DELINEATING REGIONS OF INTEREST IN MRI/S PROSTATE SCANS FOR CANCER DIAGNOSIS

Name: Daniel Morgan

Student Number: C1527793

Supervisor: Dr Frank Langbein

Moderator: Dr Jing Wu

Abstract

This report documents the implementation of a tool aimed at tackling the problem of delineating regions of interest in MRI prostate scans for cancer diagnosis in order to aid Radiologists in performing their job. It describes a tool that displays MRI scans and allows for annotation. This report also documents the future of such tool and the ways in which the application could be further expanded to incorporate tools that more effectively tackle the problem area.

Acknowledgements

I would like to thank Dr Frank Langbein for his help, guidance and patience throughout this project.

Table of Contents

1. Introduction

This report details the design, development and evaluation of an application that enables users (specifically radiologists) to View prostate scans that are saved as DICOM files in order to delineate any potentially dangerous regions within the scan. The application will enable the user to then annotate any areas in a scan and save this annotation to an output CSV file. This report will also detail future uses for the application including scope for integration with systems researching machine learning techniques aimed at identifying and delineating regions of interest in prostate scans.

1.1– Project Aims

This project aims to deliver an application that makes the process of cancer diagnosis easier through providing tools allowing a user to annotate and identify areas of interest within a prostate scan. By doing this, the project will attempt to deliver an efficient and robust system within which a user can operate without issue. The application should allow users to view the scan images contained within a DICOM file, no matter what resolution or aspect ratio this displays at. Upon display, a user should be able to easily identify any potential areas of interest within the scan image, annotate this area and save to a location for future reference. The project will also serve as a platform from which other systems will be able to integrate with in order to further aid the user.

1.2– Personal Aims

At the beginning of this project I had relatively little experience with python. I also had no experience in medical imaging or image manipulation. This project will give me the chance to develop my own personal skills in a new language as well as increase my knowledge in the subject areas mentioned above. Project management skills will be crucial in the development of this application. Due to the research that I will have to do in order to understand the problem in the early stages of development, its essential that the implementation makes good progress towards the later stages of the project. The development of this application will also test my abilities to develop an effective user interface that adequately tackles the problem.

1.3– Target Audience

When considering the target audience of the application, we must think both long term and short term. The future applications of this project will most likely be used in order to make the process of delineating areas of interest in prostate scans far easier for Radiologists. Therefore, in terms of functionality and interfacing, this application will be designed with the human user, a Radiologist, in mind.

However, currently the scope of this application does not cover clinical use. The short term uses of this application should also be considered. This application will be extended in the

future to include machine learning systems that will annotate potentially dangerous areas of the scan for human review. Therefore, a secondary target audience to this project will be researchers who are working towards solving that particular problem, as they will be able to integrate their systems with this application. This means that the codebase will have to be well documented, with a detailed explanation of how features were implemented. The application will also have to be in a state where efficiency is the highest priority, as others extending the project should not have to spend time attempting to solve speed and bug issues within the application.

1.4– Approach

In the early stages of the project, I attempted to set out a structured way to approaching the problem. At first, I had no knowledge of any of the technologies listed in section 3.1 such as Python, TkInter and image processing with matplotlib. I also knew nothing about medical technologies like the DICOM file format. Therefore, I spent a large part of the early stage researching the problem area (MRI, DICOM, etc) and attempting to further my knowledge in Python. When I felt comfortable that my skills were a sufficient level of competency with both the technologies used and the problem area, I began implementation of the application. The final weeks of the project were then spent fixing bugs and documenting the codebase as well as I could in order to write a concise and correct report.

1.5– Assumptions

There are a few assumptions that this project takes into consideration. The main assumptions are related to the nature of the DICOM file structure; however, this structure tends to be a standard in all DICOM files. The main assumptions related to the DICOM structure are –

1) The information acquired through interaction with the tags listed in the DICOM module description is correct.
2) The information acquired through interaction with said tags is in the right location.
3) The information acquired is of the correct datatype required for use in elements of this application (documented in section 2.1.2 of this report).

There are also some assumptions this application makes with regards to the user. These assumptions may make the application less useable to a certain

subset of users, however overall this application has taken what's assumed to be the vast majority of the target audience into consideration. These assumptions are –

1) The user has access to the files on their local device.
2) The user has some level of IT literacy.
3) The project assumes that the user has no deficiencies with their eyesight such as colour blindness, which would impair their ability to view annotations on screen.

1.6– Outcomes of the Project

The broad outcomes of this project will be to develop an application featuring tools for radiologists to delineate and identify regions of interest in an MRI prostate scan. Developing

an effective and efficient user interface is also of paramount priority. Another outcome would be to develop a system can be further extended to include machine learning techniques for identifying regions of interest, again in an effort to aid the Radiologist in performing their job.

There are also personal outcomes of this project, such as developing personal skills in different areas like time management and project management. Developing my own knowledge in a programming language within which I have very little experience will also be an important outcome.

2. Background

Prostate cancer is the second most common cancer in men worldwide. Recent statistics published by the Prostate Cancer Research Organisation[1] show that 1 in 8 men living in the UK will be diagnosed with Prostate cancer at some point in their life. As is true with all areas of healthcare, improving diagnosis and treatment for prostate cancer could save many lives and currently the mortality rate for males that develop prostate cancer is decreasing.

2.1 – Identified Problems

As diagnosis is of paramount importance, we should ensure that Radiologists are equipped with the most useful tools available to help them efficiently carry out their job. To understand the tools needed for said job, we must understand the technologies that Radiologists encounter in their day to day operations.

There is also additional scope to this project. Future integration with artificial intelligence will have to be considered when designing a solution to this problem. Although the nature of the artificial intelligence is not currently known, the user interface of the application will have to have certain features that are automatable. Upon automation of these tasks, such as highlighting an area in a scan that is potentially dangerous, the functionality of the application must be extendable to allow for tools enabling review of said automation.

2.1.1 – MRI (Magnetic Resonance Imaging)

The main method of diagnosing prostate cancer is the use of MRI[2]. MRI uses an external magnetic field to align protons that exists in the water nuclei of the tissue area in which we are scanning. The results of this are captured in a matrix of pixels, describing the location of the imaged plane in different shades of grayscale. There are different ways of sequencing these images, such as the T1 weighted scan and the T2 weighted scan**.** These sequences can also be separated by the orientation of the body. There are three kinds of orientations that the scan can take, these are –

1) Coronal: Front to the back
2) Axial: Head to feet view
3) Sagittal: Left and right from a front on perspective

Once a scan sequence has taken place, these images are often saved in a File format called DICOM.

From a user interface perspective, the direction in which these scan sequences are taken can cause an issue. When considering additional functionality of this application, like synchronisation between two stacks, there will be cases where each stack will have been imaged from a different direction. In such a scenario, the stacks will not be comparable in the patient space without a lot of extra work. This functionality must always be present, however this must be considered and explained to the user if they do attempt to synchronise two stacks taken from a different direction.

2.1.2 - DICOM

DICOM (Digital Imaging and Communications in Medicine)[3] is a standard medical imaging format that allows medical professionals to meta data related to a specific scan slice. This can store string values such as a patient ID, the type of scan and the date on which it was taken. These values are stored in attribute tags retrievable through queries by programming languages such as Python.

In the scope of this project, there are a few tags from within the DICOM file that will be essential to functionality of the application. The most important tag we'll need to extract is called the Pixel Array. This tag returns a NumPy array representing the raw bytes of the scan image. We can use this array to display the scan image on screen. From a UI perspective this is an interesting problem as this requires some additional functionality to plot the array as an image, for this we will use matplotlib's image plot functionality, which takes an array and plots each point in the array as a pixel on screen. This will generate the image that we will use in the centre of the screen. Tags such as the patient ID and Scan Type will be used to display information about each slice above the generated image.

Time permitted, there will also be tags that will be useful for additional functionality. Within the DICOM there is an attribute named the 'Image Plane Module'. This module contains information related to the PBCS (patient-based coordinate system). The patient-based coordinate system refers to the 3-dimensional coordinates of the scan in the patient's body. The information contained within this module can be used, for example, to compare two slices with regards to their position in the PBCS in an effort to see if the two slices are in fact comparable. The DICOM tag that we need to extract to be able to do this calculation is the 'ImagePositionPatient' tag. This tag returns the coordinates of the top left-hand pixel of the image in the PBCS. As mentioned in section 2.1.1 one however, if the two stacks in question in this particular scenario are taken from a different direction, the two stacks will not be comparable, and we shouldn't attempt to synchronise.

There is also the case where a user will try to synchronise two stacks that come from different patients, which may work with regards to two patients having similar properties such as coordinate systems, however this shouldn't be allowed. To ensure that this isn't happening, we can do two things. We can compare the patientID's to check if the user is attempting to compare two different patients scans and we can retrieve a tag from the DICOM called the 'Frame of Reference UID', which uniquely identifies the frame of

reference for a specific scan series. As discussed, a patient will usually have multiple subdirectories in their patient directory, all DICOM files in said subdirectories will have the same Frame of Reference UID.

2.1.3 – The Project Problem

The main problem that this project is trying to tackle is the difficulty in displaying this information to a radiologist with ease. The main issue radiologists face is that to view these scans, they have to use a specially calibrated screen that will allow them to display the scan in the highest possible resolution. The broader aim of this project, along with making that job easier for radiologists, is to aid them in their efforts to identify and delineate potentially malignant cells in a patient's prostate. This will in turn make the process of diagnosis for a patient far easier. The main way of doing this is to generate a program that will work on any screen and allow them to use tools to easily highlight and delineate any areas in a scan they believe to be potentially dangerous.

This project will also aim to develop an application which can be used in future to integrate machine learning systems that aim to further solve the problem area. These systems will likely attempt to identify regions of interest in the scans for human review. Another problem we will encounter is to produce an application that is sufficiently extendable to allow for this integration.

3. Specification

The application itself will be written in Python 2.7, however, to create it simply using native python would be an extremely time consuming and demanding task, so this project will make use of a few powerful external packages alongside Python that will make development of said application far easier and the overall process more time efficient. The three most important packages in question are PyDicom, Matplotlib and TkInter.

3.1 – Technologies Used

Throughout this project, a range of technologies have been used to tackle the problem discussed in section 2.1.2. These technologies are all externally developed with licensing allowing others to incorporate them in new projects.

3.1.1 - PyDicom

The integration with PyDicom[4] is essential to the completion of the project. The tool allows python to read in any DICOM file and successfully extract any of the metadata contained within the file format. For example, PyDicom will allow you to convert a DICOM file into an object (note, this 'object' is not the same as the DicomObject discussed in section 4 of this report). From this object we can extract the pixel array that details the scan image as discussed in section 2.1.

3.1.2 - Matplotlib

When the pixel array has been extracted, there needs to be some method of displaying it. Another external package I will be using is Matplotlib[5]. Matplotlib is, again, a python package that comes with several tools based around image processing and plotting. One specific feature of this package is the ability to create images from arrays. We can couple the pixel array extracted using PyDicom with the plotting capabilities of matplotlib to create an image of the patient scan. Matplotlib also provides tools for highlighting areas on an image plot, such as drawing points, polygons and circles.

3.1.3 - TkInter

One final tool that will be used to create the user interface (front end) of the application is TkInter[6]. TkInter is a popular UI toolkit that comes with several predesigned 'widgets'. These widgets are common elements in user interfaces, such as buttons, radio buttons, text entries and frames. TkInter also comes with a grid system that will be used to design the layout for the application. TkInter has integration with matplotlib which will allows the use of a matplotlib plot in a TkInter image widget.

3.1.4 - Justification

These three tools will allow for the creation of a robust and efficient application and will make the way in which the project approaches the problem of creating an effective user interface for radiologists to interact with, which in turn makes their job of delineating and diagnosing prostate cancer easier. Without these three packages (or different variations of them), this project would be both extremely difficult and time consuming.

With regards to future work, such as machine learning integration, these packages pose no issues. The user interface technologies such as Matplotlib and TkInter will be easily integrated with such systems and have functionality making the identification of the areas in question automatable from the backend of the application. The main bulk of the application being written in Python also allows for machine learning integration.

3.1.5 – Other Technologies

This project also makes use of technologies outside of the implementation scope. For version control, the project will incorporate git[7]. This will allow me to prototype, bug fix and implement new features without damaging an existing copy of the codebase. Trello will be used as a ticket tracking system, from which I will base branches of the git repository around. I will also be adopting an agile approach to development throughout the project. I will first prototype a feature, implement and test it. I will then demonstrate the new feature to my supervisor who will in turn give me feedback upon which I can refine and refactor elements of the project.

3.2 – Stakeholders in the Application

As discussed in section 1.3, the main target audience of this application will be radiologists. The aim of this project is to make the delineation of potentially dangerous and malignant cells in prostate scans far more efficient and accessible for all that wish to do so. This application has been developed for the desktop platform. The reasons for this are that mobile would make the functionality of the application far harder to implement in a UX design that makes sense, the same could be said for tablet also. A web application would mean that internet access is always needed and while this can be assumed, it doesn't necessarily mean that this is the best platform to host the application.

There are three crucial aims that the application needs to have in order to be an industry usable product and one that suits the needs of a radiologist, these are –

1. To make it easy to display and view MRI scans.
2. To make it simple and efficient to navigate through a stack of DICOM's.
3. To make the annotation and identification of potential regions of interest far easier for radiologists.

If these three pieces of functionality are met by the application then it should be of benefit to radiologists, if any of these three are missing, then it will fall short of that which is currently available for radiologists to use.

3.3 – Additional Features to Consider

Extending upon the features discussed in section 3.2, there are several other features that could also benefit the users of the application. The annotation of regions of interest is a crucial aspect of this project and extensive tools should be made available in order to make this more useful to a user. Examples of this could be the ability to draw and manipulate polygons and circles on an image.

The ability to view two slices, or two separate stacks of scans at the same time is functionality that will also be extremely beneficial to a radiologist and one that addresses the problem in question. An extension of this is to include functionality that will allow the user to 'Synchronise' two separate stacks (if this is possible in each given case). This functionality is the most difficult to implement but will make one of the crucial aims of the application, the ease of navigation through stacks, much more efficient.

3.4 – Aims and Research questions

It is clear that the overall aims of this project will be to develop a robust and efficient solution to the problem of delineating regions of interest in MRI scans for the benefit of radiologists by giving them the functionality to allow them to easily view, navigate through and annotate prostate stacks and scans. This is aim is in my opinion, the most important, as the base functionality of the application tackles this problem. However, there is also the additional scope of integration with other systems in the future.

In order to assess and demonstrate the stated aim, this project will evaluate the effectiveness of the user interface of the created application to determine whether or not it can easily be used by radiologists in an effective and efficient manner. The interface and robustness of this application should be prioritised over a more aesthetic user interface as the key features the project, discussed in section 3.2, are what will determine the usefulness of the application. In later sections this report will also discuss the ease in which the application can be extended to incorporate other systems such as machine learning algorithms aimed at solving a similar problem.

4. System Architecture and Design

This section will highlight the specification for the application, what it must and must not do. It will look at high level aspects such as UX design, work approaches and external sources needed for implementation. It will also justify approaches to solving the problem addressed in section 2.

4.1 – Basic Functionality of the Application

As discussed in sections 3.3, 3.4, for the problem to be solved the application must have some base line functionality. This functionality will be the foundation upon which additional more complex features can be implemented.

4.1.1 – Viewing the MRI Scans

The most basic level of functionality that this application must have is the ability to view MRI scans generated from metadata within the DICOM files. The application will be designed partially around the structure in which the TCIA[8] (cancer imaging archive) stores patient prostate scans. This public data will be used throughout development and testing of the application. As documented in section 2.1.1, there are different ways of performing said MRI scans from which a radiologist will attempt to identify potentially cancerous cells within a patient's prostate. The TCIA structures the information in the same way for each patient. There is a high-level directory which is usually named according to the patient ID. This directory will contain 4 different sub directories, documenting the different variations of scans performed on a patient. These subdirectories are then essentially the 'stacks' of DICOMS that the user will be able to view and navigate through. The below figure demonstrates the standard structure of a patient directory.

FIGURE 1 - EXAMPLE OF PATIENT FILE STRUCTURE

The first thing the application will need to do is extract these files from each directory. Upon extraction, it will then use PyDicom to process each individual file and generate an image

from the pixel array contained within it. Matplotlib will then be used to display the image as a plot on a figure.

4.1.2 – Navigation Through the DICOM Stacks

Another piece of basic functionality is to give the radiologist the ability to navigate through the stacks documented in 3.1.1. To do this, each folder loaded into the application must have its contents stored in an array for the application to allow easy navigation. Moving through the array is not the difficult aspect of this functionality, more updating the plot (displayed image) created based on the first instance in the stack. For memory and efficiency purposes, the project must ensure that it does not create a new figure for each iteration of the array. Instead it must create a foundation level figure upon which it will draw each DICOM image when called.

The navigation tools will be a crucial part of the application as there are plans to include functionality allowing a user to navigate through two stacks on the same screen, displaying an instance of each stack side by side. This means that there has to be two separate instances of essentially the same navigation, so again for efficiency purposes, the project should have tools are designed to be instantiated more than once. Simple 'previous' and 'next' buttons should ensure that the interface of the application stays intuitive while also being robust and useful to the user.

3.1.3 – Annotation of MRI Scans

One final piece of crucial functionality included is the ability to annotate the images generated from each DICOM. Matplotlib provides extensive tools to make this functionality reasonably easy to implement. The annotations are clearly displayed on top of each individual image and will of course change when navigating through the stack. In other words, annotations in one file should not appear in another (unless the user specifically requests it).

There should however be functionality to redraw 'saved' annotations if the user decides to navigate back to any given slice upon which annotations have previously been drawn. This will mean that the application will have to have some functionality that maps annotations to the DICOM file on which they were drawn and attempt to redraw them on load.

Thought has also been given to the method in which the annotations will be saved. The best method to do this would be to save a CSV file to each directory within the overarching patient directory that documents the annotations drawn in that particular stack. The CSV file should carry information about the annotation itself such as the type of annotation (polygon, point, etc), the coordinates of the annotation and the PI-RADS score[8] given to the area annotated by the user. This again can be done using functionality contained within the TkInter package such as a simple message box. The file should also contain information about the specific slice in question, such as its position in the stack, the scan type and the patient ID of the patient that the scan has been performed on.

4.2 – Additional Functionality of the Application

The functionality of the application documented in section 3.1 is the very basic functionality of the application. There are additional features that could be included in the project to make it more effective and usable for radiologists.

One piece of functionality the project could additionally have is the ability to view two stacks in the same window. Them being in the same window is important as it adds to the usability of the application as it would not require the user to take any additional measures to view two images side by side. This functionality could be extended by including the ability to synchronise two stacks to see which positions in stack A match that of stack B and visa versa.

Once the application has the ability to synchronise two stacks, it will be possible to display annotations from a specific slice in Stack A, in the synchronised B stack. This functionality is in my opinion, stand out functionality and will be extremely beneficial to users, however, this is possibly, along with the ability to synchronise the stacks, the most complex functionality to implement as it goes beyond simple UX design and data structure manipulation and is more to do with mathematical comparisons between two plots.

4.3 – System Architecture

4.3.1 – Code structure and approach

The application itself has been implemented according to object-oriented principles coupled with the top down nature of python scripts. Due to the additional functionality (such as dual screens), best practice is to ensure that each major part of the interface is its own object. This allows for easy implementation of the dual screen functionality as the way in which each object is instantiated has some control on its position within the application window. This approach took several refactors to get correct, as when I began the project, I was in-experienced with TkInter and Python.

One example to consider is that of the object that handles navigation through the stack of slices. If the application is going to have functionality enabling two slices to be displayed side by side, each displayed in a figure, each figure will require its own navigation functionality. To do this, we can create a navigation object that is aware of which figure it is attached to. Due to the nature of TkInter's grid, during the instantiation of the navigation object, the object could be structured to take in variables that determine its location on the screen. This improves code efficiency and decreases the time spent on implementation of simple features. It also has several functions that can be reused by both navigations in the dual screen.

4.3.2 – Interface Components

The front end of the application requires a few different components in order to make the navigation dynamic and responsive. The code has been implemented in such a way that all components are aware of their position on the grid, an important feature when considering that each component will have to be used twice. This was not initially the case, when implementation of the interface began it was extremely static and not suited to use more than once. High level descriptions of the components are as follows.

Navigation – Part of the basic functionality of the application is to allow the ability to navigate through a stack of DICOMS. The navigation consists of a 'previous' and 'next' button at its core. This initial plan for this specific object was for it to simply contain this, however due to UX design issues circling around my interpretation of the TkInter Grid layout, it also includes buttons to display two stacks side by side and synchronise said stacks.

Annotation Tools – This object contains all of the tools involved with the annotation of images. These tools include an 'annotation mode' which enables click events on a canvas to plot points. It also has buttons to draw polygons, save the annotations and delete annotations.

Figure View – The object responsible for displaying the DICOM images. Each object in the UX is bound to a specific figure view, either one or two. This is what makes implementation of the dual screen functionality simple. This also contains a header that displays information about each slice loaded in the view such as the patient ID, the scan type and the slice numbers position in the stack.

Folder Browser – Essentially the start point of the application, makes use of TkInter tools to navigate through the users directory's allowing them to select a folder of choice to be loaded into the application.

4.3.3– Back-end Components

The application also requires functionality that is not displayed on screen, instead controls all components and the information that needs to be passed through them in order for the front-end components to behave properly.

DicomObject – One of the main design decisions that has affected the way in which the application works was the decision to make each DICOM file in a given stack its own unique object. This object contains all the information about the DICOM that the application needs and is the most important object in the project. In the applications scope, the 'stack' is essentially an array containing different instances of this object. This allows for easy navigation through a directory and means that the processing of each DICOM file only has to happen once. It also makes things like saving annotations far easier.

CSVOperator – The CSV operator is the object used to carry out all CSV operations by the application. This writes all annotations to a csv file saved in the directory of the annotated stack and defines the structure in which these annotations are written and displayed in the file.

Main – The main object is the start point of the application. This stores crucial information about the state of the application such as the currently displayed DICOM and the directory in which the application has taken the array of DICOM's from. It also controls the flow of the application. This class is usually interacted with via the use of global variables and contains methods to return certain aspects of the application that other objects may need to interface with.

4.3.4 – Class Diagram

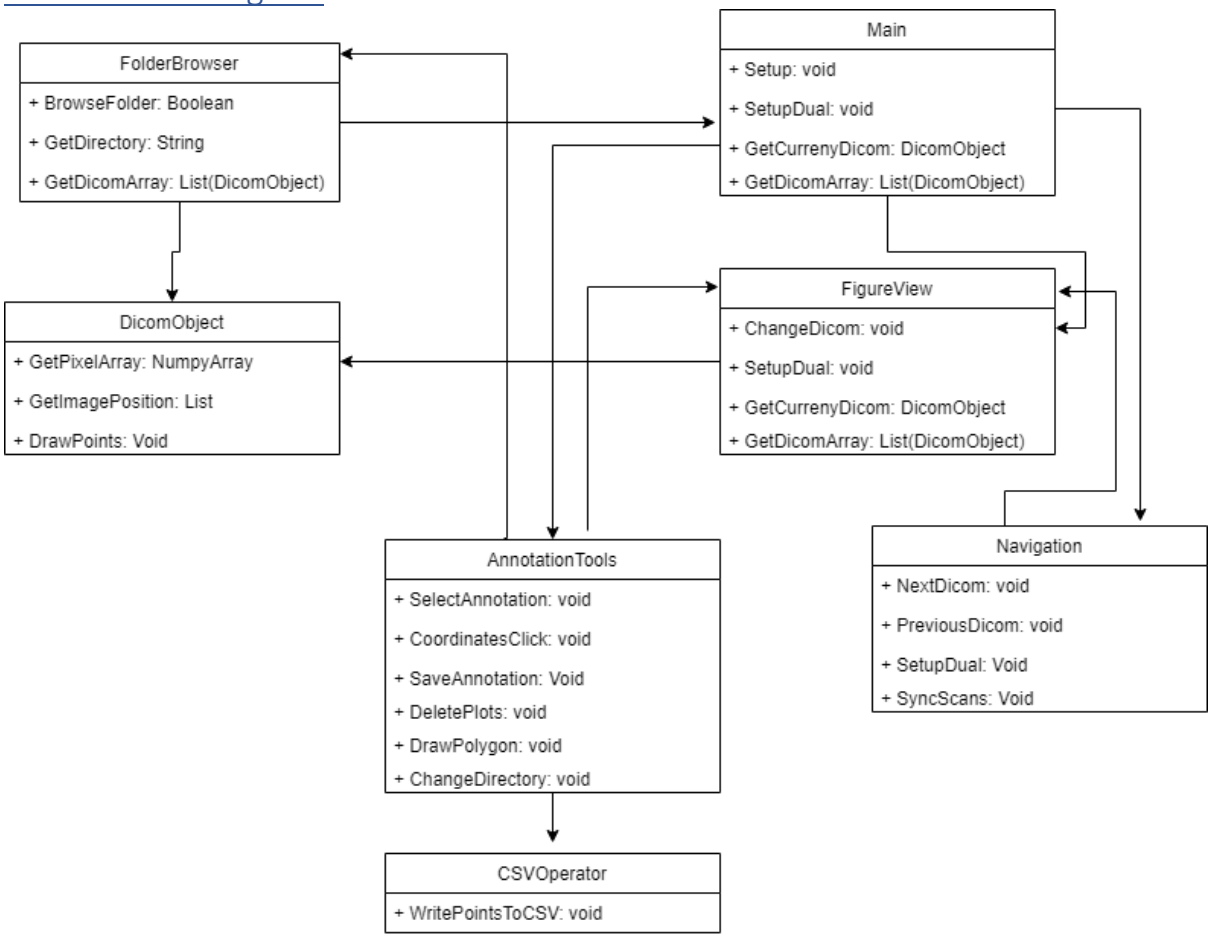

FIGURE 2 - CLASS DIAGRAM

Figure 2 - Class diagramFigure 2 describes how each class in the application interacts with others. This diagram also contains some crucial methods contained within the application, as well as what values they return. A good example of how the different classes interact with one and other can be seen between the 'Main' class, which waits for a return off the 'FolderBrowser' class before instantiating the 'FigureView', 'Navigation' and 'AnnotationTools' classes, which in turn change the behaviour of the 'FigureView' class. The 'AnnotationTools' class can also interact with 'Folderbrowser' again to Set up the dual screen and change directory.

4.3.5 – Information Flow

The way in which information flows through the components listed in section 3.1.2 and section 3.1.3 is crucial to both the efficiency in which the application runs and the user experience. A key example of this can be seen in the way that the application saves annotations to a specific DICOM file. This information must be passed through several different components.

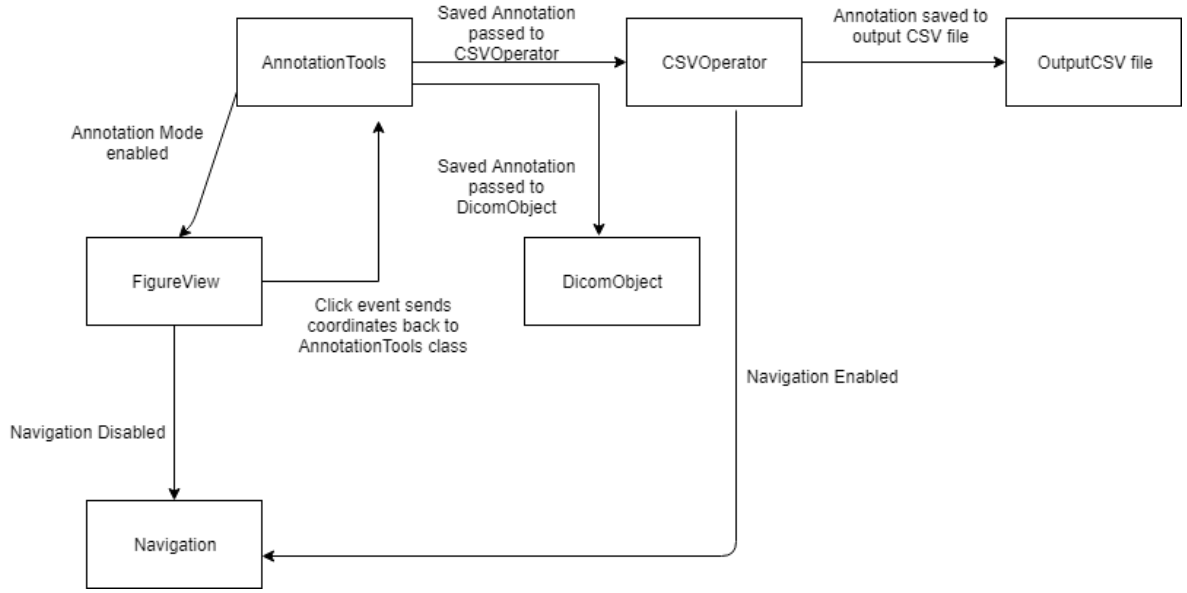

FIGURE 3 - INFORMATION FLOW DIAGRAM

As we can see from the diagram above, the user will begin the annotation workflow by enabling 'annotation mode' in the AnnotationTools component. This triggers a method in the FigureView class making the Figure interactable by way of click event. The user will then plot points on the figure by clicking different coordinates. These coordinates are then sent back to the AnnotationTools component. Upon saving this information using the 'save annotation' functionality in the AnnotationTools class, the annotation information is then passed into the CSVoperator class which will save this into it in a file with all relevant information. This information will also be sent to the specific DICOM object A which represents the slice that has been annotated. The coordinates of the annotation and the annotation type (polygon, point) get saved to DICOM object A. Then if the user revisits DICOM object A, the navigation will call a method to redraw all saved annotations for the given object. This is a prime example of how information flows through the application. Due to the object-oriented nature of the project, this is simple to do using getters and setters in each object.

The reasons for favouring a design like the one shown above are quite simple. The design considers the specification of the project; Viewing DICOMs, Navigating through DICOMs and annotating DICOMS. These three key pieces of functionality justify my decision to design this case in this way, it clearly demonstrates the ability to meet all three points.

4.4 – User Interface

The initial stages of this project involved prototyping user interface designs for the application. These designs were created using a wireframing application called Pencil[17].The user interface was designed with the requirements of the application in mind. These requirements specify that the application be efficient and practical, therefore there was no need to include any needless aesthetics.

4.4.1 – Single Screen View Design

The single view design refers to the user interface while it is displaying a single DICOM image. The below figure demonstrates the single screen view of the application.

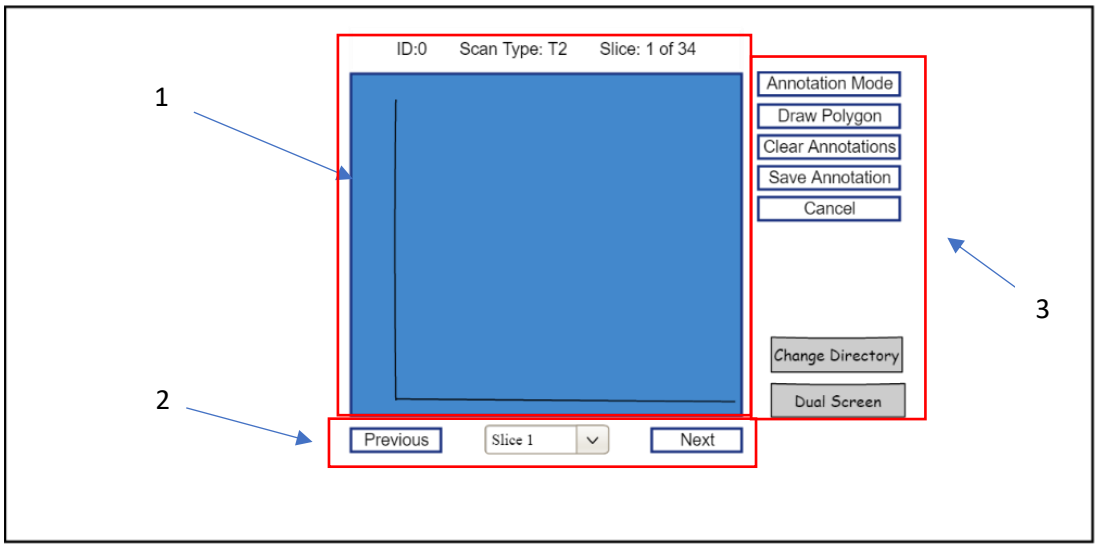

FIGURE 4 - SINGLE VIEW WIREFRAME

The single screen view has three main components, each having their own individual functionality. First, the figure(1) that displays the DICOM is clearly visible in the centre of the screen. As it is probably the most important component, it is taking the centrepiece of the screen is vital. Above the figure we see the information for the DICOM that is currently being displayed in the centre of the screen, crucial for the user experience.

The other two components then take stage around the Figure in the centre. The navigation bar (2) is placed directly below the Figure and is used for changing the image that is displayed in the Figure as we navigate through the stacks. This component also contains a combination box in the middle of the two buttons, labelled as 'Previous' and 'Next'. This combination box is to be used to pick a specific slice in the stack, making the navigation of the application far more useful.

Finally, the Annotation Tools(3)component is positioned to the right of the figure. This component will contain all functionality for annotating the image. This initial design does not include functionality to plot a circle on the screen, this is because at the time this

feature was considered additional work. The Annotation Tools component also contains the ‘dual screen’ and ‘Change directory’ functionality. The ‘Dual Screen’ functionality will be used to view two slices side by side.

4.4.2 – Single View Screen in Application

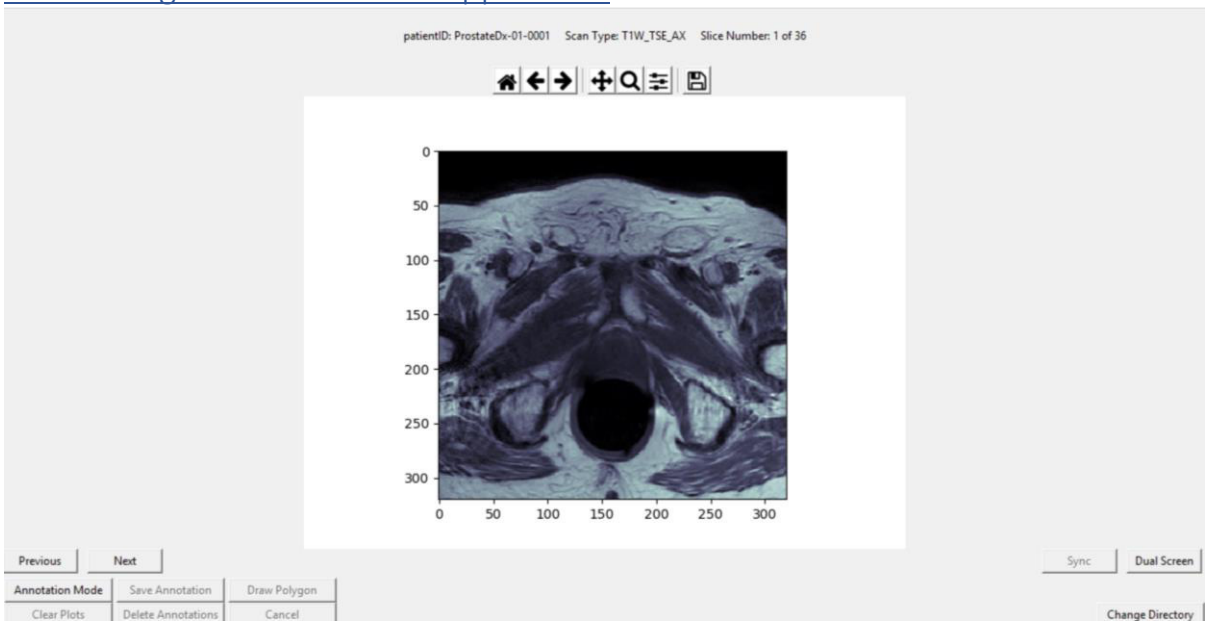

FIGURE 5 - IN APP SINGLE SCREEN VIEW

The above figure shows the User Interface for the single view screen in the application. The application contains all significant components outlined in the initial wireframes, however in application all the tools available to the user appear below the Figure. This is due to the nature of the dual screen view. The dual screen view would not fit in screen if the Annotation Tools were displayed on the right-hand side of each figure. Having all tools displayed directly below the Figure, from a UX perspective makes no difference at all. The ability for the bottom bar of tools to shrink to fit the functionality of the dual screen works perfectly in this case.

Another thing to note about the in-application interface is that some buttons are frozen out. This is due to the functionality only being available in certain cases. For example, the ‘Sync’ functionality is only available when the application has two stacks on either side of the application. A case could be made to only display buttons when they are usable, however for the purposes of the current application, making them unusable until certain conditions are met works fine.

One final difference to note between the initial designs shown in Figure 4 is the addition of the toolbar seen above the image displayed in the figure. This toolbar is a part of the matplotlib TkInter backend package[19] and allows users to manipulate the figure a few different ways such as zooming in on specific parts, panning the figure and an extremely useful coordinates tracker aiding the user even further when annotating areas.

4.4.3 – Dual Screen View Design

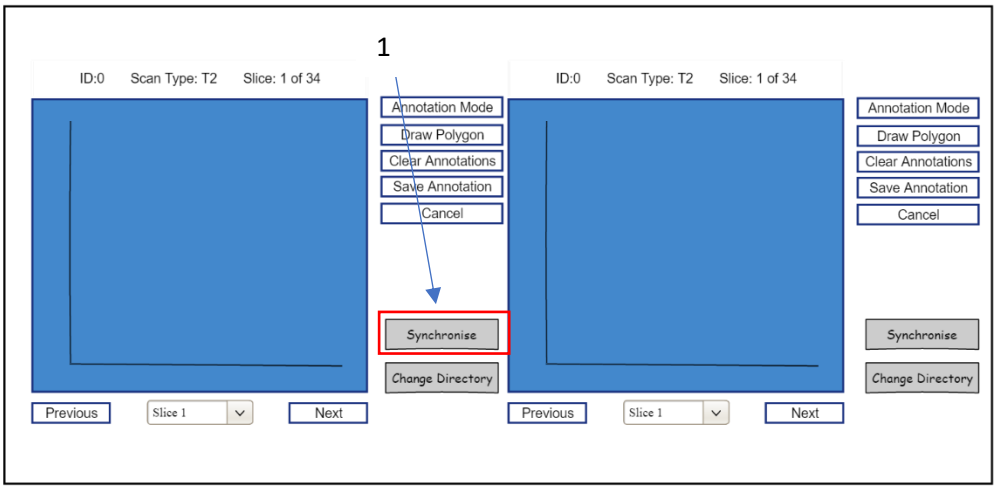

FIGURE 6 - DUAL VIEW PROTOTYPE

The above figure shows the wireframes for the dual screen functionality. Although this functionality was not part of the basic requirements of the application, it was important to design the application with the view to implement this feature. As described in 4.4.2, the implementation deviated away from the initial designs of the application, but the principles of it still stand. The basic principle of the dual screen is that each side have access to the same functionality as they would if viewed as a single screen, with the addition of the new 'Synchronise'[1] button, allowing the user to synchronise the two stacks.

4.4.4 – Dual Screen View Design

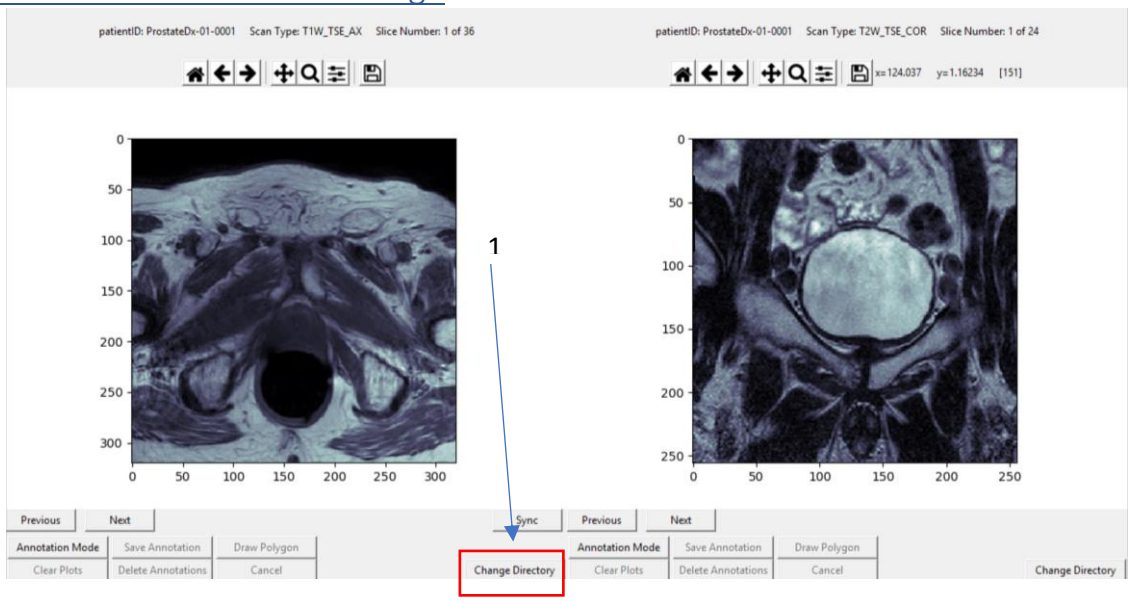

FIGURE 7 - IN APPLICATION DUAL SCREEN VIEW

Figure 7 shows the in-application view of the dual screen functionality. Again, we see that the design has changed from the original wireframes, this is purely due to screen space. There must be enough room for the DICOM to be its original resolution, therefore there could be a case where any components to the right of the main Figure would spill off screen and no longer be usable. The 'Sync' button[1] shown corresponds to the 'Synchronise' button

shown in 3.4.3. This functionality is available in this state due to the two slices being displayed side by side.

5. Implementation

In this section I will detail several components within the application, describe the design roadmap of each part, document its specific traits and features and the overarching uses it has within the project.

5.1 – Early Stages of Implementation

At the beginning of the project, much time was spent researching both the DICOM structure and PyDicom. The very first stage of implementation attempted was to display a single DICOM image in a TkInter window. PyDicom gives several different examples on how to do this[9] such as using external libraries like PIL and the stdlib TkInter module. The first successful attempt used the stdlib module which simply reads in a DICOM file and plots the pixel array as an image using Numpy[12]. This was a very simple implementation, but it would soon become apparent that there would be broader issues using this, mainly to do with making the program dynamic to allow navigation through stacks and the ability to annotate the images in question, two major requirements of the project.

The decision was then taken to approach the problem of displaying the DICOM image using the external python library Matplotlib. This library allows for plotting on a canvas within a figure. This was perfect for the needs of the project as the image will be displayed on the canvas and it allows for individual plotting of polygons and points on top of this. First implementations of the project were, in my opinion, not very good. This was due to inexperience with the packages in question, however I feel a lot of time was wasted during this period.

In the following sections, I will document specific parts of the application that are crucial to the functionality and usability of the application and that fit the requirements set out at the start of this project.

5.2 – Application Classes

5.2.1 – Main Class

The 'Main' class is the basis of the entire application. It sets up the flow of the app, creates the window in which everything else is displayed and contains important information about the global state.

```
class Main(tk.Tk):

    def __init__(self):
        # __init__ describes the initialisation of a python class - tk.Tk.__init__ describes the
        # initialisation of the tkinter module
        tk.Tk.__init__(self)
        self.title('Dicom Stack Annotator') #name of the program
        self.container = tk.Frame(self) #setup the container from which the rest of the app is displayed

        self.container.pack(side="top", fill="both", expand=True) # pack the container on screen

        #configure options for the geometry of the application
        self.container.grid_rowconfigure(0, weight=1)
        self.container.grid_columnconfigure(0, weight=1)
```

FIGURE 8 - MAIN CLASS INITIALISATION

The above figure shows how the main class is instantiated. The tk.Tk in the parameters passes a TkInter class into the main method of the application. This allows all TkInter modules and tools to be used within the app. The first few lines initialise the app, set up a container (named self.container), and configure settings for each row/column added to the main window.

This functionality shows the foundations of the application. The first stage of implementation did not function like this. Instead the app was set up outside of any class scope. This resulted in the constant use of global variables which is considered by many to be bad practice. It also disrupted the flow of the application. By designing the main class method of the application in this way, the class can fully control the way in which the app is set up before its even happened, whereas if it were to be implemented outside of the Main class scope, many aspects of the application would be on display before they were needed. We can see how the Main class is created and run in the following figure.

```
app = Main()
app.mainloop()
```

FIGURE 9 - INITIALISATION OF THE APPLICATION

The mainloop() command calls upon the tk.TK instance shown in figure 8 that creates the TkInter class.

The first instance of any interactable components is that of the ‘browse button’ shown below.

```
#instanciate the FolderBrowser class, which is the start point of the application
self.browse = FolderBrowser(self.container)
#the first thing the user see's, calls the setup method which creates the main body of the application
self.browse_button = tk.Button(self.container, width="15", text="Open Directory",
                               command=self.setup)
```

FIGURE 10 - INITIALISATION OF THE FOLDERBROWSER COMPONENT

Figure 10 demonstrates how the first interaction for the user is set up. This will display a button asking the user to open a directory from which they will load in the chosen stack of DICOMS that they require.

The FolderBrowser object will be discussed in further detail later in the report, but the basic functionality of the object is to read in a directory, if the directory contains files with the .dcm(DICOM) file extension, then it will return a Boolean value of True and create an array of processed DICOM files. The above code generates the screen shown below.

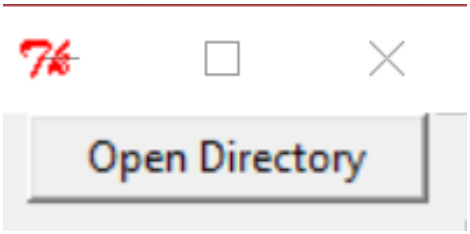

FIGURE 11 - INITIAL SCREEN SHOWN TO THE USER

This corresponds to the button “self.browse_button” shown in figure 10. The button executes the following method.

```
def setup(self):
    if(self.browse.browseFolder()):
        self.browse_button.pack_forget()
        self.state('zoomed')
        self.dicomArray = self.browse.getDicomArray()
        self.aStackFileDirectory = self.browse.getDirectory()
        self.setCurrentDicom(self.dicomArray[0], 1)
        self.figureOne = FigureView(self.container,1,self.getCurrentDicom(1),1,0,0,1,0)
        self.navigation = Navigation(self.container, self.figureOne, 1, 2, 0)
        self.annotationTools = AnnotationTools(self.container, 1, 3, 0)
    else:
        tkMessageBox.showinfo("Setup Error",
        "There are no DICOM files in this directory, try again")
```

FIGURE 12 - THE SETUP METHOD

This is where the main bulk of the application is initialised. As we can see in figure 12, this method waits on the FolderBrowser method to return true before it sets everything up, otherwise it throws a warning error to the user that they have selected a wrong directory using the tkMessageBox class.

The setup flow of the application was an extremely challenging task. From a UX point of view, the application has to be on rails at this point for the user. If all widgets were loaded without any information being passed to them, the UI would contain a lot of white space and would not useful. There for, the decision was taken to create the browseFolder method within the FolderBrowser class which allows for the application wait to until all required information was loaded before displaying all relevant components on screen. One potentially difficult piece of logic to consider was how the application would be made aware that the DICOM array has been loaded and how to then retrieve this array. To avoid error, the decision was taken for the return value of the browseFolder method to be a Boolean, instead of returning the array containing the processed DICOM files. Instead, the FolderBrowser class adopts object-oriented principles to allow for retrieval of the DICOM array using a get function.

5.2.2 – FolderBrowser Class

The initial state of the app is completely dependent on the functionality of the FolderBrowser class. The class itself is very simple, with limited but crucial functionality. It is one if two classes within the entire application that doesn't have any UI elements contained within, however the class is accessed through a UI element (the 'Open Directory' button).

```
class FolderBrowser(tk.Frame):

    def __init__(self, parent):

        self.dicomArray = []
        self.directory = None
```

FIGURE 13 - INITIALISATION OF THE FOLDERBROWSER CLASS

As shown in Figure 13, the class is initialised with a very small amount of information, only the Frame in which it belongs to. The self.dicomArray and self.directory variables are for use within the other classes within the application. In particular, the directory variable is used specifically by the CSVOperator class to write a CSV file to the same input directory, the dicomArray will be for use in all classes within the app.

```
575         def browseFolder(self):
576             folder = tkFileDialog.askdirectory(initialdir = "/")
577             self.directory = folder
578             count = 0
579             for filename in glob.glob(os.path.join(folder, '*.dcm')):
580                 self.dicomArray.append(DicomObject(filename, count))
581                 count = count + 1
582             if(len(self.dicomArray) > 0):
583                 return True
584             else:
585                 return False
586 
587         def getDicomArray(self):
588             return self.dicomArray
589 
590         def getDirectory(self):
591             return self.directory
592 
```

FIGURE 14 - FOLDERBROWSER FUNCTIONALITY

The functionality of the FolderBrowser class can be seen in figure 14. Here we loop over a directory retrieved from the tkFileDialog class, a class that allows us to view directories on the user's local computer. We limit the files we consider to that of the extension '.dcm'. This is to avoid any error when loading the remaining components in the application. If no files with this extension exist within the directory, the method will return false and the user will be given another attempt to load a directory. The main method then accesses the 'self.directory' and 'self.dicomArray' through the two get methods documented on lines 587 and 590 in figure 14.

Something to pay attention to is what the dicomArray contains. As we can see on line 580, we append a DicomObject to the array for each file in the directory. We pass in a filename and a count to tell the object what position it is in the directory. Initial implementations of the application did not work like this. Instead, the application would hold an array of filenames, processing and manipulating whenever a particular file was needed. This is an extremely inefficient way of designing the application and would lead to several problems later in implementation relating to annotation and navigation. Instead, each file in a directory gets turned into an object. Within that object, all relevant information needed is stored. An example of how this impacts performance is through the navigation of the application. Initially, when we navigated through the stack, we would iterate over the array of filenames, pass the filename into a PyDicom method to process and change the Figure Canvas to match that of the image generated from the new DICOM variable. This would have to be done multiple times for the same file if it were revisited. As we will see in the next subsection, the new structure of the DicomObject directly tackles this expensive way of handling each file in a directory.

5.2.3– DicomObject Class

Early stages of implementation involved a much different way of handling each DICOM file. Interactions with each file would involve processing the file using PyDicom on use. This was extremely inefficient because if the same file was required more than once the application would repeat the same calls multiple times. Another reason why this made things difficult was the nature of the annotations. Unless there were some way to bind the annotations to a specific DICOM file after we've navigated to a different file in the stack, then the way in which the application used annotations would be missing a crucial part of functionality. While annotations would still work as intended, consider the corner case of annotating a file with the user then accidentally navigating to a different scan in the stack. In this case all finished annotations would be lost from the screen. This could lead to them redrawing annotations, meaning that you would have multiple of the same annotation (but not on screen) in the CSV file and making the overall usability of the application much worse.

The final state of the application evokes a much cleaner and more efficient way of storing each DICOM file. As we can see in section 5.2.2, figure 14; when a user has selected a directory from which they wish to operate, the FolderBrowser.browseFolder method loops over the directory appending a DicomObject to a list. We also pass the filename of the DICOM and its position in the directory into the object.

```
444
445     class DicomObject():
446         def __init__(self,dicomFile, position):
447             self.filename = dicomFile
448             self.ds = dicom.dcmread(self.filename)
449             self.pixel_array = self.ds.pixel_array
450             self.patientID = self.ds.PatientID
451             self.scanType = self.ds.SeriesDescription
452             self.slice = self.ds.InstanceNumber
453             self.synchronised = None
454             self.stackPosition = position
455             self.polygons = []
456             self.points = []
```

FIGURE 15 - DICOM OBJECT INSTANSIATION

As seen in figure 15, the way in which the class handles the information passed to it is very simple. First we save the filename for future use, then it processes the DICOM using the 'dcmread' functionality of PyDicom, which essentially converts the file into a manipulatable object. Then it extracts information from said object that it needs like the pixel array, patient ID, etc. The object also has placeholder variables like polygons and points which set up arrays to store annotations for that individual DICOM in and finally a variable that will hold the position of the slice in a separate stack that corresponds to this DICOM, this functionality will be discussed further in later sections.

In my opinion, this piece of functionality is one of the best examples of how the project has embraced object oriented principles, as everything within this class is accessible using getters and setters, the DICOM itself is an object with all processing of information done within the class itself rather than every time a file is called to use.

```
528         def drawAnnotations(self, axes, figure):
529             if len(self.points) > 0:
530                 self.drawPoints(axes)
531             if len(self.polygons) > 0:
532                 self.drawPolygon(axes, figure)
533 
534         def drawPoints(self, axes):
535             for coords in self.points:
536                 for points in coords:
537                     point = points.split(',')
538                     x = float(point[0])
539                     y = float(point[1])
540                     axes.scatter(x, y, c='y', s=10, zorder=2)
541                     axes.figure.canvas.draw_idle()
542 
543         def drawPolygon(self, axes, figure):
544             for coords in self.polygons:
545                 polygonShape = Polygon(coords, closed=True, figure=figure, zorder=3, fill=False, edgecolor="yellow")
546                 axes.add_patch(polygonShape)
547                 axes.figure.canvas.draw_idle()
548 
```

FIGURE 16 - DICOMOBJECT DRAW FUNCTIONALITY

A good example of the way in which information flows through the application can be seen within the DICOM objects themselves. In Figure 16, we see three methods responsible for drawing saved annotations to the figure. Upon navigation back to a slice, hence changing the figure to the required DICOM, the navigation tools will call the drawAnnotations functionality of the current DICOM. As the annotation tools object has an idea of which figure is its parent, it can easily pass the figure into the method contained within the DICOM object, allowing the object itself to draw information on the canvas. The information is passed between three separate objects and the annotations are drawn on the figure at the same time as the DICOM pixel array, meaning that the user does not have to wait any time at all to see their annotations on screen.

5.2.4– FigureView Component

The FigureView component is responsible for the display of the DICOM image retrieved from the pixel array we return from the DicomObject. The main functionality of this component comes from the matplotlib package. However, in this project the package behaves in a slightly different way to normal implementations as we have to make sure it is aware that it will be implemented within a TkInter window. Below we can see how we do this.

```
11      import matplotlib
12      matplotlib.use('TkAgg')
13      import matplotlib.pyplot as plt
```

FIGURE 17 - MATPLOTLIB 'USE' METHOD

This ensures that we can use the required matplotlib methods for displaying the canvas within the TkInter window. We must also call the ‘use’ method of matplotlib before we import Pyplot[13] (Matplotlib’s specific python plotting package) as this is essentially saying that Matplotlib requires certain backend functionality. This functionality will not be usable within Pyplot if we have imported it before its aware we are using TkInter.

```
def __init__(self, parent, axes, initialDicom, number, toolRow, toolCol, figRow, figCol, labelRow, labelCol):
```

FIGURE 18 - FIGUREVIEW INSTANTIATION VARIABLES

As seen in figure 18, the FigureView component takes several parameters when we instantiate it. These parameters are mainly used to define where the figure lies in the window. This functionality is needed due to the dual screen option that the application offers. This control is a good example of the dynamism in the application as well as showing good object-oriented practices. The FigureView class is used more than once in the application, therefore when it is instantiated, it is aware of what figure it is on the screen, either one or two. This object behaves in such a way that the application can essentially have as many figures as the user would need, however for the purposes of the current state of the application this is limited to two.

```
607             self.f = plt.figure()
608             self.a = self.f.add_subplot(1,1,1)
609             self.imageShow = self.a.imshow(initialDicom.getPixelArray(), cmap=plt.cm.bone, zorder=1)
610
611             self.canvas = FigureCanvasTkAgg(self.f, master=parent)
612             self.canvas.get_tk_widget().grid(row=figRow, column=figCol)
613             self.canvas._tkcanvas.grid(row=figRow, column=figCol)
```

FIGURE 19 - FIGUREVIEW FIGURE FUNCTIONALITY

Figure 19 shows how the display of the Figure is set up. This functionality is dependent on the matplotlib.pyplot package (shown as plt in figure 19, line 607). We create the figure for display using plt.figure() then add a subplot to it, which becomes the axes on the figure on line 608. We then use the imshow method contained within the subplot object that takes an array as an argument and displays this on screen according to the values contained within the array. The ‘cmap’ property of the imshow method defines what colour mapping we use for the pixels in the array. In this case we use cm.bone, which is a standard colour map for MRI scans. The ‘zorder’ attribute is also used in order to allow for annotations to be plotted on the same figure, over the image; this will serve as the bottom layer of the plot.

The use of the TkInter backend that the application calls in the imports can then be seen on line 611 in Figure 19. This defines a canvas which will contain the figure generated on the line above. We need this specific object (FigureCanvasTkAgg) as this is what will connect the generic matplotlib canvas to the TkInter window. The lines following line 611 in Figure 19 define the position of the canvas in the window. The values used are, as described above, pulled from the values passed in as parameters when we instantiate the FigureView object. The resulting view created by this code can be seen in figure 20 below.

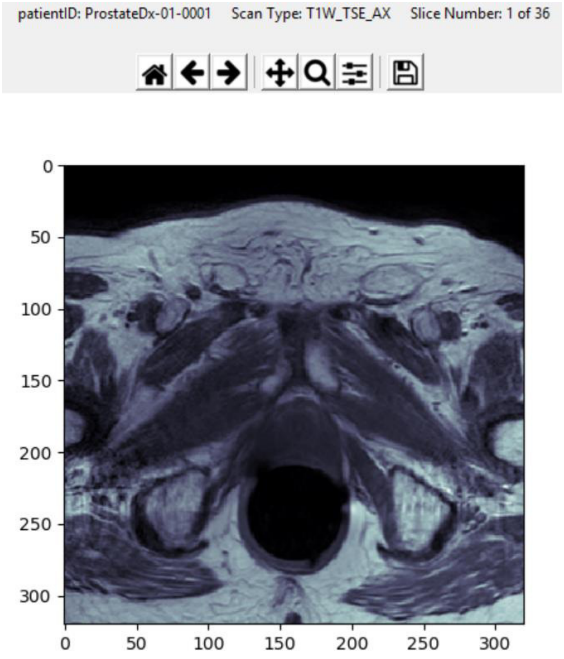

FIGURE 20 - FIGURE DISPLAY IN RUNNING APPLICATION

As shown in figure 20, the instantiation of the FigureView object results in a DICOM clearly displayed on screen, with an axis defining the dimensions of the DICOM and a canvas in which the user can interact with to annotate any potentially cancerous areas that they might identify. Figure 20 also demonstrates information about the particular slice in question shown as a header above the scan. This information is generated and displayed within the FigureView component. The justification for this not being its own component is due to it being of limited functionality and all the information we would need will be contained within the FigureView component anyway. This is a simple addition to the class, whereas displaying this as its own component would require extra UX functionality and more code.

Directly above the main figure is a toolbar which is generated using functionality in Matplotlib's TkInter backend. This toolbar is an extremely useful piece of implementation for both the user and testing the application. This Allows us to zoom in on any part of an image, changing the aspect ratio but crucially still keeping a clear indication as to what coordinates the user is viewing on the axes. In terms of testing, the toolbar is also useful for displaying what exact coordinates the user is hovering over with their mouse. This can be used to check if annotations that appear in the output CSV file are correct. Below is an example of the toolbars zoom functionality. As this figure shows, the aspect ratio of the image has changed, however the user will still be aware of what area the zoomed image resides in within the image coordinate system.

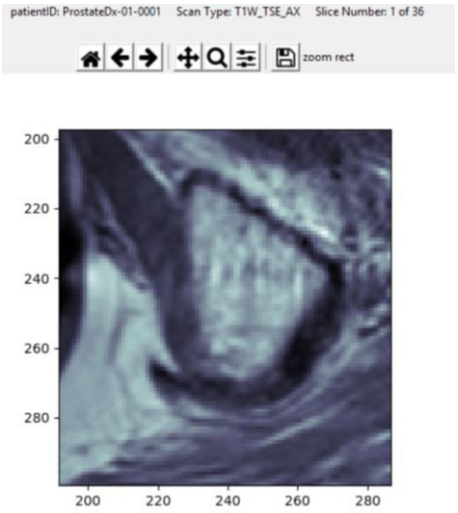

FIGURE 21 - ZOOM FUNCTIONALITY

Other classes within the application also make use of the generated Figure. Classes such as Navigation and AnnotationTools will be made aware of which figure they interact with. This allows them to interact with the FigureView class in ways such as changing the image displayed on the figure and drawing plots. Each application class that belongs to a figure gets instantiated with the figure number as a parameter. This gives it the ability to call methods within the figure externally, such as the 'ChangeDicom' Method shown below. Any class that calls this method will pass a DICOM object into it, changing the display of the figure to match that of the pixel array returned from the DICOM given in the parameters. The 'changeAll' method is also a good example of how the information above the applications figure is changed dynamically with every image.

```
def changeDicom(self, dicomFile):
        self.imageShow = self.a.imshow(dicomFile.getPixelArray(), cmap=plt.cm.bone, zorder=1)
        self.a.figure.canvas.draw_idle()
        self.changeAll(dicomFile)
        dicomFile.drawAnnotations(self.a, self.f)

def changeAll(self, dicomFile):
    self.idLabel['text'] = "patientID: " + dicomFile.getPatientID()
    self.sequenceType['text'] = "Scan Type:  " + dicomFile.getScanType()
    self.sliceNo['text'] = "Slice Number: " + str(dicomFile.getSliceNo()) + " of " + str(len(app.getDicomArray(self.figureNumber)))
```

FIGURE 22 - DYNAMICALLY CHANGING THE IMAGE AND INFORMATION SHOWN IN THE APPLICATIONS FIGURE

5.2.5– Navigation Component

The Navigation component handles the navigation between slices in the stack. The component itself also contains functionality to create the dual screen used for comparing stacks side by side as well as the functionality to synchronise said stacks. The Navigation component is displayed directly below the figure in the window, as we can see in figure 23.

FIGURE 23 - THE NAVIGATION BAR DISPLAYED IN THE APPLICATION WINDOW

Deciding on how the UX of the navigation would work was one of the more difficult stages in the implementation. In my opinion, it is also one of the weaker design aspects of the application. I believe that despite the button names being intuitive and easy to use, the buttons for the synchronisation and dual screen functionality do not belong in the same component as buttons used for navigating through the stack, however the argument could be made that the functionality of both does somewhat effect the way in which the user interacts with the navigation.

```
class Navigation(tk.Frame):
    def __init__(self, parent, figure, number, row, col, app):
        tk.Frame.__init__(self, parent)
```

FIGURE 24 - THE NAVIGATION INSTANTIATION

As Figure 24 shows, the way in which the navigation is instantiated is similar to that of other UX components. The class is passed in the row and Colum position in the parameter, required due to the dual screen functionality. We also pass the application figure in so it is aware of which figure it will be operating on. The Dual Screen and Sync buttons shown in Figure 23 will be displayed depending on what integer is passed in as the 'number' parameter. This is because due to the functionality of the application, these will only need to be displayed on the first instance of the navigation class (therefore will only be displayed if '1' is passed in).

```
    def nextDicom(self, figure, app):
        #As each dicom object in the array is aware of what position it is in the stack,
        #we can use this to navigate back by calling the getNext() method.
        #special for -1, which means we are position 0 and trying to navigate to
        # the last position in the stack
        dicom = app.getCurrentDicom(self.navNumber)
        next = dicom.getNext()
        array = app.getDicomArray(self.navNumber)
        if next < len(array):
            # Change the currently displayed dicom
            app.setCurrentDicom(array[next], self.navNumber)
            # Clear all annotations on the figure
            figure.changeClear()
            # Change the figure to display the next slice in the stack
            figure.changeDicom(array[next])

        else:
            # for cases where we are on the last slice in the stack
            app.setCurrentDicom(array[0], self.navNumber)
            figure.changeClear()
            figure.changeDicom(array[0])
```

FIGURE 25 - EXAMPLE OF NAVIGATION FUNCTIONALITY WITHIN THE NAVIGATION CLASS

Figure 25 demonstrates how the application handles navigation between slices. The first thing to note is how the main component of the application is passed down to the method using the 'app' parameter passed to the component when we instantiate it. This allows us to access the DICOM that is being currently displayed. It then uses a method within that DICOM object that will retrieve the next position in the stack. The value returned from said method is then used to retrieve the 'next' position in the stack. Then it checks to see if the number is not greater than the length of the stack itself before attempting to change the figure view (This is used to see if we're on the last slice in the stack). Finally it clears the

canvas of any annotations performed to the currently loaded DICOM and changes the image in the figure using the ChangeDicom method of the FigureView component (shown in Figure 22, section 5.2.4) passing in the next DICOM object in the stack. This process is repeated in reverse for navigating backwards through the stack.

The way in which the application handles the dual screen mode is quite simple. When the user presses the 'Dual Screen' button shown in figure 23, we call a method in the Main application that simply sets up the second screen in view creating new FigureView, Annotation and Navigation objects. All classes are designed with this functionality in mind using the parameters to determine their position on screen and in certain cases (such as that of the Navigation class) effects what functionality they have access too.

Synchronisation Functionality

The 'Sync' button calls a method that synchronises the two stacks displayed on either side of the application. This functionality is in my opinion additional functionality to the initial requirements of the app, however it is functionality that massively increases both the usability of the application and the effectiveness in which it meets the desired goal of making the process of delineating potentially cancerous areas in MRI scans. When the user clicks the 'Sync' button, it calls the 'syncScans' method within the Navigation class. This method is responsible for the synchronisation of the stacks.

```
255             # here I get the two stacks
256             stackA = app.getDicomArray(1)
257             stackB = app.getDicomArray(2)
258
259             # get the first and second slices in the stacks as reference
260             aSliceZero = stackA[0].getImagePosition()
261             aSliceOne = stackA[1].getImagePosition()
262
263             bSliceZero = stackB[0].getImagePosition()
264             bSliceOne = stackB[1].getImagePosition()
265
266             aDirectionRef = np.subtract(aSliceOne,aSliceZero)
267             bDirectionRef = np.subtract(bSliceOne,bSliceZero)
```

FIGURE 26 - THE INITIAL STEPS OF THE SYNCHRONISATION PROCESS

The first lines in Figure 26 show the retrieval of two stacks from the Main class of the application. The next step is to get the Image Position[14] of the first two slices in each stack. Each DicomObject in the stack has a variable retrievable through the getImagePosition method which corresponds to the image position of the DICOM scan in question (obtained from the meta data within the file). The image position attribute is a matrix that details the first pixel in the top left-hand corner of the DICOM file image. When attempting to synchronise the two stacks in question we must obtain the direction that each stack increases in. From here we can project points from one stack onto the other.

Before attempting to project points, it first checks to see if the two stacks are aligned (comparable). If not aligned, this makes synchronisation much harder and this application does not accept stacks that are not aligned with each other. To check if the two stacks are aligned, first is has to calculate if each slice in the stack is going in the same direction.

```
            # on each loop get the image position of slice k & k+1 in the stack
            referenceA = stackA[k].getImagePosition()
            referenceB = stackA[k+1].getImagePosition()

            # calculate the direction between the two slices
            directionA = np.subtract(referenceB,referenceA)
            # then calculate the dot product between the reference direction
            # (slices 0,1) and the direction calculated above
            dot = np.dot(directionA, aDirectionRef)
            # get the norm of the reference direction and the one above
            same = np.linalg.norm(directionA) * np.linalg.norm(aDirectionRef)


            # check if these are the same, get the absolute value as some will go into
            # negative (slight differences between the numbers could drop into exteremly small negative decimals)
            product = same - dot
            value = abs(product)
            # if the value is within a 0.1 range  could be increased  then fine
```

FIGURE 27 - CALCULATING IF EVERY SLICE IS GOING IN THE SAME DIRECTION

Figure 27 shows how this calculation is performed. It loops over each stack, comparing adjacent slices. First, it subtracts the image position of slice[k] from slice[k+1] to get the difference between two slices. Then it calculates the dot product of the difference between the two slices and the direction reference we calculate in Figure 18 (the difference between slice[0] and slice [1]). We also multiply the norm of the two matrix subtractions. We check if these two values are the same, or if not the same, virtually identical as sometimes there can be slight mathematical errors. For the purposes of this application there is a 0.1 limit for the difference between the two numbers. If this limit is met for every slice when we iterate over the stack, it can safely be said that the stack increases in the right direction. This process is repeated for the secondary stack. To perform these calculations the application makes use of methods contained within the Numpy package, shown as 'np' in Figure 27.

```
if aStackDirection & bStackDirection:
    # here we check for allignment between the two stacks.
    alignedDot = np.dot(aDirectionRef, bDirectionRef)
    alignedDist = np.linalg.norm(aDirectionRef) * np.linalg.norm(bDirectionRef)

    product = alignedDist - alignedDot
    value = abs(product)
    if value < 0.1:
        aligned = True
```

FIGURE 28 - CHECK TO SEE IF THE TWO SLICES ARE ALIGNED

If the direction of both stacks increases in a manner which is considered normal, the application then checks to see if the stacks are aligned before performing the final step in the synchronisation process. Simply, it performs the same calculation as is show in Figure 27, in this case on positions [k] and [k+1] in both stacks, shown by aDirectionRef and bDirectionRef in the figure above. If the difference between these two points is again within the 0.1 range, it can be said that the two stacks are aligned.

In the final stage of the synchronisation, we make use of the functionality of the DicomObject class. We iterate over stack A and for each iteration of A, iterate over stack B. Here we can check which slice in stack B is the 'best fit' for a given slice in A. The first calculation shown below is to get the position that slice[i] in stack A lies on the line:

pointA = np.add(stackA[0].getSliceLocation(), (i * aDirectionRef))

Then we loop over stack B, for each iteration we perform the logic shown in figure 29.

```
349                     pointB = np.add(stackB[0].getSliceLocation(), (k * bDirectionRef))
350                     #the first half of the algorithm, this corresponts to A[0] + dot(B[k]-A[0],d) * d
351                     # where pointZero is A[0], then we get the dot product of (pointB - A[0]) and d, the direction from slice 0 -> 1
352                     #and divide that by the norm of d
353                     bProject = np.dot(np.subtract(pointB,pointZero),aDirectionRef) / np.linalg.norm(aDirectionRef)
354
355                     #here we have the second half of the algrithm  d / norm(d)^2
356                     normD = aDirectionRef/(np.linalg.norm(aDirectionRef))
357
358                     #multiply the two together
359                     mult = bProject*normD
360
361                     #add to point A[0] (the first point on the line)
362                     Bpk = np.add(pointZero, mult)
363
364                     #get the norm of pointA minus the projected point to get its position on the line
365                     normPoint = abs(np.linalg.norm(np.subtract(pointA, Bpk)))
366                     compare = abs(np.linalg.norm(aDirectionRef)/2)
367
```

FIGURE 29 - SYNCHRONISATION CALCULATION FOR A GIVEN SLICE

Line 349, shows the calculation of the point in the B stack we want to project onto the A line. To project onto the line, we calculate the following formula for point [k] in b.

(A[0] + dot(B[k]-A[0],d) * d) * (d/norm(d))

Where A[0] is the first point in the stack, dot is the dot product of point B[k] – A[0] and the direction reference d of the A stack (the distance between point A[0] and A[1]). This is shown in figure 29 using the variables bProject, normD and mult. The projected point B on the A stack line is calculated on line 362. To compare it to point A along the A stack line, compare the two values calculated on line 365 and 366. If the projected B point on line A is less than the point we are comparing it to in stack A, and this distance is less than any other point previously calculated, the application takes this point in Stack B as being directly comparable to slice[i] in A. This value then gets added to the DIcomObject using the ‘setSynchronised’ method. If no slices in the stack are comparable, an error message will be displayed telling the user that this is the case. The user can then see what slice in the B stack corresponds to any given slice in stack A by using the ‘Synchronise Scans’ button, which takes the place of the ‘Sync’ button (shown in figure 23) demonstrated below.

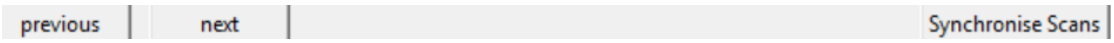

FIGURE 30 - ‘SYNCHRONISE SCANS’ BUTTON

As you can see, this is a considerable step up in complexity compared to other functionality within the application and is one of the defining features of the project. Due to the complexity of this, I consider it additional functionality to that of the initial requirements and as previously stated, one that increases the overall usability of the application for radiologists.

5.2.6– Annotation Component

The majority of the annotation functionality is contained within the Annotation class with the exception of some annotation drawn from within the DicomObject class (section 5.2.3). The instantiation of the Annotation class behaves in the same way as other UI components documented within this report such as the Navigation class, where we pass in parameters that represent the objects location in the window and the figure it is bound to (the figure upon which it can draw).

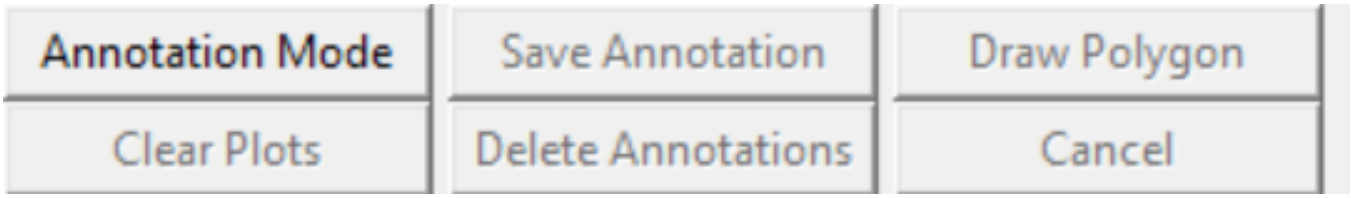

FIGURE 31 - ANNOTATION CLASS REPRESENTED IN WINDOW

The main body of the class consists of buttons, as seen in Figure 31. These buttons give the user access to different tools. The UX design flow of this class was in my opinion the most difficult phase to implement due to the functionality of certain buttons only being useful when, for example, an annotation has been drawn on screen as is the case with 'Clear plots', 'Save Annotation', 'Draw Polygon' and 'Cancel'. Another case to consider is the 'Delete Annotations' button, which needs annotations to be saved to the DicomObject to have any effect at all. Figure 31 shows the initial state of the application, which only allows the user to select 'Annotation Mode', enabling them to annotate the DICOM image on screen.

The annotation functionality itself makes use of the 'mpl_connect' method of matplotlib, creating click events accessible on the canvas. When the user clicks any point of the Figure, the data from this click event is stored, shown in Figure 32 below.

```
def coordinatesClick(self, event):
    coords = (str(event.xdata) + "," + str(event.ydata))
    self.a.scatter(event.xdata, event.ydata, c='y', s=10, zorder=2)
    self.a.figure.canvas.draw_idle()
    self.annotationCoords.append(coords)
    self.setPolygonPlot(event.xdata, event.ydata)
    self.drawPolygon.config(state='normal')
    self.saveAnnotation.config(state='normal')
    self.clearPlots.config(state='normal')
```

FIGURE 32 - ANNOTATION CLASS REPRESENTED IN THE WINDOW

As seen in Figure 32, we take the X-axis and Y-axis event data from the click event, and we use a method called 'scatter' which is part of the matplotlib package. This then draws a single point on the Figure using the 'draw_idle()' method, which again is a part of matplotlib's library and is responsible for redrawing the Figure, appending any updates which have been passed to it using the 'scatter' method. An example of this can be seen in Figure 33.

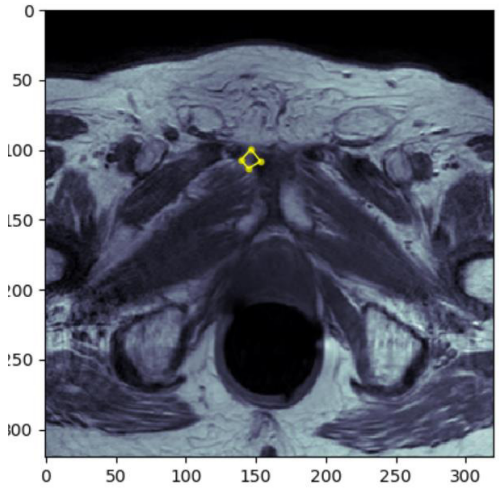

FIGURE 33 - YELLOW POINTS ON SCREEN REPRESENTING ANNOTATION

Figure 33 also shows the results of the 'Draw Polygon' functionality which works in a similar way to that of drawing points, however instead of using the 'scatter' method, we first create a polygon object through matplotlib by passing the object a set of coordinates representing the points along the polygon (drawn using the method shown in Figure 33). Then to display the polygon, the application makes use of the 'Add_patch' method, again within matplotlib, to draw this on the application Figure.

To save the annotations, the application makes use of a class called CsvOperator. This class is the simplest within the project and is simply responsible for writing the annotations to a CSV file when the user clicks the 'Save Annotation' button shown in Figure 32. The CsvOperator class uses an external module called 'csv' which handles the writing of variables to CSV files.

This functionality is a key example of how the project has met the initial requirements needed to make this a useful and usable tool to radiologists. The initial requirement to annotate and draw polygons is extended in my application by granting the annotator the ability to enter the PI-RADS score of the annotated area. This leads to the annotations having much more value then just highlighting coordinates when the radiologist is reviewing the CSV file for annotations previously marked up.

6. Results and Evaluation

To judge whether or not the application holds up to the initial requirements set at the start of the report the overall functionality has to be analysed and evaluated. In section 3.2, the requirements outlined are to:

1. To make it easy to display and view MRI scans.
2. To make it simple and efficient to navigate through a stack of DICOM's.

3. To make the annotation and identification of potential regions of interest far easier for radiologists.

The following paragraphs will outline how these requirements have been met and which parts of the application these requirements correspond to.

6.1 – How does the application meet required functionality?

6.1.1 - Requirement 1 – Display of MRI Scans

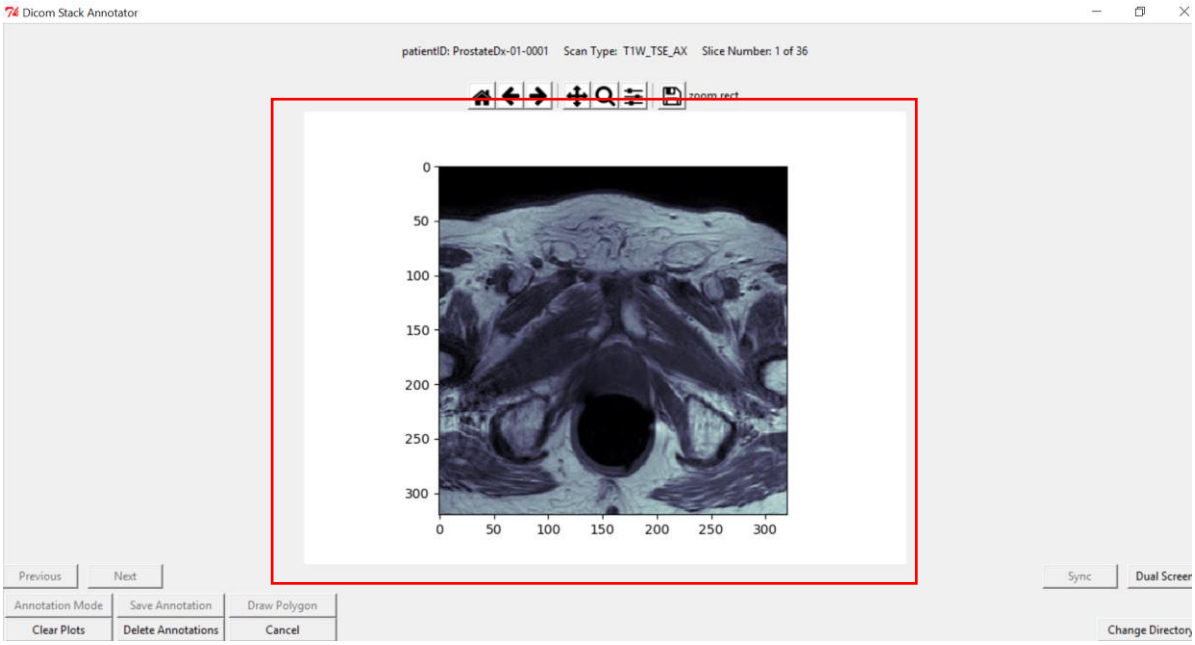

FIGURE 34 - RED BOX HIGHLIGHTING THE AREA OF THE APPLICATION THAT DISPLAYS THE MRI SCAN

As Figure 34 demonstrates, the MRI scan is clearly displayed on screen for the user to view. There are three aspects to consider that I believe are crucial to making this a useful piece of functionality for the user –

1) **Does the image keep the original resolution?**

The image does in fact keep the original resolution of the image contained within the DICOM file. This can be proved by looking at the X and Y axis attached to the applications figure. This describes the dimensions of the image and is not enlarged or reduced at all. For this reason, the application meets the requirement of keeping the original resolution of the image.
This however could be seen as a weakness of the application. As no limits to the image size have been imposed, there could be a case where the first image is too large for the screen. There is also the possibility that when using the 'dual screen' functionality, one image will overflow, hindering the extent to which this application would be useful. This scenario has yet to be encountered, but in theory this is possible.
We should also consider the aspect ratio of the image in question. The height and width of the image on initial display is specified by the dimensions in the pixel data, however the application will alter the aspect ratio of an image when the user zooms in on a region. An example of this can be seen below.

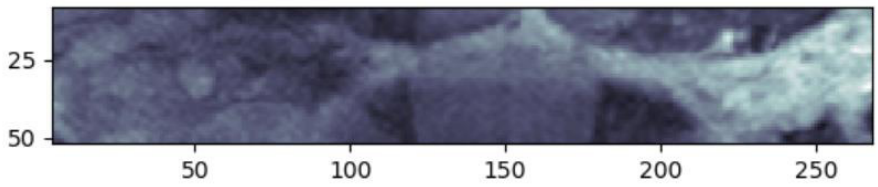

FIGURE 35 - ASPECT RATIO CHANGE ON ZOOM

2) **When annotating, will the user know what areas of the image they are delineating?**

Similarly, to the first case presented, the proof for this can also be seen in Figure 34 where the axis is clearly visible on screen. From this, the user will have a clear idea of which coordinates in the image they are annotating, an important factor when considering the contents of the output CSV file. Another scenario to consider is that of the annotations themselves. Upon annotating the image, the user can visibly see the area on screen highlighted by the yellow points or polygon around the area. This visibility greatly aids the user when delineating regions of interest.

6.1.2 - Requirement 2 – Navigation through a DICOM stack

The second requirement documents the need for the user to easily navigate through stacks. The navigation tools provided clearly give the user the ability to navigate through a stack.

The two buttons shown below are clearly labelled for the user to use intuitively, allowing them to navigate to the previous or next slice in the stack.

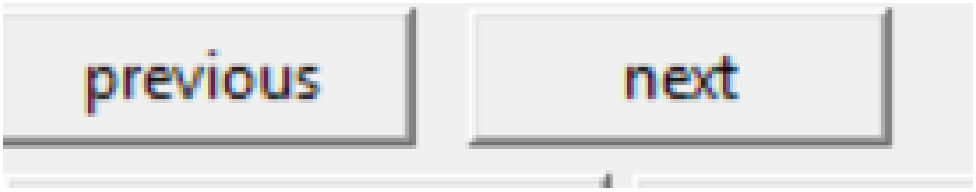

FIGURE 36 - EXAMPLE OF THE 'PREVIOUS' AND 'NEXT' BUTTONS IN THE APPLICATION

The navigation functionality is always available other than when the user is currently in the process of annotating a slice. This is for UX purposes, as they could essentially two slices and only see them represented in one slice when viewing the output CSV file. This is an important UX feature and is a clear example of how the navigation functions as intended.

We must also consider the speed in which users can switch between slices as a way of evaluating the navigation of the application. The user can switch between the slices with no noticeable delay in the display of the image in the centre of the screen. If the user holds down the button for too long, it will still only count as one move in either direction in the stack, for the purposes of this application I believe this behaviour to be desirable due to the small margin for error when navigating.

One final thing to consider when looking at the navigation through stacks is the overall size in memory of the stack of DICOM's we read in. If a stack is too large for the memory of an average computer, then this application will be unusable. When analysing the test data obtained from the TCIA, the average size of the DICOM stacks is anywhere between 16 to 256 slices. If we take the upper limit of the amount seen in these stacks, using the 'sys' package in python, we can see the size of the processed DICOMS in the application in system memory. A stack of 256 DICOM files will take up 2.2KB in system memory, a size all modern computers can easily cope with.

A possible weakness of the application is that if the user wishes to navigate to a specific slice in the stack, they must first navigate forward or backwards through the stack until reaching their intended destination. This is in my opinion, a big flaw of the application and could hinder its usefulness, however when accessing the actual requirements of the project, the navigation works to its intended purpose.

6.1.3 - Requirement 3 – Annotation of a slice

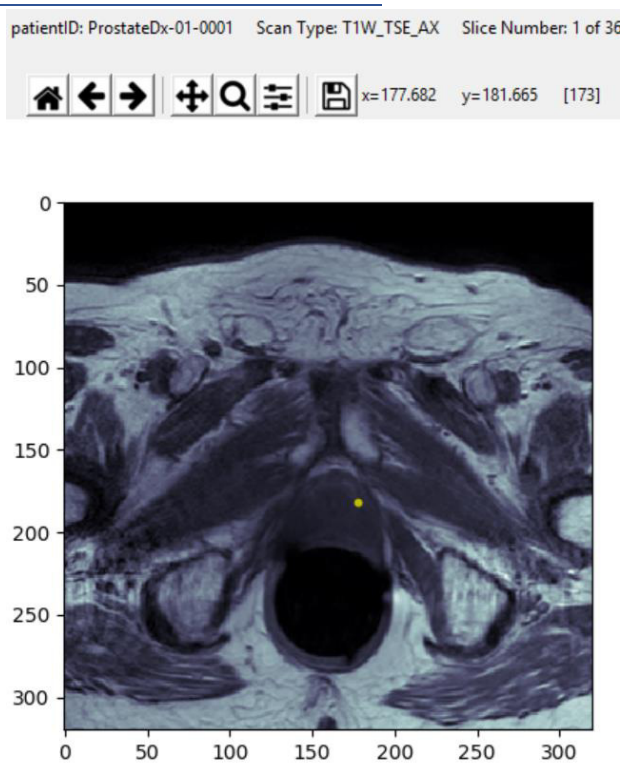

FIGURE 37 - EXAMPLE OF A SLICE ANNOTATION IN THE APPLICATION

Figure 37 clearly shows that the functionality of the annotation mode works. As documented in section 5.1.5, the annotation mode has several different tools that are usable by the user, such as plotting points, drawing polygons and saving annotations to file. If we look at the X and Y coordinates in the toolbar near the top of figure 37, this marks the location of the annotation on the screen. Figure 38 below demonstrates the output in the CSV file format of saving the annotation shown in Figure 37.

```
ProstateDx-01-0001,T1W_TSE_AX,1,1,points,"['177.6818181818182,181.66450216450215']"
```

FIGURE 38 - EXAMPLE OF AN ANNOTATION IN A CSV FILE

Figure 38 highlights the CSV output of an annotation more accurately than it does in the toolbar in figure 37, however, this output is accurate and without error, therefore a crucial piece of functionality is proved. There is an issue with regards to the readability of this output. The output has been designed with future work in mind that relates to reading it back into the program, however this does not make it very readable for the user. The user may not be able to tell what each column on a row means. For example, in Figure 38, the '1' before the 'points' field relates to the PI-RADS score that the user has given the area marked up. In the output file, there is no way of telling that this is the case.

The case could also be made that the annotation could work in a slightly more usable way. If a user makes a mistake, they must clear all plots on the screen and redraw the correct version of the annotation. This could work in a way that required less individual work from the user. When considering the requirements of the application, despite these flaws in the

design, the user still has the ability to annotate and delineate areas of interest in a scan, meaning that the key functionality of the application has been met.

6.2 – Additional Features

6.2.1 – Synchronisation Functionality

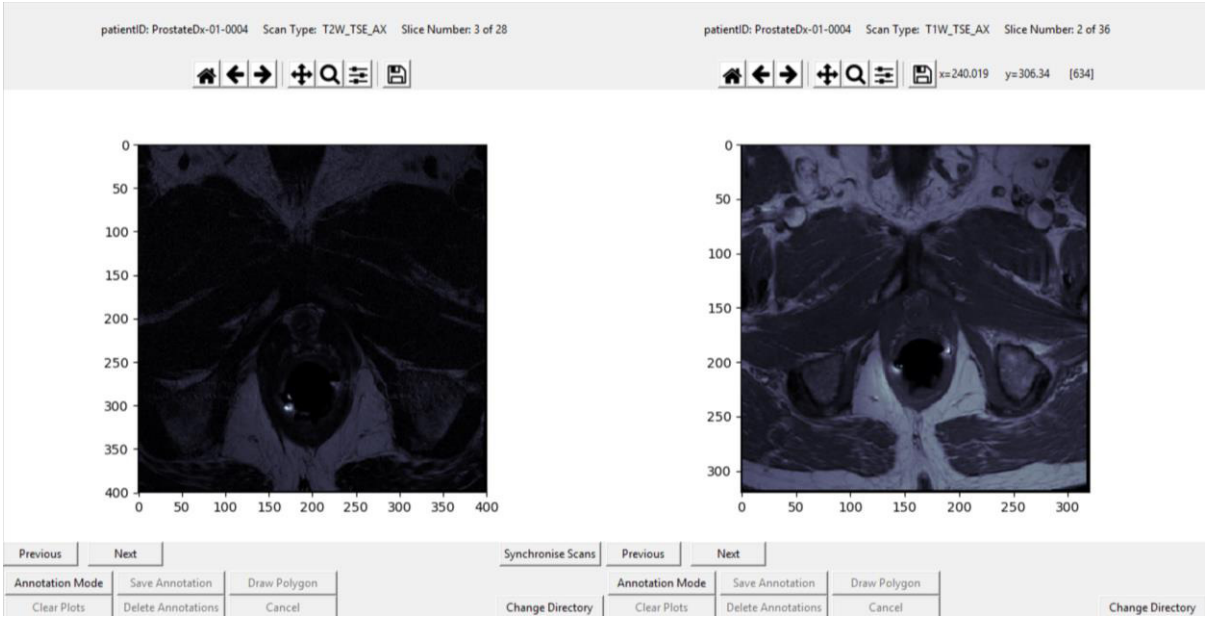

FIGURE 39 - DEMONSTRATION OF THE 'SYNCHRONISATION' FUNCTIONALITY

Figure 39 demonstrates two different stacks side by side after the 'Synchronisation' functionality has been used. The information above each figure shows that these two slices are in the T1 and T2 axial scan stacks. The two slices in question are ones that have been synchronised and correspond to the same area in the patient space.

This functionality is difficult to test. One way of telling if the two slices are in fact comparable is visually. The two images clearly document the same area in the patient space, however the left-hand space appears to view the area in a finer detail. This however is not a reliable method of testing.

The chosen method of testing this functionality is to load in two of the same stacks. When loading in two of the same stacks, it is natural to assume that position [n] in stack A would directly correspond to the same position in stack B. In this instance, the application holds up and loading in two of the same stacks will in fact result in position [n] in stack A synchronising to position [n] in stack B.

6.2.2 – Dual Screen Functionality

The ability to view two stacks side by side is not crucial to meeting the requirements set out at the start of the report and therefor can be considered as additional functionality. Testing this feature is quite simple.

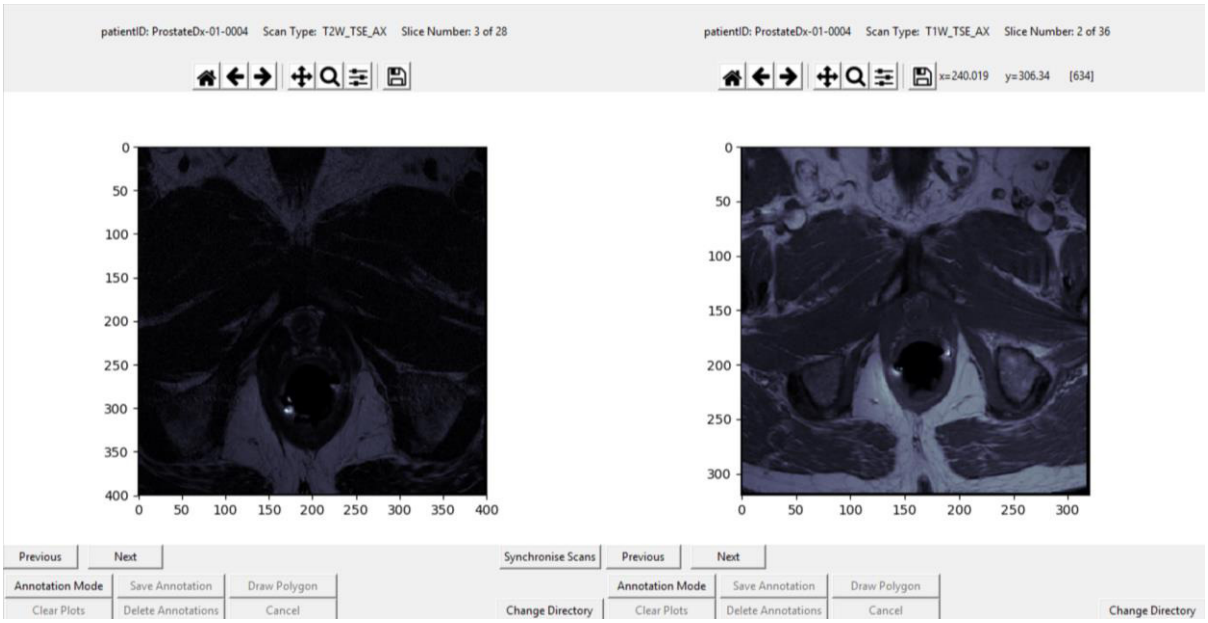

FIGURE 40 - COMPLETE VIEW OF THE APPLICATION

Figure 40 clearly shows the dual screen functionality working in app. It also clearly shows two different scan types being shown either side of the screen along with the repetition of the interface below the application figures. The only thing to consider when testing this feature is the information being passed into the second screen, specifically the directory. This works in the same way as the initial set up of the application. Figure 41 below shows code that will reject the directory, return false and the main class will display an error message if the user attempts to load in a directory that doesn't contain files with the 'dcm' extension. This code also ensures that if a directory contains .dcm files along with other file formats, only the former will be considered.

```
def browseFolder(self):
    folder = tkFileDialog.askdirectory(initialdir = "/")
    self.directory = folder
    count = 0
    for filename in glob.glob(os.path.join(folder, '*.dcm')):
        self.dicomArray.append(DicomObject(filename, count))
        count = count + 1
    if(len(self.dicomArray) > 0):
        return True
    else:
        return False
```

FIGURE 41 - BROWSEFOLDER METHOD OF THE FOLDERBROWSER CLASS

7. Future Work

In this report, I have outlined several reasons why I believe that the application has met the intended requirements outlined at the beginning of this project. I believe it to be a robust system that provides functionality that makes the process of delineating regions of interest in stacks of MRI scans easier and more efficient for radiologists, however there is still work to be done on the application to make it more useful. These features are things that I myself intended to implement within the project, however I believe the way in which I worked at the beginning of the project with a lot of time spent refactoring earlier implementations has lead to time constraints towards the end of the project, therefore there was not enough time to implement some key, additional features.

7.1 – Accessibility

The annotation functionality is in my opinion the most important feature within the application and without this, it would simply be a program allowing for users to view DICOM files, hardly useful by itself. Figure 33 shows how the annotation can be seen in the application window as a yellow dot or line.

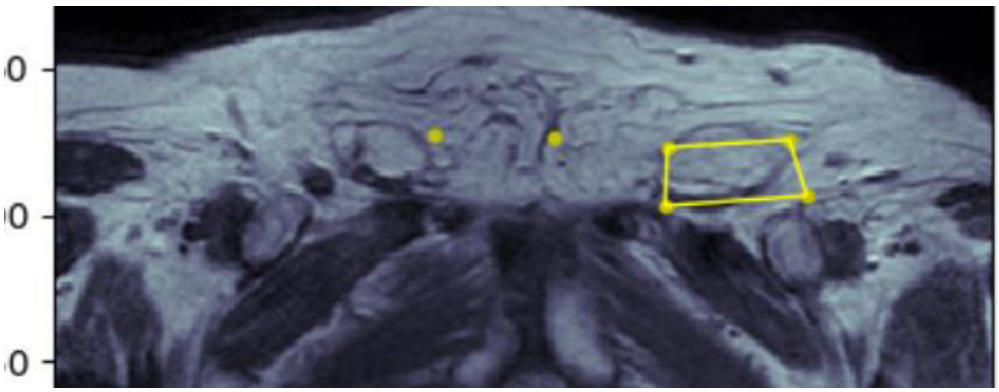

Figure 33: Annotations in application window

Colour blindness effects 1 in 12 men and 1 in 200 women worldwide[15]. Some sufferers of colour blindness are unable to see the colour yellow. This particular case of colour blindness would render the annotation functionality in the app virtually unusable except for the output in the CSV File. I believe a very useful feature to have would be to allow the user to pick a specific colour somewhere in the RGB matrix that ensures that they would have no problems seeing annotations on the screen. Another option would be to simply let the user enter a text string allowing them to pick a colour of their choice. Even though these examples would not be challenging things to add to the application, they would still require great thought and most importantly time.

Consider implementing by allowing the user to enter a string of their choice as a value for the colour. We would first have to tackle the issue of spelling mistakes, regex and language differences amongst many others. After this, we would then have to consider how the user perceives this functionality. What if they attempted to enter an RGB or hexadecimal value, should the application still accept this method of colour picking? Finally, we have to consider the capabilities of the tools we are using. If a user were to pick a colour that the matplotlib ‘scatter’ and ‘add_polygon’ functionality did not support, there would still be

errors in displaying the plots on screen. All cases presented are solvable but would require time in which to do so.

7.2 – Draw Circle Functionality

Another key piece of functionality that I didn't have time to implement is giving the user the ability to draw circles as an annotation. This could be incredibly useful in terms of allowing the user to mark-up potential areas of interest, but for UX purposes I decided not to do this.

The reason I decided against the implementation of this feature was due to the nature of Matplotlib's 'circle' functionality[16]. The way in which this part of the package works is by passing it a simple XY coordinate. This then allows for a circle to be drawn with a circumference specified in the actual code, meaning that the user can only specify where they want the circle, not how large it is or how much of the scan area it covers. This severely limits the functionality of the annotation as there could be cases where it is too big or too small, requiring the user to have to guess where the circle will be drawn.

One way of implementing this feature would be to have a circle that scales according to a drag function. This would mean that the user plots a point that they wish the circle to be drawn around, the circle gets drawn with an intermediate circumference and the user is then given the option to scale it as they please.

Due to the time taken to do this coupled with the fact that annotation currently already exists within the application, I felt that this was quite a low priority feature to implement. The most important feature to give the user is the ability to draw and save polygons to a specific slice in the stack and to then allow them to view said annotation in an output CSV file.

7.3 – In-app Directory Viewer

Another big flaw of the application is that there is only the ability to add two directories to the window to analyse at one given time. A normal patient scan sequence consists of four separate scans, within which there are multiple slices making up that scan sequence. These scan sequences are all separate subdirectories of one overarching patient directory. Only being able to add two different stacks at the same time to the application makes the navigation through patient scans much less effective. The ability to load in the entire patient's directory, then select which of the subdirectories to display on either side of the application is in my opinion a much more efficient way of navigating through the different scan sequences.

The implementation of an in-application directory viewer would be in my opinion a sufficient enough solution to this issue. The way in which such a feature would work would be rather simple. The user would select a patient directory in the same way as they select a specific folder to load in currently. This would then be displayed in the application with the subdirectories listed in view. Each subdirectory would then be interactable, giving the user the ability to click and select a folder which they wish to view in the application window. This would grant them freedom to navigate through the patient scan sequences as they

please, leading to a much better user experience, one which would make this application much more effective in tackling the problem outlined in section 2.

The way in which such a feature would work at the code level is similar to the way in which the FolderBrowser class works in section 5.2.2. We could extend this class to contain UI elements such as the viewer documented in the above paragraph. We could loop over the parent directory, first checking if the subdirectory contains files with the desired file extension (.dcm), then creating an array storing the contents of each file directory. Then, have two separate arrays representing the currently 'loaded' stacks which will be displayed in view. When a user selects a folder that they wish to view, this will take the place of currently loaded stack in that specific part of the window, be it stack one or stack two. We would then loop over the filenames, for each filename creating a DicomObject documented in section 5.2.3 and passing that into the currently loaded array in question.

There are two reasons why I have not implemented this in the application. The first reason is again time. This was initially planned to be in the application, however time constraints due to early implementations meant that I was only able to get the ability to load in two stacks at the same time working. The second reason for this is the user interface. Firstly, the current user interface doesn't have the required space to display something like this. For example, if a directory had 10 subdirectories (even though this is not normally the case), extra implementation would be required to ensure that this didn't spill over into other areas of the view. Secondly, the way in which TkInter's grid geometry works means that for this to look in any way aesthetically pleasing it would likely have to come as a part of another component to which it shouldn't really belong due to the scale of the UI elements involved. If I were to have one or two more days left in my implementation schedule, I feel with more thought given this could be easily implemented and would greatly increase the usefulness.

7.4 – Annotation Correction

One of the biggest flaws in the application relates to the way in which the annotations are removed from a slice. The program gives three different ways to delete annotations. To cancel the current annotation, removing new annotations from the figure, to remove all current annotations on screen but not delete them from the object itself, or to do a 'deep delete', removing annotations on screen and in the object. Consider a user plotting a point on screen in the wrong area while drawing a polygon around an area. If the user incorrectly plots a point on accident, they will have to either clear all plots on the screen or cancel the current annotation if there are previously saved annotations on the slice. This means more work for the user if they make a simple mistake and greatly reduces the usefulness of the annotation functionality.

One way in which this issue could be solved would be grant the user the ability to remove individual plots on the screen. There could be a button allowing the user to click and delete any plot that they wish, even entire polygons if all points in the polygon are deleted. This feature would mean that simple mistakes would not require the user to go through the entire process of re plotting points that were correct with the exception of the mistaken plot.

One way in which this could work would be to allow the user to click a plot on the application figure that they wish to remove. This click event would work in a similar way to that of the way in which users plot points on the screen, except instead of plotting the event data for the X and Y coordinates on the figure, we would loop through the array of current points plotted on the figure to find the coordinates closest to the given deletion points, remove them from the array and redraw the points on the canvas. The redraw functionality is extremely fast, so the user wouldn't notice this happening in the background.

The reason this hasn't been implemented is once again time. The implementation would likely take half a day, however the complexities of the functionality would take longer than this to work out in my opinion, as there are many things that need to be considered from a logic and UX point of view. For example, what if the user wishes to remove a single plot that is saved within the DicomObject. If a point is saved within a DicomObject, then it would have been added to the output CSV file. Alongside removing this from the object, we would then also have to delete the faulty annotation from the output CSV file. This is crucial functionality however and if I had more time would be the highest priority feature that I would attempt to implement.

7.5 – Annotation Conversation

While outlining the additional features that could be implemented in this project in section 4.2, I discussed the possibility of implementing a feature that upon synchronisation of two stacks would allow for conversion of annotations between the two. This feature would allow a user to see annotations placed in one slice show up in another. I attempted to implement this feature, however due its complexities and the remaining time I had left to implement, I did not manage to get it into the finished product.

I Believe this feature would be extremely useful to a user. The synchronise stack functionality is a crucial bit of additional functionality, however coupled with this, I believe the application would be an extremely powerful product. The way in which this would be implemented is using an affine transformation by where we convert two dimensional coordinates into 3 dimensional, in the scope of this project, patient-based coordinate system coordinates. We could then invert these coordinates based on a comparison slice in a separate stack, and project the 2D coordinates of an annotation in a specific slice into the 2D coordinates of the comparison slice.

Not getting this feature in was one of the biggest failings in the entire project, however I believe if someone where to extend this project, it would be a simple enough implementation given enough time. This feature also does not affect way in which the application meets the initial requirements.

7.6 – Integration with Machine Learning

Machine learning algorithms are impacting and progressing many areas of medicine. Currently, applications that are aimed at cancer diagnosis are being considered and developed by researchers world-wide in an attempt to speed up the process of not only

identifying cancerous areas but diagnosing at what stage the cancer currently is. This application was developed with this in mind. The integration with a machine learning algorithm would be, in my opinion, an extremely useful and impactful feature.

The way in which such a feature could work would be to have part of the program that sends the stack of DICOMS for analysis. During this analysis, a machine learning algorithm would attempt to identify potentially dangerous or malignant cells in the patient scans and then annotate them accordingly. From here we could demonstrate this to the user in two different ways. Due to the functionality of the DicomObject class documented in section 5.2.3 of this report, the application has the ability to save annotations to a specific slice object for future use in the application. The machine learning algorithm could also add annotations to this application, so that the user could navigate through the stack seeing what areas in the slices the algorithm has annotated for review. This could also be implemented by making the algorithm produce an output CSV file, which would then be either displayed to the user or fed back into the application, from which the slices that have been annotated in the file would be displayed with the annotations on them. This would mean that the user would only have to view the slices that have been annotated instead of navigating through the entire stack.

I have not implemented this feature due to the complexities and skills required to do so. Currently, this functionality is out of the scope of this project and far beyond any implementation that I have ever attempted. This would have to be its own project with its own individual requirements and time schedule. However, I do believe that this could be easily integrated into the current application and would of course greatly aid a radiologist in identifying and delineating potentially cancerous regions within a scan.

8. Conclusion

Throughout the project there have been many stages of implementation. The final application produced is one that I believe to have met all intended requirements at the start of this project. These requirements, as documented in section 2.3 are –

1. To make it easy to display and view MRI scans.
2. To make it simple and efficient to navigate through a stack of DICOM's.
3. To make the annotation and identification of potential regions of interest far easier for radiologists.

This report has clearly defined how the application delivered has met these requirements to a good standard and has shown that the project delivered adequately solves the problem of delineating regions of interest in MRI scans effectively. The project is robust and has been designed in such a way that allows for very little user error. It has an intuitive design, with all buttons being labelled according to their purpose. The aesthetics of the user interface have been sacrificed for a more efficient and purposed build; however this efficiency is crucial to the usefulness of the application.

The DICOMS are displayed clearly in view, meeting requirement 1. The annotation features provided by the application allow the user to identify areas of interest with ease, upon which the annotations will be saved to an external CSV file enabling analysis outside of the application. The annotations are clearly displayed on screen and also contain vital information about the patient and the score on the PI-RADS scale that the radiologist believes the annotated area to be. The navigation serves its intended purpose well and allows the user to navigate through a stack of DICOMs with ease. In my opinion, the final state of the codebase is in a good, well documented condition. The code runs efficiently, a major requirement of the application.

In terms of additional functionality, the application has some extremely useful and powerful features that were not expressed in the initial requirements. Firstly, the dual screen functionality of the application allows users to view to separate, or identical, stacks side by side. This allows them to more efficiently carry out their job, which is in my opinion, the main purpose of creating this application. Another useful piece of additional functionality is the ability to synchronise stacks of DICOMS in terms of their relation within the patient space is one that I believe to have greatly enhanced the usefulness of the application. This will allow radiologists to identify an area in one scan sequence, then compare it in the same region of another scan sequence. This feature will enhance their ability to delineate areas of interest, therefore is an additional feature that solely increases the effectiveness of the application in tackling the initial problem.

I do however believe that I could have done more in terms of additional features. Far too much time was spent refactoring old functionality, however in certain instances this was needed. In the middle stage of development, the application was at a stage where features tackling the main requirements were nearly fully implemented, however the overall flow and efficiency of the application was very poor. Also, as highlighted in my future work section (section 6), there are a few things that could have been in the app to improve the way in which it tackled the initial requirements, such as the in-application folder browser and the single point delete feature (section 6.5).

To conclude, I believe that the application effectively meets the requirements set out at the start of this project by clearly displaying the DICOM images in view, allowing the user to efficiently navigate through a stack of DICOM slices and annotate and delineate areas of interest within said slices. The use of additional features such as the dual screen functionality and the ability to synchronise scans aids the effectiveness of the application in tackling the problem. There are issues with the application such as the way in which a user removes plots, but the overall state of the application is one that allows for very little user error. All in all, it is a robust system that effectively aids a radiologist in delineating areas of interest in MRI scans and furthering the effectiveness in which they can diagnose prostate cancer.

9. Reflection

Throughout this project I have set personal goals on which to reflect upon come the projects conclusion. In sections 1.3 and 1.6, I described aims and outcomes that relate to my personal goals for the project. In this section, I will reflect upon my performance with regards to implementation of the application and personal skills developed throughout this project.

9.1 – Implementation

From a personal point of view, I believe that I could have implemented the application in a more efficient way. In my opinion, I spent a lot of time implementing in an in-efficient way which impacted the way in which I worked towards the end of the project. When I began implementation, I was nervous about my abilities to effectively develop the application and spent a lot of time worrying when I could have been researching and implementing. When I had gotten over this stage, I found that I thoroughly enjoyed developing the application. I also spent a lot of time developing with bad coding practices in early stages of the application. This led to me spending a lot of time refactoring that could have been spent developing additional features. In my opinion, the way in which I developed the application could have been better but was a great learning process as it will likely affect the way I approach future projects. Its also taught me to be more confident in my abilities and instead of worry about a problem, to spend the time tackling it instead.

I have learnt a lot about the problem area and the technologies involved in solving it on this project. I have acquired experience in a new programming language, how to effectively design a program from the ground up and how not to, experience with Medical imaging and image manipulation in general and gained confidence with my abilities as a developer. However, I know that coupling the knowledge I now possess with the technologies used in this application with more time to implement features, this application could be extremely powerful and up to industry standard solutions that currently exist.

9.2 – Project Management

At the beginning of this project, I had no knowledge or experience with project management at all. Throughout this project, I have had to demonstrate and exercise project management techniques in order to effectively develop the application. A good example of this is the use of Trello. I used Trello, a ticketing website, to track what aspects of the application I had implemented and those that I had not. To do this, I set up specific boards for front-end, user interface tasks such as designing wireframes and a board for back-end tasks such as developing the algorithm for slice synchronisation. This tool helped me stay on track with regards to implementation and follow a structured guide, in turn helping me develop my project management skills.

I also use GIT, an industry standard version control tool in order to keep separate implementations organised. Each major implementation featured in its own unique branch, again a massive part of project management. These two features greatly improved my knowledge in project management and have had a positive impact on the overall product delivered in this project.

9.3 – Communication

Finally, this project has helped me develop my communication skills with regards to technical conversations. Throughout the project I had regular meetings with my supervisor in order to explain what new features I had implemented, what problems I was encounter and to absorb information they had with regards to the subject area. This has helped develop the way in which I approach and understand technical conversations and will help me a lot in the computing industry. I also found that trying to explain to others that were unfamiliar with both the subject area and computing in general extremely useful as it reinforced my own understanding on the topic and this project.

References

1. https://prostatecanceruk.org/prostate-information/about-prostate-cancer
2. http://casemed.case.edu/clerkships/neurology/web%20neurorad/mri%20basics.htm
3. https://www.dicomlibrary.com/dicom/
4. https://pydicom.github.io/
5. https://matplotlib.org/
6. https://wiki.python.org/moin/TkInter
7. https://git-scm.com/
8. https://wiki.cancerimagingarchive.net/display/Public/PROSTATE-DIAGNOSIS
9. https://radiopaedia.org/articles/prostate-imaging-reporting-and-data-system-1?lang=gb
10. https://pydicom.github.io/pydicom/stable/viewing_images.html
11. https://github.com/pydicom/contrib-pydicom/blob/master/viewers/pydicom_Tkinter.py
12. https://www.numpy.org/
13. https://matplotlib.org/api/pyplot_api.html
14. https://dicom.innolitics.com/ciods/mr-image/image-plane/00200032
15. http://www.colourblindawareness.org/colour-blindness/
16. https://matplotlib.org/api/_as_gen/matplotlib.patches.Circle.html
17. https://pencil.evolus.vn/
18. https://matplotlib.org/gallery/user_interfaces/embedding_in_tk_sgskip.html -
19. http://dicom.nema.org/medical/dicom/2016e/output/chtml/part03/sect_C.7.4.html