmentations (Wallner et al. 2019). Image-based segmentation is a fundamental step to support diagnosis, surgical planning, and treatments (Wallner et al. 2019; Zhao and Xie 2013). Segmentation is often done manually, however, this process can be very time-consuming (Zhao and Xie 2013). Many software packages are available to aid in the segmentation process but, clinically, the use of software packages is limited as these packages are often functionally complex and need additional resources including financial, human, and technical resources (Wallner et al. 2019). Three examples of commercially available software packages for 3D reconstruction were compared by Martin et al. (Martin et al. 2013) to assess their use in morphometric research and educational purposes. The group compared Mimics (Materialise), Amira (Thermo Fisher Scientific), and OsiriX (Pixmeo), based on a decision matrix where the decision-makers consisted of three novice and three experienced anatomists. Of the three mentioned software packages, OsiriX was the least expensive and was considered very user-friendly but, the basic OsiriX program was limited in its features, tools, and abilities, and was only compatible with Mac operating systems (Martin et al. 2013). There is a free demo of OsiriX but, the full version is $69.99 a month (Pixmeo 2021). In 2013 when the comparison was performed, the base package of Amira cost $5,400, with an additional $6,750 for the large data set option, and Mimics base package cost $6,900, plus an additional $5,175 for extra measurement tools. The reported costs for Mimics and Amira did not include maintenance fees (Martin et al. 2013). Although the more expensive packages had additional 3D modeling and measurement tools, the decision-makers found that Amira was non-intuitive and difficult to use despite its ability to generate high-quality 3D models (Martin et al. 2013).

2.2.2: License-free Software

License-free software for image-based segmentation is available. A review by Wallner et al. assessed the quality and clinical use of six algorithms across three license-free software for use in craniomaxillofacial surgery (Wallner et al. 2019). The review used the software 3D Slicer (https://www.slicer.org/), MeVisLab (MeVis Medical Solutions AG), and MITK (https://www.mitk.org/wiki/The_Medical_Imaging_Interaction_Toolkit_(MITK)), to create segmentation volumes and compare those to two ground truth volumes created through manual segmentation by experts. The chosen software platforms were freely available, extensive, easy to download and use, offered documentation and support through a community of users, and offered options for medical image analysis including 3D reconstructions, visualizations, and the preparations for 3D printing (Wallner et al. 2019). The review determined that the freely available methods were functionally stable, faster than the manual segmentation, and some were considered accurate enough for clinical use. Despite the ease of use and accuracy of the reviewed algorithms, many of the methods required training or expertise to set parameters (Wallner et al. 2019). The application of a combination of nine open-source and commercial software packages was applied to create an STL model from the DICOM images of a dried human mandible by Kamio et al. (Kamio et al. 2020). Differences between the models were most notable in areas of thin cortical bone, even though shapes appeared to vary, there were no significant differences in the shape error between packages. It was noted by the authors that it was necessary to know how the software works to properly create the STL files (Kamio et al. 2020). The results of Kamio et al. (Kamio et al. 2020) indicated there was no significant difference between the shape errors of the different packages, this may imply that open-source packages may be able to create models that are similar to that of commercial products. Additionally, 3D Slicer (https://www.slicer.org/), was shown to be able to accurately segment hard tissue from CT images when it was used to determine the accuracy of models of the orbit of a cadaveric skull created using 3D Slicer and a paper-based 3D printer (Szymor, Kozakiewicz, and Olszewski 2016).

2.3: 3D Point Clouds

2.3.1: 3D Point Cloud Definition

A 3D point cloud was used initially to represent the surface of the bone in this project, the point cloud was then converted to a mesh. The following section describes what a point cloud is and how they have been used for different applications.

A point cloud is a set of points in 3D space that contains x, y, and z coordinates (Chua, Wong, and Yeong 2017; Racasan et al. 2010), where the coordinates indicate the location of the points relative to the base coordinate system (or origin) (Sugimoto et al. 2017). This is used to represent the surface of an object (Chua, Wong, and Yeong 2017), sometimes other attributes, such as colour, can be associated with the points (Sugimoto et al. 2017). Point clouds have the drawback that the data can have high redundancy, an uneven sampling density (which can be caused by the scanner used), noisy data, and lack of structure (Nguyen and Le 2013).

2.3.2: Obtaining 3D Point Clouds

One of the first steps in reverse engineering is obtaining information on the physical object through a scan, the scan creates a point cloud by recording the x, y, and z coordinates of features of the object (Racasan et al. 2010). The raw data output of most 3D scanners is in a point cloud format (Chua, Wong, and Yeong 2017). There are two main types of 3D scanners, contact and non-contact, contact scanners often use a probe while non-contact scanners can be classified as either passive or active. Volumetric methods, which includes CT imaging, is considered an active 3D scanner (Chua, Wong, and Yeong 2017).

2.3.3: Applications of 3D Point Clouds

In industrial applications 3D point clouds from scanners can be used to create geometric CAD models to aid in the reverse engineering process, this is particularly useful when CAD models of complex parts are difficult to create directly (Racasan et al. 2010). Information on geography can also be represented by a point cloud through techniques such as Lidar (Sugimoto et al. 2017). In terms of medical applications, 3D data from imaging sensors can be used for dynamic models for pose and walking, and real-time surgical applications (Sansoni, Trebeschi, and Docchio 2009). Data can also be used in the process of reverse engineering and rapid prototyping to create implants, transplant artificial bone scaffolds, surgical planning (Singare et al. 2009), educational models, finite element simulation and analysis of bone, and to test new designs and materials (Racasan et al. 2010).

3D point clouds can be converted to meshes composed of triangles (triangulation) to represent the surface (Chua, Wong, and Yeong 2017; Racasan et al. 2010). The points can be connected to create the mesh where with a large number of points, a fine mesh is created but at the expense of a long computational time (Chua, Wong, and Yeong 2017). The same can be said about the triangles composing the mesh, with smaller and more triangles the mesh is more detailed, but processing time is high (Racasan et al. 2010). The general steps to create a polygonal mesh from a 3D point cloud include preprocessing through sampling and denoising the point cloud, generating the mesh by triangulating the points, and refining the surface to remove defects and reduce the number of triangles (Singare et al. 2009).

2.4: Mesh File Format for Additive Manufacturing

For additive manufacturing, the industry-standard file format used by CAD systems is STL. STL represents a surface mesh made of triangular facets made up of 3 vertices and a normal, where each facet has a location defined by a position x, y, and z (Chua, Wong, and Yeong 2017). Files can be saved in either binary or ASCII formats, and file size increases with an increasing number of triangles (i.e. a higher resolution) (Chua, Wong, and Yeong 2017). STL does not contain topological information such as material properties or colour and only contains mesh data. This file format can have several errors such as gaps, overlapping facets, and non-manifold topology (Chua, Wong, and Yeong 2017).

2.5: Method for Nearest Neighbour Search

The nearest neighbours search was performed on the point cloud and used for multiple functions throughout this project. To find the nearest neighbours a k-d tree was created from the 3D point cloud to allow for quick nearest neighbours lookup, the k-d tree was implemented using the *spatial* subpackage in the Python package SciPy (The SciPy Community 2021b). The function used to create the k-d tree was called *KDTree*. For this project the optional inputs were set to their default values, the function was therefore run with: a *leafsize* of 10, *compact_nodes* set to True, *copy_data* was set to False, and *balanced_tree* set to True. The *leaf size* defines the number of points at which the algorithm switches to brute force. The hyperrectangles were shrunk to the data range so that the tree takes longer to create but is more robust and has faster query times when *compact_nodes* is True. Data was not copied as *copy_data* was set to False. The median as opposed to the midpoint was used to split the hyperrectangles which creates a compact tree that can be queried quickly but takes longer to build when *balanced_tree* is True (The SciPy Community 2021b). The tree is queried by calling the *query* method, this took an array of points to query, and the number of nearest neighbours that will be found. The output is two arrays of the same length, one of the outputted arrays is the distance to the neighbour and the second array contains the index of the neighbour in the inputted data (The SciPy Community 2021a). When the *query* function was used in this project, the input to query was the 3D point cloud itself. The outputs contained the same number of rows as the number of points in the 3D point cloud. The distances and index outputs contained same number of columns as there are determined nearest neighbours.

Chapter 3: Methods and Results

3.1 Chapter Overview

This chapter outlines the steps that were taken throughout this project. The creation of the 3D model of bone was first done in Python (Python Software Foundation n.d.), the code was then added to Blender (Blender Online Community 2018) through the Python application programming interface (API). The 3D model in Python was created by first extracting a 3D point cloud of the bone from the CT images as explained in section 3.3. Two sampling algorithms were applied to the point cloud to decrease processing time, the algorithms are explained in section 3.4. The point cloud was converted into a mesh, the meshing techniques that were tested and their resultant meshes can be found in section 3.5. The processing time on a remote desktop to create the meshes was measured for two different sets of CT images, images of the head, and of the lower limb, section 3.6 outlined this process and the processing time that was measured.

The meshes created from the two sampling algorithms were compared to the mesh that was created from the full point cloud, the methods used to compare the meshes were described in section 3.7, and the results of the comparisons of the sampling algorithms were described in section 3.8. The code developed and described in sections 3.3 and 3.5 was ported to the Blender Python API to create an add-on that allowed the user to interact with the mesh that was created, the process to create this add-on and a demonstration of its capabilities can be found in section 3.9. Throughout this project both a personal computer and a remote desktop were used, many of the methods were developed on my personal computer and then transferred to the remote desktop. Since methods were done at different times, newer versions of some of the software packages were available, I tried to keep package versions consistent however, some versions were not able to be installed so newer package versions were used. The versions of Python packages used on both computers and in the Blender API can be found in Appendix A.

3.2 Sample Data Preparation

Two sets of CT images of the head were used throughout this project to create models of the skull from both sets of images. These sets of CT images were anonymized and provided by G. Wohl. The two data sets will be described as Skull1 and Skull2. Skull1 contained 28 images and included the top portion of the skull (see Figure 3.1). Skull2 contained 77 images and included the head and a portion of the neck. A portion of the top and the back of the skull were missing from the images in Skull2 (see Figure 3. 2).

An anonymized CT dataset of the lower limbs was retrieved from a publicly available database, The Cancer Imaging Archive (TCIA) (Clark et al. 2013; Vallières et al. 2015a, 2015b). The dataset contained 267 axial CT images spanning from the top of the femur to the plantar surface of the foot and included this for both lower limbs (see Figure 3. 3). The dataset of the lower limbs was manipulated after being imported into Python to isolate a portion of the left femur and was used to test the processing time from CT images to mesh creation in Python. The whole dataset was later used as an example to demonstrate the user interface. The acquisition parameters used to obtain the images described above can be found in Appendix C.

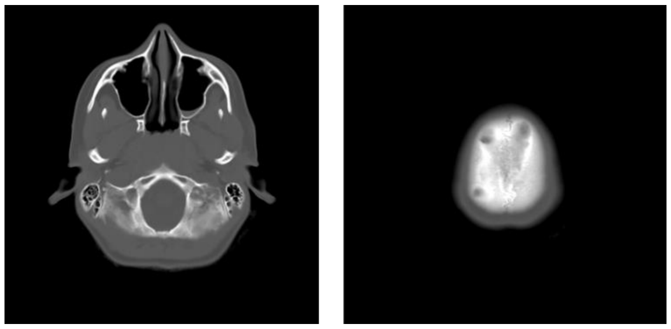


Figure 3. 1: *Images from Skull1 showing the first(left) and last(right) images in the dataset.*

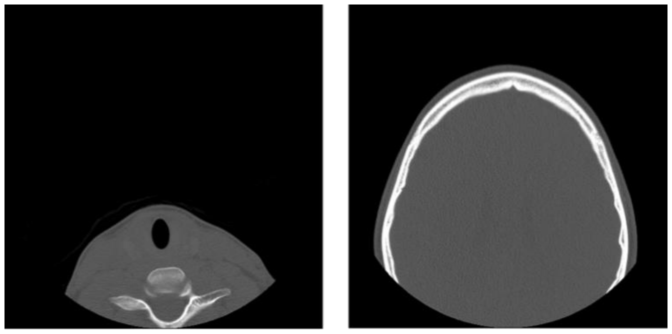


Figure 3. 2: *Images from Skull2 showing the first(left) and last(right) images in the dataset.*

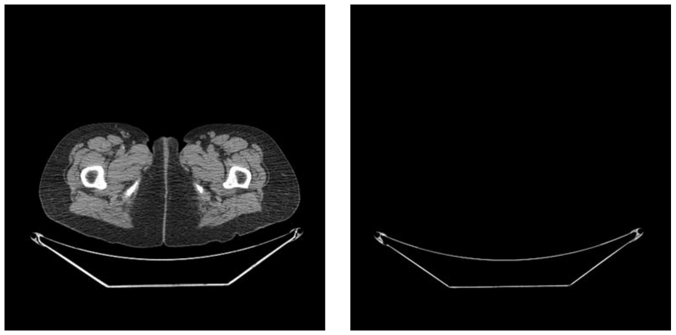


Figure 3. 3: *Images from a dataset of lower limbs showing the first(left) and last(right) images in the dataset.*

3.3 Creation of Point Cloud

The creation of the point cloud from CT data was accomplished similarly to Chougule, Mulay, and Ahuja(Chougule, Mulay, and Ahuja 2013). Obtaining the point cloud involved binarizing the images, applying an edge detection algorithm, and converting edge pixels to 3D spatial coordinates. The process to convert the CT images into a 3D point cloud was originally performed using a custom code developed in MATLAB (9.6.0.1214997 (R2019a), The MathWorks Inc.) that was split into three functions: *open_CT, thresh_edge_ct*, and *extract_PT_Cloud*. The code was then ported to Python (version 3.8.3, Python Software Foundation) because, in comparison to MATLAB, Python is free and open-sourced, which aligned with the goal of the project to develop a method that was easily accessible.

3.3.1 Function: *open_CT*

The DICOM images were imported into Python using the package Pydicom, with code adapted from (Heng 2019). Imported images were converted from the pixel array in the DICOM format to a NumPy (Harris et al. 2020) stack, this conversion allowed the pixel data to be easily accessed and manipulated. The pixels in the stack of images were converted from grayscale to Hounsfield Units (HU) through a linear transformation Output units = m*SV+b, where b is the Rescale Intercept, and m is the Rescale Slope in the DICOM header (Alshipli n.d.; Innolitics LLC n.d., n.d.). Other header information returned through this function included the slice locations, the pixel spacing in the row direction, and the pixel spacing in the column direction.

3.3.2 Function: *thresh_edge_CT*

The purpose of this function was to extract the pixels of the inner and outer surfaces of the bone (skull) from the stack of CT images (Skull2). This was accomplished using two steps: 1) segment bone from surrounding soft tissue, 2) apply an edge detector to isolate the surface of the bone. A global threshold was applied to all the CT images, this created a set of binary images, in which pixel values above the threshold were considered the bone of the skull and set to a value of 1, while those below the threshold were set to values of 0 and became the background. The methodology was developed using a set of CT images of the head and the threshold used was 350 HU. The value of 350 HU was chosen based on the process followed by Tilotta et al. (Tilotta et al. 2009) in developing a CT database for craniofacial reconstruction, this threshold to determine the bone of the skull was also demonstrated in (Hounsfield 1973).

The second stage of the process isolated the surfaces of the bone with the use of an edge detector. Edge detection reduces the amount of data contained in the images while preserving information on the structure of objects in the image (Canny 1986). The data was reduced to edges so that the point cloud produced represented the surface of the bone and all not pixel values containing bone. The isolated edges reduced the amount of data contained within the point cloud, and the actual bone tissue is not necessary to create the surface STL files that would be ultimately manipulated by the user.

In this application, the function *canny()* from the feature module in the scikit-image (van der Walt et al. 2014) package in Python was used, this function uses the Sobel operator to calculate the gradients in the horizontal and vertical directions of the images (scikit-image development team n.d.). The default parameters for this function were used, this included σ= 1.0 (standard deviation of Gaussian smoothing filter), the lower value for hysteresis thresholding was set to 10% of the maximum value of the data type of the image, and the upper value for hysteresis thresholding was set to 20% of the data type maximum value (scikit-image development team n.d.). The output of the Canny edge detector was a 2D binary edge map. The Canny edge detector is commonly used because it provides good detection, good localization, and only one response to a single edge (Canny 1986; Ding and Goshtasby 2001). To determine the edge pixels through this method, images were first convolved with a filter, commonly a Gaussian smoothing filter (Canny 1986; Jain, Rangachar, and Schunck 1995), the gradient (magnitude and direction) was calculated for each pixel, and non-maximum suppression was applied (Canny 1986; Jain, Rangachar, and Schunck 1995; Prince 2012). Non-maximum suppression thins the edges in the gradient magnitude image so that only the maximum value of the edge remains (Jain, Rangachar, and Schunck 1995). To accomplish the suppression, the angles of the gradient of each pixel were separated into four sectors (Jain, Rangachar, and Schunck 1995; Prince 2012), each pixel in the gradient magnitude image was compared to two neighbours in the direction of the gradient, if the pixel value was less than one or both neighbours it was set to zero, this kept only pixels with the maximum gradient amplitude (Jain, Rangachar, and Schunck 1995). After non-maximal suppression, the images were subjected to hysteresis thresholding which uses a lower and an upper threshold. Pixels above the high threshold, and pixels below the high threshold but above the lower threshold connected to an existing edge were chosen as edge points. The high threshold helped reduce the number of false detected edges while the lower threshold reduced the number of missed edge points (Canny 1986; Jain, Rangachar, and Schunck 1995; Prince 2012).

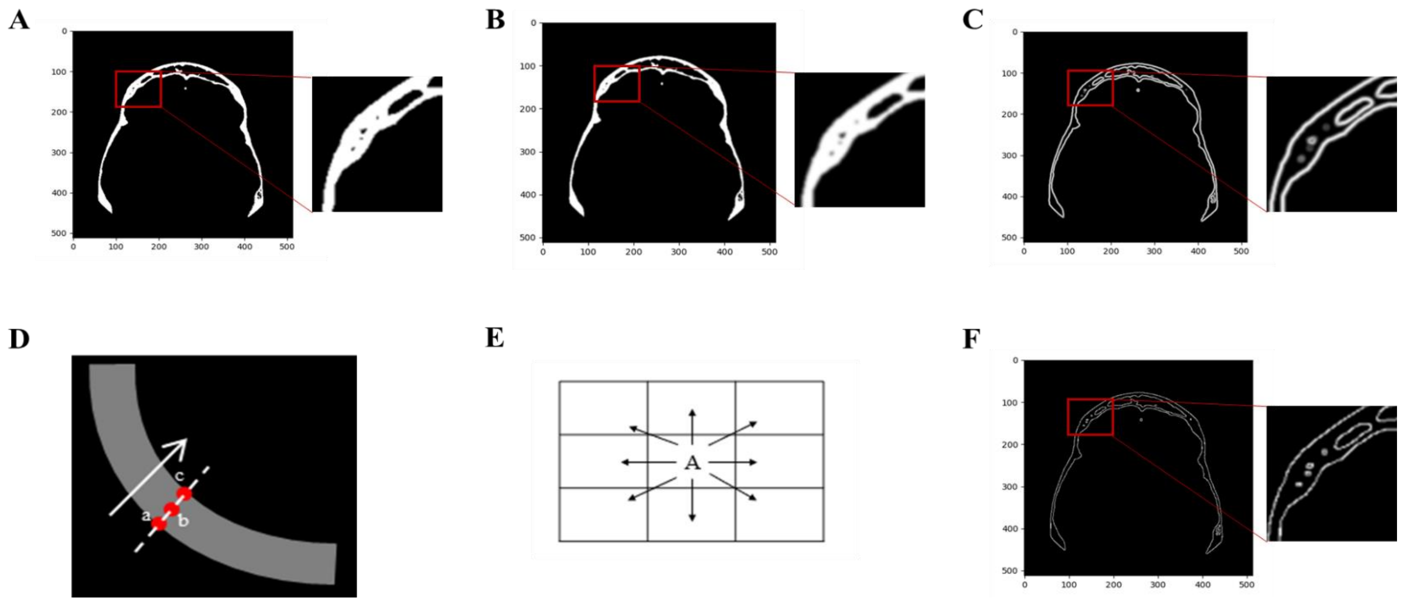


Figure 3. 4: *Depiction of the process of Canny edge detector applied on an image from Skull2. A. CT image after binarization, B. Gaussian smoothing filter with σ=1 applied, C. Sobel filter applied to obtain image gradient, D. Non-maximum suppression, E. Pixels for hysteresis thresholding, and F. Output of canny().*

Figure 3.4 demonstrates the steps used to accomplish the edge detection using the Canny edge detector applied to a slice from Skull2. Image A is one of the CT images after the threshold was applied. After the threshold was applied the first step in the edge detection was applying a Gaussian smoothing filter with a σ = 1, this was done with the Gaussian filter from scikit-image (scikit-image development team n.d.), image B. The function *canny()* used the Sobel operator to calculate the image gradients. The Sobel filter from scikit-image (scikit-image development team n.d.) was applied to the smoothed image (B) to achieve image C. After the gradient calculation non-maximum suppression was applied, this process is demonstrated in D, the grey curve represents an edge composed of three pixels labelled a, b, and c. The dotted line connecting the pixels represents the gradient of the edge, the direction of which is labelled with the arrow. The process of non-maximum suppression can be demonstrated by looking at point b, if the gradient magnitude of surrounding pixels along the gradient, a and c, are greater than b, then b is set to zero, otherwise, it is kept as an edge pixel. This enabled thinning of the edges. Following non-maximum suppression, hysteresis thresholding was applied. For the function *canny()*, first, all the points above the high threshold were labelled as edges, points above the low threshold but less than the high threshold were compared to the eight surrounding pixels demonstrated in E, where the pixel of interest is labelled A. If the labelled pixel A, was connected to an existing edge in any of the eight surrounding pixels, A was added as an edge pixel (scikit-image development team n.d.). Finally, the process culminated with the image displayed in F, the output of the *canny()* function itself on the binarized CT image.

3.3.3 Function: *extract_PT_Cloud*

To create the point cloud of the surface of the bone, pixels representing an edge point in the stack of CT images were converted to 3D spatial coordinates. The coordinates were obtained using the pixel spacing in the directions of the rows and columns as well as the slice locations of each image, these values were accessed from the DICOM headers of the original CT images and returned by the *open_ct* function. The values for the x and y positions of the coordinates were taken as the in-plane values at the center of the pixels in the row and column directions, respectively. The x-coordinate was determined by multiplying the column position as an integer by the spacing between the pixels in that direction then subtracting half of the pixel spacing so that the coordinate represented the middle of the pixel. The same logic was used to determine the y-coordinate of the pixel of every edge point but using the row values. The row and column numbers started at zero in the top left corner of the image, the row number increased towards the bottom of the image, and the column number increased towards the right of the image. The z-coordinate of the point was set as the location of the slice of the original CT image. The directions of x, y, and z are shown in Figure 3.5. Points were saved in an Nx3 matrix, where N is the number of points within the point cloud, and the x-coordinates, y-coordinates, and z-coordinates were contained in the first, second, and third columns, respectively. The order of the points in the matrix starts with all the points from the first slice followed by all the points from the second slice and so forth. Within each slice, the points contained in the first row were saved, followed by the second row and so forth. There was no grouping of the points based on the surface they belonged, in other words, the points were not grouped based on the inner and outer surfaces. An example of the resulting point cloud for Skull2 is shown in Figure 3.6.

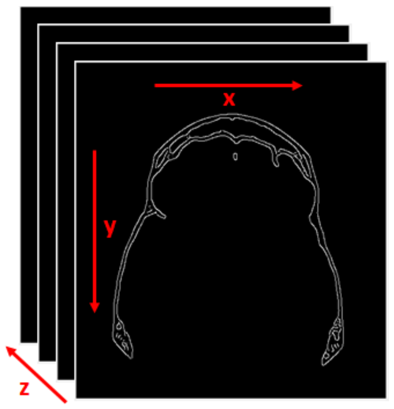


Figure 3. 5: *Demonstration of the directions chosen to create the point cloud from the stack of CT images on an image from Skull2. In this figure, the images have undergone the process of thresholding and edge detection.*

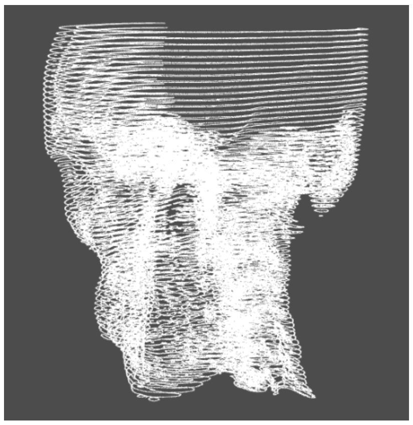


Figure 3. 6: *Example of point cloud for Skull2 data set created using the method described above (i.e. binarization, thresholding, edge detection, and point cloud extraction), the point cloud contained 282,938 points.*

3.4 Down Sampling Point Cloud Algorithms

3.4.1 Pseudo-Uniform Sampling

The point cloud was downsampled to reduce its size with the hope of reducing the computational time required to produce a mesh object from the point cloud. The first method of sampling conducted was a random sampling procedure in which the desired percentage of points was randomly sampled from the matrix containing all points of the point cloud. Although this process was fast and effectively sampled the point cloud to the desired percentage, the repeatability of the method was lacking as even with the same percentage the outputted sample could vary. The variation in the sample made it more difficult to compare the sampling method to the targeted sampling that is described below because more trials had to be completed to compensate for the different outputs. For more consistent results the random sampling was converted into a pseudo uniform sampling algorithm, although this process was more time-consuming the resultant sampled points were more consistent. The pseudo uniform sampling function took the point cloud, the locations of the slices of the point cloud, and the desired percentage to be sampled as inputs. For each value in the slice locations, all points at that z-level in the point cloud were found using the NumPy function *where()* (The NumPy Community 2021k), which returns the indices of points in the list that satisfy a conditional statement; in this case, the statement checked that the z-value of the point was equal to a certain slice position. The points with the desired slice location were saved as *z_points*. A KD tree was created for *z_points* using the *KDTree* function contained in the spatial subpackage of SciPy (The SciPy Community 2021b; Virtanen et al. 2020). The tree was queried to find the 10 nearest neighbours to all points in *z_points*. When the tree was queried, the distances between the point and its nearest neighbours were returned as well as the indices of the nearest neighbours in *z_points* (The SciPy Community 2021b). This approach kept each point itself as its closest neighbour, ideally, this would return a list of points including the initial point and 4 or 5 points on either side on the same surface. This may not always be the case, but for this sampling, it was assumed that points along the line would be closer to one another than points on the opposite surface (i.e., the outer and inner surfaces).

For the first set of neighbours in the list of points, points were chosen based on the percentage inputted into the function. The NumPy function *linspace()* (The NumPy Community 2021e) was used to choose a set of values to sample from the list of nearest neighbours, the function returned a specified number of values evenly spaced between a start and stop value with these values being included in the set of points. The number of values returned by *linspace()* was equal to the length of the list with the nearest neighbours multiplied by the desired percentage, for example with 10 nearest neighbours and the desired percentage sampled of 50%, 5 values were chosen between [start, stop]. The values returned may contain decimal points, however, indices must be integers; as such, the list returned by *linspace()* was rounded to the nearest integer using the NumPy function *rint()* (The NumPy Community 2021g). After being rounded, 1 was subtracted from all values so that the indices were between the values of 0 and 9, instead of 1 and 10. The determined indices were used to select the desired number of points for the nearest neighbour of the first point. Once the sample was taken all of the original points were removed from the array and the process was repeated until the total array was empty. Deleting the points as the code progressed ensured that points were not chosen as a sample more than once. The sampled points were saved in a matrix and returned by the function.

The disadvantage of performing the pseudo uniform sampling was that it did not guarantee that sharp corners, edges, or features would be maintained. In areas with high curvature, where the contour connecting the points may vary frequently, choosing a uniform sample of the curve did not guarantee that the curvature is maintained. As an example, Figure 3.7 shows how uniform sampling to 10% of the original sample affects two different sets of points. The first set of points represents a straight line of the equation $y = 10x$, while the second set of points is represented by the equation $y = 10\cos\left(\frac{x}{4}\right)$. The same x values were used for both equations, these values consisted of 50 evenly spaced points between 0 and 100 using the NumPy function arange (The NumPy Community n.d.). The x values were then sampled to 10% of the original values. Plots for the equations using the original x values and the sampled x values were created using matplotlib (Hunter 2007). For the straight line depicted in A and B despite the number of points being decreased from 50 to 5, the shape of the line was maintained. However, when the plot containing the cosine curve was sampled (C and D) to 10% the curvature and shape were distorted.

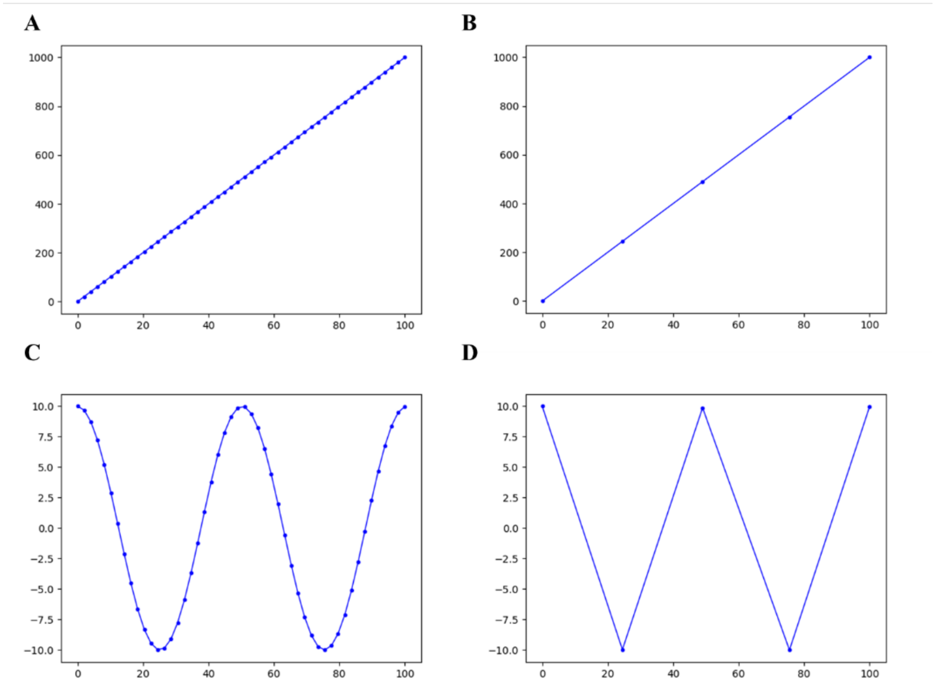


Figure 3. 7: *Depiction of the effects of sampling a point set representing a straight line in comparison to sampling a set representing a cosine curve. A. Original straight line, B. Straight line sampled to 10%, C. Original cosine curve, and D. Cosine curve sampled to 10%.*

As demonstrated, if the shape represented by the point cloud was more complex than a straight line, the sampled point set could distort the shape of the original set. To reduce the amount of shape distortion introduced by the sampling algorithm, a sampling method based on the curvature of the point set was proposed, this was referred to as targeted sampling.

3.4.2 Targeted Sampling

Due to concern that random sampling could miss features of the shape of the object, sampling the point cloud based on curvature was tested. To represent the curvature of the object, the point normals of surrounding points were compared to one another. The normal at a point is perpendicular to the tangent space, where the tangent space represents an approximation of the surface (Berger et al. 2017). Before estimating the point normal, the volume represented by the point cloud was separated into voxels by creating grids in the x, y, and z directions. The maximum and minimum x, y, and z coordinates were used to represent the bounding box of the volume represented by the point cloud.

The NumPy function *arange()* (The NumPy Community n.d.) was used to create a list of coordinates representing the boundaries of the grid in the three directions. For the x-direction, the coordinates spanned from the minimum x-value minus the pixel spacing in the column direction multiplied by a constant c, to the maximum x-value plus the same value that was subtracted from the minimum x-value. The spacing between the boundaries of the grid in the x-direction was the pixel spacing in the column direction multiplied by the constant c. The grid in the y-direction was determined in the same manner as the grid in the x-direction but, instead of c, the constant was referred to as r, and the spacing used was the pixel spacing in the direction of the rows of the images. The grid in the z-direction was determined with the use of a constant z, the step between the grid was determined as the constant z multiplied by the absolute value of the spacing in the z-direction. The absolute value of the spacing in the z-direction was calculated as the difference between the slice locations of the first and second slice of the CT images. The grid boundaries for the x and y direction are demonstrated in Figure 3.8, where the point cloud shown was from slice 60 of Skull2. The solid red line in the figure represented the maximum and minimum boundaries of the grid used. The label A demonstrated the distance from the minimum x-value of the point cloud to the boundary and B demonstrated the distance from the maximum x-value of the point cloud to the boundary. Similarly, C represented the distance to the boundary from the maximum y-value, and D represented the distance from the minimum y-value to the boundary. The minimum values of the point cloud were represented with the white dotted lines. Here the distances A and B were equal to $c * sp_x$ and the distances C and D were equal to $c * sp_y$, where c is a constant and sp_x and sp_y is the pixel spacing in the x (column) and y (row) direction respectively. The boundary (solid red line) was then divided in each direction into grids such that the spacing between the grid lines was equal to the distances A, B, C, and D. The divisions are demonstrated in Figure 3.8 as the set of dotted lines, this included both the red and white dotted lines.

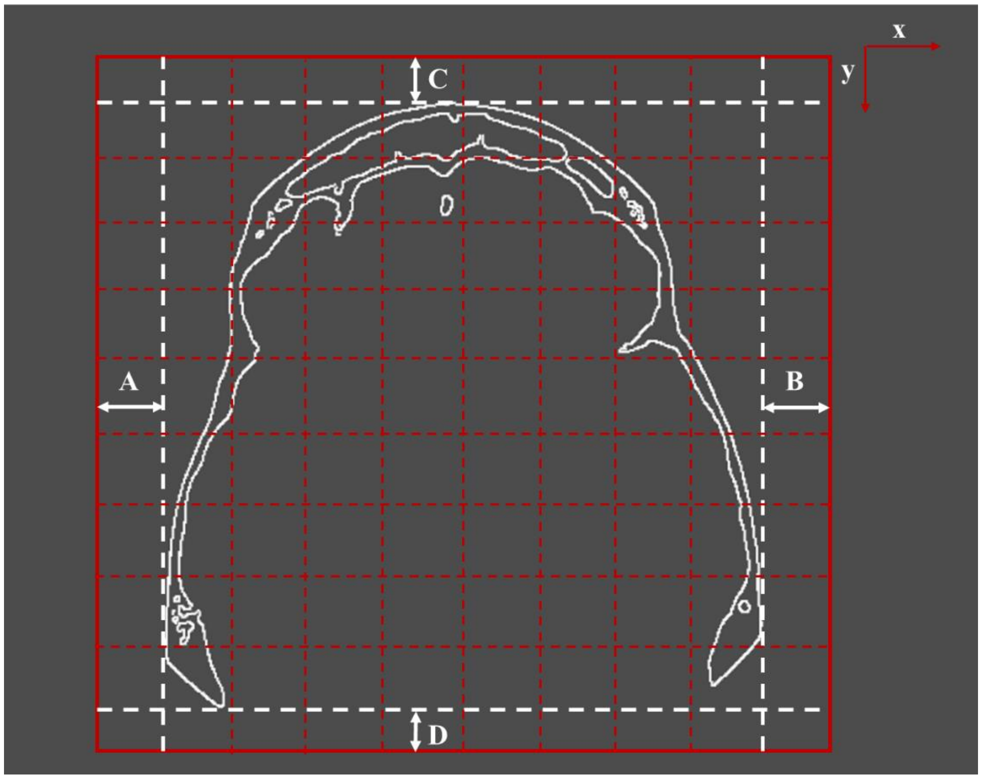


Figure 3. 8: *Demonstration of boundaries of the grids in the x and y directions for the targeted sampling algorithm on the point cloud of slice 60 of Skull2. The solid red lines are the bounds of the whole grid, the white dotted lines represent the minimum and maximum values of the point cloud, and the dotted lines (red and white) represent the divisions for the grid.*

The points within each grid space were separated based on their respective slice or z-location, the points in each slice were converted to an Open3D (Zhou, Park, and Koltun 2018) point cloud and the point normals were estimated using the *estimate_normals()* function from Open3D. This function finds the nearest neighbours to a point and uses covariance analysis to calculate the principal axis of these points (Point cloud 2020). The function uses a KD Search Tree to perform a hybrid K-nearest neighbours and radius search. The search took two inputs, the maximum number of nearest neighbours, and the search radius, the default values of 30 maximum nearest neighbours, and a 10 cm search radius were used. These values were used to simplify the process as there were other parameters such as the grid size that would be modified to change the sampling algorithm. A 10 cm search radius was commonly larger than the dimensions spanned by the points in the slice of each grid. Having a large search radius ensured no points were missed and separating the points based on their slice location and grid ensured points within the search radius and in other voxels were not included when estimating the normals. The default value of 30 nearest neighbours was used because a larger value may have incorporated points, not on the same surface so did not properly represent the surface, while a much smaller number may not have been enough to properly represent the surface. The output from the covariance analysis to determine the normal gave two possible directions since the original point cloud did not have any direction of the structure associated with it Open3D guesses which direction was correct (Point cloud 2020). Once the normals were calculated for each slice in the grid, the angular similarity between the current point and every other point in the same slice and grid was calculated, this was done for every point and the average angular similarity was calculated for each slice location in each grid.

Huang and You (Huang and You 2012) performed matching of point clouds in 3D based on both photometric and geometric properties including surface normal and curvature. The method of angular similarity was chosen as a metric for the targeted sampling because normal similarity directly describes the shape of the surface model (Huang and You 2012). Both Alexiou and Ebrahimi (Alexiou and Ebrahimi 2018) and Huang and You (Huang and You 2012) defined metrics for comparing point clouds based on the angular similarity of the normal. From the results of Huang and You, the precision, defined as the ratio of true matches to false matches was highest for the normal similarity (Huang and You 2012). Alexiou and Ebrahimi (Alexiou and Ebrahimi 2018), compared the tangent planes of points of point clouds to create a quality assessment. The difference between the normals was determined as the angle between them, this angle represented the angle between the tangent planes, the larger the angle, the greater the difference between the surfaces at these points (Alexiou and Ebrahimi 2018). The targeted sampling followed a similar principle as those described where the angular similarity was calculated and based on the average angular similarity grids were sampled by a certain percentage.

The angular similarity was calculated based on the descriptions from (Alexiou and Ebrahimi 2018; Huang and You 2012; Statistical Engineering Division Dataplot n.d.). The angular similarity was defined as $1 - angular\ distance$ where the angular distance was defined as:

$$angular\ distance = \frac{\cos^{-1} cosine\ similarity}{\pi}.$$

Cosine similarity is defined as cos(θ), where θ is the angle between two vectors. The cosine similarity was determined between the normal vectors using:

$$cosine\ similarity = \frac{n_1 \cdot n_2}{\|n_1\| \|n_2\|}$$

where n_1 and n_2 are the point normals that are being compared (Alexiou and Ebrahimi 2018; Huang and You 2012; Statistical Engineering Division Dataplot n.d.).

For every point within the slice in each grid, the angular similarity was calculated between the normal of that point and the normal of every other point in that slice of the grid. The cosine similarity was found by dividing the dot product between the two normal vectors by the multiplication of the vector norms of the two normals, where the dot product and the vector norms were determined using the NumPy functions *vdot()* (The NumPy Community 2021j) and *linalg.norm()*(The NumPy Community 2021d), respectively. The angular similarity was calculated by subtracting the arccos (found using the *acos()* function from the Python math module) of the cosine similarity divided by pi from 1. The value of the cosine similarity was rounded to contain six decimal points when calculating the angular similarity. When developing the code there were times when the cosine similarity was calculated to be a number greater than 1 that may have been caused by imprecision in the floating-point values. An example of such a value was 1.0000000000000002; this caused an error when finding the arccos as the cosine of an angle is bounded by -1 and 1. Limiting the similarity to six decimal places removed the erroneous values. This would not have a large impact on the results since the points were sampled based on ranges of angular similarity.

Once the angular similarity between all of the points in the slice and grid were determined, these values were averaged such that every slice in the grid had an overall average angular similarity associated with it. Separating the points within the grid based on the z position ensured all the points have the same z value so that when the normals and the similarity between the normals was calculated the curvature in the in-plane dimensions (x-y plane) was represented. Through this process, the points were sampled based on the angular similarity in the x-y plane and not in the z-direction. For an object such as the skull, a lot of the change in the surface will be in the x-y direction and not the z-direction. Based on the average angular similarity a certain percentage of points was sampled from each slice.

To demonstrate the calculation of the average angular similarity, the process described above was applied to two curves, a straight line, and a quadratic curve (Figure 3.9). The x values used to create the plots to depict the effect of uniform sampling, a set of 50 evenly spaced points between 0 and 100, were used. The equation of the straight line to determine the y values was $y = 10x$ and the quadratic equation used was $y = -(x - 50)^2$. The calculation of angular similarity was performed on points within the same slice so the z-coordinates for all the (x,y) points determined based on the two equations was set to a value of 0. The curves that the calculation was performed on can be seen in Figure 3. 9 and were plotted using matplotlib (Hunter 2007). The angular similarity calculated was 0.89 for the straight line and 0.52 for the quadratic curve, these values demonstrated the variation between the different shapes. The quadratic plot had more curvature and a lower angular similarity between the point normals than the straight line, thus angular similarity may be used to classify the curvature of lines.

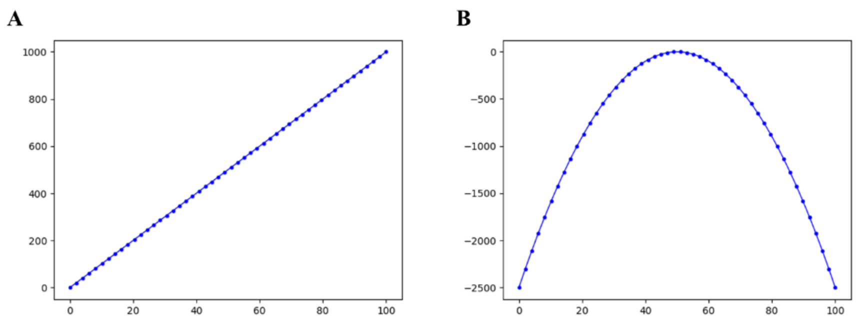


Figure 3. 9: *Depiction of two curves used of which the angular similarity was calculated. A. Straight line had an angular similarity of 0.89, and B. Quadratic curve has an angular similarity of 0.52. This demonstrated that shapes with higher curvature (B) have a lower angular similarity.*

The grid size, the ranges used for the angular similarity, and the percentages sampled were determined using a Design of Experiments (DOE) approach that compared the meshes created using the grid sampling with the mesh created from the full point cloud. The experiments also aimed to compare the pseudo-uniform and the targeted grid sampling approach. This process is discussed in Section 3.7: Comparison of Meshes.

3.5 Mesh Creation from Point Cloud

The two packages that were used to manipulate and display the point cloud in this project were Open3D (Zhou, Park, and Koltun 2018) and PyVista (Sullivan and Kaszynski 2019). The method used to create the point cloud from the mesh was determined by running tests on two models created from images of a circle, and a line (Figure 3.10). Three methods from Open3D were tested, these were accomplished using the functions *create_from_point_cloud_ball_pivoting* which uses the Ball-Pivoting Algorithm (BPA), *create_from_point_cloud_poisson* which uses the Poisson method, and creating a TetraMesh object. The second package used, PyVista, provided two functions to achieve a Delaunay triangulation of the mesh, a 2D Delaunay triangulation and a 3D Delaunay triangulation.

3.5.1 Testing on Models

To test which method would work for this application two models were created, the first consisting of a straight line, and the second consisting of a cylinder. The different methods were first applied to the model created using the straight line and based on the results of the methods on this model, successful methods were applied to the cylinder model which was more complex than the straight-line model.

Examples of point clouds of the two models used are shown in Figure 3.10. Here the spacing between the slices was 5.0 mm and the number of slices was 10. The test models were used before applying the methods to the point cloud of the skull to test on a smaller scale than the skull, and on models with less complicated geometry. The first test model used was created from a stack of images each containing a solid white line of a certain thickness on a black background, while the second was created similarly but used a circle. This simulated the pixels with values of 1 that represent the bone in the CT images after thresholding was applied.

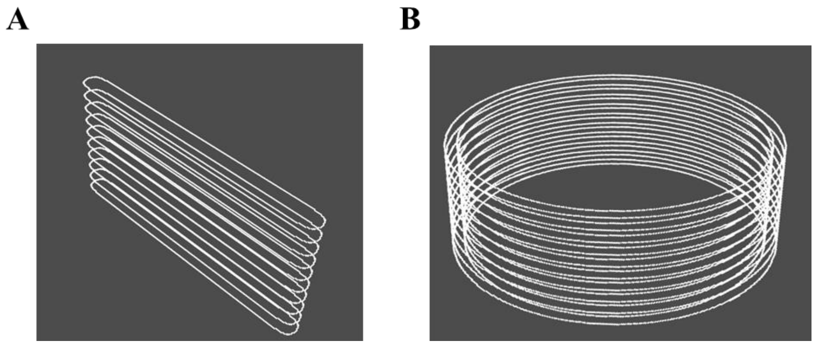


Figure 3. 10: *Two models used to test mesh methods. A. Point cloud of the model created from images of a straight line, B. Point cloud of the model created from images of a circle. Plotted using PyVista.*

To build the stack of images for the line model a function *Create_Line_Stack* was written such that it takes a number of desired slices, the thickness of the line, the size of the image, "s" (i.e., number of pixels), and spacing between the slices as inputs. The function created a black image of s x s and then draws a line in the centre of the image of the desired thickness and with a vertical length of approximately half the height of the image. The stack was formed by creating a NumPy array representing the image and repeating the array in the third dimension. The z-location of the images in the stack was determined based on the spacing entered as an input, the first slice was at a height of 0.0 mm with each subsequent slice stacked upward being higher in the z-direction by the value of the inputted slice spacing in millimeters.

To test the various meshing methods, stacks of images were created using 1, 2, and 10 slices as well as slice spacing of 1.0 mm and 5.0 mm. The thickness used was 20 pixels and the size of the images was 512x512 pixels. The size of the image and 5.0 mm pixel spacing was chosen to match the size of the DICOM CT images from one of the test skull image sets, Skull1. The stack of images that these models were based on, Skull1, contained 28 slices, a spacing between slices of 5.015 mm, and a pixel spacing of 0.4492 mm in both the x and the y directions.

The cylinder model was created similarly to the line stack model. The stack of images representing the cylinder were created by taking a black image, drawing a circle on the image with a desired radius and thickness (specified by the number of pixels), and replicating the image using a NumPy array to create a stack of images. Like the line stack, the circle drawn was white to represent the value of 1 represented by bone in the CT images after thresholding was applied. The number of times the images were repeated to get the number of slices was defined by the user, as was the spacing between these slices. To extract the point cloud from the model, the pixel spacing of Skull1 (defined previously) was used to obtain the x and y coordinates, while the spacing between the slices was used to define the z-coordinate as was done with the line stack model. An example of the point cloud of the cylinder model is shown in Figure 3.10 with 10 slices, a line thickness of the initial circle drawn of 10 pixels, a radius of 150 pixels, and spacing between slices of 5.0 mm. The point cloud had both an inner and outer ring representing the cylinder. This model was created as a more complex model to mimic the shape of the cranial vault of a skull where the bone has both an inner and out surface. The methods to create a mesh from the point cloud had to connect these inner and outer surfaces of the bone, so testing on a hollow cylinder was able to test this property. The model contained fewer points in comparison to the point cloud of the skull to reduce processing time for creating the mesh for the model. For example, the model in Figure 3.10 contained 21,560 points while the point cloud obtained from Skull1 contained 84,210 points.

The methods BPA, Poisson, and TetraMesh from Open3D as well as the 2D Delaunay and 3D Delaunay triangulation from PyVista were applied to the line stack model created with 1, 2, and 10 slices and slice spacings of 1.0 mm and 5.0 mm. The process to create the mesh using the Open3D BPA and Poisson method was based on the tutorial at (Poux n.d.). These two methods required the point clouds to have normals that were not present in the representation of the point cloud used throughout this project, the normals were calculated using an Open3D function to estimate the point cloud normals. This is the same function used in the targeted sampling algorithm described previously (Section 3.4.2). In this case, the KD search tree used throughout these tests was a radius of 1 cm and a maximum number of nearest neighbours of 30 (Poux n.d.).

The BPA function takes a list of radii of the balls that are pivoted on the point cloud to create the mesh as a parameter (Open3D.org n.d.). The list of radii used was determined in the same way as (Poux n.d.), where the nearest neighbour distances of the point cloud were calculated using the open3D function *compute_nearest_neighbour_distance()*, the average of these distances was then found. The radius used was three times this average distance, and the list of radii used in the BPA was the radius and twice the radius. The radius calculated for the line stack was 1.35 mm.

For the Poisson triangular mesh, the parameters of depth, and scale were set as the default parameters of 8, and 1.1, respectively. The width parameter was set to 0, this parameter was ignored by the function because the depth was specified. The depth parameter described the depth of the tree used in the algorithm, where larger values created more detailed meshes. The scale parameter was the ratio of the reconstruction cube and the diameter of the box bounding the point cloud. Lastly, the parameter *linear_fit* was set to false so that linear interpolation was not used to determine the isovertices (Poux n.d.). The TetraMesh object in Open3D was created using the function *create_from_point_cloud*, where the only input is the point cloud (www.open3d.org 2020). The meshes created were saved as STL files, however, when saving the output of the Poisson reconstruction, the vertex normal had to be computed using the function *computer_vertex_normals()* before it was able to be saved as an STL.

The 2D and 3D Delaunay triangulation using PyVista were both done using the functions default values. The resulting meshes were saved as STL files.

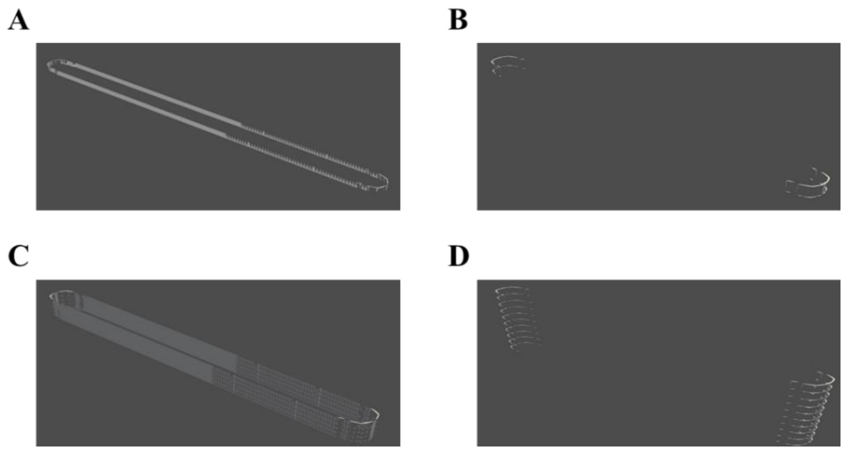


Figure 3. 11: *Output of BPA mesh function on line stack with A. 2 slices and spacing of 1.0 mm, B. 2 slices and 5.0 mm spacing, C. 10 slices and 1.0 mm spacing, and D. 10 slices and 5.0 mm spacing. Plotted using PyVista. At higher spacing (B and D) few connections were made in the z-direction. No outputs spanned across the line.*

In the meshes produced for the line model using the BPA method (Figure 3.11), there were few connections along the sides (z-direction) for the larger slice spacing. This is caused by the ball being too small to create triangles in the z-direction, since the ball was less than the z-spacing it essentially fell through the points, there was no contact with the points on 2 adjacent slices, and no triangles were created for the mesh. The algorithm did not fill in the gap between the two sides of the line. For this project, the resultant surface should be closed between the edges to properly represent the bone structure.

To determine if the BPA could create triangles between the edges as well as in the z-direction for the stacks with higher spacing, the algorithm was run again for the stack of 10 slices with 5.0 mm spacing using radii lists of 5.0 and 10.0 mm, and 7.5 and 15 mm (Figure 3.12).

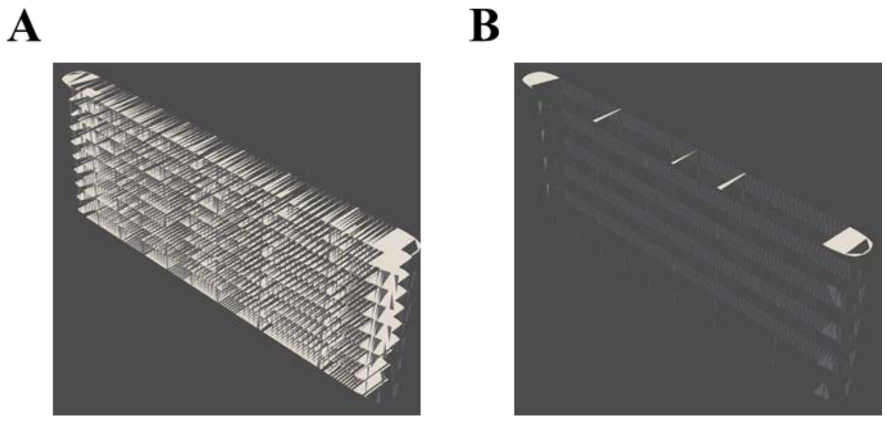


Figure 3. 12: *Output of BPA for 10 slices and spacing of mm using radii list of A. 5.0 and 10.0 mm, and B. 7.5 and 15 mm. Plotted using PyVista. The lower list of radii (A) produced some connections in the z-direction and connections that spanned the line. The higher range of radii (B) produced few connections in the z-direction and spanning the line. Line models were not fully represented by the meshes.*

The radii list using 5.0 mm and 10 mm produced connections in the z-direction and between the two sides of the line than the radii of 1.35 and 2.7 mm (Figure 3.12 A). The radii list of 7.5 and 15 mm (Figure 3.12 B) had fewer connections in the z-direction and between the edges than the mesh created using 5 and 10 mm. The thickness of the line used was 20 pixels, based on the spacing between the pixels of 0.4492 mm that the point cloud was based on, the distance between the two edges in the x-y plane is 8.984 mm. This distance meant that balls of smaller size were unable to span the gap as they were too small and fell through. When the size of the ball was increased, particularly for 5.0 and 10.0 mm, some connections were able to be made since the 10.0 mm ball is approximately 1 mm larger than the space between the edges, whereas, for the 7.5- and 15-mm balls, the 15 mm was much larger than the gap such that it was unable to touch sets of points spanning the gap so there were fewer connections made.

After the BPA method was tested, the Poisson method from Open3D was tested on the same point clouds used to test the BPA. These models included 1, 2, and 10 slices with spacings of 1.0 and 5.0 mm, and the resulting outputs of the Poisson method are shown in Figure 3.13.

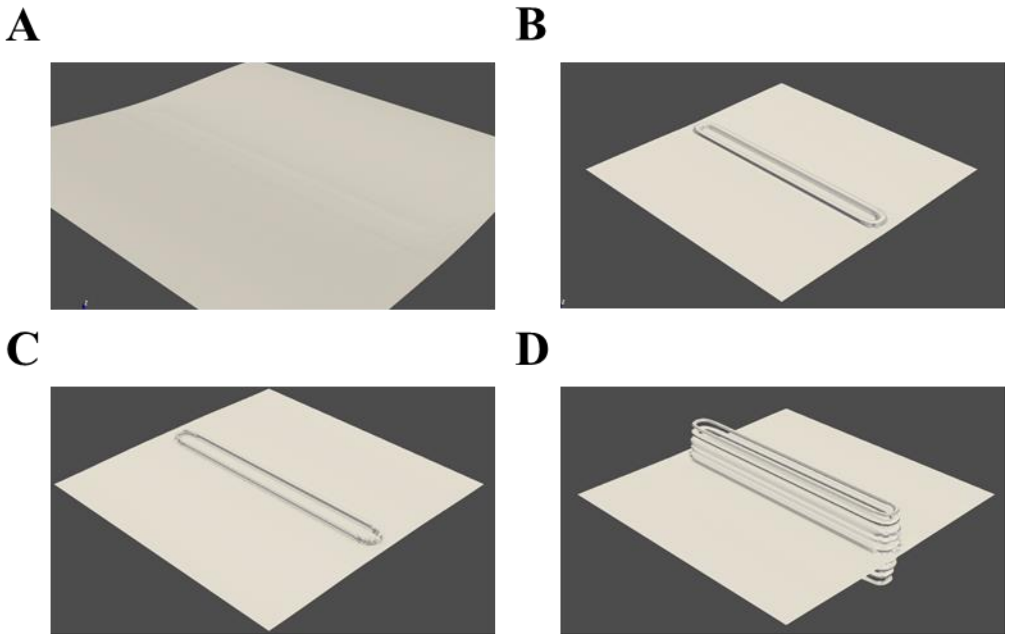


Figure 3. 13: *Results of Poisson method on the line test model using A. 2 slices and slice spacing of 1.0 mm, B. 2 slices and slice spacing of 5.0 mm, C. 10 slices and slice spacing of 1.0 mm, and D. 10 slices and slice spacing of 5.0 mm. Plotted using PyVista. Connections in the z-direction were not made, and a plane was created in the middle of the mesh. Line models were not properly represented with this method.*

In the four outputs from the Poisson algorithm (Figure 3.13), a plane appeared approximately in the middle of the meshes. In addition to the plane the gap between the two edges was not spanned by the mesh and the layers in the z-direction are not connected.

The outputs from the results of the Open3D TetraMesh are shown in Figure 3.14.

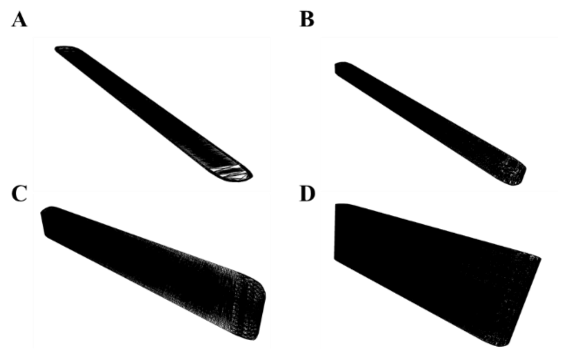


Figure 3. 14: *Output from TetraMesh for line model with A. 2 slices with 1.0 mm spacing, B. 2 slices with 5.0 mm spacing, C. 10 slices with 1.0 mm spacing, and D. 10 slices with 5.0 mm spacing. Plotted using Open3D. Connections were made spanning the line as well as connecting slices in the z-direction. The model was properly represented with this technique.*

The output from TetraMesh was plotted using Open3D and the BPA and Poisson methods were plotted using PyVista. Before plotting the BPA and Poisson meshes, they were saved as STL files.

Unlike the BPA (Figures 3.11 and 3.12) or Poisson algorithm (Figure 3.13), creating the TetraMesh using Open3D (Figure 3.14) connected the vertices in the z-direction of the point cloud, and was able to fill the gap between the edges. The connections in the z-direction ensured that there were no gaps between the points on different slices of the point cloud. It should be noted that although normals were not required to create the TetraMesh, the normals were estimated before calling the BPA and Poisson functions, so when the TetraMesh was made the estimated normals were included.

Once the methods in Open3D were tested, the Delaunay triangulations contained within PyVista were performed. The 2D Delaunay triangulation using PyVista (Figure 3.15) was unable to capture the geometry of the point cloud, no matter the number of slices or the spacing of the slices the output was the same. With the 2D triangulation, the 3D point cloud was essentially compressed into one layer. Therefore, the 2D triangulation method was not a viable option for creating a mesh from the point cloud.

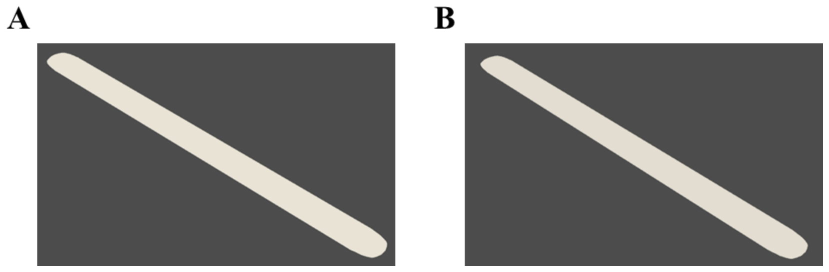


Figure 3. 15: *Output of 2D Delaunay from PyVista from models of A. 2 slices with spacing 5.0 mm, and B. 10 slices with a spacing of 5.0 mm. Plotted using PyVista. The outputs flattened the point cloud to 2D and were unable to represent the line model.*

The 3D Delaunay performed similarly to the TetraMesh shown previously (Figure 3.14). It was successful at creating connections between the edges as well as connecting the points in the z-direction. From a rough estimation for the line point clouds with 10 slices, the 3D Delaunay triangulation took between 1-2 minutes to run while the TetraMesh took between 5-6 minutes to run. This difference was not very large however, it would become more prominent when applied to the CT data, which has more slices and more points within the point cloud. The results from the 3D Delaunay are shown in Figure 3.16.

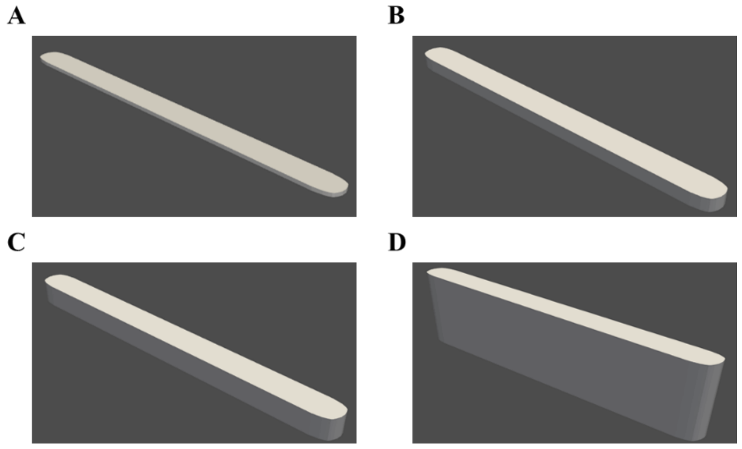


Figure 3. 16: *Output of 3D Delaunay tetrahedralization using PyVista with models A. 2 slices with 1.0 mm spacing, B. 2 slices with 5.0 mm, C. 10 slices with 1.0 mm spacing, and D. 10 slices with 5.0 mm spacing. The method successfully connected points in the z-direction and spanned between the edges of the line.*

Based on the outputs described above, the BPA, Poisson, and 2D Delaunay were removed as viable methods. The BPA and Poisson had the potential to create viable meshes however, many parameters would have to be refined for each specific CT image set inputted into the program. This would mean for each set of images, parameters such as the ball radius for the BPA and the depth and scale for Poisson would have to be estimated. In addition, both the BPA and the Poisson algorithm assumed that normals were present in the point cloud. For the current project, the point cloud is represented by the set of 3D coordinates and did not contain normals, the normals needed to be estimated before these algorithms were run. The estimation of the normals adds an additional step to the process and can introduce unknown errors.

To decide between the Open3D TetraMesha and the PyVista 3D Delaunay functions, more tests were run on the more complicated cylindrical point cloud model shown in

Figure 3. 10. The desired output mesh would connect the inner and outer rings of the cylinder and create connections in the z-direction, such that the mesh looked like a hollow cylinder or a pipe. Both methods were run on point clouds created with 20, 40, and 60 slices and spacings of 1.0 and 5.0 mm. A rough time estimate for the duration of the meshing method was determined using a stopwatch. The output meshes using Delaunay 3D and TetraMesh for the 1.0 mm spacing for 20, 40, and 60 slices are shown in Figure 3.17.

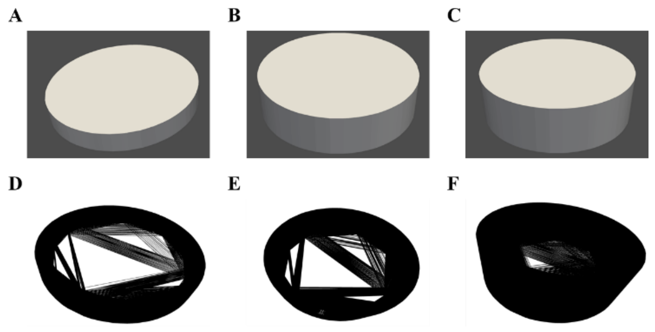


Figure 3. 17: *Output of Delaunay tetrahedralization using PyVista on cylindrical model with 1.0 mm spacing and A. 20 slices, B. 40 slices, and C. 60 slices. The output of Open3D TetraMesh of the cylindrical model with 1.0 mm spacing and D. 20 slices, E. 40 slices, and F. 60 slices. The Delaunay 3D created a solid cylinder and the TetraMesh created a hollow cylinder with connections spanning the inner radius.*

The 3D Delaunay (Figure 3.17 A, B, and C) created what appeared to be a solid cylinder while the TetraMesh (Figure 3.17 D, E, and F) had connections between the inner and outer rings but there were also connections through the centre spanning the inner ring. With the 1.0 mm spacing, the TetraMesh took a longer amount of time than the Delaunay 3D, however at 5.0 mm spacing the times were more similar to one another than at 1.0

mm spacing, and in some cases, for the 5.0 mm spacing, the TetraMesh took a shorter amount of time.

The function to create the TetraMesh only took the point cloud as an input however, the 3D Delaunay function from Pyvista had four parameters other than the point cloud that could be modified. The first parameter was alpha, which represents the radius of a circumsphere such that any vertices, edges, face, or tetrahedra within the circumsphere will be outputted, the default value is 0, and with the default, only the tetrahedra are outputted (The PyVista Developers n.d.; VTK 2021). The alpha value was based on alpha shapes (VTK 2021). The second optional parameter was the tolerance, which was represented as a fraction of the diagonal of the bounding box, controlling the removal of points that were closely spaced. The offset controlled the size of the initial bounding Delaunay triangulation (The PyVista Developers n.d.). The initial tests were done with the alpha value of zero meaning that the output was only the tetrahedra (Figure 3.17 A, B, and C). The parameter alpha was modified to determine if the desired hollow cylinder mesh could be achieved, the goal was to determine the alpha value that could be used so that the triangulation between the layers was maintained while removing the triangles spanning the inner portion of the inner surface of the cylinder.

To determine the effect of alpha on achieving the hollow cylinder, a cylinder point cloud model was created with 10 slices, a circle radius of 150 pixels, a line thickness of 10 pixels, and spacing between the slices of 1.0 mm (Figure 3.18) and 5.0 mm (Figure 3.19). The mesh generation was run with alpha values of 0.1, 0.5, 1.0, 5.0, 10.0, and 20.0. These values were chosen such that there were values equal, less than, and greater than the spacing between the slices as well as the thickness between the inner and outer edge of the cylinder. With values less than the spacing between the slices, it was expected that the slices would not be connected, while with those equal to or greater than the spacing between the slices the circumsphere with radius alpha would encompass and include the mesh connections between the slices. A similar logic was assumed for the connections between the inner and outer edges of the cylinder, with alpha values less than the distance between the edges the connections would not be maintained, but with connections greater than or equal to the distance the connections would be outputted.

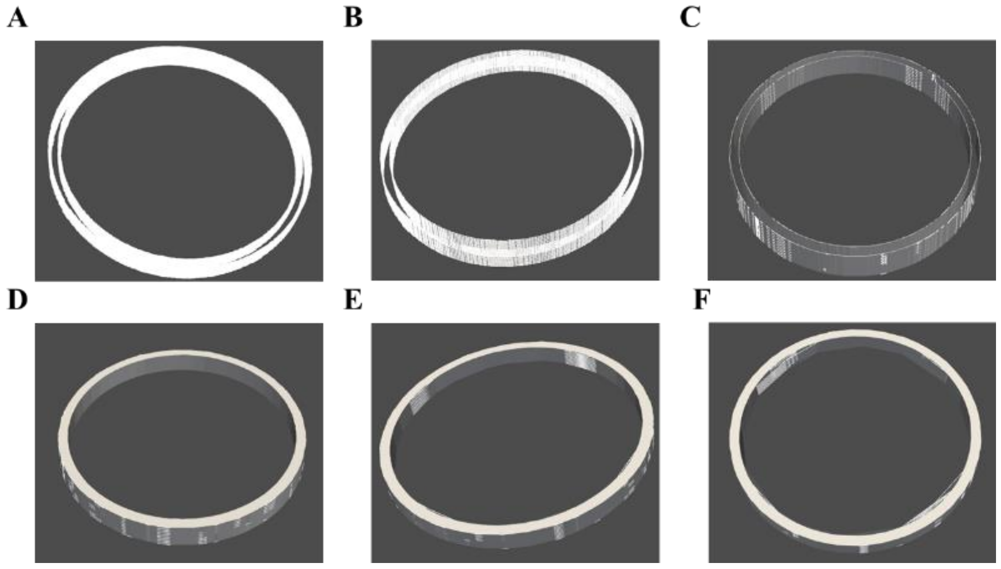


Figure 3. 18: *Output of PyVista's Delaunay 3D for cylinder mesh with 10 slices and 1.0 mm slice spacing with an alpha value of A. 0.1, B. 0.5, C. 1.0, D. 5.0, E. 10.0, and F. 20.0. The meshes with an alpha value less than 1.0 mm (A, B) had missing connections in the z-direction, and values greater than 1.0 mm (C, D, E, F) had connections in the z-direction. Meshes with alpha greater than or equal to the distance between the inner and outer surfaces (D, E, F) successfully connected the inner and outer surfaces of the cylinder.*

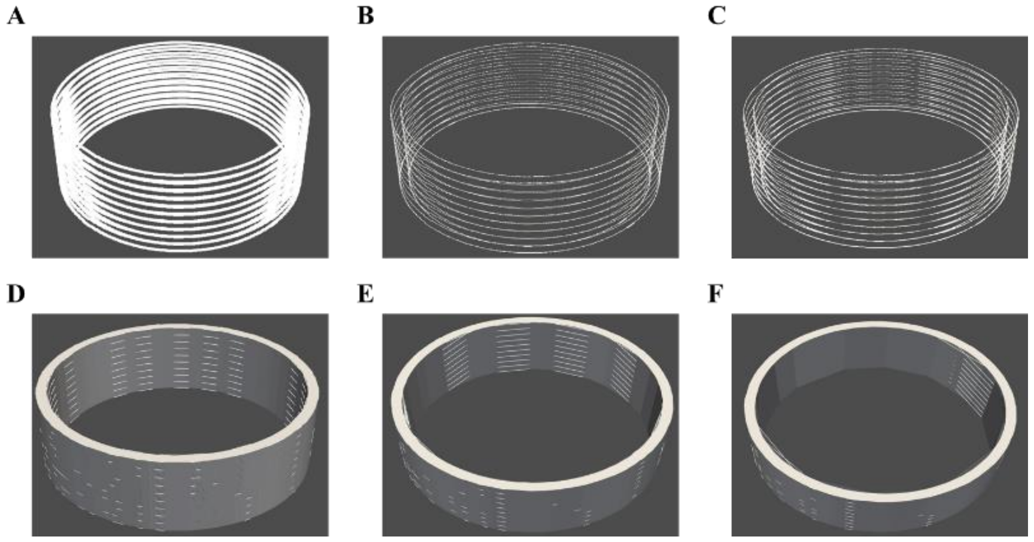


Figure 3. 19: *Output of PyVista's Delaunay 3D for cylinder mesh with 10 slices and 5.0 mm slice spacing with an alpha value of A. 0.1, B. 0.5, C. 1.0, D. 5.0, E. 10.0, and F. 20.0. Meshes with an alpha value less than 5.0 mm (A, B, C) did not have connections in the z-direction, while those greater than or equal to 5.0 mm (D, E, F) had connections in the z-direction. Only meshes with alpha values greater than or equal to the distance between the inner and outer surfaces (D, E, F) had connections spanning the inner and outer surfaces.*

Based on the pixel spacing chosen (0.4492,0.4492), the distance between the inner and outer edges was 4.492 mm. When alpha was less than the spacing between the slices, 0.1 and 0.5 for the 1.0 mm spacing, and 0.1, 0.5, and 1.0 for the 5.0 mm spacing, the outputted mesh appeared similar to the original cylinder point cloud (see Figures 3.18 and 3.19) where there were individual circles for each slice and slices were not connected. When the alpha value was greater than or equal to the spacing between the slices the individual layers became connected (Figures 3.18 and 3.19). Based on Figures 3. 18 and 3. 19, increasing alpha beyond the spacing between the slices did not affect the connections in the z-direction. When alpha was less than the spacing between the inner and outer edges of the cylinder (alpha values of 0.1,0.5, and 1.0), the edges were not connected by the mesh such that the outer and inner edges appeared as two separate circles. With alpha values greater than the spacing between the two edges (values of 5.0, 10.0, and 20.0), the edges were connected by the mesh. The values of 10.0 and 20.0 which were more than twice and four times the distance between the edges, respectively, created connections between portions of the inner circle that were not as prominent in the mesh created with an alpha value of 5.0 mm. These connections are made more visible in Figure 3.20, where the connections are highlighted by a red circle.

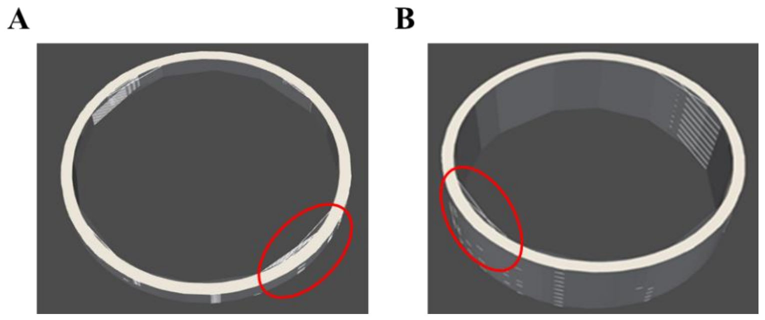


Figure 3. 20: *Output of Delaunay 3D from PyVista with an alpha value of 20.0 and spacing between slices of A. 1.0 mm and B. 5.0 mm. The red circles indicate areas where the mesh is connected through the middle of the inner radius that occurred when alpha was much larger than the distance between the inner and outer surfaces.*

3.5.2 Testing on Point Clouds from CT Images of Head

In comparison to other meshing techniques outlined throughout this chapter, the Delaunay 3D with an alpha value close to the spacing between the slices and the distance between the edges of the inner and outer cylinder gave the best results. This method was applied to the point clouds of two CT image sets to test whether this method would be effective on the higher complexity shape. For this description, the two CT image stacks were described as Skull1 and Skull2. The image stack from Skull1 contained 28 images, a spacing between the slices of 5.015 mm, a pixel spacing of (0.4492, 0.4492), and the corresponding point cloud contained 84,210 points. The image stack for Skull2 contained 77 images with a spacing between slices of 2.5 mm, a pixel spacing of (0.352,0.352), and the corresponding point cloud of size 282,938. As a note, after the meshing method was chosen, the rest of this project used Skull2 as the CT image stack used to test and refine the methodology.

After the meshing technique was applied to the point cloud extracted from the two sets of CT images of the head, the resultant meshes appeared to capture the shape of the point cloud. Holes within the mesh are noticeable, particularly when comparing the top-down views of Skull2 for the alpha values of 2.5 and 5.0 mm (Figure 3. 22 A and B). As the alpha value was increased from 2.5 to 5.0, some of the holes were filled. Following the principle behind alpha shapes, as the alpha value was increased the level of detail of the skull that was captured with the mesh decreased. The decrease in detail was apparent in the meshes produced from Skull2 when alpha was increased from 2.5 to 20.0. At an alpha value of 20.0, both meshes based on Skull1 and Skull2 lost the definition of the orbital sockets and the nasal bone of the skull. The large differences in meshes at different alpha values required the ideal value of alpha to be used to create the mesh, this parameter would be specified for each point cloud.

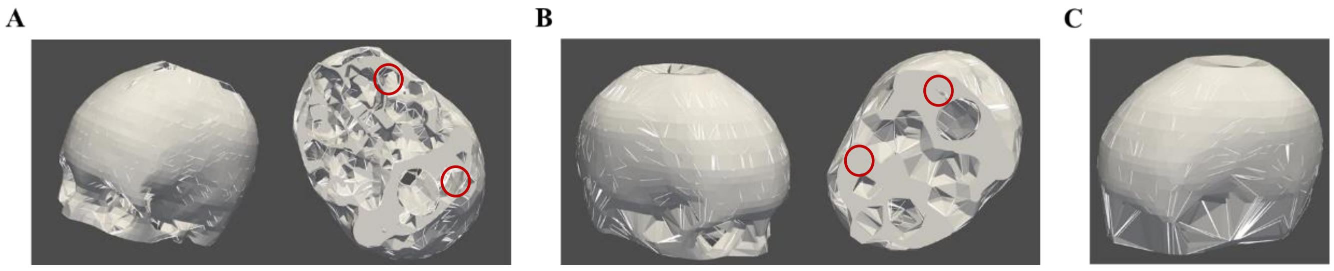


Figure 3. 21: *Output mesh using point cloud from Skull1 with alpha values of A. 5.0, B. 10.0, and C. 20.0. The detail present in the mesh decreases as the alpha value increases. Some holes (shown in red circles) are filled when alpha increases from 5.0 mm in A to 10.0 mm in B.*

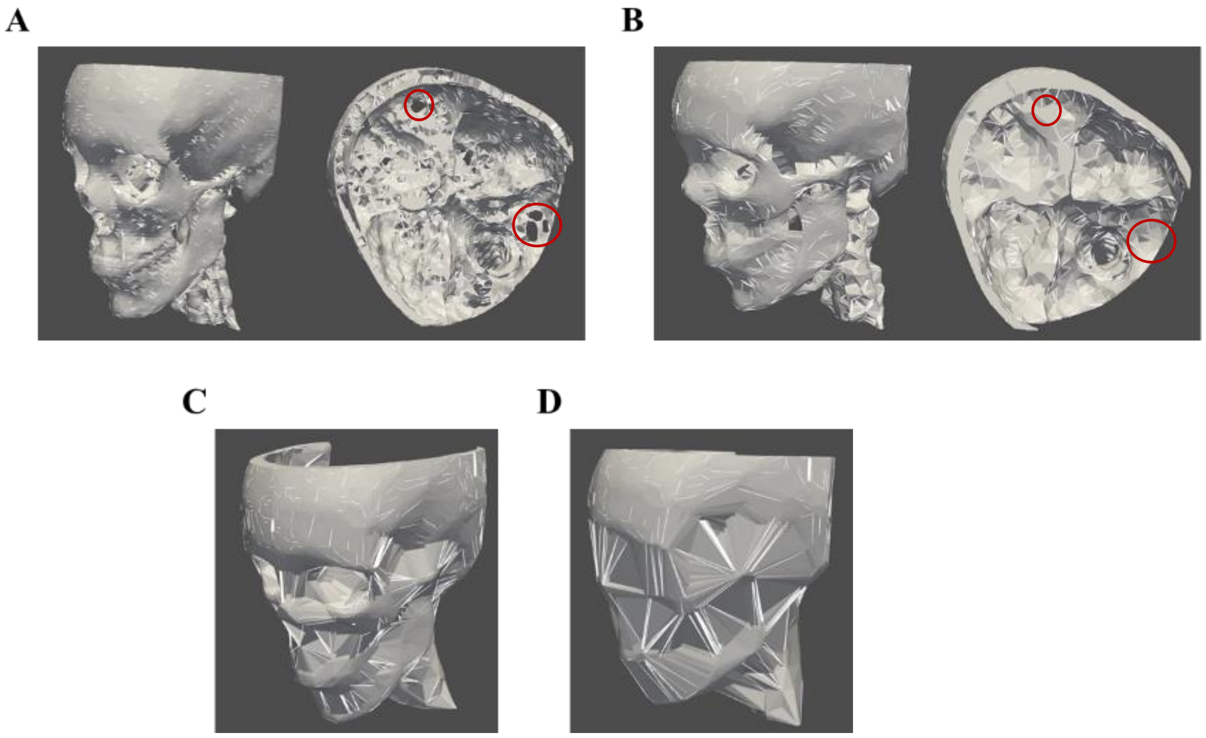


Figure 3. 22: *Output mesh using point cloud from Skull2 and Delaunay 3D using PyVista with an alpha of A. 2.5, B. 5.0, C.10.0, D.20.0. The detail present in the mesh decreases as the alpha value increases. Some holes (shown in red circles) are filled when alpha increases, shown for the transition from an alpha of 2.5 mm to 5.0 mm (A to B).*

The alpha parameter followed the principle of alpha shapes, three-Dimensional alpha shapes were described by Edelsbrunner and Mücke in 1994 (Edelsbrunner and Mucke

1994). Alpha shapes are derived from Delaunay triangulations and generalize the convex hull of a set of points, where α determines the amount of detail of the shape. The value α defines the radius of an α-ball when α approaches infinity the alpha shape approaches the convex hull of the point cloud, and when α=0 the alpha shape is simply the point cloud (Edelsbrunner and Mucke 1994; Gardiner, Behnsen, and Brassey 2018). When α decreases, portions of the alpha shape are removed if spheres of radius α do not contain any points of the set. In other words, when α is very small the α-ball passes through the point set without intersecting any points. The alpha shape develops cavities creating multiple shapes which can lead to holes being formed (Edelsbrunner and Mucke 1994; Gardiner, Behnsen, and Brassey 2018). With large values of α, the volume of the alpha-shape is larger than the object, while the volume of the alpha-shape for small values of α is smaller than the object represented by the point set (Gardiner, Behnsen, and Brassey 2018).

The alpha parameter used in the mesh creation was calculated based on the paper by Gardiner, Behnsen, and Brassey (Gardiner, Behnsen, and Brassey 2018). The authors of this paper used alpha shapes to determine the shape complexity of mammalian bacula. The authors took micro-CT images of the bacula, binarized and filled the holes in the images, and extracted a point cloud. The alpha radii used to create the alpha shapes was calculated based on a reference length of the point cloud multiplied by a coefficient. The reference length used was the average distance between a point and its nearest 100 neighbours, while a range of coefficients were tested, and the optimal coefficient was considered the value that produced an alpha-shape with a volume equal to that of the CT voxel volume.

The alpha value used in this project was calculated as the average of the 100 nearest neighbours of the point cloud. This differed from the study by Gardiner et al. (Gardiner, Behnsen, and Brassey 2018) as it did not use a coefficient as a scaling parameter. The nearest neighbours were calculated in the same manner as described previously, where a KD Tree was produced and queried using the function contained in the spatial sub-package of the SciPy module (The SciPy Community 2021b).

The final mesh method chosen to be used was the Delaunay 3D method from PyVista using an alpha value equal to the average distance of the 100 nearest neighbours of all the points. The resultant meshes had their surfaces extracted and cleaned before being saved as STL files.

3.6 Processing Time in Python

The time to apply the edge detection, extract the surface points and create a mesh was determined for the skull, previously referred to as Skull2, and a portion of a femur in Python. The time to open the CT images was not included because after the CT images of the lower limbs were opened, they were cropped so that the rest of the process was only applied to a portion of the left femur. A set of CT images of the lower limb were retrieved from TCIA (Clark et al. 2013; Vallières et al. 2015a, 2015b), containing a set of 267 axial CT images which spanned from the femur to the plantar surface of the foot.

A portion of the femur was chosen to test the sampling and the mesh creation algorithms. The structure was simpler in comparison to the skull, which improved the ability of the volume to be measured. However, there was still sufficient curvature within the bone to allow for the testing of the targeted sampling. To isolate just a portion of the femur, the CT images were imported into Python as NumPy arrays based on the CT import function created. The slices used from the CT image were slices 2 to 90, the columns chosen were 0 to 255, and the rows chosen were 0 to 345. Segmenting this portion of the arrays, before the edge detection was applied, isolated a portion of the left femur, and removed the gantry table that the individual was laying on from the images. The threshold used to isolate the cortical bone of the femur was 600 HU, chosen based on the findings of Aamodt et al. (Aamodt et al. 1999) for the proximal femur of a human cadaver. A similar HU was determined for the cortical bone of the proximal humerus by Fat et al. (Lim Fat et al. 2012). The point cloud created from the segmented femur contained 16, 264 points. The resultant meshes created when determining the processing time are shown in Figure 3.23.

The time to apply the threshold, edge detection, extract the point cloud, and create the mesh was 1275.4 seconds or 21.3 minutes for the skull (Figure 3.23 A), and 20.6 seconds for the cropped images of the femur (Figure 3.23 B). This process was run in the Shell window in the Python 3.8 IDLE and on a remote desktop with an Intel Core i7-10700 2.90 GHz 64-bit processor with 64.0 GB installed RAM.

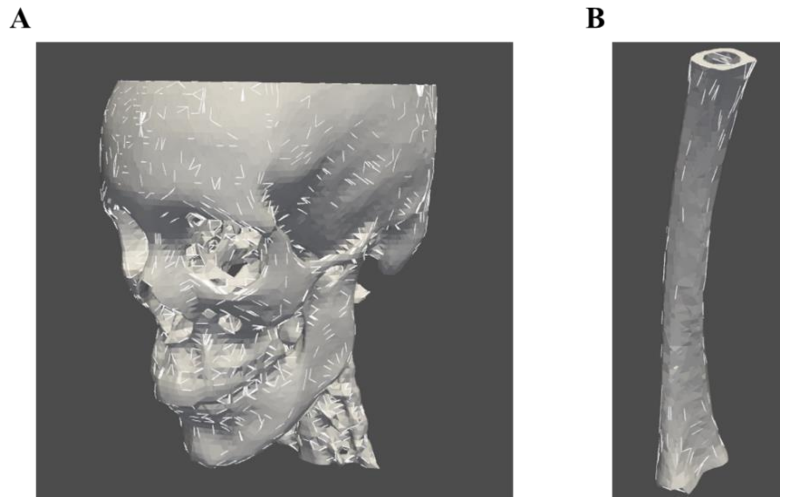


Figure 3. 23: *Resultant meshes created from the method being applied to A. Skull2 and B. a portion of a set of CT images of the lower limbs. The processing time to create these meshes was measured, as A. 21.3 min, and B. 20.6 s.*

3.7 Comparison of Meshes

A quantitative comparisons method was needed to determine if the quality of the mesh deteriorated when created from a down-sampled point cloud and to determine the down-sampling method that distorted the mesh the least relative to the full point cloud. The comparisons method was chosen to quantify the differences between the meshes, allowing for the comparison between the different sampling methods and for the ideal parameters for the targeted sampling to be determined.

3.7.1 Comparison of Point Clouds

Initially, the full and the down-sampled point clouds were compared directly, meaning that the mesh was not created. The functions *Cloud-to-Cloud distance* and the *M3C2* plugin in CloudCompare (version 3.15.0) (Girardeau-Montaut n.d.) were initially used to compute the distances between the point clouds. After some preliminary testing, it was decided that comparing the point clouds directly would not be able to quantify the differences between the sampling methods. One of the main concerns over computing the distances between the point clouds was that one of the point clouds was a sampled set of the reference or original point cloud, as a subset of the reference mesh the distances may be skewed as many of the points were the same, and this could result in shorter distances. In addition to the calculated distances being skewed, the end goal of this project was to create a mesh representation of the surface of the bone from the CT data. This mesh was composed of vertices, edges, and triangles, and so comparing the point clouds directly would have just been a comparison of the vertices and would not reflect the true changes in the surface of the mesh caused by changes in the edges and the polygons. As the resultant output of the algorithm used for this thesis was the mesh created, the meshes needed to be directly compared to assess the viability of downsampling the original point cloud.

3.7.2 Comparison by Volume Measures

Comparison of the meshes was initially done by measuring the volume of the STL files produced from the meshes. Three different software were used to measure the volume, PyVista, Meshmixer (Autodesk Inc.,v 3.5.474), and PreForm (Formlabs, v 3.15.0). PyVista is the Python package that was used to create the meshes, the volume of the mesh was determined by accessing the volume property of that particular mesh using *mesh.volume*, where *mesh* is the name of the mesh object, and this returned the total volume (The PyVista Developers 2021). The volume outputted by PyVista was in the same units as the model. The models created in the project were in mm, such that the outputted volume was in mm^3. It should be noted that the volume of the mesh in PyVista was the volume of the original mesh before the surface was extracted and exported as an STL. Once the mesh was converted to an STL file, Meshmixer and PreForm were used to measure the volume. Meshmixer is a mesh manipulation software, the volume (in mm^3) was determined through the stability measure in the analysis tab of the program. The third program used to determine the volume was PreForm, this is a 3D printing preparation software where the STL file can be imported and manipulated before 3D printing (Formlabs n.d.). PreForm reported the volume of the model in mL or cm^3. This software was chosen as a future aim is to 3D print the resultant models created through the algorithm developed in this project. To determine the differences between the volume measurements of the different software, a model of a cube was created similarly to the cylinder described previously. To create the 3D point cloud of the cube, an all-black image was created using NumPy (Harris et al. 2020) by creating a 512x512 array of zeros. OpenCV (cv2) (Bradski 2000) was then used to draw a white square (100x100 pixels) on the black image. The image with a black background and white square was repeated to create an image stack with the desired number of slices and spacing between the slices. The resultant image stack was passed through the function to create the 3D point cloud. However, the edge detection portion was removed so that the point cloud created was dense, the point cloud is shown in Figure 3.24.

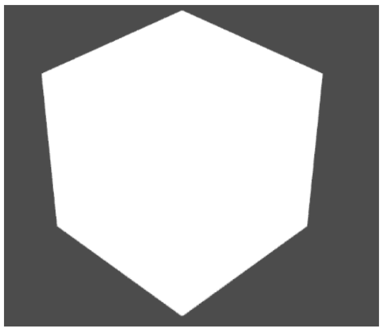


Figure 3. 24: *Dense point cloud of a cube with 100 slices and slice spacing of 0.352mm. This point cloud was created to test the volume measurement techniques.*

Five different sets of meshes involving the cube were created (Figure 3.24, Table 3.1), where the spacing between slices was defined as 0.352 mm (this is the pixel spacing of one of the CT image set Skull2): 1) one cube of size 100x100 pixels with 100 slices, 2) two cubes of size 100x100 pixels and 100 slices, 3) one cube of size 100x100 pixels with a cube of size 50x50 pixels removed from the middle with 100 slices, 4) rectangular prism of size 100x100 pixels, 100 slices and twice the spacing between the slices (i.e. approximately 0.703 mm), and 5) one cube of size 100x100 pixels with 50 slices and twice the spacing between the slices. The volume of these five meshes was measured using the three different techniques, and the percentage difference from the known volume was computed (Table 3.2).

Table 3. 1: *Parameters used to create the various meshes involving the cube.*

Mesh	**Number of Squares in Image Stack**	**Size of Square Base of Prism (pixels x pixels)**	**Number of Slices**	**Spacing Between Slices (mm)**
1	1	100x100	100	0.352
2	2	100x100	100	0.352
3	1	100x100 – 50x50	100	0.352
4	1	100x100	100	0.703
5	1	100x100	50	0.703

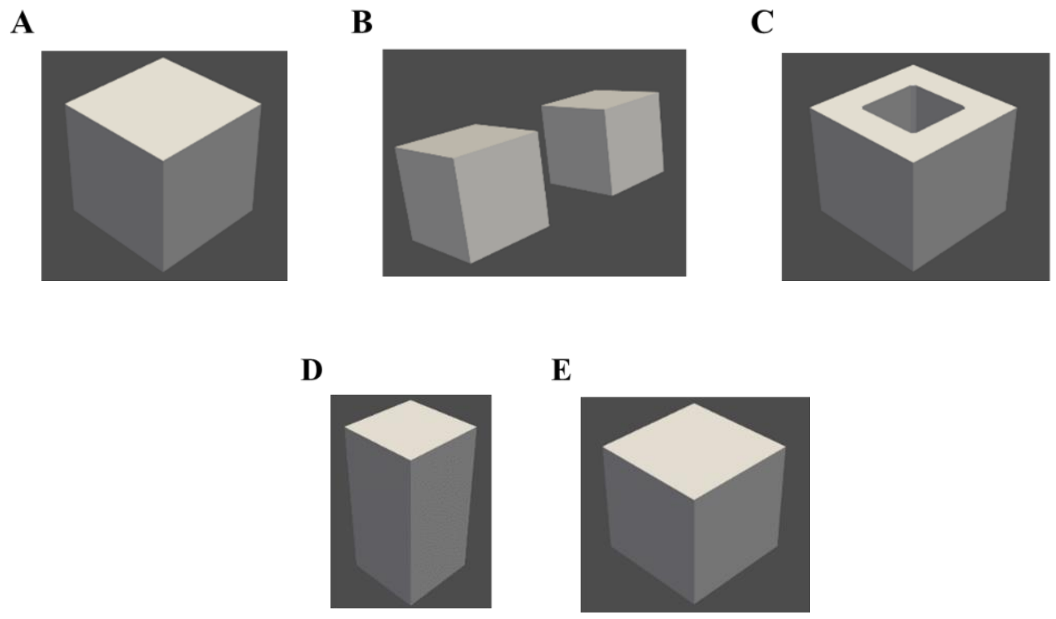


Figure 3. 25: *Images of meshes described in Table 3.1, A. 1, B. 2, C. 3, D. 4, and E. 6. These meshes were used to test the different methods used to calculate the volume of the mesh.*

Table 3. 2: *Volume Measurements of five created meshes. Values in square brackets ([]) denote the percentage difference between the theoretically calculated volume and the volume calculated for that software. (Note: volumes were converted from mm3 to cm3).*

Mesh	**Theoretical Volume (cm^3)**	**Volume from PyVista (cm^3)**	**Volume from Meshmixer (cm^3)**	**Volume from PreForm (cm^3)**
1	43.45	43.02 [-0.99%]	43.02 [-0.99%]	43.51 [0.14%]
2	86.90	86.03 [-1.0%]	86.03 [-1.0%]	87.02 [0.13%]
3	21.73	28.25 [30%]	28.34 [30.42%]	32.50 [49.56%]
4	86.90	86.03 [-1.0%]	86.03 [-1.0%]	87.02 [0.13%]
5	43.45	42.58 [-2.0%]	42.58 [-2.0%]	43.07 [-0.89%]

The volume calculations done by PyVista and Meshmixer often resulted in the same or nearly the same value (Table 3.2). The volumes from PyVista and Meshmixer were lower than that of the theoretical volume by approximately 1.0% for meshes 1,2, and 4, and

2.0% for mesh 5. The volume obtained through PreForm was higher than the calculated value for all of the meshes except for mesh 5 and varied from the calculated volume by less than 1.0% of the theoretical value.

When the number of slices was halved, and the pixel spacing was doubled (i.e., for the fifth mesh), the absolute volume difference for all of the three measurement techniques was greater. For PyVista and MeshMixer the increase was almost two times, while for PreForm the increase was approximately by a factor of 6. The increase in volume was not present when the number of slices remained the same but the spacing between the slices increased, as was done in the fourth mesh. The difference between the fourth and the fifth mesh was that the point cloud density of the 5th mesh was lower than that of the 4th mesh. This suggested that the discrepancy in volume from the calculated value is influenced by the density of the point cloud used to create the mesh. The third mesh, which contained the hole in the centre, had volumes measured by PyVista, Meshmixer, and Preform that were all larger than the calculated value. PyVista and Meshmixer recorded volumes that were higher by approximately 30%, while PreForm was higher by almost 50% of the calculated value. The inner edges of this mesh around the volume that was removed from the centre of the cube had rounded corners. The removal of the sharp corners attributed to the increase in volume calculated by the various methods. The increase in volume caused by the removal of the central cube-shaped volume was troubling when the method was to be applied to the model of the skull, the mesh of the skull was much more complex in geometry and contained more holes than the third mesh tested here. With the complexity of the mesh of the skull, the discrepancies in volume would likely be exacerbated. It was determined that measuring volume in this manner was not suitable to compare the mesh accuracy.

3.7.3 Residual Volume

As the direct volume measurement comparison proved unsuccessful, a method of computing the residual volume between the two STL file models was attempted. These models can have morphological differences as well as dimensional differences. The changes in shape are not always captured using a distance metric and vary depending on what is considered the reference model (George et al. 2017). A method to compare morphological differences is computing the residual volume. If the STL models are each considered a subset of 3D space, the residual volume can then be calculated as the difference between the union and the intersection of the two sets of the models (George et al. 2017). This method was used by Cai et al. (Cai et al. 2015) as a metric to assess the differences in topologies of 3D STL models developed from CT images with different simulated radiation doses. The principal of residual volume was applied to the STL files of the original and the sampled meshes using Meshmixer, MeshLab (version 2020.12) (Cignoni et al. 2008), and PyVista.

In Meshmixer, the method to determine the residual volume was to select two meshes and find the Boolean Union, and the Boolean Intersection using the corresponding tools. The Boolean difference could then be used to find the difference between the union and the intersection of the two volumes. Unfortunately, when this was attempted using Meshmixer, the union and intersections were unsuccessful; the program reported that an unknown fatal error occurred, and it was unable to be executed.

A similar process was attempted using PyVista using the Boolean operations union, intersection, and difference that could be applied to meshes (The PyVista Developers n.d.). When these PyVista operations were tested in the Python shell IDLE, the operations were not completed; the program crashed, and the shell restarted automatically.

The final program that was used to try and calculate the residual volume was MeshLab using the Constructive Solid Geometry (CSG) operation filter (Rocchini et al. 2001). This filter can be used to perform the union, intersection, and difference operations. When the operations of union and intersection were performed, an error occurred with the message 'non-watertight mesh'.

Due to these failed tests, residual volume measures could not be used to compare mesh accuracy.

3.7.4 Distance Measures Between Meshes

3.7.4.1 Software Methods

The initial programs used to compare the meshes included CloudCompare, Meshmixer, and MeshLab. CloudCompare is an open-sourced software that was originally developed to compare point clouds obtained from laser scanners. The software can compare two 3D point clouds or a point cloud and a triangular mesh (Girardeau-Montaut n.d.). The method tested in CloudCompare included the *Cloud-to-Mesh* distance. For the cloud-mesh distance, the inputs can be one point cloud and one mesh or two meshes. If the input is two meshes, one mesh is set as the reference mesh and the other is set as the compared mesh. When computing the distances for this function, the vertices are used from the compared mesh, and distances are computed relative to the nearest polygons of the reference mesh (Cloud-to-mesh distance 2015). The distance computed to the nearest triangle was either the orthogonal distance from the vertex of the compared mesh to the plane of the triangle if the vertex can be projected onto the plane, or it was the distance to the nearest edge of the triangle (Cloud to mesh distances 2017).

Meshlab is another software that can measure the difference between meshes. The computation of the difference between meshes is an implementation of Metro (Cignoni, Rocchini, and Scopigno 1998) which was developed to compare two triangular meshes one being the original and one being a simplified mesh. Through Metro, the compared mesh is sampled, and for each sample, the distance between the sampling point and all faces in the second mesh are determined (Cignoni, Rocchini, and Scopigno 1998). The filter in MeshLab computes the Hausdorff distance between the two meshes (ALoopingIcon n.d.). The Hausdorff distance between two triangular meshes can be described by first describing the distance between two surfaces, S and S'. This distance is denoted as d(S, S'). The Hausdorff distance between S and S' is the maximum distance between the points p in S and the surface S'. The distance between p and S' can be found as the minimum value of the Euclidean norm of the difference between p and the points in S', p' (Aspert, Santa-Cruz, and Ebrahimi 2002). The Hausdorff distance is not a symmetric measure, the symmetrical Hausdorff distance is defined as the maximum value between the Hausdorff distance of d(S, S') and d(S', S) (Aspert, Santa-Cruz, and Ebrahimi 2002). The Hausdorff distance between two discrete triangular meshes, M and M', the triangles in M, T, are sampled to create a regular grid within the triangles, the distances between the point p in M and the triangles in M', T', can be calculated. When p is projected orthogonally onto the plane of T', if the projected point lies within the triangle the distance is calculated as a point-to-plane distance but if it is projected outside the triangle the distance is the distance from p to the closest point on the triangle (Aspert, Santa-Cruz, and Ebrahimi 2002). This definition of the distance is similar to that used by CloudCompare. The implementation of this process in MeshLab samples points over one mesh and finds the closest points on the targeted mesh to these sampled points and computes the Hausdorff distance, the result greatly depends on the number of points that are sampled. The calculated Hausdorff distance is one-sided, the output will change based on which mesh is set as the sampled and targeted meshes (ALoopingIcon n.d.).

Both software options were applied successfully to the meshes created through the process of the project. Instead of using these methods, a custom algorithm was written to determine the distances between the edges of the compared mesh and the vertices on the reference mesh. The custom approach was chosen to allow for a clear understanding of the process that was being applied and modifications to be made. For the application of the comparison between sampling methods, both methods were applied on a slice-by-slice basis to sample the point cloud in the x and y directions but not the z-direction. Sampling in the z-direction was avoided because the spacing between each slice in the point cloud was larger than the minimum spacing between the points in-plane, for example for Skull2 the minimum distance between points in-plane was the pixel spacing of 0.352 mm, whereas in the z-direction was 2.5 mm. Sampling was not done in the z-direction to preserve the shape in that direction. Since the sampling was only done in-plane, only the distances between in-plane edges were to be measured, the custom algorithm allowed these distances to be measured and not include edges that connected slices.

3.7.4.2 Custom Algorithm to Calculate Distance from Midpoint of Edge

Due to the challenges with comparing the volumes that were faced, a custom function was developed in Python that acted like the comparisons done using MeshLab and CloudCompare. The function was called *compare_meshes* and took two meshes before the surface has been extracted and had been exported as an STL, one mesh was the reference mesh, and the other was the mesh to be compared to the reference mesh. Both the reference mesh and the compared mesh had their surfaces extracted, the surfaces were retriangulated, and cleaned to help remove errors in the mesh. The lines that made up the edges of the compared mesh surface were extracted. The lines are saved in a list where the first value was the n number of points that made up the edge. The following n values in the list contained the indices of the vertices of the mesh that made up the line. For each edge that had start and end points in the same plane, meaning the z-value was the same and the edge did not connect in the z-direction, the midpoint between the two endpoints of the line was determined. Figure 3.26 demonstrates the midpoints that were chosen. The black lines in the figure represented the edges of the mesh, the black dots were the vertices of the mesh, and the red dots are the midpoints that were used for the comparison of meshes. The mesh in Figure 3.26 was composed of two z-levels, Z1 and Z2, for the comparison of the meshes the edges chosen for comparison were on the same z-level, in the figure the edges chosen are demonstrated by the red dots representing their midpoints. None of the vertices that connect Z1 and Z2 were chosen for the calculations.

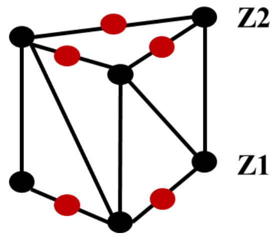


***Figure 3. 26**: Demonstration of edges chosen during the custom mesh comparison algorithm. The midpoints (red dots) represent the edges that were chosen, these edges connect two vertices on the same z-level (i.e., do not connect Z1 and Z2).*

Using the two endpoints of the line, an equation in the form $y = mx + b$ for the line was determined where $m = A/B$, this equation was rearranged into the form $-Ax + By - C = 0$, where $C = bB$. From all the points in the reference mesh on the same z-level as the start and endpoints, the 10 nearest neighbours to the midpoint were determined. From those 10 neighbours, the perpendicular distance from these points to the edge was found for points that were within a radius of half the length of the edge from the midpoint. Limiting the calculation of the distance to the points within a certain radius helped remove outliers and points that were on different surfaces of the mesh. The distance from the point to the defined line was calculated as:

$$d = \frac{|-Ax_n + By_n - C|}{\sqrt{A^2 + B^2}},$$

where x_n and y_n are the x and y coordinates of the neighbour (Bourne 2021).

The function had four outputs:

- a matrix containing the calculated distances for every line and the corresponding nearest neighbours to the midpoint of the edge

- a matrix containing the vertices of the compared mesh
- a matrix containing the vertices of the reference mesh
- a matrix containing the indices of the start and endpoints from the vertices of the compared mesh as well as the index of the point in the reference mesh from which the corresponding distance was calculated

The output containing the indices of the distances was organized such that each row corresponded to a row in the list of calculated distances. Within each row, the first value was the index in the matrix of compared mesh vertices that contained the start point. The second value was the index in the matrix of compared mesh vertices that contained the endpoint. The third value was the index in the matrix of reference mesh vertices for which the distance value from that point to the line was calculated.

3.8 Comparison of Sampling Algorithms

3.8.1 Design of Experiments

The purpose of this set of experiments was to compare the two methods of down-sampling, pseudo uniform sampling, and the targeted grid sampling based on the angular similarity, as well as to determine the parameters used by the targeted grid sampling algorithm. These parameters included: the ranges of angular similarity, the percentages sampled, and the size of the grid in the x-y plane. The 'ideal' parameters were to be considered those that achieved a mesh with the closest approximation to the mesh created from the full point cloud.

3.8.1.1 Parameters

For the tests on the pseudo uniform sampling, the percentages sampled were 90, 70, 50, 30, 20, and 10 percent of the original point cloud.

The parameters within the targeted grid sampling algorithm that can affect the points chosen from the original point cloud included: grid size in the x-y plane, the ranges chosen for the angular similarity, and the percentage sampled for each range of the angular similarity. The grid size in the z-direction, which determined the number of slices or z-levels that were included within each grid, could also affect the sampling. This parameter was set to a constant value of 5 throughout the experiments to reduce the complexity and the number of experiments that were required. The grid size in the x-y plane was defined as the length of the side of a square that made up the grid and was measured in the number of pixels. Referring to section 3.4.2, the grid size was equivalent to the constant *c*, the pixel spacing in the x and y directions were multiplied by this constant (the grid size) to partition the bounding region of the point cloud into a grid. The grid size was demonstrated in Figure 3.27. The squares in this figure represented the pixels and one grid is composed of many of the pixels as represented by the red square. The grid size represented the number of pixels on each side of the grid, in this image the grid size used was 10 so each side of the grid is composed of 10 pixels. The overall length of the grid was then the number of pixels multiplied by the pixel spacing in the x (sp_x) and y (sp_y) directions.

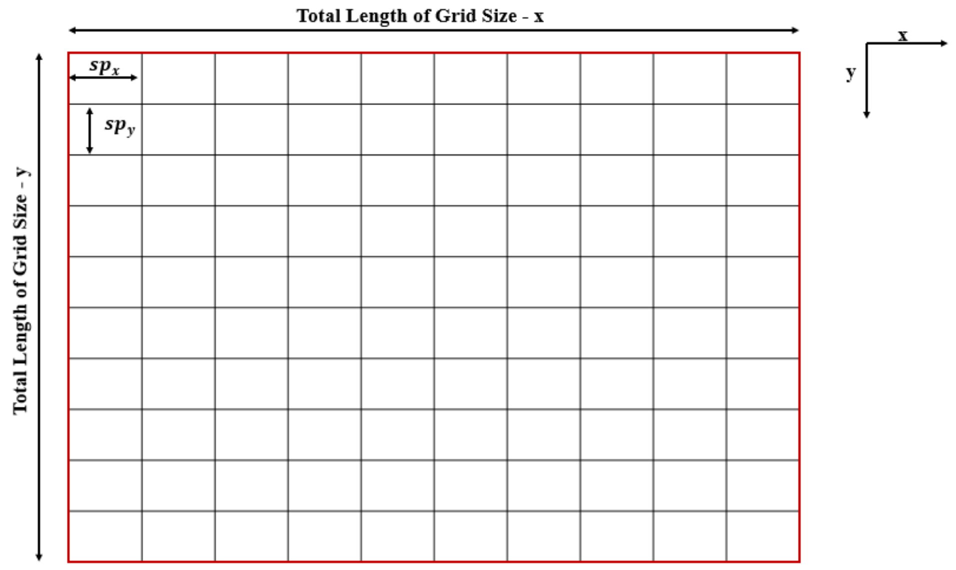


Figure 3. 27: *Representation of pixels that create a grid during the targeted sampling. The red square represents the outline of one grid, the black squares are pixels and the grid size in each direction was determined based on the number of pixels contained in the grid.*

The ranges of angular similarity values were composed of four sets of ranges, and the percentages sampled were defined as a list of percentages where the first percentage corresponded to points sampled in the last angular similarity range, and subsequent percentages corresponded to previous ranges. The pairing of angular similarity ranges and the list of percentages can be seen in Figure 3.28, the arrows represented the connection between the similarity range and the percentage sampled for that range.

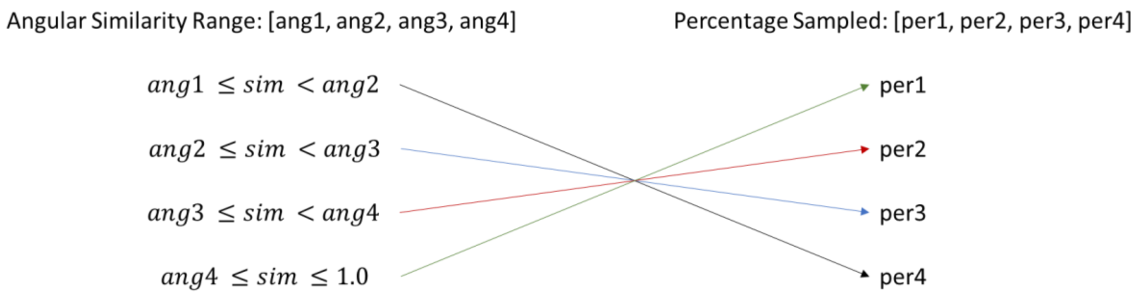


Figure 3. 28: *Relationship between angular similarity range and percentage sampled for that range for targeted sampling algorithm. The arrows connecting the similarity range to the percentage demonstrates that the lowest similarity range was sampled at the highest percentage, and the highest similarity range was sampled at the lowest percentage.*

A high and low value was chosen for the grid size, the similarity intervals, and the percentages sampled. A combination of these values created the first eight experiments, and the ninth experiment was set as the mid-point between the high and low levels for all three of the factors. The high and low values chosen were 10 and 30 pixels for the grid size, [10,25,50,75] and [50,65,80,95] for the percentages for each range, and [0,0.4,0.6,0.8] and [0,0.5.0.7,0.9] for the angular similarity intervals. The similarity intervals were defined as the list of cut-offs used to split up the interval bound by 0 and 1.0. For example, the value described as [0,0.4,0.6,0.8] defined regions of $0 \geq sim < 0.4$, $0.4 \geq sim < 0.6$, $0.6 \geq sim < 0.8$, and $0.8 \geq sim \leq 1.0$, where sim was the angular similarity. The nine experiments in their standard order and including the run orders are summarized in Table 3.3. The run order was chosen using the randint function from the Python Random library (Python Software Foundation 2021c), where an integer between 1 and 9 was chosen randomly until all experiments were assigned a run order.

Table 3. 3: *Standard Order table for the experiments used to compare the meshes created from the output of the targeted grid sampling algorithm to the mesh created from the full point cloud. Values in the table were used as inputs to the targeted sampling algorithm.*

Standard Order	Run Order	Grid Size	Percentages Sampled	Similarity Intervals
1	9	10	[10,25,50,75]	[0,0.4,0.6,0.8]
2	5	30	[10,25,50,75]	[0,0.4,0.6,0.8]
3	1	10	[50,65,80,95]	[0,0.4,0.6,0.8]
4	6	30	[50,65,80,95]	[0,0.4,0.6,0.8]
5	3	10	[10,25,50,75]	[0,0.5,0.7,0.9]
6	4	30	[10,25,50,75]	[0,0.5,0.7,0.9]
7	2	10	[50,65,80,95]	[0,0.5,0.7,0.9]
8	7	30	[50,65,80,95]	[0,0.5,0.7,0.9]
9	8	20	[30,45,65,85]	[0,0.45,0.65,0.85]

3.8.1.2 Outcome Measures

The algorithm that was described in section 3.7.4.2 was used to measure deviations between the meshes of the sampled point cloud and the original mesh. The output of the *compare_meshes* function that contained the list of all the calculated distances was used to determine the mean of the difference between the two meshes. The NumPy package was used to determine the mean using *numpy.mean* (The NumPy Community 2021f). All distance measurements were reported in mm.

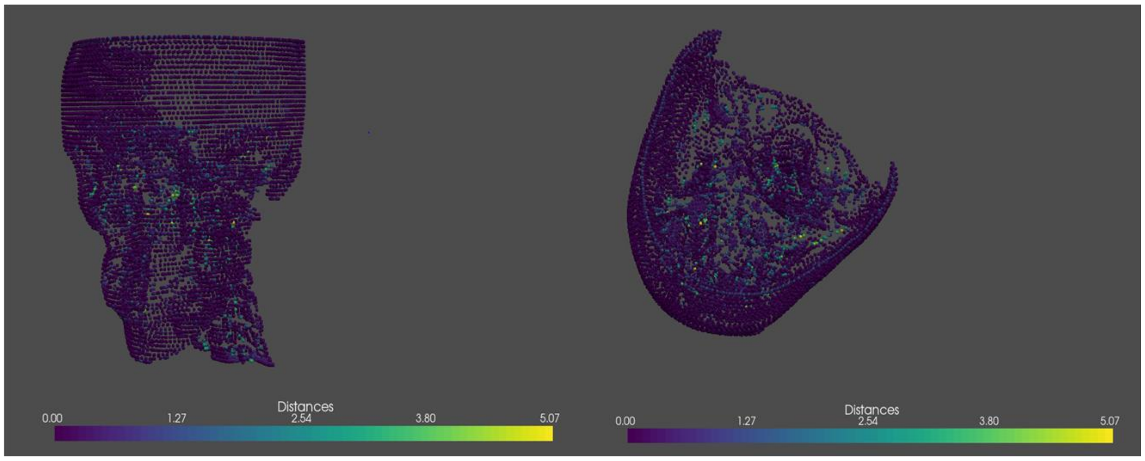
3.6.2 DOE Results

3.6.2.1 Distance Between Reference Mesh and Sampled Meshes

The mesh comparison was applied to compare the pseudo-uniform sampled meshes and the targeted sampled meshes to the full mesh created from the full point cloud. The purpose of the comparison was to quantify the distance between the two meshes. Distances between the midpoint of an edge of the sampled point clouds and the nearest neighbours of the full mesh that were within a radius of half the length of the edge point were determined. For each midpoint of the edge, the average distance was determined by averaging all of the calculated distances with the same start and end edge points.

A point cloud containing every midpoint for the mesh from the sampled point cloud was created, this point cloud consisted of a NumPy matrix of Nx3 where N was the number of midpoints for that particular mesh. The point cloud was converted to a PolyData object in PyVista so that the point cloud could be displayed. The list of average distances for the midpoints was added to the PyVista defined point cloud. When the point cloud was plotted the distances that were added were plotted as scalars. Plotting in this way not only displayed the midpoints of the edge of the mesh but the colour of the displayed point corresponded to the calculated average distance between the sampled and full mesh. This method of plotting the midpoints and their corresponding distances was done for all the pseudo-uniform and the targeted sampled meshes. Histograms for the corresponding average distances at each midpoint were created using Excel (Office 365, Microsoft) for every sampled mesh. The histograms were all set to have the same bins to help visualize the difference between the distribution of the distances as the sampling technique varied. The histograms and heat maps generated were shown for the extremes of the two sampling algorithms, for the pseudo-uniform sampling, this was 10% and 90% sampled, and for the targeted sampling, this was the 1st and the 8th experiments. The extremes were demonstrated in Figure 3.29 for the pseudo-uniform 10% sampled, Figure 3.30 for the pseudo-uniform 90% sampled. Figure 3.31 for the 1st experiment in the targeted sampling, and Figure 3.32 for the 8th targeted sampling experiment. The remaining histograms and heat maps can be found in Appendix B. Within Figure 3.29, 3.30, 3.31, and 3.32, the areas with the largest distances occurred in areas of high curvature, and as the sampling percentage increased the distributions in the histograms moved to the left.

A



B

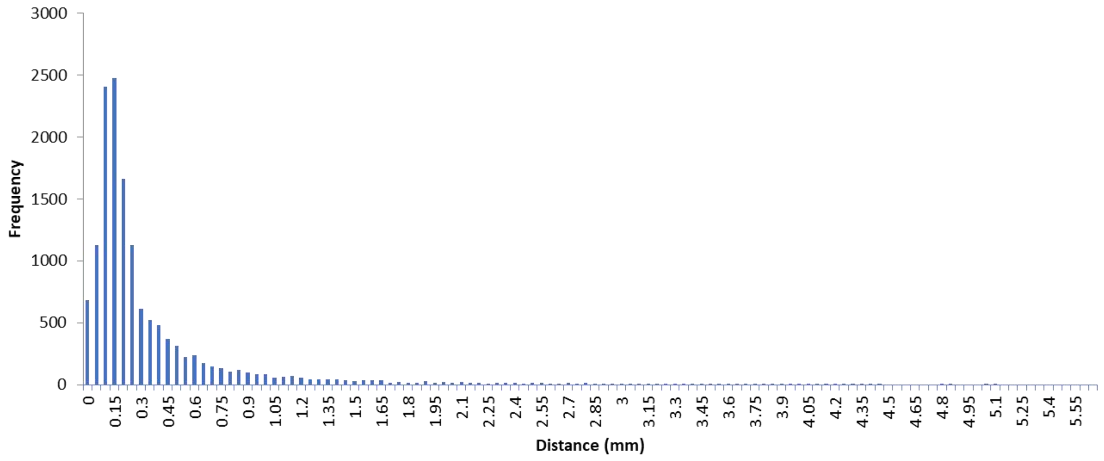
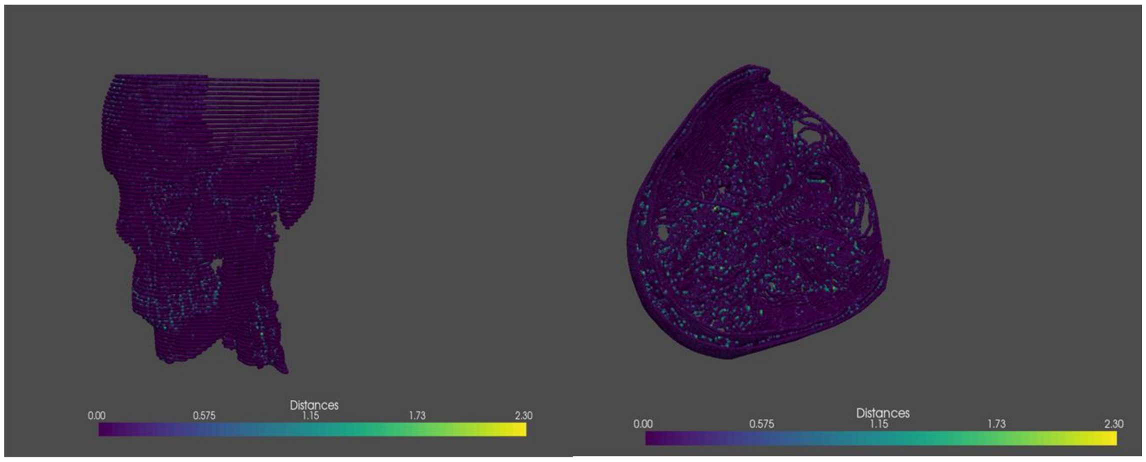


***Figure 3. 29**: Results of mesh comparison between the full mesh and 10% pseudo-uniform sampled mesh, A. heat map, and B. histogram. The larger distances occurred in areas of high curvature. The histogram is not centered around 0 and has a tail that extends further than those of the histograms sampled at higher percentages.*

A



B

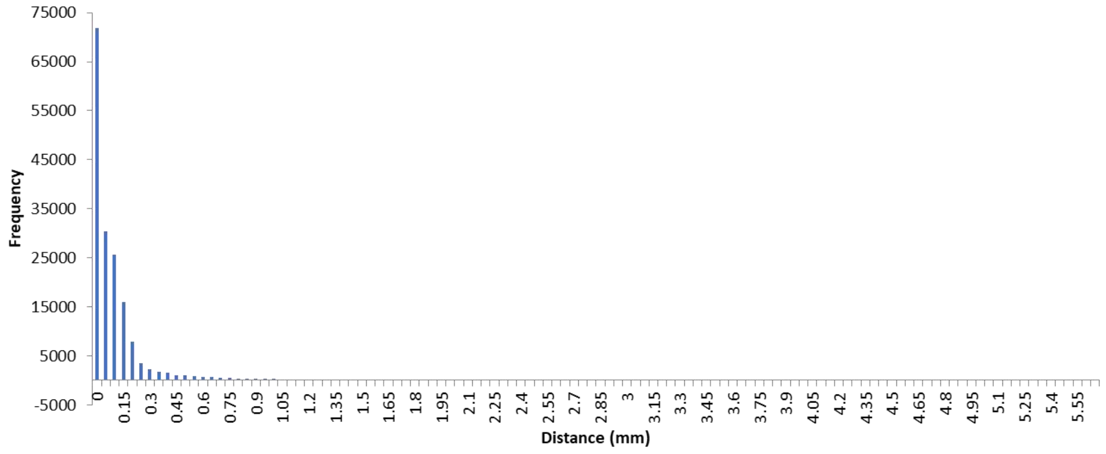
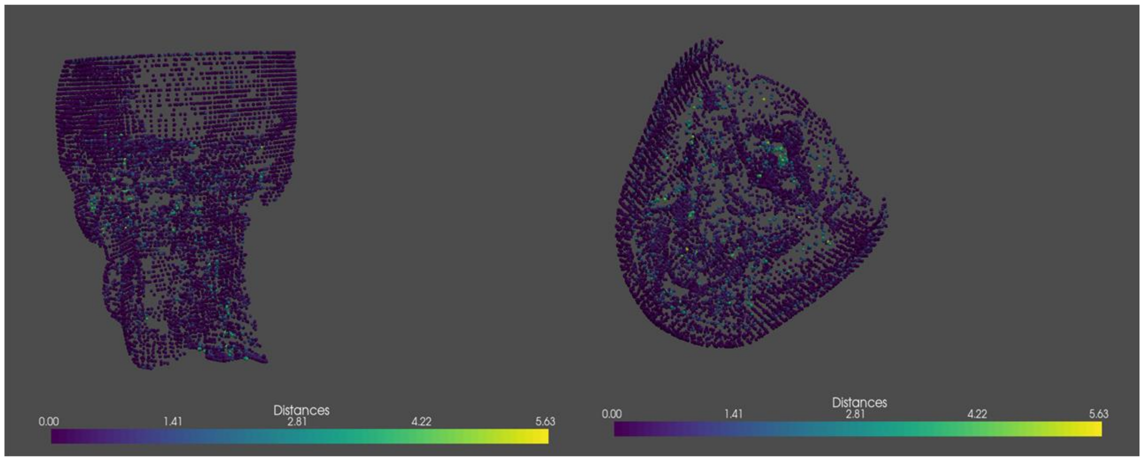


Figure 3. 30: *Results of mesh comparison between the full mesh and 90% pseudo-uniform sampled mesh. A. heat map and B. histogram. The larger distances occurred in areas of high curvature. The histogram is centered around 0 and has a short tail compared to the histograms for those sampled at a lower percentage.*

A



B

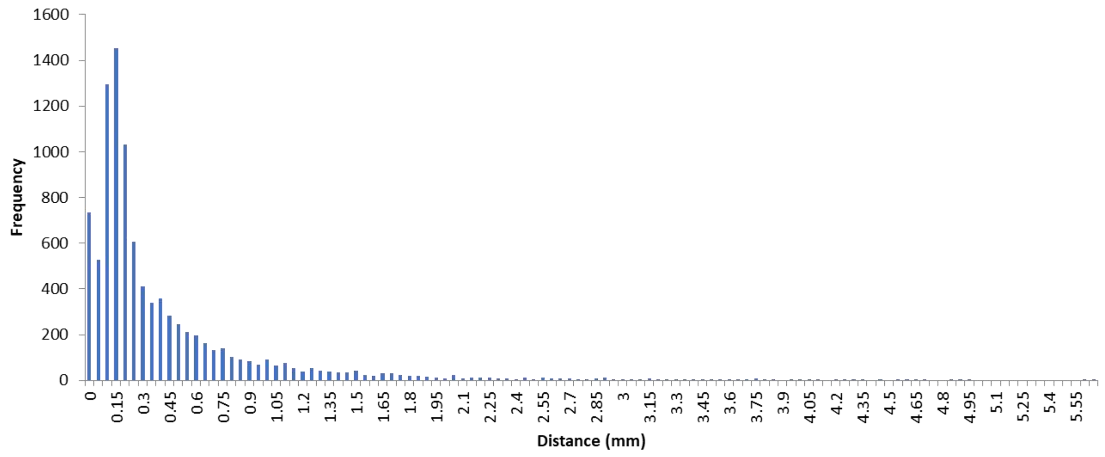
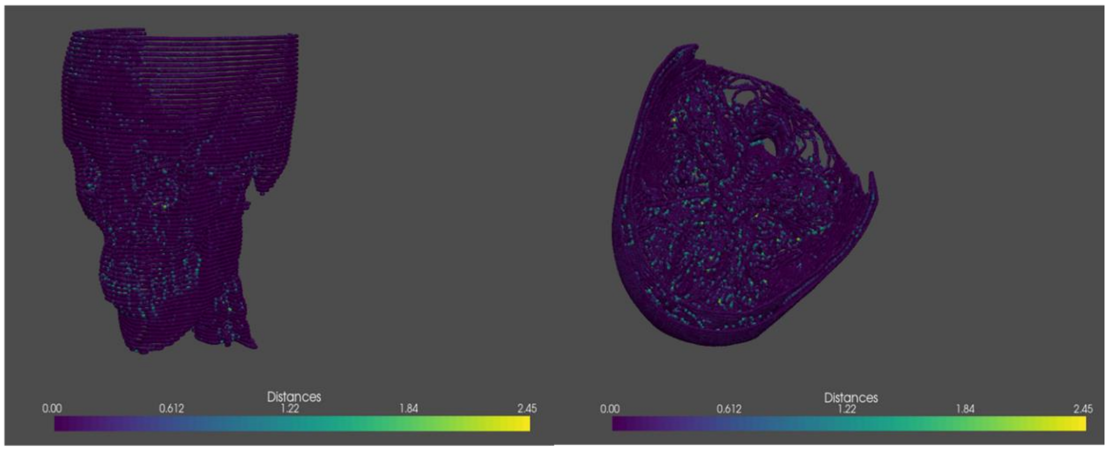


Figure 3. 31: *Results of comparison of meshes between the full mesh and the targeted sampled mesh for the first experiment in the DOE. A. heat map and B. histogram. The larger distances occurred in areas of high curvature. The histogram is not centered around 0 and has a tail that extends further than those of the histograms sampled at higher percentages.*

A



B

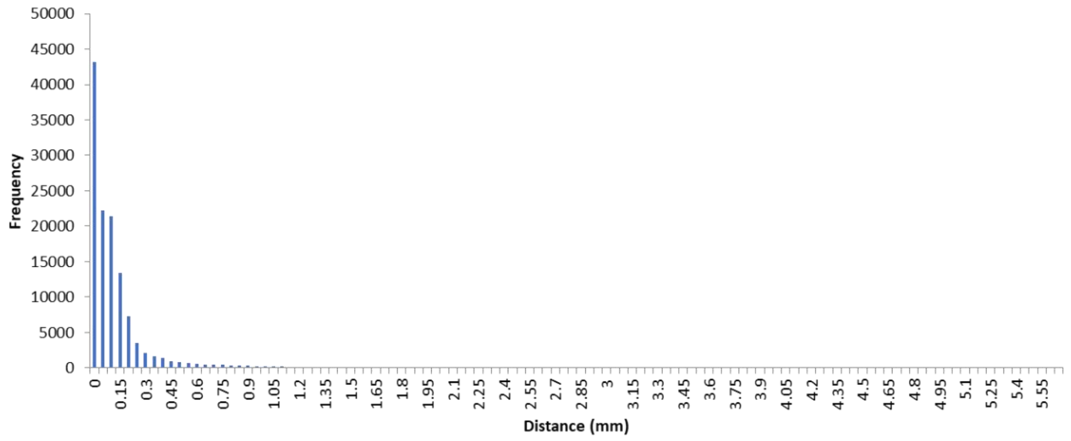


***Figure 3. 32**: Results of comparison of meshes between the full mesh and the targeted sampled mesh for the eighth experiment in the DOE. A. heat map and B. histogram. The larger distances occurred in areas of high curvature. The histogram is centered around 0 and has a short tail compared to the histograms for those sampled at a lower percentage.*

Table 3. 4: *Overall average for all the distances and overall average of the average of each midpoint for the comparison of meshes created after uniform sampling to that created from the full point cloud.*

Sampling Percentage (%)	10	20	30	50	70	90
Overall Average Distance (mm)	0.437	0.334	0.283	0.220	0.189	0.160
Overall Average of Midpoint Distance (mm)	0.340	0.230	0.177	0.126	0.102	0.083

Table 3. 5: *Overall average for all the distances and overall average of the average of each midpoint for the targeted sampling experiments where the [] contains the resultant percentage that was sampled.*

Standard Order for Targeted Sampling [%]	**Overall Average Distance (mm)**	**Overall Average of Midpoint Distance (mm)**
1 [8.04]	0.501	0.403
2 [31.87]	0.292	0.177
3 [49.46]	0.221	0.128
4 [66.93]	0.192	0.108
5 [8.58]	0.492	0.393
6 [38.49]	0.267	0.156
7 [49.84]	0.220	0.128
8 [70.98]	0.186	0.104
9 [44.59]	0.237	0.143

Throughout the uniform and the targeted sampling approaches, the areas with the largest distances were around bone sections with high curvature: the teeth, the vertebral bodies, and surrounding the sinuses (Figures 3.29-3.32). Distance values of 0 (indicated by the colour purple in the heat maps shown in Figures 3.29-3.32) were the most common value along the cranial vault. The areas where the distances were greater were areas of greater complexity, while the smoother shape of the cranial vault was relatively well preserved.

The distribution of distances in the histograms for the pseudo-uniform sampling (Figures 3.29, and 3.30) changed as the percentage sampled changed. As the percentage of points sampled increased from 10% to 90%, the distribution of the distances became more right-skewed, while at the lower percentage of 10% the distribution had a greater frequency of larger distances, and that caused the histogram to broaden. When the data in the histograms moved to the left, distributions became more right-skewed, this indicated that most of the distances were closer to 0 at higher sampling percentages. At the lower percentages, particularly when the point cloud was sampled to 10%, many of the distances were clustered around the bins 0.1, 0.15, and 0.2. The right tails on the distributions for the higher percentages were shorter than those of the lower percentages, this indicated that distances of the lower percentages contained higher values than those of higher sampling percentages. The trend of distances increasing as the sampling percentage decreased was also apparent in the average distances. Both the overall average distance and the average of the average at each midpoint increased when the percentage sampled decreased, with the difference between the 90% sampled and 10% sampled being 0.277 mm and 0.257 mm for the overall average and average of the midpoint average, respectively. The largest average of the midpoint average occurred for the 10% sampled point cloud with a value of 0.340 mm, this value was less than the in-plane pixel spacing of the original image of approximately 0.352 mm. The largest value for the overall average distance also occurred at 10% sampled, with a value of 0.437 mm, this was slightly larger than the pixel spacing.

The distribution in the histograms for the targeted sampling (Figures 3.31, and 3.32) followed the same pattern as the pseudo-uniform sampled point clouds. Where the resultant percentage sampled increased, the histogram became more right-skewed. This was particularly evident when comparing the histograms for 1 and 8: 1 represents the parameters which resulted in the lowest percentage sampled of 8.04%, while 8 represents the highest resultant percentage of 70.98%. The distances measured in 8 were concentrated around 0, while the distances for 1, similar to the 10% pseudo-uniform sampled, were concentrated around the bins 0.1, 0.15, and 0.2. As indicated by the right tails of the histograms, the lower percentages contained higher distances than that of the higher percentages. The largest overall average distance and the average of the midpoint average occurred with the parameters used for the first targeted sampling resulting in a percentage of approximately 8%, both values were larger than the in-plane pixel spacing of 0.352 mm.

It should be noted that although the same colours were used for all the heat map type plots, the actual scale that the colours represented differed between the sampling techniques. For example, the colour scale for the 10% pseudo-uniform sampling was from 0 to 5.07 while the 90% sampled was from 0 to 2.30. The maps were originally plotted on the same colour scale, however, with the higher sampling percentages plotted on the same scale as the lower percentages the distinction between high and low values was difficult to see. The distinction was unclear because the distances for the higher percentages sampled were on the lower end of the scale of the lower sampling percentages.

To compare the targeted sampling ("tar") to the pseudo-uniform ("u") sampling, similar sampled percentages were compared. Specifically, the targeted sampling 5 (tar5 = 8.58%) was compared to the 10% pseudo-uniform (u10), targeted 2 (tar2 = 31.87%) was compared to 30% pseudo-uniform (u30), targeted 7 (tar7 = 49.84%) was compared to 50% pseudo-uniform (u50), and targeted 8 (tar8 = 70.98%) was compared to 70% pseudo-uniform (u70). Although the compared percentages were not the same, they were close enough to provide an approximate comparison between the two methods. The differences between the overall average and the average of the midpoint values were determined, compared, and demonstrated in Table 3.6.

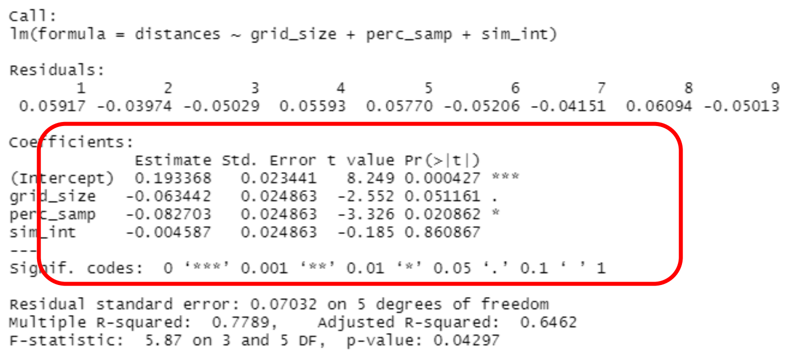
Table 3. 6: *Difference between averages for pseudo-uniform and targeted sampling, where tar# describes the standard order of the targeted sampling algorithm and u# represents the pseudo-uniform where the # is the percentage sampled.*

Compared Methods	**The difference in overall average (mm)**	**The difference in the average of midpoint values (mm)**
tar5 – u10	0.055	0.053
tar2 – u30	0.009	0.0
tar7 – u50	0.0	0.002
tar8 – u70	-0.003	0.002

In all of the cases except for the difference in the overall average between tar8 and u70, the targeted sampled meshes had average distances that were slightly greater or equal to that of similar percentages of the pseudo-uniform sampled methods (Table 3.6). The absolute value of the differences between the techniques were small with the largest being approximately 0.055 mm, and the smallest being approximately 0.0 mm. The small differences between the techniques indicated that there was not a large difference between using the two techniques to downsample the meshes.

3.8.2.2 Linear Regression for Targeted Sampling

The original purpose of the sampling DOE was to determine the values for the parameters that should be used for the targeted sampling. The three parameters were the in-plane grid size, the ranges of angular similarities, and the percentage that each similarity range was sampled. The average of the average distance at each midpoint was found for each targeted sampling approach used. A linear model was fit to these values with the parameters of the targeted sampling function as the independent variables. The model was fit using the R *lm* function (R Core Team 2020). The average of the average distance was set as a vector in the order of the standard order; each parameter was set as a vector in the standard order where the low-level values were set to a value of -1, the high-level values were set to a value of 1, and the midpoint between the high and low levels which was used for the ninth experiment was set to a value of 0. The linear model was then fit to the distance data using a combination of the three parameters.



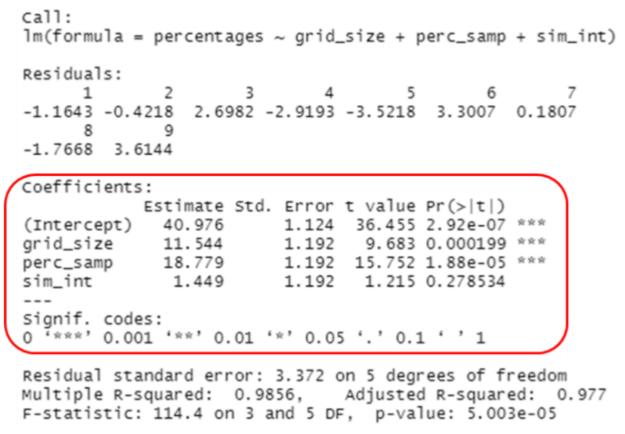
```
Call:
lm(formula = distances ~ grid_size + perc_samp + sim_int)

Residuals:
       1        2        3        4        5        6        7        8        9 
 0.05917 -0.03974 -0.05029  0.05593  0.05770 -0.05206 -0.04151  0.06094 -0.05013 

Coefficients:
             Estimate Std. Error t value Pr(>|t|)    
(Intercept)  0.193368   0.023441   8.249 0.000427 ***
grid_size   -0.063442   0.024863  -2.552 0.051161 .  
perc_samp   -0.082703   0.024863  -3.326 0.020862 *  
sim_int     -0.004587   0.024863  -0.185 0.860867    
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.07032 on 5 degrees of freedom
Multiple R-squared:  0.7789,	Adjusted R-squared:  0.6462 
F-statistic:  5.87 on 3 and 5 DF,  p-value: 0.04297
```

Figure 3. 33: *Summary of the linear model for the average of the average distance for each midpoint created using R. The coefficients of the linear model and the probability that the value is greater than the value t, are highlighted by the red box.*

The above, Figure 3.33, is a sample of the resulting output of creating a linear model for the targeted sampling experiments. The coefficients of the linear model are highlighted by the red box. The column titled “*Estimate*” contains the estimated coefficient for each independent variable used for the model. The coefficients for all three parameters were negative indicating that as the parameter changed from the low level to the high level, the distance between the mesh and the reference mesh created from the full point cloud decreased. The distance between the compared mesh and the reference mesh decreased as grid size, percentages sampled, and angular similarity range increased. For example, the first test of the targeted sampling (tar1) was done with all three parameters at their low level, while the eighth experiment (tar8) used the highest level for all parameters. Ultimately tar1 had the lowest percentage sampled (8.04%), and tar8 had the highest percentage sampled (70.98%). The percentage of points sampled increased as the parameters were changed from low to high levels. The distance between the reference mesh understandably decreased since in meshes with a higher percentage of the original points remaining there was less deviation from the original point cloud, so the distances were lower.

The column labeled Pr(>|t|) indicates the probability that the value will be greater than or equal to the value of t; this is the p-value. The significance codes at the bottom indicate coefficients that were significant based on certain significance levels. Within this linear model, the grid size was significant using a significance level of 0.1. The percentage sampling was significant using a significance of 0.05, and the similarity interval used was not significant for any of the displayed significance levels.

The same process to create a linear model for the distance was applied to the percentages sampled. This was to determine the parameters that affected the percentage sampled using the targeted sampling algorithm.



```
Call:
lm(formula = percentages ~ grid_size + perc_samp + sim_int)

Residuals:
      1       2       3       4       5       6       7 
-1.1643 -0.4218  2.6982 -2.9193 -3.5218  3.3007  0.1807 
      8       9 
-1.7668  3.6144 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept)   40.976      1.124  36.455 2.92e-07 ***
grid_size     11.544      1.192   9.683 0.000199 ***
perc_samp     18.779      1.192  15.752 1.88e-05 ***
sim_int        1.449      1.192   1.215 0.278534    
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.372 on 5 degrees of freedom
Multiple R-squared:  0.9856,	Adjusted R-squared:  0.977 
F-statistic: 114.4 on 3 and 5 DF,  p-value: 5.003e-05
```

Figure 3. 34: *Summary of the linear model applied to the percentages sampled for the targeted sampling experiments. The coefficients of the linear model and the probability that the value is greater than the value t, are highlighted by the red box*

From the linear model fit to the percentages sampled using the targeted sampling (Figure 3.34), the grid size and the percentage sampled within each range of angular similarity were significant with a significance level of 0. The coefficients of the linear model were all positive indicating that as the parameters were changed from their low to high levels the percentage sampled increased.

3.8.3 Comparing Sampling using a Simplified Shape

3.8.3.1 Targeted Vs. Pseudo-Uniform Sampling

After the targeted and the pseudo-uniform sampling were applied to the skull, they were applied to a simpler shape. The sampling for both techniques appeared effective for a complex structure like the skull, which contained a large number of points but testing the methods on a simpler point cloud that was much smaller than the skull was done to compare how the two methods preserved a shape. The simpler shape consisted of one sharp corner (approximately 90º), and the remaining three corners were rounded. The shape was designed in this way such that it had multiple straight edges as well as curved edges with varying radii. This would allow the techniques to be tested on a variety of different edges with varying complexities. The initial shape can be seen in Figure 3.35.

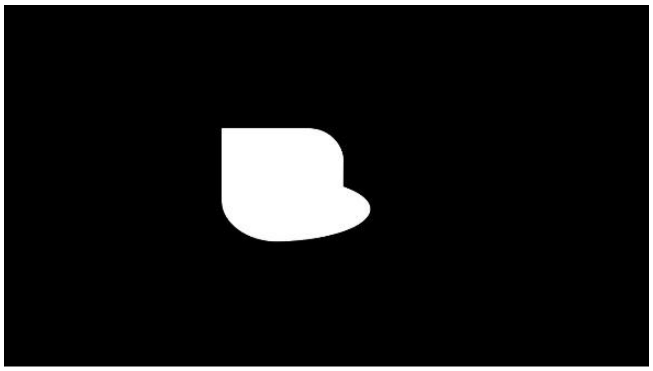


Figure 3. 35: *Simplified shape to test the pseudo-uniform and targeted sampling methods. This shape contained one 90° corner and three corners with varying radii to test if the sampling algorithms were able to preserve shape.*

To extract the point cloud from the image (Figure 3.35), saved as a JPG (Joint Photographic Experts Group) file, was imported into Python using the *imread*() function from the *io* subpackage from scikit-image (scikit-image development team n.d.), and converted to a NumPy array. The image, as a NumPy array, was converted from an RGB image to a grayscale image following the method outlined in the tutorial (How to convert an image from RGB to Grayscale in Python n.d.) where the grayscale image was the output of the dot product between the original image array and the vector [0.2989, 0.5780, 0.1140]. The array was then stacked twice in the third dimension to create a stack of 2 images. The point cloud from the stacked images was determined in the same manner as the function *Extract_Cylinder_PT_Cloud* except that the σ value for the Canny edge detector was modified. For the CT images, the cylinder stack and line stack, the default σ of 1 for the Canny edge detector was sufficient to capture the edges but, with this image, the σ was changed to a value of 3.0 to remove some of the false edges that were detected (Figure 3.36).

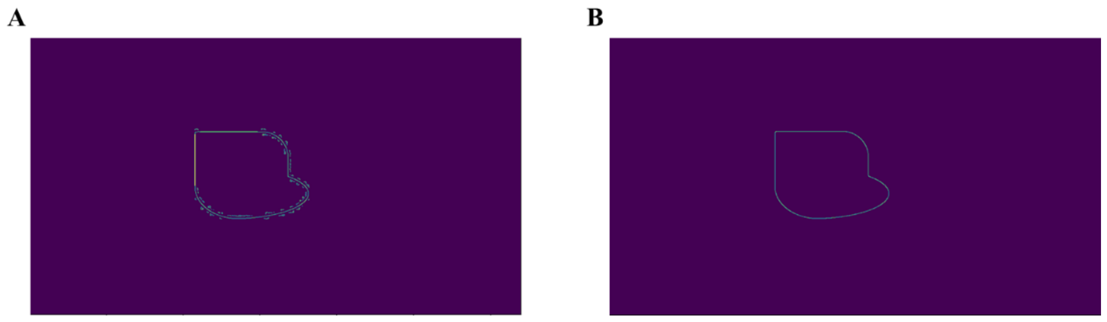


Figure 3. 36: *Simplified object with Canny edge detector applied with A. σ value of 1, and B. σ value of 3. With the σ of 1, some points are falsely identified as edges resulting in noisy edges, while the σ of 3 produced cleaner edges.*

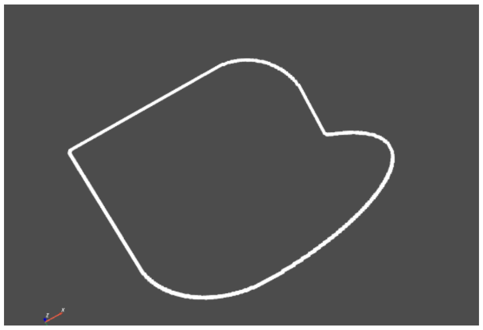


Figure 3. 37: *Resultant point cloud extracted after the Canny edge detector was applied to the stack of simplified images.*

The resulting point cloud (Figure 3.37) contained 1,952 points and was sampled using the pseudo-uniform technique to 10%, 5%, and 2.5% of the original size of the point cloud. The pseudo-uniform sampling algorithm sampled based on the 10 nearest neighbours of the points, so if the desired percentage sampled was less than 10% no points were returned because the number of points to be returned was less than 1. The percentage of 5% was obtained by sampling the 10% sampled point cloud by 50%, and the sample of 2.5% was accomplished by sampling the 5% sampled point cloud again by 50%.

Five different combinations of parameters for the targeted sampling were used, three of these combinations were the same as combinations used in the DOE performed and were tar1, tar2, and tar5, all of which created a resultant point cloud of close to 9% of the original size. Two other targeted sampling combinations were tested to get the percentage sampled close to 2%. These combinations were called tar225 and tar205, which resulted in percentages sampled of 2.25% and 2.05%, respectively. The parameters used for tar225 were a grid size of 5 pixels, angular similarity ranges of [0,0.95,0.99,1.0], and percentages sampled of [10,15,20,25], with the last value of the angular similarity range being 1.0. This meant all grids with angular similarities of 1.0 were sampled to 10%. The parameters used for tar205 were a grid size of 5 pixels, angular similarity ranges of [0,0.4,0.6,0.8], and percentages of [10,25,50,75]. The percentage of 2% was chosen because I wanted to sample the point clouds to a low percentage to evaluate the abilities of the methods in maintaining the shape. I could have sampled as low as 1%, but this would have resulted in approximately 19 points over two slices, and I was worried that despite the sampling method used the shape would not be maintained and the comparison would have been difficult to make.

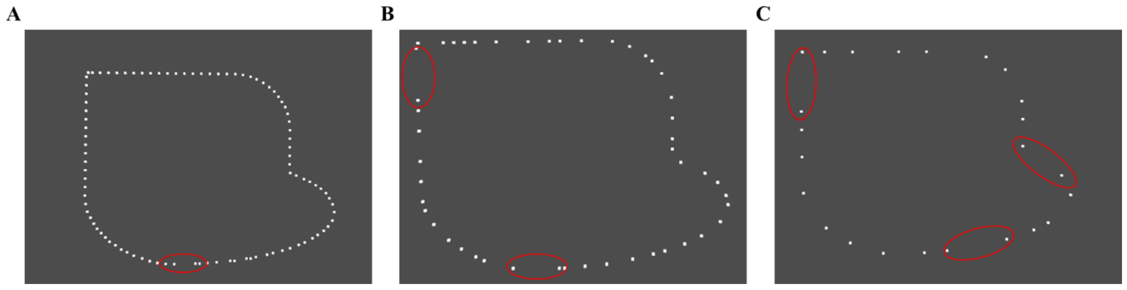


Figure 3. 38: *Pseudo-uniform sampling with A. 9.94% (194 points), B. 4.92% (96 points), and C. 2.46% (48 points). Gaps created during the sampling algorithm are highlighted by red ovals but overall the shape was maintained.*

The pseudo-uniform sampling appeared to maintain the majority of the shape as the sampling decreased from approximately 10% to approximately 2.5%. The algorithm produced gaps within some of the edges as highlighted by the red ovals on the images (Figure 3.38). These gaps would not greatly affect the shape of the straight edges but in the areas with more curvature, this could cause areas of the curves to be converted to straight lines losing some of the curvature. The algorithm checked the 10 nearest neighbours to a point that was on the same z-plane. From these 10 points, a certain percentage was taken, and all of these points were removed, and the process was repeated. The combination of the nearest neighbours, the deletion of points, and the nature of the point cloud having an odd number of points likely caused the gaps within the sampled points. If points were deleted as the function progressed, there would be portions where the number of nearest neighbours was less than 10, and areas where the nearest neighbour might not have been the true nearest neighbour if all the points were considered. Areas with less than 10 points would be unable to be sampled to lower percentages because the number of points to be returned would be a fraction, this would have caused gaps in the sampled points. Although gaps were present, visually the pseudo-uniform sampling maintained the shape of the image.

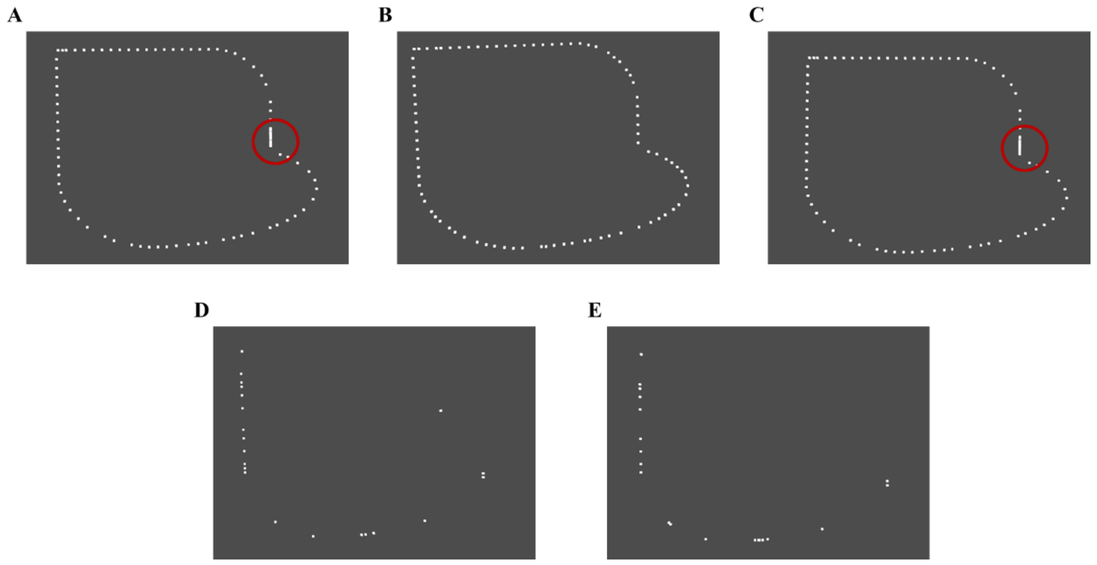


Figure 3. 39: *Sampling with targeted sampling using A. tar1 (9.12%, 178 points), B. tar2 (9.02%, 176 points), C. tar5 (9.32%, 182 points), D. tar225 (2.25%, 44 points), and E. tar205 (2.05%, 40 points). Areas with straight portions were densely sampled (A, C) and highlighted by the red circles. At low percentages (D, E) the shape was not maintained.*

Similar to the pseudo-uniform sampling there appeared to be some gaps between points when the targeted sampling was applied (Figure 3.39). This was unsurprising since the targeted sampling called the pseudo-uniform sampling algorithm at a certain percentage depending on the average angular similarity. In Figure 3.39 A and C, there is a portion highlighted by the red circle that had a dense sampling of points. The samples of tar1 (A) and tar5 (C) had a grid size of 10 while tar2 (B) used a grid size of 30. With the grid size of 10, the grids used in A and C likely contained fewer points along the straight line highlighted in red than B. If the grids contained points on the straight line and along the curved portion below, particularly points along the corner, the average angularity similarity would be lower. With fewer points the 10-pixel grid likely contained points along the straight portion and a few points along the corner, this would have caused the angular similarity to be lower and that particular grid would have been sampled at a higher percentage. Despite appearing to have more gaps than the 10% pseudo-uniform sampled points, A, B, and C visually maintained the shape of the object.

Errors and loss of shape of the object became very apparent when parameters were changed to have a resultant sampling of approximately 2% (images D and E in Figure 3.39). The grid size used for the targeted sampling affected the percentage that was sampled. To achieve a percentage close to 2%, the size of the grid was reduced to 5; although this created point clouds of the desired percentage, the shape was almost entirely lost. In comparison to the pseudo-uniform sampling, the shape was much less apparent in the targeted sampling. The poor maintenance of shape was likely caused by the small grid size as this was the common element between D and E. As the grid size was small it contained fewer points than other grid sizes, since the lowest percentage sampled was set to 10% for the targeted sampling. When the uniform sampling was called at this value if there were less than 10 points within the grid, no points would be returned creating large gaps and great loss of shape. The angular similarity of the grids also would have contributed to the loss of shape. Looking specifically at the sampling done in D, there were 370 grids used in total and 42 of those grids fell into the angular similarity range of $0.0 \geq \text{ang} < 0.95$ while the rest of the 328 grids fell into the range of $1.0 \leq \text{ang}$. The angular similarity could have been so high because there were so few points within each grid. The 328 grids in the last range would have been sampled to 10% and with small grid sizes, the gaps occurred. The majority of the angular similarity being in a high range, meaning they were considered to be similar, was concerning as the shape was made specifically to try and have varying degrees of similarity.

The majority of grids being in a high similarity range may be a fundamental issue with the targeted sampling approach, to see if the same issues would occur on a more complicated structure with many more points, like the skull, the parameters used in the DOE were run once again. The number of points in each range of angular similarity and the average number of points within each grid were recorded and shown in Table 3.7.

Table 3. 7: *Percentage of grids within each range of angular similarity and the average number of points in each grid for each targeted sampling run as part of the DOE for Skull2. The sampling method refers to the standard order from the DOE.*

Sampling Method	**Average Number of Points in Each Grid**	**% of grids in the first range**	**% of grids in the second range**	**% of grids in the third range**	**% of grids in the fourth range**
1	52.85	0.418	0.919	1.79	96.87
2	271.67	0.163	30.99	13.82	55.02
3	52.85	0.418	0.919	1.79	96.87
4	271.67	0.163	30.99	13.82	55.02
5	52.85	0.817	1.44	1.38	96.36
6	271.67	8.86	30.83	8.41	51.90
7	52.85	0.817	1.44	1.38	96.36
8	271.67	8.86	30.83	8.41	51.90
9	154.19	0.240	28.30	6.33	65.13

As seen in Table 3.7, the targeted sampling methods with the grid size of 10 (1,3,5,7) all contain more than 96% of all the grids in the highest range of angular similarity, while those with a grid size of 30 (2,4,6,8) had between approximately 51-55% of the grids in the highest range. The tests that had a larger grid size had a larger number of points within each grid, this contributed to the percentage of grids contained in each angular similarity range being more distributed across all four ranges, as opposed to the low grid size that had most of the grids in the highest range despite the ranges themselves varying. Even with the grid size of 10, the average number of points in each grid was approximately 52, this meant that large gaps created in the sampling in areas with less than 10 points would occur less in the skull point cloud compared to the simplified shape, although there could still have been areas where this may have occurred. Comparing the point cloud of the skull to the simplified shape, the skull point cloud contained more points so it is more likely that the grids would have been more populated with points, which would have reduced the number of large gaps contained in the sampled point cloud.

Ultimately the pseudo-uniform sampling was chosen as the sampling method to be used. The targeted sampling has potential, but some errors within the current algorithm need to be investigated further. The pseudo-uniform sampling appeared to maintain the shape of the simplified shape even at the lowest sampling percentage of approximately 2.5%, while at similar percentages for the targeted sampling the shape all but disappeared. The other advantage of the pseudo-uniform sampling algorithm was that the desired percentage to be sampled could be specified, while it is clear that the different parameters in the targeted sampling affect the percentage sampled, the ideal choice of each parameter for each specific point cloud is difficult and time-consuming to determine. There was also concern over the targeted sampling not functioning properly on the simplified shape, although it appeared to function properly when applied to the skull, if the algorithm did not work properly on a simple shape it was difficult to determine if it could have been translated to more complicated shapes or if the size of the skull point cloud ensured it would be sufficiently sampled to effectively maintain the shape. If the lowest sampling percentage was 10% for the targeted sampling, even if all grids were sampled at that value, the Skull2 point cloud contained such a large number of points that this would likely maintain the shape. Due to this, it was difficult to determine if calculating the angular similarity for Skull2 produced effective sampling or if the number of points in the point cloud was so large that no matter the targeted sampling applied the shape was maintained. The validity of the targeted sampling algorithm as applied to Skull2 was difficult to determine and the lack of shape preserved by this method for the simple shape made me question the true validity of the technique. The pseudo-uniform sampling performed similarly to the targeted sampling on Skull2 in terms of the distance calculations, preserved the shape when applied to the simple shape, and it allowed a percentage to be specified by the user. For these reasons, it was determined that pseudo-uniform sampling was the better choice for down-sampling than targeted sampling.

3.8.3.2 Pseudo-Uniform Sampling vs Random Sampling

A visual comparison was done between pseudo-uniform sampling and random sampling. To create a random sample from the full point cloud, the number of samples to be taken was determined by multiplying the size of the point cloud by the desired percentage to be sampled. This value was rounded up to the nearest integer using the *ceil()* (Python Software Foundation 2021a) function from the Python math module. A list of random numbers between 0 and the number of points in the point cloud was created by calling the *choice()* function from the NumPy subpackage *random*. The first input to the function was the number of points in the point cloud, since this was an integer, the output was generated as a list of samples. The second input was the number of samples desired (i.e., determined by multiplying the number of points by the percentage), and the *replace* input was set to False so that values were not repeated (The NumPy Development Team 2021). The random and the pseudo-uniform sampling were performed on the simplified curved shape to sample to approximately 10% of the original size and shown in Figure 3.40.

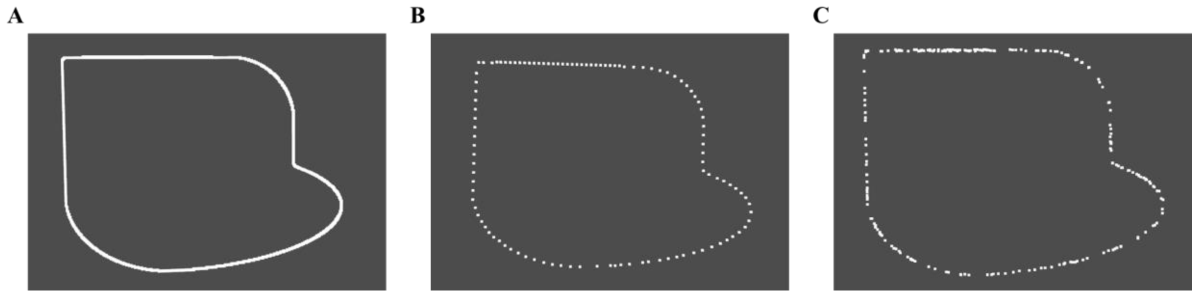


Figure 3. 40: *Resultant point clouds, A. full point cloud, B. 10% pseudo-uniformly sampled, and C. 10% randomly sampled. Both random and pseudo-uniform sampling preserved the shape but, pseudo-uniform was more consistent and the randomly sampled point cloud (C) appeared noisy.*

The random sampling acted on the whole point cloud (across slices) while the pseudo-uniform sampling was performed individually on each slice. Due to this, the pseudo-uniform sampled point cloud was consistent between both slices of the point cloud (Figure 3.40 B). The random sampling on the other hand was inconsistent between slices, which is why the point cloud appears noisier (Figure 3.40 C). Both methods maintained the shape of the point cloud, but the random sampling had lower repeatability such that the output of multiple tests even with the same percentage sampled would yield different points. The lack of repeatability and inconsistent sampling did not guarantee that curves or edges were maintained, so the pseudo-uniform approach was ultimately better because it had more consistency.

3.8.4 Processing Time for Pseudo-Uniform Sampling and Full Point Cloud

The main goal of sampling the point cloud was to reduce the time to create the mesh using PyVista. To determine the efficacy of the pseudo-uniform sampling in reducing the computational time, the time to sample and create the mesh was determined for samples of 90%, 50%, and 10% of the original point cloud. The time to create the mesh from the full point cloud was also measured so that it could be compared to the overall time of sampling and creating the mesh. The computational time for each function was measured using the function from the Python time module *perf_counter()*. This was called before and after the execution of a function, the difference between the outputted time at the end of the execution and the time at the beginning of the function was taken as the execution time of the function in a similar manner done in the tutorial (time.perf_counter() function in Python 2021). This process was completed on a remote desktop with an Intel Core i7-10700 2.90 GHz 64-bit processor with 64.0 GB installed RAM and calculated values are shown in Table 3. 8.

Table 3. 8: *Time to sample, time to create mesh, and total time (time to sample + time to create mesh) for the full point cloud and three uniformly sampled point clouds (90%, 50%, and 10%).*

Percentage of Point Cloud	**Time to Sample (s)**	**Time to Create Mesh (s)**	**Total Time (s)**
100	NA	1216.2 (20.27 min)	1216.2 (20.27 min)
90	9590.6 (159.84 min)	888.5 (14.81 min)	10479.1 (174.65 min)
50	9325.2 (155.42 min)	310.9 (5.18 min)	9636.1 (160.60 min)
10	9424.9 (157.08 min)	32.3 (0.54 min)	9457.1 (157.62 min)

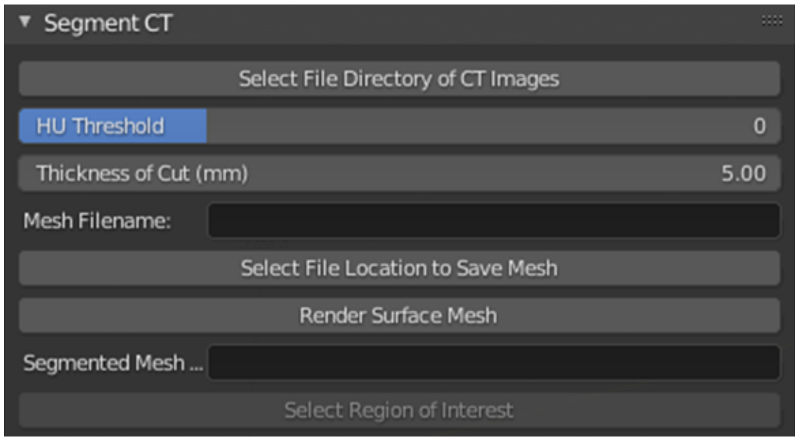
As the percentage of points used to create the mesh decreased, the time to create the mesh decreased (Table 3. 8). When the full point cloud was used, the meshing algorithm took approximately 20 minutes, but when the point cloud was reduced to 10% of its original size, the meshing algorithm took approximately 30 seconds. Although fewer points meant less computational time to create the mesh, the pseudo-uniform sampling algorithm used to create the reduced point cloud had a long processing time when compared to the time to create the mesh from the full point cloud. The time to create the sampled point cloud ranged from approximately 155 to 160 minutes. The long computational time of the sampling algorithm was concerning, and due to the extra time needed to sample, the total time from sampling to creating a mesh was not reduced by introducing the sampling method. This suggested that there are improvements that could be made in the pseudo-uniform sampling process to reduce the time needed to sample. The long processing time was likely caused by the nearest neighbour search being repeated many times within a loop. Future work on the sampling algorithm may lead to improvements to reduce sampling time, such as reducing the time by organizing or sorting the point cloud. The organized point cloud would allow for a uniform sample over the matrix of points which would be less computationally strenuous.

3.9 User Interface

3.9.1 Blender Interface – Buttons

After the process to create the mesh from the point cloud extracted from the set of CT images was established, a user interface was created. The interface was established using Blender (version 2.91.0), an open-source software that enables the user to create and manipulate 3D objects including modelling, rendering, and simulation (Blender Online Community 2018). This software platform was chosen for this project since it had the benefit of being open-source as well as including a Python interface. The extraction of the point cloud and the creation of the mesh that was done previously in Python was used within the interface in Blender to merge the creation of the mesh with software that would allow user manipulation.

To manage the CT images, create the mesh, and save a segmented portion of the mesh, a Blender Add-on was created. In Blender, Add-ons extend the functionality of Blender and scripts created (Blender Foundation n.d.). An Add-on script was created using the Python interface. This Add-on appears as a panel in the 3D viewport of Blender and includes the following options: a button to select the directory location of the set of DICOM images, a slider for the HU value used to binarize the CT images, a slider for the thickness of cut, an input for the user to write the desired name of the mesh created from the CT images, a button to select the location where the mesh will be saved, an input for the filename of the mesh that is created from the segmentation done by the user, and a button to select a region of interest which are used to create the segmented mesh (Figure 3.41).



▾ Segment CT

Select File Directory of CT Images

HU Threshold 0

Thickness of Cut (mm) 5.00

Mesh Filename:

Select File Location to Save Mesh

Render Surface Mesh

Segmented Mesh ...

Select Region of Interest

Figure 3. 41: *Panel Created for Blender Interface Add-on.*

These buttons (Figure 3.41) and the script that is run when they are pressed are described in further detail below. When using the Python API, a Python subclass of a Blender class is created, this allows the script to access variables and functions of the Blender class to interface with Blender. The script to create the Add-on defines subclasses as a member one of three *bpy.types*: Operators, Panels, and Property Groups. When registering the subclass, the definition of the *bpy.types* allows Blender to distinguish between the types (Blender Foundation 2021d). In Blender, buttons, keys, and menus are used to call Operators, which act as tools (Blender Foundation 2021e), and can have *invoke()* and/or *execute()* functions within them. The *invoke()* function initializes the operator at the time it is called based on the context. Properties can be assigned, which are then used by the *execute()* function (Blender Foundation 2021b). The context depends on what is being used in Blender at the moment and contains properties including the scene, the mode (i.e., Object, Edit, etc.), and objects including active objects (Blender Foundation 2021a). The *execute()* function has inputs of the instance of the operator and the current context (Blender Foundation 2021d) and performs the task of the defined operator. Within this project classes which are of the Operator type were used to select the file directories, create the mesh, and select the region of the mesh to be segmented. The Panel type defines and draws new panels and Blender also allows new buttons to be added to pre-existing panels. In the Add-on for this project, the Panel that was drawn contained areas for the user to add input including the names of files, the HU threshold, and the thickness of cut, as well as buttons that applied the Operators when pressed. The last type used is the Property Group, which defined a custom set of properties that can be changed and accessed dynamically through the user interface (Blender Foundation 2021c). The Property Group in this project contained six properties. One of these properties was a float property that contained the thickness of the cut in mm that was used to segment a portion of the model. The thickness of the cut was defined by the user, and when a set of points were selected on the outer surface these points were projected inward by this thickness to extract the desired region. Four string properties were saved: the file directory path for the CT images, the file path where the meshes were saved, the name of the created mesh, and the name of the segmented mesh. The final property saved in the Property Group was an integer property that saved the inputted HU threshold; this property had a default value of 0, a minimum value of -1024, and a maximum value of

3071. Commonly CT scanners use images with a 12-bit depth, this leads to a range of HU of -1024 to 3071 (Glide-Hurst et al. 2013).

3.9.1.1 Selecting Input and Output File Directories

Two separate operators were defined to select the file directory where the set of DICOM images was located and where the user wanted to save the resultant meshes that would be created from the images. Although the two operators functioned in the same way, two were defined so that the panel was created with two separate buttons. The method used to implement the file selection was based on the tutorial (Gangl n.d.). The two operators *SelectInputDirectory*, and *SelectOutputLoc* followed the same steps and process to select a file directory. The file paths chosen when these operators were invoked were saved as the StringProperties *file_direc* and *out_direc* within the Property Group. The process to select the two file-paths file_direc and out_direc is outlined in Figure 3.42, the red square is the button that was pushed by the user. After the button is pushed a file browser pop-up appeared to allow the user to select the file directory. If the file path was a directory it was saved in the Property Group.

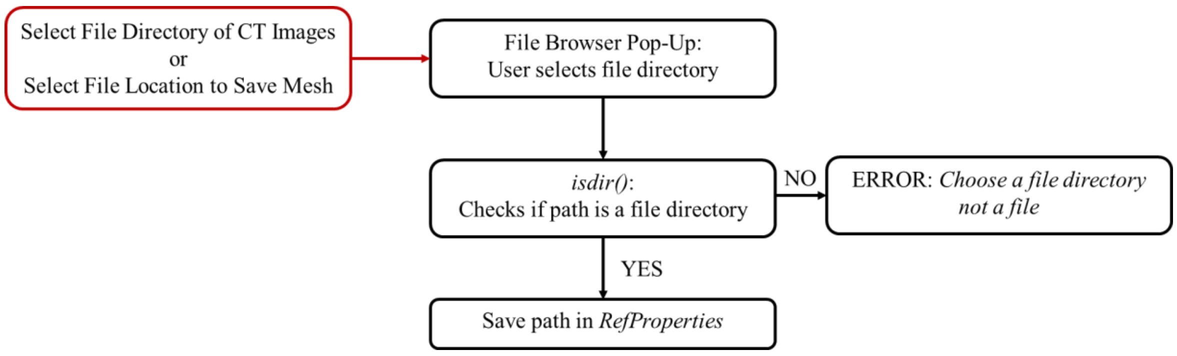

Select File Directory of CT Images
or
Select File Location to Save Mesh
File Browser Pop-Up:
User selects file directory
isdir():
Checks if path is a file directory
NO
ERROR: Choose a file directory not a file
YES
Save path in RefProperties

Figure 3. 42: *Process for choosing a file directory and saving it in the PropertyGroup. The red square was a button push and the black were functions/processes.*

The *ImportHelper* is a class contained within the submodule *bpy_extras*. It was imported by the Python script and was included within the class definition when creating the Operator such that it became a subclass of the Operator (Gangl n.d.). The *ImportHelper* contained an *invoke()* function that opened a file browser. The *invoke()* function was called when the defined Operators were called by the push of a button in the panel. The file path was saved as *self.filepath*, a StringProperty inherited from the *ImportHelper* (Gangl n.d.). The goal of these operators was to determine the file path or file directory where the user had the CT images saved and where they wanted the output mesh to be saved. Once the file browser was opened and the user had selected a path, the function *isdir()* from the Python module *os.path* was used to check if the path selected by the user was a file directory and not a file itself. The *isdir()* function returned True if it was an existing directory (Python Software Foundation 2021b). If the selection was not a directory, an error was thrown and the Operator is cancelled, and if the selection was a directory, the path was saved as a StringProperty in the PropertyGroup *RefProperties*.

3.9.1.2 HU Threshold and Thickness of Cut

The buttons on the panel labelled 'HU Threshold' and 'Thickness of Cut' were created in similar ways. Both buttons contained sliders such that the user could click and drag their mouse along the bar to change the value. Values could also be changed by directly typing the number into the slider box. The first of the two buttons allowed the user to choose the threshold in units of HU that would be applied to the CT images to isolate the bone from surrounding tissue. The second acted as the input for the thickness of the cut that would be made after the user drew a region of interest on the rendered surface mesh. The value in the property group that contained the HU threshold set by the user was defined as *thresh*, this value was described earlier and had a default of 0, a minimum value of -1024, and a maximum value of 3071. The thickness of the cut was defined in mm, this was consistent with the units that were used when creating the 3D point cloud and the model in Python. The scene properties in Blender were set such that the dimensions for length were in mm. This thickness of the cut was the distance along the normal along which the selected points on the outer surface would be projected inward to select the points along the inner surface. The default for the thickness of the cut was 5.0 mm, the minimum value that could be set is 0.0 mm, and there was no maximum value set for the property.

3.9.1.3 Mesh filename

The panel input for the mesh filename took a string inputted by the user that became the filename for the STL created by the Render Surface Mesh button. The String Property was defined in the Property Group as *out_name*, and the user was not required to include the file extension (i.e. .stl) in the filename as it was added once the mesh was saved.

3.9.1.4 Render Surface Mesh

The Render Surface Mesh button in the panel functioned in the same manner as the *create_mesh* function that was described earlier, the process to render the surface mesh is shown in Figure 3.43. The function written in a Python module was ported to the Python interface within Blender. This was accomplished by creating a Blender Operator called *CreateMeshOperator*, the *execute()* function in the Operator called a separately defined function called *Create_Mesh*. The function *Create_Mesh* in Blender varied from the function written in the Python module as it combined the functions *open_CT*, *thresh_edge_CT*, *extract_PT_Cloud*, and *create_mesh* that were described in previous sections. Figure 3. 43 shows the process used in Blender to render the surface mesh. The box outlined in red represents the button being pushed to render the mesh. The squares outlined in black demonstrate conditions that are checked within the function and the functions described previously that are executed. The boxes outlined in blue represent inputs to the functions that are retrieved from buttons and user inputs from the add-on.

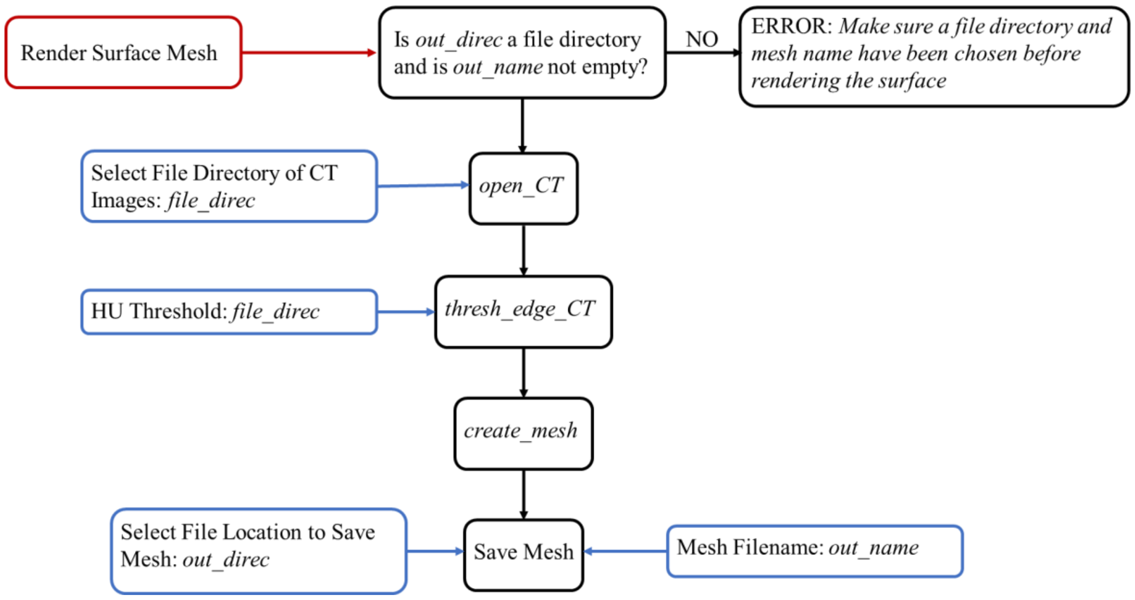

Render Surface Mesh
Is out_direc a file directory and is out_name not empty?
NO
ERROR: Make sure a file directory and mesh name have been chosen before rendering the surface
Select File Directory of CT Images: file_direc
open_CT
HU Threshold: file_direc
thresh_edge_CT
create_mesh
Select File Location to Save Mesh: out_direc
Save Mesh
Mesh Filename: out_name

***Figure 3. 43**: Flow chart depicting the process used to render a surface mesh in Blender. The red square represents a button push, black squares are functions that are executed, and blue squares are inputs to those functions.*

The DICOM images in the file directory, selected using the file browser opened after the 'Select File Directory of CT Images' button was pressed, were opened as described in *open_CT* (Section 3.3.1), the images were saved as a NumPy stack, and rescaled using the RescaleSlope and RescaleIntercept from the DICOM headers. The threshold inputted by the user using the slider described previously was applied to the stack of images, and then a Canny edge detector was applied. Once the images were binarized and the edges were isolated, every point was converted into a 3D point described in the *extract_PT_Cloud* function (Section 3.3.3). The 100 nearest neighbours were found using the KD Tree method described in Section 2.5, since the point cloud was used to build the tree and then was used as the query input, the first nearest neighbour (i.e., the nearest of nearest neighbours) was the point in the point cloud itself so the distance in the first column always had a value zero. This also meant that when 100 points were queried, the number of true nearest neighbours found was 99. Ignoring the first column from the array of distances to the nearest neighbours, the overall average distance between all neighbours was calculated using the *mean()* function contained in NumPy. The 3D point cloud was converted from a NumPy array into the PyVista PolyData type and the average distance between the nearest neighbours was used as the alpha value for the *Delaunay_3d* function in PyVista used to create the mesh. The surface of the mesh was extracted, triangulated, and cleaned through the corresponding functions in PyVista. The surface was saved as an STL file with the corresponding file name defined by the user in the Mesh Filename portion of the panel. The STL file was then opened by calling the Blender Operator *import_mesh.stl* and using the file path and name of the created STL. Within the CreateMeshOperator *execute()* function before the *Create_Mesh* function was called, the *out_direc* and *out_name* properties in the Property Group were checked to ensure that an output directory and output filename had been defined. The *out_direc* property was checked if it was a file directory using the *isdir()* function as described previously. If the *out_direc* was not a file directory and the *out_name* property was an empty string, an error was thrown that informed the user '*Make sure a file directory and mesh name have been chosen before rendering the surface*'; following the error, the Operator was cancelled. If the output directory and filename had been specified correctly the *Create_Mesh* function was called.

3.9.1.5 Segmented Mesh Filename

The Segmented Mesh Filename portion of the panel functioned the same way as the Mesh Filename. The user inputted the desired name of the file without the extension, and this name was used to save the mesh that was segmented after the user drew a region on the surface. The STL file was saved in the location defined by the file directory for the output mesh that was saved in the Property Group.

3.9.1.6 Select ROI

3.9.1.6.1 Selection of Vertices

The final Operator defined in the Blender add-on was the RoiSelectionOperator. The purpose of this operator was to take the region of interest drawn on the outer surface of the mesh by the user and project the selection inward a certain thickness to cut out the inner and outer portions of the desired region of the mesh. The *poll* function in an operator is an optional function that checks if an operator can be run (Blender Foundation 2021b). The *poll* function in this case was used to check if there was an active object. If there was an active object, there was some form of a selection that had been made by the user. If there was no active object, the button would not appear as an option to press. Once the user had selected a region using the selection tool, which was done when the viewer was in 'Edit' mode, and the 'Select Region of Interest' button had been pressed, the operator switched from the 'Edit' mode to the 'Object' mode, which allowed the selected vertices on the STL to be saved as a NumPy array called *surf_pts*. All the vertices contained in the mesh were selected and saved in an array called *full_pts*.

3.9.1.6.2 Estimating Normal Direction

The normal direction of the selected points was determined by finding two vectors, **a** and **b**, that were close to perpendicular. The starting point of the two vectors was the centroid of the selection and the cross product of these points was used to determine the normal direction. The centroid of the points contained in *surf_pts* was determined as the mean in the three directions, x, y, and z using the NumPy function *mean()*. All the possible values of z contained in the selected points, *surf_pts*, were determined using the NumPy function *unique()*, which returns a sorted array containing all of the unique values contained in the input array (The NumPy Community 2021i). The z-values were contained in the third column of *surf_pts*, only the unique values in this column were determined. The difference between every unique z-value and the z-value of the centroid were found. The points within *surf_pts* that had the minimum difference between their z-value and the z-value of the centroid were found using the NumPy *where()* function (The NumPy Community 2021k), these points were called *surf_z1*.

From the selected points that were on the z-level closest to that of the z-value of the centroid, the point with the maximum y-value was determined. If there were multiple points with this y-value the first point found was used, and the vector, **a**, was created from the centroid to the point with the maximum y-value.

If all the points in *surf_pts* were on the same z-level, then the selection of the points was mostly in the x-y plane, the point with the maximum y-value was used to create the vector **a** and the point with the maximum x-value was used to create a vector **b**. The vector from the centroid to the maximum y-value, **a**, was determined as previously described. To determine the point with the maximum x-value and thus create the vector **b**, all the points with the z-value of *surf_z1* and having the maximum x-value were found. A loop was created to compare these points to the selected maximum y-value point, if the point did not have the same x and y coordinates as the maximum y point it was saved as the maximum value for x, otherwise, the loop continued until such a point was found. The point with the determined maximum x was used to create a vector, **b**, which was the vector from the centroid to the maximum x.

If there was more than one z-level with the minimum distance from the z-value of the centroid, it was not immediately clear if the selection was mostly in the x-z or the y-z plane. Initially, the distance between the minimum and maximum values in three directions was used to determine the plane of the selection, where the selection plane was determined as the directions with the largest distance. The final algorithm did not use the distance because the shape could vary depending on the user selection and was not a reliable measure of the plane of selection. To determine the estimated normal of the selection for both these cases, vectors from the centroid to the maximum z and from the centroid to the maximum y were used to determine the normal direction. As it was difficult to determine the plane of selection, the same vector selection process was used whether the selection was mostly in the x-z or the y-z plane. The vector from the centroid to the maximum y-value had previously been determined and was saved as the vector **a**. To determine the vector **b**, the maximum z-value was used, all points with the maximum z-value were found, their x and y coordinates were compared to the coordinates of the maximum y-value that was previously determined. The first point that was found that had the maximum z-value but not the same x and y coordinate as the maximum y point was chosen and used to create the vector from the centroid to this point called **b**.

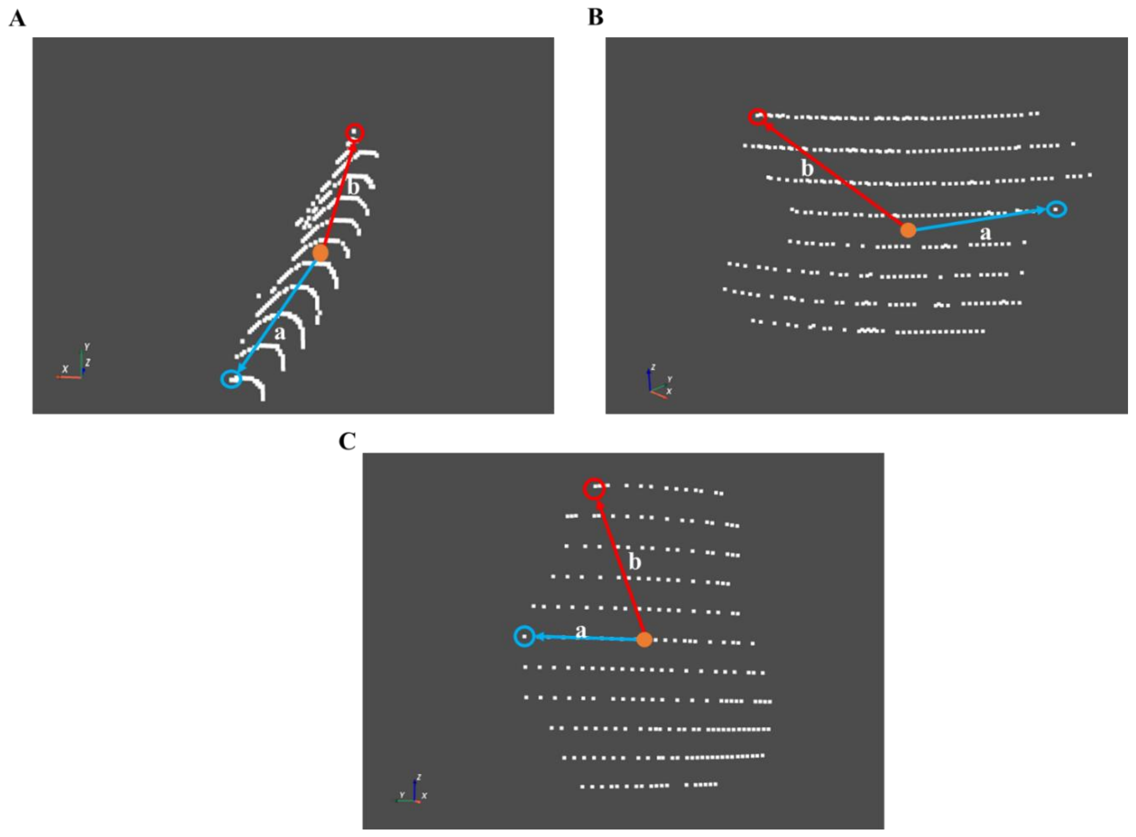


Figure 3. 44: *Depiction of creation of vectors a and b the cross product of which was used to calculate the normal of the selected point when the majority of the selection was in the A. x-y plane, B. x-z plane, and C. y-z plane. The orange point represents the centroid of the selection, the blue circle and arrow the creation of vector a, and the red circle and red arrow the creation of vector b.*

The vectors calculated for examples of the three cases of the plane of selection x-y (A), x-z (B), and y-z (C) are shown in Figure 3.44. Within these images, the orange circle represented the centroid of the selected points, the blue circle represented the maximum y-value, the blue arrow represented the vector **a**, the red circle represented the maximum x-value in image A and the maximum z-value in images B and C, and the red arrow represented the vector **b**. In the image for the x-z plane selection, the point chosen for the maximum y-value (point circled in blue) appeared to be the same as the maximum x-value but, due to the curvature of the object in the y-direction, this point was the maximum y-value along the z-level closest to the centroid.

The cross product between vector **a** and **b**, $\boldsymbol{a} \times \boldsymbol{b}$, was calculated using the NumPy *cross()* function which calculated the cross product between the two vectors and returned a vector that was perpendicular to both **a** and **b** (The NumPy Community 2021a). The purpose of finding the cross-product of these two vectors was to estimate the normal of the selected points by calculating the normal at the centroid. The dot product between the previously calculated cross product and the vector from the origin to the centroid was calculated using the NumPy function *dot()* which returned the inner product of the two vectors without complex conjugation (The NumPy Community 2021c). The dot product was used to check the direction of the cross product as the direction of the cut after the ROI selection was assumed to be inward towards the origin. If the calculated dot product was greater than 0, then the direction of the cross-product pointed in the same direction as the vector from the origin to the centroid which was outward. In this case, the calculated normal was flipped by multiplying all dimensions by -1. If the dot product was less than

0, the normal pointed inward or the opposite direction as the vector to the centroid so no changes were made. The normal vector was transformed into a unit vector by dividing each component by the norm of the vector, this norm was calculated with the use of the NumPy *linalg.norm()* function (The NumPy Community 2021d).

3.9.1.6.3 Point Selection for Inner Surface

After the estimated normal for the selected points was determined, vertices on the outer surface were projected inward the desired thickness of cut along the normal to select points on the inner surface. This was accomplished in two stages. The first stage selected all the points of the mesh that were within a sphere with the radius of the desired thickness from any of the points in *surf_pts*. The second stage aimed to refine the selection from the first stage to help maintain the shape of the initial selection by the user.

The first stage of the point selection was accomplished by comparing every point within *surf_pts* to every point within *full_pts*. The distance was estimated as the sum of the absolute difference between the x, y, and z values of the *surf_pts* and the *full_pts*. The equation to estimate the distance was defined as:

$$dist = |surfpts[i,0] - fullpts[j,0]| + |surfpts[i,1] - fullpts[j,1]| + |surfpts[i,2] - fullpts[j,2]|$$

Where *i* was the row index for the *surf_pts* and *j* was the row index for the *full_pts*. The columns accessed using 0, 1, and 2 represent the x, y, and z values of each point, respectively. For every estimated distance that was less than the inputted thickness, points from *full_pts* were saved into a matrix called *surround_pts* if they were not within the original points selected by the user. The points contained in *surround_pts* only contained the points on the inner surface and did not contain those in *surf_pts* creating a clear delineation between the points selected by the user and those that were determined based on the thickness. The separate matrices of points allowed an easier comparison during the second stage of shape refinement because the list of points was unordered separating the lists from the beginning ensured that only the *surf_pts* were projected. If there was only one z-level contained in *surf_pts*, the thickness was projected only in the z-direction, so *surround_pts* only contained points where the estimated distance was less than the thickness and not on the same z-level as *surf_pts*. If the *surf_pts* contained multiple z-values, the selected point in *surf_pts* was projected along the normal that was calculated earlier by the magnitude of the thickness. The distance between the point in *full_pts* and the projected point was estimated as *dist*. If the distance between the point in *full_pts* and the projected point was less than the estimated distance between the point in *full_pts* and the point contained in *surf_pts* it was added to *surround_pts*. If the point in *full_pts* was closer to the projected point than the point in *surf_pts* it was assumed that the point was not contained in *surf_pts*. Comparing the distance between the point in *full_pts* and that of the projected point and the point in *surf_pts* allowed for the distinction between the points selected for the inner and outer surfaces. Once all of the points in *surf_pts* were compared to *full_pts* and the matrix *surround_pts* was created, the NumPy function *unique()* was used to remove any duplicated points. The steps used in the first stage are demonstrated in Figure 3.45 showing the initial selection of points (A), the points selected within the radius of the thickness from points of A (B), the distances used to determine if the points are on the inner surface, and the points of the inner surface after they have been separated from the outer points (D).

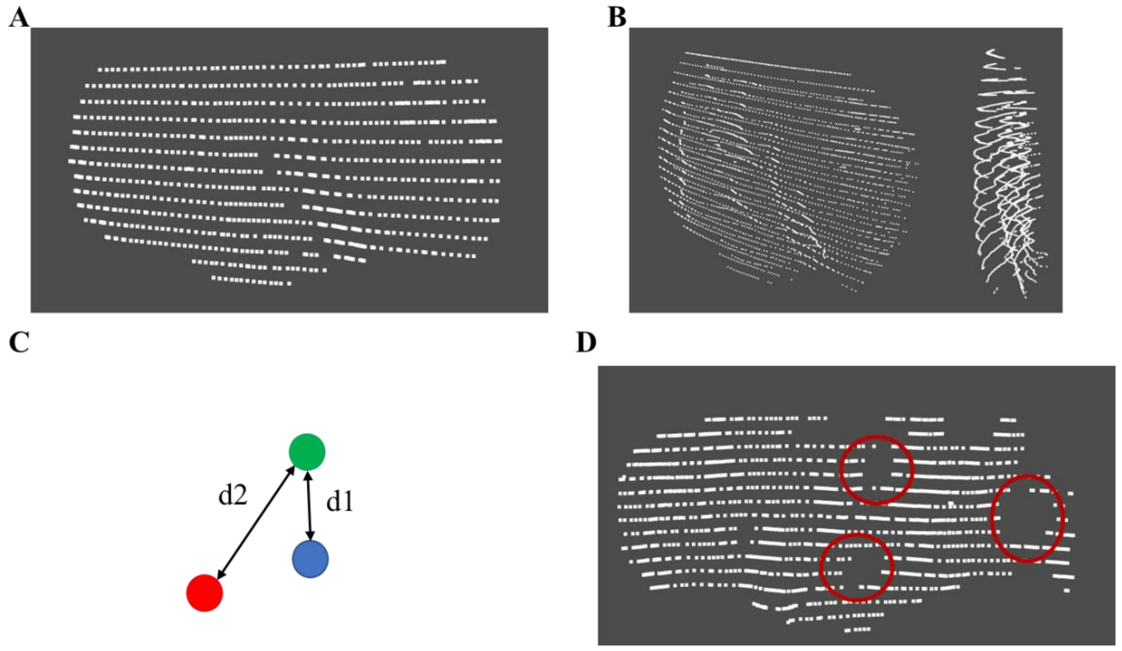


Figure 3. 45: *Depiction of the first stage of selection of inner points. A. Selected outer points. B. Selected inner and outer points based on the thickness. C. Comparison of distances to remove outer surface points, where the red point is the initially selected outer point, the green point represents a point selected from the first stage (B), and the blue point is the outer point (red) projected along the normal, points are kept is d1 < d2. D. Final inner selected points, the red circles indicate areas where holes occurred during the refinement stage of the point selection.*

The first image (A) in Figure 3.45 demonstrated the initial outer surface points selected by the user. All the points contained in the mesh were compared to the points in A, and all points that were within a radius of the thickness from the points in A were kept creating the point set seen in B. In the image shown in C, the red dot represents an outer surface point from A and the green point represents a point from B where it is within the thickness from the red point. The final blue point represents the red point as it was projected along the normal by the distance of the thickness. The distance *d1* is the distance from the point in B (green point) to the projected point (blue point), and the distance *d2* was the distance from the point in A (redpoint) to the point in B (green point). If *d1* < *d2*, the green point is classified as a point on the inner surface and saved to *surround_pts*, the described process was repeated until all points of the mesh were compared to the points in A and the point set in D was created. There are gaps in the point set in D (red circles), this is where *d1* > *d2* and the point was not included as part of *surround_pts*.

The refinement stage compared the points in *surf_pts* to those contained in *surround_pts*. Points from *surround_pts* were removed that deviated from the original shape chosen by the user. The refinement filtered points in two steps. The first step started by finding the maximum z-value contained in *surf_pts*, this value was projected by the value of the thickness in the direction of the z-component of the normal. The projected maximum z-value became the maximum z-value contained within the inner surface. All points contained in *surround_pts* that were greater than the inner z maximum were removed using NumPy *delete()* (The NumPy Community 2021b). The mesh was created from points extracted from a set of CT images so there were certain z-levels contained in the mesh. Taking advantage of this property, the points were further filtered by investigating all the points in *surf_pts* contained in each z-level. When the maximum z-value of *surf_pts* was projected there was no guarantee that the projected value would be one of the possible z-values of the vertices. The z-level in *surf_pts* that was the closest value and less than the maximum inner z-value was found by calculating the difference between the maximum inner z level and the values contained in *surf_pts* and choosing the minimum difference. This value acted as the starting level for the inner points that would be extracted from *surround_pts*, and all z-levels that could be contained in the inner surface were determined based on this value. All of the z-values contained within *full_pts* were determined using the NumPy *unique()* function, which was used to determine all the unique values contained in the column that contained the z-values. If the *surf_pts* contained only one z-level, the z-levels contained in the inner points were found by projecting the z-value by the desired thickness along the normal. This value became the maximum z value. The list of z-values for the inner points was defined as the z-levels in *full_ pts* that were greater or equal to the minimum value of the z-value in *surf_pts* and the projected z value, and less than or equal to the maximum of the z-value in *surf_pts* and the projected z. If there is more than one z-level in *surf_pts*, the z-levels for the inner surface were determined by finding the location in the list of z-values for *full_pts* that contained the z-value in *surf_pts* that was the closest value that was also less than the maximum projected z-value. Taking that z-value as the maximum, the next lower z_n-1 values were chosen as the rest of the z-levels of the inner surface, where z_n is the number of z-values in *surf_pts*. The z-values for the inner surface were referred to as *z_vals_in* and were then used to filter the remaining *surround_pts* such that comparisons were done for every z-level and points that fit certain criteria were selected. The method in which these points were selected varied based on the dominant direction of the normal that was estimated previously. The filtering of these points worked to compare the points on *surf_pts* to that of the inner surface points (*surround_pts*) and keep points from *surround_pts* that corresponded to the shape of *surf_pts*. The comparison of points varied based on the largest component of the normal for *surf_pts*.

Some variable names were used for all directions of the largest component of the normal, those variables included:

> *Surf_z*: For each z-value in *z_vals_in*, this variable contained the points from *surf_pts* that were at the z-level from *z_vals* that corresponded to the inner z-level.
>
> *Sel_z*: For each z-value in *z_vals_in*, this variable contained the points from *surround_pts* that were at the z-value defined by *z_vals_in*.
>
> *New_surf*: This variable was defined as the projection of the *surf_pts* along the normal with a magnitude of the thickness.
>
> *New_sel*: This variable defined the final selected points, it was the final output of filtering *surround_pts*.

When the largest component of the normal was in the x-direction the y-value of the points projected from *surf_pts* was used to select points from *surround_pts* that would be kept. Based on the list of inner z-levels (*z_vals_in*), the outer z-level (*z_vals*) that corresponded to the inner z-level of interest was determined. The outer surface points (*surf_pts*) that were on the outer z-level corresponding to the inner z-level were saved as *surf_z*, the inner surface points (*surround_pts*) that corresponded to the inner z-level of interest were set as *sel_z*. The method to choose points to keep from *sel_z* depended on the number of points contained in *surf_z*. If *surf_z* contained no points at this z-level, no points were selected. If *surf_z* contained one point, this point was projected along the normal by the distance of the thickness and was labelled *new_surf*. Points from *sel_z* were added to the final set of selected points (*new_sel*) if the y-value of the point in *sel_z* was less than the *new_surf* y-value plus the z-spacing (i.e., the difference between values in *z_vals*) and were greater than the *new_surf* y-value minus the z-spacing. When *surf_z* contained 2 points, these points were projected along the normal by the thickness, and points in *sel_z* that were between the y-values of the two projected points were added to new_sel. If *surf_z* contained more than two points, a KD tree for the *surf_z* points was created using the *KDTree* function in the *spatial* package from SciPy as described previously. The KD tree was used to find the two nearest neighbours to every point contained in *surf_z*. The *surf_z* points were then projected by the distance of the thickness in the direction of the normal, to create *new_surf*. The points in *new_surf* were selected to be saved in *new_sel* if their y-values were between the y-value of the projected point and the y-value of at least one of its two nearest neighbours. The nearest neighbours search was performed on the original *surf_z* points and not the projected points, since all the points were projected in the same manner and were on the same z-level, the nearest neighbours in the original would correspond to the same nearest neighbours in the projected points. The process of projecting and selecting points when *surf_z* had one point and more than two points is shown in Figure 3.46.

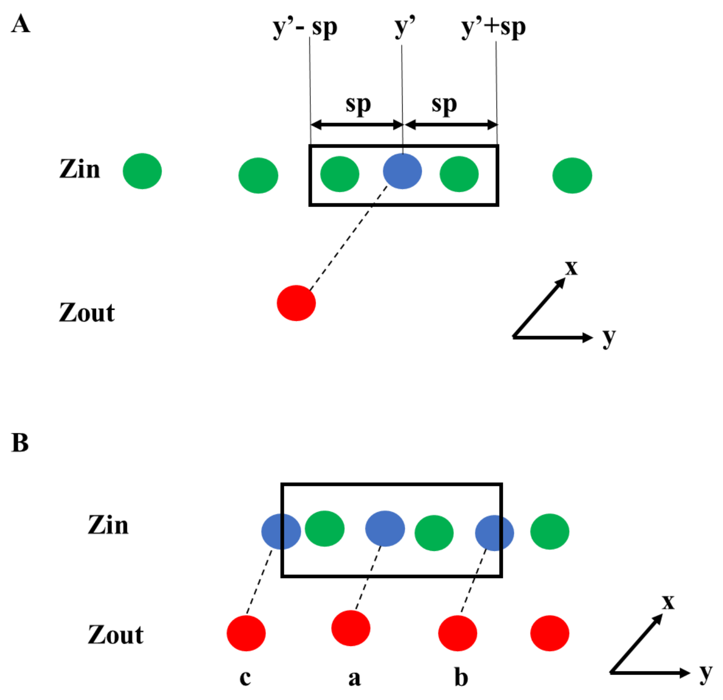


Figure 3. 46: *Refinement stage for selection of inner points when the x-component of the normal was the largest and the outer points at the z-level Zout had A. one point, and B. two or more points. The red points are those on the outer surface, the green points are points on the inner surface that are being refined, and the blue points are projections of the outer surface points (red) along the black dotted line. Blue points are included in the final selection if they were within the black box.*

The process when there was one outer point and when there were two or more points was demonstrated in Figure 3.46. The outer points (on the z-level Zout) were labelled in red, the inner points (on the z-level Zin) were labelled as green, the points that were projections of the outer points were labelled as blue, and the direction that the points were projected was represented by the dotted black lines. Points on the inner surface (green) that were selected to be added to *new_sel* were those contained in the black boxes. When there was one point on the outer surface (A), the point was projected and points in the box that spanned from the y-value of the projected point (y') plus and minus the spacing in the z-direction (sp). When there were more than 2 points (B), the point of interest (labelled a) and its two nearest neighbours (labelled b and c) were projected (blue points), and points with y-values bounded by the y-values of b and c were added to *new_sel*. The process when there were two points was very similar to the nearest neighbour approach, but the selected points were bounded by the projection of the two points.

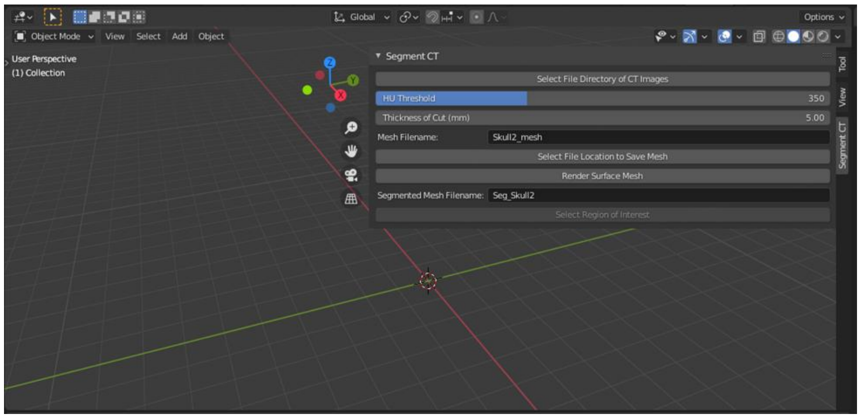
The method to determine the points added to *new_sel* when the largest component of the normal was in the y-direction was similar to that used when the largest component was in the x-direction. The only difference between the two methods was when selecting the points, the x-value of the projected points (*new_surf*) was compared to the inner points (*sel_z*) instead of the y-value. This method looked at the x-z plane such that for every z-level the x-values of the projected points were compared to that of the points originally selected by the user and was used to choose the points from *surround_pts* that remained. A representation of this process would be accomplished by switching the x and y axes in Figure 3.46.

When the largest component of the normal was in the z-direction, all the points contained in *surround_pts* at the desired inner z-value were saved as *sel_z*, and the points on the corresponding outer surface z-level were saved as *surf_z*. The values within *surf_z* were divided based on their x-value by finding all the x-values in *surf_pts* using the NumPy function *unique()*. The x-values were looped through and for each iteration, the set of points in *surf_pts* at that corresponding x-value was saved as *surf_x*. The points in *surf_x* were projected by the thickness along the direction of the unit normal. If *surf_x* contained one point, for that projected point all the values that had y-values that were greater than the y-value of the projected point plus the *z_spacing*, and points that were greater than the projected y-value minus the z-spacing were saved. If there were only two points projected, points from *sel_z* were selected if they were between the y-values of the two projected points. Finally, if *surf_x* contained more than two points, the two nearest neighbours for each value in *surf_x* were found using the KD Tree method explained previously. For each value in *surf_x*, the points from *sel_z* that had y-values that were between that of the current point and its two nearest neighbours were selected and added to *new_sel*.

This method to refine *surround_pts* did not guarantee that points were not repeated or selected more than once. The NumPy function *unique()* was applied to the final output of *new_sel* to remove repeated points. The *new_sel* points were then added to the set of points on the exterior surface that were initially selected by the user. The combination of the selected points from *surround_pts* and those initially selected by the user were used to create a new mesh accomplished by following the same steps as the *Create_Mesh* function described in Section 3.9.1.4. The mesh that was created was saved as an STL in the output file directory specified by the user through the 'Select File Location to Save Mesh' button. The filename used was the name chosen by the user that was inputted into the 'Segmented Mesh Filename' portion of the panel.

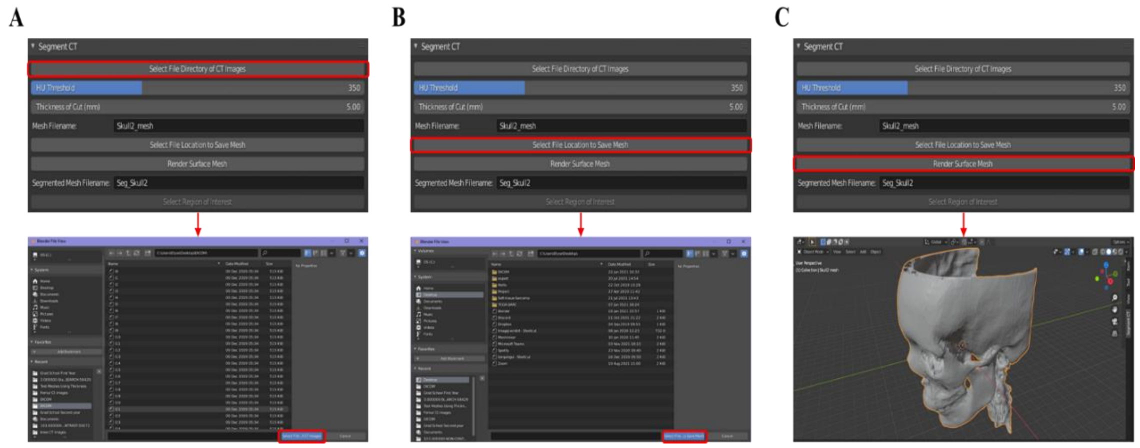
3.9.2 Demonstration of Process in Blender

The following section provides images showing the steps the user followed to create a mesh and then perform a selection from that mesh. Figure 3.47 shows the interface in Blender before any mesh is created from the CT images. Figure 3.48 demonstrates the steps taken by the user to open a set of CT images and create a mesh of the bone given a user inputted threshold value. The images in A show the pop-up dialog box that appears when the button (highlighted by the red square in the top images) 'Select File Directory of CT Images' is pressed. Within the dialogue box (the bottom image of A) the user selected a folder or file directory that contains all of the DICOM images for the CT scan and the button at the bottom of the box is pressed. The user does not need to select a file itself but ensure the browser is in a file directory or folder. The images in B show a similar progression as A but the process selects the file directory in which the created mesh and the segmented ROI mesh are saved as STL files. The bottom figure in C demonstrates the output mesh for the imported set of CT images when the 'Render Surface Mesh' button was pressed, the threshold used was set to 350 HU.



HU Threshold 350
Thickness of Cut (mm) 5.00
Mesh Filename: Skull2_mesh
Segmented Mesh Filename: Seg_Skull2

Figure 3. 47: *Blender interface before the user has opened a set of CT images and created the mesh object.*


A
B
C
Segment CT
Select File Directory of CT Images
Select File Location to Save Mesh
Render Surface Mesh
Select Region of Interest

HU Threshold 350
Thickness of Cut (mm) 5.00
Mesh Filename: Skull2_mesh
Segmented Mesh Filename: Seg_Skull2

Figure 3. 48: *Demonstration of Blender interface, A. pop-up dialog when the button 'Select File Directory of CT Images' is pressed, B. pop-up dialog when the button 'Select File Location to Save Mesh' is pressed, and C. output after 'Render Surface Mesh' is pressed. The red boxes highlight the buttons that are pressed by the user, the bottom image is the result of pressing the button in the top image.*

Examples of the output of the 'Select Region of Interest' button in the Blender interface are demonstrated in Figure 3.49. This was applied to the mesh shown in Figure 3.48. Within both A and B, the image on the left shows the user interface when the region is selected by the user. The selected surface points are shown in orange and the unselected points are black. The modes that the user must be in to select the ROI are demonstrated in A with the two red rectangles labeled 1, and 2. To select the ROI, the user must be in 'Edit Mode', this is changed in Blender using the dropdown menu highlighted by 1. To draw the ROI, the user can use the selection tool highlighted by 2. This tool has different options for selection that include a box, circle, and lasso selection. The lasso selection was used in Figure 3.49 and provided the ability to draw any desired shape. The right images demonstrate the outputted STL files that were generated based on the user selection from a front and a side view. The thickness of the cut for A was set to 5.0 mm, and 10.0 mm for B. The selections were not the same, but they were made in similar regions of the skull.

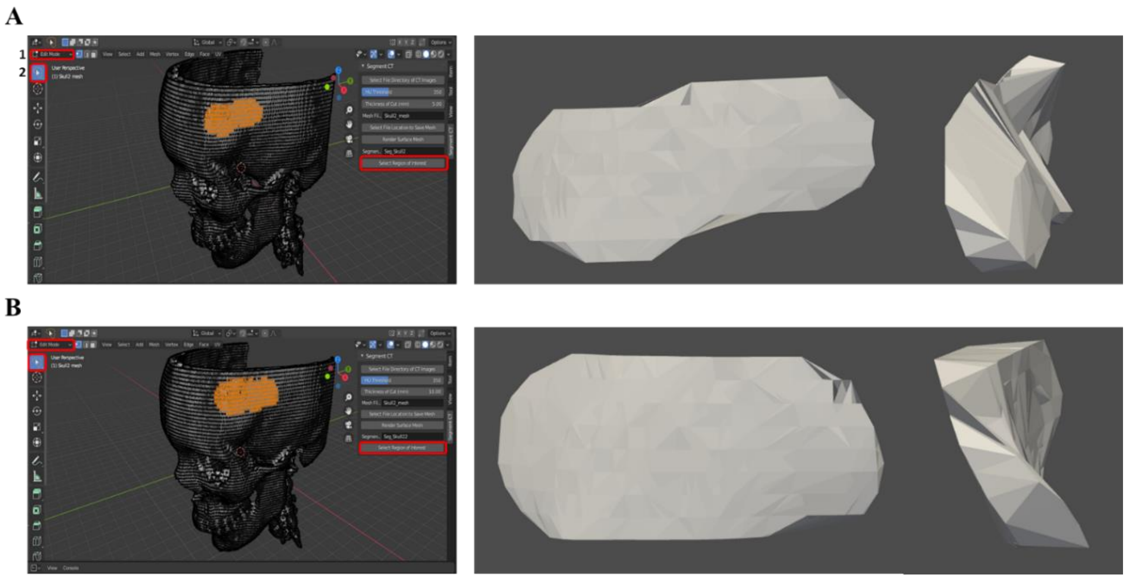


Figure 3. 49: *Examples of ROI selection made by the user (left) and the resulting STL output (right) after the button 'Select Region of Interest' is pressed using a thickness of the cut of A. 5.0 mm and B. 10.0 mm. The buttons pressed by the user are highlighted in red, where 1 and 2 are the modes the user must be in to select an ROI. The orange vertices in the left images are the points selected by the user. The mesh in B had a more continuous rear surface than A, this highlights the importance of the thickness of cut.*

The STL in Figure 3.49 that used the thickness of 5.0 mm (A) appeared to have a surface on the back that was not continuous and had an uneven thickness, this was evident in the side-view image. The STL in Figure 3.49 with a thickness of 10.0 mm (B) had a more continuous interior surface of the segmented portion of the skull. The differences in the appearances of the STL files A and B demonstrate the importance of the thickness of the cut. If the skull portion in A is thicker than 5.0 mm, the full interior surface of the skull will not be captured, and it will be distorted. If the thickness of the cut is much larger than the thickness of the skull, the projected points will encompass the inner structures of the skull and distort the mesh such that it does not properly represent the bone.

The selection of the region of interest was applied to the frontal bone of the skull, where the thickness of the cut was set to 10.0 mm. The selected region and the resulting STL mesh files are shown in Figure 3.50. The original shape drawn by the user was maintained and the curvature of the inner portion of the bone was maintained once the region was selected and re-meshed.

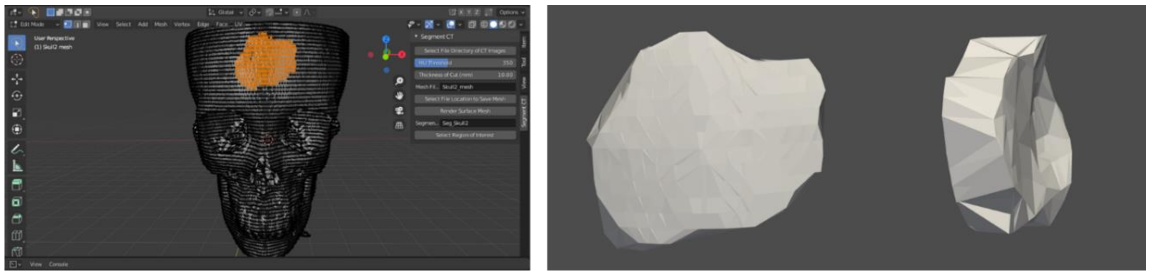

Segment CT
Select File Directory of CT Images
Select File Location to Save Mesh
Render Surface Mesh
Select Region of Interest
HU Threshold 350
Thickness of Cut (mm) 10.00
Mesh Fil... Skull2_mesh
Segmen... Seg_Skull2

Figure 3. 50: *Selection of portion of the frontal bone and resulting STL file created from selection (frontal and side view), created with a thickness of the cut of 10.0 mm. The shape of the selection and the shape of the inner and outer surfaces of the bone were maintained.*

To further test this, the lower limb dataset (described in 3.2) was imported into Blender. The full set of images was used to create a model of the lower limbs (Figure 3.51). The threshold used was again 600 HU, and the thickness of the cut was 10.0 mm. Two selections in Blender were made, one on the outer surface of the left femur, and one on the outer surface of the left tibia.

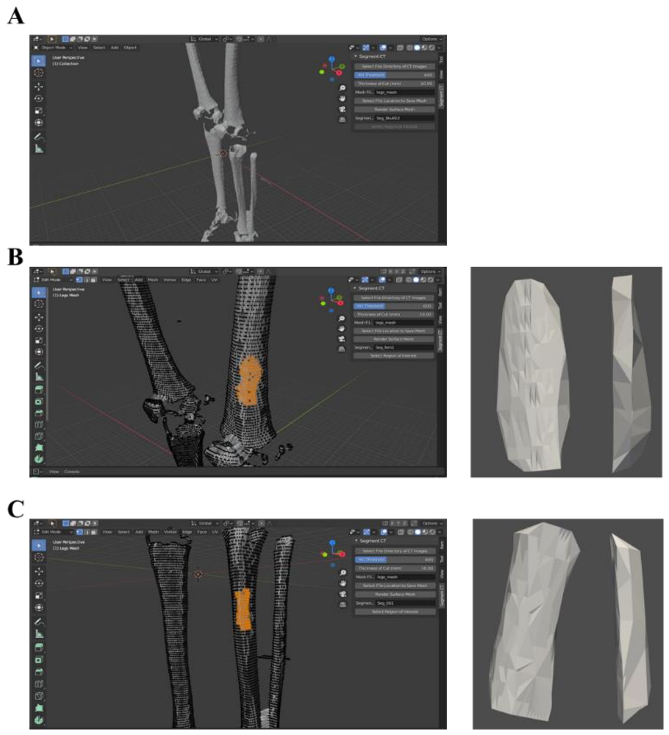


Figure 3. 51: *Blender interface for a set of CT images of the lower limbs, A. mesh created by ‘Render Surface Mesh’, B. user selection and resulting STL (frontal and side view) for femur selection, and C. user selection and resulting STL (frontal and side view) for tibia selection. Both B. and C. used a thickness of the cut of 10.0 mm. Smoothing of the shape defined by the user occurred and the curvature of the inner and outer surface appeared to be maintained.*

Applying the process to the femur and the tibia (Figure 3.51) was done to test if this process could be applied to bones other than the skull, such as long bones. It demonstrated the applicability of the procedure in other applications. The selections performed on the skull were done mostly on the cranial vault as this area was less complex with fewer curvature changes than other areas of the skull. In addition, the areas around the cranial vault had the least amount of difference between the sampled and full mesh when the point cloud was downsampled, indicating that the error in these areas was lower than in other areas of the skull. The curvature of the long bones is different than the skull, and the radius of the long bones is less than the radius of the cranial vault. The smaller radius of curvature provided an opportunity for the selected region of interest to span more of the circumference of the bone, which could impact the resulting STL files. The distance between the two sides of the cortical bone of the long bones is smaller than the distance between that of the skull. For example, if a portion of the left side of the skull is selected the thickness of the cut required to create a mesh containing the right side of the skull would be larger than that required to span the right side of the femur if the left side of the femur was selected. The results of the ROI selection on the femur and the tibia are shown in Figure 3.51. The user-defined shape remains mostly the same but some of the edges of the selection appear to have been smoothed, and the curvature on the outer and internal surfaces appeared to be maintained.

3.9.3 Processing Time in Blender

The time to create the full mesh of the skull and the time to create the segmented mesh after the user selection was measured in the same manner as described previously. Figure 3.52 shows the original skull mesh in Blender, the region selected by the user, and the resulting STL file of the segmented portion.

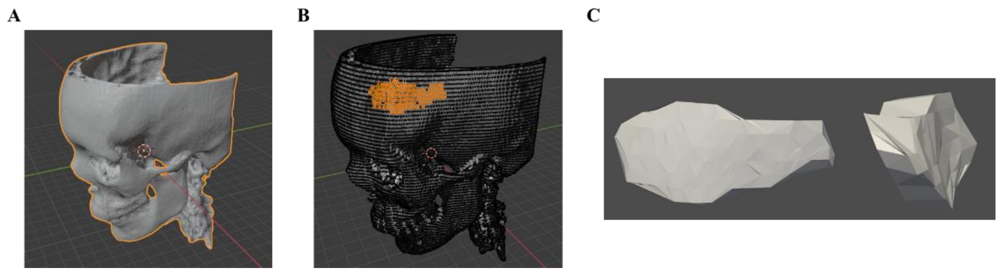


Figure 3. 52: *Mesh used to measure the execution time in Blender, A. full mesh, B. user selection of ROI, and C. resulting STL of the selected region.*

The creation of the mesh in Blender involved opening the CT images, applying the threshold, applying the edge detection, extracting the point cloud, and creating the mesh from the point cloud. The execution time for these sets of processes was 1136 seconds or 18.9 minutes, and the time to create the mesh after the ROI points were selected was 58.4 seconds.

Both sets of time measurements, in IDLE and Blender, were conducted on the same remote desktop with an Intel Core i7-10700 2.90 GHz 64-bit processor with 64.0 GB installed RAM.

Chapter 4: Discussion and Conclusion

4.1 Implications of Qualitative Data

The methodology created throughout this project was able to create a mesh object that could be visualized and manipulated by the user. With the process applied to the skull, the rendering was able to capture the different portions of the skull including the frontal bone, nasal bone, and mandible. Some areas were not as well visualized as others, where it was more difficult to visually distinguish the shape of the bone. These areas included around the orbit of the eye, around sinuses, the inferior portion of the interior of the skull (seen from superior view), the vertebrae, and the top portion of the cranial vault where the bone appears to be cut off in the provided sample (Skull2).

The Blender interface itself required a few steps and inputs to be done by the user. The buttons were written so that their use would be intuitive and not require large amounts of technical expertise to understand their function. One of the benefits of using Blender to create this interface is the user can hover over a button or tool and a description of what the tool does will be displayed. This ultimately makes the interface friendly for many types of users. For example, if the user hovers their mouse over the button 'Render Surface Mesh' the description 'This Operation will apply the thresholding, edge detection and extract a 3D point cloud from the imported stack of CT images which is then used to render a mesh' is displayed.

Although the interface was designed to be intuitive, if the user wishes to use this approach, they still need a fundamental knowledge of the type of bone they wish to segment. For the methodology to function properly, the user must know what HU value should be used as a threshold for that bone. When the method was applied to the CT images of the head and the images of the lower body, different threshold values were used. To determine the value that was required, research had to be done to ensure that the mesh represented the correct bony object. In addition to the threshold value, the user must also have a general idea of the thickness of the bone from which they wish to take the ROI to create a new 3D object. For example, if the user wished to segment a portion of the frontal bone of the skull, knowing the approximate thickness of the bone would ensure that the segmentation contains both the inner and outer surface.

4.2 Implications of Quantitative Data

4.2.1 Sampling and Distance Measurements

4.2.1.1 Pseudo-Uniform Sampling

A pseudo-uniform sampling algorithm was developed for this project to down-sample the point cloud data set to improve processing speed for subsequent functions. This function appeared to be more repeatable and create a more uniform sampling of the point cloud than a random sample of the whole point cloud. The overall average distance of the average at each midpoint during the mesh comparison algorithm resulted in the largest distance occurring at 10% sampled, and the smallest at 90% sampled. The 90% sampled contains more of the original point cloud so the distance was lower. As the percentage sampled decreased, the distance between the meshes increased. The largest value at 10% of 0.340 mm was smaller than the in-plane resolution of the CT images of 0.352 mm. This suggested that the average distance between the meshes was less than the width of a pixel. This indicated that the pseudo uniform sampling was sufficient in maintaining the shape even at the lowest sampling percentage of 10%. Abdullah et al. (Abdullah et al. 2006) studied the dimensional accuracy of models of skulls created through rapid prototyping. Their models were considered to be acceptable clinically as the variation of 0.5 mm or 0.5% was considered negligible. Other studies have suggested different levels of accuracy for different types of surgeries. Gelaude, Sloten and Lauwers (Gelaude, Sloten, and Lauwers 2008) assessed the 3D accuracy of meshes of the outer surface of the femur. They noted that for surgical planning and implant design for bone, the mean 3D deviations should be within the range of 1 mm. Studies involving orthognathic surgery suggested that the clinically relevant error range is 0.5 mm (Baan et al. 2016; Shaheen et al. 2017). In the current study, the largest mean distance between the full mesh and the sampled mesh was below the previously mentioned error margins suggesting that the sampled meshes sufficiently represent the shape for surgical applications.

The time to sample the original point cloud and create a mesh was longer than the time taken to go directly from the original full point cloud to a mesh object for the three percentages sampled of 10, 50, and 90%. Timings were determined on a remote desktop with an Intel Core i7-10700 2.90 GHz 64-bit processor with 64.0 GB installed RAM. The time to create the mesh from the original full point cloud was 20.3 minutes, while the total time to sample the point cloud and create the mesh was 174.7, 160.6, and 157.6 minutes for the 90%, 50%, and 10% sampled point clouds, respectively. The time to create the mesh from the already sampled point cloud was 14.8 minutes for the 90% sampled, 5.2 minutes for the 50% sampled, and 0.5 minutes for the 10% sampled. It was clear that the size of the point cloud reduced the time required to generate the mesh from the point cloud. The most time-consuming portion of sampling and creating the mesh was the sampling, it accounted for approximately 91.5%, 96.8%, and 99.7% of the total time for the 90%, 50%, and 10% sampled, respectively. The purpose of the sampling was to reduce the time needed to process the point cloud but, in this case, it did the opposite of what was hoped. The difference between the geometry of the mesh that was represented by the distance between the midpoints of the sampled mesh to the full mesh was small demonstrating that the difference in the geometry is not large. Although sampling maintained the geometry, for it to fulfill its purpose the algorithm must be improved to reduce the computational time required.

4.2.1.2 Targeted Sampling

When the targeted sampling was compared to the distance values for the pseudo-uniform sampling with similar percentage values, the largest absolute difference between the two was between the overall average and had a value of 0.055 mm. The maximum difference between the overall average of the midpoints had a similar value of 0.053 mm. The difference between these values suggests that the two sampling methods were comparable at similar percentage values. Although the distances were comparable to the pseudo-uniform sampling for the skull, when the two methods were applied to the simplified shape, at low percentages of sampling, particularly at 2.05% and 2.25%, the targeted sampling was more inconsistent and resulted in more shape distortion than the pseudo-uniform sampling method. In terms of the parameters for the targeted sampling, the linear model that was fit to the average of the average at each midpoint revealed that the grid size was significant at a significance level of 0.1, and the percentage sampled had a significant effect with a significance level of 0.05. The overall percentage sampled linear model demonstrated that the grid size and the percentage sampled for each angular similarity range significantly affected the percentage sampled with a significance level of 0.

The targeted sampling appeared to be successful on the skull but appeared to be flawed when applied to the simpler structure, this suggested that there was something fundamentally wrong with the algorithm. It was difficult to determine if the ability of the targeted sampling to maintain the shape of the skull was due to the algorithm or if the point cloud was dense enough so that despite the percentage sampled the shape was maintained. Additionally, the downfall of the targeted sampling was that determination of the ideal parameters would have to be done for each point cloud. With the current algorithm, it was difficult for the user to specify the desired percentage while the pseudo-uniform algorithm has the advantage of having a user-specified percentage value without specifying multiple parameters. The angular similarity intervals did not have a significant effect on the linear models for the average distance of midpoint distances between the models or on the linear model for the overall percentage sampled. For each combination of parameters used for the targeted sampling experiment, the grids that were within the highest angular similarity range was more than 50% for all of the experiments. When the grid size was 10, the percentage of grids in the highest similarity range was over 96%. When the grid size was 30, the percentage in the highest range was closer to 50%. With a grid size of 20, the percentage in the highest range was approximately 65%. The combination of the insignificance of the angular similarity ranges on both linear models and the large distribution of the grids being in high similarity ranges, suggested that either the calculation of the angular similarity must be modified, or another method of measuring similarity should be used.

4.2.1.3 Choice of Sampling Technique

Based on the discussion above, a combination of the pseudo-uniform sampling and the targeted sampling with a modified angular similarity calculation would be implemented in the future to improve the resulting shape of the sampled point cloud and decrease the execution time.

4.2.2 Time to Create Models

The process of applying the threshold, edge detection, extracting the point cloud, and creating the mesh was applied to a set of CT images of a skull as well as a portion of a set of CT images containing the lower limb. The overall time for these steps was determined in Python IDLE 3.8 on a remote desktop with an Intel Core i7-10700 2.90 GHz 64-bit processor with 64.0 GB installed RAM and was 21.3 minutes for the skull and 20.6 seconds for the cropped images of the lower limb. A study comparing freeware software for creating a mesh of bones from CT images (Matsiushevich et al. 2019) compared three free software and one standard software that was available at their institution. The free software included 3DSlicer, InVesalius (https://invesalius.github.io/), and Itk-SNAP(PICSL and SCI); the standard software was VuePACS3D. The processor of the machine used for the free software was an AMD Dual Core 1.65 GHz processor with 8 GB RAM, while the machine for the standard software has a 64 bit Intel Xeon CPU

2GHz with 8GB RAM (Matsiushevich et al. 2019). Five anatomical regions were chosen for each of the three subjects so 15 models were created. The mean time to create the mesh over these 5 regions and 3 subjects for 3DSlicer was 15-20 minutes, 10-15 minutes for InVesalius, 35-50 minutes for Itk-SNAP, and 1-2 minutes for VuePACS3D (Matsiushevich et al. 2019). The measurements of the timing in the current project were conducted on a machine with an Intel Core i7-10700 2.90 GHz 64-bit processor with 64 GB installed RAM. The time to create the mesh model through this methodology in the Python shell IDLE for the full skull, Skull2, was 21.3 minutes and in Blender, the process took 18.9 minutes. In comparison to the study of the 4 techniques mentioned previously, the developed methodology is comparable in processing time to that of 3DSlicer, it was slightly longer than that of InVesalius and shorter than that of Itk-SNAP. Although this is a rough comparison, it demonstrates the ability of the technique to create a mesh in a reasonable time.

4.3 Limitations

4.3.1 Basis for Comparison/Error Analysis

One of the limitations of this project was that the generated 3D model was not compared to a ground truth. Eijnatten et al. (van Eijnatten et al. 2018) reviewed a total of 32 publications that dealt with the accuracy of bone segmentation. All of the reviewed studies used a ground truth model to determine the accuracy. The ground truth models consisted of manual segmentation by an expert, linear measurements using a coordinate machine, or a laser scanner (van Eijnatten et al. 2018). Comparisons can also be done by measuring the distances between landmarks on the physical object and the created model;

the choice of landmarks and measurements can introduce error (Lalone et al. 2015). Mechanical contact scanners have been used to create reference models to compare segmentation techniques for long bones (Rathnayaka et al. 2011) and to compare 3D models of long bones generated from CT and MRI images (Rathnayaka et al. 2012). Other methods used for ground truth models for assessing accuracy that have been used include optical scanners (Gelaude, Sloten, and Lauwers 2008; Szymor, Kozakiewicz, and Olszewski 2016), a 3D scanner using structured light (Stephen et al. 2020), and a laser scanner has been used to determine the effect of slice spacing on 3D models from CT images (Schmutz, Wullschleger, and Schuetz 2007). The project would have benefitted from validating the model against a ground truth or reference model.

The only comparison method performed was comparing the mesh created from point clouds subjected to different sampling methods to the mesh created from the full point cloud. This comparison helped validate the use of sampling and compare the methods, however, it provided little insight into how well any of the models represented the physical object. The project was developed using a set of CT images of the head such that the skull bone was segmented. This contributed to the lack of a reference model as the CT images used were taken in vivo, so although their use helped develop the methodology so that bone was isolated from surrounding tissues, there was no physical skull to make measurements on. For example, when Lalone et al. (Lalone et al. 2015) assessed the accuracy of bone models from CT images, a set of intact cadaveric elbow joints were used. After CT imaging was performed the joints were disarticulated and the soft tissue was removed, an optical tracking system was then used to digitize the surface of the bone.

The digitization resulted in a point cloud that acted as the ground truth. A point-distance measurement between the 3D models of the bone and the digitized surface was then measured to determine the accuracy of the models (Lalone et al. 2015). In the previous example, the use of a cadaveric specimen allowed the authors to directly measure the bone once the tissue was removed. For the current project, the use of the anonymized skull data from a live person meant that direct measurements on the bone of the skull were not able to be made. Ultimately the lack of a full error analysis limited the ability to validate the model.

4.3.2 Mesh Quality

The meshes created in Chapter 3 were created using PyVista (Sullivan and Kaszynski 2019). Within these meshes holes were visible, these errors were more visible after the meshes were exported as STL files and opened using Meshmixer (Autodesk Inc. 2020). When the STL file of the skull was imported into Meshmixer, many of the edges were highlighted in red. According to the forum (Why does my file display this red stuff? n.d.), the red indicates some areas are non-manifold; these could be edges or vertices. When the Inspector tool was opened in Meshmixer, it displayed issues with the mesh. Issues coloured blue are minor, red are large defects, and pink indicates areas of the mesh that are separated (Carolo n.d.). After being imported into Meshmixer, the mesh is shown for the full skull STL file and a portion that was segmented and created using the Blender interface. These are shown in Figure 4.1, where the image on the left is the file as soon as it is imported into Meshmixer. The middle image shows the display when the Inspector tool is used, and the right image is after the defects are automatically repaired using the

Inspector tool. Both the full skull and the segmented portion contain edges and vertices that are red indicating they both contain non-manifold elements. Both meshes contained separated components, which indicated the meshes were not fully connected (represented by the pink colour in the middle image). The images on the right show the resultant meshes after all the defects were attempted to be automatically repaired using the Inspector tool. The repaired skull composed of small elements that did not appear to be connected and no longer resembled a skull. The repaired version of the segmented portion of the skull appeared to have little change; it still had edges and vertices identified in red for nonmanifold geometries. Considering the automatic repair did not remove all the defects, the meshes would need further and more advanced repair techniques. The models with the present defects would likely not be able to be directly 3D printed, this limits the methodology as it could need extensive repair before printing can be done.

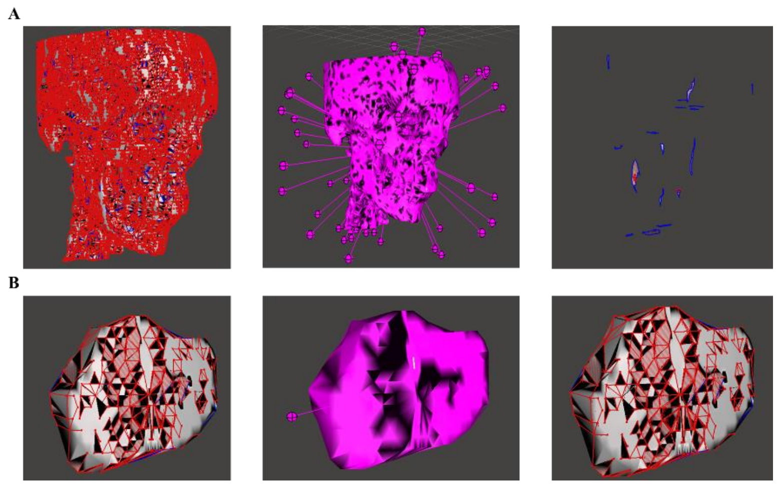


Figure 4. 1: *STL files imported into Meshmixer (left), display for Inspector tool (middle), and mesh after auto repair all was used in the Inspector tool (right) for A. full skull and B. portion of skull segmented using Blender. Errors shown in blue, red, and pink were present, and doing an automatic repair of the mesh (right) was unsuccessful at removing errors.*

4.4 Areas of Improvement and Future Work

4.4.1 Sampling Improvement

As discussed previously, the sampling algorithms developed did not work as desired. The pseudo-uniform sampling generated a sample that was able to maintain the shape. However, due to the execution time of the sampling algorithm (155-159 minutes on a remote desktop with an Intel Core i7-10700 2.90 GHz 64-bit processor with 64.0 GB installed RAM)), the overall execution time to create the 3D mesh model was not improved as hoped. The algorithm for pseudo-uniform sampling iteratively computed the nearest neighbours after each set of 10 points was deleted. The number of times the nearest neighbours are computed must be reduced to reduce the processing time. The pseudo uniform sampling should also be improved so that percentages below 10% can be easily sampled. As mentioned, the current algorithm looks at the 10 nearest neighbours and creates a list of indices based on the desired percentage. When this percentage was less than 10%, the number of points to be sampled was less than 1 so none are sampled. To accomplish the 5% and the 2.5% sampling, the lower percentages were achieved by applying the pseudo-uniform sampling algorithm again to the already 10% sampled point cloud. Running the algorithm multiple times to achieve the desired percentage is cumbersome and would be confusing for the user.

Although not implemented, these improvements may be accomplished through a couple of methods. The first method is to organize the point cloud such that all points along one surface are grouped for each slice and ordered based on their position. An example of an implementation of this could be used to separate the inner and outer surface of the cranial vault with the points being ordered based on their position starting from the top left corner of the images and following a clockwise direction along the surface. If the points are organized in this manner, a uniform sample across each surface can be determined by uniformly indexing the list of points. The uniform sample from an individual list for each surface would help reduce the processing time and most surfaces would be composed of more than 10 points which would also allow sampling below 10% to be applied. The other method would be to separate the points into grids much like was done for the targeted sampling algorithm and apply the same pseudo-uniform sampling algorithm. The reason this may be suggested is that the method of targeted sampling took less time to run than the pseudo-uniform sampling even though it called the pseudo-uniform function to perform the sampling. For example, the time to perform the pseudo-uniform sampling to 50% was approximately 155 min, but when using the parameters for the targeted sampling used in the 7th experiment to get a percentage sampled of 49.84% the execution time was 3.42 minutes (determined on a remote desktop with an Intel Core i7-10700 2.90 GHz 64-bit processor with 64.0 GB installed RAM). The time to apply the targeted sampling was much less than that of the pseudo-uniform sampling. The division of the point cloud into the grids and the calculation of the angular similarity were the main differences between the two algorithms, likely adding the grid division will reduce processing time, and containing more than 10 points would allow lower sampling percentages.

Future work could also be applied to improve the use of the targeted sampling algorithm. The current targeted sampling algorithm contains three parameters including the grid size, the angular similarity ranges, and the percentage sampled for each angular similarity range. The grid size can be modified in the x, y, and z directions. The angular similarities within each grid were mostly in the highest range defined, and the similarity ranges did not have a significant effect on the linear models for the percentages sampled or the distances between the meshes. The angular similarity was chosen to classify the curvature in areas of the point cloud but, with little effect on the percentage sampled and the comparison of meshes, improvements in the calculation itself or another measure of curvature could be used in the future. Open3D(Zhou, Park, and Koltun 2018) was used to estimate the normals of each point in the point cloud since the original creation of the point cloud did not contain normals. The use of an external function to calculate the normals likely contributed to issues with the calculation of the angular similarity because it made it difficult to understand errors in the calculations as I did not personally write the code. Angular similarity as a metric for the similarity of a point cloud was based on its use in (Alexiou and Ebrahimi 2018) where it was used as a metric to assess the quality of point clouds. Improvements in the calculation of the angular similarity through different methods of calculating point normals would be done to improve the results of the targeted sampling. Other metrics to compare point cloud curvature would be explored to create a sampling algorithm that samples based on curvature. An example of a robust method to estimate the normal and curvature of a point cloud obtained using a laser scanner was described in (Nurunnabi, West, and Belton 2015) where outliers were removed, and then the normal and curvature was based on the eigenvalues and eigenvectors of the Principal

Component Analysis (PCA) used to fit the plane. The final targeted sampling algorithm should also be developed such that the user can choose the desired percentage sampled.

4.4.2 Improvement of Mesh Quality

The errors in the mesh were discussed previously. The meshing technique should be refined and improved to help reduce the distortions in the mesh and thus reduce the reparation and processing time before the STL files can be 3D printed. One portion of the meshing method that could be improved to help reduce errors in the mesh would be to refine the choice of α used for the Delaunay tetrahedralization. The calculation of α was based on (Gardiner, Behnsen, and Brassey 2018) where α was determined based on the average distance between the 100 nearest neighbours and then scaled by a coefficient. The scaling coefficient used in (Gardiner, Behnsen, and Brassey 2018) was determined by choosing the coefficient that achieved an alpha value that was closest to the raw volume. In the defined methodology, just the nearest neighbour definition was used, this may have resulted in the α values for the meshing being too small creating holes in the mesh. The volume comparison was not made throughout this project because the calculation of the volume of the mesh directly proved difficult, as described in Section 3.7.2, where different methods achieved different volume values, and they were not as expected based on theoretical calculations. Future iterations can include a process to determine the ideal value of α based on a volume or error analysis. Other techniques including the Marching Cubes algorithm can be explored to determine if they can improve the quality of the mesh.

The edge detector that was applied before the extraction of the point cloud had σ as an input, this affects the Gaussian smoothing filter applied during the edge detection. A comprehensive review of the ideal parameter for bone segmentation was not done and the default value for this filter was used. In the future, an investigation into the value that produces the best representation of the bone after edge detection is applied to the binarized CT images should be done. Improving the detector's ability to capture the edges of the bone will improve the model such that it better matches the geometry of the bone. This could in turn improve the overall quality of the mesh.

4.4.3 Thresholding of CT Images

Improvements to the mesh quality and the improvement of the ability of the mesh to represent the bone could be accomplished through different thresholding and segmentation techniques. Global thresholding was used in this project as it was the most common method used in the review conducted by Eijnatten et al. (van Eijnatten et al. 2018). Although the most common, this technique may not entirely capture areas of structures where the structure may have different pixel intensities across it (Rathnayaka et al. 2011). Further investigation into other thresholding techniques would help inform which method would produce a point cloud that could create the mesh that most closely resembles the physical object. Some examples of thresholding techniques that could be applied that have been used in literature include multi-level thresholding where different thresholds are used for different ROIs, 3D adaptive thresholding that updates the threshold after global thresholding is applied (van Eijnatten et al. 2018), or a method similar to what was introduced by Rathnayaka et al. (Rathnayaka et al. 2011) where they introduced the selection of the threshold based on a Canny filter. Segmentation methods other than thresholding could also be investigated including region growing and statistical shape models (van Eijnatten et al. 2018).

4.4.4 Additions/Improvements to User Interface

There are some additions to the Blender interface that could be added to help improve the functionality and improve the ease of use. These are options that at the beginning of the project were hoped to be added but, unfortunately, have not been implemented. To aid the user in selecting the desired threshold for the global thresholding technique, a histogram displaying the distribution of the pixel intensities would be shown. ImageJ (Schindelin et al. 2012) also uses a technique like this when adjusting the threshold in a set of images. The histogram would allow the user to visualize the intensity of the images and choose the threshold based on this distribution. Visualizing the distribution informs the user on the images themselves, which can differ as CT numbers vary based on the scanner used and the even parameters such as the kilovoltage can change the CT number (Levi et al. 1982). A study on a rabbit femur (Giambini et al. 2015) found that throughout the bone the HU was affected by the reconstruction kernel and the voltage used, with the cortical bone being more affected by the voltage and the cancellous bone being more affected by the kernel. Since variations in the HU which is used as a global threshold can occur, a histogram would help determine the value for that particular set of CT images.

As mentioned previously, the default σ for the edge detector was used, this value controls the smoothing which reduces the noise in the image. The addition of a button for this value in the Blender interface would allow the user to control how much their images are smoothed when the edges are found. This would give the user more control of the outputted model.

Once the sampling algorithms are refined and improved, these can be added to the Blender interface. At the time that the add-on was created, the testing of the sampling algorithms was ongoing, these tests revealed that the sampling did not perform as was hoped and had to be refined. The addition of sampling would allow a user to select a percentage to sample the point cloud and after improvements to the sampling decrease the time to render the mesh in Blender.

Another addition that would be added to the Blender interface would include the ability to mirror the desired selection onto the other side of the structure. The ability to mirror the selection would allow the user to select an ROI around a defective area of the bone. This would then be mirrored to select the unaffected side of the structure. The selection for the healthy or unaffected side could then act as a model for how to improve the defected side of the bone. A case study on maxillofacial reconstruction used a similar technique where the healthy side was mirrored onto the affected side and used as a template to create a patient-specific implant (Scolozzi 2012). This would extend the capability of the interface such that not only can a model be made of the desired structure, but a model can also be made of the unaffected side of a structure and be used as a template for reconstruction implants.

4.4.5 Error Analysis

One of the big limitations of this project was the lack of error analysis to validate the geometric validity of the created meshes. An error analysis should be done in the future to compare the resultant mesh to a reference or ground truth mesh. As discussed previously, there are various methods of creating the reference mesh, some possible methods would include performing CT scans of a phantom of known dimensions so that measurements of the mesh including such as the dimensions or volume could be performed and directly compared to the phantom. Similarly, a process in which CT scans of cadaveric specimens could be used similar to (Lalone et al. 2015; Rathnayaka et al. 2011, 2012; Stephen et al. 2020) where after scanning, the soft tissue was removed, and a scanner was used to create a reference model. The reference model could also be created by following a similar process in a standardly used software so that the resultant model is compared to a clinical or practical standard.

4.4.6 3D Printing Testing

The basis of this project was to create a methodology that would create a 3D printable object. No 3D printing was done throughout the project and in the future segmented ROIs would be 3D printed. This would allow the number of repair processes and the length of time to repair the mesh required for 3D printing to be evaluated. It would also provide an opportunity to evaluate the size and shape of the resultant model itself. An object of known dimensions, such as a dried femur, could be scanned. The resultant full model and multiple segmented ROI models would then be 3D printed. The full 3D printed model would be directly compared to the dried bone, and the curvature and the thickness of the segmented ROI would be compared to the bone. In this way, the method to create 3D printable objects would be validated.

4.4.7 Comparison to Other Methods

A beneficial comparison that could be made in the future is the direct comparison of this technique to standard techniques in the field. Currently, there have been no direct comparisons to other techniques performed. A rough comparison of processing time was discussed earlier however, the study that was used as comparison times (Matsiushevich et al. 2019), used different anatomical structures than what was done throughout this project. Since the structures were different, the comparison of the time is limited as the structures have different complexities and sizes so the processing time would likely vary. In the future, the application of this technique to more anatomical structures would be beneficial for validating the methodology and its use in various fields. A comprehensive comparison of this method to standard techniques and software should also be done. This comparison would include multiple different types of anatomical structures and apply similar techniques in at least three other standard software. The processing time, error of the resultant meshes, the ease of use, and the various tools and capabilities of the software would be evaluated. This type of comparison could be done by experts in the field, which would also provide insight into improvements, and desired capabilities of the software.

4.5 Conclusion

Through this project, a methodology was created that extracted a 3D point cloud of bone from a set of CT images after thresholding and edge detection were applied. The 3D point cloud was used to create a 3D mesh of the bone, and a panel was created in Blender to act as a user interface. The user interface provided the ability to specify the location of CT images to open, specify a threshold used to segment bone from the CT images, and save the resultant mesh to the desired file location. The interface also allowed for a section of the bone mesh model to be selected by the user which was projected to a certain thickness through the mesh, the ROI was segmented and a separate STL file that contained a shape with the desired thickness was saved. Despite the mesh itself requiring refinement to help improve its quality and additional features which could be added to the interface to make it more user-friendly and expand its capabilities, the project was successful at creating a model from the bone which was able to be manipulated by a user. Future iterations or expansions on this project would be useful in helping to create 3D objects that may be tracked during surgical procedures and the planning of surgical procedures through modelling of complex anatomy or aid in the creation of implants.

Bibliography

Aamodt, A et al. 1999. "Determination of the Hounsfield Value for CT-Based Design of Custom Femoral Stems." *The Journal of Bone and Joint Surgery. British volume* 81-B(1): 143–47. https://doi.org/10.1302/0301-620X.81B1.0810143.

Abdullah, Nizam et al. 2006. "Dimensional Accuracy of the Skull Models Produced by Rapid Prototyping Technology Using Stereolithography Apparatus." *Archives of Orofacial Sciences* 1.

Alexiou, Evangelos, and Touradj Ebrahimi. 2018. "Point Cloud Quality Assessment Metric Based on Angular Similarity."

ALoopingIcon. "Measuring the Difference between Two Meshes." http://meshlabstuff.blogspot.com/2010/01/measuring-difference-between-two-meshes.html.

Alshipli, M. "How Can I Convert Pixel Intensity Values to Housefield (CT Number) ?" *ResearchGate*. https://www.researchgate.net/post/How_can_I_convert_pixel_intensity_values_to_housefield_CT_number.

Aspert, N, D Santa-Cruz, and T Ebrahimi. 2002. "MESH: Measuring Errors between Surfaces Using the Hausdorff Distance." In *Proceedings. IEEE International Conference on Multimedia and Expo*, , 705–8 vol.1.

Autodesk Inc. 2020. "Autodesk Meshmixer." https://www.meshmixer.com/.

Baan, Frank et al. 2016. "A New 3D Tool for Assessing the Accuracy of Bimaxillary Surgery: The OrthoGnathicAnalyser." *PLOS ONE* 11(2): e0149625. https://doi.org/10.1371/journal.pone.0149625.

Barrett, Julia F, and Nicholas Keat. 2004. "Artifacts in CT: Recognition and Avoidance." *RadioGraphics* 24(6): 1679–91. https://doi.org/10.1148/rg.246045065.

Bell, Daniel, and Candace Moore. 2020. "Segmentation." *Radiopaedia.org*. http://radiopaedia.org/articles/73551 (November 15, 2021).

Berger, Matthew et al. 2017. "A Survey of Surface Reconstruction from Point Clouds." *Computer Graphics Forum* 36(1): 301–29. https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.12802.

Bernardini, F et al. 1999. "The Ball-Pivoting Algorithm for Surface Reconstruction." *IEEE Transactions on Visualization and Computer Graphics* 5(4): 349–59.

Bibb, Richard, and John Winder. 2010. "A Review of the Issues Surrounding Three-Dimensional Computed Tomography for Medical Modelling Using Rapid Prototyping Techniques." *Radiography* 16(1): 78–83.

https://www.sciencedirect.com/science/article/pii/S1078817409000996.

Bishop, Elliot S et al. 2017. "3-D Bioprinting Technologies in Tissue Engineering and Regenerative Medicine: Current and Future Trends." *Genes & Diseases* 4(4): 185–95. https://www.sciencedirect.com/science/article/pii/S2352304217300673.

Blender Foundation. 2021a. "Context Access (Bpy.Context)." *Context Access (bpy.context) - Blender Python API*. https://docs.blender.org/api/current/bpy.context.html#bpy.context.scene.

———. 2021b. "Operator(BPY_STRUCT)." *Operator(bpy_struct) - Blender Python API*. https://docs.blender.org/api/current/bpy.types.Operator.html.

———. 2021c. "PropertyGroup(BPY_STRUCT)." *PropertyGroup(bpy_struct) - Blender Python API*. https://docs.blender.org/api/current/bpy.types.PropertyGroup.html.

———. 2021d. "Python API Overview." *Python API Overview - Blender Python API*. https://docs.blender.org/api/current/info_overview.html.

———. 2021e. "Quickstart." *Quickstart - Blender Python API*. https://docs.blender.org/api/current/info_quickstart.html.

———. "Add-Ons." *Add-ons - Blender Manual*. https://docs.blender.org/manual/en/2.91/editors/preferences/addons.html.

Blender Online Community. 2018. "Blender - a 3D Modelling and Rendering Package." http://www.blender.org.

Boris. "Marching Cubes 3D Tutorial." *BorisTheBrave.Com*. https://www.boristhebrave.com/2018/04/15/marching-cubes-3d-tutorial/ (November 15, 2021).

Bourne, M. 2021. "Perpendicular Distance from a Point to a Line." *Interactive Mathematics*. https://www.intmath.com/plane-analytic-geometry/perpendicular-distance-point-line.php.

Bradski, G. 2000. "The OpenCV Library." *Dr. Dobb's Journal of Software Tools*.

Cai, Tianrun et al. 2015. "The Residual STL Volume as a Metric to Evaluate Accuracy and Reproducibility of Anatomic Models for 3D Printing: Application in the Validation of 3D-Printable Models of Maxillofacial Bone from Reduced Radiation Dose CT Images." *3D Printing in Medicine* 1(1): 2. https://doi.org/10.1186/s41205-015-0003-3.

Canny, John. 1986. "A Computational Approach to Edge Detection." *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-8(6): 679–98.

Carolo, L. "Meshmixer Tutorial for Beginners." *All3DP*. https://all3dp.com/2/meshmixer-tutorial-easy-steps-beginners/ (November 9, 2021).

Cheng, Siu-Wing, Tamal K. Dey, Jonathan Shewchuk, and Sartaj Sahni. 2012. *Delaunay Mesh Generation : Algorithms and Mathematical Analysis*. 1st ed. CRC Press LLC. https://ebookcentral.proquest.com/lib/mcmu/detail.action?docID=1565535.

Chougule, Vikas, Arati Mulay, and B.. Ahuja. 2013. "Conversions of CT Scan Images into 3D Point Cloud Data for the Development of 3D Solid Model Using B-Rep Schemeitle." In *Conference: Proceedings of the Intern Ational Conference on Precision, Meso, Micro and Nano Engineering,*.

Chua, Chee Kai, Chee How Wong, and Wai Yee Yeong. 2017. "Chapter Four - Software and Data Format." In *Standards, Quality Control, and Measurement Sciences in 3D Printing and Additive Manufacturing*, eds. Chee Kai Chua, Chee How Wong, and Wai Yee Yeong. Academic Press, 75–94. https://www.sciencedirect.com/science/article/pii/B9780128134894000040.

Cignoni, Paolo et al. 2008. 1 Computing *MeshLab: An Open-Source Mesh Processing Tool.*

Cignoni, Paolo, Claudio Rocchini, and Roberto Scopigno. 1998. "METRO: Measuring Error on Simplified Surfaces." *Computer Graphics Forum* 17: 167–74.

Clark, Kenneth et al. 2013. "The Cancer Imaging Archive (TCIA): Maintaining and Operating a Public Information Repository." *Journal of Digital Imaging* 26(6): 1045–57. https://doi.org/10.1007/s10278-013-9622-7.

"Cloud-to-Mesh Distance." 2015. *CloudCompareWiki*. https://www.cloudcompare.org/doc/wiki/index.php?title=Cloud-to-Mesh_Distance.

"Cloud to Mesh Distances." 2017. *CloudCompare forum*. http://www.danielgm.net/cc/forum/viewtopic.php?t=2668.

Das, Birendra Kishore. 2015. "Basic Principles of CT Imaging." In *Positron Emission Tomography: A Guide for Clinicians*, ed. Birendra Kishore Das. New Delhi: Springer India, 181–84. https://doi.org/10.1007/978-81-322-2098-5_20.

Ding, Lijun, and Ardeshir Goshtasby. 2001. "On the Canny Edge Detector." *Pattern Recognition* 34(3): 721–25. https://www.sciencedirect.com/science/article/pii/S0031320300000236.

Diwakar, Manoj, and Manoj Kumar. 2018. "A Review on CT Image Noise and Its Denoising." *Biomedical Signal Processing and Control* 42: 73–88. https://www.sciencedirect.com/science/article/pii/S1746809418300107.

Edelsbrunner, Herbert, and Ernst Mucke. 1994. "Three-Dimensional Alpha Shapes." *ACM Transactions on Graphics* 13.

van Eijnatten, Maureen et al. 2018. "CT Image Segmentation Methods for Bone Used in Medical Additive Manufacturing." *Medical Engineering & Physics* 51: 6–16. https://www.sciencedirect.com/science/article/pii/S1350453317302631.

Fakhar, Hooriyeh Bashizade, Raheleh Emami, Kave Moloudi, and Farzaneh Mosavat. 2018. "Effects of Artifact Removal on Cone-Beam Computed Tomography Images." *Dental research journal* 15(2): 89–94. https://pubmed.ncbi.nlm.nih.gov/29576771.

Formlabs. "Preform 3D Printing Software: Prepare Your Models for Printing." *Formlabs*. https://formlabs.com/software/.

Friedli, Luca et al. 2020. "The Effect of Threshold Level on Bone Segmentation of Cranial Base Structures from CT and CBCT Images." *Scientific Reports* 10(1): 7361. https://doi.org/10.1038/s41598-020-64383-9.

Gangl, D. "Using Blender's Filebrowser with Python." *Sinestesia*. https://sinestesia.co/blog/tutorials/using-blenders-filebrowser-with-python/.

Gardiner, James D., Julia Behnsen, and Charlotte A. Brassey. 2018. "Alpha Shapes: Determining 3D Shape Complexity across Morphologically Diverse Structures." *BMC Evolutionary Biology*.

Gelaude, F, J Vander Sloten, and B Lauwers. 2008. "Accuracy Assessment of CT-Based Outer Surface Femur Meshes." *Computer Aided Surgery* 13(4): 188–99. https://doi.org/10.3109/10929080802195783.

George, Elizabeth, Peter Liacouras, Frank J Rybicki, and Dimitrios Mitsouras. 2017. "Measuring and Establishing the Accuracy and Reproducibility of 3D Printed Medical Models." *RadioGraphics* 37(5): 1424–50. https://doi.org/10.1148/rg.2017160165.

Giambini, Hugo et al. 2015. "The Effect of Quantitative Computed Tomography Acquisition Protocols on Bone Mineral Density Estimation." *Journal of biomechanical engineering* 137(11): 114502. https://pubmed.ncbi.nlm.nih.gov/26355694.

Girardeau-Montaut, D. "Cloud Compare." *CloudCompare*. https://www.cloudcompare.org/.

Glide-Hurst, C, D Chen, H Zhong, and I J Chetty. 2013. "Changes Realized from Extended Bit-Depth and Metal Artifact Reduction in CT." *Medical Physics* 40(6Part1): 61711. https://doi.org/10.1118/1.4805102.

Goldman, Lee W. 2007. "Principles of CT and CT Technology." *Journal of Nuclear Medicine Technology* 35(3): 115 LP – 128. http://tech.snmjournals.org/content/35/3/115.abstract.

Graham, R N J, R W Perriss, and A F Scarsbrook. 2005. "DICOM Demystified: A Review of Digital File Formats and Their Use in Radiological Practice." *Clinical Radiology* 60(11): 1133–40. https://www.sciencedirect.com/science/article/pii/S0009926005002199.

Harris, Charles R et al. 2020. "Array Programming with NumPy." *Nature* 585(7825):

357–62. https://doi.org/10.1038/s41586-020-2649-2.

Heng, Franklin. 2019. "A Comprehensive Guide To Visualizing and Analyzing DICOM Images in Python." *Medium*. https://hengloose.medium.com/a-comprehensive-starter-guide-to-visualizing-and-analyzing-dicom-images-in-python-7a8430fcb7ed (July 1, 2020).

Hounsfield, G. N. 1973. "Computerized Transverse Axial Scanning (Tomography): I. Description of System." *British Journal of Radiology*.

"How to Convert an Image from RGB to Grayscale in Python." *Kite*. https://www.kite.com/python/answers/how-to-convert-an-image-from-rgb-to-grayscale-in-python#:~:text=Use numpy.,image from RGB to grayscale&text=imread(fname) to get a,previous result array to grayscale.

Huang, Jing, and Suya You. 2012. "Point Cloud Matching Based on 3D Self-Similarity." In *2012 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, , 41–48.

Hunter, J D. 2007. "Matplotlib: A 2D Graphics Environment." *Computing in Science \& Engineering* 9(3): 90–95.

Huotilainen, Eero et al. 2014. "Inaccuracies in Additive Manufactured Medical Skull Models Caused by the DICOM to STL Conversion Process." *Journal of Cranio-Maxillofacial Surgery* 42(5): e259–65. https://www.sciencedirect.com/science/article/pii/S1010518213002862.

Iassonov, Pavel, Thomas Gebrenegus, and Markus Tuller. 2009. "Segmentation of X-Ray Computed Tomography Images of Porous Materials: A Crucial Step for Characterization and Quantitative Analysis of Pore Structures." *Water Resources Research* 45(9). https://doi.org/10.1029/2009WR008087.

Innolitics LLC. "Rescale Intercept Attribute – DICOM Standard Browser." *Dicom Standard browser*. https://dicom.innolitics.com/ciods/ct-image/ct-image/00281052.

———. "Rescale Slope Attribute – DICOM Standard Browser." *Dicom Standard browser*. available: https://dicom.innolitics.com/ciods/ct-image/ct-image/00281053.

Jain, Ramesh, Kasturi Rangachar, and Brian G. Schunck. 1995. "Edge Detection." In *Machine Vision*, McGraw-Hill Inc., 140–85. https://www.cse.usf.edu/~r1k/MachineVisionBook/MachineVision.files/MachineVision_Chapter5.pdf.

Javaid, Mohd., and Abid Haleem. 2018. "Additive Manufacturing Applications in Medical Cases: A Literature Based Review." *Alexandria Journal of Medicine* 54(4): 411–22. https://doi.org/10.1016/j.ajme.2017.09.003.

Kamio, Takashi, Madoka Suzuki, Rieko Asaumi, and Taisuke Kawai. 2020. "DICOM Segmentation and STL Creation for 3D Printing: A Process and Software Package

Comparison for Osseous Anatomy." *3D Printing in Medicine* 6(1): 17. https://doi.org/10.1186/s41205-020-00069-2.

Kazhdan, Michael, Matthew Bolitho, and Hugues Hoppe. 2006. "Poisson Surface Reconstruction." In *Proceedings of the Fourth Eurographics Symposium on Geometry Processing*,.

Lalone, Emily A et al. 2015. "Accuracy Assessment of 3D Bone Reconstructions Using CT: An Intro Comparison." *Medical Engineering & Physics* 37(8): 729–38. https://www.sciencedirect.com/science/article/pii/S1350453315001058.

Levi, C, J E Gray, E C McCullough, and R R Hattery. 1982. "The Unreliability of CT Numbers as Absolute Values." *American Journal of Roentgenology* 139(3): 443–47. https://doi.org/10.2214/ajr.139.3.443.

Lim Fat, Daren et al. 2012. "The Hounsfield Value for Cortical Bone Geometry in the Proximal Humerus—an in Vitro Study." *Skeletal Radiology* 41(5): 557–68. https://doi.org/10.1007/s00256-011-1255-7.

Lobos, Claudio, and Rodrigo Rojas Moraleda. 2013. *From Segmented Medical Images to Surface and Volume Meshes, Using Existing Tools and Algorithms.*

Lorensen, William E, and Harvey E Cline. 1987. "Marching Cubes: A High Resolution 3D Surface Construction Algorithm." In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, New York, NY, USA: Association for Computing Machinery, 163–169. https://doi-org.libaccess.lib.mcmaster.ca/10.1145/37401.37422.

Marro, Alessandro, Taha Bandukwala, and Walter Mak. 2016. "Three-Dimensional Printing and Medical Imaging: A Review of the Methods and Applications." *Current Problems in Diagnostic Radiology*.

Martin, Charys M et al. 2013. "Comparison of 3D Reconstructive Technologies Used for Morphometric Research and the Translation of Knowledge Using a Decision Matrix." *Anatomical Sciences Education* 6(6): 393–403. https://doi.org/10.1002/ase.1367.

Matsiushevich, Katsiaryna, Claudio Belvedere, Alberto Leardini, and Stefano Durante. 2019. "Quantitative Comparison of Freeware Software for Bone Mesh from DICOM Files." *Journal of Biomechanics* 84(Complete): 247–51. http://resolver.scholarsportal.info/resolve/00219290/v84icomplete/247_qcofsfbmfdf.

Mitsouras, Dimitris et al. 2015. "Medical 3D Printing for the Radiologist." *RadioGraphics* 35(7): 1965–88. https://doi.org/10.1148/rg.2015140320.

Nguyen, Anh, and Bac Le. 2013. "3D Point Cloud Segmentation: A Survey." In *2013 6th IEEE Conference on Robotics, Automation and Mechatronics (RAM)*, , 225–30.

Noser, Hansrudi, Thomas Heldstab, Beat Schmutz, and Lukas Kamer. 2011. "Typical

Accuracy and Quality Control of a Process for Creating CT-Based Virtual Bone Models." *Journal of Digital Imaging* 24(3): 437–45. https://doi.org/10.1007/s10278-010-9287-4.

Nurunnabi, Abdul, Geoff West, and David Belton. 2015. "Outlier Detection and Robust Normal-Curvature Estimation in Mobile Laser Scanning 3D Point Cloud Data." *Pattern Recognition* 48(4): 1404–19. http://resolver.scholarsportal.info/resolve/00313203/v48i0004/1404_odarnels3pcd.

Open3D.org. "Surface Reconstruction." *Surface Reconstruction - Open3D latest (664eff5) documentation*. http://www.open3d.org/docs/latest/tutorial/Advanced/surface_reconstruction.html.

Pagès, Alban, Maxime Sermesant, and Pascal Frey. 2005. "Generation of Computational Meshes from MRI and CT-Scan Data." *ESAIM: Proc.* 14: 213–23. https://doi.org/10.1051/proc:2005016.

Parthasarathy, Jayanthi. 2014. "3D Modeling, Custom Implants and Its Future Perspectives in Craniofacial Surgery." *Annals of Maxillofacial Surgery*.

Pixmeo. 2021. "OsiriX DICOM Viewer." https://www.osirix-viewer.com/ (November 4, 2021).

"Point Cloud." 2020. *Point Cloud - Open3D 0.10.0 documentation*. http://www.open3d.org/docs/0.10.0/tutorial/Basic/pointcloud.html.

Poux, Florent. "5-Step Guide to Generate 3D Meshes from Point Clouds with Python." *Towards Data Science*. https://towardsdatascience.com/5-step-guide-to-generate-3d-meshes-from-point-clouds-with-python-36bad397d8ba.

Prince, Simon J D. 2012. "Image Preprocessing and Feature Extraction." In *Computer Vision: Models, Learning, and Inference*, Cambridge University Press, 269–294. http://www.computervisionmodels.com/.

Python Software Foundation. 2021a. "Math-Mathematical Functions." https://docs.python.org/3.8/library/math.html.

———. 2021b. "Os.Path - Common Pathname Manipulations." *os.path - Common pathname manipulations - Python 3.8.12 documentation*. https://docs.python.org/3.8/library/os.path.html#os.path.isdir.

———. 2021c. "Random - Generate Pseudo-Random Numbers." *random - Generate pseudo-random numbers - Python 3.8.12 documentation*. https://docs.python.org/3.8/library/random.html.

———. "Python." https://www.python.org/.

R Core Team. 2020. "R: A Language and Environment for Statistical Computing." https://www.r-project.org/.

Racasan, R, D Popescu, C Neamtu, and M Dragomir. 2010. "Integrating the Concept of Reverse Engineering in Medical Applications." In *2010 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*, , 1–5.

Rapponotti, Brett, Michael Snowden, and Allen Zeng. "Point Cloud to Mesh, Ball-Pivoting Algorithm." *Point Cloud to Mesh: BPA*. https://cs184team.github.io/cs184-final/writeup.html (November 15, 2021).

Rathnayaka, Kanchana et al. 2012. "Quantification of the Accuracy of MRI Generated 3D Models of Long Bones Compared to CT Generated 3D Models." *Medical Engineering & Physics* 34(3): 357–63. https://www.sciencedirect.com/science/article/pii/S1350453311001925.

Rathnayaka, Kanchana, Tony Sahama, Michael A Schuetz, and Beat Schmutz. 2011. "Effects of CT Image Segmentation Methods on the Accuracy of Long Bone 3D Reconstructions." *Medical Engineering & Physics* 33(2): 226–33. https://www.sciencedirect.com/science/article/pii/S1350453310002225.

Rengier, F et al. 2010. "3D Printing Based on Imaging Data: Review of Medical Applications." *International Journal of Computer Assisted Radiology and Surgery* 5(4): 335–41. https://doi.org/10.1007/s11548-010-0476-x.

Rocchini, C et al. 2001. "Marching Intersections: An Efficient Resampling Algorithm for Surface Management." In *Proceedings International Conference on Shape Modeling and Applications*, , 296–305.

Sansoni, Giovanna, Marco Trebeschi, and Franco Docchio. 2009. "State-of-The-Art and Applications of 3D Imaging Sensors in Industry, Cultural Heritage, Medicine, and Criminal Investigation." *Sensors* 9(1): 568–601. https://www.mdpi.com/1424-8220/9/1/568.

Schindelin, Johannes et al. 2012. "Fiji: An Open-Source Platform for Biological-Image Analysis." *Nature Methods*.

Schmutz, Beat et al. 2014. "Magnetic Resonance Imaging: An Accurate, Radiation-Free, Alternative to Computed Tomography for the Primary Imaging and Three-Dimensional Reconstruction of the Bony Orbit." *Journal of Oral and Maxillofacial Surgery* 72(3): 611–18. https://www.sciencedirect.com/science/article/pii/S0278239113011397.

Schmutz, Beat, Martin Wullschleger, and Michael Schuetz. 2007. "The Effect of CT Slice Spacing on the Geometry of 3D Models." *Proceedings 6th Australasian biomechanics conference*.

scikit-image development team. "Module: Feature." *Module: feature - skimage v0.17.2 docs*. https://scikit-image.org/docs/0.17.x/api/skimage.feature.html?highlight=canny#skimage.feature.canny.

———. "Module: Filters." *Module: filters - skimage v0.17.2 docs*. https://scikit-image.org/docs/0.17.x/api/skimage.filters.html?highlight=gaussian#skimage.filters.gaussian (November 16, 2021b).

———. "Module: Io." https://scikit-image.org/docs/dev/api/skimage.io.html#skimage.io.imread.

Scolozzi, Paolo. 2012. "Maxillofacial Reconstruction Using Polyetheretherketone Patient-Specific Implants by 'Mirroring' Computational Planning." *Aesthetic Plastic Surgery* 36(3): 660–65. https://doi.org/10.1007/s00266-011-9853-2.

Sculpteo. "How to Fix Non-Manifold Geometry Issues on 3D Models." *Sculpteo*. https://www.sculpteo.com/en/3d-learning-hub/create-3d-file/fix-non-manifold-geometry/ (November 15, 2021).

Seol, Young-Joon et al. 2014. "Bioprinting Technology and Its Applications." *European Journal of Cardio-Thoracic Surgery* 46(3): 342–48. https://doi.org/10.1093/ejcts/ezu148.

Shaheen, E, Y Sun, R Jacobs, and C Politis. 2017. "Three-Dimensional Printed Final Occlusal Splint for Orthognathic Surgery: Design and Validation." *International Journal of Oral & Maxillofacial Surgery* 46(1): 67–71. http://resolver.scholarsportal.info/resolve/09015027/v46i0001/67_tpfosfosdav.

Shin, Ji Hoon et al. 2002. "The Quality of Reconstructed 3D Images in Multidetector-Row Helical CT: Experimental Study Involving Scan Parameters." *kjr* 3(1): 49–56. http://dx.doi.org/10.3348/kjr.2002.3.1.49.

Sing, Swee Leong, Jia An, Wai Yee Yeong, and Florencia Edith Wiria. 2016. "Laser and Electron-Beam Powder-Bed Additive Manufacturing of Metallic Implants: A Review on Processes, Materials and Designs." *Journal of Orthopaedic Research* 34(3): 369–85. https://doi.org/10.1002/jor.23075.

Singare, Sekou et al. 2009. "Rapid Prototyping Assisted Surgery Planning and Custom Implant Design." *Rapid Prototyping Journal* 15(1): 19–23. https://doi.org/10.1108/13552540910925027.

Statistical Engineering Division Dataplot. "Cosine Distance Cosine Similarity Angular Cosine Distance Angular Cosine Similarity." *National Institute of Standards and Technology*. https://www.itl.nist.gov/div898/software/dataplot/refman2/auxillar/cosdist.htm.

Stephen, Joanna M., James DF Calder, Andy Williams, and Hadi El Daou. 2020. "Comparative Accuracy of Lower Limb Bone Geometry Determined Using MRI, CT, and Direct Bone 3D Models." *Journal of Orthopaedic Research*: jor.24923. https://onlinelibrary.wiley.com/doi/10.1002/jor.24923.

Stock, Michala K et al. 2020. "The Importance of Processing Procedures and Threshold Values in CT Scan Segmentation of Skeletal Elements: An Example Using the

Immature Os Coxa." *Forensic Science International* 309: 110232. https://www.sciencedirect.com/science/article/pii/S0379073820300943.

Sugimoto, Kazuo, Robert A Cohen, Dong Tian, and Anthony Vetro. 2017. "Trends in Efficient Representation of 3D Point Clouds." In *2017 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*, , 364–69.

Sullivan, C., and Alexander Kaszynski. 2019. "PyVista: 3D Plotting and Mesh Analysis through a Streamlined Interface for the Visualization Toolkit (VTK)." *Journal of Open Source Software*.

Szymor, Piotr, Marcin Kozakiewicz, and Raphael Olszewski. 2016. "Accuracy of Open-Source Software Segmentation and Paper-Based Printed Three-Dimensional Models." *Journal of Cranio-Maxillofacial Surgery* 44(2): 202–9. https://www.sciencedirect.com/science/article/pii/S101051821500373X.

The NumPy Community. 2021a. "Numpy.Cross." *numpy.cross - NumPy v1.21 Manual*. https://numpy.org/doc/stable/reference/generated/numpy.cross.html.

———. 2021b. "Numpy.Delete." *numpy.delete - NumPy v1.21 Manual*. https://numpy.org/doc/stable/reference/generated/numpy.delete.html.

———. 2021c. "Numpy.Dot." *numpy.dot - NumPy v1.21 Manual*. https://numpy.org/doc/stable/reference/generated/numpy.dot.html.

———. 2021d. "Numpy.Linalg.Norm." *numpy.linalg.norm - NumPy v1.21 Manua*. https://numpy.org/doc/stable/reference/generated/numpy.linalg.norm.html.

———. 2021e. "Numpy.Linspace." *numpy.linspace - NumPy v1.23.dev0 Manual*. https://numpy.org/devdocs/reference/generated/numpy.linspace.html?highlight=linspace#numpy.linspace.

———. 2021f. "Numpy.Mean." https://numpy.org/doc/stable/reference/generated/numpy.mean.html (October 15, 2021).

———. 2021g. "Numpy.Rint." *numpy.rint - NumPy v1.21 Manual*. https://numpy.org/doc/stable/reference/generated/numpy.rint.html.

———. 2021h. "Numpy.Std." https://numpy.org/doc/stable/reference/generated/numpy.std.html.

———. 2021i. "Numpy.Unique." *numpy.unique - NumPy v1.21 Manual*. https://numpy.org/doc/stable/reference/generated/numpy.unique.html.

———. 2021j. "Numpy.Vdot." *numpy.vdot - NumPy v1.21 Manual*. https://numpy.org/doc/stable/reference/generated/numpy.vdot.html.

———. 2021k. "Numpy.Where." *numpy.where - NumPy v1.23.dev0 Manual*.

https://numpy.org/devdocs/reference/generated/numpy.where.html?highlight=where #numpy.where.

———. "Numpy.Arange." *numpy.arange - NumPy v1.21 Manual*. https://numpy.org/doc/stable/reference/generated/numpy.arange.html.

The NumPy Development Team. 2021. "Numpy.Random.Choice." *numpy.random.choice - NumPy v1.21 Manual*. https://numpy.org/doc/stable/reference/random/generated/numpy.random.choice.html.

The PyVista Developers. 2021. "Volume." *volume - PyVista 0.32.0 documentation*. https://docs.pyvista.org/api/core/_autosummary/pyvista.DataSet.volume.html#pyvista.DataSet.volume.

———. "Boolean Operations." *Boolean Operations - PyVista 0.32.0 documentation*. https://docs.pyvista.org/examples/01-filter/boolean-operations.html (November 17, 2021a).

———. "Delaunay_3d." *delaunay_3d - PyVista 0.32.0 documentation*. https://docs.pyvista.org/api/core/_autosummary/pyvista.UnstructuredGridFilters.delaunay_3d.html (March 3, 2021b).

The SciPy Community. 2021a. "Scipy.Spatial.KDTree.Query." https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.KDTree.query.html#scipy.spatial.KDTree.query.

———. 2021b. "Scipy.Spatial.Kdtree." *scipy.spatial.KDTree - SciPy v1.7.1 Manual*. https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.KDTree.html.

Tilotta, Françoise et al. 2009. "Construction and Analysis of a Head CT-Scan Database for Craniofacial Reconstruction." *Forensic Science International*.

"Time.Perf_counter() Function in Python." 2021. *tutorialspoint*. https://www.tutorialspoint.com/time-perf-counter-function-in-python.

Vallières, M, C R Freeman, S R Skamene, and I El Naqa. 2015a. "A Radiomics Model from Joint FDG-PET and MRI Texture Features for the Prediction of Lung Metastases in Soft-Tissue Sarcomas of the Extremities." *Physics in Medicine and Biology* 60(14): 5471–96. http://dx.doi.org/10.1088/0031-9155/60/14/5471.

Vallières, M, Carolyn R. Freeman, Sonia R. Skamene, and Issam El Naqa. 2015b. "A Radiomics Model from Joint FDG-PET and MRI Texture Features for the Prediction of Lung Metastases in Soft-Tissue Sarcomas of the Extremities." *The Cancer Imaging Archive*. http://doi.org/10.7937/K9/TCIA.2015.7GO2GSKS.

Varma, Dandu Ravi. 2012. "Managing DICOM Images: Tips and Tricks for the Radiologist." *The Indian journal of radiology & imaging* 22(1): 4–13. https://pubmed.ncbi.nlm.nih.gov/22623808.

Virtanen, Pauli et al. 2020. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python." *Nature Methods* 17(3): 261–72.

VTK. 2021. "VtkDelaunay3D Class Reference." https://vtk.org/doc/nightly/html/classvtkDelaunay3D.html.

Wallner, Jürgen et al. 2019. "A Review on Multiplatform Evaluations of Semi-Automatic Open-Source Based Image Segmentation for Cranio-Maxillofacial Surgery." *Computer Methods and Programs in Biomedicine* 182: 105102.

van der Walt, Stéfan et al. 2014. "Scikit-Image: Image Processing in {P}ython." *PeerJ* 2: e453. https://doi.org/10.7717/peerj.453.

"Why Does My File Display This Red Stuff?" *Home - Autodesk Community*. https://forums.autodesk.com/t5/meshmixer/why-does-my-file-display-this-red-stuff/td-p/8472198 (November 10, 2021).

Wittek, Adam et al. 2016. "From Finite Element Meshes to Clouds of Points: A Review of Methods for Generation of Computational Biomechanics Models for Patient-Specific Applications." *Annals of Biomedical Engineering* 44(1): 3–15. https://doi.org/10.1007/s10439-015-1469-2.

www.open3d.org. 2020. "Open3d.Geometry.Tetramesh." *open3d.geometry.TetraMesh - Open3D 0.10.0 documentation*. http://www.open3d.org/docs/0.10.0/python_api/open3d.geometry.TetraMesh.html#open3d.geometry.TetraMesh.

Zhang, Xiaomeng, Jing Wang, and Lei Xing. 2011. "Metal Artifact Reduction in X-Ray Computed Tomography (CT) by Constrained Optimization." *Medical physics* 38(2): 701–11. https://pubmed.ncbi.nlm.nih.gov/21452707.

Zhang, Yongjie, Chandrajit Bajaj, and Bong-Soo Sohn. 2003. "Adaptive and Quality 3D Meshing from Imaging Data." In *Proceedings of the Eighth ACM Symposium on Solid Modeling and Applications*, SM '03, New York, NY, USA: Association for Computing Machinery, 286–291. https://doi.org/10.1145/781606.781653.

Zhao, Feng, and Xianghua Xie. 2013. "An Overview on Interactive Medical Image Segmentation." *The Annals of the BMVA* 2013: 1–22.

Zhou, Qian Yi, Jaesik Park, and Vladlen Koltun. 2018. "Open3D: A Modern Library for 3D Data Processing." *arXiv*.

Appendix A: Python Package Versions

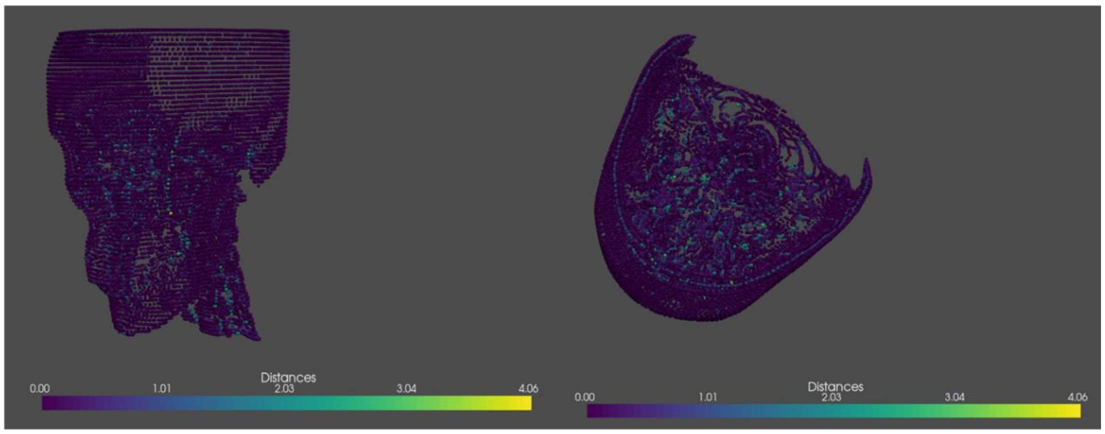
Different Python package versions were used throughout this project on the different computers and software that were used. I tried to make sure all the package versions were the same across all platforms, but in some cases, multiple versions of Python were installed or Blender was not compatible with the certain version, so they varied.

Table A. 1: *Python package versions used throughout this project.*

Python Package	**Personal Laptop**	**Remote Desktop**	**Blender (Personal Laptop)**	**Blender (Remote Desktop)**
Python	3.8.3	3.8.3	3.7.7	3.7.7
PyVista	0.26.1	0.26.1	0.31.1	0.32.1
Open3D	0.10.0.1	0.10.0.1	NA	NA
Pydicom	2.0.0	2.0.0	2.1.2	2.2.2
NumPy	1.18.4	1.20.3	1.20.3	1.17.5
SciPy	1.4.1	1.4.1	1.6.3	1.7.2
Scikit-image	0.17.2	0.17.2	0.18.1	0.18.3
OpenCV	4.3.0	4.3.0	NA	NA
Matplotlib	3.2.1	3.4.2	3.4.2	3.4.3

Appendix B: Histograms and Heat Maps for Comparison of Meshes for Sampling Algorithms

A



B

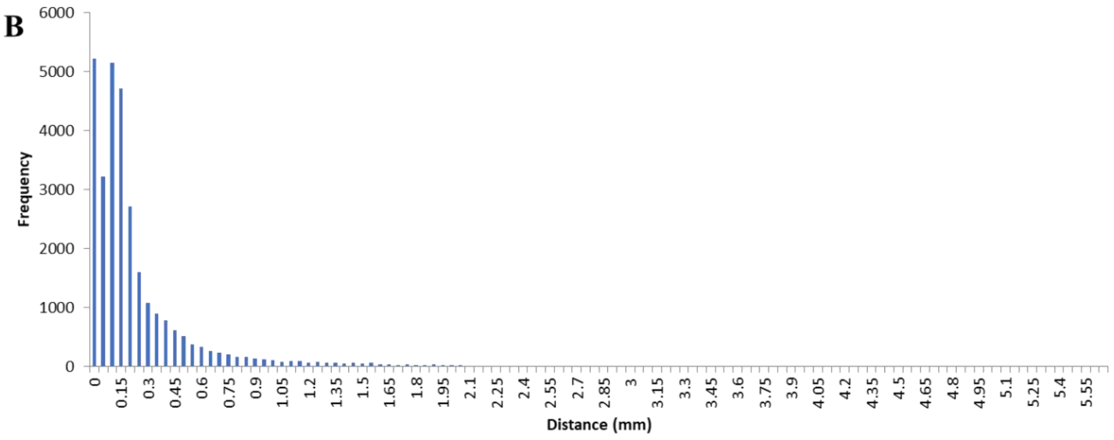
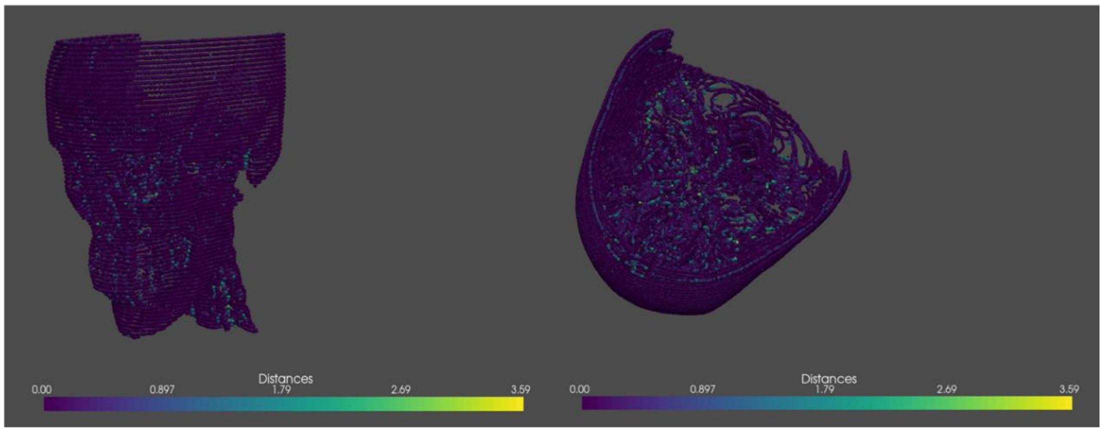


Figure B 1: *Heat map (A) and histogram (B) for the pseudo-uniform sampling of 20%.*

A



B

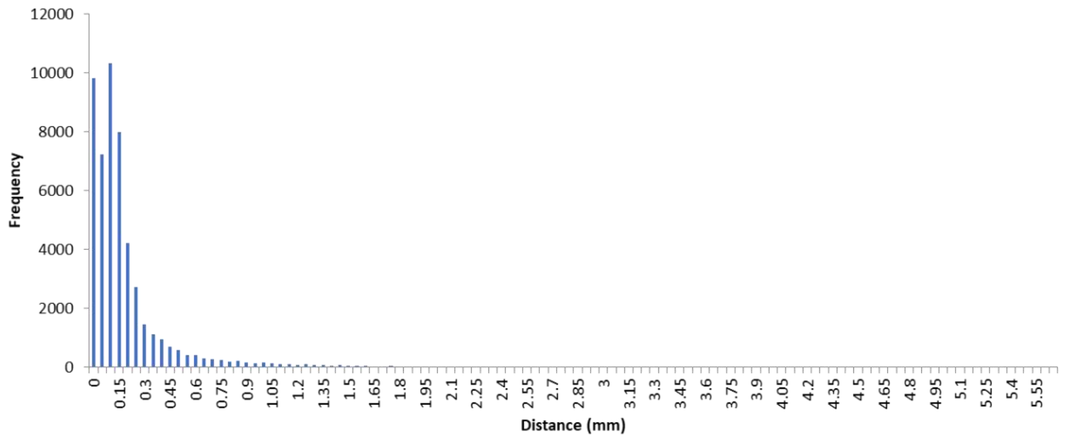
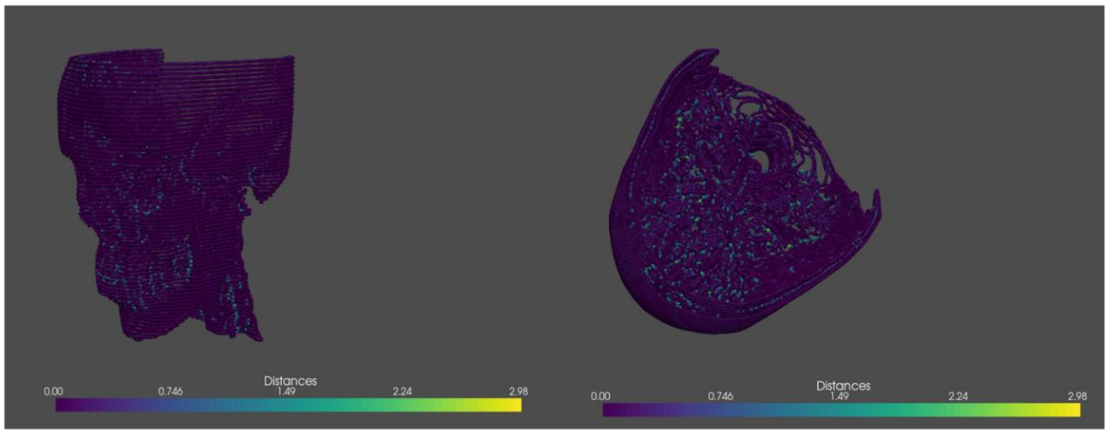


Figure B 2: *Heat map (A) and histogram (B) for the pseudo-uniform sampling of 30%.*

A



B

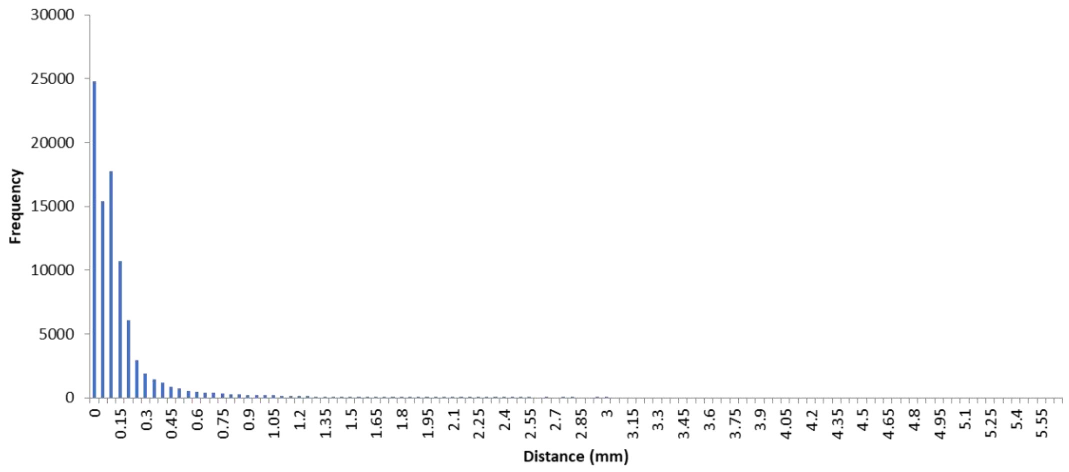
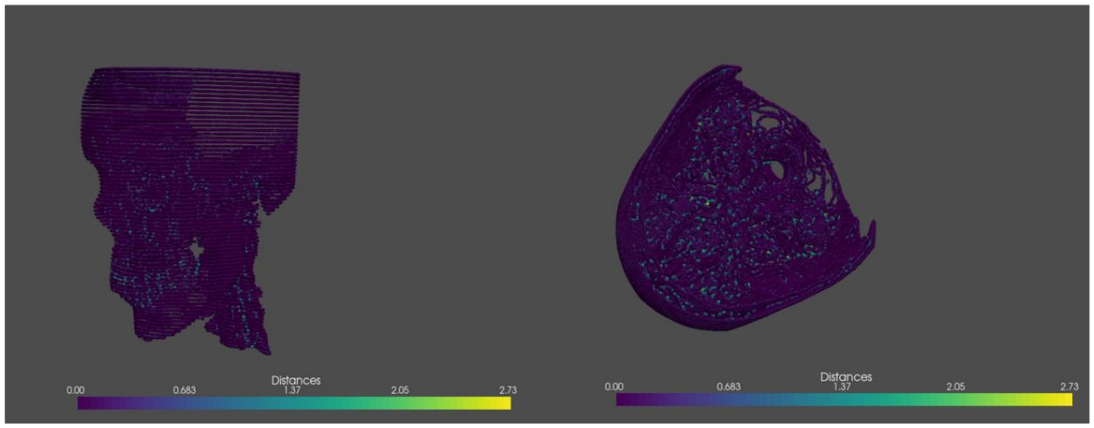


Figure B 3: *Heat map (A) and histogram (B) for the pseudo-uniform sampling of 50%.*

A



B

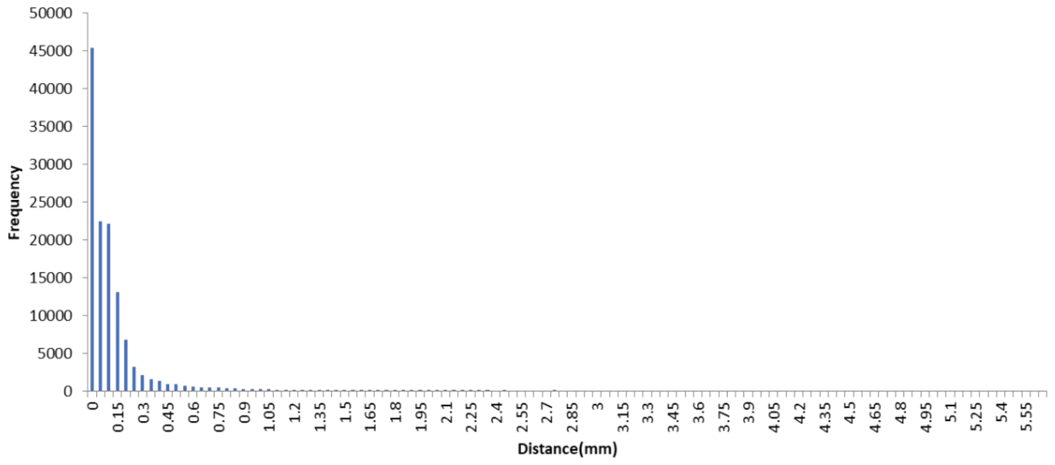
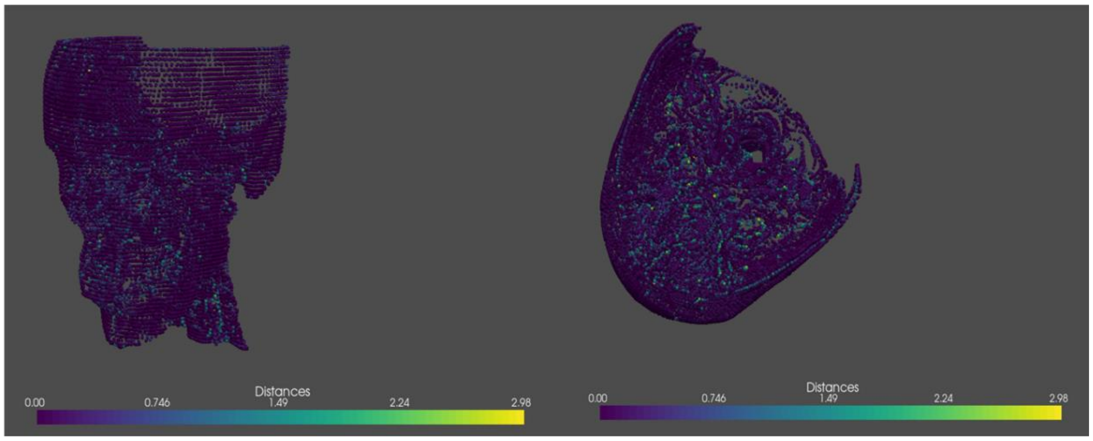


Figure B 4: *Heat map (A) and histogram (B) for the pseudo-uniform sampling of 70%.*

A



B

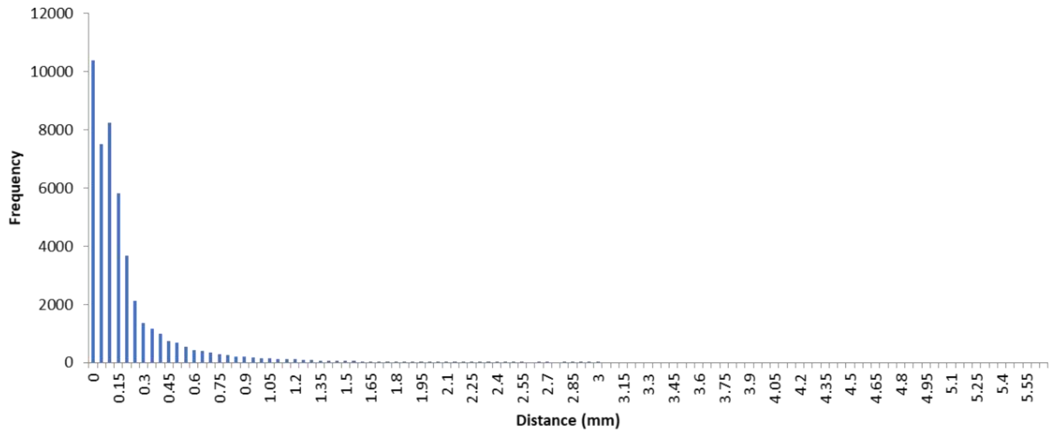
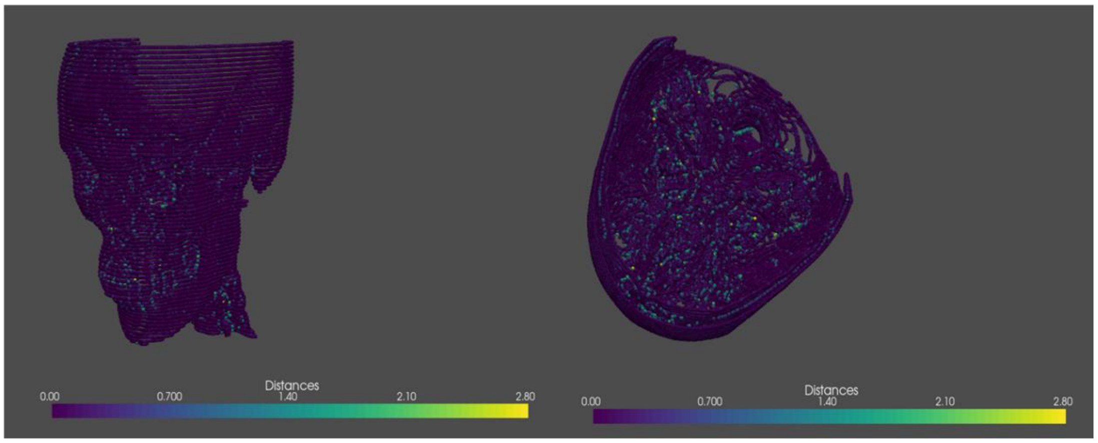


Figure B 5: *Heat map (A) and histogram (B) for the second targeted sampling experiment.*

A



B

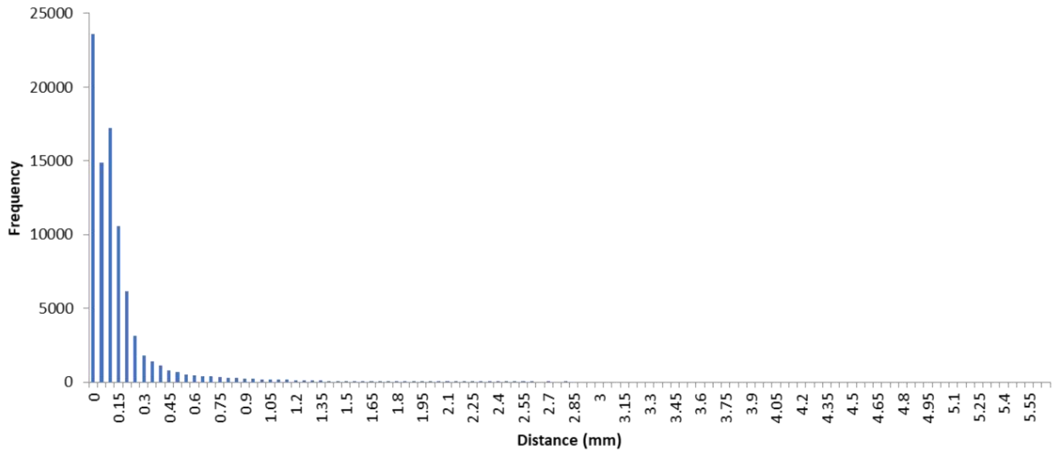


Figure B 6: *Heat map (A) and histogram (B) for the third targeted sampling experiment.*

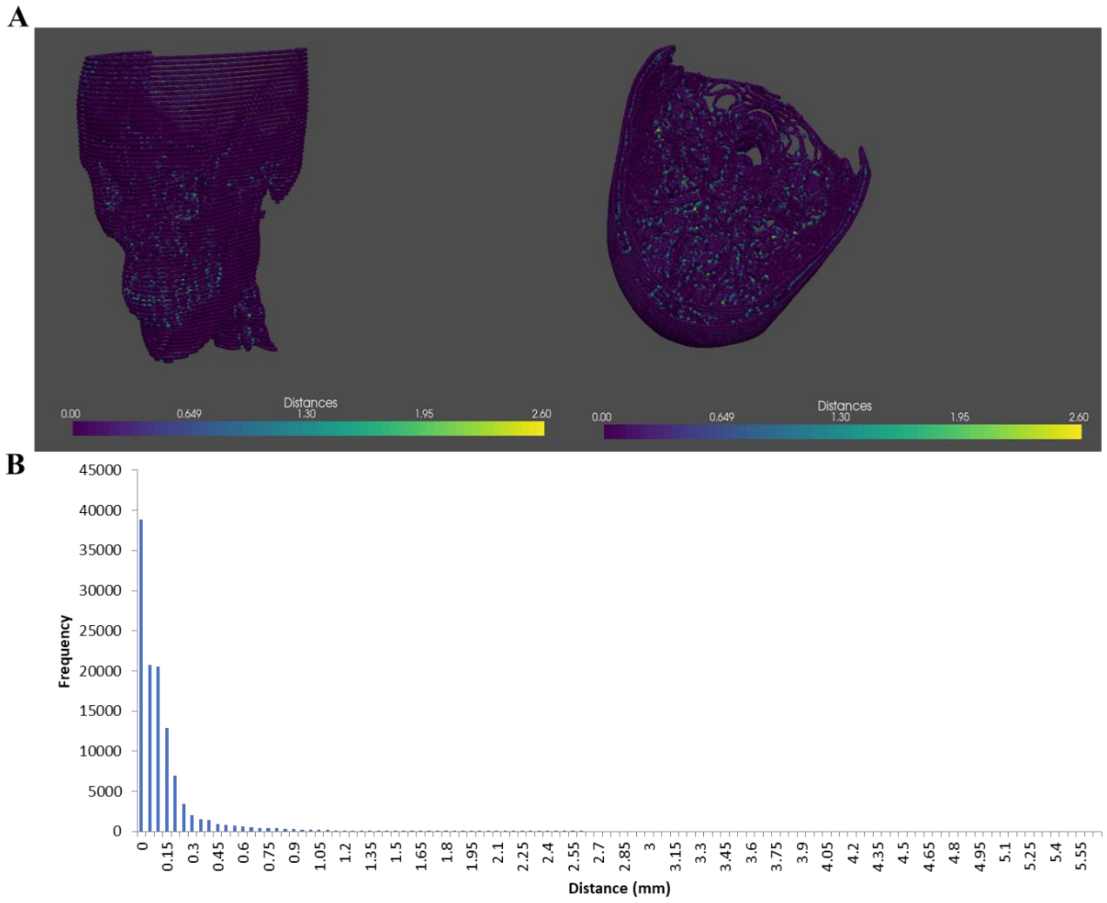
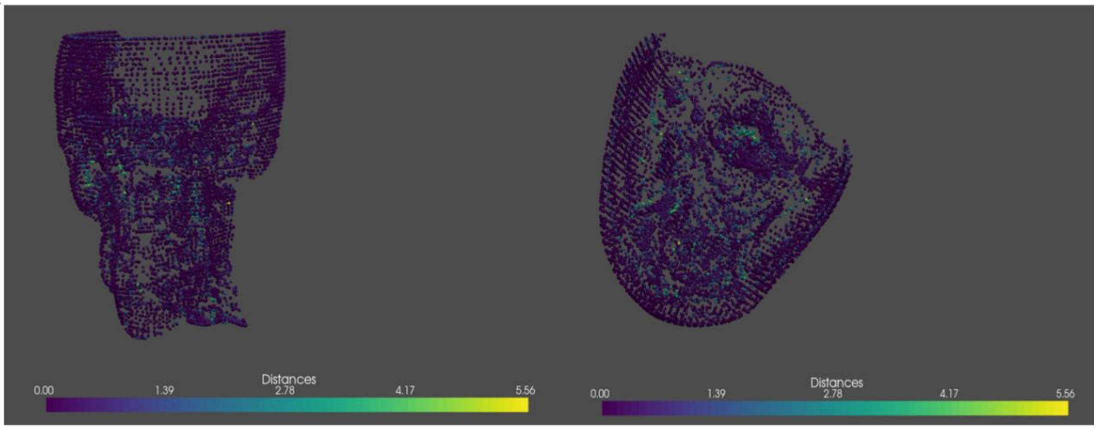


Figure B 7: *Heat map (A) and histogram (B) for the fourth targeted sampling experiment.*

A



B

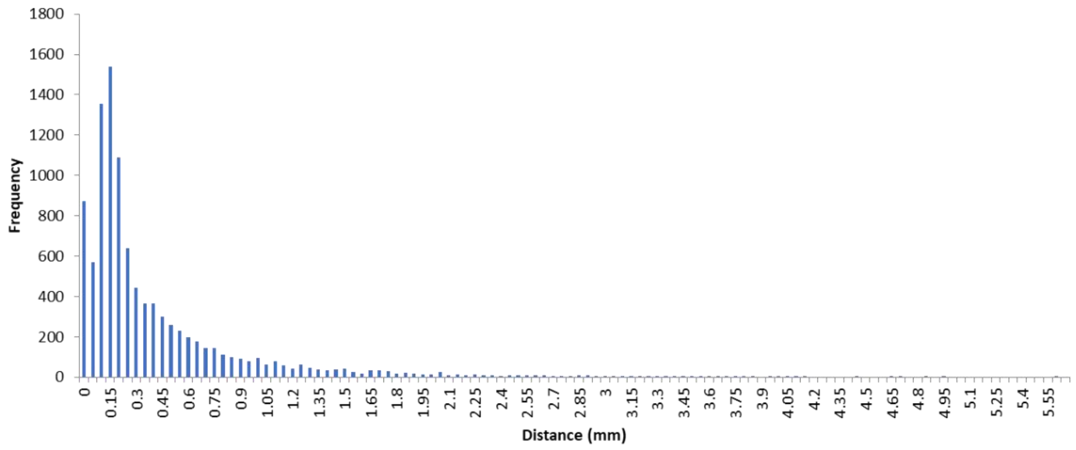
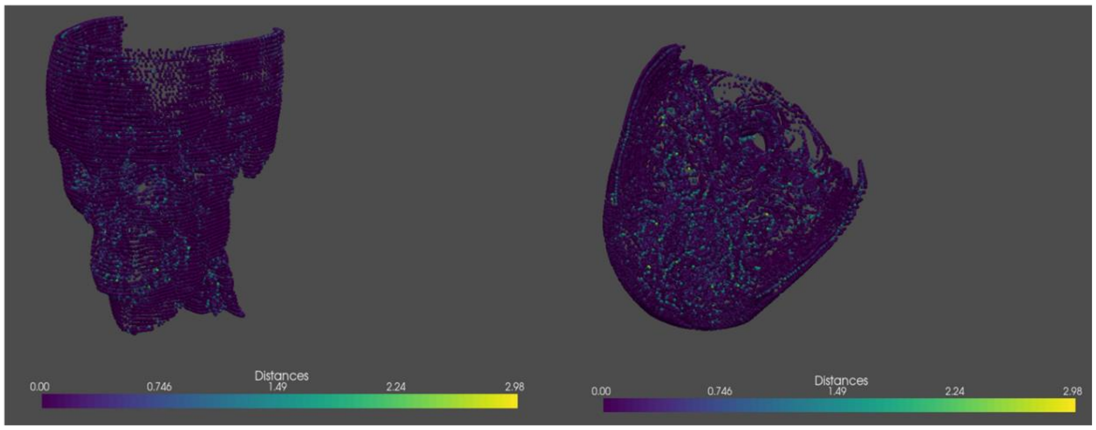


Figure B 8: *Heat map (A) and histogram (B) for the fifth targeted sampling experiment.*

A



B

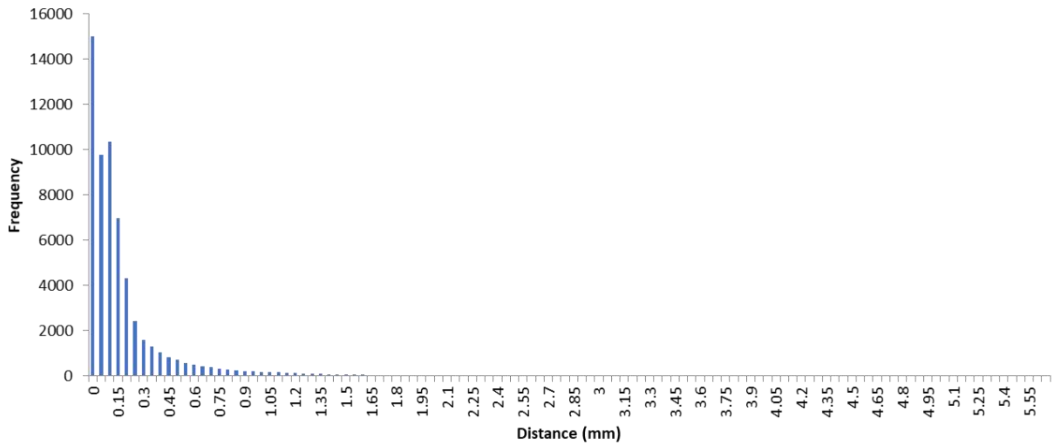
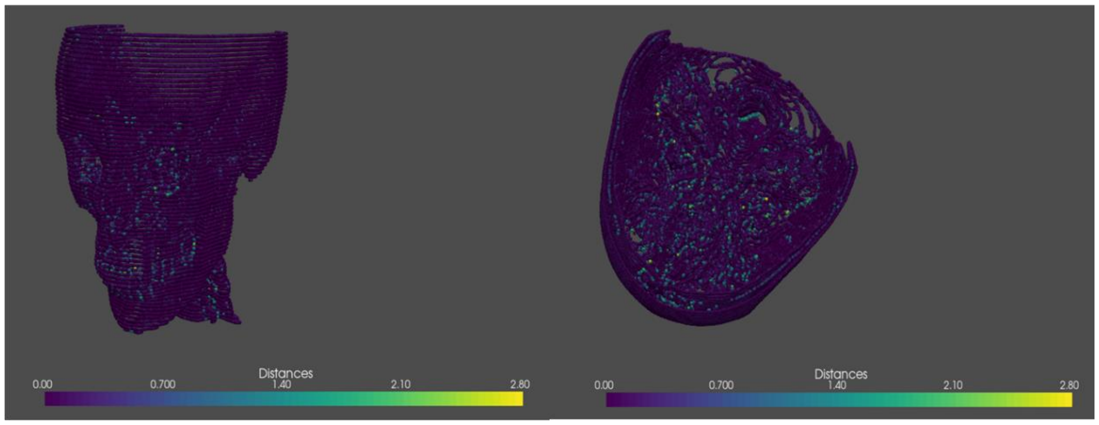


Figure B 9: *Heat map (A) and histogram (B) for the sixth targeted sampling experiment.*

A



B

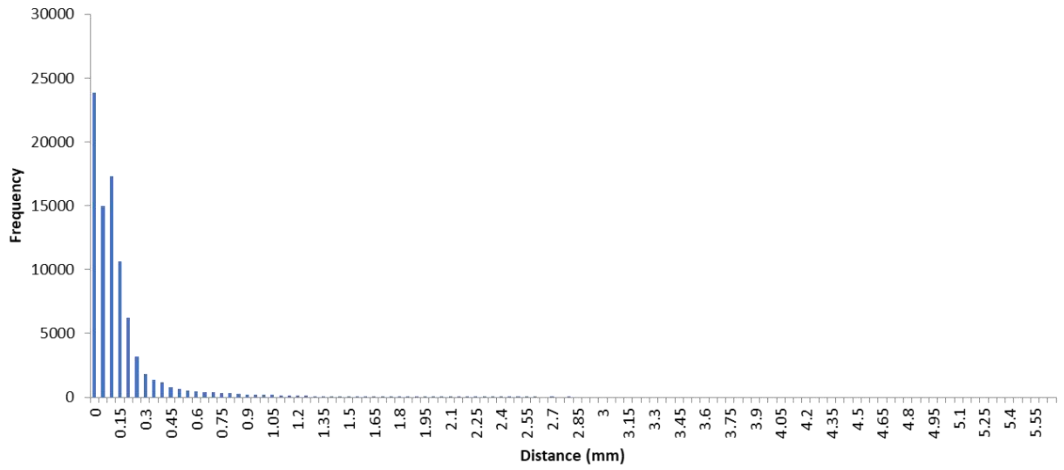
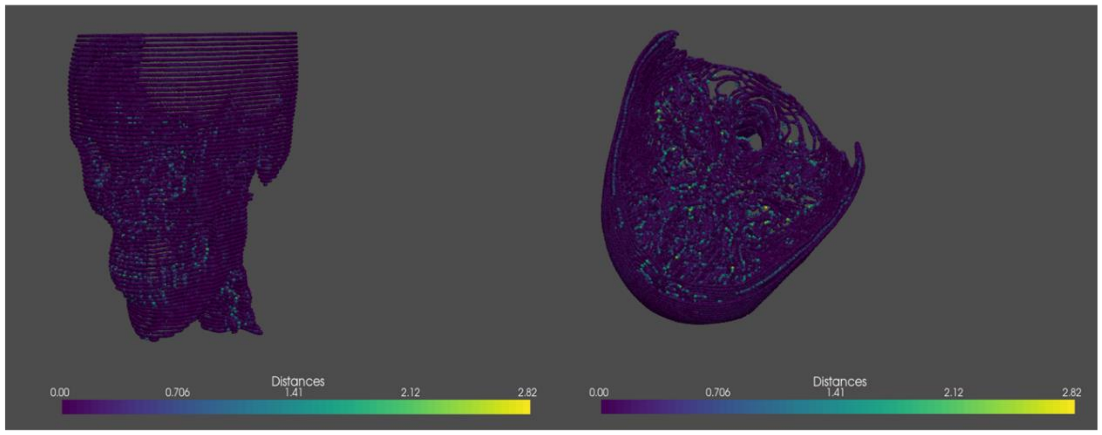


Figure B 10: *Heat map (A) and histogram (B) for the seventh targeted sampling experiment.*

A



B

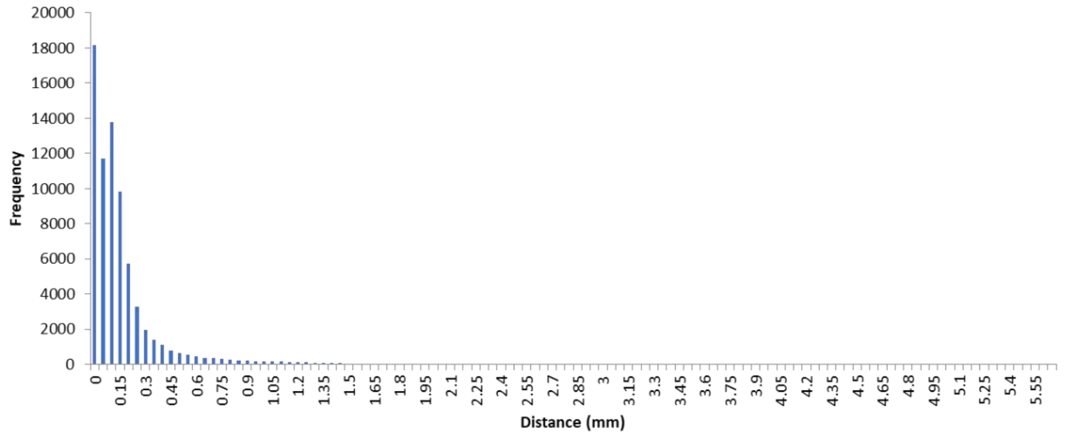


Figure B 11: *Heat map (A) and histogram (B) for the ninth targeted sampling experiment.*

Appendix C: CT Acquisition Parameters

This appendix contains information on the acquisition parameters of the CT images used throughout this project (Section 3.2).

Table C. 1: *CT Acquisition Parameters for CT images used throughout this project (as described in Section 3.2). The CT images of the lower limbs did not contain a tag for the spacing between the slices.*

	Skull1	**Skull2**	**Lower Limbs**
Scan Options	Axial	Helical	Helical
Slice Thickness	5.0	2.5	3.75
kVp	140	120	140
Spacing between Slices	20.062	0.625	NA
X-Ray Tube Current	150	220	70

Appendix D: Python Code

```
#This code will be used to create a point cloud representation
#of a stack of CT images
##Elyse Rier

#First we need to open every CT image in the stack
#Import functions open and plot DICOM images
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import pydicom
import numpy as np
import os
from skimage import feature
import cv2
import open3d as o3d
import pyvista as pv
import time
import scipy
import math
import decimal

#Open the DICOM files
#the code to open the files was modified from [1]
def open_CT(file_path): #this function takes the inputted file directory,
opens all CT images in the directory and converts them to HU and saves it as
a large array
    slices = [pydicom.dcmread(file_path+'/'+s) for s in
os.listdir(file_path)]#this creates a list containing all the dicom files in
the directory

    slices.sort(key = lambda sl: int(sl.InstanceNumber))#sort the created
list based on the Instance Number

    #create an array containing the slice locations for every slice
    locations = np.stack([sl.SliceLocation for sl in slices])

    #create a an array which is the stack of pixel arrays of the images
    images = np.stack([sl.pixel_array for sl in slices])

    #now convert the pixel array to HU
    slope = slices[0].RescaleSlope
    intercept = slices[0].RescaleIntercept

    images = slope * images.astype(np.float64)
    images += intercept


    #find the attributes needed to create the point cloud, ie slice
thickness, spacing between slices and the pixel spacing
    slice_thickness  = slices[0].SliceThickness
    #spacing_bw_slices = slices[0].SpacingBetweenSlices
    spacing_bw_rows = float(slices[0].PixelSpacing[0])
    spacing_bw_cols = float(slices[0].PixelSpacing[1])


    #return the desired variables
    return images,slice_thickness,spacing_bw_rows,spacing_bw_cols,locations
```

```
def thresh_edge_CT(images,threshold):
    #this function works to binarize the stack of CT images based on the
given threshold
    #this function also takes the binarized image and applies a canny edge
detector
    num_slices= images.shape[0]

    edges = images
    #loop through each image in the stack and apply the desired threshold and
apply the canny edge detector
    for i in range(num_slices):
        edges[i] = feature.canny(images[i]>threshold)#the canny edge
detection function taken from skimage


    #return the stack of images containing the edges
    return edges


def extract_PT_Cloud(edges,sl_thickness,row_spacing,col_spacing,locations):
    #this function takes a stack of edge images as inputs and returns the
point cloud obtained from these images

    dimensions = edges.shape
    count =1

    #loop through every pixel in each of the slices in the stack
    for sl_num in range(dimensions[0]):
        for rows in range(dimensions[1]):
            for cols in range(dimensions[2]):
                #creating the point cloud, the x dimension is the columns, y
dimension is rows and z is in the direction of the slices
                if edges[sl_num][rows][cols] == 1:
                    if count==1:
                        #if this is the first point you find, initialize the
array as the first point
                        point_cloud = np.array([float((float(cols) *
col_spacing) - (col_spacing/2)), float((float(rows) * row_spacing) -
(row_spacing/2)), float(locations[sl_num])])
                    else:
                        holder = np.array([float((float(cols) * col_spacing)
- (col_spacing/2)), float((float(rows) * row_spacing)- (row_spacing/2)),
float(locations[sl_num])])
                        point_cloud = np.vstack((point_cloud,holder))

                    count += 1


    return point_cloud
```

```
def down_sample_PT_Cloud(pt_cloud,locations,per):
    #The function downsamples the point cloud based on the percentage
provided
    #a pseudo uniform sample is created by sampling a percentage from 10
nearest neighbours for each slice
    num_points=0
    down_first = 0
    for i in range(0,len(locations)):#loop through each slice
        z_points = pt_cloud[np.where(pt_cloud[:,2]==locations[i])[0],:]#find
all the points at that z location
        first = 0
        while len(z_points)>0:#loop through the nearest neighbours
            if first==0:
                init_lenz = len(z_points)#if this is the first set of points
for that slice the length is the full length
                first = 1
            tree = scipy.spatial.KDTree(z_points)#create the KD tree
            if len(z_points)<10:
                dist,ind = tree.query(z_points,len(z_points))#if there are
less than 10 points find distances between all the neighbours
            else:
                dist,ind = tree.query(z_points,10)#if more than 10 points
find 10 nearest neighbours
            if ind.ndim == 1:#this means there is only one row ie one point
                vals =
np.rint(np.linspace(1,len(ind[:]),int((per/100)*len(ind[:]))))-1#determine as
uniformly spaced indices as possible rounded to the nearest integer
                vals = vals.astype(int)
                include = ind[vals]#thus are the points selected to be
included in the down sampled point cloud
                if down_first==0:#if first set of points, create new matrix
                    down_sampled = z_points[include,:]
                    z_points = np.delete(z_points,ind[:],axis=0)#delete the
points so they aren't checked again
                    down_first=1
                else:#if points are already in the downsampled stack points
                    down_sampled =
np.vstack((down_sampled,z_points[include,:]))
                    z_points = np.delete(z_points,ind[:],axis=0)#delete
points so they are not checked again
            else:
                vals =
np.rint(np.linspace(1,len(ind[0,:]),int((per/100)*len(ind[0,:]))))-1#if there
are more than 1 points find the indices to include
                vals = vals.astype(int)
                include = ind[0,vals]
                if down_first==0:#if first set of points to be added create
new matrix
                    down_sampled = z_points[include,:]
                    z_points = np.delete(z_points,ind[0,:],axis=0)#delete
points so not selected again
                    down_first=1
                else:#if points are already in downsampled then stack points
                    down_sampled =
np.vstack((down_sampled,z_points[include,:]))
                    z_points = np.delete(z_points,ind[0,:],axis=0)
    return down_sampled
```

```
def plot_PT_Cloud(pt_cloud):
    #this function takes the 3D point cloud represented as a numpy array and
displays a 3D scatterplot of the points
    #code adapted from [2]

    #extract x,y,z from point cloud as individual vectors
    x = pt_cloud[:,0]
    y = pt_cloud[:,1]
    z = pt_cloud[:,2]

    #create the figure
    disp = plt.figure()
    #define the axes
    axis = disp.add_subplot(111,projection = '3d')

    #create scatterplot
    axis.scatter(x,y,z,s=0.5,marker='.')

    #display the plot
    plt.show()

    return

def Create_Cylinder_Stack(num_slice,size,radius,thickness,spacing): #size
should be an even number, and the radius cannot be bigger than size/2
    #this function creates stack of images representing a hollow cylinder
with a given number of slices and a given radius and thickness
    #the num_slice represents the total slice, there will be five slices of
empty black images on both sides of te cylinder

    bl_im = np.zeros((size,size),np.float64)#creates blank black image
    empty_bl = bl_im

    img =
cv2.circle(bl_im,(int(size/2),int(size/2)),radius,(255,255,255),thickness)#dr
aw the circle of the image
    cylinder_stack = np.repeat(img[:,:,np.newaxis],num_slice,axis=2)
    cyl_loc = np.arange(0.0,(num_slice)*spacing,spacing)

    return cylinder_stack,cyl_loc

def Create_Line_Stack(num_slice,thickness,size,spacing):#this function will
draw two lines on an image and create a stack which repeats this image
    #for the desired size of the stack (num_slice), the line is centred based
on the size of the image, this will not have any blank images on either side
    bl_im= np.zeros((size,size),np.float64)
    img= cv2.line(bl_im,(int(1/2*size),(int(1/2*size)-
int(1/4*size))),(int(1/2*size),(int(1/2*size)+int(1/4*size))),(255,255,255),t
hickness)#draw the first line

    line_stack= np.repeat(img[:,:,np.newaxis],num_slice,axis=2)#repeat the
line image to make the stack

    #create a vector containing the locations of the 'slices'
    line_loc = np.arange(0.0,(num_slice)*spacing,spacing)

    return line_stack,line_loc
```

```
#Here we want to extract a point cloud from the created cylinder stack
simulating a dicom image
def Extract_Cylinder_PT_Cloud(cyl,spacing_r,spacing_c,locations):
    #This function extracts a point cloud from the created cylinder stack
    #The first thing the function does is create the edge images and then
call the function Extract_PT_Cloud
    num_slice = cyl.shape[2]
    edge = cyl

    for i in range(num_slice):
        edge[:,:,i] = feature.canny(cyl[:,:,i])#find the edges of the
cylinder
        #edge[:,:,i]= cyl[:,:,i]

    #once the edges have been found the function, you need to loop through
and get all of the points, using the same process as the extract_PT_Cloud
function
    edge_dim = edge.shape
    count=1
    for l in range(edge_dim[2]):#loop through each slice
        for m in range(edge_dim[0]):
            for n in range(edge_dim[1]):
                if edge[m,n,l] == True:#to get full point cloud and not edges
need to compare to 255.0 instead of True:

                    if count==1:
                       cyl_pt_cloud = np.array([float(float(n) * spacing_c),
float(float(m) * spacing_r), float(locations[l])])
                    else:
                        holder = np.array([float(float(n) * spacing_c),
float(float(m) * spacing_r), float(locations[l])])
                        cyl_pt_cloud = np.vstack((cyl_pt_cloud,holder))
                    count += 1



    return cyl_pt_cloud
```

```
def
sample_grid_norm(point_cloud,r_spacing,c_spacing,locations,r,c,zs,ang_interva
ls,per1,per2,per3,per4):
    #where ang_intervals is a list [ang1,ang2,ang3,ang4]
    #this function will take the point cloud, the row spacing and col spacing
(2d plane) as well as find the z-spacing from the locations
    #it will grid a grid of rectangular prisms starting at the min x,y,z
direction +/- the corresponding spacing with dimensions being 2x the spacing
in all 3 directions
    #within each grid the corresponding point clouds will be extracted and
point normals will be calculated
    #the angular similarity will be found and averaged for each point

    box =
[min(point_cloud[:,0]),max(point_cloud[:,0]),min(point_cloud[:,1]),max(point_
cloud[:,1]),min(point_cloud[:,2]),max(point_cloud[:,2])]#bounds of point
cloud
    #loop through all the slice locations and find the max number of decimal
points
    # how to find number of decimal points code adpated from [3]
    dec_list=np.zeros(locations.size)
    for i in range(0,locations.size):
        dec = decimal.Decimal(str(locations[i]))
        dec_pts = dec.as_tuple().exponent
        dec_list[i] = abs(dec_pts)
    round_pt = int(np.max(dec_list))
    #create a grid
    x_grid = np.arange(box[0]-
(c*c_spacing),box[1]+(c*c_spacing),(c*c_spacing))
    y_grid = np.arange(box[2]-
(r*r_spacing),box[3]+(r*r_spacing),(r*r_spacing))
    dec = decimal.Decimal(str(locations[1]))
    z_spacing = round((locations[1]-locations[0]),round_pt)
    z_grid = np.arange(box[4]-
(zs*abs(z_spacing)),box[5]+(zs*abs(z_spacing)),(zs*abs(z_spacing)))
    count=1#this will be used to see if a grid has been downsampled or not
yet
    count1=0
    count2=0
    count3=0
    count4=0
    pts_grid=0
    for z in range(0,len(z_grid)-1):
        for y in range(0,len(y_grid)-1):
            for x in range(0,len(x_grid)-1):
                #find the indices of the points in the point cloud that are
within the x,y,z range
                xi =
np.logical_and(point_cloud[:,0]<x_grid[x+1],np.greater_equal(point_cloud[:,0]
,x_grid[x]))
                yi =
np.logical_and(point_cloud[:,1]<y_grid[y+1],np.greater_equal(point_cloud[:,1]
,y_grid[y]))
                zi =
np.logical_and(point_cloud[:,2]<z_grid[z+1],np.greater_equal(point_cloud[:,2]
,z_grid[z]))
```

```
                combo = xi & yi & zi #combine the conditional statements to 
find the points in the desired ranges
                combo_true = np.where(combo==True)#find the indices of the 
elements in the combination that are true
                if combo_true[0].size!=0:
                    #if you found points within the range, calculate the 
normals of those points
                    new_cloud = point_cloud[combo_true[0],:]
                    first_new = 0
                    z_loc = np.arange(z_grid[z],z_grid[z+1],abs(z_spacing))
                    for slice_loc in 
np.arange(z_grid[z],z_grid[z+1],abs(z_spacing)):#for each slice in the voxel
                        indices = 
np.where(new_cloud[:,2]==round(slice_loc,round_pt))#this finds the indices of 
the points in the specific slice
                        cloud2 = new_cloud[indices[0],:]#this is the portion 
of the point for each individual slice
                        if cloud2.shape[0]!=0:
                            #create the point cloud in open 3d
                            pcd = o3d.geometry.PointCloud()
                            pcd.points = o3d.utility.Vector3dVector(cloud2)
                            #estimate normals of the new point cloud
                            o3d.geometry.PointCloud.estimate_normals(pcd)
                            norms = np.asarray(pcd.normals)
                            num = 0
                            sum_sim = 0
                            for i in range(0,len(norms)):#i is the index of 
the point we want to compare everything to
                                    for j in range(0,len(norms)):
                                        #check to make sure you are 
calculating the similarity between the same points
                                        if i != j:
                                            #calculate the cosine 
similarity between the two normals
                                            cos_sim = 
(np.vdot(norms[i,:],norms[j,:]))/(np.linalg.norm(norms[i,:])*np.linalg.norm(n
orms[j,:]))
                                            #find the angular 
similarity
                                            ang_sim = 1 - 
math.acos(round(cos_sim,6))/math.pi
                                            #add to the sum
                                            sum_sim = sum_sim + 
ang_sim
                                            num = num + 1
                            if num!=0:
                                avg_sim = sum_sim/num
                                #if similarity is ang1-ang2% percent keep 
per4%, ang2-ang3 keep per3%,ang3-ang4 keep per2% and ang4-1.0 keep per1%
                                if (avg_sim >= ang_intervals[0])&(avg_sim < 
ang_intervals[1]):
                                    new_points = 
down_sample_PT_Cloud(cloud2,z_loc,per4)#sample just those slice points using 
pseudo uniform sampling
                                    if first_new==0:
                                        down_new = new_points
                                        first_new=1
                                        else:
                                            down_new = 
np.vstack((down_new,new_points))
                                        count1+=1
                                        pts_grid+=new_cloud.shape[0]
                                    elif (avg_sim >= ang_intervals[1])&(avg_sim < 
ang_intervals[2]):
                                        new_points = 
down_sample_PT_Cloud(cloud2,z_loc,per3)#sample just those slice points using 
pseudo uniform sampling
                                        if first_new==0:
                                            down_new = new_points
                                            first_new=1
                                        else:
                                            down_new = 
np.vstack((down_new,new_points))
                                        count2+=1
                                        pts_grid+=new_cloud.shape[0]
                                    elif (avg_sim >= ang_intervals[2])&(avg_sim 
<ang_intervals[3]):
                                        new_points = 
down_sample_PT_Cloud(cloud2,z_loc,per2)#sample just those slice points using 
pseudo uniform sampling
                                        if first_new==0:
                                            down_new = new_points
                                            first_new=1
                                        else:
                                            down_new = 
np.vstack((down_new,new_points))
                                        count3+=1
                                        pts_grid+=new_cloud.shape[0]
                                    else:#this would be for between ang4 and 1.0
                                        new_points = 
down_sample_PT_Cloud(cloud2,z_loc,per1)#sample just those slice points using 
pseudo uniform sampling
                                        if first_new==0:
                                            down_new = new_points
                                            first_new=1
                                        else:
                                            down_new = 
np.vstack((down_new,new_points))
                                        count4+=1
                                        pts_grid+=new_cloud.shape[0]
                        if (count==1):#create the new outputted sampled point 
cloud
                            sampled_pcd = down_new
                            count += 1
                        else:
                            sampled_pcd = np.vstack((sampled_pcd,down_new))
    print('The number of grids with ang1 to ang2 is %d, ang2 to ang3 is %d, 
ang3 to ang4 is %d, and ang4 to 1 is %d.'%(count1,count2,count3,count4))
    print('avg number of points in each grid is: 
%f'%(pts_grid/(count1+count2+count3+count4)))
    return sampled_pcd
```

```
def create_mesh(point_cloud):
    #before the mesh is created, a nearest neighbour search is conducted to
calculate the average distance
    #the average distance is used for the alpha value in the delaunay_3d()
triangulation
    tree= scipy.spatial.KDTree(point_cloud)#create a KD tree from the point
cloud

    dist, ind= tree.query(point_cloud,100) #this returns the indices and the
distances of the number of nearest neighbours specified

    dist_new= dist[:,1:] #this removes the first column which is zeros
(contains the distance between the point and itself)

    averages= np.zeros((dist.shape[0],1))
    for i in range(dist.shape[0]):#loop through the distances and calculate
the average
        averages[i] = np.mean(dist[i])

    alph= np.mean(averages)#take alpha as the average of all the nearest
neighbour averages

    pcd= pv.PolyData(point_cloud)#initialize point cloud in pyvista
    mesh= pcd.delaunay_3d(alpha=alph)#run delaunay_3d

    return mesh,alph
```

```
def compare_meshes(mesh,ref_mesh):
    ##This function calculates the average distance between the midpoint of
each edge in mesh
    ##and the nearest 10 points in the ref_mesh
    #extract the surface of mesh
    surf = mesh.extract_surface().triangulate().clean()
    surf_pts = surf.points
    ref_surf = ref_mesh.extract_surface().triangulate().clean()
    #extract the edges of the suraface
    edges= surf.extract_all_edges()
    #extract the lines of the edges, this is saved as first the number of
points in the line followed by the indices of the line
    lines = edges.lines
    #edges_points = edges.points
    ref_pts = ref_surf.points
    index = 0
    first=0
    count = 0
    while index<len(lines):#loop through the lines of the edges
        inc = lines[index]#number of points in the corresponding line
        if (surf_pts[lines[index+1],2]==surf_pts[lines[index+inc],2]):#want
both points on the line to be on the same z-level, so that you only compare
for each z-level
            #now do the comparison of the midpoints of the edges and the
original point cloud
            start = surf_pts[lines[index+1]]
            end = surf_pts[lines[index+inc]]
            #find the equation of the line between start and end
            #find the slope
            A = end[1]-start[1]
            B = end[0]-start[0]
            if B!=0:
                m = A/B
                #solve for the intercept of the line using start point
                b = start[1]- m*start[0]
                C = B*b
            #A, B and C are the coefficients of the line in the standard form
of the form -Ax+By-C=0
            mid = np.array([(start[0]+end[0])/2,(start[1]+end[1])/2])
            #now find all of the points on the same z-level
            z = surf_pts[lines[index+1],2]
            z_vals = ref_pts[np.where((ref_pts[:,2]==z))[0]]
            #find the length of the line
            len_line = math.sqrt((end[0]-start[0])**2+(end[1]-start[1])**2)
            #find the 100 nearest neighbours to the midpoint, loop through
these neighbours to see which are less than half the length of the line away
            tot_pts = np.vstack((z_vals[:,:2],mid))
            tree = scipy.spatial.KDTree(tot_pts)
            dist, ind = tree.query(tot_pts,11)
            dist_new = dist[len(dist)-1,1:]
            ind_new = ind[len(ind)-1,1:]
            #loop through all the z points (ie points with the same z value
and calculate the distance between the midpoint and each point
            for m in range(0,len(dist_new)):
                #if the distance to the mid point of the line is less than
half the length of the line then calculate the distance of that point to the
line
```

```
                if (dist_new[m] <= len_line/2):
                    if B!=0:
                        dist_val = abs(-
A*tot_pts[ind_new[m],0]+B*tot_pts[ind_new[m],1]-
C)/math.sqrt(A**2+B**2)#calculate distance from point to line
                    else:
                        dist_val = abs(tot_pts[ind_new[m],0]-end[0])#if the
line is a vertical line the distance is just the difference in the x values
                    if m==0:#if it is the first time save as new arrays
                        edge_dist = np.array([dist_val])#distance to that
edge
                        pt1 =
np.where((surf_pts[:,0]==start[0])&(surf_pts[:,1]==start[1])&(surf_pts[:,2]==
z))[0][0]##location in original points of start point
                        pt2 =
np.where((surf_pts[:,0]==end[0])&(surf_pts[:,1]==end[1])&(surf_pts[:,2]==z))[
0][0]#location in original points of end point
                        pt3 =
np.where((ref_pts[:,0]==tot_pts[ind_new[m],0])&(ref_pts[:,1]==tot_pts[ind_new
[m],1])&(ref_pts[:,2]==z))[0][0]#location in ref points that is the compared
point
                        dist_ind = np.array([pt1,pt2,pt3])
                    else:#if it is not the first set of points stack points
                        edge_dist = np.vstack((edge_dist,dist_val))
                        pt1 =
np.where((surf_pts[:,0]==start[0])&(surf_pts[:,1]==start[1])&(surf_pts[:,2]==
z))[0][0]
                        pt2 =
np.where((surf_pts[:,0]==end[0])&(surf_pts[:,1]==end[1])&(surf_pts[:,2]==z))[
0][0]
                        pt3 =
np.where((ref_pts[:,0]==tot_pts[ind_new[m],0])&(ref_pts[:,1]==tot_pts[ind_new
[m],1])&(ref_pts[:,2]==z))[0][0]
                        hold = np.array([pt1,pt2,pt3])
                        dist_ind = np.vstack((dist_ind,hold))

            #after the distances are found for each edge, create a matriz for
all the edge distances, a matrix for the indices (start pt, end pt, and ref
pt) for each edge distance
            if first==0:
                if edge_dist.shape[0]!=0:
                    first = 1
                    distances = edge_dist
                    dist_indices = dist_ind

            else:
                if edge_dist.shape[0]!=0:
                    distances = np.vstack((distances,edge_dist))
                    dist_indices = np.vstack((dist_indices,dist_ind))
        index = index+inc+1

    return surf_pts,ref_pts,distances,dist_indices#return the distances, the
indices, and sets of points for the surface and the reference surface
```

```
##The following functions were created to determine the execution time of the
different functions defined above

def
time_to_grid_samp(pcd,row_spacing,col_spacing,sl_locations,r,c,z,ang_interval
s,per1,per2,per3,per4):#determine the time to grid sample
    tic= time.perf_counter()
    new_surf_points =
sample_grid_norm(pcd,row_spacing,col_spacing,sl_locations,r,c,z,ang_intervals
,per1,per2,per3,per4)
    toc= time.perf_counter()
    exec_time = toc-tic
    print(exec_time)
    return new_surf_points,exec_time

def time_to_rand_samp(pcd,locs,per):
    tic= time.perf_counter()
    new_surf_points = down_sample_PT_Cloud(pcd,locs,per)
    toc= time.perf_counter()
    exec_time = toc-tic
    print(exec_time)
    return new_surf_points,exec_time

def time_to_mesh(pcd):
    tic= time.perf_counter()
    mesh,alpha= create_mesh(pcd)
    toc= time.perf_counter()
    exec_time = toc-tic
    print(exec_time)
    return mesh,exec_time

def save_mesh_stl(mesh,name):
    surf=mesh.extract_surface().clean()
    surf.save(name)
    return
def test_rand_samp(pcd,per,name):
    samp,samp_time = time_to_rand_samp(pcd,per)
    print(samp.shape)
    mesh,mesh_time = time_to_mesh(samp)
    save_mesh_stl(mesh,name)
    return samp,samp_time,mesh,mesh_time

def
test_grid_samp(pcd,row_spacing,col_spacing,sl_locations,r,c,z,ang_intervals,p
er1,per2,per3,per4,name):
    samp,samp_time =
time_to_grid_samp(pcd,row_spacing,col_spacing,sl_locations,r,c,z,ang_interval
s,per1,per2,per3,per4)
    print(samp.shape)
    mesh,mesh_time = time_to_mesh(samp)
    save_mesh_stl(mesh,name)
    return samp,samp_time,mesh,mesh_time
```

```
def main():#main function runs when the script is run, this is set for the
skull CT images so the threshold was set to 350
    #Ask user for file directory containing CT images
    file_direc = input("Please Enter the File Directory Containing Desired CT
Images: ")

    #open the CT files and extract the necessary parameters
    image_stack, sl_thickness, row_spacing, col_spacing , sl_locations =
open_CT(file_direc)

    #apply the edge detection and threshold the CT images
    edge_image = thresh_edge_CT(image_stack,600)

    #extract the surface points
    surface_points =
extract_PT_Cloud(edge_image,sl_thickness,row_spacing,col_spacing,sl_locations
)

    #create mesh
    mesh,alpha = create_mesh(surface_points)


if __name__== "__main__":
    main()


###References:
##[1] F. Heng, "A comprehensive guide to visualizing and analyzing DICOM
images in Python," Medium, 31-Jul-2019. [Online]. Available:
https://hengloose.medium.com/a-comprehensive-starter-guide-to-visualizing-
and-analyzing-dicom-images-in-python-7a8430fcb7ed. [Accessed: Jun-2020].
##[2] Geo-code, "3D plotting with matplotlib," 3D plotting with matplotlib -
Geo-code - My scratchpad for geo-related coding and research. [Online].
Available:
http://chris35wills.github.io/courses/PythonPackages_matplotlib/matplotlib_3d
/. [Accessed: Jun-2020].
##[3] "Easy way of finding decimal places," Stack Overflow, 01-Jun-2011.
[Online]. Available: https://stackoverflow.com/questions/6189956/easy-way-of-
finding-decimal-places. [Accessed: Jul-2021].
```

Appendix E: Blender Add-on code

```
bl_info = {
    "name": "Segment CT Model",
    "author": "Elyse Rier",
    "version": (1, 0),
    "blender": (2, 91, 0),
    "location": "View3D > N",
    "description": "Segments 3D model of CT data",
    "warning": "",
    "doc_url": "",
    "category": "",
}

import bpy
import os
import bmesh
import numpy as np
import math
from bpy_extras.io_utils import ImportHelper,ExportHelper
import pydicom
from skimage import feature
import pyvista as pv
import scipy

xval= 0.0
yval= 0.0
zval= 0.0
selected_points= []
```

```
#define the function to open the CT images and extract the point cloud
def Create_Mesh(direc_filename,threshold,output_direc,output_name):
    #the first portion of this function opens the CT images
    #images opened based on code from [1]
    slices = [pydicom.dcmread(direc_filename+'/'+s) for s in
os.listdir(direc_filename)]#this creates a list containing all the dicom
files in the directory
    slices.sort(key = lambda sl: int(sl.InstanceNumber))#sort the created
list based on the Instance Number

    #create an array containing the slice locations for every slice
    locations = np.stack([sl.SliceLocation for sl in slices])

    #create a an array which is the stack of pixel arrays of the images
    images = np.stack([sl.pixel_array for sl in slices])

    #now convert the pixel array to HU
    slope = slices[0].RescaleSlope
    intercept = slices[0].RescaleIntercept

    images = slope * images.astype(np.float64)
    images += intercept


    #find the attributes needed to create the point cloud, ie slice
thickness, spacing between slices and the pixel spacing
    slice_thickness  = slices[0].SliceThickness
    row_spacing = float(slices[0].PixelSpacing[0])
    col_spacing = float(slices[0].PixelSpacing[1])

    #now that the CT images have be opened and converted to numpy arrays they
need to be thresholded
    #and have the edge detection applied to them
    #this function works to binarize the stack of CT images based on the
given threshold
    #this function also takes the binarized image and applies a canny edge
detector
    num_slices= images.shape[0]

    edges = images
    #loop through each image in the stack and apply the desired threshold and
apply the canny edge detector
    for i in range(num_slices):
        edges[i] = feature.canny(images[i]>threshold)#the canny edge
detection function taken from skimage

    #Since the images are now binarized and the edges are isolated the point
cloud can be extracted
    dimensions = edges.shape
    count =1
    for sl_num in range(dimensions[0]):
        for rows in range(dimensions[1]):
            for cols in range(dimensions[2]):
                #creating the point cloud, the x dimension is the columns, y
dimension is rows and z is in the direction of the slices
                if edges[sl_num][rows][cols] == 1:
                    if count==1:
                        #if this is the first point you find, initialize the
array as the first point
                        point_cloud = np.array([float((float(cols) *
col_spacing) - (col_spacing/2)), float((float(rows) * row_spacing) -
(row_spacing/2)), float(locations[sl_num])])
                    else:
                        holder = np.array([float((float(cols) * col_spacing)
- (col_spacing/2)), float((float(rows) * row_spacing)- (row_spacing/2)),
float(locations[sl_num])])
                        point_cloud = np.vstack((point_cloud,holder))

                    count += 1

    #now that the point cloud has been created we want to use pyvista to
create the mesh
    #before the mesh is created, a nearest neighbour search is conducted to
calculate the average distance
    #the average distance is used for the alpha value in the delaunay_3d()
triangulation
    tree= scipy.spatial.KDTree(point_cloud)#create a KD tree from the point
cloud

    dist, ind= tree.query(point_cloud,100) #this returns the indices and the
distances of the number of nearest neighbours specified

    dist_new= dist[:,1:] #this removes the first column which is zeros
(contains the distance between the point and itself)

    averages= np.zeros((dist.shape[0],1))
    for i in range(dist.shape[0]):#loop through the distances and calculate
the average
        averages[i] = np.mean(dist[i])

    alph= np.mean(averages)#take alpha as the average of all the nearest
neighbour averages

    pcd= pv.PolyData(point_cloud)#initialize point cloud in pyvista
    mesh= pcd.delaunay_3d(alpha=alph)#run delaunay_3d

    #now save the mesh as an stl in the desired place
    surf = mesh.extract_surface().triangulate().clean()
    name = output_direc+output_name+'.stl'
    surf.save(name)
    bpy.ops.import_mesh.stl(filepath=name)
```

```
#define a property group
class RefProperties(bpy.types.PropertyGroup):
    x_val : bpy.props.FloatProperty()
    y_val : bpy.props.FloatProperty()
    z_val : bpy.props.FloatProperty()
    file_direc : bpy.props.StringProperty()
    out_direc : bpy.props.StringProperty()
    out_name : bpy.props.StringProperty(name= 'Mesh Filename')
    new_mesh_name : bpy.props.StringProperty(name= 'Segmented Mesh Filename')
    thresh : bpy.props.IntProperty(name='HU Threshold',default=0,min=-
1024,max=3071)
    thickness : bpy.props.FloatProperty(name = 'Thickness of Cut
(mm)',default = 5.0, min=0.0)

class SelectInputDirectory(bpy.types.Operator,ImportHelper):
    """This will open the file browser so the directory containing
    containing the CT images will be saved
    """
    bl_idname = "ctfile.direc"
    bl_label = "Select File Directory of CT Images"

    def execute(self,context):
        defpath = self.properties.filepath
        if(not(os.path.isdir(defpath))):
            mes = "Choose a directory not a file\n" + defpath
            self.report({'ERROR'},mes)
            return {'CANCELLED'}
        else:
            scene = context.scene
            myprop = scene.ref_properties
            myprop.file_direc = self.properties.filepath
            self.report({'INFO'},myprop.file_direc)
            return {'FINISHED'}

class SelectOutputLoc(bpy.types.Operator,ImportHelper):
    """This will open the file browser so that the folder in which the STL
    files will be saved can be chosen
    """
    bl_idname = "meshfile.direc"
    bl_label = "Select File Location to Save Mesh"

    def execute(self,context):
        defpath = self.properties.filepath
        if(not(os.path.isdir(defpath))):
            mes = "Choose a directory not a file\n" + defpath
            self.report({'ERROR'},mes)
            return {'CANCELLED'}
        else:
            scene = context.scene
            myprop = scene.ref_properties
            myprop.out_direc = self.properties.filepath
            self.report({'INFO'},myprop.out_direc)
            return {'FINISHED'}
```

```
class CreateMeshOperator(bpy.types.Operator):
    """This Operation will apply the thresholding, edge detection and extract 
a
       3D point cloud from the imported stack of CT images which is then used 
to render a mesh"""
    bl_idname = "ctpt.create"
    bl_label = "Render Surface Mesh"
    bl_options = {'REGISTER','UNDO'}


    def execute(self,context):
        scene = context.scene
        myprop = scene.ref_properties
        if((not(os.path.isdir(myprop.out_direc)))and(not(myprop.out_name))):
           message = "Make sure a file directory and mesh name have been 
chosen before extracting point cloud \n"
           self.report({'ERROR'},message)
           return{'CANCELLED'}
        else:
            point_cloud = 
Create_Mesh(myprop.file_direc,myprop.thresh,myprop.out_direc,myprop.out_name)
            mesh = bpy.ops.object
            mesh.origin_set(type='GEOMETRY_ORIGIN')
            return{'FINISHED'}


class RoiSelectionOperator(bpy.types.Operator):
    """ This Operator takes the selection of desired points, extracts the 
points
        closest to the reference points and saves the vertices in a csv file
    """
    bl_idname = "ctseg.roi_selection"
    bl_label = "Select Region of Interest"

    @classmethod
    def poll(cls,context):
        #poll checks if there is an active object, if not the operator cannot 
run
        return context.active_object != None

    def execute(self,context):
        #here we get the coordinates for the selected region of points
        #compare the distance between those points and the reference point
        #and the point and the centroid points on the desired side are 
selected
        bpy.ops.object.mode_set(mode='OBJECT')
        #save the selected vertices as a numpy array
        #points accessed based on code from [2]
        selverts = np.array([v for v in 
bpy.context.active_object.data.vertices if v.select])
        #go through and save the coordinates of all the selected points in a 
numpy array
        surf_pts = np.array([v.co for v in selverts])
        len_sel = len(surf_pts)
        #now that the surface coordinates have been selected we have to 
select all the vertices and save them
```

```
        bpy.ops.object.mode_set(mode='EDIT')#switch back to edit mode to
select the point
        bpy.ops.mesh.select_all(action='SELECT') #select all vertices
        bpy.ops.object.mode_set(mode='OBJECT')
        full = np.array([p for p in bpy.context.active_object.data.vertices
if p.select])
        full_pts = np.array([p.co for p in full])#save the coordinates of all
the vertices
        scene = context.scene
        myprop = scene.ref_properties

        ref_coords = np.array([myprop.x_val,myprop.y_val,myprop.z_val])
        #get inputted thickness
        thick = myprop.thickness
        #find centroid of selected points
        centroid = np.mean(surf_pts,axis=0)
        #find all the values of z in surf_pts
        z_vals = np.unique(surf_pts[:,2],axis=0)
        #find the difference between the z value of the centroid and every z
level
        z_cent = z_vals - centroid[2]
        #find index in z_vals that has the minimum absolute difference
between the centroid
        ind_z = np.where(abs(z_cent[:])== min(abs(z_cent[:])))[0][0]
        #find all the points of surf_pts that have the z value closest to the
centroid
        surf_z1 = surf_pts[np.where(surf_pts[:,2]==z_vals[ind_z])[0],:]
        max_y_val =
surf_z1[np.where(surf_z1[:,1]==max(surf_z1[:,1]))[0][0],:]#find point with
max y
        veca = max_y_val - centroid #vector between max value of y and the
centroid
        if len(z_vals)==1:
            #if there is only one z value use the max x to find the vecb
instead
            max_x_vals =
surf_pts[np.where(surf_pts[:,0]==max(surf_pts[:,0]))[0],:]
            #loop through max vals and take the first point that is not equal
to the y point used
            for i in range(0,len(max_x_vals)):
                if
((max_x_vals[i,0]!=max_y_val[0])&(max_x_vals[i,1]!=max_y_val[1])):
                    #use this point
                    vecb= max_x_vals[i,:]- centroid
                    break
        else:
            #find max z val in surf_pts and use it to create the second
vector
            max_z_val=
surf_pts[np.where(surf_pts[:,2]==max(surf_pts[:,2]))[0],:]
            for i in range(0,len(max_z_val)):
                if
((max_z_val[i,0]!=max_y_val[0])&(max_z_val[i,1]!=max_y_val[1])):
                    vecb = max_z_val[i,:]-centroid
                    break
        #find the cross product between the two vectors
        cross = np.cross(veca,vecb)
        #if the dot product between the cross product and the vector from
origin to centre is positive, flip direction of cross
        dot = np.dot(cross,centroid)
        if dot>0:
            cross = -1*cross
        unit_cross = cross/np.linalg.norm(cross)
        #find direction (x,y,x) with greatest change in cross, this is main
direction of normal
        indm = np.where(abs(unit_cross[:])==max(abs(unit_cross[:])))[0][0]
        #Find all the points that are within the provided thickness of the
surface points-not including the surface points
        surr=0
        for i in range(0,len(surf_pts)):
            for j in range(0,len(full_pts)):
                dist = abs(surf_pts[i,0]-full_pts[j,0])+abs(surf_pts[i,1]-
full_pts[j,1])+abs(surf_pts[i,2]-full_pts[j,2])
                if (dist<=thick):
                    if len(z_vals)==1:
                        #if one z value just need to filter it so that the
points are not on the same z as the surf points
                        if full_pts[j,2]!= z_vals[0]:
                            if surr==0:
                                surround_pts = full_pts[j,:]
                                surr=1
                            else:
                                surround_pts =
np.vstack((surround_pts,full_pts[j,:]))
                    else:

                        #if it is less than the distance compare it to the
surf_pt and its projected point
                        #if it is closer to the projected keep it
                        new_surf = surf_pts[i,:] + thick*unit_cross
                        dist_new = abs(new_surf[0]-
full_pts[j,0])+abs(new_surf[1]-full_pts[j,1])+abs(new_surf[2]-full_pts[j,2])
                        if dist_new<dist:
                            if surr==0:
                                surround_pts = full_pts[j,:]
                                surr=1
                            else:
                                surround_pts =
np.vstack((surround_pts,full_pts[j,:]))
        surround_pts = np.unique(surround_pts,axis=0)#this removes duplicates
of the points
        #find all the values of z in surf_pts
        #find the max z value of the inner surface by adding thick in the
direction of the normal from the max of the x of surf_pts
        maxz_inner = max(surf_pts[:,2]) + thick*unit_cross[2]
        #remove points from the surround_pts that are greater than this z
value
        above_innmaxz = np.where(surround_pts[:,2]>maxz_inner)
        surround_pts = np.delete(surround_pts,above_innmaxz[0],axis=0)
        #find the z-values corresponding to the inner layer
        #loop through the z_vals and find the value of z that is closest to
the maximum of the inner but still less than it
```

```
        for i in range(0,len(z_vals)):
            if i==0:
                    min_diff = z_vals[i]-maxz_inner
                    i_min = i
            elif (abs(z_vals[i]-maxz_inner)<abs(min_diff)) & ((z_vals[i]-
maxz_inner)<=0):#want the difference to be negative so that the next z value
is always less than the max of the inner
                    min_diff = z_vals[i]-maxz_inner
                    i_min = i
        #create the list of z_vals for the inner portion, starting from the
closest to the max and moving down by the spacing in z
        z_full = np.unique(full_pts[:,2],axis=0)
        z_spac = z_full[1]-z_full[0]
        if len(z_vals)==1:
            #if there is just one z value then the levels of z go from the
initial point to a certain thickness away
            #if one z value then the normal is fully in the z direction so
the unit cross will either be positive or negative one telling which
direction to change z
            max_z = z_vals[0] + unit_cross[2]*thick
            locs = np.where((z_full[:]>= min(z_vals[0],max_z))&(z_full[:]<=
max(z_vals[0],max_z)))[0]
            z_vals_in = z_full[locs]
        else:
            loc = np.where(z_full[:]==z_vals[i_min])[0][0]
            z_vals_in = z_full[(loc-len(z_vals)+1):(loc+1)]
        #based on the direction of the normal with the largest magnitude
(this would be the same as posno)
        #filter the surrounding points
        if indm ==0: #this is the x direction so for every z check the y
dimension
            check = 0
            for k in range(0,len(z_vals_in)):
                surf_z = surf_pts[np.where(surf_pts[:,2]==z_vals[len(z_vals)-
k-1])[0],:]
                sel_z =
surround_pts[np.where(surround_pts[:,2]==z_vals_in[k])[0],:]
                if surf_z.shape[0]==0:
                    pass
                elif surf_z.shape[0]==1:
                    new_surf = surf_z+thick*unit_cross#project point along
normal
                    #find points that lie between the points in the x
direction
                    pts = np.where((sel_z[:,1]<=(new_surf[0,1]+(z_vals[1]-
z_vals[0])))&(sel_z[:,1]>=(new_surf[0,1]-(z_vals[1]-z_vals[0]))))[0]
                    if k==0:
                        new_sel = sel_z[pts,:]
                    else:
                        new_sel = np.vstack((new_sel,sel_z[pts,:]))
                elif surf_z.shape[0]==2:
                    new_surf = surf_z+thick*unit_cross
                    pts =
np.where((sel_z[:,1]<=(max(new_surf[0,1],new_surf[1,1])))&(sel_z[:,1]>=(min(n
ew_surf[0,1],new_surf[1,1]))))[0]
                    if k==0:
                        new_sel = sel_z[pts,:]
                    else:
                        new_sel = np.vstack((new_sel,sel_z[pts,:]))
                 else:
                     #find the two nearest neighbours for each value in surf_z
                     tree = scipy.spatial.KDTree(surf_z)
                     dist,ind = tree.query(surf_z,3)
                     ind_new = ind[:,1:]
                     #now loop through every surf_z and add the points between
the x values of the two nearest neighbours
                     new_surf = surf_z + thick*unit_cross
                     for h in range(0,len(surf_z)):
                         #for each point find the points that fall between the
two nearest neighbours
                         pts =
np.where(((sel_z[:,1]<=(max(new_surf[h,1],new_surf[ind_new[h,0],1])))&(sel_z[
:,1]>=(min(new_surf[h,1],new_surf[ind_new[h,0],1]))))|((sel_z[:,1]<=(max(new_
surf[h,1],new_surf[ind_new[h,1],1])))&(sel_z[:,1]>=(min(new_surf[h,1],new_sur
f[ind_new[h,1],1])))))[0]
                         if k==0 & check==0:
                             new_sel = sel_z[pts,:]
                             check=1
                         else:
                             new_sel = np.vstack((new_sel,sel_z[pts,:]))
         elif indm==1:#filter based on the x value for each value of z
             check=0
             for k in range(0,len(z_vals_in)):
                 surf_z = surf_pts[np.where(surf_pts[:,2]==z_vals[len(z_vals)-
k-1])[0],:]
                 sel_z =
surround_pts[np.where(surround_pts[:,2]==z_vals_in[k])[0],:]
                 if surf_z.shape[0]==0:
                     pass
                 elif surf_z.shape[0]==1:
                     new_surf = surf_z+thick*unit_cross#project point along
normal
                     #find points that lie between the points in the x
direction
                     pts = np.where((sel_z[:,0]<=(new_surf[0,0]+(z_vals[1]-
z_vals[0])))&(sel_z[:,0]>=(new_surf[0,0]-(z_vals[1]-z_vals[0]))))[0]
                     if k==0:
                         new_sel = sel_z[pts,:]
                     else:
                         new_sel = np.vstack((new_sel,sel_z[pts,:]))
                 elif surf_z.shape[0]==2:
                     new_surf = surf_z+thick*unit_cross
                     pts =
np.where((sel_z[:,0]<=(max(new_surf[0,0],new_surf[1,0])))&(sel_z[:,0]>=(min(n
ew_surf[0,0],new_surf[1,0]))))[0]
                     if k==0:
                         new_sel = sel_z[pts,:]
                     else:
                         new_sel = np.vstack((new_sel,sel_z[pts,:]))
                 else:
                     #find the two nearest neighbours for each value in surf_z
                     tree = scipy.spatial.KDTree(surf_z)
                     dist,ind = tree.query(surf_z,3)
                     ind_new = ind[:,1:]
                    #now loop through every surf_z and add the points between
the x values of the two nearest neighbours
                    new_surf = surf_z + thick*unit_cross
                    for h in range(0,len(surf_z)):
                        #for each point find the points that fall between the
two nearest neighbours
                        pts =
np.where(((sel_z[:,0]<=(max(new_surf[h,0],new_surf[ind_new[h,0],0])))&(sel_z[
:,0]>=(min(new_surf[h,0],new_surf[ind_new[h,0],0]))))|((sel_z[:,0]<=(max(new_
surf[h,1],new_surf[ind_new[h,1],0])))&(sel_z[:,0]>=(min(new_surf[h,1],new_sur
f[ind_new[h,1],0])))))[0]
                        if k==0 & check==0:
                            new_sel = sel_z[pts,:]
                            check=1
                        else:
                            new_sel = np.vstack((new_sel,sel_z[pts,:]))
        else: #if the normal changes the most in the z-direction for each z,
for each value of x find
            check=0
            for k in range(0,len(z_vals_in)):
                #find the values at this z in the surf points
                if len(z_vals)==1:
                    #if just one z value surf_z is surf_pts
                    surf_z= surf_pts
                else:
                    surf_z =
surf_pts[np.where(surf_pts[:,2]==z_vals[len(z_vals)-k-1])[0],:]
                #within that portion of surf_pts at that z find all the
unique values of x
                x_vals = np.unique(surf_pts[:,0],axis=0)
                sel_z =
surround_pts[np.where(surround_pts[:,2]==z_vals_in[k])[0],:]
                for h in range(0,len(x_vals)):#for each x value, find max and
min value of y and project it along the normal
                    #find all points in surf_z with that particular x value
                    surf_x = surf_z[np.where(surf_z[:,0]==x_vals[h])[0],:]
                    print(surf_x.shape)
                    if surf_x.shape[0]==0:
                        #if it is empty do nothing
                        pass
                    elif surf_x.shape[0]==1:
                        new_surf = surf_x+thick*unit_cross
                        pts =
np.where((sel_z[:,1]<=(new_surf[0,1]+(z_spac)))&(sel_z[:,1]>=(new_surf[0,1]-
(z_spac))))[0]
                        if k==0:
                            new_sel = sel_z[pts,:]
                        else:
                            new_sel = np.vstack((new_sel,sel_z[pts,:]))
                    elif surf_x.shape[0]==2:
                        new_surf = surf_x+thick*unit_cross
                        pts =
np.where((sel_z[:,1]<=(max(new_surf[0,1],new_surf[1,1])))&(sel_z[:,1]>=(min(n
ew_surf[0,1],new_surf[1,1]))))[0]
                        if k==0:
                            new_sel = sel_z[pts,:]
                        else:
```

```
                        #find the two nearest neighbours for each value in
surf_z
                        tree = scipy.spatial.KDTree(surf_x)
                        dist,ind = tree.query(surf_x,3)
                        ind_new = ind[:,1:]
                        #now loop through every surf_z and add the points
between the x values of the two nearest neighbours
                        new_surf = surf_x + thick*unit_cross
                        for h in range(0,len(surf_x)):
                            #for each point find the points that fall between
the two nearest neighbours
                            pts =
np.where(((sel_z[:,1]<=(max(new_surf[h,1],new_surf[ind_new[h,0],1])))&(sel_z[
:,1]>=(min(new_surf[h,1],new_surf[ind_new[h,0],1]))))|((sel_z[:,1]<=(max(new_
surf[h,1],new_surf[ind_new[h,1],1])))&(sel_z[:,1]>=(min(new_surf[h,1],new_sur
f[ind_new[h,1],1])))))[0]
                            if k==0 & check==0:
                                new_sel = sel_z[pts,:]
                                check=1
                            else:
                                new_sel = np.vstack((new_sel,sel_z[pts,:]))
        #add surf_pts back to new_sel
        new_sel = np.unique(new_sel,axis=0)
        new_sel = np.vstack((new_sel,surf_pts))
        #find 100 nearest neighbours to calculate alpha
        tree = scipy.spatial.KDTree(new_sel)
        dist, ind= tree.query(new_sel,100) #this returns the indices and the
distances of the number of nearest neighbours specified
        dist_new= dist[:,1:] #this removes the first column which is zeros
(contains the distance between the point and itself)
        averages= np.zeros((dist.shape[0],1))
        for i in range(dist.shape[0]):#loop through the distances and
calculate the average
            averages[i] = np.mean(dist[i])
        alph= np.mean(averages)#take alpha as the average of all the nearest
neighbour averages
        pcd= pv.PolyData(new_sel)#initialize point cloud in pyvista
        mesh= pcd.delaunay_3d(alpha=alph)#run delaunay_3d
        name = myprop.out_direc+myprop.new_mesh_name+'.stl'
        surf = mesh.extract_surface().triangulate().clean()
        surf.save(name)#saves segmented mesh in same folder as original mesh
but with a new name
        self.report({'INFO'}, "The number of selected vertices is:
%d"%(len(selected_points)))
        return {'FINISHED'}
```

```
class ObjectCTSegPanel(bpy.types.Panel):
    """ This Panel will allow the user to tell the program once a reference
point is selected
        as well as when the full desired points are selected
    """
    bl_idname = "OBJECT_PT_ctseg"
    bl_label = "Segment CT"#this is the name of the panel
    bl_category = "Segment CT"#name of tab
    bl_space_type = 'VIEW_3D'#space where panel will be used, here it is in
the 3d viewport
    bl_region_type = "UI"#region where panel will be used, UI means in the
view area and not the header

    def draw(self,context):
        scene = bpy.context.scene
        myprop= scene.ref_properties
        self.layout.operator('ctfile.direc')#make button for selecting file
directory
        self.layout.prop(myprop,'thresh', slider=True)
        self.layout.prop(myprop,'thickness',slider=True)
        self.layout.prop(myprop, 'out_name')
        self.layout.operator('meshfile.direc')#button for output directory
        self.layout.operator('ctpt.create')#make button for creating mesh
        self.layout.prop(myprop, 'new_mesh_name')
        self.layout.operator('ctseg.roi_selection')#make button for ROI
selection


classes =
[RefProperties,SelectInputDirectory,SelectOutputLoc,CreateMeshOperator,RoiSel
ectionOperator,ObjectCTSegPanel]

def register():
    for cl in classes:
        bpy.utils.register_class(cl)#register all the classes

    bpy.types.Scene.ref_properties =
bpy.props.PointerProperty(type=RefProperties)#set the scene properties as the
defined property group

def unregister():
    for cl in classes:
        bpy.utils.unregister_class(cl)#unregister classes
    del bpy.types.Scene.ref_properties#delete the property group from the
scene properties



if __name__ == "__main__":
    register()
```

```
##References:
## [1] F. Heng, “A Comprehensive Guide To Visualizing and Analyzing DICOM
Images in Python,” Medium, 2019. [Online].
## Available: https://hengloose.medium.com/a-comprehensive-starter-guide-to-
visualizing-and-analyzing-dicom-images-in-python-7a8430fcb7ed. [Accessed: 01-
Jul-2020].
## [2] Blender Python, “Getting index of currently selected vertex ( or
vertices),” 2011. [Online].
## Available: https://blenderscripting.blogspot.com/2011/07/getting-index-of-
currently-selected.html. [Accessed: Apr-2021]
```