Mini-project 2018: Image processing

The purpose of this mini-project is to consolidate all that you have learned about MATLAB so far. After completing this project you should feel confident in tackling computational problems using MATLAB.

1 Marking

This project will **not** be counted towards your overall grade, but your performance will be taken into consideration when you are choosing projects in the third and fourth year. You may discuss your understanding of the problems and solutions with your peers but you must not share your solution code. Your marks will be based entirely on the runtime performance of your code. Although there are no marks for comments, you are advised to include them so that you can quickly recapture your thought processes at some distant future time e.g. at a job interview.

The performance of your code will be tested on a variety of images. The marks out of 100 are displayed in the header for each exercise.

2 Submission

Submission will be through Blackboard, the deadline is **Wednesday 20th June at 18:00**. Please submit a separate `.m` file for each function. Ensure that the names of the functions are the same as the names given in this document.

3 Overview

In this project you will be writing code to process medical images. Imagine you are part of a team that develops medical analysis software and you have been tasked to code some of the functions which should fit into a larger programme. You are required to write two functions:

1. one function to rotate an image and
2. another function to detect edges in an image,

3.1 Rotation [70]

Write a function named `rotate_image`, this function should take a matrix representing a grayscale image as input and should output a rotated version of the image as another matrix. In addition it should take another input which specifies the angle to rotate the image at, given in radians. Thus, the function should have the following signature:

```
function [output_image] = rotate_image(input_image, theta)
```

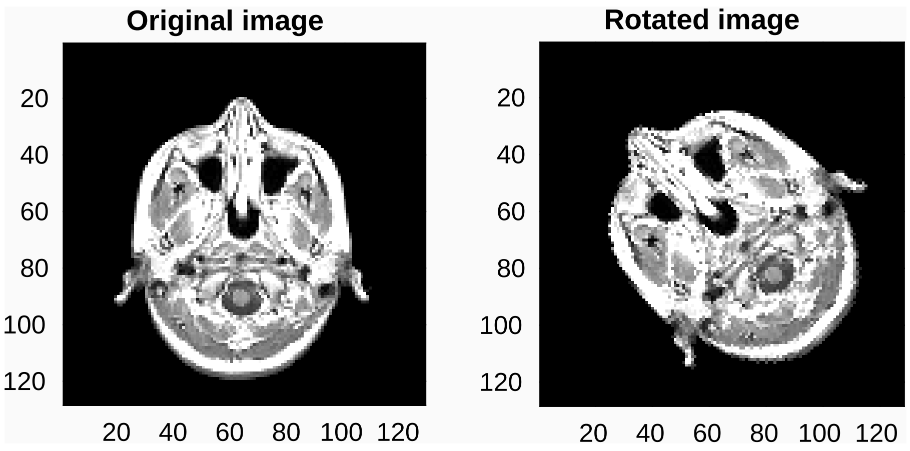
Image rotation can be achieved by performing what is called a reverse mapping operation. You take the rotated image and for each pixel of the rotated image, you determine where in the original image the pixel came from. Mathematically, this can be expressed as:

$$\begin{bmatrix} x_{output} \\ y_{output} \end{bmatrix} = \begin{bmatrix} cos(\theta) & sin(\theta) \\ -sin(\theta) & cos(\theta) \end{bmatrix}^{-1} * \left(\begin{bmatrix} x_{input} \\ y_{input} \end{bmatrix} - \begin{bmatrix} x_{centre} \\ y_{centre} \end{bmatrix} \right) + \begin{bmatrix} x_{centre} \\ y_{centre} \end{bmatrix}, \quad (3.1)$$

where (x_{input}, y_{input}) are the coordinates for a pixel in the input (un-rotated) image and (x_{output}, y_{output}) are the coordinates of the output (rotated) image. The extra terms involving the coordinates for the centre of the image (x_{centre}, y_{centre}) are included because we want the rotation to occur around the centre of the image and not the origin. The angle θ is the angle of the rotation given in radians.

The following shows the expected behaviour of the function given an image called `test_image`:

```matlab
clear all

% Load the image
load mri % Demo image within MATLAB
test_image = D(:,:,:,1);

% Rotate the image by pi/4 radians using the function
rotated_image = rotate_image(test_image, pi/4);

% Display rotated image and original image side-by-side
figure(1) % Create figure

subplot(1,2,1)
imagesc(test_image)
title('Original image')
axis square

subplot(1,2,2)
imagesc(rotated_image)
title('Rotated image')
axis square

colormap gray % Change the colourmap to gray
```



3.2 Edge detection [30]

Write a function named `detect_edges`, which takes a grayscale image as input and outputs the edges of that image as another image. The function signature should look like the following:

```
function [output_image] = detect_edges(input_image)
```

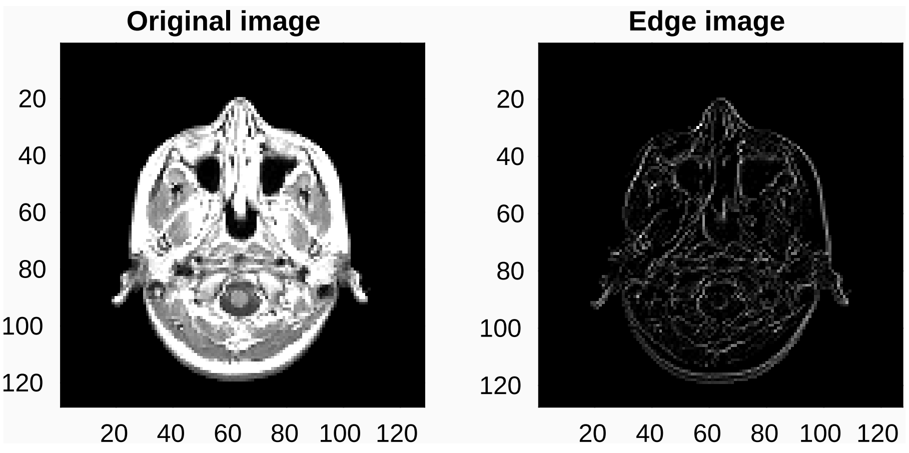
A simple way to perform edge detection is by using a two pass approach:

1. one pass through the image to look for horizontal edges and
2. one pass through the image to look for vertical edges.

The result of these two passes is then averaged to yield the final image. In other words:

$$Out_{horizontal}(x,y) = |In(x,y) - In(x+1,y)|$$
$$Out_{vertical}(x,y) = |In(x,y) - In(x,y+1)|$$
$$Out(x,y) = \frac{Out_{horizontal}(x,y) + Out_{vertical}(x,y)}{2}$$

where Out and In denote the output and input image respectively. $|x|$ denotes the absolute value function. The expected behaviour:



3.3 (Optional) Vectorisation

Try rewriting both `detect_edges` and `blur_image` to use vectorisation instead of loops. Investigate the difference in execution time between your original code and the vectorised code. Call these functions `detect_edges_vectorised` and `blur_image_vectorised`.