An improved effective method for generating 3D printable models from medical imaging

Gaurav Dilip Rathod

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Master of Science

In

Mechanical Engineering

October 12, 2017

Blacksburg, Virginia

Keywords: Additive manufacturing, Medical imaging, Image processing, Intensity interpolation

An improved effective method for generating 3D printable models from medical imaging

Gaurav Dilip Rathod

Abstract

Medical practitioners rely heavily on visualization of medical imaging to get a better understanding of the patient’s anatomy. Most cancer treatment and surgery today are performed using medical imaging. Medical imaging is therefore of great importance to the medical industry.

Medical imaging continues to depend heavily on a series of 2D scans, resulting in a series of 2D photographs being displayed using light boxes and/or computer monitors. Today, these 2D images are increasingly combined into 3D solid models using software. These 3D models can be used for improved visualization and understanding of the problem at hand, including fabricating physical 3D models using additive manufacturing technologies.

Generating precise 3D solid models automatically from 2D scans is non-trivial. Geometric and/or topologic errors are common, and often costly manual editing is required to produce 3D solid models that sufficiently reflect the actual underlying human geometry. These errors arise from the ambiguity of converting from 2D data to 3D data, and also from inherent limitations of the .STL fileformat used in additive manufacturing.

This thesis proposes a new, robust method for automatically generating 3D models from 2D scanned data (e.g., computed tomography (CT) or magnetic resonance imaging (MRI)), where the resulting 3D solid models are specifically generated for use with additive manufacturing. This new method does not rely on complicated procedures such as contour evolution and geometric spline generation, but uses volume reconstruction instead. The advantage of this approach is that the original scan data values are kept intact longer, so that the resulting surface is more accurate. This new method is demonstrated using medical CT data of the human nasal airway system, resulting in physical 3D models fabricated via additive manufacturing.

An improved effective method for generating 3D printable models from medical imaging

Gaurav Dilip Rathod

General Audience Abstract

Medical practitioners rely heavily on medical imaging to get a better understanding of the patient's anatomy. Most cancer treatment and surgery today are performed using medical imaging. Medical imaging is therefore of great importance to the medical industry.

Medical imaging continues to depend heavily on a series of 2D scans, resulting in a series of 2D photographs being displayed using light boxes and/or computer monitors. With additive manufacturing technologies (also known as 3D printing), it is now possible to fabricate real-size physical 3D models of the human anatomy. These physical models enable surgeons to practice ahead of time, using realistic true scale model, to increase the likelihood of a successful surgery. These physical models can potentially also be used to develop organ implants that are tailored specifically to each patient's anatomy.

Generating precise 3D solid models automatically from 2D scans is non-trivial. Automated processing often causes geometric and topological (logical) errors, while manual editing is frequently too labor intensisve and time consuming to be considered practical solution.

This thesis proposes a new, robust method for automatically generating 3D models from 2D scanned data (e.g., computed tomography (CT) or magnetic resonance imaging (MRI)), where the resulting 3D solid models are specifically generated for use with additive manufacturing. The advantage of this proposed method is that the resulting fabricated surfaces are more accurate.

Acknowledgements

A great many people have contributed to this thesis in different ways, and I wish to express my gratitude to each and everyone involved.

- Dr. Jan Helge Bøhn, my research advisor and committee chair, for providing me with the opportunity to work on this project. His knowledge of the subject has guided me throughout my research.
- My advising committee, Dr. Christopher B. Williams and Dr. Xiaoyu (Rayne) Zheng, for providing me with excellent feedback, encouragement, and advice.
- Dr. Karen P. DePauw, Dean of the Virginia Tech Graduate School, for providing me with an exception to defend under SSD enrollment past its deadline. I appreciate her support in my hour of need.
- Dr. Corina Sandu, Associate Department Head for Graduate Studies, and her staff for their advice and support.
- The *3D Design Studio* in the Virginia Tech Library for providing 3D printing services, which were used to 3D print the models shown in this thesis.
- Tino Kluge, for publishing the *Simple cubic spline interpolation library* under GNU General Public License, which was used for the cubic interpolation computations in this thesis.

Finally, I would like to thank my family and friends. Without their love and encouragement, I would never have gotten close to where I am today.

Table of Contents

List of Tables

Table Of figures

Chapter 1 Introduction

Medical imaging plays a vital role when it comes to diagnosis amongst medical practitioners. Since the advent of medical imaging technologies, it has revolutionized the medical industry by providing the doctors with tools to peek inside the human body in a noninvasive manner. Furthermore, with improvements in medical imaging technologies, more informative data with better accuracy can be obtained. This has led to increased dependency on imaging in fields of medical sciences, wherein surgeons now use imaging to cross-reference the operational procedure as well as perform image-guided surgeries [1].

Medical imaging continues to depend heavily on a series of 2D scans resulting in a series of 2D photographs being displayed using light boxes and/or computer monitors. Today, these 2D images are increasingly combined into 3D solid models using software. There has been an exponential growth in number of commercial as well as open-source software that provide the tools to visualize the Medical data in a constructive way. Examples include 3Dslicer, OsiriX, MITK, and MeVisLab are a few of them [2]. However, there are limitations to how much information can be inferred from 2D images or from a flat-screen. Although, 3D reconstruction technologies are readily available, the scope of insight gathered from them is still limited.

With the advent of 3D printing, inexpensive and accurate physical 3D models can be generated that provide an improved understanding of the patient's anatomy. A real sized 3D printed model can further improve doctor's understanding of the patient's anatomy. 3D models can also be used to educate people and students towards a better understanding of anatomy. Farooqi & Sengupta [3] highlights the potential of 3D printed model to create cardiac replicas. Bio printing technologies can potentially use these models to develop organ implants specifically tailored as per the patient's requirements [4]. The potential for 3D printing in medical industry is huge.

For printing 3D models, medical imaging data has to be first converted to 3D printable models. 3D medical imaging data is typically obtained in the form of a series of 2D images separated from each other at regular intervals. This series of 2D data needs to be converted to a fileformat named Stererolithography (.STL) file. A .STL file describes the approximate 3D topology of a surface using an unordered list of triangular facets, without the explicit topological data connecting the facets. However, the conversion if done manually is often tedious, and if done automatically, it can insert undesirable artifacts and inaccuracies in the model resulting in life-threatening consequences in case of organ implants. There is a need to develop robust methodology that can generate 3D printable models automatically without inaccuracies and artifacts.

1.1 Problem statement

Medical Imaging data from computed tomography (CT) and magnetic resonance imaging (MRI) technologies are available in the form of series 2D images referred as slices. A slice can be considered as cross-sectional bitmap describing the anatomical configuration on a specific plane of the scanned volume of interest. A series of slices therefore contains information regarding the entire volume at a regular intervals or pitch.

Traditionally, the process flow of generating 3D printed models begins with acquisition of medical imaging in the form of digital images and is followed by image segmentation process. During segmentation process, the image data is converted from grayscale values to binary values, in order to distinguish a particular structure, based on its grayscale value. The segmented images are then used for mesh generation and stored in the .STL (stereolithography) fileformat. Post processing of mesh (mesh refinement) is often required before finally converting the mesh into a physical 3D printed model [5].

Accuracy of the developed model can be affected at any step in the production process. The accuracy of the model is primarily dependent on the resolution of the 2D CT scan data. Low resolution data provides less tolopologcal information and 3D models developed using these data are often less accurate geometrically. Due to the limitation on permissible amount of radiation for an average human being, obtaining higher resolution medical imaging data is not a feasible solution.

Besides from low image resolution, image segmentation introduces many artifacts as during this step, the grayscale data is converted into binary data having value of either 1 or 0. Exclusion of data during the segmentation process may lead to holes, degenerate faces, and non-manifold conditions in the generated mesh due to insufficient information. The mesh generated needs to be post processed for refinement purposes. During mesh refinement new facets are produced usually by sub-diving existing ones. These facets may not retain the original geometry leading to topological inaccuracies.

Slicing software can introduce gaps while slicing the model due to difference in the slicing and vertical resolution of the data. These gaps needs to then fill in using specific contour stitching algorithms. These stitching algorithms can make changes in contours deviating from the original topology furthermore. In any case, excessive post processing is required before printing the model.

There is a requirement of a robust method for automatically generating 3D models from 2D scanned data (e.g., computed tomography (CT) or magnetic resonance imaging (MRI)), where the resulting 3D solid models generated specifically for use with additive manufacturing.

1.2 Solution Overview

This thesis solves the problem in three stages: Stage 1 resamples the 2D CT scan images to increase their in-plane resolution effectively increasing the available information before generating the surface mesh; Stage 2 generates the intermediate slices between the input images to match printer resolution with an objective of reducing errors introduced potentially during the slicing procedure; and Stage 3 generates mesh layer-by-layer segmenting the voxels at the same time, to obtain a more robust mesh.

The approach is to take advantage of the information stored in each pixel and generate intermediate information. Each pixel in a CT scan image represents an integration of values across a specific volume. Stage 1 helps to subdivide the voxels in the horizontal direction providing better in-plane resolution. A weighted resampling algorithm ensures that the quality of the images do not deteriorate. Stage 2 subdivides the voxel vertically to improve the out-of plane resolution. In Stage 2, the vertical resolution of the new slices is matched with the 3D printer's print resolution to ensure no gaps in contours can be introduced in the slicing software. In essence, Stages 1 and 2 develops a high resolution volume of grayscale values, which approximately represents the actual scanned volume of the patient's anatomy.

Stage 3 generates the mesh from the developed voxel space. This is done by implementing marching cube algorithm. A virtual cube marches pixel by pixel between two consecutive layers and assigns triangular facets at the location of a surface. The criteria to locate the surface is to compare each pixel value with a threshold (grayscale) value provided by the user ,depending on the material densityof the body part to be segmented The advantage of this approach is that the original scan data values are kept intact longer, so that the resulting surface is more accurate. Layer-to-layer mesh generation allows continuity of surface and smoother finish.

1.3 Thesis organization

This thesis aims to provide a simple and an improved method of 3D printing medical imaging. A thorough description of the procedure is presented. Systematic experiments are conducted to validate each step in the process. The work done is organized as follows:

Chapter 1 introduces of scope of 3D printing in medical industry and the need for better methods to convert medical imaging to 3D printable models. This chapter also outlines the objectives of the thesis.

Chapter 2 describes the theory behind medical imaging, standard 3D printing friendly formats, image processing techniques and previous work.

Chapter 3 proposes an improved method to convert 2D image data set to 3D printable surface model.

Chapter 4 describes the experimental data, important parameters and setups used to validate the proposed method.

Chapter 5 discusses the results obtained from the experiment.

Chapter 6 concludes and summarizes the thesis.

Chapter 2 Literature Review

This chapter will discuss about the concepts required to understand the proposed approach. This includes the medical imaging and the resulting 2D scanned data; the .STL fileformat used to describe geometries to be fabricated via additive manufacturing; image processing techniques used to process 2D scanned data; and methods for converting 2D scanned data into 3D solid models and then back into 2D data sutable for additive manufacturing. This is then followed by a brief set of obervations summarizing the finding of this literature review.

2.1 Medical imaging

Computed tomography (CT) imaging is a widely used medical imaging technique used for obtaining visual representations of the interior of the body [1]. Medical imaging has found its applications in the medical industry for diagnosis of diseases and abnormalities in the anatomy of the patient in consideration. It is used extensively in medical industry as it gives a better idea of patient’s anatomy and helps to pinpoint the underlying problem. Medical imaging consists of variety of techniques like X-ray radiography, medical ultrasonography or ultrasound, endoscopy, elastography, tactile imaging, thermography, medical photography, positron emission tomography (PET), magnetic resonance imaging (MRI), and computed tomography (CT), the latter being of particular interest for this thesis [6]. The following three subsections will therefore discuss the fundamentals of computed tomography, followed by typical scan resolutions, and finally the fileformat used to describe the 2D scanned images.

2.1.1 Computed Tomography

A CT scanner is a device capable of obtaining cross-sectional slices of images of the patient’s anatomy. It consists of a doughnut shaped gantry, with an X-ray source at the one end and a series of detectors at the other end (Figure 1). The X-ray source emits a fan shaped X-ray beam which attenuates as it passes through the patient and is measured at the detectors [7]. A CT scan is a measurement of linear attenuation coefficient of X-ray (μ) as the X-ray passes through the patient. The attenuation coefficient (μ) is a measurement of absorption of X-ray by a particular material. The gantry is rotated all around the patient and a series of projections are obtained.

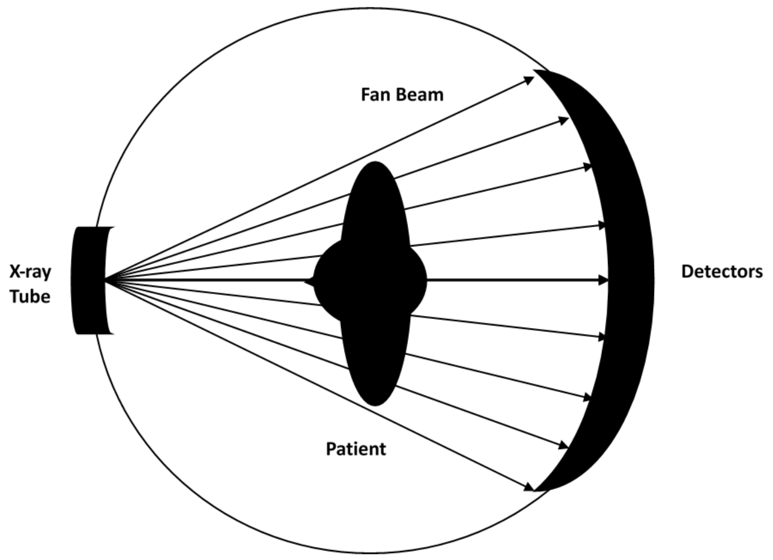


Figure 1. Schematic of a CT procedure

Using back projection method and appropriate filters [8], the density of material at a particular location is obtained. The attenuation coefficient values are normalized with respect to air or water and converted to CT numbers or Hounsfield units (HU). The HU are calculated as follows [8] :

$$HU = \frac{\mu_{object} - \mu_{water}}{\mu_{water}} * 1000$$

The HU is around -1000 for air, 0 for water, and around 700-3000 for denser materials like bones. Table 1 shows the HU for different tissues and parts. These HU are then stored as grayscale values in the form of CT scan images. The higher the density of a material, the higher its grayscale value. The grayscale values in a CT image represents the material density at that location. Figure 2 shows a typical CT image.

Matter	**HU value**
Air	-1000
Lung	-850 to-910
Fat	-50 to 1000
Water	0
White matter	20 to 30
Bones	700 -3000

Table 1. Hounsfield numbers of different structures based on their X-ray attenuation coefficient.

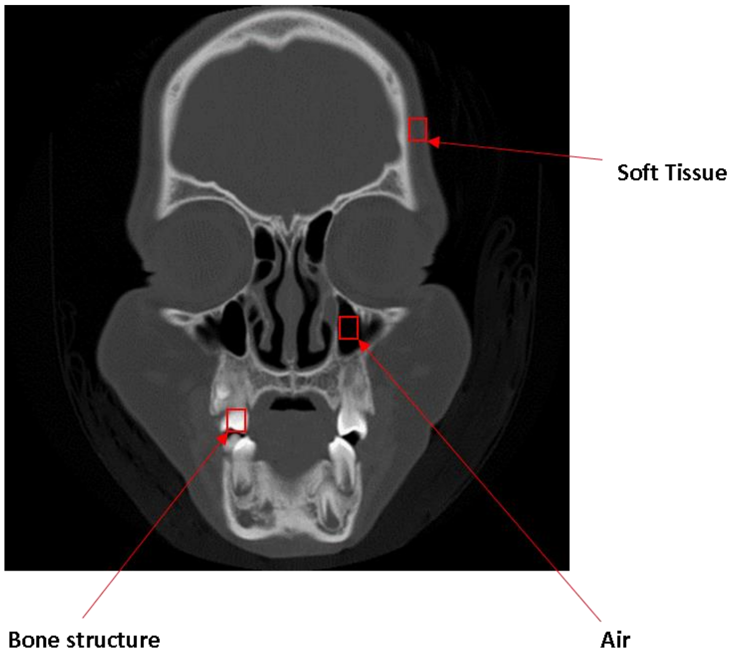


Figure 2. CT scan output image depicting different material densities in the form of grayscale values

2.1.2 Data Resolution

Spatial resolution of a typical CT scan output can be divided into two types: the XY resolution in the scan plane, and the Z resolution (i.e., the separation between the scanned planes). The XY resolution is the pixel size in X and Y directions. Since the pixel is square in geometry, the spatial resolution in the X and Y direction are usually same. Typically, a CT scan image has a pixel resolution of 512 x 512. The spatial resolution depends on the field of view of the scanner, spatial resolution of the detector channel, and it is limited by the permissible radiation dose [9]. If the field of view is smaller, then a higher spatial resolution in the X and Y directions can be obtained, and vice versa. For instance, if the diameter of the field of view (scan circle) is 250 mm and the matrix size of the image is 250 x 250, then in-plane spatial resolution of the image would be 1 mm x 1 mm. However, if the matrix size is doubled or diameter of the scan circle is reduced to half its orginal value, then the new in-plane special resolution would be 0.5 mm x 0.5 mm [9]. Typically, the XY resolution is in range of 0.05 to 5 mm, depending on the field of view.

The Z resolution (or slice thickness) depends on the method of acquisition, the scan time, the CT scanner power, etc. Generally, the slice thickness is an order of magnitude greater than the XY resolution, and somewhere between 0.5 and 10 mm [8]. Thus, the voxels produced are always rectangular in shape, with a typical X:Y:Z aspect ratio in the range of 1:1:5 to 1:1:20. High resolution CT scanners can however obtain an aspect ratio of 1:1:2.

2.1.3 Data Format

Medical images are usually available in the form of grayscale images described in the *Digital Imaging and communications in Medicine* (DICOM) fileformat, which is widely used in the radiology industry [10]. The DICOM fileformat also contains metadata such as patient's information, clinic information, radiologist's comments, embedded tags etc. The image data can be easily extracted from DICOM fileformat and stored as portable gray map (PGM) format. A PGM image is a single channel pixel array that is capable of storing data in grayscale. The PGM file can be either in binary or in ASCII format, with the latter being more memory intensive. The value null corresponds to the color black and the highest value corresponds to the color white.

The PGM format definition as given by *Netpbm* is as follows [11]:

1. A "magic number" for identifying the file type. This is "P5" in case of a binary PGM image, and "P2" in case of ASCII PGM image.

2. Whitespace (blank, TAB, CR, or LF).

3. The width, formatted as ASCII characters in decimal (number of columns).

4. Whitespace (blank, TAB, CR, or LF).

5. The height, again in ASCII decimal (number of rows).

6. Whitespace (blank, TAB, CR, or LF).

7. The maximum grayscale value (Maxval), again in ASCII decimal. Must be less than 255 (for 8-bit data), and more than zero.

8. A single whitespace character (usually a newline).

9. A raster of height rows, in order from top to bottom. Each row consists of width grayscale values, in order from left to right. Each grayscale value is a number from 0 through *Maxval*, with 0 being black and *Maxval* being white. Each grayscale value is represented in pure binary by either 1 or 2 bytes. If the *Maxval* is less than 256, it is 1 byte. Otherwise, it is 2 bytes.

2.2 The .STL Fileformat

The stereolithography (.STL) fileformat is the *de facto* standard throughout the additive manufacturing industry. The format was developed in 1987 by Albert Consulting Group [12] for 3D Systems Inc. and their stereolithography aparatus. The .STL fileformat describes a CAD model as a boundary representation (BRep) that ideally corresponds to the surface of a rigid solid. Unfortunately, it frequently fails to achieve this goal. The following three subsections will therefore discuss the sources of these errors, looking at the details of the fileformat, the expecations for rigid solids, and some examples of common problems.

2.2.1 ASCII and Binary Fileformats

The .STL fileformat describes is a collection of triangular facets without explicit topological data connecting the facets. Each facet (Figure 3) is defined by the unit normal and vertices of the triangles using a three-dimensional Cartesian coordinate system described by the following numerical data [13]:

$$n_x \;, n_y \,, n_z$$

$$p_x^1 \,,\, p_y^1,\, p_z^1$$

$$p_x^2 \,,\, p_y^2,\, p_z^2$$

$$p_x^3 \,,\, p_y^3,\, p_z^3$$

Where ($n_x \;, n_y \,, n_z$) represents the surface normal calculated from the three coordinates ($p_x^1 \,,\, p_y^1,\, p_z^1$), ($p_x^2 \,,\, p_y^2,\, p_z^2$) and ($p_x^3 \,,\, p_y^3,\, p_z^3$) in accordance with the right-hand rule. The .STL file does not retain any information of the texture, color or other common CAD attributes. Also, the units not specified.

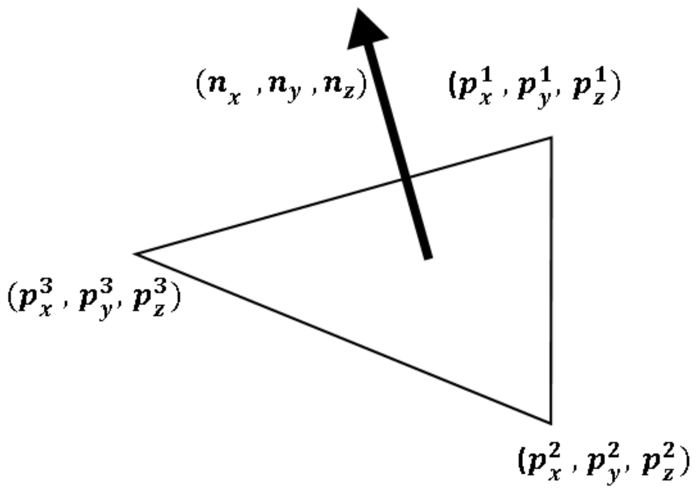


Figure 3. Typical mesh Facet

It is available in both an ASCII and a binary version. ASCII formats are more suitable for debugging purposes due to their comprehensible nature, while binary .STL formats are used almost exclusively for rapid prototyping purposes. The primary reason for this is the difference in storage requirement by each format to define same topology having same number of triangular facets. Each facet in a binary format only requires 50 bytes based on the IEEE floating-point Standard [14], while ASCII format will require about 300% times the space required by the Binary version [15] due to its verbose nature. Both formats has a specific format to follow. Figure 4 and Figure 5 shows the .STL binary and ASCII formats, respectively [15].

It should be noted that the ASCII and binary formats are incompatible. The conversion between these two formats will in general result in changes in the XYZ values of the vertices. Likewise, the binary format is based on the IEEE single-precision floating-point standard [14], which provides for approximately 5 digits precsion (base 10). Hence, when a program internally computes vertex values using double-precission (approximately 15 digits precision), and then saves the data to a file in the .STL fileformat (either ASCII or binary), the resulting XYZ vertex data will change, which will also change the topology of the the resulting CAD model. The following two sections will examine the expecations for rigid solids, and the errors that might arise from there conversion errors and other errors arising from converting from other CAD model representations (e.g., non-uniform rational B-spline surfaces).

Address	Length	Type	Description
0	80	char	Header Information
80	4	Long	Number of facets in solid
First facet (50 bytes):			
84	4	float	Normal (x-component)
88	4	float	Normal (y-component)
92	4	float	Normal (z-component)
96	4	float	Vertex 1 (x-component)
100	4	float	Vertex 1 (y-component)
104	4	float	Vertex 1 (z-component)
108	4	float	Vertex 2 (x-component)
112	4	float	Vertex 2 (y-component)
116	4	float	Vertex 2 (z-component)
120	4	float	Vertex 3 (x-component)
124	4	float	Vertex 3 (y-component)
128	4	float	Vertex 3 (z-component)
132	2	short	Attribute info. (Not used)
Second facet:			
134 ….			

Figure 4. .STL Binary format [15]

```
Solid Test_part
   Facet normal    0.000000e+00    1.000000e+00    2.6344149e-09
      Outer loop
         Vertex    3.000000e+00    1.400000e+00    4.000000e+00
         Vertex    4.000000e+00    1.400000e+00    4.000000e+00
         Vertex    3.000000e+00    1.400000e+00    3.000000e+00
      endloop
   endfacet
facet normal ….
:
<remaining facets>
:
endsolid
```

Figure 5. .STL ASCII format [15]

2.2.2 Rigid Solids

For a .STL file to sufficiently define a topology, each triangular facet in the file should satisfy the following conditions [13]:

- Each edge in the triangulation is shared by at most two triangles.
- A vertex in the triangulation can be shared by any number of triangles.
- Each triangle has at least one point in common with another triangle (connectivity; requires at least two triangles).
- If a vertex of a triangle is shared by a second triangle, then this point is also a vertex of the second triangle.
- No triangle has an intersection with the interior of any other triangle (no piercing, no overlapping)

2.2.3 Common problems

Conversion of a 3D CAD model to .STL files if done incorrectly results in a number of errors in the file if done incorrectly. The common problems in .STL files are as follows [16] :

1. Gaps (cracks, holes, punctures) that is, missing facets (Figure 6.a)
2. Overlapping facets (Figure 6.b)
3. Degenerate facets (where all its edges are collinear)(Figure 6.c)
4. Non-manifold topology conditions

Tessellation of surfaces with large curvature can result in errors at the intersections between such surfaces, leaving gaps or holes along edges of the part model [16]. A surface intersection anomaly which results in a gap is shown in (Figure 6.a)

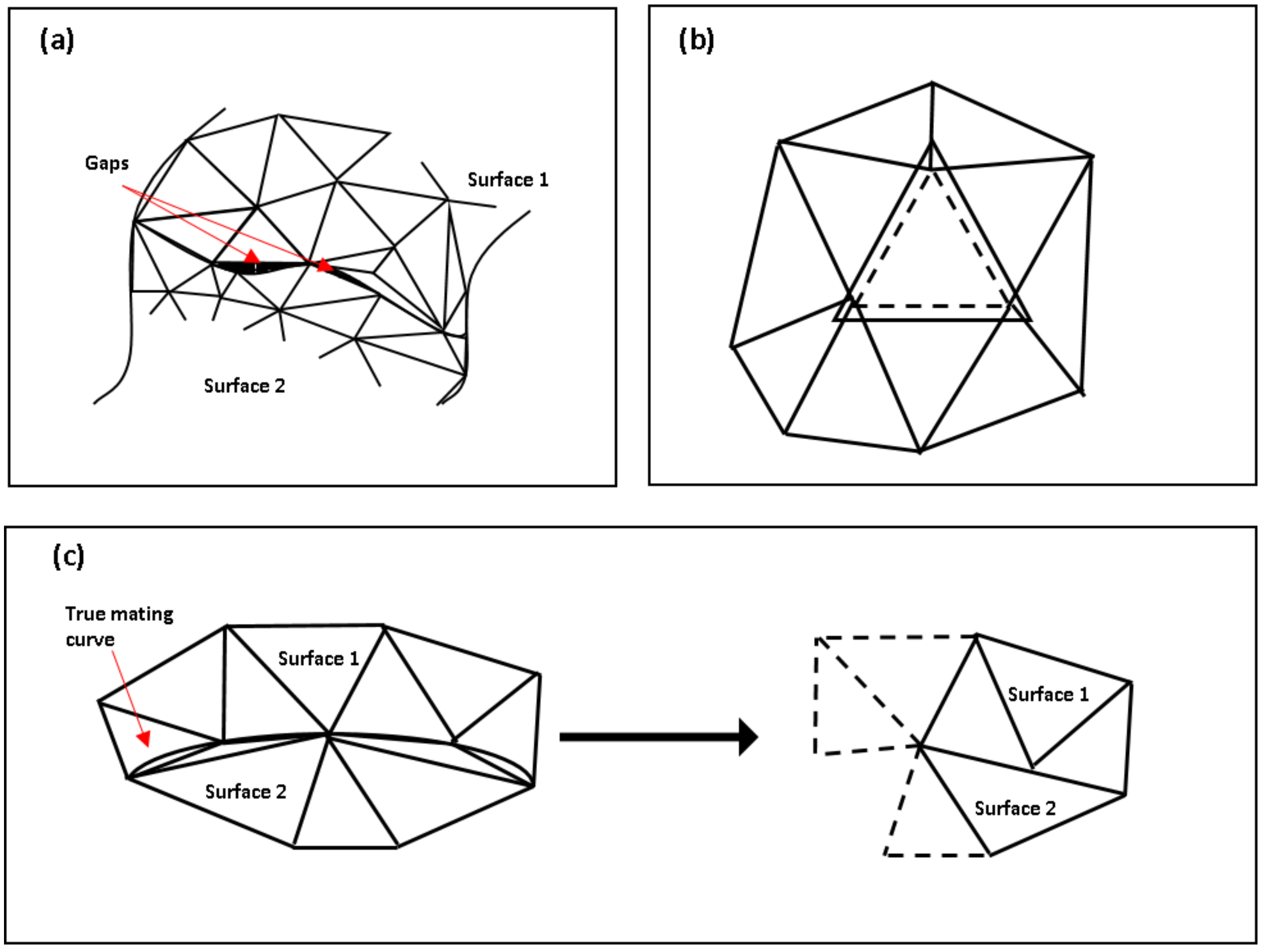


Figure 6. Common errors in .STL meshes (a) Gaps (b) Overlapping Facets and (c) Shell punctures owing to Degenerate facets [16]

Geometrically degenerate facets can be introduced in a .STL file when the edges of facets are collinear even if their vertices are distinct. Sometimes a stitching algorithm can introduce a degenerate facet while attempting to avoid shell punctures by generating new facets. Overlapping facets are yet another errors that are added due to improper triangulation process. The vertices in 3D space are stored as floating-point numbers. If the precisions are set too liberally, these can lead to numerical round-off errors resulting in overlapping facets. Figure 6b shows an example of overlapping facet. Round-off errors in triangulation of the fine features can generate non-manifold conditions. There are three types of non-manifold errors viz.

1. A non-manifold edge (Figure 7.a &b)
2. A non-manifold point (Figure 7.c)
3. A non-manifold face (Figure 7.d)

A valid manifold would be the one whose facets have only one adjacent facet each, that is, one edge shared by two facets only. A non-manifold edge is generated when an edge is shared among four different facets. Hence the non-manifold edges must be resolved such that each facet has only one neighboring facet along each edge, which reconstructs a topologically manifold surface. A non-manifold point is created when a vertex of one facet lies on the face of another. While non-manifold face simply arise when two facets lie on each other but do not share any common edges.

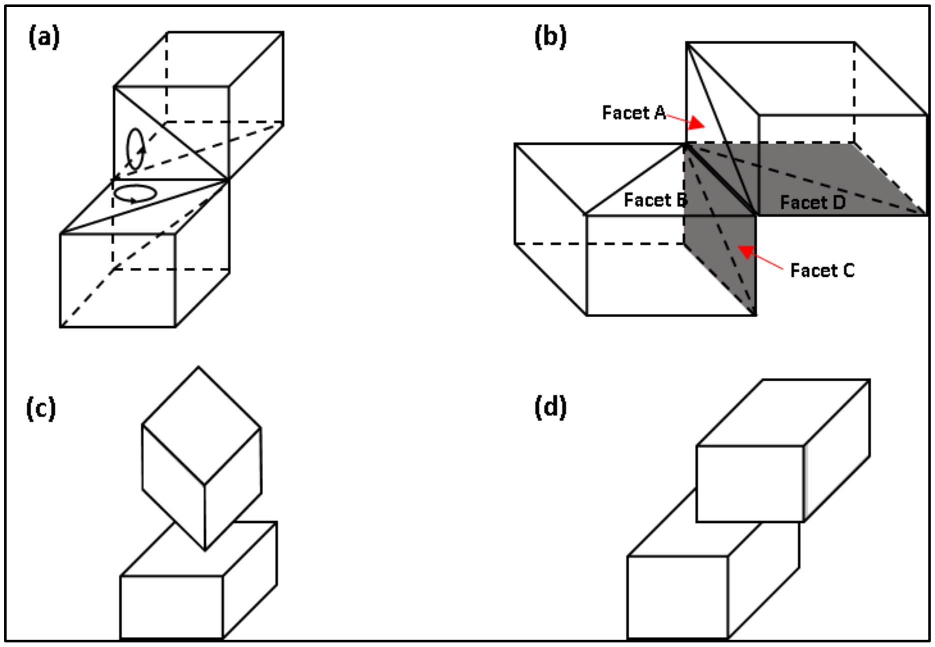


Figure 7. Non-manifold errors in .STL meshes (a) & (b) Non-manifold edges, (c) Non-manifold point (d) Non-manifold face [16]

If the .STL mesh has gaps or conflicting geometry, it will lead to open contours during the slicing process. Also, flaws in the contour generation algorithms, may lead to erroneous contours which is not favorable. To overcome these problems, a variety of advanced slicing procedures have been developed [17] [18] [19]. However, there is still a room of error in cases of ambiguity.

2.3 Image Processing

Image processing has been an integral part of medical imaging for the purpose of image compression and image resampling. It helps in analysis, enhancement and display of images obtained from medical imaging. This section will discuss some image processing techniques required to understand the proposed approach in this thesis.

2.3.1 Image resampling

Resampling is a process of transforming images geometrically. There are two types of image resampling: Image magnification and image minification. Image magnification increases the image resolution, and image minification reduces the image resolution. For instance a 512 x 512 image, if upscaled 4 times, results in a 2084 x 2084 image.

In each case, the discrete image to be resampled is first mapped to a continuous intensity function by convoluting the input grid with a continuous interpolation function. The new continuous function is then rescaled/resampled to match the range of the new grid points. The values at the new grid points can then be interpolated from the resampled function.

However, rescaling the image affects its quality. Upscaling may result in undesired jaggedness. Downscaling may result in visible quality loss as each pixel in the new image is a weighted average of multiple pixels. To overcome these problems, a variety of image interpolation functions have been developed. Lehmann compared 8 different interpolation kernels with sizes varying from 1x1 to 8x8 [20]. The comparison was done on the basis of spatial and Fourier analysis, computational complexity, runtime evaluations, qualitative and quantitative error analysis. The following interpolations were compared :1) truncated and windowed sinc; 2) nearest neighbor; 3) linear; 4) quadratic; 5) cubic B-spline; 6) cubic; g) Lagrange; and 7) Gaussian interpolation and approximation techniques with kernel sizes ranging from 1 x 1 up to 8 x 8. Of these, Lehmann suggests that the cubic interpolation kernels are the most suitable for medical imaging purposes due to their excellent smoothing properties.

A bicubic interpolation scheme takes into consideration 16 adjacent points to derive the value of the a single pixel. It is an extension of cubic interpolation for two-dimensional space. The cubic interpolation algorithm proceeds by generating 3rd degree polynomial splines to obtain a smooth continuous function. Bicubic interpolation algorithm generates smooth continuous function in two dimension. If a unit grid of data with it's corners located at (0,0),(0,1),(1,0) and (1,1) is resampled using the bicubic interpolation algorithm, then it's continuous function f can be written as

$$f(x,y) = \sum_{i=0}^{3}\sum_{j=0}^{3} a_{ij}x^i\, y^j$$

The 16 coefficients (a_{ij}) can be determined by four boundary conditions in the C^0 domain, eight equations derived in the C^1 domains (f_x, f_y) and another four equations derived in the C^2 domain $(f_{x,y})$. The resulting interpolation function should fulfill the following criteria

- It is continuous in the C^0 domain
- It is continuous in the C^1 domain (i.e. continous slope); and
- The interpolation is independent of neighborhood samples.

2.3.2 Histogram normalization

CT windowing (or contrast enhancement) is an image processing technique that is widely used to change the visual properties of an image. Often referred to as histogram normalization, it involves transforming the histogram of an image by using its normalized sum. The normalized histogram is then mapped to the new intensity scale. The effect of histogram normalization is to enhance the contrast between two adjacent features to make them easier to see. CT windowing uses two parameters to achieve this: window level, and window width

- Window level - Window level is the midpoint of the CT values to be displayed.
- Window width - Window width is the range of CT values to be displayed. A wider width corresponds to less stretching, and vice versa.

Histogram stretching thus consists of first truncating the histogram in a given range and then mapping the truncated histogram to the original scale. The values in the given range gets evenly distributed around the window level and scaled across the window width. Figure 8 shows a histogram before and after applying the contrast enhancement. Interpolation is used in contrast enhancement to interpolate values in between. In essence, contrast enhancement is similar to resampling the data, but within a given range. In Figure 8, the histogram for the given image is weighted more near the higher grayscale values. The lower values are thus truncated, and the resulting histogram is then rescaled to the original range.

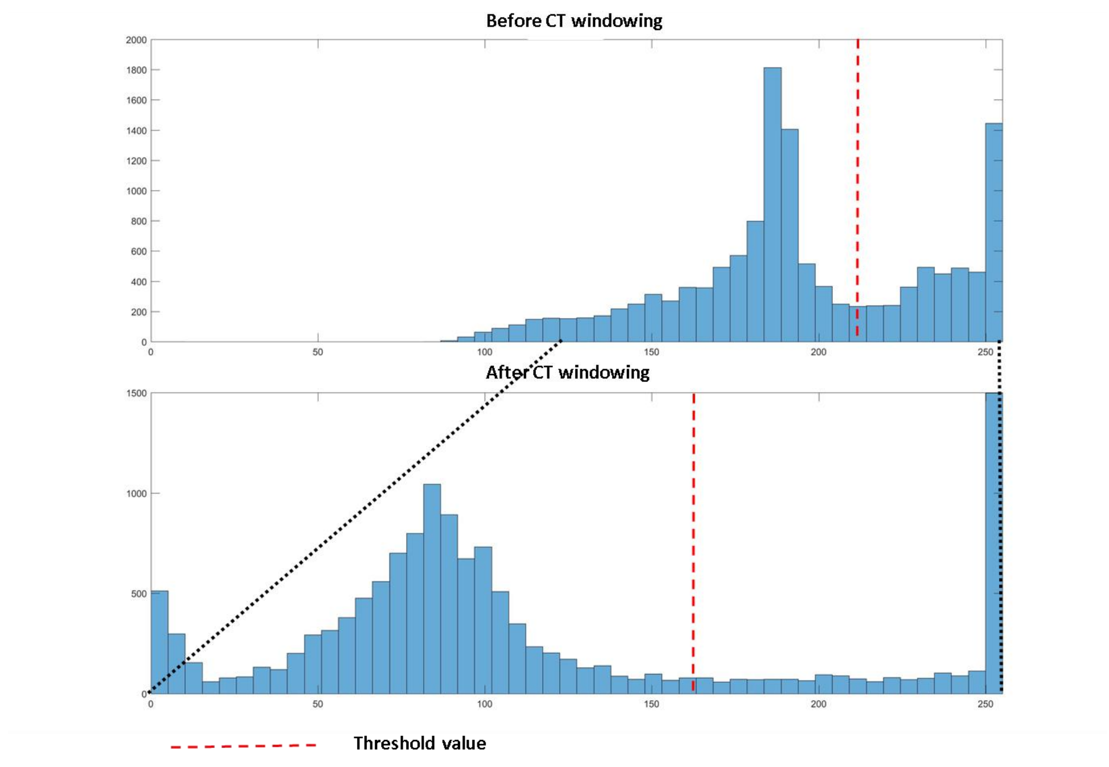


Figure 8. Histograms of a data set before and after contrast enhancement procedure

2.4 Production process

With the increasing popularity of additive manufacturing, there are now several open-source software packages available that convert a .STL model into a 3D printed part. The production procedure of a 3D printed part starts with the creation of a solid model described in the .STL fileformat. This solid model is then sliced into layers corresponding to the fabrication layer thickness. This results in contours on the slicing layers, separating material from non-material. From here a tool path is generated that traces the generated contour to realize each layer.

The following subsections will first discuss the fundametals of reconstruction of 2D medical scanned data into 3D solid models, and then the slicing procedure, where the 3D solid models are converted into 2D contours used to fabricate the physical model layer by layer using additive manufacturing.

2.4.1 3D Model reconstruction

Reconstruction of 3D surface model from medical imaging data has been an unceasing topic in the scientific community. Reconstruction of surface is an important step in data visualization and mesh generation. There are three methods popularly used for reconstruction: sweep blend modelling by generating curves from input data points, grayscale intensity interpolation which uses voxel based mesh generation techniques, and contour interpolation methods which uses contour evolution algorithms to obtain intermediate contours and generate mesh from the contour points.

The most common sweep blend method implemented by majority of the software is by non-uniform rational B-splines (NURBS) based modelling [21] [22]. In NURBS based modelling, the surface data points are first acquired from the input images based on grayscale level prediction and geometric splines (curves) are defined using the data points to obtain a solid model. These splines can be considered as mathematical representation of a 3D shape effectively by using 2D circle, line or any curve for that matter [23]. Surface mesh is then generated by defining triangular facets between the obtained curves. This method is computationally complex due to requirement of generating geometric splines before constructing the surface model.

Another common method is intensity interpolation for object reconstruction from serial cross-sections medical imaging [24] [25]. Here the grayscale value of input images are used to generate intermediate data before generating the surface model. However, surface generation is done using contour extraction from the interpolated slices and by sweeping between the contour points similar to the sweep blend methods.

Contour interpolation method first detects contours from the input images, and then interpolates between two consecutive contours to generate contours of intermediate slices. Mesh is generated by defining vertices of the triangular facets on two parallel contours. However, it may lead to intersecting or missing facets due to mismatch in the number of contour points in each contour/polygon. Attempt to implement a better contour evolution method by defining feature-lines have been made [26]. Nonetheless, reproducibility can be a problem if feature-lines are specified manually. Contour or line-segment interpolation methods showed artifacts or even failed in some cases [24] [26]. Attempts at combining intensity interpolation and contour extraction to generate layer-by-layer surface mesh has also been made [27].

It is notable that in most of these cases, artifacts were introduced from the data generated by the surface reconstruction algorithm itself. Also, a lot of data was discarded at the segmentation steps, wherein the data was converted from grayscale values to binary value. None of these methods consider printer resolution while constructing surface mesh. The above methods can be used efficiently for visualization purposes, but they often introduce errors in the generated surface mesh which leaves a chance for further errors introduced during the slicing process.

2.4.2 Slicing procedure

Slicing is an important step in the process of converting the 3D model into a series of 2D slices corresponding to the layers to be fabricated. In slicing, sets of parallel planes are intersected with CAD model (Figure 9.a). The space between any two consecutive horizontal planes is referred to as a slice thickness. The slice thickness corresponds to the 3D printer's vertical print resolution (layer thickness). A conventional slicing software starts off by defining a slicing plane for 3D printing. Each facet in a .STL file is then checked for intersection with the defined planes. If the slicing planes intersects the facet, then coordinates of the points of intersection are stored for the respective plane. Four different kinds of intersection are possible [17]:

1. Intersection with two sides of the facets (Figure 9.b Case 1)
2. Intersection with a vertex (Figure 9.b Case 2)
3. Intersection with a vertex and a side (Figure 9.b Case 3)
4. Entire facet lying on the slicing plane (Figure 9.b Case 4)

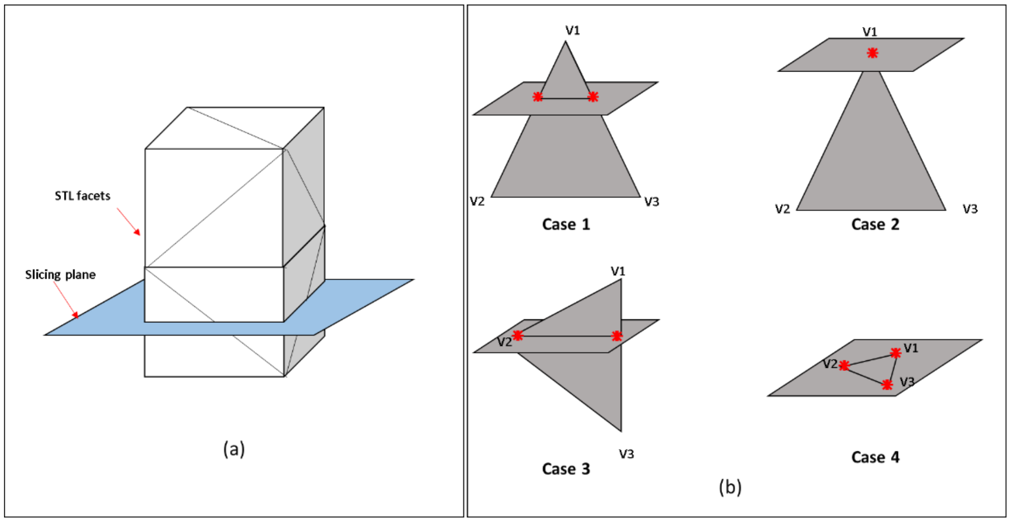


Figure 9. (a) Slicing plane in slicing algorithms (b) Types of intersections possible between slicing plane and the facet

The coordinates of intersection points are obtained by linearly interpolating from the facet vertices (Figure 10). The obtained coordinates are subject to round-off errors during interpolation. If a high resolution mesh is used to define the surface, wherein the triangular facets are packed together closely, then these round-off errors may lead to disproportionate distance between two consecutive points, or maybe even overlapping points. The points obtained in a given plane are then connected to each other by contour generation algorithms to generate closed contours/polygons. This leads to a situation wherein, the contours generated are closed, but deviate from the geometry defined by the .STL.

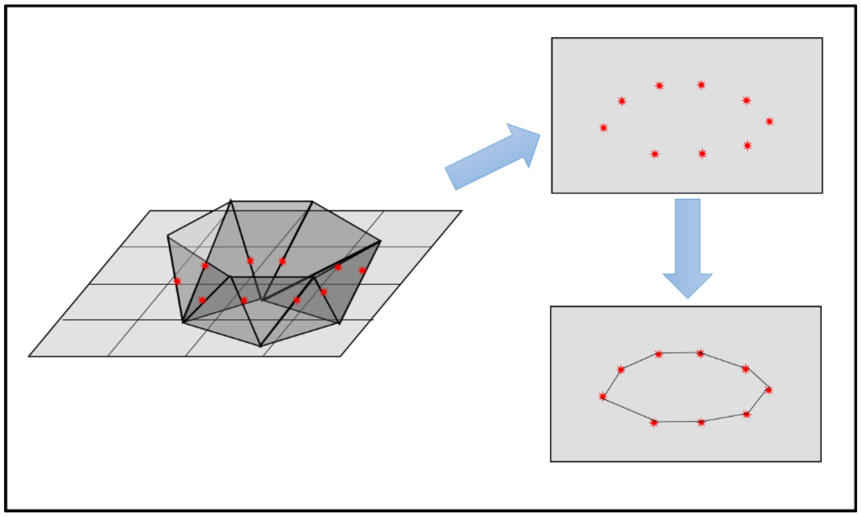


Figure 10. Slicing prcocedure: Intersection of horizontal plains with the .STL file to obtain contour point cordinates by interpolation techniques followed by contour generation process

Once each slice contains the contour information in the form of closed contours or polygons, it is then converted into machine readable language: G-code. G-Code file is a bunch of ordered specific instructions to a specific printer which systematically defines all of the movements the printer nozzle head should make to reproduce the part(s) to be printed. Thus slicing software takes in geometrical information from the .STL file, converts it into series of closed contours at regular interval, and translates it into specific movement commands for the 3D printer.

2.5 Observations

Based on the literature discussed in Section 2.1 to 2.4, the following inferences can be made:

1) Obtaining high resolution CT scan data is not feasible in most cases due to the limitations on the permissible radiation dose in a patient’s lifetime. Suitable image processing techniques should be applied to improve the resolution of CT scan images for developing precise 3D printed models.

2) The .STL format although extensively used in the additive manufacturing industry is prone to errors due to its inherent limitations. The .STL file saves information of the triangular facets based on IEEE floating-point standard [14]. Hence, when a program internally computes vertex values using double-precission (approximately 15 digits precision), and then saves the data to a file in the .STL fileformat (either ASCII or binary), the resulting XYZ vertex data will change, which will also change the topology of the the resulting CAD model.
3) Traditional mesh generation methods are often complicated and fail to handle ambiguous situations. They don't consider 3D printer's layer thickness while generating 3D surface models from 2D medical imaging data.
4) Slicing software takes in geometrical information from the .STL file, converts it into series of closed contours at regular intervals based on 3D printer's vertical print resolution and translates it into specific movement commands for the 3D printer. Any errors introduced during the slicing procedure will reflect in the physical 3D printed part, and caution should be taken to avoid them.

Chapter 3 Methodology

This section describes the algorithm used in developing a 3D .STL file from 2D CT scan images. The basic idea is to generate a volume of grayscale data from the input slices, and convert it directly to .STL format suitable for additive manufacturing (Figure 11). The net effect is to eliminate complicated computational steps like NURBS generation or contour evolution adopted by traditional methods. It also eliminates major chunk of contour extraction work by the slicing software by feeding it closed contours for every slice, minimizing errors introduced during these steps. This is done by generating .STL meshes layer-by-layer in accordance with the printer layer thickness. This enables obtaining contours directly from the .STL files and converting it to machine readable G-code eliminating the ambiguities introduced by slicing algorithms.

A C++ code has been developed to conduct the experiment. The code contains specially defined structures for images, triangular facets and virtual mesh grid, etc. The code is capable of performing image processing algorithms like resizing, cropping, reading, writing, inverting the colors, and segmenting the images. Other capabilities of the algorithm includes image intensity interpolation, mesh generation, and writing .STL files. The code is appended at the end of this thesis in Appendix A

3.1 Process flow

The process starts off by reading a sequence of input images and assigning it to an array of specially defined Image structure (Figure 12). Each image is passed through an algorithm which reads the image header consisting of magic number, width, height, and maximum grayscale value, and the image data which comprises of the actual pixel data. When all the images are read, essentially a copy of images is generated in the program memory on which other operations can now be performed. No changes are made in the original images.

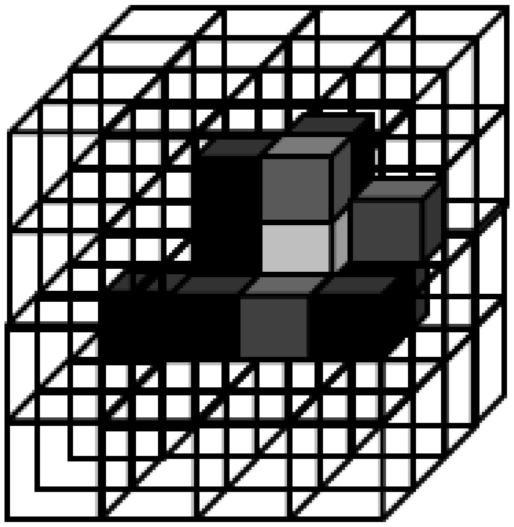


Figure 11. Schematic of a voxel space

The next step is to resample the images. The idea behind resampling the images is to generate new data points increasing resolution of the images. This step is essential if the input data is very coarse in nature. It is important to use appropriate interpolation scheme so as to ensure a smoother transition between newly generated data points. Once the horizontal resolution is taken care of, the images can be cropped to focus on the area of interest if requested by the user. User has to provide the new dimensions of the images and the cropping offsets. Cropping at this step helps saving program memory and allows faster execution in comparison to cropping after the next stage.

Generating intermediate images is the next step in the process. A new array of image structure is initialized at this point. The size of the array is calculated by comparing the slice resolution and the desired resolution. This structures in this array as the same dimensionality as the array initialized earlier to hold input images. The size of the array is however is much larger. The interpolation algorithm is discussed in detail in sub section 3.4. This array can be assumed as a container that holds virtual volume full of voxel information representing the entire scanned volume. Voxel dimensions are in-built in the structure itself

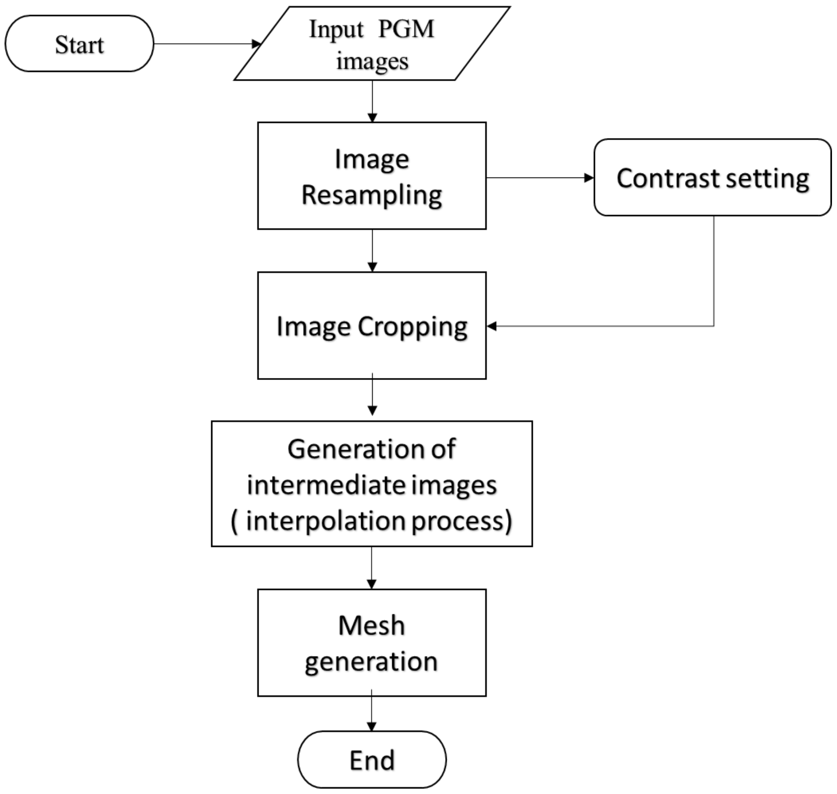


Figure 12. Flowchart of the adopted process.

The virtual volume generated is padded in all direction before moving on to mesh generation. Padding is necessary to avoid any open manifolds in the generated mesh. Mesh generation is done layer-by-layer from bottom to top. The output of this stage is a list of triangles containing vertex and normal coordinates. A special function takes the list of triangles and writes it into a .STL file.

3.2 Stage 1: Image processing

The images are resampled using bicubic interpolation as described in Section 2.3.1 . The objective is to create finer data by increasing the image resolution. Bicubic interpolation makes sure that the resampled data is continuous in nature (Figure 13). Contrast enhancement is also applied at this stage. An appropriate window width and window level has to be provided by the user. The new images are then cropped and inverted as per user requirement. The purpose is to purely reduce computation effort by discarding unimportant data.

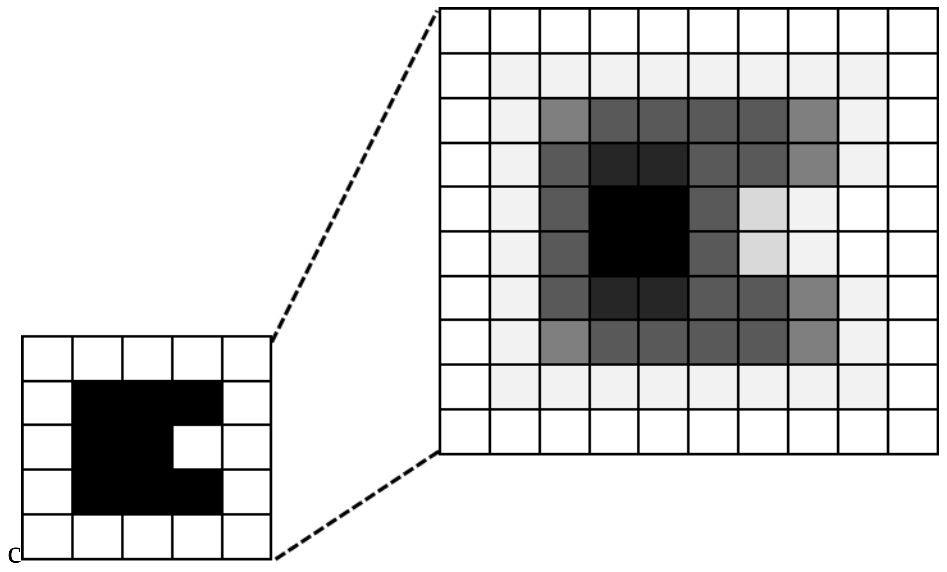


Figure 13. Schematic of Image resampling procedure using Bi-cubic interpolation

3.3 <u>Stage 2</u>: Image Interpolation

Each pixel on an image represents a rectangular voxel averaged over a certain pitch based on the CT scanner parameters. Thus, it contains information regarding the entire volume of the voxel rather than the volume density at that particular location. This property of the pixel is exploited in generating intermediate slices between consecutive input slices, matching the fabrication layer thickness. The motivation in generating intermediate slices is taking advantage of all the available information for generation of a more accurate 3D model.

The number of intermediate slices is determined by comparing scan resolution of the input images with the desired resolution dependent on 3D printer resolution. If *zScan* is the resolution of the input images (CT scan's vertical resolution), *zDesired* is the desired resolution (fabrication layer thickness) and *N* is the of input images, then total number of images created (including the input images) is given by

$$Total\ images = \left[\left((N-1) * \frac{zScan}{zDesired}\right) + 1\right]$$

The values after the decimal points are ignored as number of images cannot be a fraction. If the scan resolution is not completely divisible by the desired resolution, it may result in a case where the location of the input images would not match the location of the new images. However, the interpolation algorithms take care of this problem, as interpolation functions are continuous in nature.

The grayscale value of each pixel denotes the material density at that location in space. As the given data is continuous in the direction of the slices, interpolation of grayscale value helps to obtain the original data for the entire volume in the scanned space. This is done by employing pixel wise interpolation of the slices. It is expected that by doing so, contour interpolation between two consecutive slices can be achieved. Two different interpolation schemes are executed viz. piecewise linear and piecewise cubic spline interpolation.

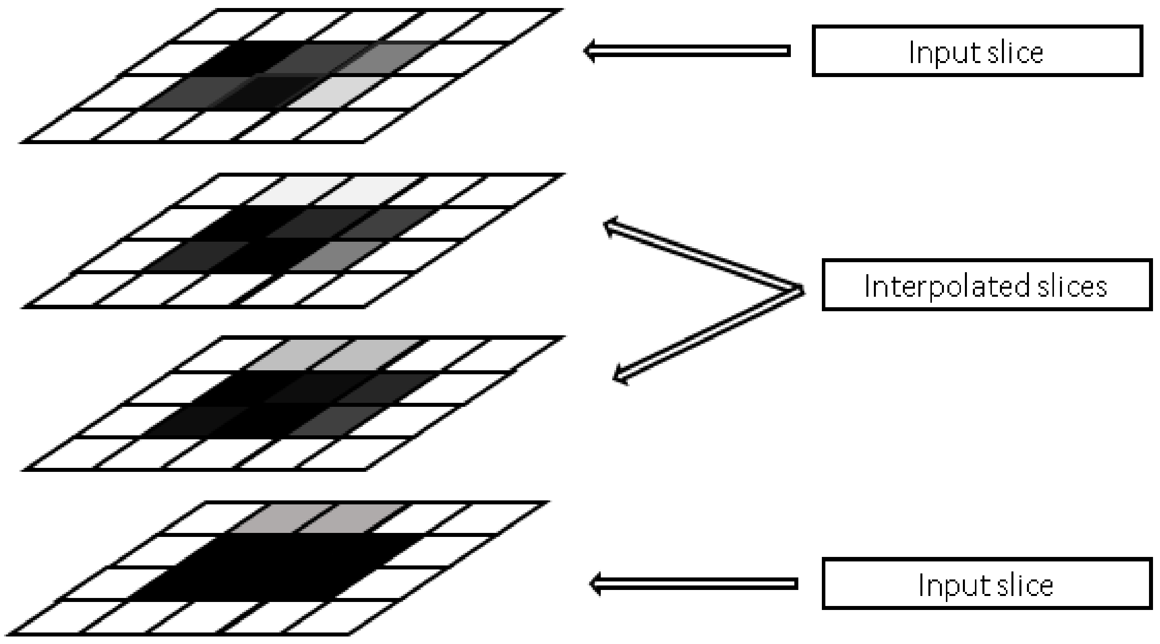


Figure 14. Interpolation process

Suppose for a set of data points $[x_i,\ y_i]$ where I = 0, 1, 2….n needs to be interpolated. In case of this thesis, x_i would correspond to the location of the input slices and y_i would correspond to the intensity values at x_i of a given pixel. There are multiple ways of interpolating between the data points. Two interpolation schemes have been adopted: Piecewise linear interpolation and piecewise cubic interpolation.

3.3.1 Piecewise linear interpolation

Linear interpolation is a method of curve fitting which uses linear polynomials to construct new data points between two input data points. The order of the polynomial obtained is 1 and it is continuous in C^0 Domain. If (x_i, y_i) and (x_{i+1}, y_{i+1}) are the coordinates of the known points, then the value of any point x can be given by

$$y = y_i + \frac{(y_{i+1} - y_i)}{(x_{i+1} - x_i)}(x - x_i)$$

The interpolation algorithm interpolates by cross checking the distance of the slice in consideration with the location of the input slices. This ensures the pitch between each slice remains constant irrespective of the location of the input slices. Only the first and the last slices remains unchanged as interpolating below and above them is not possible.

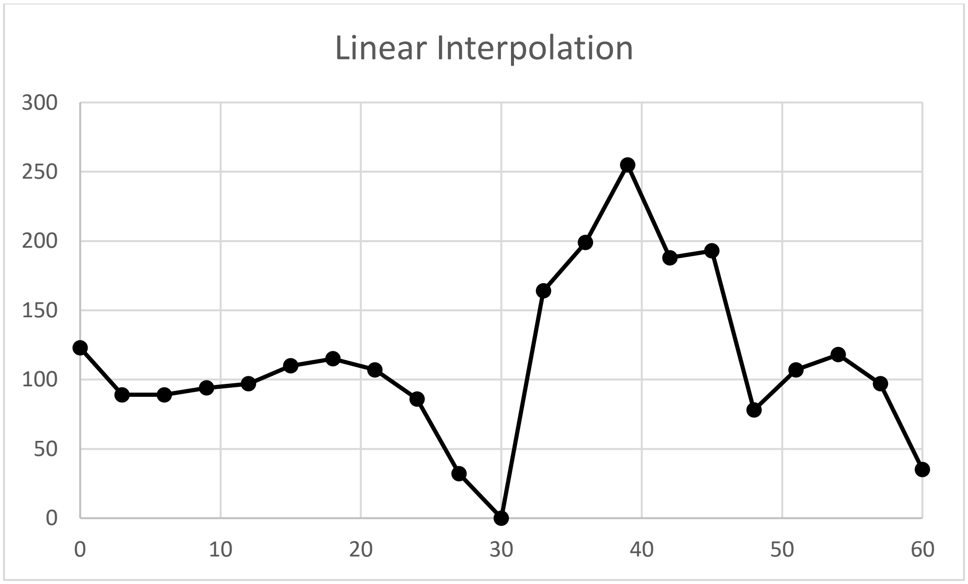


Figure 15. Curve obtained from linear interpolation of a data set

The interpolation algorithm starts by taking in two consecutive input slices in the memory at an instance. The algorithm then starts scanning them pixel-by-pixel beginning from the top left corner. Value of a pixel in the lower slice is compared to the value of the same pixel in the upper layer. The slope for linear interpolation for that particular pixel is obtained by comparing the difference in the pixel value by the distance (resolution) between the two slices. The pixel value in the lower input slice is considered as the basic value. This slope is then used to calculate the pixel value in the intermediate slices. This is repeated for each pixel in those two slices. Once, each pixel in all the intermediate slices that can fit between the two input slices are interpolated, the algorithm moves on to the next set of consecutive slices until it reaches the last input slice.

3.3.2 Piecewise cubic interpolation

A piecewise cubic interpolation is an interpolation technique which uses a continuous cubic curve passing through all the grid points. There is a separate cubic polynomial defined for each interval given by

$$F(x) = \left\{ \begin{array}{c} f_1(x) = a_1x^3 + b_1x^2 + c_1x + d_1 \ \ldots\ldots for\ [x_0, x_1] \\ f_2(x) = a_2x^3 + b_2x^2 + c_2x + d_2 \ \ldots\ldots for\ [x_1, x_2] \\ f_3(x) = a_3x^3 + b_3x^2 + c_3x + d_3 \ \ldots\ldots for\ [x_2, x_3] \\ . \\ . \\ . \\ . \\ f_{n-1}(x) = a_{n-1}x^3 + b_{n-1}x^2 + c_{n-1}x + d_{n-1} \ \ldots\ldots for\ [x_{n-1}, x_n] \end{array} \right\}$$

Where the coefficients for each polynomials (a_i, b_i, c_i, d_i) needs to be calculated. Since there are *n* intervals and 4 unknown coefficients between each interval, 4*n* conditions are required to solve the equation.The first criteria is that the splines generated should be continuous in C^0 domain. Following this criteria 2*n* conditions can be derived as follows:-

$$f_i(x_i) = y_i \ \ and \ \ f_i(x_{i+1}) = y_{i+1}$$

The splines generated should be continuous in C^1 and C^2 domain to ensure smooth curve giving yet another 2(*n*-1) conditions as follows:-

$$f'_{i-1}(x_i) = f'_i(x_i) \ and \ f''_{i-1}(x_i) = f''_i(x_i)$$

Two more conditions can be satisfied using wither of these two assumption

1. The spline is considered as a “natural” spline i.e. there is no change is slope at the end points of the splines and the spline can naturally continue with the same slope.
$$f''_0(x_0) = 0\ ; \ f''_{n-1}(x_n) = 0$$
2. The spline is considered as a “clamped” spline wherein the first derivatives at the end points are equated to known values.

For the sake of this thesis, the splines are considered as “natural” splines are the values of first derivative of polynomials at end points are not known. It is safe to make this assumption. Figure 16 shows a cubic spline interpolation of the data$[x_i,\ y_i]$. Comparing with Figure 15 , it can be inferred that the cubic spline interpolation gives a much smoother curve as expected. However the curve can undershoot or overshoot in some cases, where it needs to be truncated to either 0 or 255 in case of PGM images. As the extreme values are anyways segmented out in this case, truncation of values to extreme grayscale value does not create any problem.

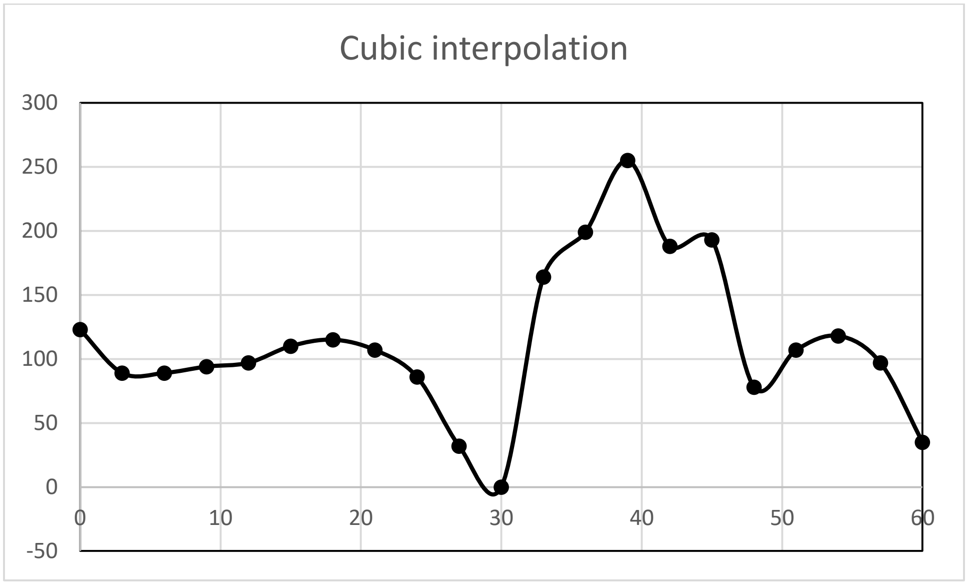


Figure 16. Curve obtained from piecewise cubic interpolation of a data set

Algorithm for cubic piecewise interpolation developed by Tino Kluge [28] and licensed under the terms of GNU General Public License was incorporated in the original source code. For each pixel, a vector of input slice location and grayscale intensity value at those location is created. This vector is used as the input for spline generation. The grayscale intensity values of new intermediate slice is then interpolation based on their location in space. Once the interpolation is done on one pixel throughout, the spline is discarded and the algorithm moves to next pixel and repeats the entire process. Thus at any given time, the memory requirement of the algorithm is not much.

3.4 Stage 3: Mesh Generation

After, preparing the virtual voxel, mesh generation is carried out using marching cube algorithm proposed by Lorenson [29]. The marching cube algorithm takes advantage of the fact that either a pixel is inside a surface or outside a surface. The algorithm starts by taking two slices in memory. A virtual cube is generated between the two slices, wherein the lower four vertices are assigned values from the lower slice and the upper vertices from the upper slice (Figure 17). The orientation of the virtual cube is predefined. Figure 17 shows the oriemtation of the virtual cube.

The assigned vertex value is compared to a threshold value or range defined by the user, to check if the vertex is inside or outside the surface. A special hex flag is designed to keep a track of which vertex is inside or outside the surface using binary operations on the flag. If the vertex is inside the surface, the flag for that particular vertex is set to 1, else it is 0 by default.

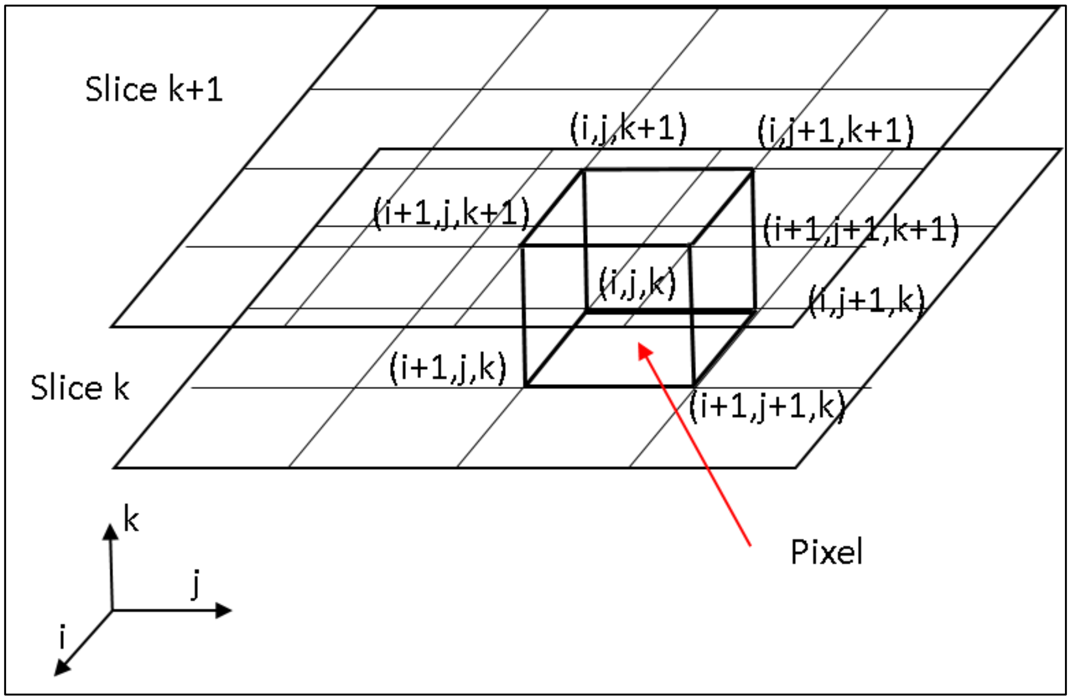


Figure 17.Virtual cube generated in marching cube algorithm [29]

As each vertex can have two states (0 or 1) and there are 8 vertices, a total of $2^8 = 256$ possible scenarios of how the triangles can be oriented. Using two different symmetries, 256 cases can be reduced to a total of 14 fundamental cases as shown in Figure 18. Based on the value of hex flag, triangular facets are generated in the cube using a look-up table containing information regarding triangle orientations for the possible scenarios. A look up table is a table containing list of edge numbers on which the triangle's vertices lies on, for all triangles and for all possible combinations. Look up table containing information regarding all the 256 cases can be found along with the code in Appendix A. The coordinates of a triangle's vertex needs to be linearly interpolated from the edge's vertices depending on the threshold greyscale value used for segmentation. Using the triangle vertices, normal is calculated which represents the triangle orientation. The orientation of the triangle is very important since it decides whether the facet in consideration is a part of inner or outer surface

Once the cube scans through all the pixels of a given set of slices, it moves on to the next set covering the entire volume with triangular facets. The generation of tessellated faces is continuous. It should be noted that the smoother the transition between two consecutive slices, the smoother the finish of the .STL file.

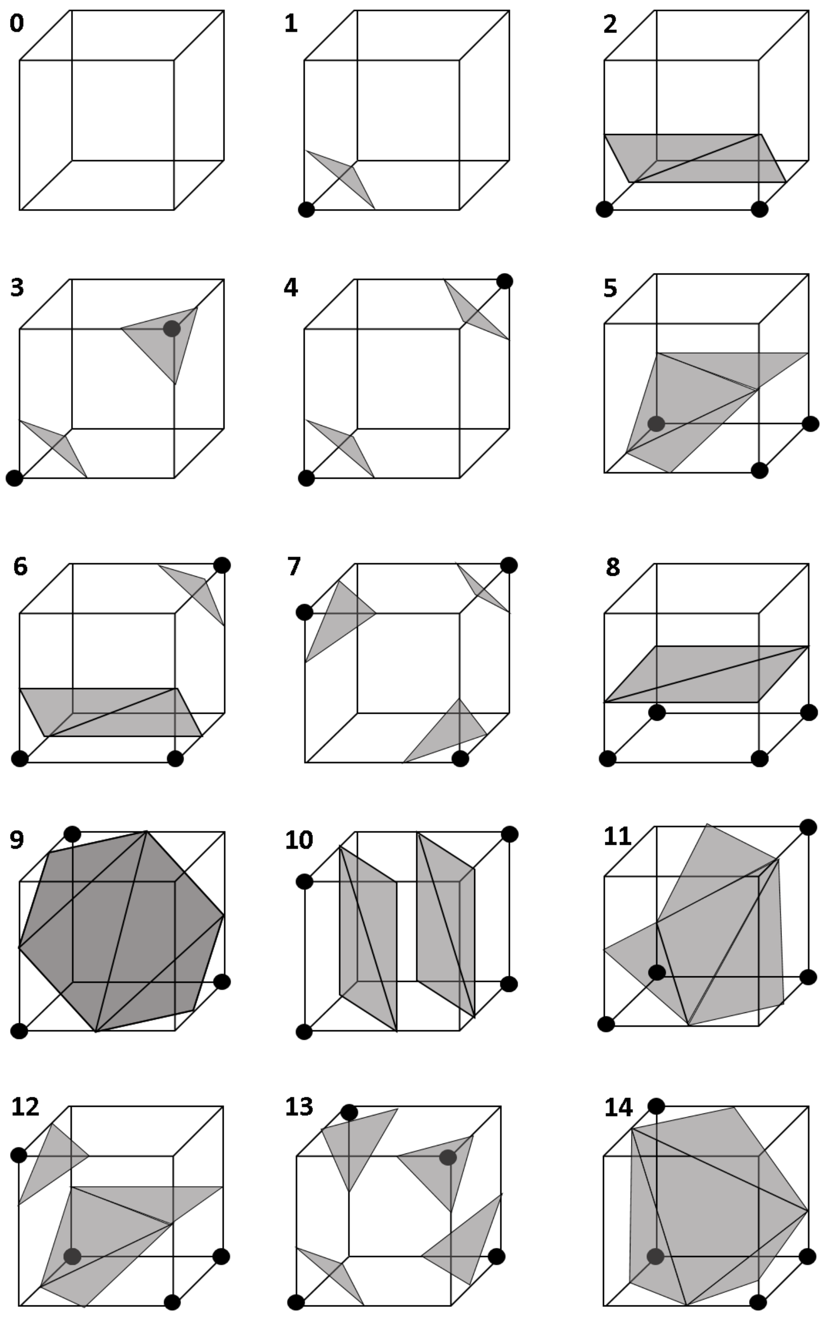


Figure 18. Cases possible in triangulating cube [29]

Since the slices are generated as such to match the 3D printer resolution, the mesh generated is expected to provide closed contours to the slicing software, at the location of the fabrication layers as shown in Figure 19. The motivation in doing so is it can prevent errors that otherwise could be generated during slicing as described in section 2.4.2 . It helps in reducing numerical round-off errors introduced due to interpolation of contour points from facet coordinates. It attempts to favor the type of intersection where the edge lies on the slicing plane. However, the final result depends on the type of slicing algorithm used by the slicer.

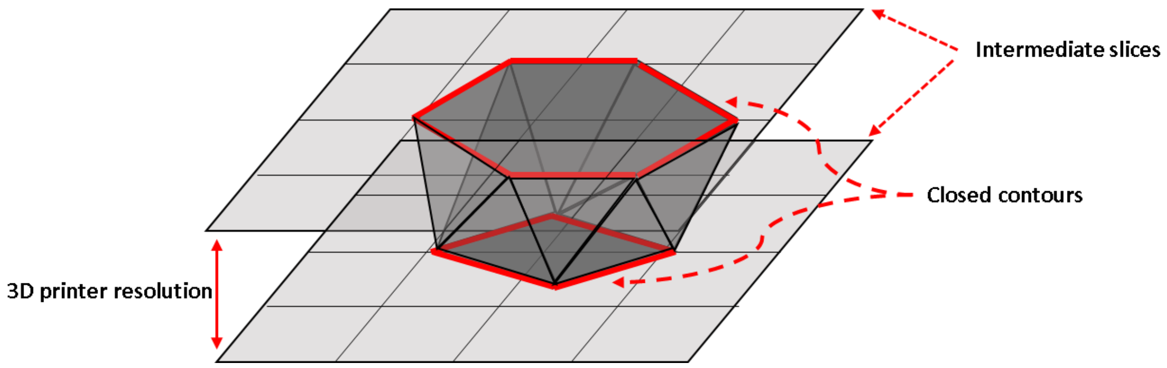


Figure 19. Schematic of expected closed contours at the location of slicing planes

Chapter 4 Experiment Description

This section describes the experimental data used to validate the proposed method, followed by parameters used for different image processing techniques used in the proposed method. Finally it discusses various setups derived by changing different settings used for conversion of 2D scan data to 3D model, to check the effect of each stage on the surface finish of the resulting mesh.

4.1 Experimental Data

The algorithm is tested on a head CT scan data. The data consists of 21 slices oriented in coronal plane. The image resolution of the data is 512 x 512 pixels. The grayscale values are truncated from 16- bit to 8-bit i.e. they range from 0 to 255. Here, 0 represents the smallest value (black color) and the value 255 is the largest value (white color).

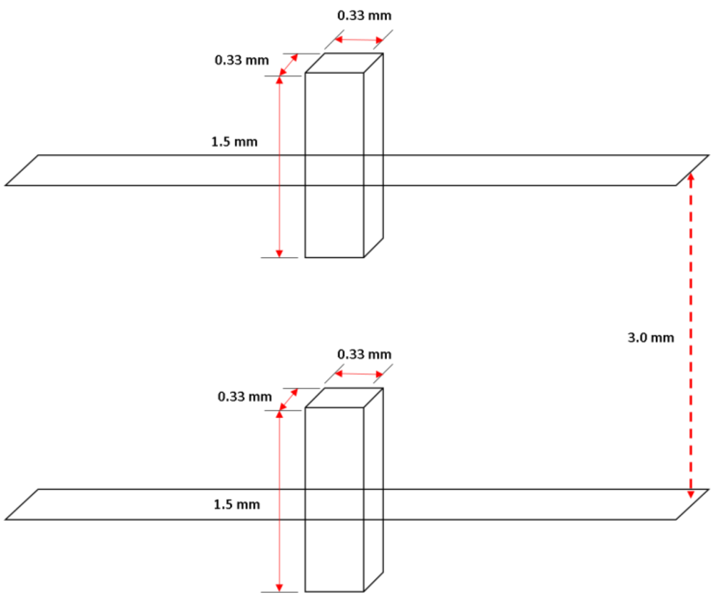


Figure 20. Spatial resolution of experimental data

The spatial resolution (in mm) is 0.33 mm x 0.33 mm. Slice thickness is 3.0 mm with voxel length being 1.5 mm. There is 1.5 mm of unknown data between two consecutive slices (Figure 20). The actual PGM images used for the experiment are shown in Figure 21.

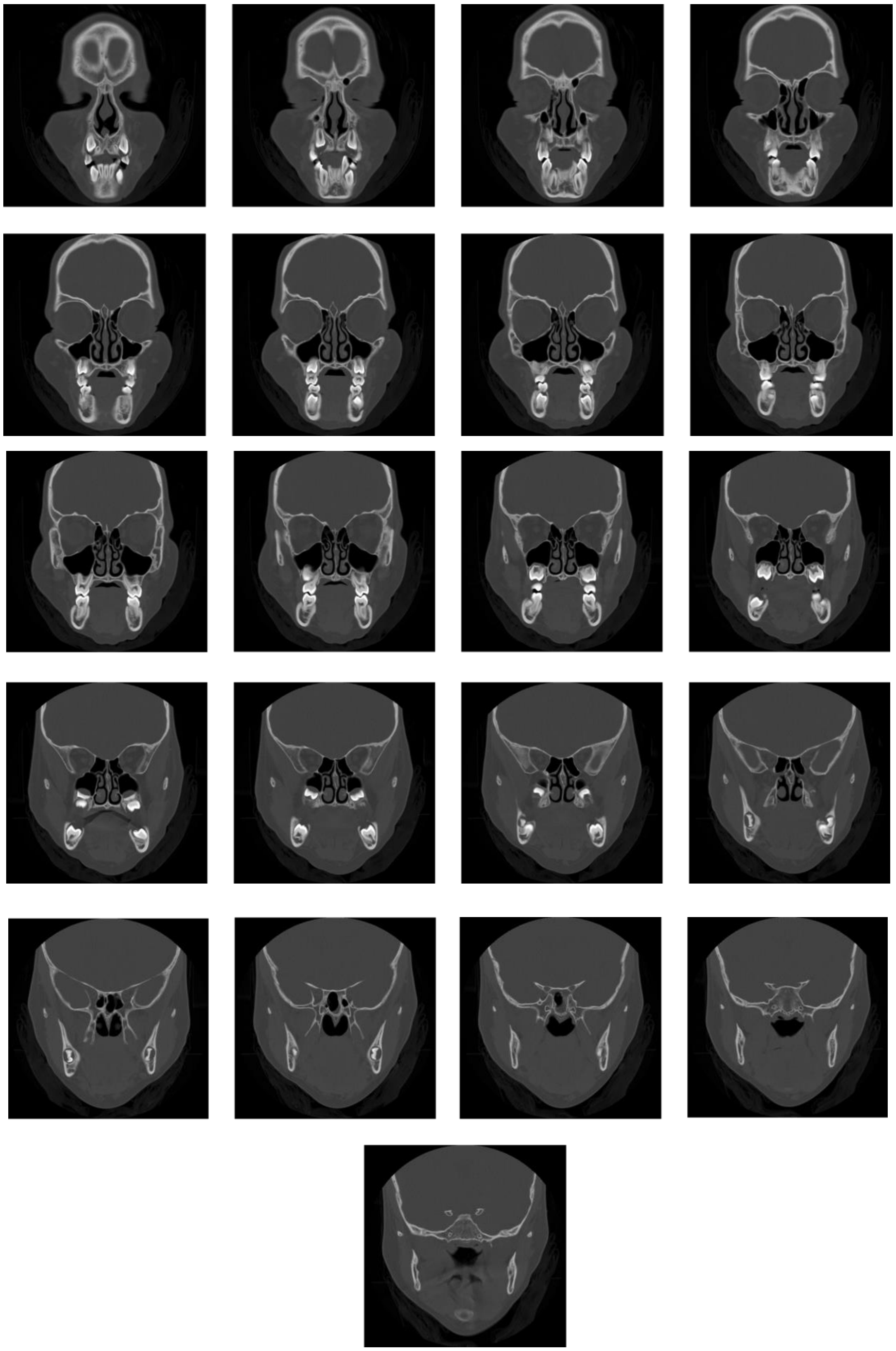


Figure 21. CT scan of a Human head (experimental data)

4.2 Objective

An attempt to 3D print the nasal passage has been made. The nasal passage is the proximal portion of the passages of the respiratory system, extending from the nares to the pharynx; it is divided into left and right halves by the nasal septum and is separated from the oral cavity by the hard palate. Figure 22 and Figure 23 shows the anatomy of nasal cavity in coronal and sagittal views respectively.

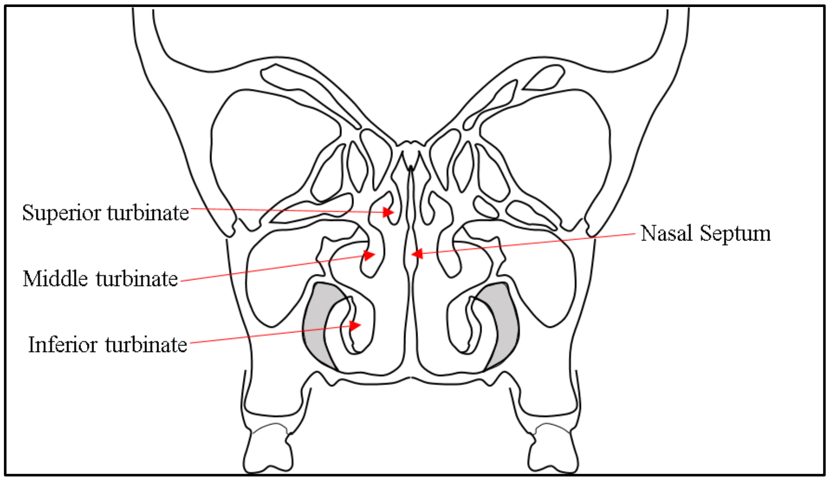


Figure 22. Diagram of nasal cavity in coronal plane

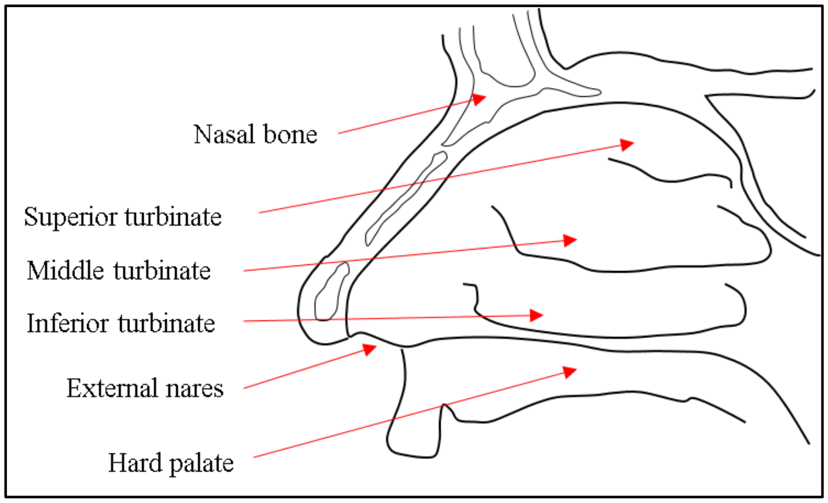


Figure 23. Diagram of nasal cavity in sagittal view

A number of studies related to dimensions of the nasal cavities have been conducted [30], [22], [31]. These dimensions are subject to change based on patient’s age, ethnicity, gender etc.

[22] has successfully proposed a standardized model based on data received from 30 different patients.

4.3 Parameters

As described in Section 3.2 , the images obtained from CT scan needs to be processed further to obtain an accurate model. Parameters for processing at each step should be selected with careful consideration. This sub-section will go through the parameters selected for the purpose of this experiment.

The images are resized 4 times its original size using bicubic resampling technique. It should be kept in consideration that upscaling the images increases the computational effort required and the size of the produced .STL file. Thus, image magnification parameter should be decided based on area of interest and the smallest feature that needs to be extracted. A balance between these two parameters should be struck to avoid making the .STL file too huge.

The images are cropped as the area of interest is much smaller than the entire image. Thus, computational effort can be saved by cropping the images to accommodate just the nasal cavity. The paranasal sinus are also omitted. The dimensions of the cropped images are 150 x 90 pixels with an offset of 215 columns from x-axis and 185 rows from z-axis (Figure 24)

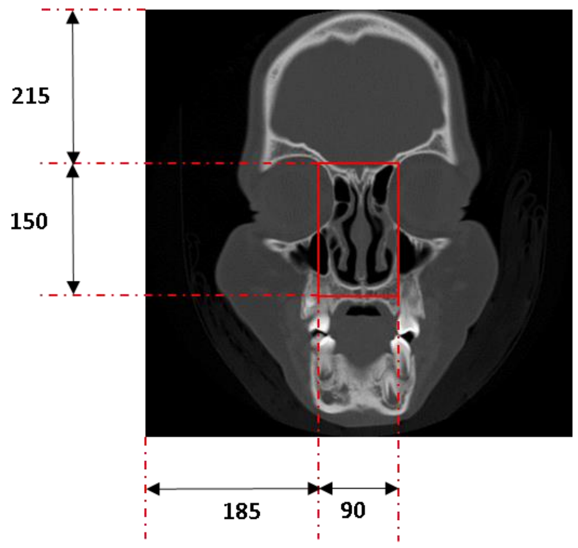


Figure 24. Cropping parameters used for the experiment

Since the volume to be extracted is an air cavity, the images are inverted such that grayscale value of air is highest. The grayscale value of soft tissue surrounding the nasal passage is around 30-40 without inversion. The threshold value to segment the images is 219 after inversion found through experimentation. The CT window width in terms of grayscale intensity value (0-255) for contrast enhancement was selected to be 175 and window level is 87.

The printer used for 3D printing the .STL file was Lulzbot MINI. The nozzle diameter of the printer is 0.5 mm and vertical resolution of the printer is 0.38 mm. The vertical resolution for the intermediate slices (*zDesired*) was matched with the print resolution of the 3D printer (0.38 mm). The filament used for printing purpose was PushPlastic PLA (polylactide) with a filament diameter of 3 mm. The filament meets the quality standard of +/- 0.05 mm. [32].

4.4 Setups

To see the effects of various image processing techniques on the quality of 3D printed model, .STL files were obtained by different Setups described in Table 2. Each Setup is derived by either inclusion or exclusion of one of the steps described in the proposed method while producing 3D printable models from the 2D input mages. The Setups are designed to check the effect of resampling, contrast enhancement, type of interpolation scheme used to generate intermediate slices in Stage 2, and image segmentation before mesh generation on the quality of mesh generated. A comparison was made between the models generated by these Setups to find the most optimum method. The most basic Setup comprises of using data with original resolution excluding the resampling step, without contrast enhancement using linear interpolation scheme to generate the intermediate images. The images were segmented prior to generating the mesh in this Setup.

Effects of resampling the data is tested in Setup A1 by resampling the raw data 4 times the original resolution. Rest of the parameters were kept the same. Similar to basic step, the images were segmented prior to mesh generation. In essence, Setup A1 varies from the basic Setup owing to the fact that the images were resampled.

Setup B1 examines the effects of contrast enhancement of the raw data on the mesh generated. The images are not resampled but a contrast enhancement is applied on the input images. Similar to Basic Setup, the images were segmented prior to mesh generation.

It is expected that mesh quality should improve if the mesh is generated using grayscale values, instead of binary data obtained by segmenting the images. Setup C1 attempts to validate the improvement in mesh quality, by not segmenting the images prior to mesh generation, and generating the mesh using grayscale data.

The effects of different types of interpolation schemes can be concluded by comparing Setup D1 with the basic Setup. Setup D1 implements cubic interpolation scheme to generate intermediate images, instead of linear interpolation scheme used in all the previous Setups. The images are not segmented prior to mesh generation in this Setup.

Setup number	**Resampling**	**Contrast Enhancement**	**Linear**	**Cubic**	**Threshold before Mesh generation**
Basic Setup	✗	✗	✓	✗	✓
A. Effect of resampling					
A1	✓	✗	✓	✗	✓
B. Effect of Contrast enhancement					
B1	✗	✓	✓	✗	✓
C. Effect of applying threshold during Mesh generation					
C1	✗	✗	✓	✗	✗
D. Effect of type of interpolation scheme					
D1	✗	✗	✗	✓	✗

Table 2. Description of Setups used for experiment

Chapter 5 Results

The objective of the proposed method is to generate precise 3D physical models from 2D medical imaging data. This Section will examine the results obtained from the experimental Setups, and discuss the ability of the proposed method to produce the desired outcome. This includes a discussion on contours obtained from the generated intermediate slices, effect of techniques like image resampling, and contrast enhancement on the surface mesh quality of the generated 3D model. This is followed by examining the effect of segmention process, and type of interpolation scheme (for generating intermediate images) on the mesh quality of the generated 3D model. The Section concludes by discussing the quality of the generated .STL fileformat, and physical 3D printed model.

5.1 Basic Setup

Interpolation of the original data results in contour evolution. Figure 25 shows the intermediate slices generated between input slices (denoted by red boxes). The results obtained from Basic Setup are promising as on inspection not many artifacts were present. A comparison between the contours obtained from the intermediate images, and contours extracted from the .STL model using an open source slicing software Slice3r has been made (Figure 26). It can be inferred that the mesh generation process is geometrically accurate since the contours are easily replicated before and after mesh generation. The objective of supplying closed contours to the slicing software seems to be fulfilled as there is a high resemblance in the contours obtained from intermediate slices before mesh generation and the contours obtained from the slicing software (Figure 26).

In essence, using grayscale interpolation to generate intermediate slices can produce satisfactory contour evolution between the input slices. The need to use complicated contour interpolation methods is eliminated.

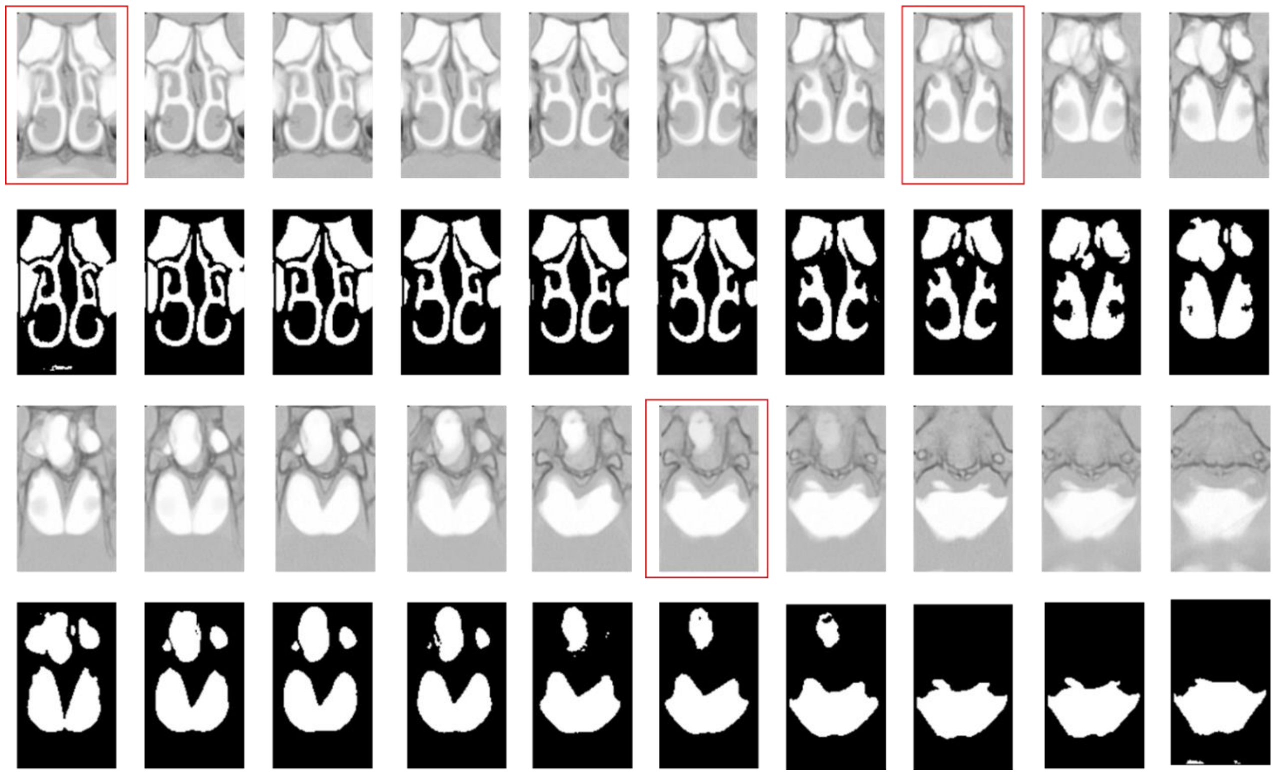


Figure 25. Intermediate slices and their extracted contours obtained from Setup described in basic Setup. The red box denotes the location of input slices.

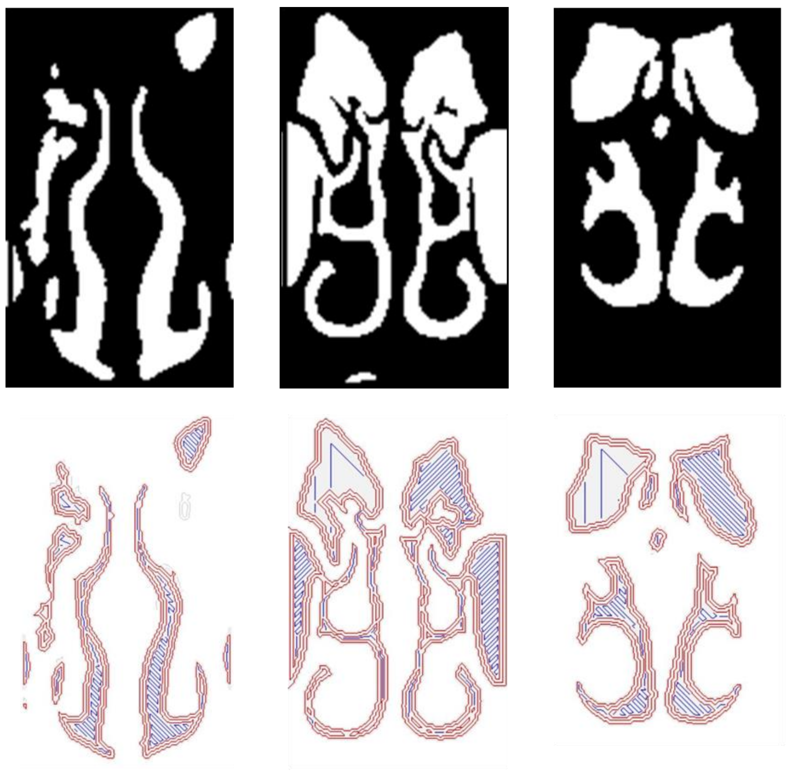


Figure 26. Comparison of contours generated by interpolation with contour extracted by Slicer3r at same height

The mesh generated from the Basic Setup shows jaggedness as seen in Figure 27. To improve the quality of mesh, slice thickness was decreased 3 times. The mesh obtained from high resolution showed improved smoothness in the plane perpendicular to the images. However the mesh was still grainy. This jaggedness is a result of combination of low resolution of images and the fact that the images were segmented prior to mesh generation. The value of the pixel can either be 255 or 0 after applying threshold. Marching cube applied to these images will produce facets with vertices located at a fix position with respect to the two pixels in consideration. It is expected that, linear interpolation between two pixels in the plane of slices can produce smoother finish.

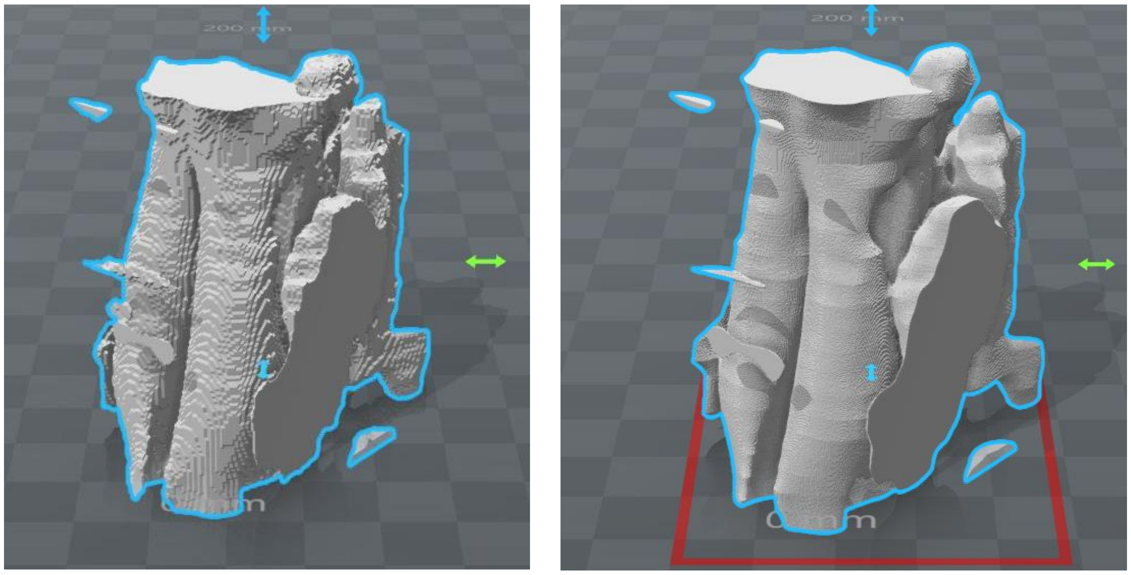


Figure 27. .STL file generated for basic Setup using (a) regular vertical resolution and (b) High vertical resolution

5.2 Effect of Image resampling

Setup A1 was designed to determine the effect of resampling procedure on the final mesh quality. A bicubic resampling of the input images proved to be effective as the contours generated are much smoother (Figure 28). Resampling removes bumpiness from the contours. The curves are much smoother and it is reflected in the surface finish of the generated mesh.

Figure 29 and Figure 30 shows a comparison between mesh generated from the Basic Setup with no resampling and Setup A1 with resampling. It can be concluded that there is a definite improvement in surface smoothness. The mesh is less grainy comparatively. However, the surface is still not smooth enough. Also, small artifacts are produced by the new generated data as shown in Figure 30. These artifacts are local in nature and does not span more than a pixel length. In most cases, these artifacts will be completely ignored by the slicing software as they are simply too small to be printed. These artifacts are produced when value of a small group of pixels is either too high or too low as compared to ithe pixel values in its neighborhood.

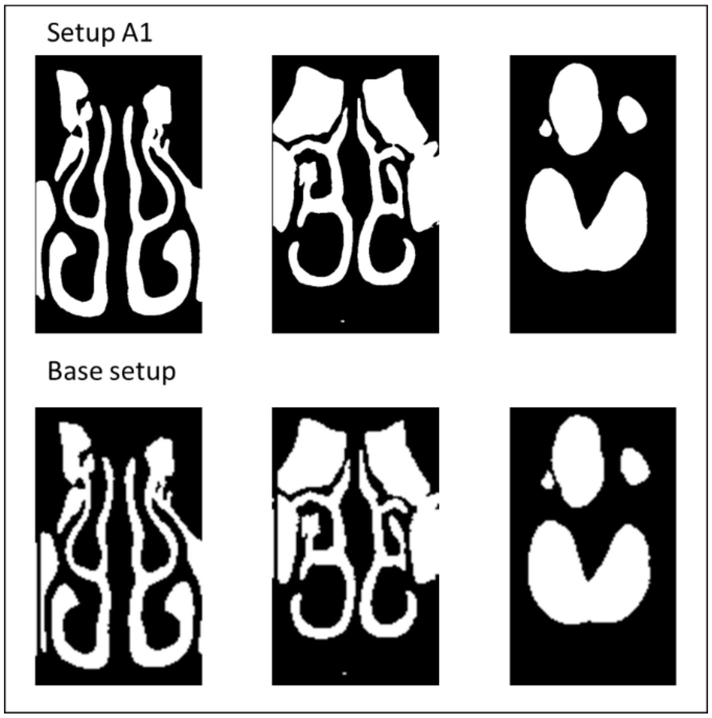


Figure 28. Comparison of contour smoothness between contours obtained after image resampling (Setup A1) and before image resampling (basic Setup)

If *n* numbers of output slices, then resampling each slice by 4 times will result in 4 x n times the memory requirements and computational effort due to pixel wise operations involved. The size of the .STL file shows an increase of approximately 600% for resampling the given data by 4 times. The user should strike a balance between the memory requirements and the surface finish of the .STL file. In case of high resolution images, the images can be cropped to focus on area of interest for efficient memory management.

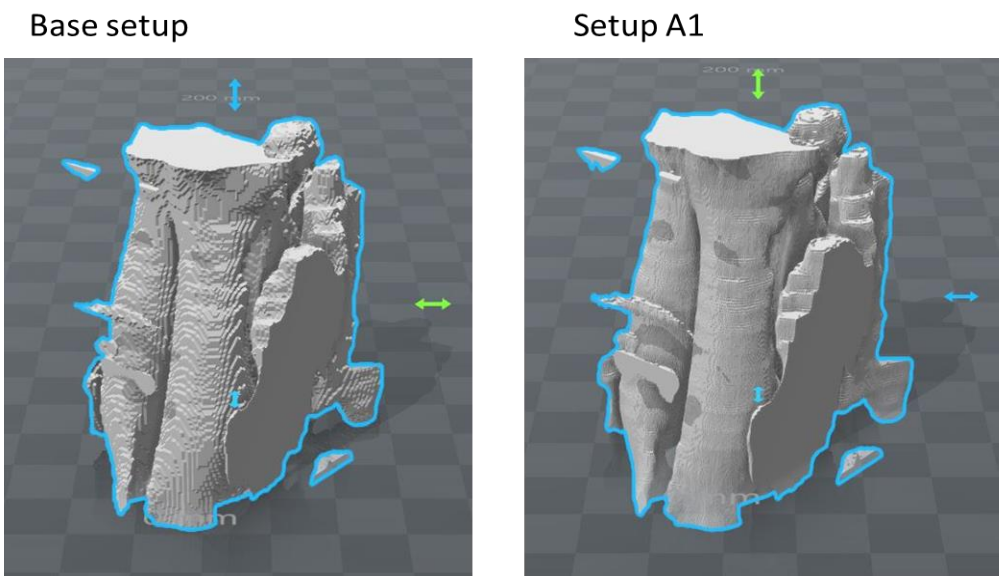


Figure 29. .STL file generated from (a) Basic setup and (b) setup A1

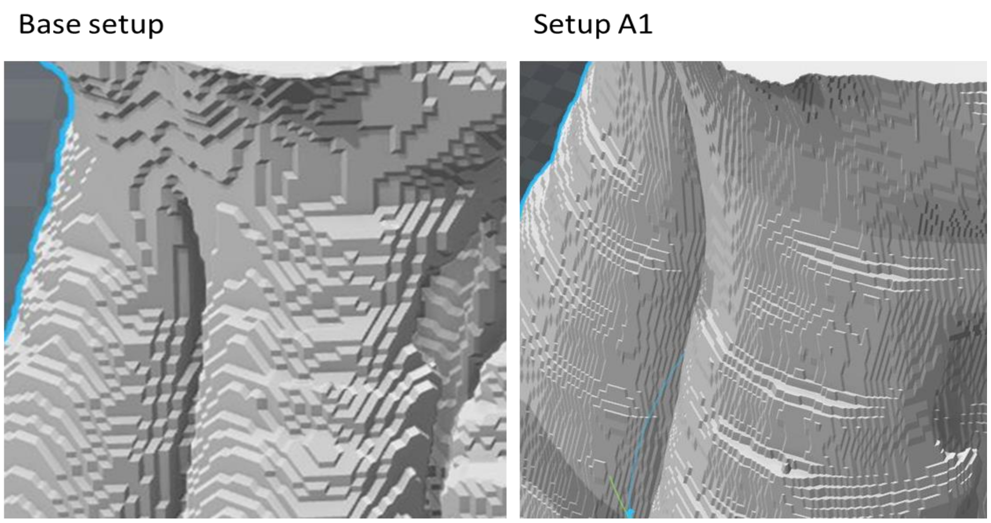


Figure 30. Difference in surface smoothness of mesh generated before (basic setup) and after image resampling (setup A1)

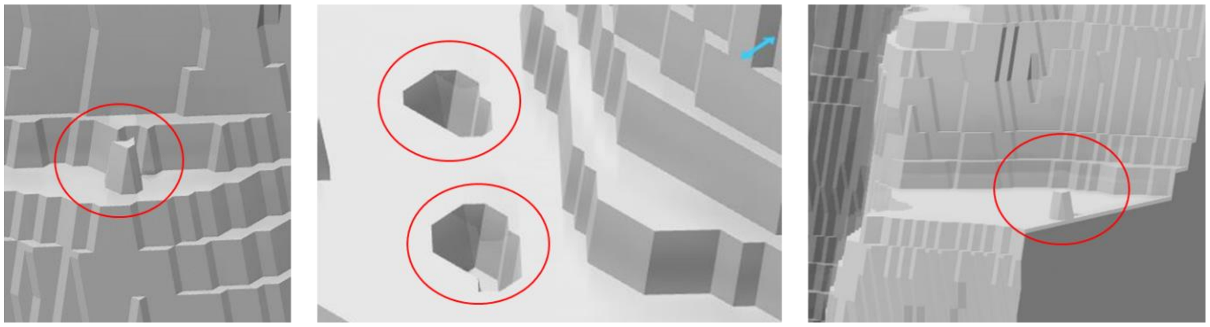


Figure 31. Local artifacts produced in the mesh due to image resampling

5.3 Effect of contrast enhancement

As described in Section2.3.2 , contrast enhancement is a technique used to focus on particular structures by highlighting certain grayscale values more than others. A contrast enhancement operation was conducted on the original data to see its effect on the resulting meshes. The contrast between the soft and the hard tissues is enhanced (Figure 32). However, there is no difference in the contours obtained from intermediate images (Figure 33). The .STL mesh shows no improvement as compared to basic Setup i.e. without any contrast enhancement. It can be safely concluded from Figure 33 and Figure 34, contrast enhancement does not provide any improvement to the results and should be avoided to save computational effort.

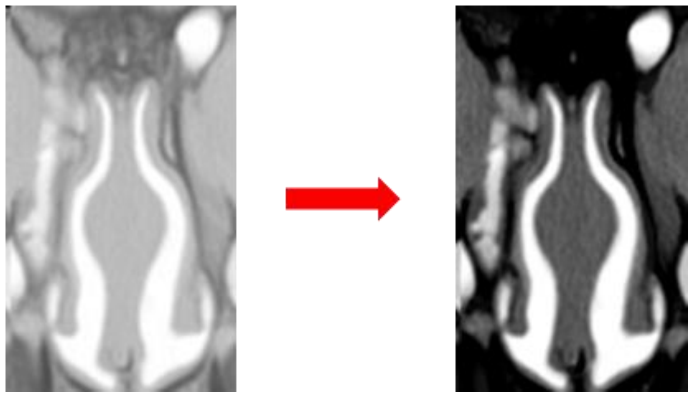


Figure 32. Effect of contrast enhancement on a CT scan image

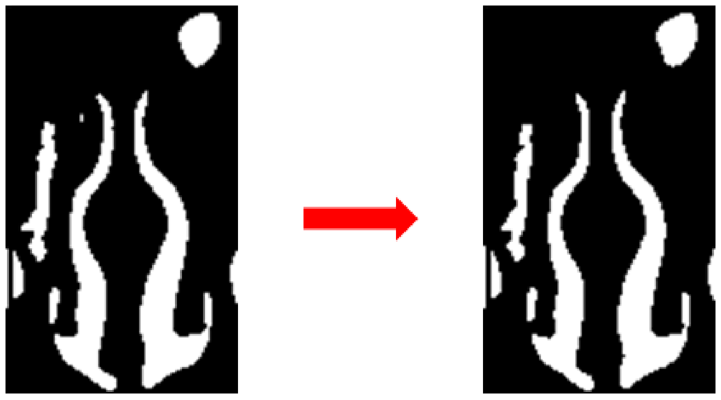


Figure 33. Effect of contrast enhancement on the resultant contours

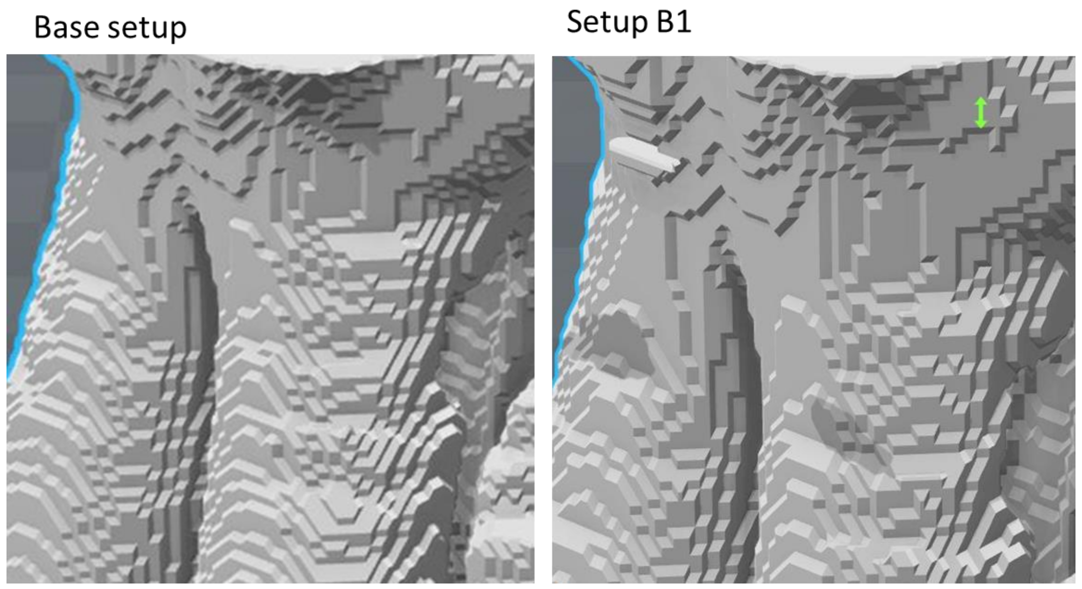


Figure 34. Difference in surface smoothness of mesh generated before (basic Setup) and after contrast enhancement (Setup B1)

5.4 Effect of mesh generation from grayscale value data

To improve the surface smoothness of the generated mesh, the original data of the images was kept intact (Setup C1). Marching cube was applied on these images, where surface information was extracted by comparing the grayscale value of a pixel with the user-defined threshold value. If the pixel’s grayscale value was below the given threshold value, it was considered outside the surface else it was considered inside the surface. This information of pixel being inside or outside the surface was stored in a hex flag. The coordinates of facet vertices were linearly interpolated from the original grayscale value data. The mesh generated showed improved surface smoothness with no jaggedness as previously seen in the mesh. Figure 35 shows a comparison between meshes generated using Basic Setup and Setup C1.

The contours obtained from slicing the .STL in Slice3r was compared with similar contours obtained from the Basic Setup (Figure 36). The contours are smoother with no jagged edges. It can be concluded that image segmentation prior to mesh generation reduces the quality of the mesh generated and hence should be avoided.

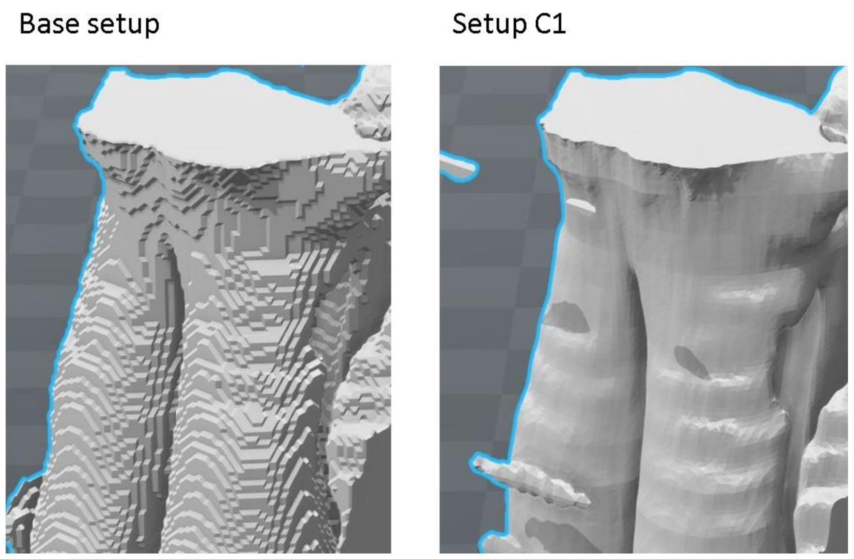


Figure 35. Difference in surface smoothness of mesh generated regular resolution basic Setup (left), and Setup C1 (right)

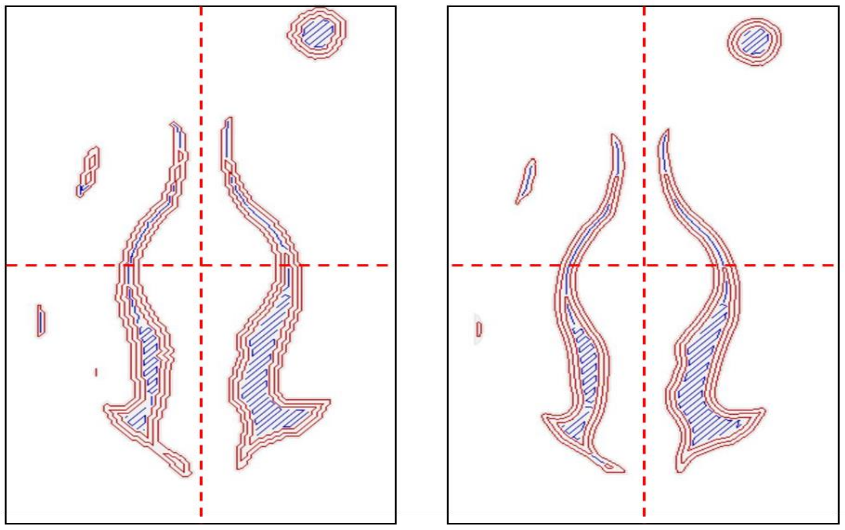


Figure 36. Contours extracted by Slicer3r for the same slice showing jagged curve for basic Setup (left) and smooth curves for Setup C1 (right)

5.5 Effect of interpolation scheme

The mesh generated from images generated using linear interpolation scheme, in Setup A1, B1, C1, and the Basic Setup all showed irregularities in slope at the location of the original slices. As linear interpolation is only continuous in C^0 domain, discontinuity should be expected in C^1 and other higher order domains. Linear interpolation scheme only takes two slices in consideration and no information is conveyed from other slices above and below the two slices in consideration. This results in staircase effect at the location of the original data.

To overcome this problem, piecewise cubic spline interpolation scheme was adopted. Cubic interpolation scheme in continuous in all three C^0, C^1 and C^2 domains. Cubic interpolation already satisfies all the necessary conditions required for the smooth model. Piecewise cubic interpolation takes the data from all the given slice into consideration while interpolating the intermediate slices.

The effect of cubic interpolation is evident in intermediate slices shown in Figure 37. The images obtained from cubic interpolation seems sharper as compared to the ones obtained from linear interpolation, especially at the location of the original slices. The improvement in surface smoothness of the .STL files is evident from comparing the .STL files produced by Setup C1 and Setup D1 (Figure 38). The staircase effect seen in the generated 3D models has been eliminated to great extent, by switching to the cubic interpolation scheme. Small amount of irregularities in slope is still present at some locations. The source of these irregularities is unknown.

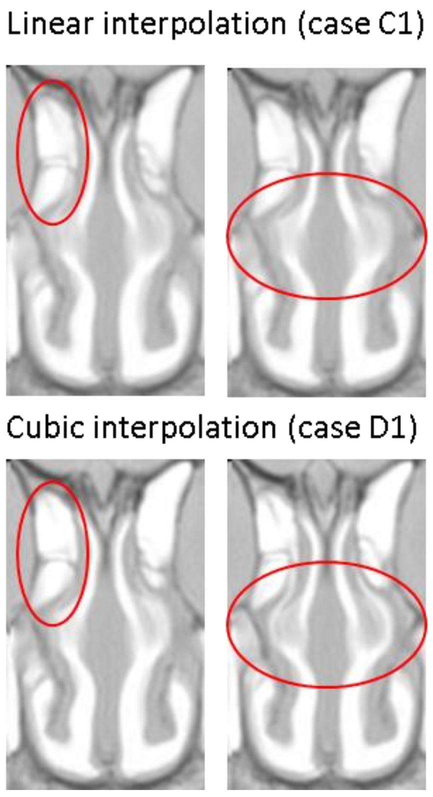


Figure 37. Comparison of image quality of intermediate images obtained by linear (Setup C1) and piecewise cubic (Setup D1) interpolation scheme

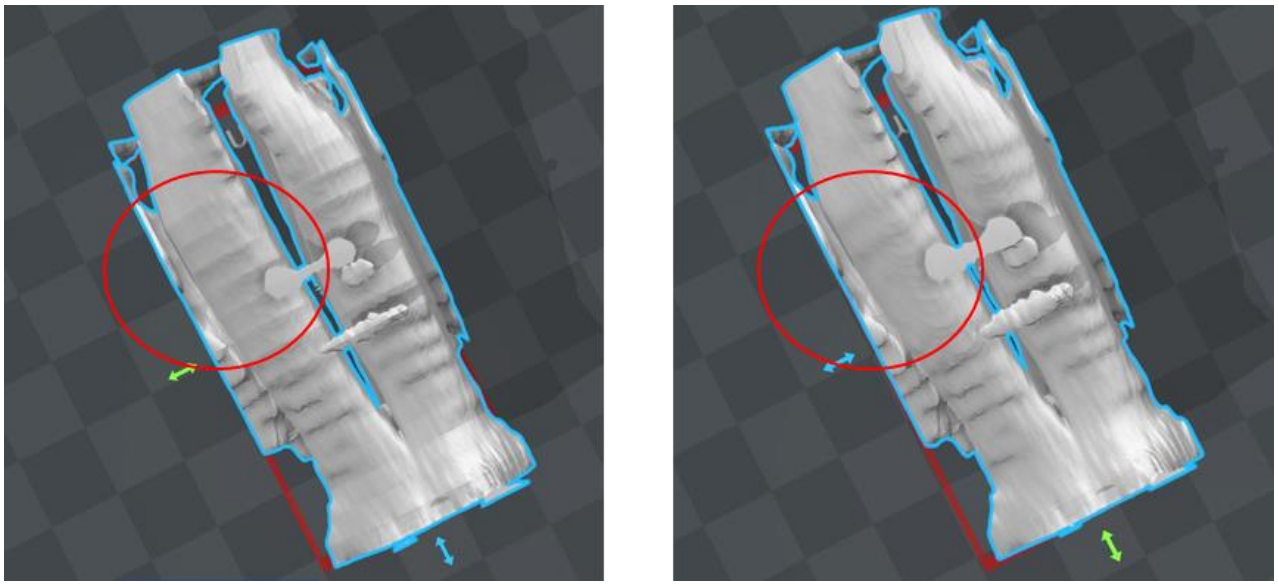
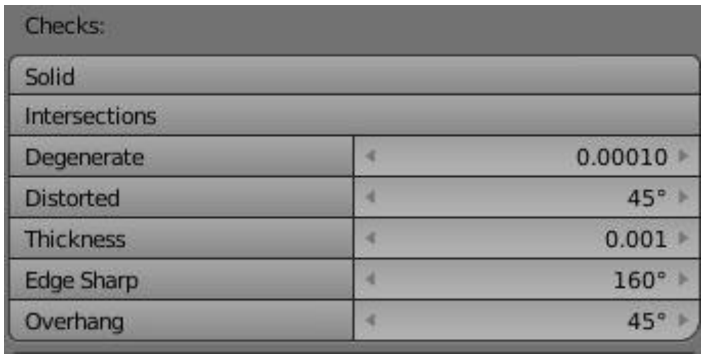


Figure 38. Comparison of existing staircase effect in mesh generated by linear (left) and cubic (right) interpolation scheme

5.6 Mesh quality

A mesh quality study was conducted on Blender using the 3D printing toolbox available in the software. The mesh was checked for the frequency of occurrence of degenerate faces, zero thickness geometry, intersecting faces, distorted faces, non-manifold edges, points and faces. The parameters used to assess the mesh is shown in Figure 39. The tolerances were set to default values defined in the software.



Checks:

Solid	
Intersections	
Degenerate	0.00010
Distorted	45°
Thickness	0.001
Edge Sharp	160°
Overhang	45°

Figure 39. Mesh quality parameters used as input in Blender 3D printing tool box for the purpose of mesh quality analysis

Results for each Setup (Table 3 - Table 7) shows the described methodology is capable of producing meshes of excellent quality. The errors are within 0.02 % and no post-processing of mesh is required. This eliminates the usage of mesh repair software as well as the errors introduced by them.

Basic Setup		
No of faces	495,124	
Errors	Count	% error
Non –manifold edges	0	0.0 %
Overlapping faces	0	0.0 %
Intersect faces	0	0.0 %
Zero faces	0	0.0 %
Zero edges	0	0.0 %
Thin faces	2	0.0008 %
Sharp edges	0	0.0 %

Table 3. Output of mesh quality analysis for basic Setup conducted on Blender

Setup A1		
No of faces	3,035,264	
Errors	Count	% error
Non –manifold edges	0	0.0 %
Overlapping faces	0	0.0 %
Intersect faces	6	0.0002 %
Zero faces	0	0.0 %
Zero edges	0	0.0 %
Thin faces	0	0.0 %
Sharp edges	0	0.0 %

Table 4. Output of mesh quality analysis for setup A1 conducted on Blender

Setup B1		
No of faces	492,896	
Errors	Count	% error
Non –manifold edges	0	0.0 %
Overlapping faces	0	0.0 %
Intersect faces	2	0.0004 %
Zero faces	0	0.0 %
Zero edges	0	0.0 %
Thin faces	0	0.0 %
Sharp edges	0	0.0 %

Table 5. Output of mesh quality analysis for Setup B1 conducted on Blender

Setup C1		
No of faces	461,878	
Errors	Count	% error
Non –manifold edges	26	0.0056 %
Overlapping faces	0	0.0 %
Intersect faces	0	0.0 %
Zero faces	3	0.0006%
Zero edges	0	0.0 %
Thin faces	96	0.02%
Sharp edges	68	0.0147%

Table 6. Output of mesh quality analysis for Setup C1 conducted on Blender

Setup D1		
No of faces	472,062	
Errors	Count	% error
Non –manifold edges	19	0.004 %
Overlapping faces	0	0.0 %
Intersect faces	0	0.0 %
Zero faces	0	0.0 %
Zero edges	0	0.0 %
Thin faces	68	0.0144 %
Sharp edges	55	0.01165 %

Table 7. Output of mesh quality analysis for Setup D1 conducted on Blender

It should be noted that Setup C1 and Setup D1 shows an increase in thin edges and sharp edges. This edges are located at the boundaries of the cropped images. Due to sudden termination of surface at the image boundary (Figure 40), some facets gets skewed resulting in thin faces and sharp edges. However, the number of thin faces are still within the acceptable limit.

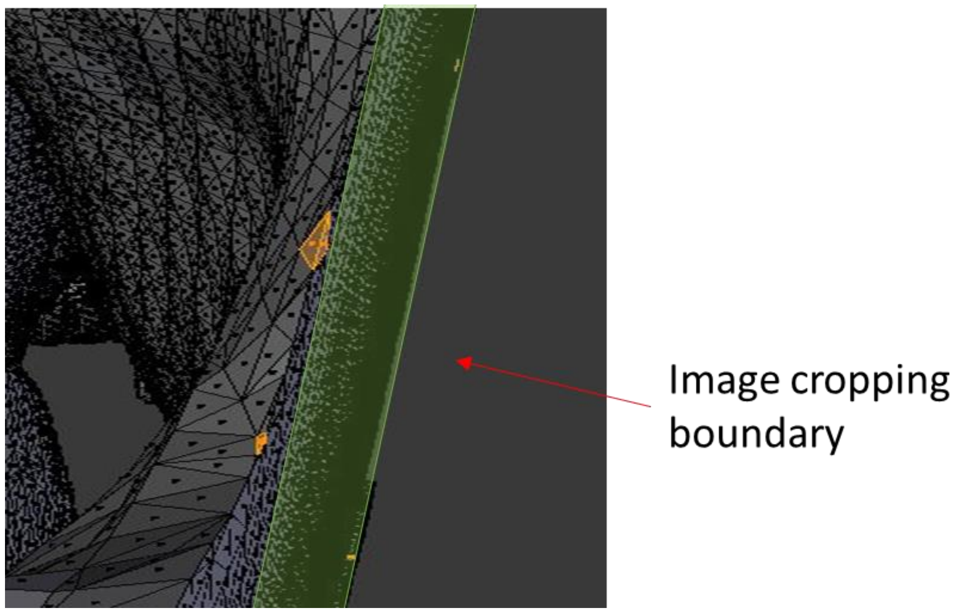


Figure 40. Location of thin edge in mesh develop using Setup D1

5.7 3D printed model

3D printed models were generated for the four Setups except Setup B1, since contrast enhancement did not show any improvement in mesh quality. Figure 41 - Figure 44 shows the 3D printed models obtained from all these Setups. No post-processing of the .STL file was done and the models were printed using slice3r slicing software on a Lulzbot mini printer.

The models generated follow the same trend as discussed in previous sub-sections. It is evident that the model generated from the basic Setup is very grainy. Resampling the images in Setup A1, improved the surface finish of the model to a certain extent, but it is still grainy. Setup C1 showed excellent surface, but showed prominent staircase effect at the location of orginial slices. Switching to cubic spline interpolation in Setup D1 eliminates the staircase effect present in all the previous models.

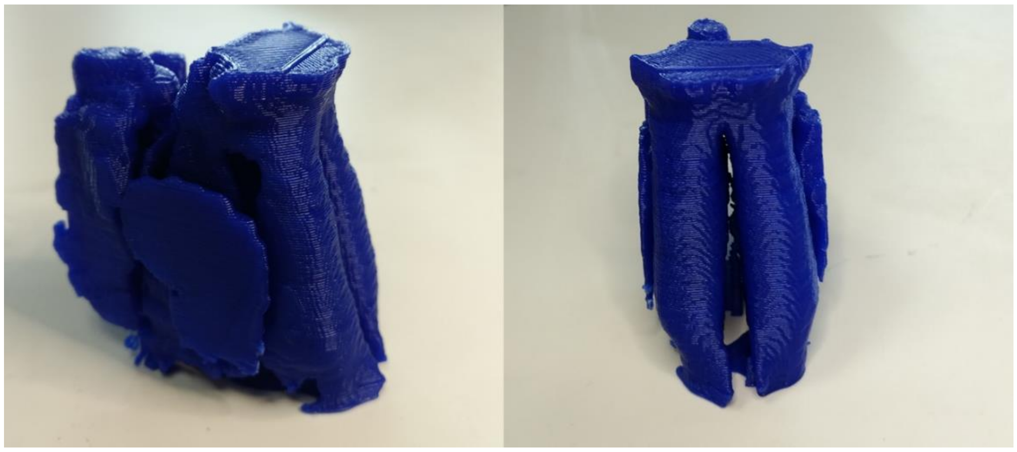


Figure 41. 3D printed model from basic Setup

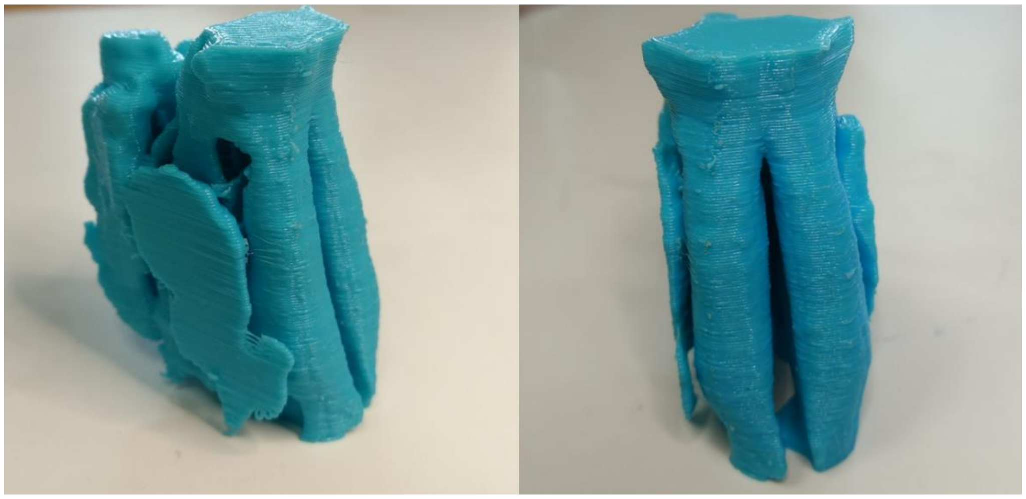


Figure 42. 3D printed model from Setup A1

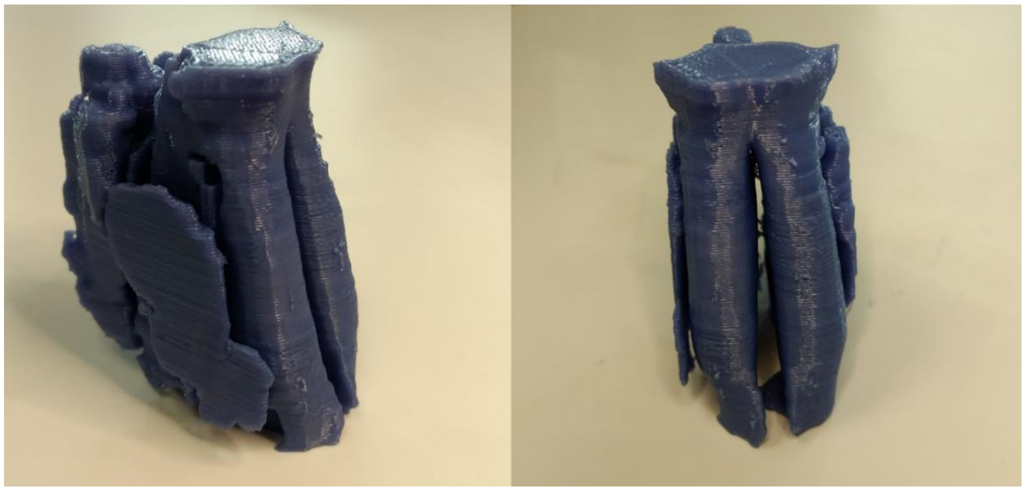


Figure 43. 3D printed model from Setup C1

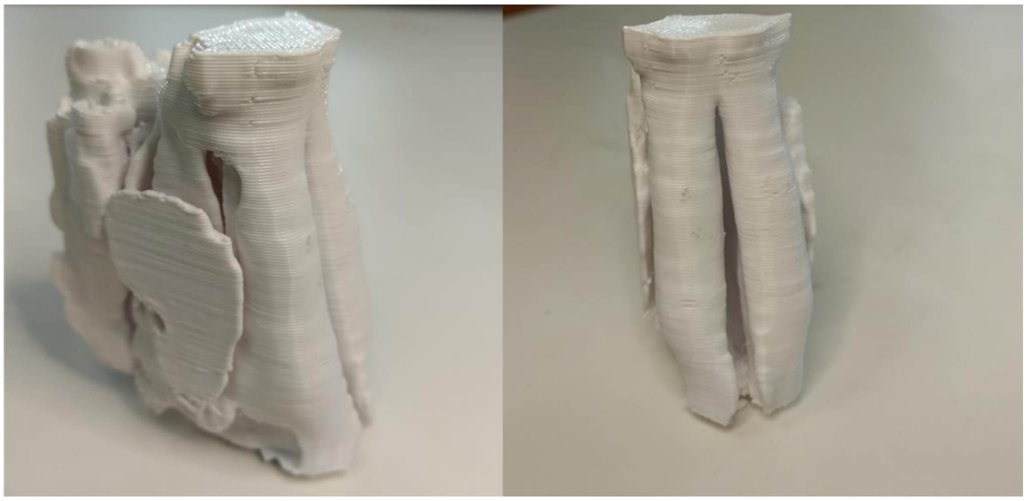


Figure 44. 3D printed model from Setup D1

Chapter 6 Conclusions

A novel method of generating physical 3D printed models from a series of 2D scanned medical images is proposed with the objective of obtaining more accurate surfaces of the resulting printed part. The method consist of three stages: Stage 1 resamples the images to increase their resolution; Stage 2 generates the intermediate slices between the input images so as to match printer resolution; and Stage 3 generates the mesh layer by layer segmenting the voxels at the same time. This method has been implements as a C++ software application, and sample 3D printed parts have been facicated based of a series of 2D scanned medical data.

A systematic study of effect of different techniques in the proposed method on the quality of the 3D printed part is conducted. By comparing the 3D printed models developed using five different Setups, the following conclusions can be drawn:-

1. Resampling the input images improves the quality of 3D printed model to a great extent and thus should always be carried if the input data is coarse. The contours obtained after image resampling are smoother. However, size of the resulting .STL files puts a limitation on the image magnification factor.
2. Contrast enhancement has little or no effect on the quality of mesh generated. To save computational effort, contrast enhancement should only be conducted when the contrast between the part to be segmented and its neighborhood is too low.
3. Segmenting the images prior to mesh generation should be avoided. Mesh generation using the original data generates smoother and a more precise surface.
4. Piecewise cubic interpolation is favored over linear interpolation. Prominent staircase effect is introduced by linear interpolation at the location of original slices. The sudden change in surface slopes can be eliminated to a great extent using piecewise cubic interpolation scheme. However, it does not completely eliminate irregularities in the slope, which are prominenet at certain locations.
5. The mesh generated by the proposed methods shows excellent surface properties with minimal errors. There are no significant gaps, degenerate faces, non-manifold geometries present in the mesh. Due to layer-by-layer pixel wise mesh generation, continuity in mesh is guaranteed. No post-processing of mesh is required for 3D printing the generated models.

It can be safely concluded that the proposed method is capable of producing precise 3D models with good surface finish and excellent mesh quality.

6.1 Contributions

The thesis represents a first step in proposing an improved method to generate 3D models from 2D medical imaging data specifically for 3D printing application. The main contributions can be summarized as follows:

1. Traditionally, 3D reconstruction of 2D medical imaging data was done without considering the 3D printer's fabrication layer thickness. This thesis introduces the idea of generating precise 3D models from 2D medical imaging data, specifically for use with additive manufacturing technologies. The advantage of this approach is that the models produced has smoother transitions in slopes and better surface finish.
2. A step-by-step procedure to convert the 2D medical imaging to 3-D surface model with high accuracy mesh quality is proposed and demonstrated. The proposed method startergically eliminate errors during various stages of conversion of 2D imaging data to 3D model. The major advantage of this method is the elimination of the need for post proceeing of the generated 3D models before they can be fabricated using additive manufacturing technologies. The models generated are 3D print ready saving expensive and time-consuming post processing.

References

[1] D. Elson and G.-Z. Yang, "The principles and roles in Medical Imaging in Surgery," in *Key Topics in Surgical Research and Methodology*, Springer, Berlin, Heidelberg, pp. 529-543.

[2] W. Niessen and M. Modat, "Workshop on open-source mediccal image analysis doftware," ISBI, 2012. [Online]. Available: http://www0.cs.ucl.ac.uk/opensource_mia_ws_2012/links.html.

[3] K. M. Farooqi and P. P. Sengupta, "Echocardiography and Three-Dimensional Printing: Sound Ideas to Touch a Heart," *Journal of the American Society of Echocardiography,* vol. 28, no. 4, pp. 398-403, 2015.

[4] A. B. Dababneh and I. T. Ozbolat, "Bioprinting Technology: A Current State-of-the-Art Review," *ASME. J. Manuf. Sci. Eng,* vol. 136, no. 6, 2014.

[5] B. Ripley, D. Levin, T. Kelil, J. L. Hermsen, K. Sooah, J. H. Maki and G. J. Wilson, "3D printing from MRI Data: Harnessing strengths and minimizing weaknesses," *Journal of Magnetic resonance Imaging,* vol. 45, no. 3, pp. 635-645, 2017.

[6] "Medical imaging," Wikipedia foundation, Inc, 11 October 2017. [Online]. Available: https://en.wikipedia.org/wiki/Medical_imaging.

[7] "National Institute of Biomedical imaging and Bioengineering," [Online]. Available: https://www.nibib.nih.gov/science-education/science-topics/computed-tomography-ct.

[8] N. Keat, "X ray CT," ImPact Group, 17 06 2002. [Online]. Available: http://www.impactscan.org/slides/xrayct/index.htm.

[9] L. W. Goldman, "Principles of CT: Radiation Dose and Image Quality," *Journal of Nuclear Medicine Technology,* vol. 35, no. 4, pp. 213-215, 2007.

[10] "DICOM," [Online]. Available: http://dicom.nema.org/.

[11] J. Poskanzer, "Pgm," Netpbm, 1989. [Online]. Available: http://netpbm.sourceforge.net/doc/pgm.html.

[12] A. L. Cohen, "Interfacing CAD and Rapid prototyping," *Rapid Prototyping Report,* vol. 2, no. 1, pp. 4-6, 1992.

[13] S. M. Nagy and G. Matayasi, "Anaysis of STL Files," *Mathematical and computer modelling,* vol. 38, pp. 945-960, 2003.

[14] ANSI, "IEEE Standard for Binary Floating-Point Arithematic," *IEEE Computor society*, pp. 754-785, 1985.

[15] J. H. Bohn, "Automatic CAD-model Repair," U.M.I Dissertation Services, Ann Arbor, Michigan, 1993.

[16] K. F. leong, C. K. Chua and Y. M. Ng, "A study of stereolithography files and Repair. part 1.," *The international journal of Additive manufacturing Technology,* vol. 12, pp. 407-414, 1996.

[17] M. vatani, F. Barazandeh, A. R. Rahimi and A. S. Nehzad, "Improved slicing algorithm employing Nearest distance method," *Journal of engineering manufacture*, vol. 224, no. B, pp. 745-752, 2010.

[18] A. Dolenc and I. Makela, "Slicing procedure for layerd manufacturing techniques," *Computer-aided Design* , vol. 26, no. 2, pp. 119-126, 1994.

[19] P. M. Pandey, V. N. Reddy and S. G. Dhande, "Slicing procedure in layered manufacturing; a review," *Rapid manufacturing Journal,* vol. 9, no. 5, pp. 274-288, 2003.

[20] T. M. Lehmann, C. Gonner and K. Spitzer, "Survey: Interpolation Methods in Medical Imaging Processing," *IEEE Transactions on Medical Imaging,* vol. 18, no. 11, pp. 1049-1075, 1999.

[21] D. Ma, F. Lin and C. K. Chua, "Rapid Prototyping Applications in Medicine. Part 1:NURBS-based volume modelling," *The International journal of Addititive Manufacturing Technology,* vol. 18, pp. 103-117, 2001.

[22] Y. Liu, M. R. Johnson, A. Matida, S. Kherani and J. Marsan, "Creation of a standardized geometry of the human nasal cavity," *Journal of Applied Physiology,* vol. 106, no. 3, pp. 784-795, 2009.

[23] P. J. Schneider, "MACTECH," Xplain corporation, 1984. [Online]. Available: http://www.mactech.com/articles/develop/issue_25/schneider.html.

[24] C.-C. Liang and W.-C. Lin, "Intensity Interpolation for reconstruction 3-D medical images from serial crosssections," in *IEEE Engineering in Medicine & biology society 10th AnnualL International Conference*, 1988.

[25] W.-C. Lin, C.-C. Liang and C.-T. Chen, "Dynamic Elastic Interpolation for 3-D Medical Image Reconstruction from serial cross-section," *IEEE transaction on Medical Imaging,* vol. 7, no. 3, pp. 225-232, 1988.

[26] T.-Y. Lee and C.-H. Lin, "Feature-guided Shaped-based Image interpolation," *IEEE transaction on Medical Imaging,* vol. 21, no. 12, pp. 1479-1489, 2002.

[27] C.-S. Wang, W.-H. A. Wang and M.-C. Lin, "STL rapid prototyping bio-CAD model for CT medical image segmentation," *Computers in Industry,* vol. 61, pp. 187-197, 2010.

[28] T. Kluge, "Cubic Spline interpolation in C+=," 2011. [Online]. Available: http://kluge.in-chemnitz.de/opensource/spline/spline.h. [Accessed 2017].

[29] W. E. Lorenson and H. E. Cline, "Marching cube: A high resolution 3D surface reconstruction algorithm," in *ACM*, New york, 1987.

[30] K.-H. Cheng, Y.-S. Cheng , H.-C. Yeh, R. A. Guilmette, S. Q. Simpson, Y.-H. Yang and D. L. Swift, "In vivo mesurement of nasal airway dimensions and ultrafine aerosol deposition

in the human nasal and oral airways," *Journal of Aerosol science,* vol. 27, no. 5, pp. 785-801, 1996.

[31] J. P. Corey, A. Gungor, R. Nelson, J. Fredberg and V. Lai, "A Comparison of the Nasal Cross-Sectional Areas and Volumes Obtained with Acoustic Rhinometry and Magnetic Resonance Imaging," *Otolaryngology-Head and Neck Surgery,* vol. 117, no. 4, pp. 249-354, 2016.

[32] "PLA," PushPlastic, [Online]. Available: https://www.pushplastic.com/collections/pla-filament.

Appendix A

Header file to define integer points and float points and related functions

Points.h

```
#ifndef points_hpp
#define points_hpp

/*Structure to save Integers pair in 2D, IP# denotes interger point
where # is the dimension */
struct Ip2
{ int x, z;
  Ip2 (int x = 0, int z = 0): x(x), z(z) {}
  inline  bool operator==(const Ip2& p) const { return x==p.x&&z==p.z;
}
  inline  bool operator!=(const Ip2& p) const { return x!=p.x||z!=p.z;
}

};
/*Structure to save floating pair in 2D, IP# denotes floating point
where # is the dimension */
struct Dp2
{
  float x;
  float z;
  Dp2 (float x = 0, float z = 0) : x(x), z(z) {}
  Dp2 (Ip2 ip) : x((float)ip.x), z((float)ip.z) {}
  inline  bool operator==(const Dp2& p) const { return x==p.x&&z==p.z;
}
  inline  bool operator!=(const Dp2& p) const { return x!=p.x||z!=p.z;
}

};

struct  Ip3
{ int x, y, z;
  Ip3 (int x = 0, int y = 0, int z =0): x(x), y(y), z(z) {}
  inline  bool operator==(const Ip3& p) const { return
x==p.x&&y==p.y&&z==p.z; }
  inline  bool operator!=(const Ip3& p) const { return
x!=p.x||y!=p.y||z!=p.z; }

};

 struct Fp3
{
  float x;
  float y;
  float z;
  Fp3(float x = 0, float y = 0, float z =0) : x(x), y(y), z(z) {}
  Fp3 (Ip3 i) : x((float)i.x), y((float)i.y), z((float)i.z) {}
  inline  bool operator==(const Fp3& p) const { return
x==p.x&&y==p.y&&z==p.z; }
  inline  bool operator!=(const Fp3& p) const { return
x!=p.x||y!=p.y||z!=p.z; }

};

//Functions to convert between the types

   inline Fp3 int2ToFloat3 ( const Ip3& p, const float z){ return Fp3(
(float)p.x, (float)p.y, (float)z);}
   inline Ip3 float2ToInt3 (const Fp3& p, const float  z){ return
Ip3((int)p.x, (int)p.y, (int)z);}
   inline Ip2 add (Ip2& p, Ip2& q) { return Ip2(p.x+q.x, p.z+q.z);}
   inline Ip2 add (Ip2& p, int x, int z) { return Ip2(p.x+x, p.z+z);}
   inline Ip3 add (Ip3& p, Ip3& q) { return Ip3(p.x+q.x, p.y+q.y,
p.z+q.z);}
   inline Ip3 add (Ip3& p, int x, int y, int z) { return Ip3(p.x+x,
p.y+y, p.z+z);}
   inline Dp2 add (Dp2& p, Dp2& q) { return Dp2(p.x+q.x, p.z+q.z);}
   inline Dp2 add (Dp2& p, float x, float z) { return Dp2(p.x+x,
p.z+z);}
   inline Fp3 add (Fp3& p, Fp3& q) { return Fp3(p.x+q.x, p.y+q.y,
p.z+q.z);}
   inline Fp3 add (Fp3& p, float x, float y, float z) { return
Fp3(p.x+x, p.y+y, p.z+z);}



 #endif
```

Header file for image definition and image related functions

CTimage.h

```
//Header file for Image structure

#ifndef _CT_IMAGE
#define _CT_IMAGE

#include"utils/points.h"
#include<iostream>
#include<cstdlib>
#include<cstdio>
#include<math.h>
#include <fstream>
using namespace std;
```

```
// Structure for storing the image
 class Image
{ public:
   int numberOfColumns, numberOfRows, maxVal;
//variables for store spatial resolution in all directions
   float  x_pixelSize, z_pixelSize, y_pixelSize;
// Array to store pixel data
   unsigned char * imagedata;
// String to store image header for Image writting functions
   char * imageheader;
//Variable used to track the position of the image
   int z_level;
// Variable storing imagedata size (used in intializing the array
   long stringsize;

//functions
// Constructers 3 different types
   void initialize ();
   void initialize(int, int, int);
   void initialize(int, int, int, float , float);
//Function to reallocate the image for cropping or maginification
purposes
   void reallocate(int, int);
//Image reading and writing functions
   int readImage (int ,char**);
   void readHeader(FILE*);
   void writeImage (char *);
   void writeHeader();
//Function to calculate x and y pixel position (in plane) from array
index number
   Ip2  stringToCords (long int);
//Function to calculate array index number from x and y pixel position
   long int cordsToString (Ip2&);
//Function to calculate pixel cordinates in global 3D cartesian system
   Fp3  stringToRealCords (long int, int);
//Function to calculate array index from 3D cartesian cordinates
   int realCordsToString (float, float, char*);
//Function to padd croped images with a null value border
   void paddImageBorders();
//Destructer
   ~Image(){};
};


#endif
```

CTimage.cpp

```
#include"CTimage.h"
```

```
using namespace std;
// In class functions
//Default constructer
void Image::initialize()
{ this->numberOfColumns =512;
  this->numberOfRows =512;
  this->x_pixelSize =0.4;
  this->z_pixelSize =0.4;
  this->maxVal =255;
  this-> stringsize = ((this->numberOfRows)*(this->numberOfColumns));
  this->imagedata  =(unsigned char*) malloc (sizeof(unsigned
char)*(stringsize));
  this->imageheader  =( char*) malloc (sizeof(char)*(15));
  if(this->imagedata==NULL)
    { cout<<"\n  space allocation for Imagedata was not sucessfull ";
    }
}

//Dynamic constructer (puts pixel size as 1 by default)
void Image::initialize( int columns , int rows, int maxval)
{this->numberOfColumns =columns;
 this->numberOfRows =rows;
 this->maxVal=maxval;
 this->x_pixelSize =1;
 this->z_pixelSize =1;
 this->y_pixelSize = 1;
 this-> stringsize = ((this->numberOfRows)*(this->numberOfColumns));
 this->imagedata  =(unsigned char*) malloc (sizeof(unsigned
char)*(stringsize));
this->imageheader  =( char*) malloc (sizeof(char)*(15));
 if(this->imagedata==NULL)
    { cout<<"\n  space allocation for Imagedata was not sucessfull ";
    }
}
//Dynamic constructer (preferred)
void Image::initialize( int columns , int rows, int maxval, float
x_z_pixel, float y_pixel)
{this->numberOfColumns =columns;
 this->numberOfRows =rows;
 this->maxVal=maxval;
 this->x_pixelSize =x_z_pixel;
 this->z_pixelSize =x_z_pixel;
 this->y_pixelSize =y_pixel;
 this-> stringsize = ((this->numberOfRows)*(this->numberOfColumns));
 this->imagedata  =(unsigned char*) malloc (sizeof(unsigned
char)*(stringsize));
this->imageheader  =( char*) malloc (sizeof(char)*(15));
 if(this->imagedata==NULL)
    { cout<<"\n  space allocation for Imagedata was not sucessfull ";
    }
}
```

```
/* Input : Desired no of rows and columns to change the size of
imagedata */
void Image::reallocate(int columns, int rows)
{this->numberOfColumns =columns;
 this->numberOfRows =rows;
 this-> stringsize = ((this->numberOfRows)*(this->numberOfColumns));
 this->imagedata  =(unsigned char*) realloc (this->imagedata,
sizeof(unsigned char)*(stringsize));
this->imageheader  =( char*) malloc (sizeof(char)*(15));
 if(this->imagedata==NULL)
    { cout<<"\n  space reallocation for Imagedata was not sucessfull
";
    }
}

/**********************************************************/
/***************** read image functions********************/
/**********************************************************/

void Image::readHeader (FILE *fp)
{  int rows=0;int  columns=0; int maxval=0;
   fscanf(fp, " P5 %d %d %d", &columns, &rows, &maxval);
   this->numberOfColumns = columns;
   this->numberOfRows = rows;
   this->maxVal = maxval;
}

int  Image::readImage ( int fileno, char** argv)
{
   long lSize;
   size_t result; // to cross-check size of the imagedata
   int location =0;

   FILE *fp = fopen(argv[fileno],"rb");

   if(fp == NULL )
    { cout<<"\n Such a file does not exist \n";
      exit(1);;
    }
   else
    { cout<<"\n Successfully loaded the image";
    }
   fseek( fp,0,SEEK_END);
   lSize = ftell (fp)-15;
   rewind(fp);

   readHeader(fp); // reads image header
   fseek(fp, 15, SEEK_SET);

   if( this->imagedata ==NULL) { fputs(" Memory error", stderr);
                                 exit(2);
```

```
                              }

   result = fread(this->imagedata, sizeof(unsigned char), lSize , fp);
   cout<<"\n Done reading the image.. \n";
}

/*************************************************************/
/****************write image functions************************/
/*************************************************************/

// rewrites imageheader as size of the image might change
void  Image::writeHeader ()
{
  snprintf(this->imageheader, 15, "P5\n%d %d\n%d\n",this-
>numberOfColumns,this->numberOfRows, this->maxVal);

}
 // Writes image to a file named by foutname[]
void  Image::writeImage (char foutname[])
{ FILE *fout;
 fout =fopen(foutname, "wb");
 if(fout == NULL )
    { cout<<"\n error while opening the fout \n";
      exit(1);;
    }

  writeHeader();
  fwrite(this->imageheader, sizeof(char),15, fout);
  fwrite("\n",sizeof(char),1,fout);
  fwrite(this->imagedata, sizeof(unsigned char),this->stringsize,
fout);
  if( fout ==NULL)
  { cout<<"\n Error writing the image \n ";
  }
  else
  {
  cout<<"\n Done writing the image \n";
  }
  fclose(fout);
}

/*************************************************************/
/******conversion functions between bitmamap and imagedata*******/
/* Here the bitmap is assumed to start from 0 to N in both x and y
directions    */
/*****  Also imagedata is an array so the index starts from 0 ***/
/*************************************************************/

   Ip2 Image:: stringToCords (long int i)
   { int x, z;
     x=(i % this->numberOfColumns);
     z= (i/this->numberOfColumns);
     return Ip2(x,z);
   }

  long  int Image::cordsToString (Ip2& p)
   { long int i;
     i = (p.x+p.z*this->numberOfColumns);
     return i;
   }


  Fp3 Image:: stringToRealCords (long int i, int yInt)
  { float x, y, z;
    x=(i % this->numberOfColumns)*this->x_pixelSize;
    z= (i/this->numberOfColumns)*this->z_pixelSize;
    y =(yInt * this->y_pixelSize);
    return Fp3(x,y,z);
  }


/*****************************************************************/
/************* additional functions ************************/
/******************************************************************/
void Image:: paddImageBorders()
{ Ip2 check;
   for(long int i=0; i< this->stringsize-1; i++)
   { check = stringToCords (i);
     if (check.x == 0 || check.z==0 || check.z == (this->
numberOfRows-1) ||check.x == (this->numberOfColumns-1))
        { this-> imagedata[i]=0;
        }
   }
}


//int realCordsToString (float, float, char*);
//   ~Image(){};
```

Header file for the code that works on the proposed method. Imagearray.cpp contains main()

Imagearray.h

```
#ifndef IMAGE_ARRAY
#define IMAGE_ARRAY
#include <list>
#include"CTimage.h"

// array of our defined structure
  Image *imagearray;
  Image *inputarray;
  Image *paddedarray;
  int no_layers;
  int total_images;


// Structure to store traingles constructed usin marching cube
typedef struct {
   Fp3 p[3];
   Fp3 normal;
} TRIANGLE;

//list of all traingles structures
std::list<TRIANGLE> triangles;

// Structure used as the virtual cube in the marching cube algorithm
typedef struct {
   Fp3 p[8];
   unsigned char val[8];
} GRIDCELL;


// Fuctions for mesh generation

long  marchingCube(Image[],int,unsigned char);
void assignCube (Image[],GRIDCELL &, long  , int);
Fp3 getEdgeIntersection(GRIDCELL &, int , int );
Fp3 getVertex(int, Fp3*);
void getNormal( TRIANGLE & );
void writeSTL();

// functions on array of structure
 void preInitializeCheck(int&, int&, int& , char**)
 void allocateInputArray ( int , char**, int, int, float, float);
 void interpolate ( float, float, int);
 bool initialize_imagearray ( float, float, int, int, int, int, float,
float);
 void deconstruct_arrays();
 void initialize_paddedarray(int, int , int, float ,float);
 void threshold(Image&, unsigned char);

#endif
```

Imagearray.cpp

```
#include<sys/stat.h>
#include<iostream>
#include"imagearray.h"
#include"spline.h"
using namespace tk;

/***************************************************************/
```

```
/*********** Allocate and deallocate arrays functions **********/
/***************************************************************/

/* Used to check the dimesnions of the input image. Initiates the
image structure using check  */

void preInitializeCheck(int& columns, int& rows, int& maxVal,
char**argv)
{Image check;
 FILE *fcheck = fopen(argv[1],"rb");
 check.readHeader(fcheck);
 columns= check.numberOfColumns;
 rows= check.numberOfRows;
 maxVal= check.maxVal;
}

/****************imagearray***********************************/
/* Imagearray is the structure that stores the new intermediate images
*/

bool initialize_imagearray ( float y_scan, float  y_AM, int argc, int
columns, int rows, int maxVal, float x_z_pixel)
{  /* Here y_AM gives us the  new y-resolution we expect from the 3-D
printed parts, y_scan gives us the current y-resolution
from the given ct images and total_images gives us the no of pgm
images generated
   */
   if (y_AM>y_scan)
  { cout<<"\n New resolution cannot be bigger than current
resolution\n";
    return false;
  }
   if (y_AM !=0)
   {  total_images = int((argc-2)*y_scan/y_AM)+1;
      imagearray =(Image*)malloc(sizeof(Image)*total_images);
      if(imagearray==NULL)
        { cout<<"\n Imagearray  allocation was not successfull \n";
        }
      for(int i=0; i<total_images;i++)
       { imagearray[i].initialize(columns, rows, maxVal, x_z_pixel,
y_AM);
       }
   }
  else
   {  cout<<"\n New z-resolution cannot be zero";
      exit;
   }

  cout<<"\n Total no of images created are "<< total_images <<"\n";
  return true;
}
```

```
/*********************** inputarray***************************/
/* Input array is the struture which only stires the input images. The
size of the array is equal to the no of input images*/

void allocateInputArray (int argc, char**argv, int columns, int rows,
int maxVal, float x_z_pixel,float y_pixel)
{ inputarray = (Image*) malloc (sizeof(Image)*(argc-1));

  for (int i=0; i <argc-1; i++)

        { inputarray[i].initialize(columns, rows, maxVal, x_z_pixel,
y_pixel);
          inputarray[i].readImage (i+1, argv);
        }
}

/********************* paddedarray***************************/
/* This array has a size of imagearray +2 in whhich the first and the
last image has all the pixel defined as null pixel. Marching cube is
only done on paddedarray. Here lenght denotes the original lenght of
the array to be transfered to paddedarray */

int initialize_paddedarray(Image array[], int length)
{paddedarray =(Image*)malloc(sizeof(Image)*(length+2));
 int columns = array[0].numberOfColumns;
 int rows = array[0].numberOfRows;
 int maxVal = array[0].maxVal;
 float x_z_pixel = array[0].x_pixelSize;
 float y_pixel = array[0].y_pixelSize;
 for (int i =1; i< length+1; i++)
    {paddedarray[i].initialize(columns, rows, maxVal, x_z_pixel,
y_pixel);
    for(int j=0; j<paddedarray[i].stringsize; j++)
{   paddedarray[i].imagedata[j]= array[i-1].imagedata[j];}
     paddedarray[i].paddImageBorders();
    }
 paddedarray[0].initialize(columns, rows, maxVal, x_z_pixel, y_pixel);
 paddedarray[length+1].initialize(columns, rows, maxVal, x_z_pixel,
y_pixel);
 for(long i =0; i< paddedarray[0].stringsize; i++)
   {
    paddedarray[0].imagedata[i]=0;
    paddedarray[length+1].imagedata[i] = 0;
   }
 cout<<"\n Done initializing the paddedarray \n";
 return(length+2);
}

/* function to reallocate the paddedarray when new array is to shifted
to paddedarray for mesh generation purposes */
 int  reallocate_paddedarray(Image array[], int length1, int length2)
{if (length1>length2)
```

```
  { for (int i = length2+2; i<length1; i++)
      { delete[] paddedarray[i].imagedata;
      }
  }
 paddedarray=(Image*)realloc(paddedarray, sizeof(Image)*(length2+2));
 int columns = array[0].numberOfColumns;
 int rows = array[0].numberOfRows;
 int maxVal = array[0].maxVal;
 float x_z_pixel = array[0].x_pixelSize;
 float y_pixel = array[0].y_pixelSize;
 for (int i =0; i<(length2+2); i++)
    { paddedarray[i].numberOfColumns =columns;
      paddedarray[i].numberOfRows =rows;
      paddedarray[i].maxVal= maxVal;
      paddedarray[i].x_pixelSize =x_z_pixel;
      paddedarray[i].z_pixelSize =x_z_pixel;
      paddedarray[i].y_pixelSize =y_pixel;
      paddedarray[i].reallocate(columns, rows);
      if(i!=0 && i!=(length2+1))
      { paddedarray[i].imagedata = array[i-1].imagedata;
        paddedarray[i].paddImageBorders();
      }
      if (i==0|| i==(length2+1))
      { for(long j =0; j< paddedarray[0].stringsize; j++)
           {
            paddedarray[i].imagedata[j]=0;
           }
      }
    }
  cout<<"\n exiting reallocation of paddedarray\n";
  return (length2+2);
}

/********************Free all arrays *************************/

void  deconstruct_arrays (int argc, int length)
{  /*  free struct */
    for (int i=0; i<total_images; i++)
        { delete[] imagearray[i].imagedata;
        }
    for(int i=0; i<argc-1; i++)
        { delete[] inputarray[i].imagedata;
        }
/*    for(int i=0;i< length; i++)
        { delete[] paddedarray[i].imagedata;
        }*/
    free(imagearray);
    imagearray = NULL;
    cout<<"\n Imagearray  deallocation successfull \n";
    free( inputarray);
    inputarray = NULL;
    cout<<"\n Inputarray  deallocation successfull \n";
    free( paddedarray);
    paddedarray = NULL;
    cout<<"\n Paddedtarray  deallocation successfull \n";

}


/*****************************************************************/
/************* Image arrays processing functions***************/
/*****************************************************************/

/* Interpolate used for pixelwise linear interpolation of images*/

void interpolate ( float y_scan, float y_AM, int argc )
{ cout<<"\n Entered interpolation function \n";
  for( long i=0; i<(imagearray[0].stringsize);i++)
  {  unsigned char  pixeldifference =0;//difference in pixel value
     unsigned char basepixel =0;
     bool pixelpositive = true;
     int inputno = 0; // tracking position in inputarray
     float yabsolute = 0.0; // tracking spatial position of the image
     float yreference = y_scan; // reference used to keep a check on
interpolation
     float ycorrection =0.0; // correction to yreference after it
moves to next consecutive pair of images
     int inpcheck = 1.0;  // counter used to avoid calculating slope
in every iteration

    for(int j=0; j<total_images; j++)
       { if(inputno <argc-1 && inpcheck!=inputno)
         {

if(inputarray[inputno+1].imagedata[i]>inputarray[inputno].imagedata[i]
)
          { pixeldifference = inputarray[inputno+1].imagedata[i]-
inputarray[inputno].imagedata[i];
            basepixel = inputarray[inputno].imagedata[i];
            pixelpositive= true;
          }

if(inputarray[inputno+1].imagedata[i]<inputarray[inputno].imagedata[i]
)
          { pixeldifference = -
inputarray[inputno+1].imagedata[i]+inputarray[inputno].imagedata[i];
            basepixel = inputarray[inputno].imagedata[i];
            pixelpositive= false;
          }

if(inputarray[inputno+1].imagedata[i]==inputarray[inputno].imagedata[i
])
          { pixeldifference = inputarray[inputno+1].imagedata[i]-
inputarray[inputno].imagedata[i];
```

```
            basepixel = inputarray[inputno].imagedata[i];
            pixelpositive= true;
          }
        inpcheck = inputno;
      }

       if (yabsolute<= yreference && yreference <=
(total_images+1)*y_AM)
          {
              if(pixelpositive)
                { imagearray[j].imagedata[i]= basepixel +(yabsolute-
ycorrection)*(pixeldifference)/y_scan;
             yabsolute = yabsolute + y_AM;
                }

              if(!pixelpositive)
                { imagearray[j].imagedata[i]= basepixel -(yabsolute-
ycorrection)*(pixeldifference)/y_scan;
                  yabsolute = yabsolute + y_AM;
                }
              }
        if (yabsolute >yreference && yreference <=
(total_images+1)*y_AM)
              { ycorrection = yreference;
           yreference = yreference +y_scan;
             inputno++;

            }
      }
   } cout<<"\n Exiting interpolate ";
}

/* Function to interpolate images using piiecwise cubic interpolation.
Spline genrating function taken from tino klug repository. Here xin
and yin are used to represent the spline cordinates in thier local
system where x is the base axis  and y is the value axis...........
    y ^
      |    ''                 ''           '
      |   '  ''''         '  '  '''    '''
      |..'       ''.....'' '.      '..'
      |______________________________________________>
                                                     X
Here xin dedones input grid points and xout denotes output points.
Similarily yin and yout denotes corresponding values
 */

void cubicInterpolate (float y_scan, float y_AM, int argc)
{ cout<<"\n Entered cubic interpolation function \n";
  bool cubic_spline = true;
  for(long l =0; l<imagearray[0].stringsize;l++)
  { std:: vector<float> X(argc-1), Y(argc-1);
     float xin=0.0; //
     float yin;
     float xout=0.0;
     float yout;
      for(int m =0; m<argc-1; m++)
     {
       X[m]= xin;
       xin = xin + y_scan;
       yin =(float) inputarray[m].imagedata[l];
       Y[m]= yin;
     }
    tk::spline s;
    s.set_points(X,Y, cubic_spline);
    for(int n =0; n<total_images; n++)
    {
      yout= s(xout);
      if (yout>255.0)
         { yout = 255.0;}
      if (yout < 0.0)
         { yout = 0.0;}
      imagearray[n].imagedata[l] = (unsigned char)yout;
      xout= xout+y_AM;
     }
   }
 cout <<"\n Done with cubic interpolation \n ";
}





/*********************marching cube ***************************/

void assignCube (GRIDCELL &grid,long iIndex, int yIndex )
{ float x_offset, y_offset, z_offset;
   x_offset= paddedarray[yIndex].x_pixelSize/2;
   z_offset = paddedarray[yIndex].z_pixelSize/2;
   y_offset = paddedarray[yIndex].y_pixelSize /2;

// Set virtual cube cordinates
   grid.p[0]= paddedarray[yIndex].stringToRealCords( iIndex, yIndex );
   grid.p[1]= paddedarray[yIndex].stringToRealCords( iIndex+1, yIndex
);
   grid.p[2]= paddedarray[yIndex].stringToRealCords(
iIndex+paddedarray[yIndex].numberOfColumns+1, yIndex );
     grid.p[3]= paddedarray[yIndex].stringToRealCords(
iIndex+paddedarray[yIndex].numberOfColumns, yIndex );
   grid.p[4]= paddedarray[yIndex].stringToRealCords( iIndex, yIndex+1
);
    grid.p[5]= paddedarray[yIndex].stringToRealCords( iIndex+1,
yIndex+1);
```

```
     grid.p[6]= paddedarray[yIndex].stringToRealCords(
iIndex+paddedarray[yIndex].numberOfColumns+1, yIndex+1 );
    grid.p[7]= paddedarray[yIndex].stringToRealCords(
iIndex+paddedarray[yIndex].numberOfColumns, yIndex+1 );

//Offset the virtual cube such that traingle base lies on the loaction
of the slices
  for(int v =0; v<8; v++)
     { grid.p[v]= add (grid.p[v],x_offset, y_offset, z_offset);
     }

//Assign grayscalevalue to the cube vertices
  grid.val[0]= paddedarray[yIndex].imagedata[iIndex];
  grid.val[1]= paddedarray[yIndex].imagedata[iIndex+1];
  grid.val[2]=
paddedarray[yIndex].imagedata[iIndex+paddedarray[yIndex].numberOfColum
ns+1];
  grid.val[3]=
paddedarray[yIndex].imagedata[iIndex+paddedarray[yIndex].numberOfColum
ns];
  grid.val[4]= paddedarray[yIndex+1].imagedata[iIndex];
  grid.val[5]= paddedarray[yIndex+1].imagedata[iIndex+1];
  grid.val[6]=
paddedarray[yIndex+1].imagedata[iIndex+paddedarray[yIndex].numberOfCol
umns+1];
  grid.val[7]=
paddedarray[yIndex+1].imagedata[iIndex+paddedarray[yIndex].numberOfCol
umns];

}

/* Find the location of a traingle vertex on the edge of the virtual
cube using interpolation */

Fp3 getEdgeIntersection(GRIDCELL &grid, int vertex1 , int vertex2,
unsigned char Tlevel )
{  Fp3 v1,v2, e;
   v1 = grid.p[vertex1];
   v2= grid.p[vertex2];
   e.x= v1.x+(v2.x-v1.x)*(Tlevel -
grid.val[vertex1])/(grid.val[vertex2] - grid.val[vertex1]);
   e.y=v1.y+(v2.y-v1.y)*(Tlevel -
grid.val[vertex1])/(grid.val[vertex2] - grid.val[vertex1]);
   e.z= v1.z+(v2.z-v1.z)*(Tlevel -
grid.val[vertex1])/(grid.val[vertex2] - grid.val[vertex1]);
   return(e);
}

//Get vertex from
Fp3 getVertex(int edgeno, Fp3 * edgeIntersectList)
{ Fp3 e;
 e = edgeIntersectList[edgeno];
 return(e);
}

void  getNormal(TRIANGLE &tri)
{ Fp3 p1,p2,p3;
  p1 = tri.p[0];
  p2 = tri.p[1];
  p3 = tri.p[2];
  tri.normal.x = (p2.y -p1.y)*(p3.z -p1.z) - (p3.y -p1.y)*(p2.z -
p1.z);
  tri.normal.y = (p2.z -p1.z)*(p3.x -p1.x) - (p2.x -p1.x)*(p3.z -
p1.z);
  tri.normal.z = (p2.x -p1.x)*(p3.y -p1.y) - (p3.x -p1.x)*(p2.y -
p1.y);
}




/*
   Given a grid cell and an isolevel, calculate the triangular
   facets required to represent the isosurface through the cell.
   Return the number of triangular facets, the array "triangles"
   will be loaded up with the vertices at most 5 triangular facets.
      0 will be returned if the grid cell is either totally above
   of totally below the isolevel.
*/

long marchingCube(int length,unsigned char Tlevel)
{
   long nTriang=0;
   int cubeIndex, edgeIndex;
   Fp3 vertlist[12];
   Fp3 edgeIntersectList[12];
   GRIDCELL grid;


   extern int edgeTable[256];
   extern int triTable[256][16];
   extern int edgeConnection[12][12];


  for (int yIndex = 0; yIndex < length-1; yIndex++)
  {   int columns = paddedarray[yIndex].numberOfColumns;
      int rows = paddedarray[yIndex].numberOfRows;
      for ( long iIndex =0; iIndex< paddedarray[yIndex].stringsize-1;
iIndex++)
       {  Ip2 check;
          check = paddedarray[yIndex].stringToCords( iIndex);
         if ((check.x < (columns-1)) && (check.z< (rows -1)))
          { assignCube(grid , iIndex, yIndex);
```

```
            //cout<< check.x << "\t" << check.z <<"\t"<<grid.p[0].x
<<"\t" ;
    /*
      Determine the index into the edge table which
      tells us which vertices are inside of the surface
    */
            cubeIndex = 0;
           for(int v = 0; v < 8; v++)
            {
                if(grid.val[v] > Tlevel)
                     {  cubeIndex |= 1 <<v; }
            }
   /* Cube is entirely in/out of the surface */
           if (cubeIndex == 0)
             { continue;}
         //   cout << check.x <<"\t " << cubeIndex <<" \t" ;
            edgeIndex =edgeTable[cubeIndex];
   /* Find the vertices where the surface intersects the cube */
   //Then find the normal to the surface at those points
          for (int iEdge = 0; iEdge < 12; iEdge++)
            {
   //if there is an intersection on this edge
              if(edgeIndex & (1<<iEdge))
                {
                    edgeIntersectList[iEdge] =
getEdgeIntersection(grid,edgeConnection[iEdge][0] ,
edgeConnection[iEdge][1], Tlevel );

                }
            }
//Generate triangles
           for(int iTriangle = 0; iTriangle < 5; iTriangle++)
           {
                TRIANGLE tri;
                if(triTable[cubeIndex][3*iTriangle] < 0)
                      {  break; }
                for(int iCorner = 0; iCorner < 3; iCorner++)
                   {
                        tri.p[iCorner] =
getVertex(triTable[cubeIndex][3*iTriangle+iCorner],edgeIntersectList);
                   }
                getNormal( tri );
                triangles.push_back(tri);
                nTriang ++;
           }
        }
     }
  }

   return(nTriang);
 cout<<"\n Done Tessellating ";
}
```

```
/*************************************************************/
/**************** Image processing functions ******************/
/*************************************************************/

void smoothing(Image& img, float sigma,int kernalsize)
{ Image test;
  int columns=img.numberOfColumns;
  int rows = img.numberOfRows;
  int maxval = img.maxVal;
  int xgint;
  test.initialize(columns, rows, maxval);
  float r;
  float  s=2.0 * sigma * sigma;
  float sum =0.0;
  float xg =0.0;
  int width = test.numberOfColumns;
  float gKernal[kernalsize][kernalsize];
  float matrix[kernalsize][kernalsize];
  int k = kernalsize/2;
  Ip2 point,matrixmaker;
  long int  matrixindex;

 for(int x =-k; x<=k; x++)
  {  for(int z= -k; z<= k; z++)
     {  r= sqrt(x*x+z*z);
        gKernal[x+k][z+k]= (exp(-(r*r)/s))/(M_PI*s);
        sum+=gKernal[x+k][z+k];
     }
  }
 cout<<"\n Kernal sum is "<<sum<<"\n";
 for(int i =0; i<kernalsize; i++)
   { for(int j=0; j <kernalsize; j ++)
     {  gKernal[i][j]/=sum;
        cout<< gKernal[i][j]<<"\t";;
     } cout<<"\n";
   }


 for (long int i =0; i< test.stringsize-1; i++)
    {xg=0;
     point = img.stringToCords(i);
     cout<<(int) img.imagedata[i];
     cout<< "\n";
     if(point.x <rows-k && point.z<columns-k && point.x>k &&
point.z>k)
       {   for(int x =-k; x<=k; x++)
             {  for(int z= -k; z<= k; z++)
                  { matrixmaker.x = point.x +x;
                    matrixmaker.z = point.z +z;
                    matrixindex =img.cordsToString(matrixmaker);
```

```
                      matrix[x][z] =img.imagedata[matrixindex];
                      cout<< matrix[x][z]<<"\t";
                 xg += gKernal[x][z]*matrix[x][z];
                    }
              }
       /*    for(int x =-k; x<=k; x++)
              {  for(int z= -k; z<= k; z++)
                    {  xg += gKernal[x][z]*matrix[x][z];
                    }
              } */

        }
        xgint = xg;
        test.imagedata[i] =(unsigned char) xgint;
        cout<<"\t\t"<<xg <<"\t"<<xgint <<"\n";
        xg =0;
    }

  for (long int i=0; i< test.stringsize; i++)
     { img.imagedata[i] = test.imagedata[i];
     }
 }



//Threshold on original image

void threshold( Image& img , unsigned char tLevel)
{
   for (long int j =0; j< img.stringsize; j++)
     {  if ((tLevel+(unsigned char)5) <= img.imagedata[j] || (tLevel-
(unsigned char)5) <= img.imagedata[j] )
        { img.imagedata[j] =(unsigned char) 255;
        }
        else
         { img.imagedata[j]=(unsigned char) 0;
         }
     }
}

// Threshold and generate new image
Image thresholdseries (Image& img, unsigned char tLevel)
{ Image filt;
  filt.initialize(img.numberOfColumns, img.numberOfRows, img.maxVal);
 for (long int j =0; j< img.stringsize; j++)
     {  if (tLevel <= img.imagedata[j])
          { filt.imagedata[j] =(unsigned char) 255;
          }
        else
           { filt.imagedata[j]=(unsigned char) 0;
           }
     }
  return(filt);
}

// Cropping the image

void imageCrop (Image& img,int rows, int columns,int x_offset, int
z_offset)
{ Image temp;
  temp.initialize(columns, rows, 255);
  Ip2 check, doublecheck;
  Ip2 startpoint, endpoint;
  char outimgname[20];
  long counter =0;
  long startindex;
  long endindex;
  startpoint.x= x_offset;
  startpoint.z= z_offset;
  startindex = img.cordsToString(startpoint);
  endpoint.x= startpoint.x+columns-1;
  endpoint.z= startpoint.z+rows-1;
  endindex = img.cordsToString(endpoint);

  for(long i=startindex; i<=endindex; i++)
    {
       check = img.stringToCords(i);
     if( check.x >=startpoint.x && check.x <= endpoint.x && check.z >=
startpoint.z  && check.z <= endpoint.z)
       {
         temp.imagedata[counter] = img.imagedata[i];
         counter++;
         doublecheck.x = check.x; doublecheck.z=check.z;
       }
   }
  cout<< counter;
   img.reallocate(columns, rows);
  for( long i = 0; i< img.stringsize; i++)
      {  img.imagedata[i]= temp.imagedata[i];
      }

 cout<< "\n Done cropping the images ";
}

// Invert the image
void inversion(Image& temp)
{ unsigned char val;
  for (long int j =0; j<temp.stringsize; j++)
     {  val = temp.imagedata[j];
        temp.imagedata[j]= temp.maxVal -val;
     }
}

/*****************************************************************/
```

```
/******************** write stl functions ********************/
/*************************************************************/

//ASCII STL
void writeSTL(char outname[])
{ FILE *fstl;
  fstl = fopen(outname, "w");
  fprintf( fstl, " solid ctscan \n");
  for (std::list<TRIANGLE> ::iterator it = triangles.begin(); it !=
triangles.end(); ++it)
   { TRIANGLE ves = *it;
     fprintf (fstl, " facet normal %e %e %e \n", ves.normal.x,
ves.normal.y, ves.normal.z );
     fprintf ( fstl,"   outer loop \n" );
     fprintf ( fstl,"      vertex %e %e %e\n", ves.p[0].x, ves.p[0].y,
ves.p[0].z );
     fprintf ( fstl,"      vertex %e %e %e\n", ves.p[1].x, ves.p[1].y,
ves.p[1].z );
     fprintf ( fstl,"      vertex %e %e %e\n", ves.p[2].x, ves.p[2].y,
ves.p[2].z );
     fprintf (fstl, "    endloop\n");
     fprintf (fstl, " endfacet\n");
  }
  fprintf(fstl,  " endsolid ctscan");
  fclose(fstl);
}

//Binary STL
void writeSTLbin(char outname[], long nTriang)
{ char head[80] =" GR ctscan";
  short attribute =0;
  float convertor;
  unsigned long n= (unsigned long)nTriang;
  FILE *fstl;
  fstl = fopen(outname, "wb");
  /*for(list<TRIANGLE>::iterator i = triangles.begin(); i !=
triangles.end(); ++i)
   {n=triangles.size();} */
  fwrite((char*) &head, sizeof(char),80,fstl);
  fwrite((char*) &n, sizeof(n),1,fstl);
  for (std::list<TRIANGLE> ::iterator it = triangles.begin(); it !=
triangles.end(); ++it)
   { TRIANGLE ves = *it;
     convertor = (float) ves.normal.x;
 fwrite (( char*) &convertor, sizeof(convertor),1, fstl);
    // fprintf(fstl, "%f", convertor);
     convertor = (float) ves.normal.y;
 fwrite (( char*) &convertor, sizeof(convertor),1, fstl);
  //   fprintf(fstl, "%f", convertor);
     convertor = (float) ves.normal.z;
 fwrite (( char*) &convertor, sizeof(convertor),1, fstl);
   //  fprintf(fstl, "%f", convertor);
```

```
     convertor = (float) ves.p[0].x;
 fwrite (( char*) &convertor, sizeof(convertor),1, fstl);
    // fprintf(fstl, "%f", convertor);
     convertor = (float) ves.p[0].y;
 fwrite (( char*) &convertor, sizeof(convertor),1, fstl);
   //  fprintf(fstl, "%f", convertor);
     convertor = (float) ves.p[0].z;
 fwrite (( char*) &convertor, sizeof(convertor),1, fstl);
    // fprintf(fstl, "%f", convertor);
     convertor = (float) ves.p[1].x;
 fwrite (( char*) &convertor, sizeof(convertor),1, fstl);
   //  fprintf(fstl, "%f", convertor);
     convertor = (float) ves.p[1].y;
 fwrite (( char*) &convertor, sizeof(convertor),1, fstl);
   //  fprintf(fstl, "%f", convertor);
     convertor = (float) ves.p[1].z;
 fwrite (( char*) &convertor, sizeof(convertor),1, fstl);
   //  fprintf(fstl, "%f", convertor);
     convertor = (float) ves.p[2].x;
 fwrite (( char*) &convertor, sizeof(convertor),1, fstl);
   //  fprintf(fstl, "%f", convertor);
     convertor = (float) ves.p[2].y;
 fwrite (( char*) &convertor, sizeof(convertor),1, fstl);
   //  fprintf(fstl, "%f", convertor);
     convertor = (float) ves.p[2].z;
 fwrite (( char*) &convertor, sizeof(convertor),1, fstl);
   //  fprintf(fstl, "%f", convertor);
 fwrite (( char*) &attribute, sizeof(attribute),1, fstl);
 //    fprintf(fstl, "%h", attribute);
  }
  fclose(fstl);
}


/*****************************************************************/
/************************** spline functions *****************/
/****************************************************************/


// ---------------------------------------------------------------------
--
// implementation part, which could be separated into a cpp file
// ---------------------------------------------------------------------
--


// band_matrix implementation
// -------------------------

band_matrix::band_matrix(int dim, int n_u, int n_l)
{
    resize(dim, n_u, n_l);
```

```
}
void band_matrix::resize(int dim, int n_u, int n_l)
{
    assert(dim>0);
    assert(n_u>=0);
    assert(n_l>=0);
    m_upper.resize(n_u+1);
    m_lower.resize(n_l+1);
    for(size_t i=0; i<m_upper.size(); i++) {
        m_upper[i].resize(dim);
    }
    for(size_t i=0; i<m_lower.size(); i++) {
        m_lower[i].resize(dim);
    }
}
int band_matrix::dim() const
{
    if(m_upper.size()>0) {
        return m_upper[0].size();
    } else {
        return 0;
    }
}


// defines the new operator (), so that we can access the elements
// by A(i,j), index going from i=0,...,dim()-1
float & band_matrix::operator () (int i, int j)
{
    int k=j-i;       // what band is the entry
    assert( (i>=0) && (i<dim()) && (j>=0) && (j<dim()) );
    assert( (-num_lower()<=k) && (k<=num_upper()) );
    // k=0 -> diogonal, k<0 lower left part, k>0 upper right part
    if(k>=0)   return m_upper[k][i];
    else       return m_lower[-k][i];
}
float band_matrix::operator () (int i, int j) const
{
    int k=j-i;       // what band is the entry
    assert( (i>=0) && (i<dim()) && (j>=0) && (j<dim()) );
    assert( (-num_lower()<=k) && (k<=num_upper()) );
    // k=0 -> diogonal, k<0 lower left part, k>0 upper right part
    if(k>=0)   return m_upper[k][i];
    else       return m_lower[-k][i];
}
// second diag (used in LU decomposition), saved in m_lower
float band_matrix::saved_diag(int i) const
{
    assert( (i>=0) && (i<dim()) );
    return m_lower[0][i];
}
float & band_matrix::saved_diag(int i)
```

```
{
    assert( (i>=0) && (i<dim()) );
    return m_lower[0][i];
}

// LR-Decomposition of a band matrix
void band_matrix::lu_decompose()
{
    int  i_max,j_max;
    int  j_min;
    float x;

    // preconditioning
    // normalize column i so that a_ii=1
    for(int i=0; i<this->dim(); i++) {
        assert(this->operator()(i,i)!=0.0);
        this->saved_diag(i)=1.0/this->operator()(i,i);
        j_min=std::max(0,i-this->num_lower());
        j_max=std::min(this->dim()-1,i+this->num_upper());
        for(int j=j_min; j<=j_max; j++) {
            this->operator()(i,j) *= this->saved_diag(i);
        }
        this->operator()(i,i)=1.0;          // prevents rounding
errors
    }

    // Gauss LR-Decomposition
    for(int k=0; k<this->dim(); k++) {
        i_max=std::min(this->dim()-1,k+this->num_lower());  //
num_lower not a mistake!
        for(int i=k+1; i<=i_max; i++) {
            assert(this->operator()(k,k)!=0.0);
            x=-this->operator()(i,k)/this->operator()(k,k);
            this->operator()(i,k)=-x;                         //
assembly part of L
            j_max=std::min(this->dim()-1,k+this->num_upper());
            for(int j=k+1; j<=j_max; j++) {
                // assembly part of R
                this->operator()(i,j)=this->operator()(i,j)+x*this-
>operator()(k,j);
            }
        }
    }
}
// solves Ly=b
std::vector<float> band_matrix::l_solve(const std::vector<float>& b)
const
{
    assert( this->dim()==(int)b.size() );
    std::vector<float> x(this->dim());
    int j_start;
    float sum;
    for(int i=0; i<this->dim(); i++) {
        sum=0;
        j_start=std::max(0,i-this->num_lower());
        for(int j=j_start; j<i; j++) sum += this-
>operator()(i,j)*x[j];
        x[i]=(b[i]*this->saved_diag(i)) - sum;
    }
    return x;
}
// solves Rx=y
std::vector<float> band_matrix::r_solve(const std::vector<float>& b)
const
{
    assert( this->dim()==(int)b.size() );
    std::vector<float> x(this->dim());
    int j_stop;
    float sum;
    for(int i=this->dim()-1; i>=0; i--) {
        sum=0;
        j_stop=std::min(this->dim()-1,i+this->num_upper());
        for(int j=i+1; j<=j_stop; j++) sum += this-
>operator()(i,j)*x[j];
        x[i]=( b[i] - sum ) / this->operator()(i,i);
    }
    return x;
}

std::vector<float> band_matrix::lu_solve(const std::vector<float>& b,
        bool is_lu_decomposed)
{
    assert( this->dim()==(int)b.size() );
    std::vector<float>  x,y;
    if(is_lu_decomposed==false) {
        this->lu_decompose();
    }
    y=this->l_solve(b);
    x=this->r_solve(y);
    return x;
}




// spline implementation
// -----------------------

void spline::set_boundary(spline::bd_type left, float left_value,
                          spline::bd_type right, float right_value,
                          bool force_linear_extrapolation)
{
//    assert(m_x.size()==0);          // set_points() must not have
happened yet
```

```
    m_left=left;
    m_right=right;
    m_left_value=left_value;
    m_right_value=right_value;
    m_force_linear_extrapolation=force_linear_extrapolation;
}


void spline::set_points(const std::vector<float>& x,
                        const std::vector<float>& y, bool
cubic_spline)
{
    assert(x.size()==y.size());
    assert(x.size()>2);
    m_x=x;
    m_y=y;
    int   n=x.size();
    // TODO: maybe sort x and y, rather than returning an error
   /* for(int i=0; i<n-1; i++) {
        assert(m_x[i]<m_x[i+1]);
    }*/

    if(cubic_spline==true) { // cubic spline interpolation
        // setting up the matrix and right hand side of the equation
system
        // for the parameters b[]
        band_matrix A(n,1,1);
        std::vector<float>  rhs(n);
        for(int i=1; i<n-1; i++) {
            A(i,i-1)=1.0/3.0*(x[i]-x[i-1]);
            A(i,i)=2.0/3.0*(x[i+1]-x[i-1]);
            A(i,i+1)=1.0/3.0*(x[i+1]-x[i]);
            rhs[i]=(y[i+1]-y[i])/(x[i+1]-x[i]) - (y[i]-y[i-1])/(x[i]-
x[i-1]);
        }
        // boundary conditions
        if(m_left == spline::second_deriv) {
            // 2*b[0] = f''
            A(0,0)=2.0;
            A(0,1)=0.0;
            rhs[0]=m_left_value;
        } else if(m_left == spline::first_deriv) {
            // c[0] = f', needs to be re-expressed in terms of b:
            // (2b[0]+b[1])(x[1]-x[0]) = 3 ((y[1]-y[0])/(x[1]-x[0]) -
f')
            A(0,0)=2.0*(x[1]-x[0]);
            A(0,1)=1.0*(x[1]-x[0]);
            rhs[0]=3.0*((y[1]-y[0])/(x[1]-x[0])-m_left_value);
        } else {
            assert(false);
        }
        if(m_right == spline::second_deriv) {
            // 2*b[n-1] = f''
            A(n-1,n-1)=2.0;
            A(n-1,n-2)=0.0;
            rhs[n-1]=m_right_value;
        } else if(m_right == spline::first_deriv) {
            // c[n-1] = f', needs to be re-expressed in terms of b:
            // (b[n-2]+2b[n-1])(x[n-1]-x[n-2])
            // = 3 (f' - (y[n-1]-y[n-2])/(x[n-1]-x[n-2]))
            A(n-1,n-1)=2.0*(x[n-1]-x[n-2]);
            A(n-1,n-2)=1.0*(x[n-1]-x[n-2]);
            rhs[n-1]=3.0*(m_right_value-(y[n-1]-y[n-2])/(x[n-1]-x[n-
2]));
        } else {
            assert(false);
        }

        // solve the equation system to obtain the parameters b[]
        m_b=A.lu_solve(rhs);

        // calculate parameters a[] and c[] based on b[]
        m_a.resize(n);
        m_c.resize(n);
        for(int i=0; i<n-1; i++) {
            m_a[i]=1.0/3.0*(m_b[i+1]-m_b[i])/(x[i+1]-x[i]);
            m_c[i]=(y[i+1]-y[i])/(x[i+1]-x[i])
                   - 1.0/3.0*(2.0*m_b[i]+m_b[i+1])*(x[i+1]-x[i]);
        }
    } else { // linear interpolation
        m_a.resize(n);
        m_b.resize(n);
        m_c.resize(n);
        for(int i=0; i<n-1; i++) {
            m_a[i]=0.0;
            m_b[i]=0.0;
            m_c[i]=(m_y[i+1]-m_y[i])/(m_x[i+1]-m_x[i]);
        }
    }

    // for left extrapolation coefficients
    m_b0 = (m_force_linear_extrapolation==false) ? m_b[0] : 0.0;
    m_c0 = m_c[0];

    // for the right extrapolation coefficients
    // f_{n-1}(x) = b*(x-x_{n-1})^2 + c*(x-x_{n-1}) + y_{n-1}
    float h=x[n-1]-x[n-2];
    // m_b[n-1] is determined by the boundary condition
    m_a[n-1]=0.0;
    m_c[n-1]=3.0*m_a[n-2]*h*h+2.0*m_b[n-2]*h+m_c[n-2];   // = f'_{n-
2}(x_{n-1})
    if(m_force_linear_extrapolation==true)
        m_b[n-1]=0.0;
}
```

```
float spline::operator() (float x) const
{
    size_t n=m_x.size();
    // find the closest point m_x[idx] < x, idx=0 even if x<m_x[0]
    std::vector<float>::const_iterator it;
    it=std::lower_bound(m_x.begin(),m_x.end(),x);
    int idx=std::max( int(it-m_x.begin())-1, 0);

    float h=x-m_x[idx];
    float interpol;
    if(x<m_x[0]) {
        // extrapolation to the left
        interpol=(m_b0*h + m_c0)*h + m_y[0];
    } else if(x>m_x[n-1]) {
        // extrapolation to the right
        interpol=(m_b[n-1]*h + m_c[n-1])*h + m_y[n-1];
    } else {
        // interpolation
        interpol=((m_a[idx]*h + m_b[idx])*h + m_c[idx])*h + m_y[idx];
    }
    return interpol;
}




/****************************************************************/
/********************** main *********************************/
/****************************************************************/

 int main( int argc, char** argv)
{  bool structarray;
   unsigned char  Tlevel=0;
   int t,paddedarraylength, kernalsize;
   short crop,smooth, invert;
   float y_scan, y_AM, x_z_pixel, sigma;
   char outimgname[256];
   long nTriang;
   int columns, rows, maxVal,xCropOffset, zCropOffset;
   cout <<"\n Enter the current pixel size in mm \n";
   cin>>x_z_pixel;
   cout<<"\n Enter the current y-resolution in mm \n";
   cin>>y_scan;
   cout<<"\n Enter the desired y-resolution in mm\n";
   cin>>y_AM;
   cout<<"\n Enter Threshold value \n";
   cin >> t;
   Tlevel = (unsigned char)t;
   cout <<(int) Tlevel;
   preInitializeCheck(columns, rows, maxVal, argv);
   allocateInputArray ( argc, argv, columns, rows, maxVal,
x_z_pixel,y_scan);


/*   cout<<"\n Do wou want to smooth the images ? \n Enter 1 for yes
\n 2 for No \n";
   cin>>smooth;
   if (smooth ==1)
   {  cout <<"\n Enter kernal size (Must be an odd number) \n";
      cin>> kernalsize;
      cout<<"\n Enter sigma value \n";
      cin>> sigma;
      for(int i=0; i<argc-1; i++)
          {smoothing(inputarray[i],sigma, kernalsize);
          }
   } */

   for(int i=0; i<argc-1;i++)
      { snprintf(outimgname, sizeof(outimgname),
"output/input_%d.pgm", i+1001);
        inputarray[i].writeImage(outimgname);
      }

   cout<<" \n Do you want to crop image? \n Enter \n 1 for YES \n 2
for NO \n";
   cin>>crop;
   if( crop == 1)
   { cout<< "\n Enter the desired number of rows \n" ;
     cin>> rows;
     cout<<"\n Enter the desired number of columns \n";
     cin >> columns;
     cout<<"\n Enter the offset in x and z direction respectively \n";
    cin >> xCropOffset >> zCropOffset;
    for(int i=0; i<argc-1; i++)
    {imageCrop(inputarray[i],rows, columns, xCropOffset, zCropOffset);
    }
   }

   cout<<"\n Do you want to invert the colors of the image \n  Enter
\n 1 for YES \n 2 for NO \n";
   cin>> invert;
   if (invert==1)
    { for(int i=0; i<argc-1;i++)
      { inversion(inputarray[i]);
      }
    }

   structarray = initialize_imagearray( y_scan, y_AM, argc , columns,
rows, maxVal, x_z_pixel);
   if(!structarray){ cout<<" \n Structure array was not initailized
properly \n "; }
```

```
  else
  { /*Switch between linear and Cubic by commenting one and
uncommentig the other */
     interpolate( y_scan, y_AM, argc);
  // cubicInterpolate( y_scan, y_AM, argc);

  for(int i=0; i<total_images;i++)
      { snprintf(outimgname, sizeof(outimgname),
"output/Output_%d.pgm", i+1001);
        imagearray[i].writeImage(outimgname);
      }

/*  //threshold switch
    for(int i=0; i<total_images;i++)
      { threshold(imagearray[i], Tlevel);
      }
*/

 // Stl of imagearray (output)
  paddedarraylength =initialize_paddedarray(imagearray,total_images);
  for(int i=0; i<paddedarraylength;i++)
      {threshold(paddedarray[i] , Tlevel);
        snprintf(outimgname, sizeof(outimgname),
"output/padded1_%d.pgm", i+1001);
        paddedarray[i].writeImage(outimgname);
     }

  nTriang = marchingCube (paddedarraylength,Tlevel);
  snprintf(outimgname, sizeof(outimgname), "output/outstlbin.stl");
  writeSTLbin(outimgname, nTriang);
  cout <<" #triangles in output = "<< nTriang <<"\n ";

  deconstruct_arrays(argc, paddedarraylength);
  std::cout<<"\n Operation completed successfull ";
  return 0;
}
}



/*****************************************************************/
/**************  tables to be used in functions****************/
/****************************************************************/

 int edgeConnection[12][12] =
{
        {0,1}, {1,2}, {2,3}, {3,0},
        {4,5}, {5,6}, {6,7}, {7,4},
        {0,4}, {1,5}, {2,6}, {3,7}
};
 int edgeTable[256]={
0x0  , 0x109, 0x203, 0x30a, 0x406, 0x50f, 0x605, 0x70c,
0x80c, 0x905, 0xa0f, 0xb06, 0xc0a, 0xd03, 0xe09, 0xf00,
0x190, 0x99 , 0x393, 0x29a, 0x596, 0x49f, 0x795, 0x69c,
0x99c, 0x895, 0xb9f, 0xa96, 0xd9a, 0xc93, 0xf99, 0xe90,
0x230, 0x339, 0x33 , 0x13a, 0x636, 0x73f, 0x435, 0x53c,
0xa3c, 0xb35, 0x83f, 0x936, 0xe3a, 0xf33, 0xc39, 0xd30,
0x3a0, 0x2a9, 0x1a3, 0xaa , 0x7a6, 0x6af, 0x5a5, 0x4ac,
0xbac, 0xaa5, 0x9af, 0x8a6, 0xfaa, 0xea3, 0xda9, 0xca0,
0x460, 0x569, 0x663, 0x76a, 0x66 , 0x16f, 0x265, 0x36c,
0xc6c, 0xd65, 0xe6f, 0xf66, 0x86a, 0x963, 0xa69, 0xb60,
0x5f0, 0x4f9, 0x7f3, 0x6fa, 0x1f6, 0xff , 0x3f5, 0x2fc,
0xdfc, 0xcf5, 0xfff, 0xef6, 0x9fa, 0x8f3, 0xbf9, 0xaf0,
0x650, 0x759, 0x453, 0x55a, 0x256, 0x35f, 0x55 , 0x15c,
0xe5c, 0xf55, 0xc5f, 0xd56, 0xa5a, 0xb53, 0x859, 0x950,
0x7c0, 0x6c9, 0x5c3, 0x4ca, 0x3c6, 0x2cf, 0x1c5, 0xcc ,
0xfcc, 0xec5, 0xdcf, 0xcc6, 0xbca, 0xac3, 0x9c9, 0x8c0,
0x8c0, 0x9c9, 0xac3, 0xbca, 0xcc6, 0xdcf, 0xec5, 0xfcc,
0xcc , 0x1c5, 0x2cf, 0x3c6, 0x4ca, 0x5c3, 0x6c9, 0x7c0,
0x950, 0x859, 0xb53, 0xa5a, 0xd56, 0xc5f, 0xf55, 0xe5c,
0x15c, 0x55 , 0x35f, 0x256, 0x55a, 0x453, 0x759, 0x650,
0xaf0, 0xbf9, 0x8f3, 0x9fa, 0xef6, 0xfff, 0xcf5, 0xdfc,
0x2fc, 0x3f5, 0xff , 0x1f6, 0x6fa, 0x7f3, 0x4f9, 0x5f0,
0xb60, 0xa69, 0x963, 0x86a, 0xf66, 0xe6f, 0xd65, 0xc6c,
0x36c, 0x265, 0x16f, 0x66 , 0x76a, 0x663, 0x569, 0x460,
0xca0, 0xda9, 0xea3, 0xfaa, 0x8a6, 0x9af, 0xaa5, 0xbac,
0x4ac, 0x5a5, 0x6af, 0x7a6, 0xaa , 0x1a3, 0x2a9, 0x3a0,
0xd30, 0xc39, 0xf33, 0xe3a, 0x936, 0x83f, 0xb35, 0xa3c,
0x53c, 0x435, 0x73f, 0x636, 0x13a, 0x33 , 0x339, 0x230,
0xe90, 0xf99, 0xc93, 0xd9a, 0xa96, 0xb9f, 0x895, 0x99c,
0x69c, 0x795, 0x49f, 0x596, 0x29a, 0x393, 0x99 , 0x190,
0xf00, 0xe09, 0xd03, 0xc0a, 0xb06, 0xa0f, 0x905, 0x80c,
0x70c, 0x605, 0x50f, 0x406, 0x30a, 0x203, 0x109, 0x0   };

 int triTable[256][16] =
{       {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -
1},
        {0, 8, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {0, 1, 9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {1, 8, 3, 9, 8, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {1, 2, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -
1},
        {0, 8, 3, 1, 2, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {9, 2, 10, 0, 2, 9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {2, 8, 3, 2, 10, 8, 10, 9, 8, -1, -1, -1, -1, -1, -1, -1},
        {3, 11, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -
1},
        {0, 11, 2, 8, 11, 0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {1, 9, 0, 2, 3, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {1, 11, 2, 1, 9, 11, 9, 8, 11, -1, -1, -1, -1, -1, -1, -1},
        {3, 10, 1, 11, 10, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {0, 10, 1, 0, 8, 10, 8, 11, 10, -1, -1, -1, -1, -1, -1, -1},
        {3, 9, 0, 3, 11, 9, 11, 10, 9, -1, -1, -1, -1, -1, -1, -1},
        {9, 8, 10, 10, 8, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {4, 7, 8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {4, 3, 0, 7, 3, 4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {0, 1, 9, 8, 4, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {4, 1, 9, 4, 7, 1, 7, 3, 1, -1, -1, -1, -1, -1, -1, -1},
        {1, 2, 10, 8, 4, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {3, 4, 7, 3, 0, 4, 1, 2, 10, -1, -1, -1, -1, -1, -1, -1},
        {9, 2, 10, 9, 0, 2, 8, 4, 7, -1, -1, -1, -1, -1, -1, -1},
        {2, 10, 9, 2, 9, 7, 2, 7, 3, 7, 9, 4, -1, -1, -1, -1},
        {8, 4, 7, 3, 11, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {11, 4, 7, 11, 2, 4, 2, 0, 4, -1, -1, -1, -1, -1, -1, -1},
        {9, 0, 1, 8, 4, 7, 2, 3, 11, -1, -1, -1, -1, -1, -1, -1},
        {4, 7, 11, 9, 4, 11, 9, 11, 2, 9, 2, 1, -1, -1, -1, -1},
        {3, 10, 1, 3, 11, 10, 7, 8, 4, -1, -1, -1, -1, -1, -1, -1},
        {1, 11, 10, 1, 4, 11, 1, 0, 4, 7, 11, 4, -1, -1, -1, -1},
        {4, 7, 8, 9, 0, 11, 9, 11, 10, 11, 0, 3, -1, -1, -1, -1},
        {4, 7, 11, 4, 11, 9, 9, 11, 10, -1, -1, -1, -1, -1, -1, -1},
        {9, 5, 4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {9, 5, 4, 0, 8, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {0, 5, 4, 1, 5, 0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {8, 5, 4, 8, 3, 5, 3, 1, 5, -1, -1, -1, -1, -1, -1, -1},
        {1, 2, 10, 9, 5, 4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {3, 0, 8, 1, 2, 10, 4, 9, 5, -1, -1, -1, -1, -1, -1, -1},
        {5, 2, 10, 5, 4, 2, 4, 0, 2, -1, -1, -1, -1, -1, -1, -1},
        {2, 10, 5, 3, 2, 5, 3, 5, 4, 3, 4, 8, -1, -1, -1, -1},
        {9, 5, 4, 2, 3, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {0, 11, 2, 0, 8, 11, 4, 9, 5, -1, -1, -1, -1, -1, -1, -1},
        {0, 5, 4, 0, 1, 5, 2, 3, 11, -1, -1, -1, -1, -1, -1, -1},
        {2, 1, 5, 2, 5, 8, 2, 8, 11, 4, 8, 5, -1, -1, -1, -1},
        {10, 3, 11, 10, 1, 3, 9, 5, 4, -1, -1, -1, -1, -1, -1, -1},
        {4, 9, 5, 0, 8, 1, 8, 10, 1, 8, 11, 10, -1, -1, -1, -1},
        {5, 4, 0, 5, 0, 11, 5, 11, 10, 11, 0, 3, -1, -1, -1, -1},
        {5, 4, 8, 5, 8, 10, 10, 8, 11, -1, -1, -1, -1, -1, -1, -1},
        {9, 7, 8, 5, 7, 9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {9, 3, 0, 9, 5, 3, 5, 7, 3, -1, -1, -1, -1, -1, -1, -1},
        {0, 7, 8, 0, 1, 7, 1, 5, 7, -1, -1, -1, -1, -1, -1, -1},
        {1, 5, 3, 3, 5, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {9, 7, 8, 9, 5, 7, 10, 1, 2, -1, -1, -1, -1, -1, -1, -1},
        {10, 1, 2, 9, 5, 0, 5, 3, 0, 5, 7, 3, -1, -1, -1, -1},
        {8, 0, 2, 8, 2, 5, 8, 5, 7, 10, 5, 2, -1, -1, -1, -1},
        {2, 10, 5, 2, 5, 3, 3, 5, 7, -1, -1, -1, -1, -1, -1, -1},
        {7, 9, 5, 7, 8, 9, 3, 11, 2, -1, -1, -1, -1, -1, -1, -1},
        {9, 5, 7, 9, 7, 2, 9, 2, 0, 2, 7, 11, -1, -1, -1, -1},
        {2, 3, 11, 0, 1, 8, 1, 7, 8, 1, 5, 7, -1, -1, -1, -1},
        {11, 2, 1, 11, 1, 7, 7, 1, 5, -1, -1, -1, -1, -1, -1, -1},
        {9, 5, 8, 8, 5, 7, 10, 1, 3, 10, 3, 11, -1, -1, -1, -1},
        {5, 7, 0, 5, 0, 9, 7, 11, 0, 1, 0, 10, 11, 10, 0, -1},
        {11, 10, 0, 11, 0, 3, 10, 5, 0, 8, 0, 7, 5, 7, 0, -1},
        {11, 10, 5, 7, 11, 5, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {10, 6, 5, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -
1},
        {0, 8, 3, 5, 10, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {9, 0, 1, 5, 10, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {1, 8, 3, 1, 9, 8, 5, 10, 6, -1, -1, -1, -1, -1, -1, -1},
        {1, 6, 5, 2, 6, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {1, 6, 5, 1, 2, 6, 3, 0, 8, -1, -1, -1, -1, -1, -1, -1},
        {9, 6, 5, 9, 0, 6, 0, 2, 6, -1, -1, -1, -1, -1, -1, -1},
        {5, 9, 8, 5, 8, 2, 5, 2, 6, 3, 2, 8, -1, -1, -1, -1},
        {2, 3, 11, 10, 6, 5, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {11, 0, 8, 11, 2, 0, 10, 6, 5, -1, -1, -1, -1, -1, -1, -1},
        {0, 1, 9, 2, 3, 11, 5, 10, 6, -1, -1, -1, -1, -1, -1, -1},
        {5, 10, 6, 1, 9, 2, 9, 11, 2, 9, 8, 11, -1, -1, -1, -1},
        {6, 3, 11, 6, 5, 3, 5, 1, 3, -1, -1, -1, -1, -1, -1, -1},
        {0, 8, 11, 0, 11, 5, 0, 5, 1, 5, 11, 6, -1, -1, -1, -1},
        {3, 11, 6, 0, 3, 6, 0, 6, 5, 0, 5, 9, -1, -1, -1, -1},
        {6, 5, 9, 6, 9, 11, 11, 9, 8, -1, -1, -1, -1, -1, -1, -1},
        {5, 10, 6, 4, 7, 8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {4, 3, 0, 4, 7, 3, 6, 5, 10, -1, -1, -1, -1, -1, -1, -1},
        {1, 9, 0, 5, 10, 6, 8, 4, 7, -1, -1, -1, -1, -1, -1, -1},
        {10, 6, 5, 1, 9, 7, 1, 7, 3, 7, 9, 4, -1, -1, -1, -1},
        {6, 1, 2, 6, 5, 1, 4, 7, 8, -1, -1, -1, -1, -1, -1, -1},
        {1, 2, 5, 5, 2, 6, 3, 0, 4, 3, 4, 7, -1, -1, -1, -1},
        {8, 4, 7, 9, 0, 5, 0, 6, 5, 0, 2, 6, -1, -1, -1, -1},
        {7, 3, 9, 7, 9, 4, 3, 2, 9, 5, 9, 6, 2, 6, 9, -1},
        {3, 11, 2, 7, 8, 4, 10, 6, 5, -1, -1, -1, -1, -1, -1, -1},
        {5, 10, 6, 4, 7, 2, 4, 2, 0, 2, 7, 11, -1, -1, -1, -1},
        {0, 1, 9, 4, 7, 8, 2, 3, 11, 5, 10, 6, -1, -1, -1, -1},
        {9, 2, 1, 9, 11, 2, 9, 4, 11, 7, 11, 4, 5, 10, 6, -1},
        {8, 4, 7, 3, 11, 5, 3, 5, 1, 5, 11, 6, -1, -1, -1, -1},
        {5, 1, 11, 5, 11, 6, 1, 0, 11, 7, 11, 4, 0, 4, 11, -1},
        {0, 5, 9, 0, 6, 5, 0, 3, 6, 11, 6, 3, 8, 4, 7, -1},
        {6, 5, 9, 6, 9, 11, 4, 7, 9, 7, 11, 9, -1, -1, -1, -1},
        {10, 4, 9, 6, 4, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {4, 10, 6, 4, 9, 10, 0, 8, 3, -1, -1, -1, -1, -1, -1, -1},
        {10, 0, 1, 10, 6, 0, 6, 4, 0, -1, -1, -1, -1, -1, -1, -1},
        {8, 3, 1, 8, 1, 6, 8, 6, 4, 6, 1, 10, -1, -1, -1, -1},
        {1, 4, 9, 1, 2, 4, 2, 6, 4, -1, -1, -1, -1, -1, -1, -1},
        {3, 0, 8, 1, 2, 9, 2, 4, 9, 2, 6, 4, -1, -1, -1, -1},
        {0, 2, 4, 4, 2, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {8, 3, 2, 8, 2, 4, 4, 2, 6, -1, -1, -1, -1, -1, -1, -1},
        {10, 4, 9, 10, 6, 4, 11, 2, 3, -1, -1, -1, -1, -1, -1, -1},
        {0, 8, 2, 2, 8, 11, 4, 9, 10, 4, 10, 6, -1, -1, -1, -1},
        {3, 11, 2, 0, 1, 6, 0, 6, 4, 6, 1, 10, -1, -1, -1, -1},
        {6, 4, 1, 6, 1, 10, 4, 8, 1, 2, 1, 11, 8, 11, 1, -1},
        {9, 6, 4, 9, 3, 6, 9, 1, 3, 11, 6, 3, -1, -1, -1, -1},
        {8, 11, 1, 8, 1, 0, 11, 6, 1, 9, 1, 4, 6, 4, 1, -1},
        {3, 11, 6, 3, 6, 0, 0, 6, 4, -1, -1, -1, -1, -1, -1, -1},
        {6, 4, 8, 11, 6, 8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {7, 10, 6, 7, 8, 10, 8, 9, 10, -1, -1, -1, -1, -1, -1, -1},
        {0, 7, 3, 0, 10, 7, 0, 9, 10, 6, 7, 10, -1, -1, -1, -1},
        {10, 6, 7, 1, 10, 7, 1, 7, 8, 1, 8, 0, -1, -1, -1, -1},
        {10, 6, 7, 10, 7, 1, 1, 7, 3, -1, -1, -1, -1, -1, -1, -1},
        {1, 2, 6, 1, 6, 8, 1, 8, 9, 8, 6, 7, -1, -1, -1, -1},
        {2, 6, 9, 2, 9, 1, 6, 7, 9, 0, 9, 3, 7, 3, 9, -1},
        {7, 8, 0, 7, 0, 6, 6, 0, 2, -1, -1, -1, -1, -1, -1, -1},
        {7, 3, 2, 6, 7, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {2, 3, 11, 10, 6, 8, 10, 8, 9, 8, 6, 7, -1, -1, -1, -1},
        {2, 0, 7, 2, 7, 11, 0, 9, 7, 6, 7, 10, 9, 10, 7, -1},
        {1, 8, 0, 1, 7, 8, 1, 10, 7, 6, 7, 10, 2, 3, 11, -1},
        {11, 2, 1, 11, 1, 7, 10, 6, 1, 6, 7, 1, -1, -1, -1, -1},
        {8, 9, 6, 8, 6, 7, 9, 1, 6, 11, 6, 3, 1, 3, 6, -1},
        {0, 9, 1, 11, 6, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {7, 8, 0, 7, 0, 6, 3, 11, 0, 11, 6, 0, -1, -1, -1, -1},
        {7, 11, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -
1},
        {7, 6, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -
1},
        {3, 0, 8, 11, 7, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {0, 1, 9, 11, 7, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {8, 1, 9, 8, 3, 1, 11, 7, 6, -1, -1, -1, -1, -1, -1, -1},
        {10, 1, 2, 6, 11, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {1, 2, 10, 3, 0, 8, 6, 11, 7, -1, -1, -1, -1, -1, -1, -1},
        {2, 9, 0, 2, 10, 9, 6, 11, 7, -1, -1, -1, -1, -1, -1, -1},
        {6, 11, 7, 2, 10, 3, 10, 8, 3, 10, 9, 8, -1, -1, -1, -1},
        {7, 2, 3, 6, 2, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {7, 0, 8, 7, 6, 0, 6, 2, 0, -1, -1, -1, -1, -1, -1, -1},
        {2, 7, 6, 2, 3, 7, 0, 1, 9, -1, -1, -1, -1, -1, -1, -1},
        {1, 6, 2, 1, 8, 6, 1, 9, 8, 8, 7, 6, -1, -1, -1, -1},
        {10, 7, 6, 10, 1, 7, 1, 3, 7, -1, -1, -1, -1, -1, -1, -1},
        {10, 7, 6, 1, 7, 10, 1, 8, 7, 1, 0, 8, -1, -1, -1, -1},
        {0, 3, 7, 0, 7, 10, 0, 10, 9, 6, 10, 7, -1, -1, -1, -1},
        {7, 6, 10, 7, 10, 8, 8, 10, 9, -1, -1, -1, -1, -1, -1, -1},
        {6, 8, 4, 11, 8, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {3, 6, 11, 3, 0, 6, 0, 4, 6, -1, -1, -1, -1, -1, -1, -1},
        {8, 6, 11, 8, 4, 6, 9, 0, 1, -1, -1, -1, -1, -1, -1, -1},
        {9, 4, 6, 9, 6, 3, 9, 3, 1, 11, 3, 6, -1, -1, -1, -1},
        {6, 8, 4, 6, 11, 8, 2, 10, 1, -1, -1, -1, -1, -1, -1, -1},
        {1, 2, 10, 3, 0, 11, 0, 6, 11, 0, 4, 6, -1, -1, -1, -1},
        {4, 11, 8, 4, 6, 11, 0, 2, 9, 2, 10, 9, -1, -1, -1, -1},
        {10, 9, 3, 10, 3, 2, 9, 4, 3, 11, 3, 6, 4, 6, 3, -1},
        {8, 2, 3, 8, 4, 2, 4, 6, 2, -1, -1, -1, -1, -1, -1, -1},
        {0, 4, 2, 4, 6, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {1, 9, 0, 2, 3, 4, 2, 4, 6, 4, 3, 8, -1, -1, -1, -1},
        {1, 9, 4, 1, 4, 2, 2, 4, 6, -1, -1, -1, -1, -1, -1, -1},
        {8, 1, 3, 8, 6, 1, 8, 4, 6, 6, 10, 1, -1, -1, -1, -1},
        {10, 1, 0, 10, 0, 6, 6, 0, 4, -1, -1, -1, -1, -1, -1, -1},
        {4, 6, 3, 4, 3, 8, 6, 10, 3, 0, 3, 9, 10, 9, 3, -1},
        {10, 9, 4, 6, 10, 4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {4, 9, 5, 7, 6, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {0, 8, 3, 4, 9, 5, 11, 7, 6, -1, -1, -1, -1, -1, -1, -1},
        {5, 0, 1, 5, 4, 0, 7, 6, 11, -1, -1, -1, -1, -1, -1, -1},
        {11, 7, 6, 8, 3, 4, 3, 5, 4, 3, 1, 5, -1, -1, -1, -1},
        {9, 5, 4, 10, 1, 2, 7, 6, 11, -1, -1, -1, -1, -1, -1, -1},
        {6, 11, 7, 1, 2, 10, 0, 8, 3, 4, 9, 5, -1, -1, -1, -1},
        {7, 6, 11, 5, 4, 10, 4, 2, 10, 4, 0, 2, -1, -1, -1, -1},
        {3, 4, 8, 3, 5, 4, 3, 2, 5, 10, 5, 2, 11, 7, 6, -1},
        {7, 2, 3, 7, 6, 2, 5, 4, 9, -1, -1, -1, -1, -1, -1, -1},
        {9, 5, 4, 0, 8, 6, 0, 6, 2, 6, 8, 7, -1, -1, -1, -1},
        {3, 6, 2, 3, 7, 6, 1, 5, 0, 5, 4, 0, -1, -1, -1, -1},
        {6, 2, 8, 6, 8, 7, 2, 1, 8, 4, 8, 5, 1, 5, 8, -1},
        {9, 5, 4, 10, 1, 6, 1, 7, 6, 1, 3, 7, -1, -1, -1, -1},
        {1, 6, 10, 1, 7, 6, 1, 0, 7, 8, 7, 0, 9, 5, 4, -1},
        {4, 0, 10, 4, 10, 5, 0, 3, 10, 6, 10, 7, 3, 7, 10, -1},
        {7, 6, 10, 7, 10, 8, 5, 4, 10, 4, 8, 10, -1, -1, -1, -1},
        {6, 9, 5, 6, 11, 9, 11, 8, 9, -1, -1, -1, -1, -1, -1, -1},
        {3, 6, 11, 0, 6, 3, 0, 5, 6, 0, 9, 5, -1, -1, -1, -1},
        {0, 11, 8, 0, 5, 11, 0, 1, 5, 5, 6, 11, -1, -1, -1, -1},
        {6, 11, 3, 6, 3, 5, 5, 3, 1, -1, -1, -1, -1, -1, -1, -1},
        {1, 2, 10, 9, 5, 11, 9, 11, 8, 11, 5, 6, -1, -1, -1, -1},
        {0, 11, 3, 0, 6, 11, 0, 9, 6, 5, 6, 9, 1, 2, 10, -1},
        {11, 8, 5, 11, 5, 6, 8, 0, 5, 10, 5, 2, 0, 2, 5, -1},
        {6, 11, 3, 6, 3, 5, 2, 10, 3, 10, 5, 3, -1, -1, -1, -1},
        {5, 8, 9, 5, 2, 8, 5, 6, 2, 3, 8, 2, -1, -1, -1, -1},
        {9, 5, 6, 9, 6, 0, 0, 6, 2, -1, -1, -1, -1, -1, -1, -1},
        {1, 5, 8, 1, 8, 0, 5, 6, 8, 3, 8, 2, 6, 2, 8, -1},
        {1, 5, 6, 2, 1, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {1, 3, 6, 1, 6, 10, 3, 8, 6, 5, 6, 9, 8, 9, 6, -1},
        {10, 1, 0, 10, 0, 6, 9, 5, 0, 5, 6, 0, -1, -1, -1, -1},
        {0, 3, 8, 5, 6, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {10, 5, 6, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -
1},
        {11, 5, 10, 7, 5, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {11, 5, 10, 11, 7, 5, 8, 3, 0, -1, -1, -1, -1, -1, -1, -1},
        {5, 11, 7, 5, 10, 11, 1, 9, 0, -1, -1, -1, -1, -1, -1, -1},
        {10, 7, 5, 10, 11, 7, 9, 8, 1, 8, 3, 1, -1, -1, -1, -1},
        {11, 1, 2, 11, 7, 1, 7, 5, 1, -1, -1, -1, -1, -1, -1, -1},
        {0, 8, 3, 1, 2, 7, 1, 7, 5, 7, 2, 11, -1, -1, -1, -1},
        {9, 7, 5, 9, 2, 7, 9, 0, 2, 2, 11, 7, -1, -1, -1, -1},
        {7, 5, 2, 7, 2, 11, 5, 9, 2, 3, 2, 8, 9, 8, 2, -1},
        {2, 5, 10, 2, 3, 5, 3, 7, 5, -1, -1, -1, -1, -1, -1, -1},
        {8, 2, 0, 8, 5, 2, 8, 7, 5, 10, 2, 5, -1, -1, -1, -1},
        {9, 0, 1, 5, 10, 3, 5, 3, 7, 3, 10, 2, -1, -1, -1, -1},
        {9, 8, 2, 9, 2, 1, 8, 7, 2, 10, 2, 5, 7, 5, 2, -1},
        {1, 3, 5, 3, 7, 5, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {0, 8, 7, 0, 7, 1, 1, 7, 5, -1, -1, -1, -1, -1, -1, -1},
        {9, 0, 3, 9, 3, 5, 5, 3, 7, -1, -1, -1, -1, -1, -1, -1},
        {9, 8, 7, 5, 9, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {5, 8, 4, 5, 10, 8, 10, 11, 8, -1, -1, -1, -1, -1, -1, -1},
        {5, 0, 4, 5, 11, 0, 5, 10, 11, 11, 3, 0, -1, -1, -1, -1},
        {0, 1, 9, 8, 4, 10, 8, 10, 11, 10, 4, 5, -1, -1, -1, -1},
        {10, 11, 4, 10, 4, 5, 11, 3, 4, 9, 4, 1, 3, 1, 4, -1},
        {2, 5, 1, 2, 8, 5, 2, 11, 8, 4, 5, 8, -1, -1, -1, -1},
        {0, 4, 11, 0, 11, 3, 4, 5, 11, 2, 11, 1, 5, 1, 11, -1},
        {0, 2, 5, 0, 5, 9, 2, 11, 5, 4, 5, 8, 11, 8, 5, -1},
        {9, 4, 5, 2, 11, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {2, 5, 10, 3, 5, 2, 3, 4, 5, 3, 8, 4, -1, -1, -1, -1},
        {5, 10, 2, 5, 2, 4, 4, 2, 0, -1, -1, -1, -1, -1, -1, -1},
        {3, 10, 2, 3, 5, 10, 3, 8, 5, 4, 5, 8, 0, 1, 9, -1},
        {5, 10, 2, 5, 2, 4, 1, 9, 2, 9, 4, 2, -1, -1, -1, -1},
        {8, 4, 5, 8, 5, 3, 3, 5, 1, -1, -1, -1, -1, -1, -1, -1},
        {0, 4, 5, 1, 0, 5, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {8, 4, 5, 8, 5, 3, 9, 0, 5, 0, 3, 5, -1, -1, -1, -1},
        {9, 4, 5, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {4, 11, 7, 4, 9, 11, 9, 10, 11, -1, -1, -1, -1, -1, -1, -1},
        {0, 8, 3, 4, 9, 7, 9, 11, 7, 9, 10, 11, -1, -1, -1, -1},
        {1, 10, 11, 1, 11, 4, 1, 4, 0, 7, 4, 11, -1, -1, -1, -1},
        {3, 1, 4, 3, 4, 8, 1, 10, 4, 7, 4, 11, 10, 11, 4, -1},
        {4, 11, 7, 9, 11, 4, 9, 2, 11, 9, 1, 2, -1, -1, -1, -1},
        {9, 7, 4, 9, 11, 7, 9, 1, 11, 2, 11, 1, 0, 8, 3, -1},
        {11, 7, 4, 11, 4, 2, 2, 4, 0, -1, -1, -1, -1, -1, -1, -1},
        {11, 7, 4, 11, 4, 2, 8, 3, 4, 3, 2, 4, -1, -1, -1, -1},
        {2, 9, 10, 2, 7, 9, 2, 3, 7, 7, 4, 9, -1, -1, -1, -1},
        {9, 10, 7, 9, 7, 4, 10, 2, 7, 8, 7, 0, 2, 0, 7, -1},
        {3, 7, 10, 3, 10, 2, 7, 4, 10, 1, 10, 0, 4, 0, 10, -1},
        {1, 10, 2, 8, 7, 4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {4, 9, 1, 4, 1, 7, 7, 1, 3, -1, -1, -1, -1, -1, -1, -1},
        {4, 9, 1, 4, 1, 7, 0, 8, 1, 8, 7, 1, -1, -1, -1, -1},
        {4, 0, 3, 7, 4, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {4, 8, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {9, 10, 8, 10, 11, 8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {3, 0, 9, 3, 9, 11, 11, 9, 10, -1, -1, -1, -1, -1, -1, -1},
        {0, 1, 10, 0, 10, 8, 8, 10, 11, -1, -1, -1, -1, -1, -1, -1},
        {3, 1, 10, 11, 3, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {1, 2, 11, 1, 11, 9, 9, 11, 8, -1, -1, -1, -1, -1, -1, -1},
        {3, 0, 9, 3, 9, 11, 1, 2, 9, 2, 11, 9, -1, -1, -1, -1},
        {0, 2, 11, 8, 0, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {3, 2, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -
1},
        {2, 3, 8, 2, 8, 10, 10, 8, 9, -1, -1, -1, -1, -1, -1, -1},
        {9, 10, 2, 0, 9, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {2, 3, 8, 2, 8, 10, 0, 1, 8, 1, 10, 8, -1, -1, -1, -1},
        {1, 10, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -
1},
        {1, 3, 8, 9, 1, 8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {0, 9, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {0, 3, 8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
        {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -
1}};
```