**Ultrasound QA code explanation sheet**

Object-Oriented Programming (OOP):

The code for this project has been written in an object-oriented style, meaning that the DICOM images themselves are treated as objects and have attributes assigned to them (as seen in DICOMimages.py). This allows this code to be reused for other projects in future.

You can read more about OOP here: <https://realpython.com/python3-object-oriented-programming/>

Parts of the code:

1. Attributes of the DICOM image are set here. These self.\_\_\_\_ lines mainly search the DICOM tags for useful information (e.g. Date, Scanner Model etc). There is also a self.region attribute which is essential for the refactoring of the curved image pixel data.
2. The analyse() method takes in a cropped pixel array and outputs the coefficient of variation(COV), skew, and low values for the image (as seen in the paper: <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5676531/>). These are outputted as 7 values (as the low values are based on different locations on the image).
3. The linearDICOMimage class has special methods that crop linear DICOM images down to size. These differ from the curved DICOM image methods due to the refactoring process. The crop\_bottom() method can be found in the DICOMimage class as this is used for both linear and curved images and is inherited by both the curvedDICOMimage and linearDICOMimage classes (see OOP link for more info).
4. The curvedDICOMimage class has some key differences to the DICOMimage class. This is because a curved image needs to be linearised before it can be analysed. This is done by:
5. Finding the centre of the circle of which the curved lines on the image rotate around. self.centre is an attribute for this (see diagram) (REF:4.1.1 in code comment)
6. Finding the two points (labelled point 1 and point 2 on the diagram) (REF:4.1.2)
7. Finding the maximum and minimum radius of the circle. (REF:4.1.3)
8. Finding the maximum and minimum angle that the image sweeps between. (REF:4.1.4)
9. Finding the arc length of the curved image (REF:4.1.5)
10. Dividing the sweep angle by the arc length to find the angle increment when sweeping. (REF:4.1.6)
11. Taking the max radius – min radius and multiplying by a stretch constant to find the scan radius. (REF:4.1.7)
12. Creating a blank numpy array with a y-range of the scan radius and an x-range of the arc length. (REF:4.1.8)
13. Incrementing over the scan radius in polar coordinates and converting these to cartesian coordinates. (REF:4.1.9)
14. Interpolating these points using the nearest neighbour function. (REF:4.2.0)
15. Inputting the pixel value into the blank numpy array. (REF:4.2.1)
16. A close up of a map

    Description automatically generatedRepeat steps x-xii for each radial point and then iterate over the scan angle. (REF:4.2.2)
17. Analyse function creates a DICOMimage object, reads if the image is linear or curved, deletes the object and makes a new one with either the linearDICOMimage class or the curvedDICOMimage class depending on the image type. The image is then cropped and analysed using the image.analyse() method and outputted to a csv file which can be viewed in Excel or used in other ways.

Cropping Methods:

Used in crop\_bottom() method:

Condense(List, factor):

Condense takes in a list of pixel values and a condensing factor. It looks through the list and every (factor) pixels it takes a mean value and adds this to a new list (e.g. if the factor = 5, every 5 pixel values, it will take the mean of those 5 pixel values (i.e. a 400 pixel list would be condensed to an 80 pixel list)). The outputs of this function are two lists: One list for the mean of each (factor) pixels and one for the standard deviation of each (factor) pixels. This helps in the crop\_bottom() method.

middle\_values(image, width):

Middle values takes in the image pixel data and a width parameter (set in crop\_bottom() method). This function looks at an area of pixels in the middle of the image (width) pixels wide and the height of the image deep. It runs horizontally along the pixels from (middle -width/2) to (middle + width/2) adding each vertical pixel to a list called values. This values list is created for each horizontal iteration from (middle -width/2) to (middle + width/2) and then the mean of each vertical pixel is taken (i.e. pixel (5, 1) (where pixels are referenced (x, y)) is added to pixel (6, 1) and so on and then divided by the width. This is then outputted as a single list of mean pixel values.

cutoff\_index(standard\_dev\_list, mean\_list, factor):

Cutoff index takes two lists (one for standard deviation and one for mean pixel values (both come from the condense() function in crop\_bottom()). This function looks through the standard deviation list and detects the index of the element that has a std of less than 0.015 (as long as the mean pixel value is < 230 (if this was >230 there is a low chance that this is noise)). The index of the element with a std of < 0.015 is noted and multiplied by the factor (reversing the condense function), this allows the index of where the image turns from reverb to noise to on the ORIGINAL pixel data (not condensed) to be returned. NOTE: It is not wise to hard code in values like I have (the 0.015 and 230), it would be better to have these as variables that are inputted.

crop\_bottom():

Crop bottom uses all the functions listed above to crop the bottom of the image (leaving just the reverberation image). It first uses middle\_values() to get a list of mean central values; then a factor constant is hardcoded in (make this a variable), the height of the image (or length of the mean central values outputted by middle\_values()) is then divided by this factor constant to get the factor used in condense() and cutoff index(). Once the cutoff index is found, the image is then cropped vertically to this index.

Used in main\_crop() method:

main\_crop():

This uses the image region points from the DICOM header to crop the image and remove the information around the sides of the image.

Used in crop\_sides() method:

nonzero\_threshold(pixels, threshold):

This function takes in a list of pixels and a threshold value. The list of pixels is normally a column of pixels. The function then adds 1 to a variable called nonzeros every time it counts a pixel in the list that is not zero. It then divides nonzeros by the length of the list (total pixels) and if this value is above the threshold set, it returns True, otherwise it returns False.

crop\_sides():

This method looks from the centre of the image outwards horizontally. For each horizontal iteration it then adds each vertical pixel to a list called values. This list is then passed through the nonzero\_threshold() function with a threshold of 0.05 and if the result is False, the index is noted and the image is cropped to the centre +/- that index (as the image is symmetrical around the centre point).

alt\_crop\_sides():

This method is used to crop the sides of the image if there are no region attributes (no DICOM tags). This finds the centre of the pixel data (not necessarily the centre of the image) and does the same thing as crop\_sides() using nonzero\_threshold() in the same way. The only difference is that, as the centre of the pixel data isn’t necessarily the centre of the image, the image is not symmetrical around the centre of the pixel data so the program needs to repeat the process in the other direction to find the crop index on the other side of the image.

alternative\_crop():

Like with alt\_crop\_sides(), this method is only used for linear images without the required DICOM tags. Firstly, the top 8% of the image is cropped off as it can cause problems later when the largest nonzero pixel area is found (the 8% figure was found through trial and error and you may want to make this a variable instead of hardcoding it). The image is then converted to a black and white image where the pixel data is changed from pixel values to either “True” or “False” statements depending on whether the pixel values were zero or non zero. Two variables are then generated: labels and properties. Labels finds all of the connected regions of nonzero pixels in the pixel data and properties gives the properties of these regions (size, location etc.). There is then a loop that finds the biggest of these nonzero pixel areas and gives the size and the location. An imported function called bbox then gives the bounding coordinates of this area, allowing you to crop the image to these. There is then an IF statement which reads:

IF miny > int(shape[0] / 6) ….

This statement means that if the minimum y value of the image boundary (the top of the new cropped image) is more than 1/6th of the way down the image, then more code needs to be executed. This is because the top of the cropped image should not be more than 1/6th of the way down the image and this bug is often caused by a large patch of noise at the bottom of the image being larger than the reverberation image and therefore being chosen to be cropped around. If this statement is true, the image is cropped so that the miny value is instead used as the max y value, cutting off the noise and leaving a reverberation image. The bbox process is then repeated on this cropped image to leave a perfectly cropped reverb image.

Information:

There are no special methods for cropping curved images as the sides are cropped in the refactoring process and the bottom is cropped buy crop\_bottom() which is used for both linear and curved images and so is in the main DICOMimage class.