Image processing in Python: assignment

This assignment is the second part of the assessed assignments that you need to complete. In this assignment we will be modifying the code you wrote yesterday to perform automatic image registration. Everything you need should be in the repository.

Part 1

I. Start a new script

II. Write some useful information like a title and your names in some comments at the top of the code.

III. Import the modules that you will need. You'll need `matplotlib.pyplot` and `numpy`, but also `imread` from `skimage.io`, and `interpolation` and `rotate` from `scipy.ndimage`. You will also need some functions in `scipy.optimize`, namely `brute` and `differential_evolution`. In this part of the practical, we will also use DICOM images for the first time, so you will need to import `pydicom`.

IV. Load the images "IMG-0004-00001.dcm" and "IMG-0004-00002.dcm". get the pixel data from the dicom file as a numpy array using `img1_array = img1_dcm.pixel_array`

V. Display the images using transparency so that you can see they are not registered. Save this as a png with a sensible name. Add it to your repo

VI. Yesterday, you wrote a function called shiftImage, that took a shift as a list and applied it to a global copy of an image. Copy-paste that function into your new script. Now we will modify it in the following ways:

- Add another argument, the image to which we will apply the shifts
- Remove any plot-updating code
- Make the function return the image after it has been shifted

The function should look something like this:

```
def shiftImage(shifts, image):
    shifted_image = interpolation.shift(…)
    return shifted_image
```

Evaluate your function by trial and error guessing the required shift that registers the first two images.

VII. Recall from your lecture this morning that automated image registration requires a cost function (aka metric). Pick one from that lecture (mean squared error is a good start) and implement it. The function should take two arguments (the fixed and floating image arrays). Verify that this function does what you expect it to, i.e. zero cost when comparing an image to itself, positive, large cost when comparing two unregistered images.

VIII. We now have two of the key parts of the registration process, an interpolating function and a metric. The next bit we need is an optimizer – for this we will use `scipy.optimize`, but it will require a bit of work to link everything together.

Write a function called `registerImages`. This function accepts three arguments: `shift`, `fixed` and `floating`. The arguments must be in that order! Inside, we do the following things:

- Call the `shiftImages` function using `floating` and the `shift`, store the result as `shifted_image`
- Evaluate the cost function between `fixed` and `shifted_image`
- Return the cost value

By organising the function in this way, we make it something that the optimizers in `scipy.optimize` can work with. To automatically register your images, we can now do something like this:

```
registering_shift = brute(registerImages, ((-100, 100),(-100, 100)), args=(fixed, floating))
```

This will return a tuple, which is the shift the optimizer applied to minimize the cost function.

IX. See what the registration looks like. To do this, take the `registering_shift` and apply it to `floating` using your `shiftImage` function, then plot the result over `fixed` with transparency. Does it look ok? Make a note in a comment of what the shifts were.

X. There are many options for the optimizer, have a look at the documentation and try some things out. For example:
- How does changing the ranges over which the optimizer runs affect the quality of the registration?
- How does changing the number of evaluation points affect the registration? What is the downside of trying a lot of points?
- How does changing the coft function affect the registration?

XI. There are also other optimizers, for example `differential_evolution` which has largely the same signature as `brute`. Try some of the other optimizers. Are they better? Faster?

XII. Now you can register two images, we can move on to register all the dicom images to the first. To do this, you will need to load them all, and then put the pixel data arrays of the three floating images into a list, something like this:

```
floating_list = [img_2, img_3, img_4]
```

Now we can make a loop over the floating images, and register them to the fixed one, storing the registration result as we go, something like:

```
registrations = []
for floating in floating_list:
    registering_shift = brute(…)
    registrations.append(registering_shift)
```

After the registration is finished, make a new list of shifted images by looping over the floating list again and using the `shiftImage` function.

XIII. At this point, we will be doing some analysis on the registered images, and waiting for the registration to run every time will get old fast. Create a numpy array from your list of registering shifts (e.g. `registrations_arr = np.array(registrations)`) and save it as a numpy file: `np.save("registrations.npy", registrations_arr)`.

Part 2

XIV. Start a new script, called something like regression_analysis.py. Put the usual info at the top.

XV. In the new script, import everything you will need and copy the `shiftImages` function from the previous script. Load the registrations array (`registrations = np.load(registrations.npy")`) and the four images, apply the corresponding registration to each to get back to where you were at the end of the previous script.

XVI. Now you should be able to plot the four images side-by-side (i.e. on a set of subplots, not overlaid) and see that the tumour visible in them is "regressing". We will now attempt to measure that regression!

To do this, we will link up some code in the "`interface.py`" file with a figure you will create showing only the first image. Then, we will use the interface to put a red box over the tumour. The red box will be used to get the indices in the pixel data array that define that region of interest. From that we will extract the ROI and analyse the numbers within it

XVII. Display the first image in the timeseries, make sure you keep the figure handle and axes. (look at the code in `interface.py` to see what things you will need)

XVIII. Copy and paste the code from `interface.py` into your script. Make sure nothing gets messed up! You will have to modify the bits at the top & bottom that create/attach to figures to make them attach to the figure you just created

XIX. Now everything is linked up, try visualising the first image. You should also see a red box. The interface code allows you to click on and move the box, then use the arrow keys on the keyboard to make it bigger and smaller in each direction.

XX. Use the interface to select the region of intertest you want to analyse. The interface will give you a variable called indices, which are the first and second axis indices of the region of interest. Extract the region of interest like this:

```
roi = fixed[indices[0]:indices[1], indices[2]:indices[3]]
```

(NB: This may be transposed, see what you get)

Repeat this for each of the four images after registration.

XXI. Using the same configuration as before, visualise the regions of interest side by side.

XXII. Come up with a metric that will describe the way the tumour is disappearing inside this ROI. This can be as simple or as complicated as you like. Evaluate your metric on the four images and plot it. Make sure this plot:
- Has labelled axes
- Has a legend

Save this plot as a png and add it to your repo. Make some comments in your code noting what you see.

XXIII. If you still have time, change the ROI to something else, e.g. a bit of the spine, and verify that the metric you chose does not change over time in that ROI.