Information Processing for Medical Imaging
MPHY0025 - 2020/2021

Registration exercises 1
MATLAB version

You have been provided with some utility functions for use in these exercises:

dispImage.m

defFieldFromAffineMatrix.m

resampImageWithDefField.m

resampImageWithDefFieldPushInterp.m

as well as a template script and function for your solutions:

templateScript.m

affineMatrixForRotationAboutPoint.m

You have also been provided with a 2D lung MRI image:

lung_MRI_slice.png

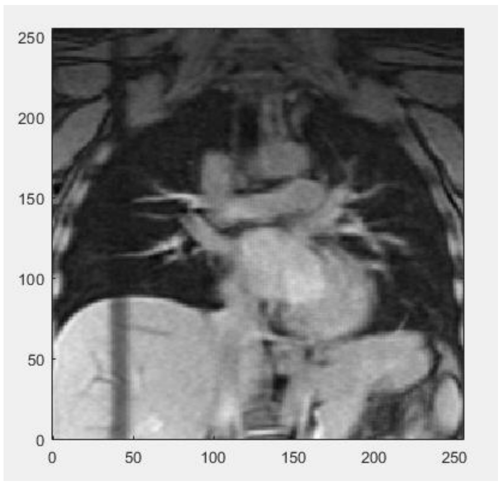
Loading and displaying images

Load the 2D lung MRI image using MATLAB's `imread` function. If you check the data type of the image you will see it is stored as 8-bit integers. You should convert the image to double precision so that errors do not occur when processing the image due to the limited precision. The template script already includes the code to load the image and convert it to double precision.

As discussed in the lecture, MATLAB indexes images using matrix coordinates. The first cooridnate is the row number (i.e. the y coordinate of the image) and the second coordinate is the column number (i.e. the x coordinate of the image). The rows are also numbered from top to bottom.

It is possible to do image processing using matrix coordinates, but I find this can get confusing, especially when working with deformation fields or in 3D. Therefore, the approach I use, and the one you should use for these exercises, is to store the image in memory in 'standard orientation', so that the first coordinate indexes the x (horizontal) dimension and the second coordinate indexes the y (vertical) dimension, and the first pixel (1,1) is at the bottom-left of the image. Then, when displaying the image a wrapper function such as the `dispImage` function can be used to reorientate the image into matrix coordinates as required by MATLAB's functions for displaying images, such as the `imagesc` function used here.

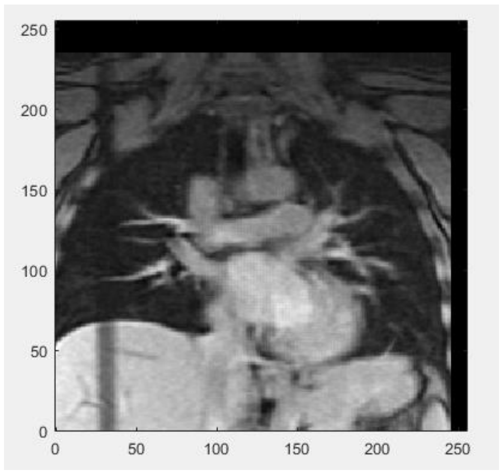
Therefore, before proceeding you should reorientate the image into 'standard orientation'. This can be done by first taking the transpose of the matrix (switching x and y dimensions) and then flipping along the second dimension (moving the first pixel from the top to the bottom of the image). Add code to the template script to do this. Now display the image using the `dispImage` function. If you have reorientated the image correctly it should appear like this:



Translating and resampling images

Create an affine matrix representing a translation by 10 pixels in the x direction and 20 pixels in the y direction. The template script provides code that creates an identity matrix and can be edited to create the desired matrix.

Add/edit code in the template script to use the provided `defFieldFromAffineMatrix` function to create a deformation field from the affine matrix, and resample the image with the deformation field using the provided `resampImageWithDefField` function. The functions include comments explaining what the inputs and outputs of the functions should be. Now display the transformed image. If you have done this correctly it should appear like this:



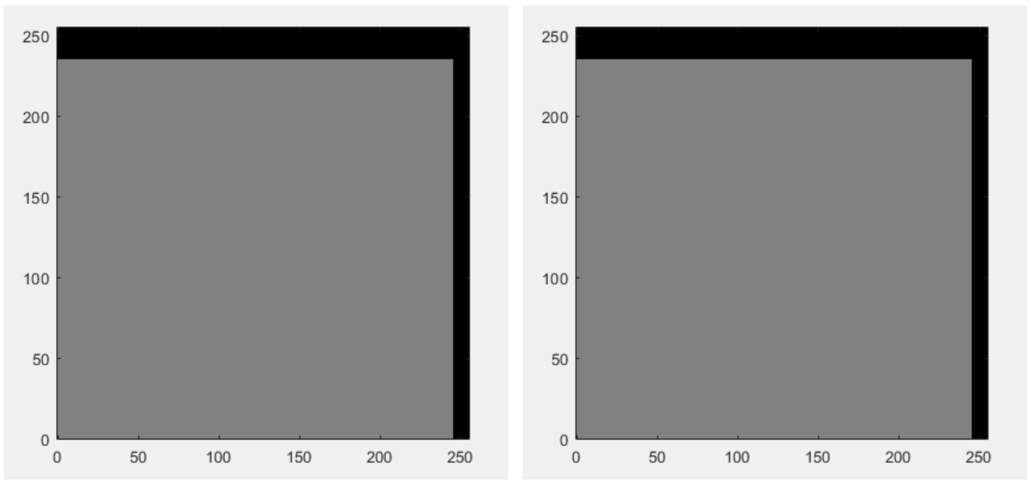
Does this appear as expected? Has the image been translated in the direction you expected? The `resampImageWithDefField` function uses pull-interpolation, so the image will appear to have been transformed by the inverse of the transformation in the affine matrix.

Check what value has been assigned to pixels that were originally outside the image by printing the value of the top right pixel (255,255). This is known as the ‘padding value’ or ‘extrapolation value’. A value of NaN (not a number) is often used to indicate that the true value for these pixels is unknown, and therefore they should be ignored when calculating similarity measures during image registration.

The `resampImageWithDefField` function uses *linear* interpolation by default. It is implemented using MATLAB’s `interpn` function, so can also use the other interpolation methods available for this function, such as *nearest neighbour* or *cubic* interpolation. Add code to the template script to resample the image using *nearest neighbour* and *cubic* interpolations, and display the results in separate figures.

Do the different interpolation methods give different results?

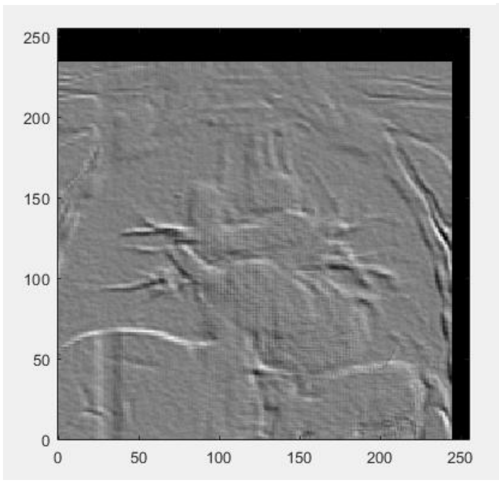
This will be easier to see if you display difference images, i.e. one image minus another, between the original result with *linear* interpolation and the results using *nearest neighbour* and *cubic* interpolations. Add code to the template script to display the difference images. They should appear like this:

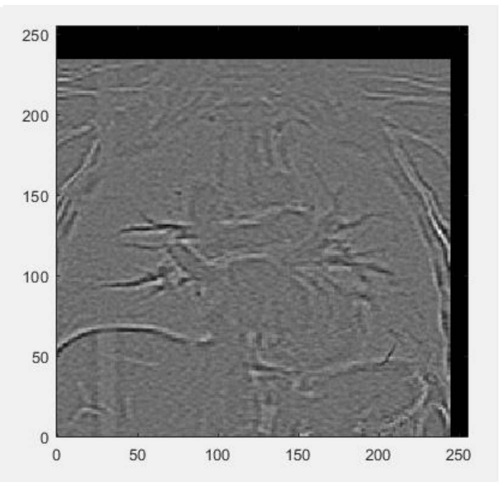


(the difference image for *nearest neighbour* is on the left, and *cubic* on the right)

As you can see, the difference images are all 0, and the results are exactly the same for all 3 interpolation methods. Do you understand why this is the case?

Now add code to the template script that repeats the steps above using a translation of 10.5 pixels in the x direction and 20.5 pixels in the y direction. The difference images should look like this:

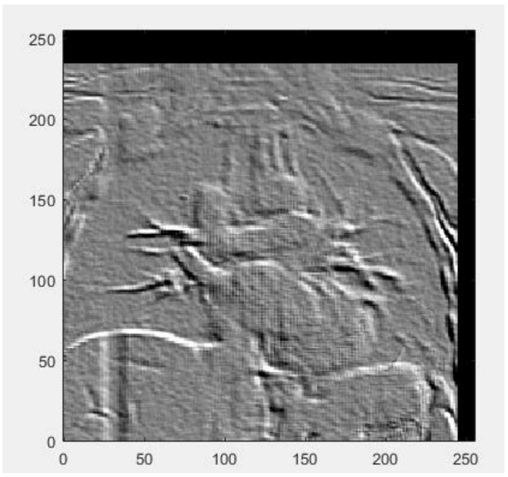


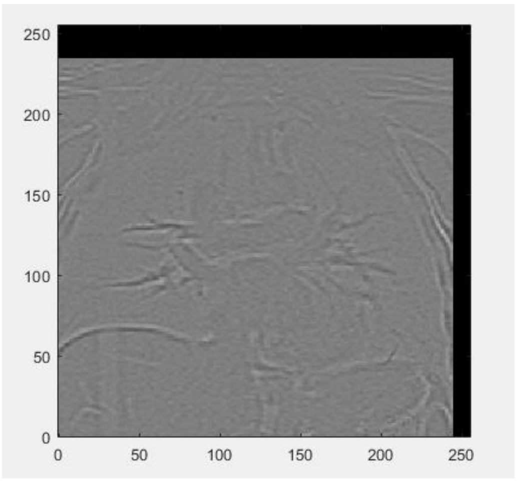


(the difference image for *nearest neighbour* is on the left, and *cubic* on the right)

Make sure you understand why you get these results. If you are not sure check back over the lecture notes. If you are still not sure ask the lecturer/PGTAs in the lab session.

By default, the `dispImage` function displays images by scaling the values so that they use the full intensity range, i.e. the lowest value in the image is set to black and the highest to white. However, this can lead to unintentional differences in appearance when comparing images. Therefore, it is often a good idea to ensure exactly the same intensity range is used when displaying and comparing different images by using the `int_lims` input to the `dispImage` function. For example, if you check the intensity ranges of the difference images above you will notice that the range of values is larger for the nearest neighbour difference image. Add code to redisplay the difference images, in both cases using an intensity limits of [-20, 20]. The difference images should now appear like this:



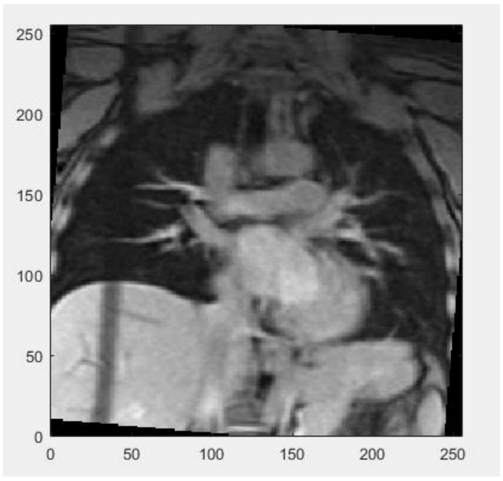


(the difference image for *nearest neighbour* is on the left, and *splinef2d* on the right)

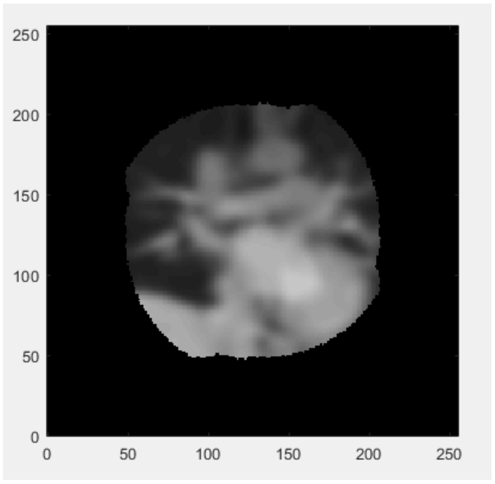
Rotating images and composing transformations

Add code to the template function `affineMatrixForRotationAboutPoint` that calculates the affine matrix corresponding to a rotation about a point, P. The inputs to the function should be the angle of rotation (in degrees) and the coordinates of the point and the output should be the affine matrix.

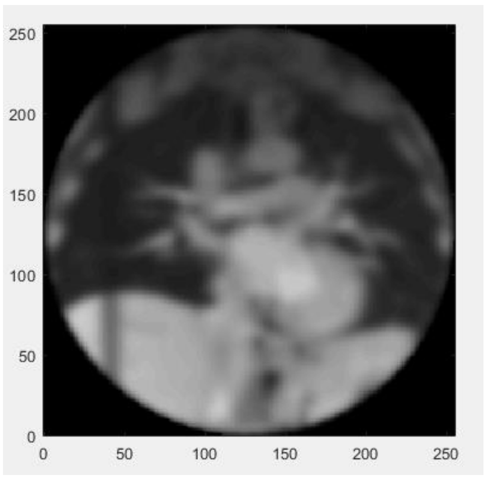
Now add code to the template script that uses the above function to calculate the affine matrix representing an anticlockwise rotation of 5 degrees about the centre of the image. Note – the image has an even number of pixels in each dimension, so the centre of the image will not be the centre of a pixel. The width and height of the image referred to in the lecture slides are the width and height from the centre of the first pixel to the centre of the last pixel, i.e. width = number of pixels in x – 1, height = number of pixels in y – 1. Add code to transform the original image using the rotation you just created and display the result. Linear interpolation should be used when resampling the image, and the intensity limits from the original image when displaying the results. The result should look like this:



Add/edit the code in the template script to apply the same transformation again to the resampled image and display the result. This should be repeated 71 times, so that the image appears to rotate a full 360 degrees. So that the figure is updated each time the image is displayed a short pause (0.05 seconds) is added using `plt.pause(0.05)` after displaying the image. The final image displayed should look like this:

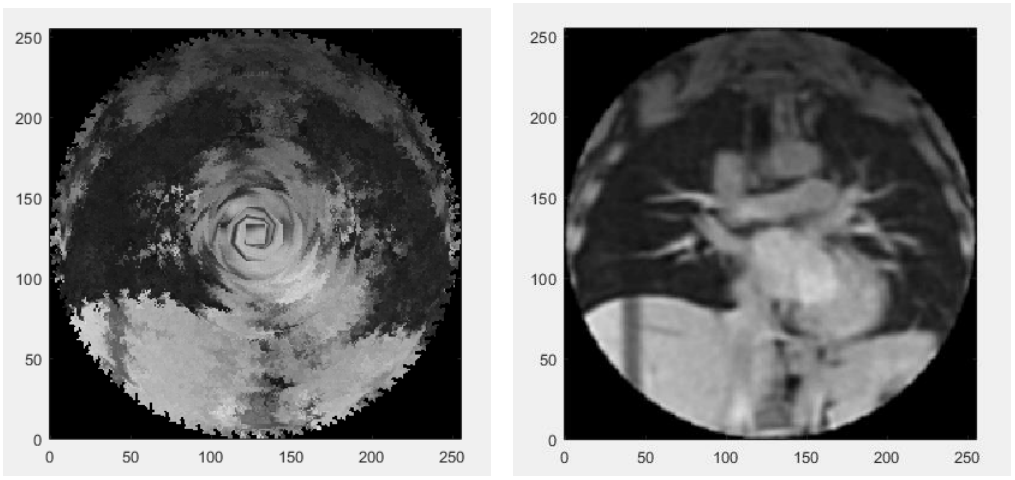


You will notice that the image gets smaller and smaller as it rotates. This is because of the NaN padding values – when a pixel value is interpolated from one or more NaN values it also gets set to NaN, so the pixels at the edge of the image keep getting set to NaN, and the image gets smaller after each rotation. To prevent this, edit your code so that it uses a padding value of 0 rather than NaN and rerun your code. This time the final image should look like this:



You will notice that the corners of the image still get 'rounded off' as it rotates so that it has become a circle after rotating 90 degrees. Do you understand why this happens?

Now add code to the template script to repeat the above, but first using *nearest neighbour* interpolation, and then using *cubic*. The final images should look like this:



(the final image for *nearest neighbour* is on the left, and *cubic* on the right)

Now edit your code to experiment with using different angles (both smaller and larger) and rotating about a different point. Make sure you understand all the results you get.

The blurring artefacts and the ‘rounding off’ of the images seen above are caused by multiple re-samplings of the image. This can be prevented by composing the rotations and applying the resulting transformation to the original image instead of the transformed image. Add/edit the template script to create animations of the rotating image as above, but at each step the initial rotation is composed with the current rotation, and the current rotation is used to resample the original image. Try this using *nearest neighbour*, *linear*, and *cubic* interpolation. You should notice that the corners of the images do not get ‘rounded off’ during these animations. Furthermore, you should notice that the intermediate images are different for the different interpolation methods (although this is only really noticeable for nearest neighbour interpolation), but the blurring artefacts do not get worse as the animation progresses, and the final image should be the same as the original image.

As discussed in the lectures, it is possible to resample an image using push-interpolation, but it is far less computationally efficient than using pull-interpolation. Copy your code from above and modify it to perform push interpolation instead of pull interpolation by using the `resampImageWithDefFieldPushInterp` function instead of the `resampImageWithDefField` function. Other than the animation being much slower, what other difference do you notice?