MPHYGB24 - MATLAB Coursework Assignment

Student Number - 14062340

26th January 2018

Task 1

The function **LoadDICOMVolume.m** makes use of the inbuilt MATLAB function **uigetdir** to allow the user to load a DICOM volume by opening a standard folder selection dialog box. This step will not occur if a folder has been specified as an optional second argument to the function.

Error checking occurs at the beginning of the function. If the user clicks cancel in the folder selection dialog box a modal error dialog box is created which displays an appropriate message. Program execution is blocked until the user closes the box, and the function then returns early. A similar situation occurs if the user either chooses or specifies a folder which exists but contains no DICOM files.

The function also checks that the the range of Z values to be loaded from the DICOM folder is valid. If the range is not valid then a modal error dialog box is created which informs the user of the number of Z slices in the DICOM folder.

Therefore the following exceptions can be handled by **LoadDICOMVolume.m**:

- The user clicks cancel in the folder selection dialog box.
- The user selects, or specifies as an optional second function argument, a folder which exists, but which contains no DICOM files.
- The user specifies a range of slices with a start value of less than 1.
- The user specifies a range of slices with an end value either greater than the number of Z slices in the DICOM folder, or less than the start value.

Once the error checks have been passed, a status update is displayed to the user in the command window, and the DICOM volume is loaded according the to specifications in the question sheet. The volume metadata is read using **dicominfo**, and each slice of the volume is loaded using **dicomread**. The voxel dimensions are then extracted from the metadata and the slices are correctly reordered within the volume. The volume is then truncated to the start and end slice values specified in the first function argument. A status update is then displayed to the user in the command window once the DICOM volume has been loaded.

test_LoadDICOMVolume.m is a script which tests **LoadDICOMVolume.m** and demonstrates its correct operation. When **LoadDICOMVolume.m** is first invoked in line 21, the error messages can be checked by clicking cancel in the folder selection box or by choosing a folder which contains no DICOM files. If a DICOM volume is loaded successfully its properties are printed to the command window, showing the **VoxelDimensions** field and the size and data type of the **ImageData** field. All of the slices are looped through and displayed in a figure to confirm that the volume has been loaded correctly.

The above procedure is then repeated using the optional second function argument to specify the folder from which to load the DICOM files. Error message checking then occurs by specifying a folder which contains no DICOM files, specifying a Z range start value of 0, a Z range end value of 182 (greater than the maximum of 181 in this case), and a Z range end value which is less than the Z range start value. Finally the ability to load only one Z slice from the volume is checked.

Task 2

ComputeOrthogonalSlice.m makes use of the inbuilt MATLAB function **interp3** to perform 3D interpolation of a loaded DICOM volume on a specified plane and orthogonal position, with a specified in-plane resolution. **interp3** was chosen as it supports the three interpolation methods specified in the question. With regard to an orientation and indexing convention to be used in this report and in the MATLAB files associated with it, the Z-axis corresponds to the longitudinal axis of the subject (with 0 toward the head, and the first slice centred around 0 mm). The X-axis corresponds to the left-right axis of the subject (with 0 toward the left, and the last slice centred around 0 mm). The Y-axis corresponds to the cranial-caudal axis of the patient (with 0 toward the front, and the last slice centred around 0 mm). These X and Y conventions where chosen in an attempt to be consistent with Figure 1 of the question sheet.

Using this convention, the size of the volume in mm could then be calculated as follows:

```
% To get size of image as 3D vector in form of (y,x,z) i.e.
% (number rows, number of columns, number of slices)
image_dim = size(Image.ImageData);
% To get size of voxel as 3D vector in mm in form of (y,x,z) i.e.
% (voxel height, voxel width, voxel length)
vox_dim = Image.VoxelDimensions;
% vox_dim = [0.9766, 0.9766, 1]
% To get size of image as 3D vector in mm in form of (y,x,z) i.e.
% (image height, image width, image length)
image_size = (image_dim - [1 1 1]).*vox_dim;
```

A 3D sampling grid of appropriate in-plane resolution for interpolation was formed using the **meshgrid** function. Nearest-neighbour and linear interpolation require 2 grid points in each dimension. Therefore, for these two interpolation methods, the two orthogonal slices either side the orthogonal query point were selected for interpolation using the 3D sampling grid described above. Cubic spline interpolation requires 4 grid points in each dimension, therefore, when this interpolation method was used, two orthogonal slices either side (four in total) of the orthogonal query point were selected. The query point was adjusted at the image boundary for all interpolation methods.

This technique of extracting either two or four slices from the original volume made the function run much faster than when interpolating the entire volume. The test script **test_ComputeOrthogonalSlice** took 19.45 seconds to run using this technique; however, when interpolating over the entire volume the test script took 290 seconds to run (80 seconds of which was occupied by one cubic spline interpolation in the XY plane).

ComputeOrthogonalSlice.m was broken down into a series of orientation specific and interpolation method specific tasks. It was important to use the **squeeze** and **imresize** functions to ensure that a 2D slice of appropriate size was returned (this latter function corrected for sampling at a lower in-plane resolution than the original voxel dimensions).

Error checking also occurs within the function, with a modal error dialog box created when an orthogonal position (specified in mm) outside the volume dimensions is used as a function argument. In this situation program execution is blocked until the user closes the box, and the function then returns early.

test_ComputeOrthogonalSlice.m tests the above function by displaying slices in each of the three image view planes, using each of the three interpolation methods, using each of the following three sampling resolutions:

a) Both slice pixel dimensions are equal to the smallest in-plane pixel dimension of the original image.

b) Both slice pixel dimensions are equal to the largest in-plane pixel dimension of the original image.

c) Both slice pixel dimensions are equal to four times the largest in-plane pixel dimension of the original image.

The voxel dimensions used to determine the in-plane pixel dimensions of the original image are shown in line 7 of the code snippet above, in the form of (Y, X, Z).

The slices were displayed with the correct scaling using **daspect** in order to adjust the figure scaling by the appropriate voxel dimensions (the order of which depended upon the image slice plane). The slices were also rotated and flipped as necessary so that they are shown in the correct radiological orientation.

The following three figures were produced for the XY view plane:

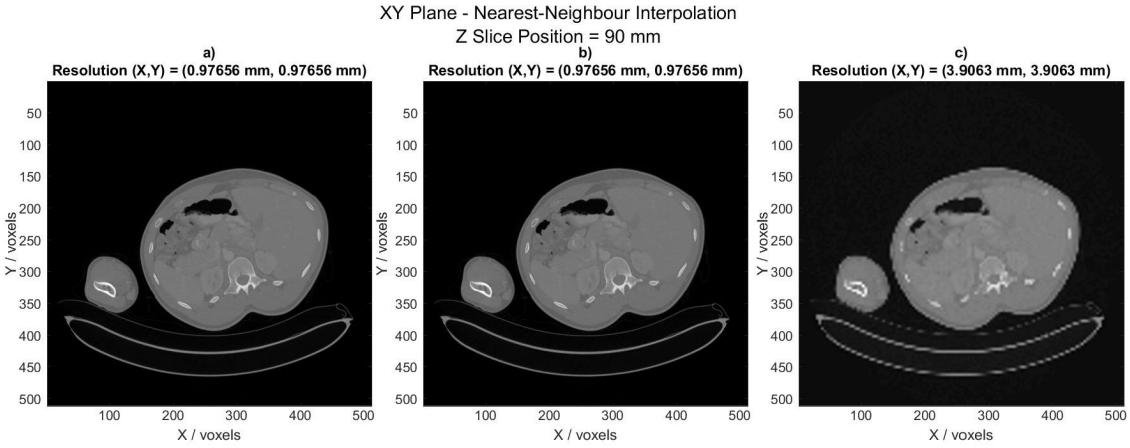


Figure 1: The first XY plane figure.

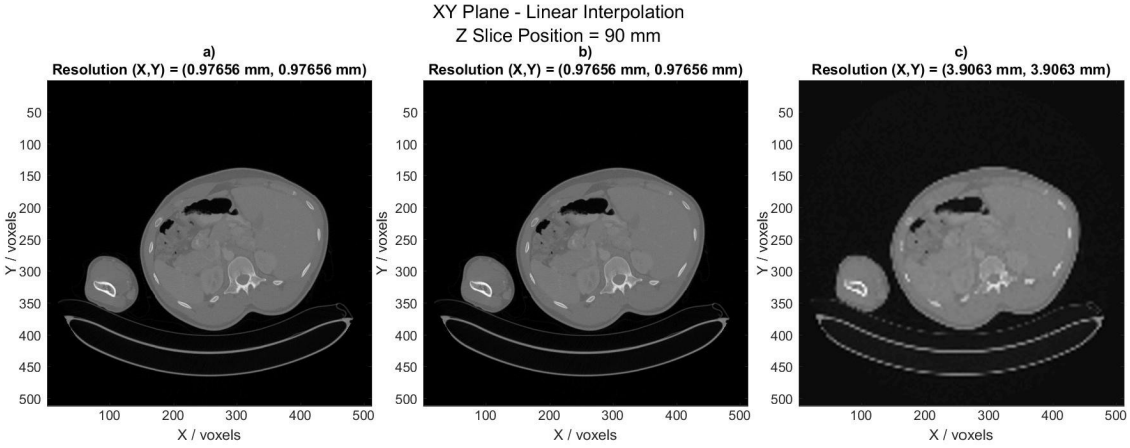


Figure 2: The second XY plane figure.

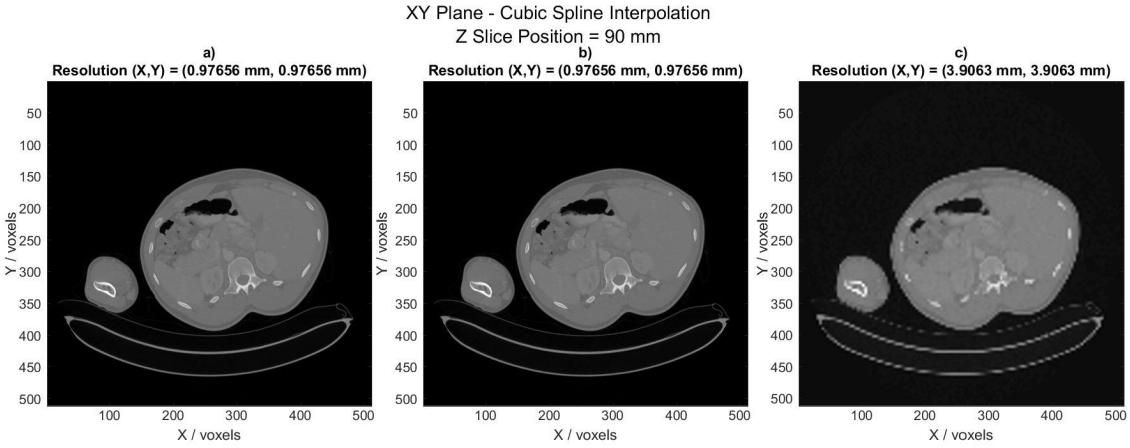


Figure 3: The third XY plane figure.

The following three figures were produced for the YZ view plane:

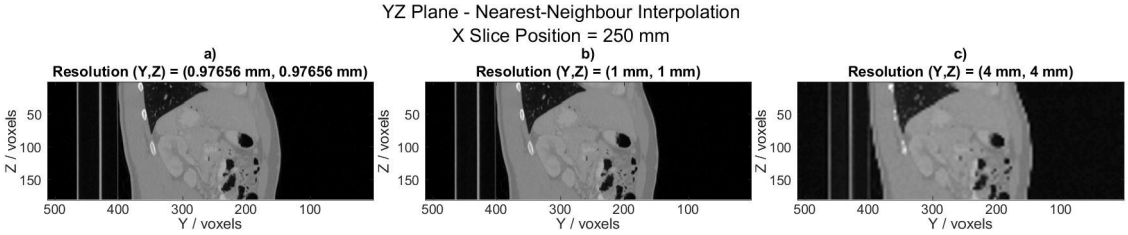


Figure 4: The first YZ plane figure.

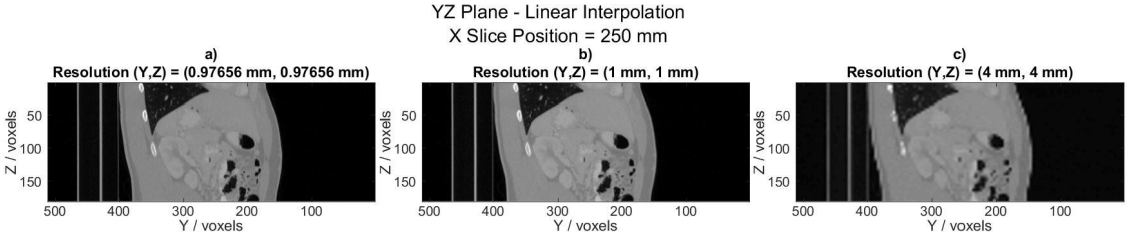


Figure 5: The second YZ plane figure.

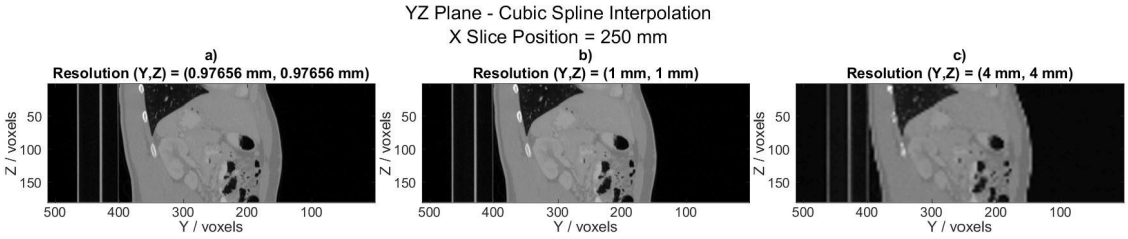


Figure 6: The third YZ plane figure.

The following three figures were produced for the XZ view plane:

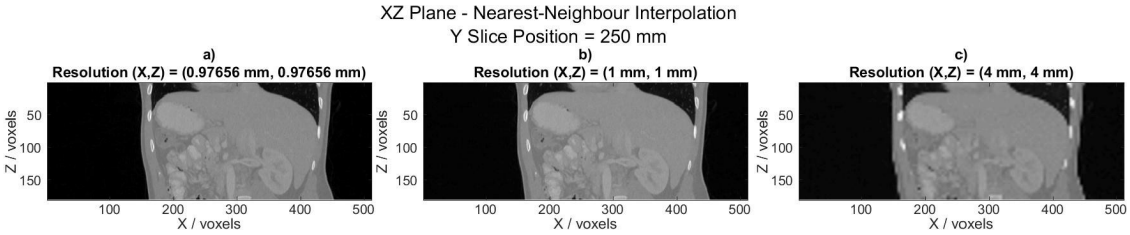


Figure 7: The first XZ plane figure.

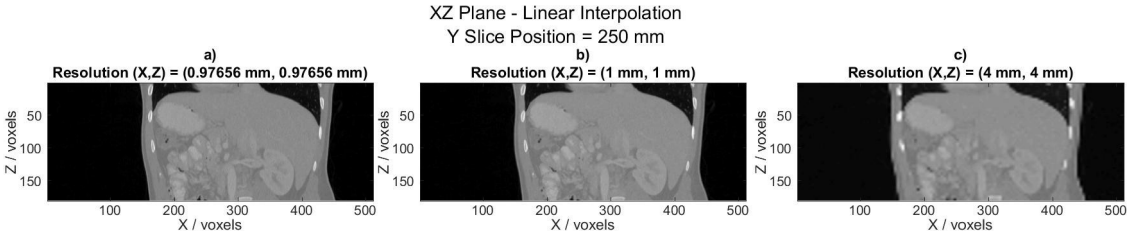


Figure 8: The second XZ plane figure.

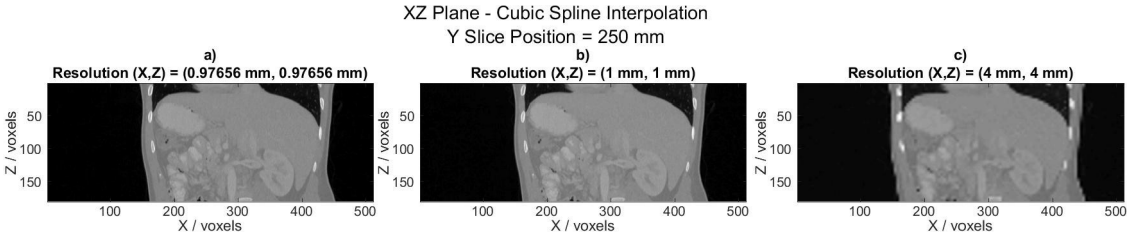


Figure 9: The third XZ plane figure.

The images for parts a) and b) of the above nine figures are all very similar, as the pixel resolutions are either the same or only very subtly different. However, the images for part c) are noticeably pixelated due to the decreased sampling resolution. No difference between the three interpolation methods is visually discernible for each view plane.

The last few lines of **test_ComputeOrthogonalSlice.m** test the modal error dialog boxes that can be produced by **ComputeOrthogonalSlice.m**. These are tested in situations where the input slice position is either negative or exceeds the relevant image dimension. The whole of this test script is enclosed by a **try** and **catch** block, so that errors resulting from incorrect loading of the DICOM volume do not cause the program to crash.

Task 3

A graphical user interface (GUI) was implemented by using **GUIDE** (a MATLAB GUI design environment) to produce two files: **SliceViewer.m** and **SliceViewer.fig**. The GUI was implemented so that after loading

a DICOM volume, the default view was the central XY plane, with an in-plane resolution of the X and Y voxel dimensions of the original image, respectively. This is also the view that the GUI defaults to when the reset button is clicked (if a volume is loaded). The functionality of the GUI was implemented as described in the question, with the following additions.

- The minor increment on the slice position slider was set to one slice in the respective orthogonal dimension, with the major increment set to ten slices.
- The resolution and slice position edit boxes always show to a 3dp precision (this seemed like a reasonable level of accuracy given that the X and Y voxel dimensions were both stated as being 0.9765625 mm).
- The range of allowable resolutions was set from 0.5 mm to 20 mm. Input values outside this range would result in the resolution being set to either the minimum or maximum value, respectively, and a modal error dialog box being created.
- Likewise, if an input value outside the allowable range was entered into the slice position edit box, the slice position would be set to either the minimum or maximum value, respectively, and a modal error dialog box would be created.
- A drop-down menu was also added to specify which of the three interpolation methods should be used to sample the volume.
- When the view plane is changed, the default display output is the central slice with the maximum achievable in-plane resolution.
- When any of the controls (other than the load button) is used when a volume is not loaded, a warning dialog box is created advising the user of such.

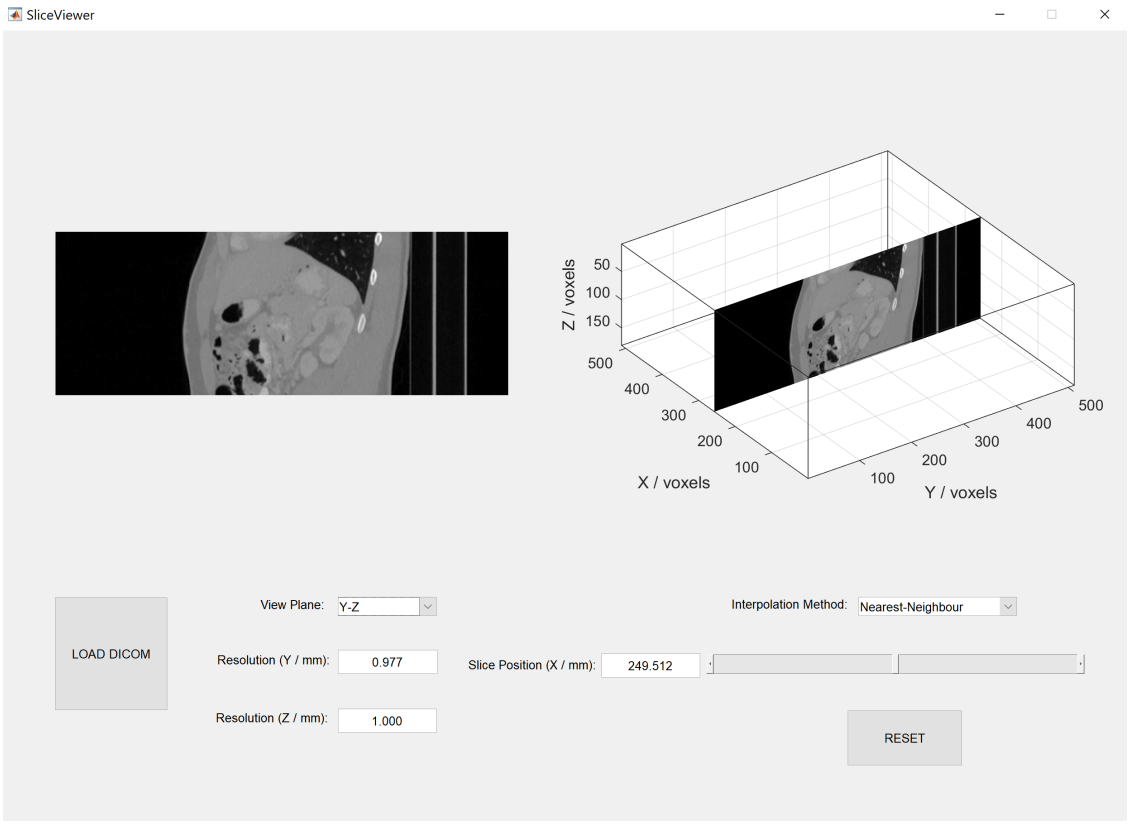


Figure 10: Screenshot of the default YZ view plane GUI output.

The 2D view on the left of Figure 10 was obtained using the **imagesc** function (as per task 2). The 3D wire-frame representation on the right of Figure 10 was obtained by inserting the 2D image on the left into a 3D matrix of zeros of identical dimensions to the original volume, at the appropriate location. A volumetric slice plot of this matrix was obtained using the **slice** function, with the plot and axes properties adjusted accordingly. This included setting the `edgeColour` property of the plot to `none`, setting the `Box` and `BoxStyle` and properties of the axes to `on` and `full`, respectively, as well as adjusting the view angle.

Task 4

In order to perform Gaussian blurring of an image, a zeroth order derivative Gaussian distribution must first be produced. This was achieved using the function **CompGaussian.m** (submitted for Exercise Sheet #1) which computes a 1D Gaussian kernel of a specified width. This was used by the function **GaussianImfilterSep.m** (adapted from the function submitted for Exercise Sheet #4) which convolves the image with a 1D or 2D Gaussian kernel, in order to perform Gaussian blurring. This function was adapted so that it could blur a 2D image in one or both in-plane dimensions, as dictated by its input arguments.

The function **BlurSlices.m** was fairly straightforward to construct. A slab of one or more 2D slices of a certain view plane orientation is input into the function, as is a specification of which dimension/s to blur the image in, along with the width/s of any one or two of the Gaussian kernel/s to be used (i.e. σ_x, σ_y or σ_z). The function then loops through each slice of the slab, blurring each slice individually, before returning these blurred slices as a recombined slab. A test script **test_BlurSlices.m** was written which tests the ability of **BlurSlices.m** to handle slabs in each of the three view plane orientations, with either 1D (in either in-plane direction) or 2D blurring, on single or multiple slice slabs. Two of the six figures produced by this test script are shown below.

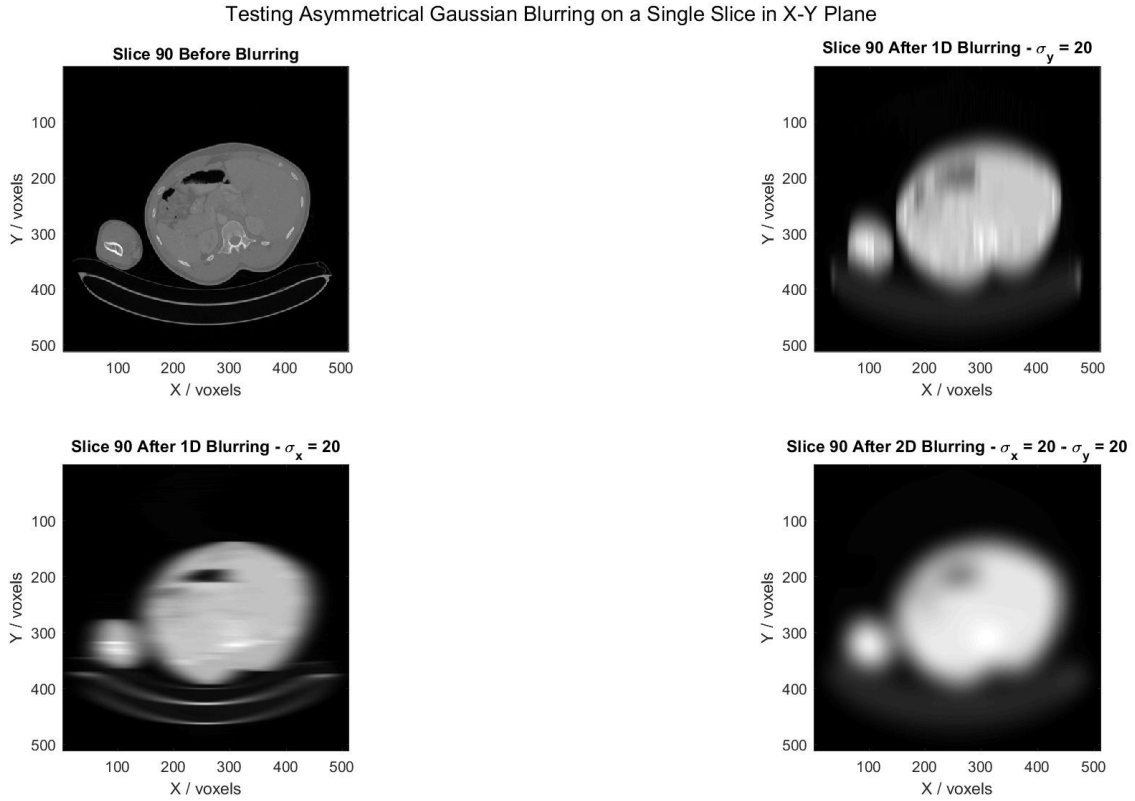


Figure 11: Both 1D blurring options and 2D blurring on a one slice slab in the XY plane.

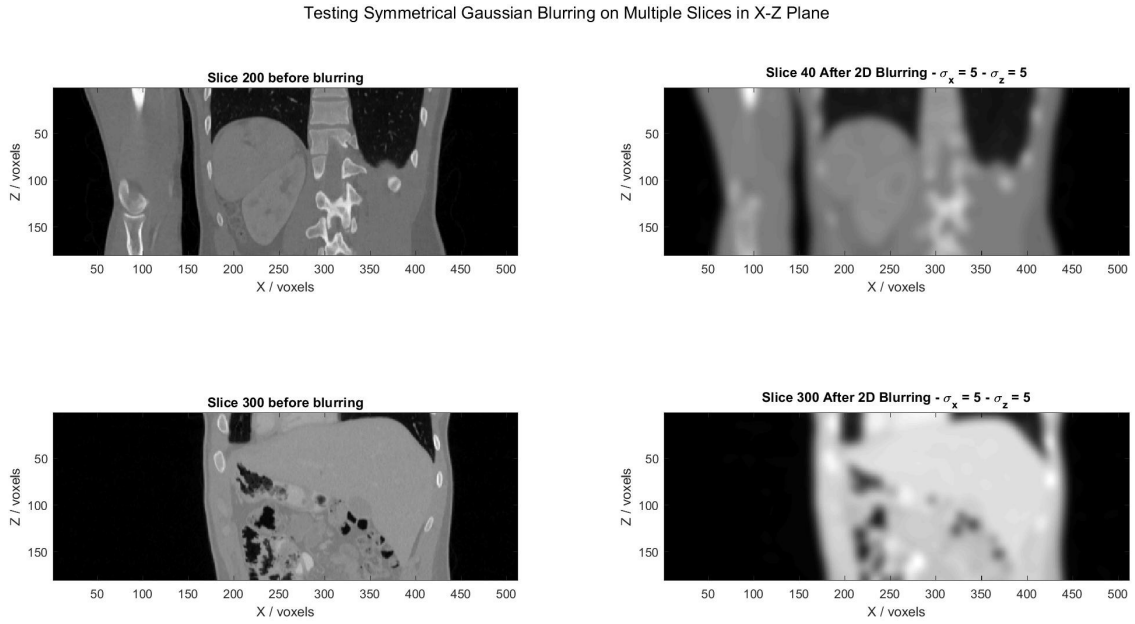


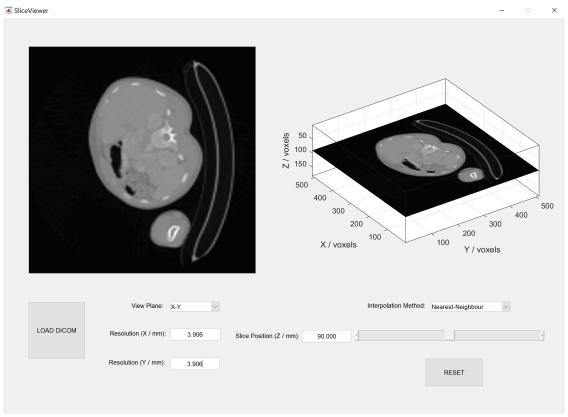
Figure 12: 2D blurring of a two slice slab in the XZ plane.

Task 5

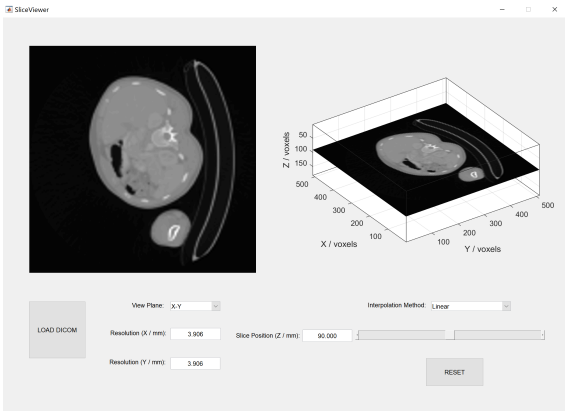
A new function, **ComputeOrthogonalSliceBlur.m**, was written, which is capable of blurring the two or four slice slab centred around the orthogonal query point, prior to 3D interpolation. This function blurred all of the slices in the slab automatically if one or both of the two in-plane sampling resolutions exceeded the respective voxel dimensions of the original image. For example, in the XY plane, if the ratio of the the X sampling resolution to the original voxel width was greater than one, then blurring occurred in the X direction (the threshold for blurring was actually set to 1.001, due to the 3DP accuracy of the GUI resolution edit boxes).

It was assumed that Δ in the equation in question 5 referred to this *dimensionless ratio* of the target pixel dimension in the corresponding direction, to the original voxel dimension in the corresponding direction. **ComputeOrthogonalSliceBlur.m** calculates this Δ value for both in-plane pixel dimensions, which is then converted to a σ value for each in-plane dimension (using the equation given in the question sheet). If either of these Δ values exceeds the threhold for blurring, then Gaussian blurring of slices occurs in one or both in-plane dimensions (using the appropriate σ values as calculated above), prior to 3D interpolation.

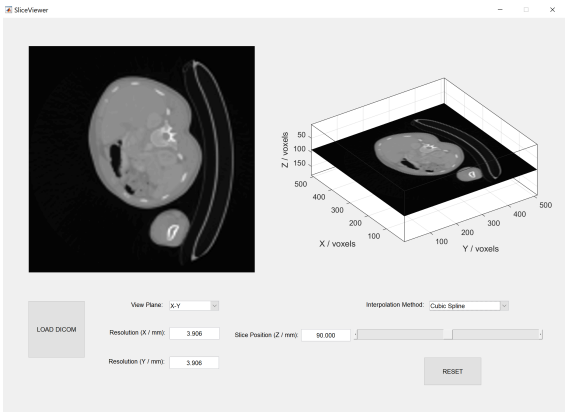
The GUI function **SliceViewer.m** was updated to use **ComputeOrthogonalSliceBlur.m**, rather than **ComputeOrthogonalSlice.m**, and the test carried out in Task 2 c) was repeated by using the GUI. The results, as requested, are incorporated below, and show the effect of incorporating blurring when the resolution of the displayed slice is relatively low (again this is performed for each of the three view planes, and for each of the three interpolation methods).



(a) Nearest-neighbour interpolation.

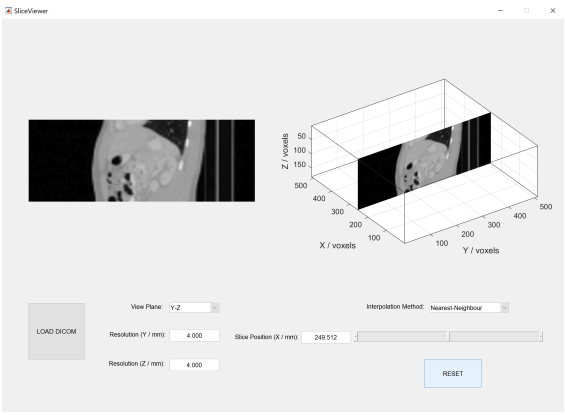


(b) Linear Interpolation.

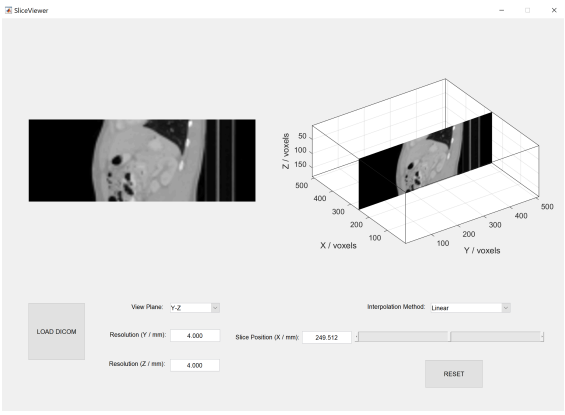


(c) Cubic Spline Interpolation.

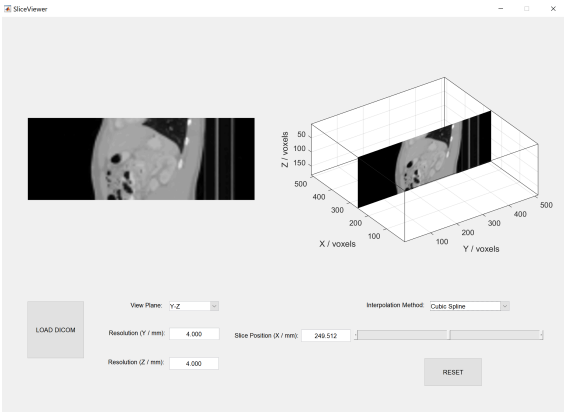
Figure 13: Screenshots of the GUI in the XY view plane, using the in-plane resolutions of Task 2 c), for each interpolation method.



(a) Nearest-neighbour interpolation.

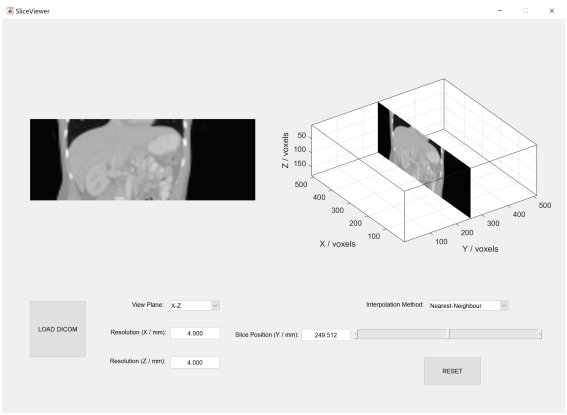


(b) Linear Interpolation.

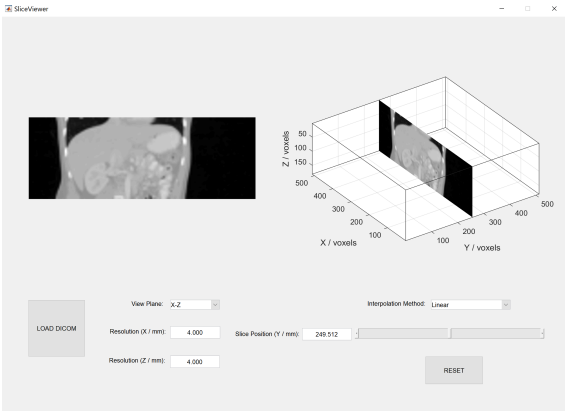


(c) Cubic Spline Interpolation.

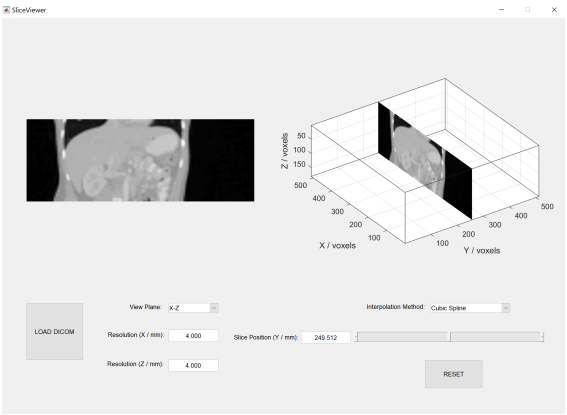
Figure 14: Screenshots of the GUI in the YZ view plane, using the in-plane resolutions of Task 2 c), for each interpolation method.



(a) Nearest-neighbour interpolation.



(b) Linear Interpolation.



(c) Cubic Spline Interpolation.

Figure 15: Screenshots of the GUI in the XZ view plane, using the in-plane resolutions of Task 2 c), for each interpolation method.

A test script, **test_ComputeOrthogonalSliceBlur.m**, was also written which computes an image with and without pre-blurring for all nine permutations of Task 2 c), and directly compares them in the same figure. Three of the nine figures produced by this test script are shown below. The images on the left of the figures are pixelated due to their being resampled to a lower resolution than the original volume view plane.

However, the images on the right have been blurred prior to resampling, and as such are are smoother with less pixellation effects.

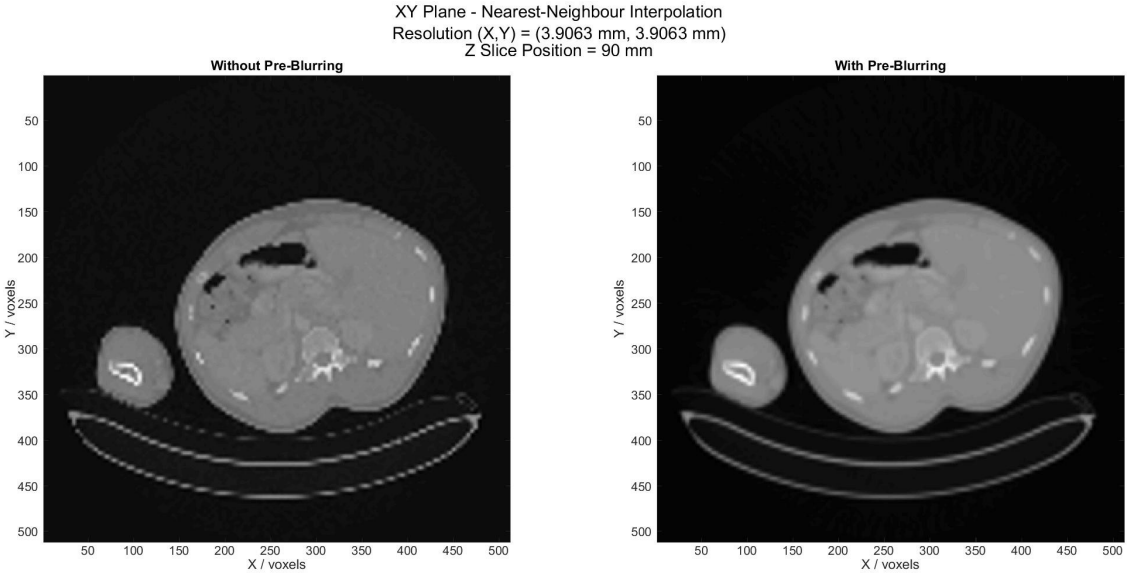


Figure 16: Images with and without pre-burring in the XY plane, using nearest-neighbour interpolation.

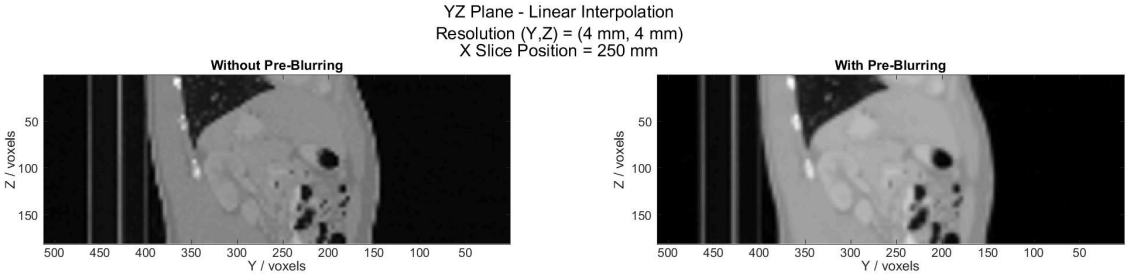


Figure 17: Images with and without pre-burring in the YZ plane, using linear interpolation.

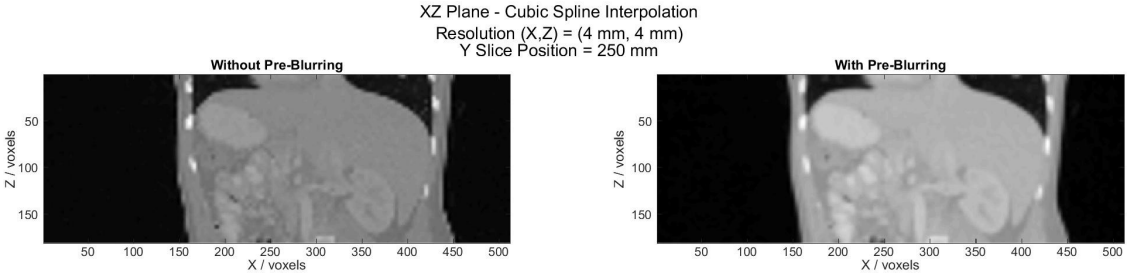


Figure 18: Images with and without pre-burring in the XY plane, using cubic spline interpolation.

Task 6

The function **ComputeObliqueSlice.m** was written such that the three rotations matrices (R_x, R_y and R_z) were multiplied together to form a rotation matrix which incorporated up to three separate 3D rotation

transformations. This was then used to compute the *rotated* voxel dimensions of the oblique slice, before being used to compute the oblique slice itself.

This latter step was done by forming a slab of two or four slices (depending upon the interpolation method being used), and transforming the centre of rotation to be at the centre of each orthogonal slice. Then within two nested for loops, each pixel location within each slice was multiplied by the rotation matrix, before being transformed back to the original coordinate system. Should the rotated coordinate still be inside the original volume, then the rotated pixel was assigned the nearest corresponding pixel intensity. However, if the coordinate had been rotated outside the original, then it was assigned a `low_value` intensity, which was the lowest value intensity present in the whole original volume (in this case -1024).

3D interpolation (with pre-blurring if necessary) then proceeded as for **ComputeOrthogonalSlice.m**, but this time sampling occurred with respect to the appropriate *rotated* voxel dimensions. In order to facilitate subsequent 2D plotting of an oblique slice, the function also returned the variable `stencil_2D`, which was effectively a record of which pixels, if any, had been rotated outside the original volume. If a pixel had been rotated to within the volume, its corresponding `stencil_2D` was set to 1; alternatively, if it had been rotated outside the volume, its corresponding `stencil_2D` value was set to NaN.

In an attempt to be consistent with Figures 3, 4 and 5 of the question sheet, an oblique slice was computed such that either of its dimensions could be less than those of its orthogonal reference slice (i.e. the slice could shrink as it rotated out of the volume). However, the slice could not *grow* to fill a larger plane as it rotated within the volume. This is demonstrated in Figure 23, in which the 3D representation of the oblique slice has eight edges: four from the original volume, and four which are an artefact of rotation (these artefacts are shown as black on the 2D image as they are converted to NaN values).

Rather than performing the rotational transformation within two for loops, a vectorised version was attempted by reorganising all of the X, Y and Z coordinates of an orthogonal slice to be in a matrix of 3 rows, and of a number of columns corresponding to the product of the two orthogonal image dimensions. This matrix would then be pre-multiplied by the rotation matrix (remembering to shift the centre of rotation before and after). However, I only manged to completely this successfully in the XY view plane; further, the assignment of pixel intensities still had to occur within two nested for loops. As such I wasn't convinced by the improvements in computational efficiency offered by this approach, and it wasn't pursued further.

A function **StripBorderObliqueSlice.m** was also written which takes an oblique slice and its corresponding `stencil_2D` as its inputs, and then strips the rows and columns of the oblique slice which have been completely rotated out of the original volume. Corners which are a product of rotation are left as NaN values, which are shown as the lowest image intensity when plotted using **imagesc**.

The script **test_ComputeObliqueSlice.m** tests the use of both **ComputeObliqueSlice.m** and **StripBorderObliqueSlice.m** to produce and plot 2D oblique slices. This script tests single and multiple rotation parameters, all three view plane orientations, varying resolutions, and all three of the interpolation methods mentioned in this report. Each oblique image is shown from an orthogonal view point, and is therefore scaled by the appropriate *rotated* voxel dimensions. Three of the six figures produced by this test script are shown on the next two pages.

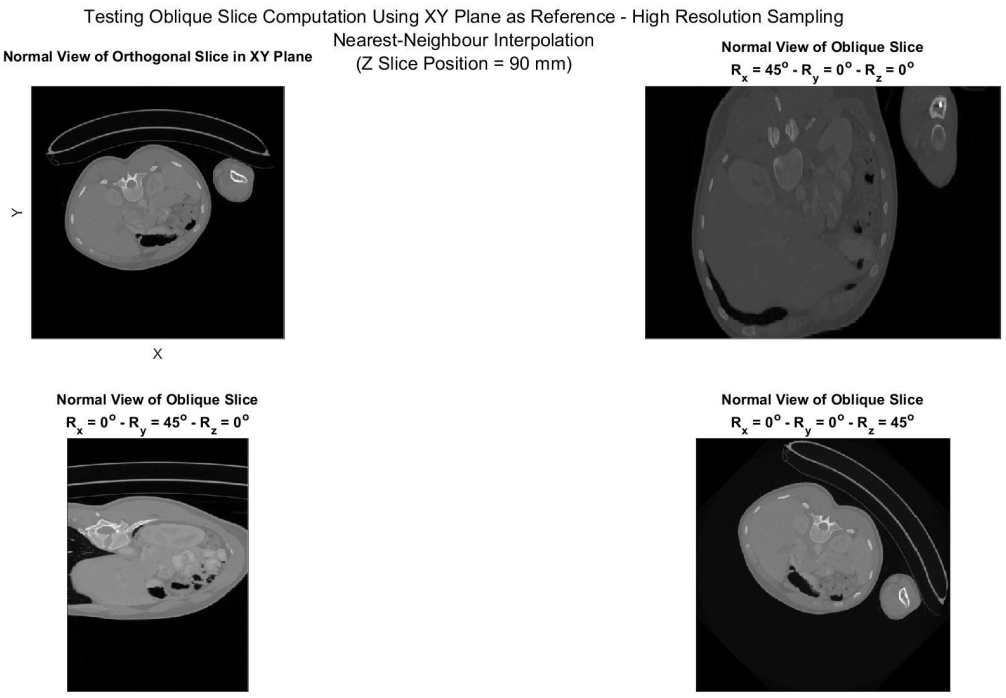


Figure 19: Example oblique slices rotated from the XY plane, rotational parameters as per individual subplot titles.

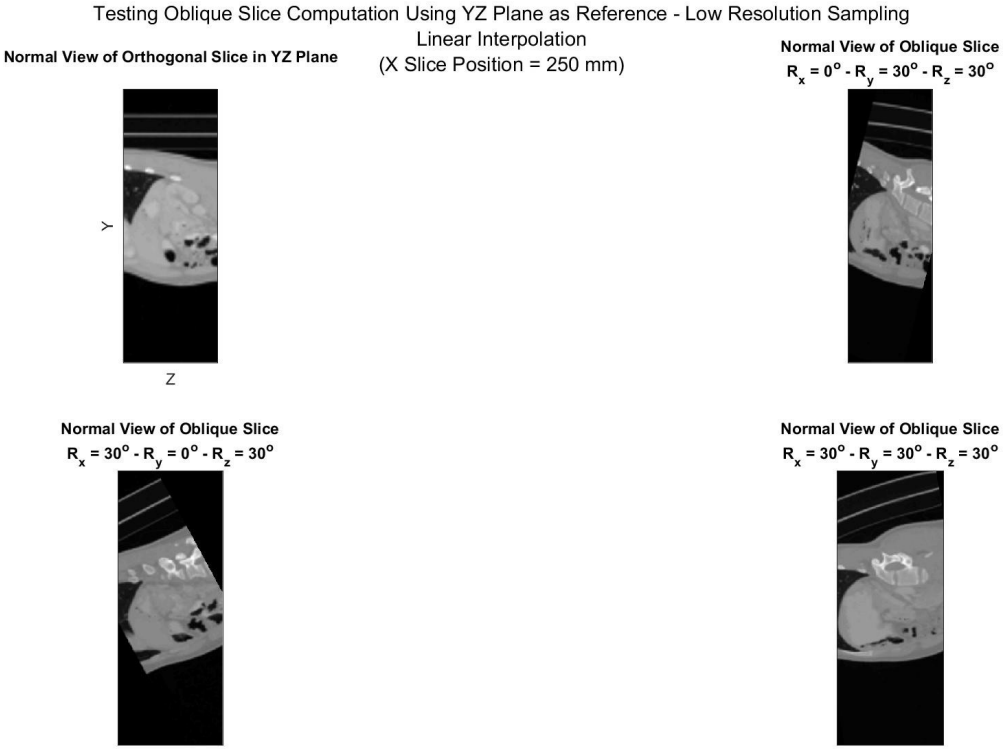


Figure 20: Example oblique slices rotated from the YZ plane, rotational parameters as per individual subplot titles.

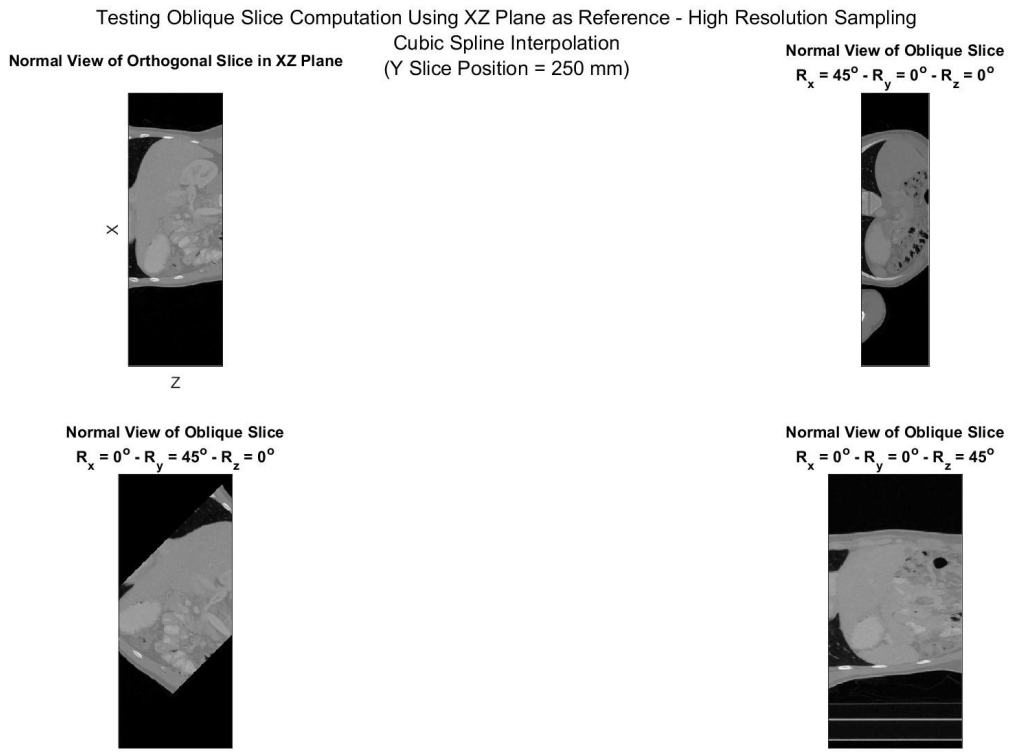


Figure 21: Example oblique slices rotated from the XZ plane, rotational parameters as per individual subplot titles.

Task 7

The *SliceViewer* GUI was updated as requested in the question sheet, using the two functions described in Task 6, with the following additions:

- The minor and major steps of each of the three rotational parameter sliders was set to 1^0 and 10^0, respectively.
- Each of the three rotational parameter edit boxes shows to a fixed precision of 1DP.
- If a value outside the range of $\pm 180^0$ is entered into any of the three rotational parameter edit boxes, a modal error dialog box is created, and the value is set to respective end of this range of allowable values.
- All three of the rotational parameter values was reset to zero each time the view plane was changed.

The 2D image on the left of the GUI display was shown using the same technique that was used in Task 6 (i.e. image scaling and sampling according to rotated voxel dimensions). The 3D image on the right of the GUI display was shown in a similar fashion to Tasks 3 and 5, except that this time the inbuilt function **rotate** was used to rotate the relevant slice into the the correct position. This rotation occurred around a specified origin (the centre of the orthogonal reference slice), and in a certain direction (as specified by the three rotational parameters).

Figure 22 shows the patient viewed with their anatomical left-right axis aligned horizontally (i.e. in a more conventional supine position). The parameters required to achieve this were an XY view plane, and a R_z value of -166.0^0. The rotation matrices given in the question sheet are in a counterclockwise direction in a right-handed Cartesian coordinate system. Therefore, the fact that this *negative* R_z parameter lead to a counterclockwise rotation must mean that this field of view in the GUI is a left-handed Cartesian coordinate system.

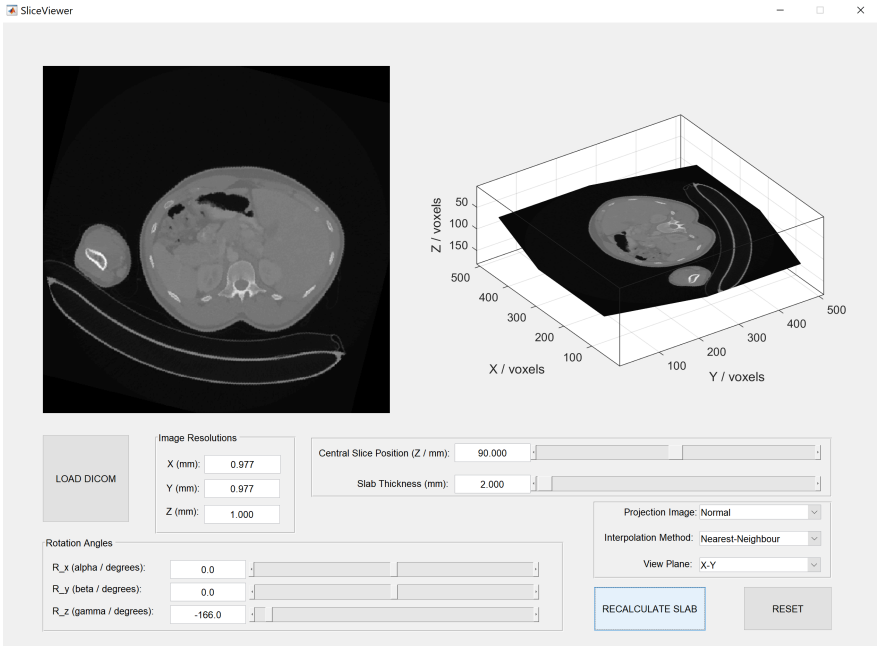


Figure 22: The rotational parameters required to view the patient in a supine orientation. NB - this screenshot was taken after completion of Task 10.

Two further screenshots demonstrating the functioning of the GUI at this stage of the assignment are shown below.

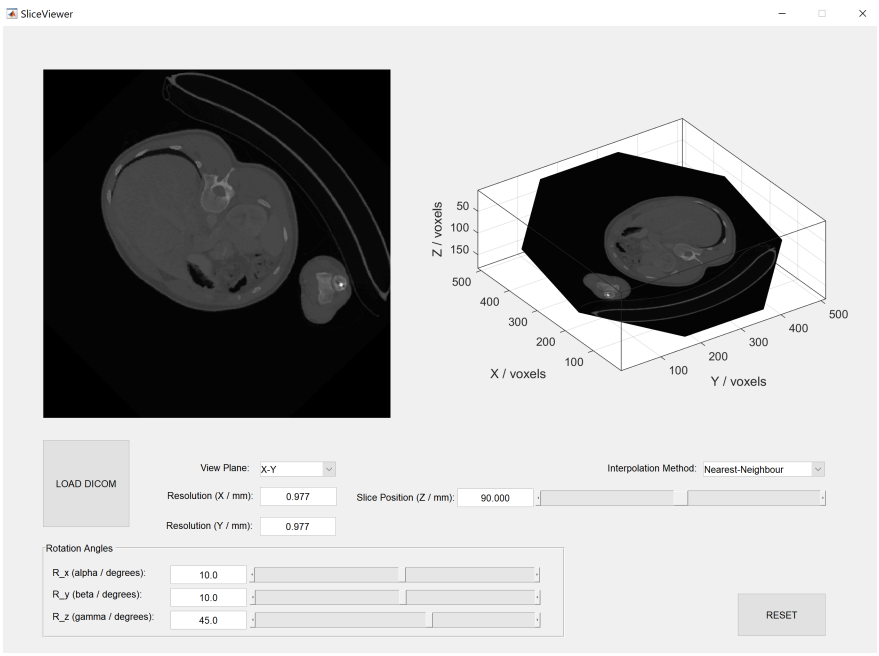


Figure 23: Demonstration of a 3D view similar to that of Figure 3 in the question sheet.

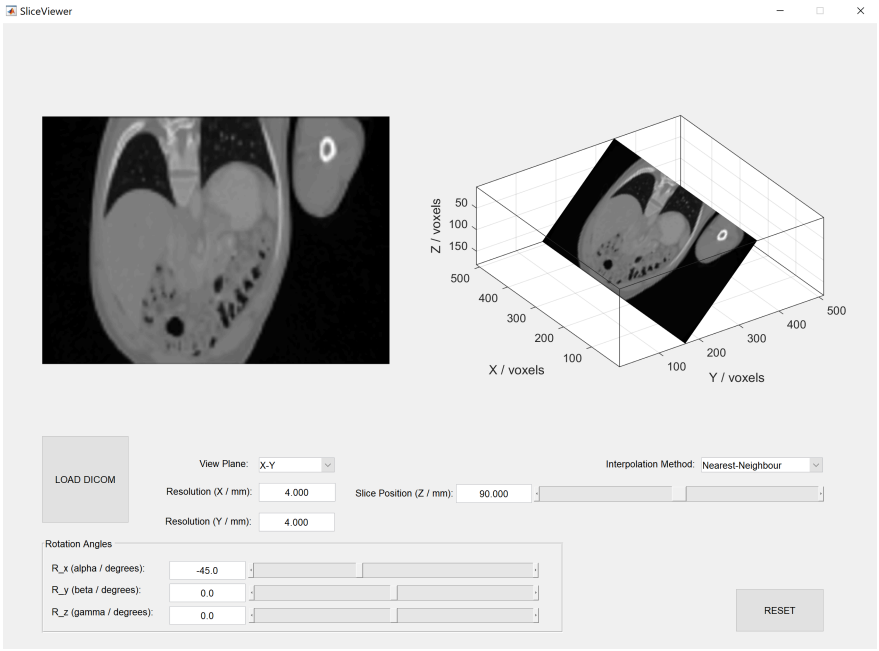


Figure 24: Demonstration of an oblique slice which has become truncated as it is rotated from the reference XY plane.

Task 8

The function **ComputeObliqueSlab.m** was written so that it would make use of other previously written functions. As such, when invoked, it makes a call to the function **ComputeObliqueSlice.m** in order to determine the central slice of the slab and its rotated voxel dimensions. This information is then used to calculate the number of oblique slices in the slab, followed by the appropriate number of calls to **ComputeObliqueSlice.m**, in order to build the slab slice by slice (either side of the already computed central slice).

ComputeObliqueSlice.m was modified from its form in Task 7 in such a way that, when being used to calculate slices for a slab, it would only calculate one slice and the return this slice. This meant that no interpolation would occur within **ComputeObliqueSlice.m**, and that interpolation would only occur within **ComputeObliqueSlab.m** once the whole slab had been constructed. It was hoped that this alteration to **ComputeObliqueSlice.m** would significantly speed up the process of computing a slab. Each oblique slice of a slab was also computed in such a way that it was rotated around the orthogonal centre of the *slab* from which it was derived, rather than its *own* orthogonal centre.

Error checking also occurred at the beginning of the code for each view plane orientation within **ComputeObliqueSlab.m**, in order to ensure that the orthogonal resolution was less than or equal to the requested slab thickness. Once this error check had been passed and the slab had been constructed, it was blurred as necessary (in a similar manner to Task 6) and 3D interpolation was then performed. The slab was then re-sized and returned as an output of the function.

The function also returns the variable `stencil_3D` (the 3D analogue of `stencil_2D`), the centre index of the slab, the number of slices in the slab, and the rotated voxel dimensions. All of these variables are used to assist in the 3D plotting of the slab, and in the 2D plotting of any projection images extracted from the slab.

A further function, **DisplaySlab3D.m**, was written to facilitate the 3D plotting of oblique slabs. This function was able to plot an oblique slab either in a new figure window, or straight into an axes of the *SliceViewer* GUI. This was performed in a similar manner to Task 6, except that this time a *slab* was inserted into a *range* of indices within a blank matrix before being rotated around the orthogonal centre of the slab. The script **test_ComputeObliqueSlab_and_DisplaySlab3D.m** was written in order to test the

two new functions built for this Task.

This script calculates slabs from all three view plane angles, and tests all three interpolation methods used in this report, along with a range of sampling resolutions, rotation angles, central slice positions and slab thicknesses. The three figures produced by this test script are shown below.

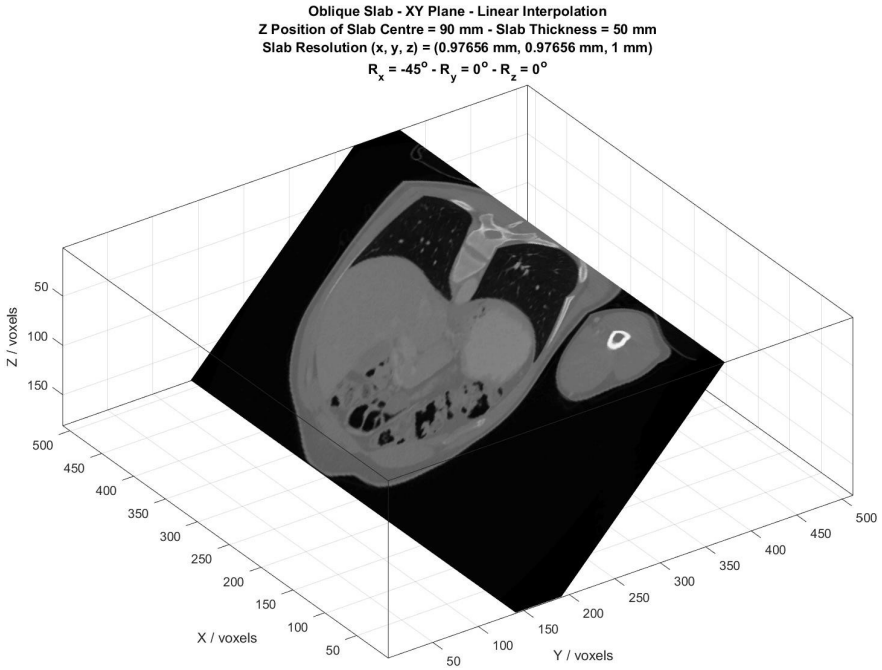


Figure 25: An oblique slab produced from the XY plane.

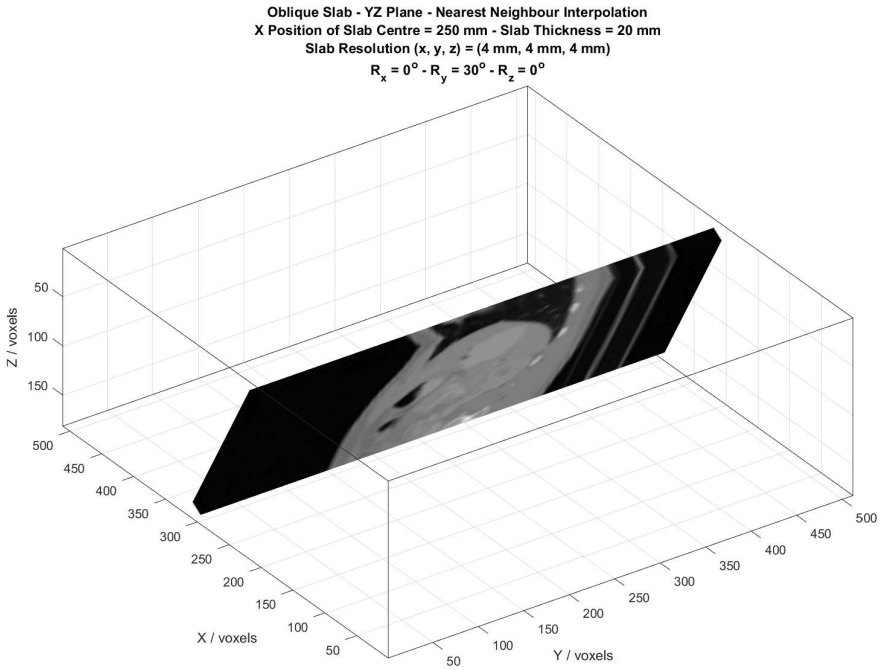


Figure 26: An oblique slab produced from the YZ plane.

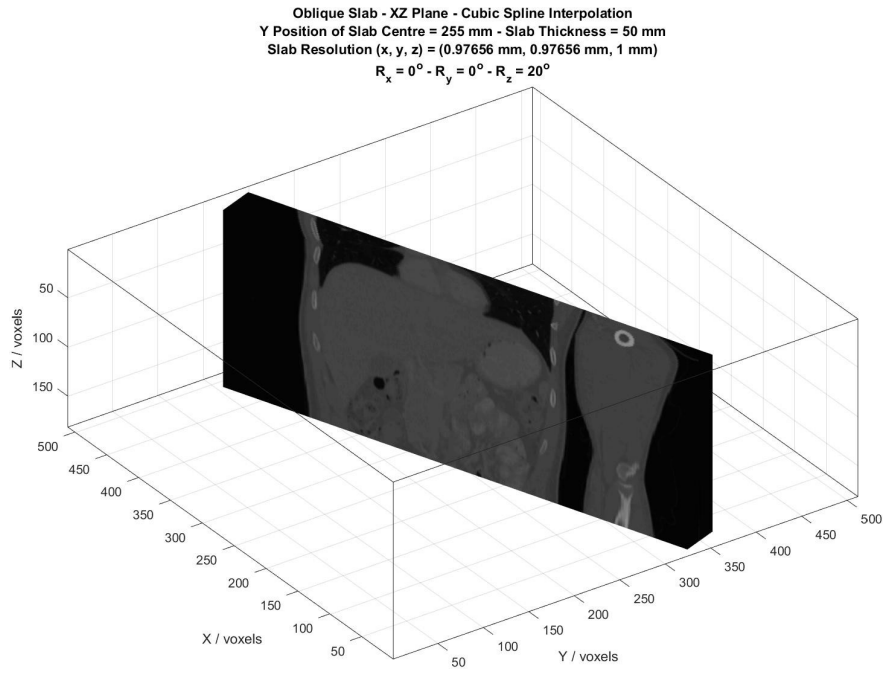


Figure 27: An oblique slab produced from the XZ plane.

Task 9

The function **ComputeProjectionImage.m** was written to receive the following inputs: an oblique slab and its associated `stencil_3D` variable, the number of slices in the slab and the index of its centre slice, the view plane orientation of the slice, and also the projection image required. The function either returned the central slice of the slab, or a maximum/minimum/median projection image calculated using the central slice as the projection plane.

The function first converts all intensities within the slab which have been rotated outside the original volume to NaN values using the `stencil_3D` variable. The `stencil_3D` variable can then be used as necessary to compute a `stencil_2D` variable, which is used subsequently to calculate either a central slice or a projection image.

The projection images themselves were fairly straightforward to compute using the inbuilt MATLAB functions **max**, **min** and **median**. It was important to specify the dimension of the slab along which to perform these projection operations (this was complicated slightly by the fact that if the slab was only composed of one slice then this was always the third dimension, regardless of view plane orientation).

The projection images then had their NaN borders removed using the function **StripBorderObliqueSlice** and `stencil2_D`. It is worth noting that `stencil2_D` is itself a maximum projection image of `stencil3_D`; this meant that as much information as possible was retained from the oblique slab when computing a projection image (as different slices within the slab could have been rotated out of the original volume to varying degrees).

The inbuilt function **squeeze** was used frequently in **ComputeProjectionImage.m** in order to ensure that 2D variables were indeed 2D, and that they did not have any singleton dimensions. The plotting function **DisplayImage2D.m** was also written, which was capable of displaying a 2D projection image either within a figure or within an axes of the *SliceViewer* GUI. The two functions built for this Task were then tested in the script **test_ComputeProjectionImage_and_DisplayImage2D.m**.

This script calculates central, maximum, minimum and median projection images from slabs extracted from all three view plane angles, and tests all three interpolation methods used in this report, along with a range of sampling resolutions, rotation angles, central slice positions and slab thicknesses. The three figures produced by this test script are shown below.

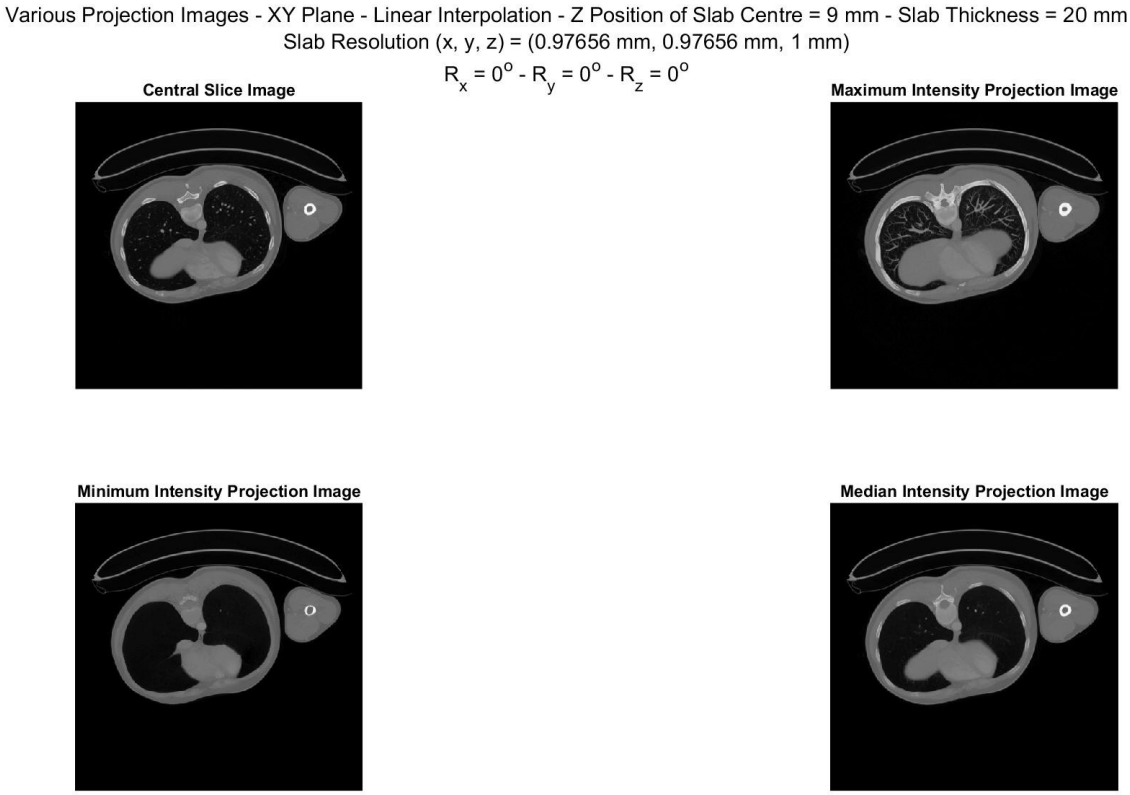


Figure 28: A central slice image and three projection images from the XY plane. The top row corresponds to the two images in Figure 5 of the question sheet.

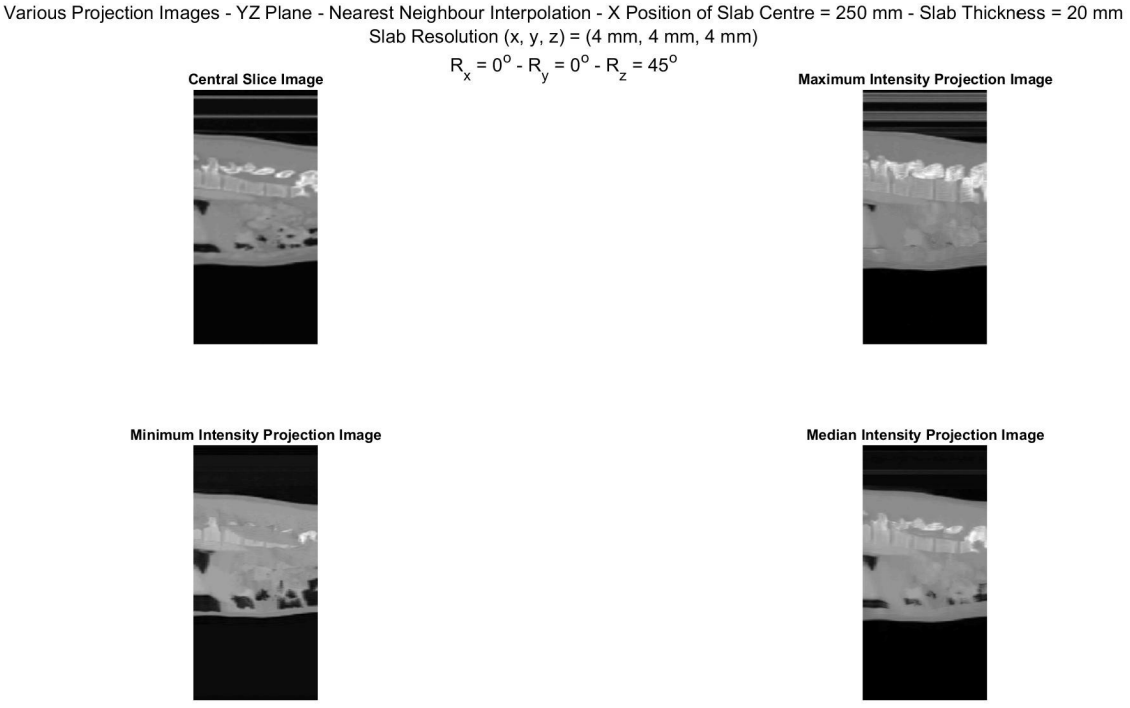


Figure 29: A central slice image and three projection images from a slab rotated from the YZ plane.

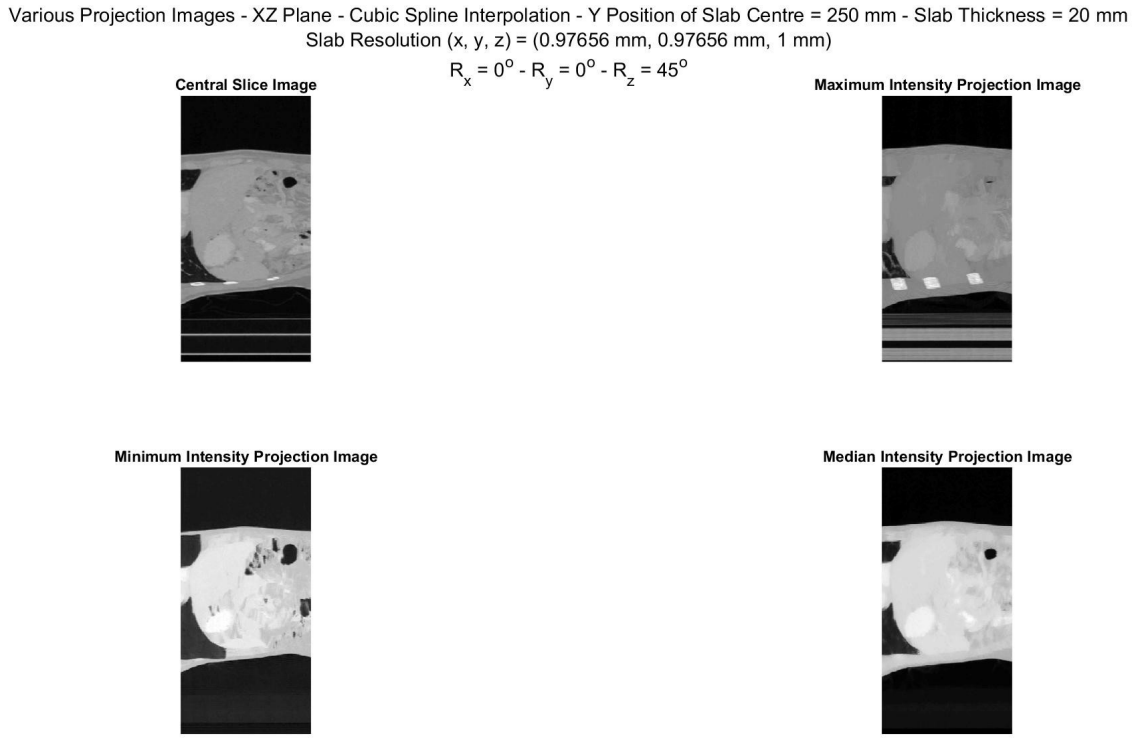


Figure 30: A central slice image and three projection images from a slab rotated from the XZ plane.

Task 10

The *SliceViewer* GUI was updated as requested in the question sheet, using the four functions written for Tasks 8 and 9, with the following additions:

- The minor and major steps of the slice thickness slider were set to 1 and 10 slices of the current view plane, respectively. The minimum and maximum allowable values were set to the non-rotated orthogonal voxel dimension and the orthogonal image length, respectively.
- The slab thickness edit box displays to a precision of 3DP. Should the user enter a value outside the allowable range, a modal error dialog box is created, and the value is set to the respective end of the range of allowable values.
- A third image resolution edit was also added, which behaved in a similar fashion to the first two resolution boxes described in Task 3. Whenever the view plane is changed, this value automatically defaults to the orthogonal voxel dimension, and the static text box associated with it is also updated.
- Further functionality was added to the reset button such that, whenever it was clicked, a normal (or central slice) projection image is shown in the left hand display of the GUI, and the slab thickness reverts to its minimum value.
- A "recalculate slab" push button was added to the GUI . This allowed the user to refresh and display the slab currently being held in the memory of the GUI *after* making changes to its parameters. This prevented the GUI from recalculating a slab each time a single parameter was changed. However, the projection image is automatically updated *whenever* its drop-down menu is altered (as this is a less costly thing to compute).
- Error checking also occurs before a slab is computed, in order to ensure that that the slab that the user requests is entirely contained within the volume. This avoids lengthy calculations of a thick slab, which fail at the end if the volume boundaries are exceeded.

The figures below demonstrate the functioning of the GUI on completion of this assignment.

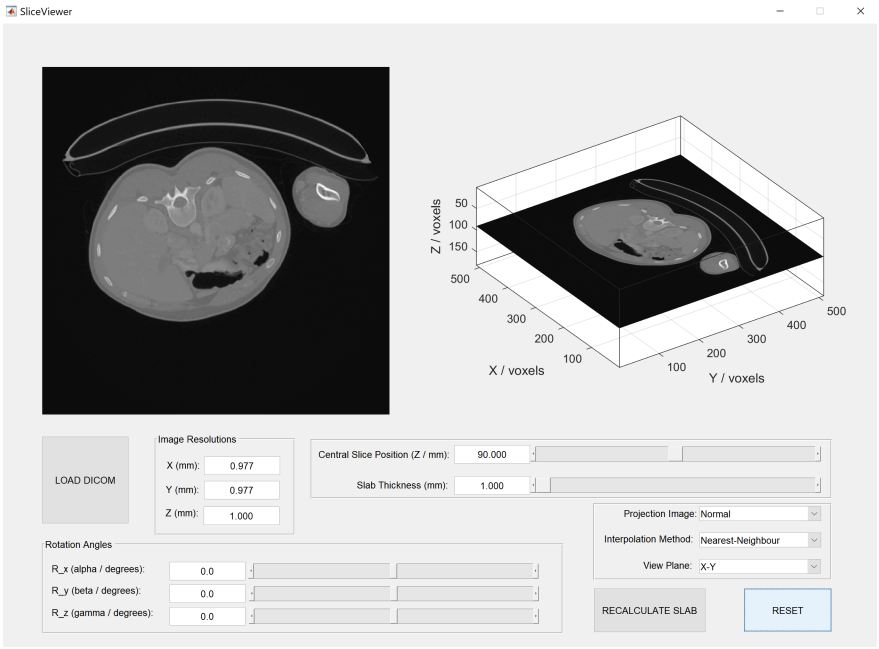


Figure 31: The default view of the GUI having loaded in the DICOM volume (this view can also be achieved at any point by clicking the reset button).

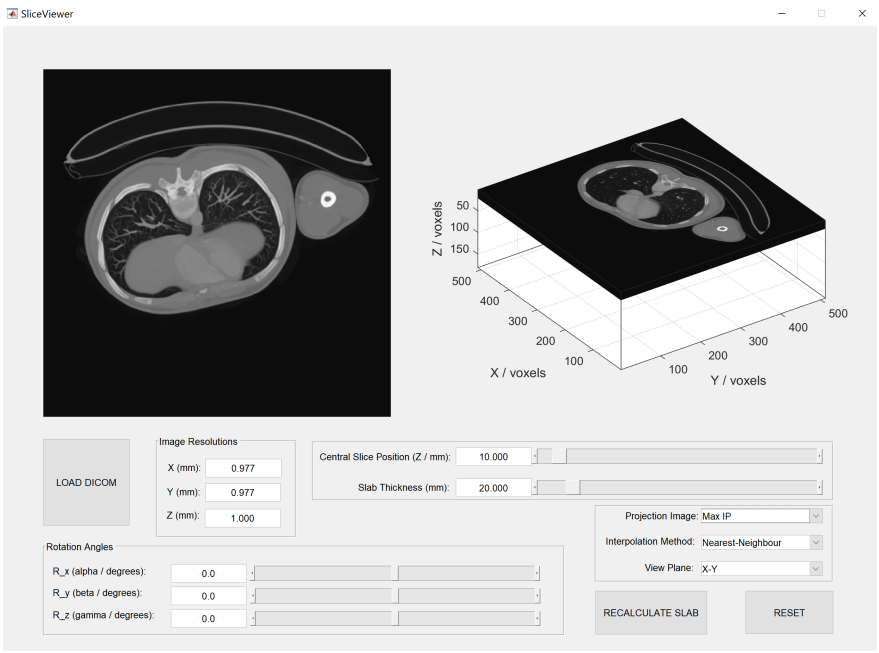


Figure 32: The maximum intensity image shown in Figure 5 of the question sheet, and its corresponding slab.

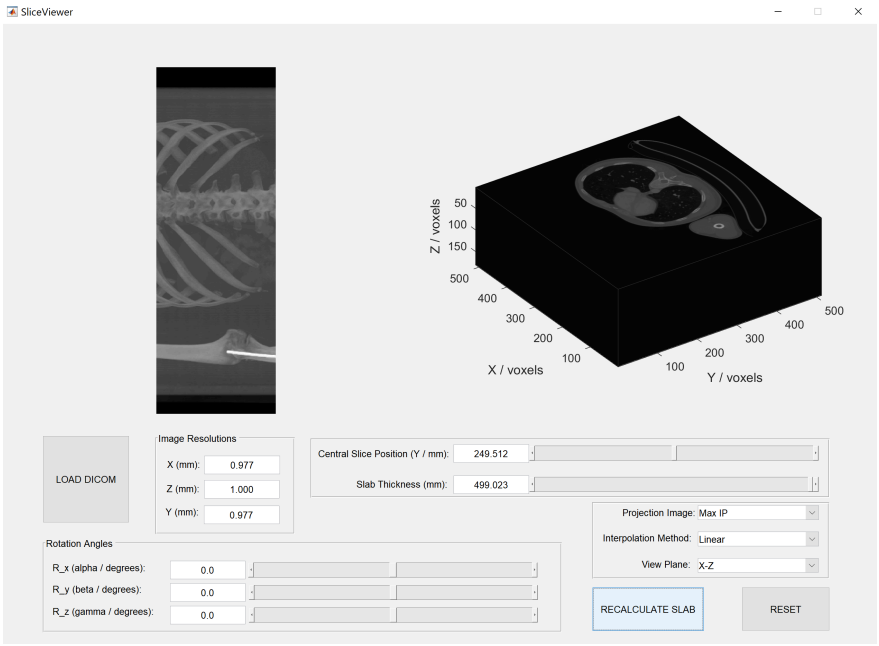


Figure 33: Demonstrating the ability of the GUI to handle a full thickness slab in the XZ view plane, and its corresponding maximum intensity projection image. This projection image clearly outlines the patient's skeleton and an orthopaedic implant around the left elbow.

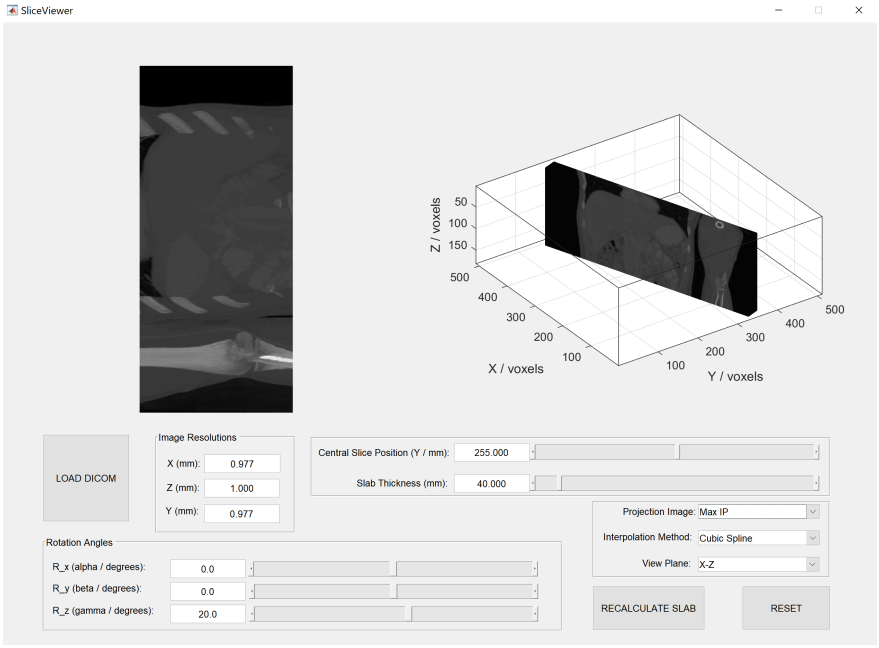


Figure 34: A slab rotated from the XZ view plane and its corresponding maximum intensity projection image.