Information Processing for Medical Imaging
MPHY0025 - 2019/2020

Registration workshop 1, 16th January 2020
MATLAB version

Load the 2D lung MRI image stored in lung_MRI_slice.png using MATLAB's `imread` function. The image is stored as unsigned 8 bit integers – convert it to type double so that errors do not occur when processing the image due to the limited precision.

Note to MATLAB users – MATLAB has 'it's own special way' of orientating images when loading and displaying them. This is because MATLAB was originally developed to work with matrices (that's where the 'MAT' in MATLAB comes from), so it uses matrix coordinates where the first coordinate is the row number (y coordinate of the image) and the second coordinate is the column number (x coordinate of the image), and the rows (y coordinates of the image) are numbered from top-to-bottom.

While it is possible to do image processing in 'MATLAB orientation', it can easily get confusing, so my advice, and the approach you should use for these exercises, is to store the images in memory in 'standard orientation', i.e. with the first coordinate corresponding to the x (horizontal) dimension, and the second coordinate corresponding to the y (vertical) dimension, and with the first pixel at the bottom-left of the image. Then, when displaying the image, it should be manipulated so as to display it correctly – the provided `dispImage` function demonstrates how to do this for 2D images (and also displays the image in grey scale).

When the image is read in using MATLAB's `imread` function it is initially stored in 'MATLAB orientation' – therefore, before proceeding it is necessary to re-orientate it 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), i.e.:

```
I = flip(I',2);
```

Display the image using the provided `dispImage` function. Use the inter-active plot controls to zoom in on the image and note that the origin (0,0) is set as the bottom-left pixel. For these exercises the origin will always be set as the bottom left pixel.

Create an affine matrix representing a translation by 10 pixels in the x direction and 20 pixels in the y direction. Create a deformation field for the transformation using the provided `defFieldFromAffineMatrix` function, and then resample the image with the deformation field using the provided `resampImageWithDefField` function.

Display the transformed image. Does it appear as expected? Note, 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. 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, but can also use nearest neighbour or cubic (or any other interpolation method supported by MATLAB's `interpn` function). Resample the image again using nearest neighbour interpolation and cubic interpolation and display the resulting images.

Do the different interpolation methods give different results? It may be useful to use difference images (one image minus the other image) to assess this.

What about if you now use a translation of 10.5 pixels in the x direction, and 20.5 pixels in the y direction?

Make sure you understand why you get the results that you do (if you are not sure ask one of the lab assistants).

Note – the transformed images may not have exactly the same intensity ranges as the original images due to interpolation and pixels moving outside of the image. This can cause unintentional differences in appearance if the images are displayed using their full intensity ranges (which is the default behaviour with the `dispImage` function). Therefore, it is often a good idea to ensure exactly the same intensity range is used when displaying and comparing different images (e.g. the intensity range of the original image). This can be done using the `int_lims` input to the `dispImage` function.

Write a function that will calculate 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, i.e.:

```
function aff_mat = affineMatrixForRotationAboutPoint(theta, p_coords)
%function to calculate the affine matrix corresponding to an anticlockwise
%rotation about a point
%
%INPUTS:    theta: the angle of the rotation, specified in degrees
%           p_coords: the 2D coordinates of the point that is the centre of
%               rotation. p_coords(1) is the x coordinate, p_coords(2) is
%               the y coordinate
%
%OUTPUTS:   aff_mat: a 3 x 3 affine matrix
```

Use the above function to calculate the affine matrix for 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.

Create a deformation field from the affine matrix above and use it to resample the image using linear interpolation. Display the resulting transformed image. Now apply the same transformation to the transformed image and display the result. Repeat this 71 times so that the image will appear to rotate a full 360 degrees.

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 replace the NaN values in the transformed image with 0s before applying the next rotation (this is effectively using a padding value of 0 rather than NaN).

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 repeat the above using nearest neighbour and cubic interpolation and compare the final results to the result obtained using linear interpolation.

Now 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. Use this approach to create animations of the rotating image as above, but which do not suffer from blurring artefacts or 'rounding off' of the images. Try this using nearest neighbour, linear, and cubic interpolation.

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.

Create an affine matrix representing a rotation by -30 degrees about the point 150,150, and use this to calculate the corresponding deformation field. Use the `resampImageWithDefFieldPushInterp` function to create the transformed image using push-interpolation. Is the result as expected?
Now use pull-interpolation (i.e. the `resampImageWithDefField` function) to create the same result. Compare the results – you should notice that they appear very similar, but if you display a difference image between the results you will see some small differences. However, the main difference is in the computation time. Re-create the rotating image animation above but using push-interpolation and you will notice the difference!