The University of Hong Kong
ENGG1111 Computer Programming and Applications

ENGG1111Shop: Background

This document contains additional background information and illustrations related to the image enhancement functions you will implement in Assignment 2. You don't need to study this material to complete the assignment, but it may serve as a reference if you are unsure about any of the tasks or if you are interested in how the techniques work to enhance images.

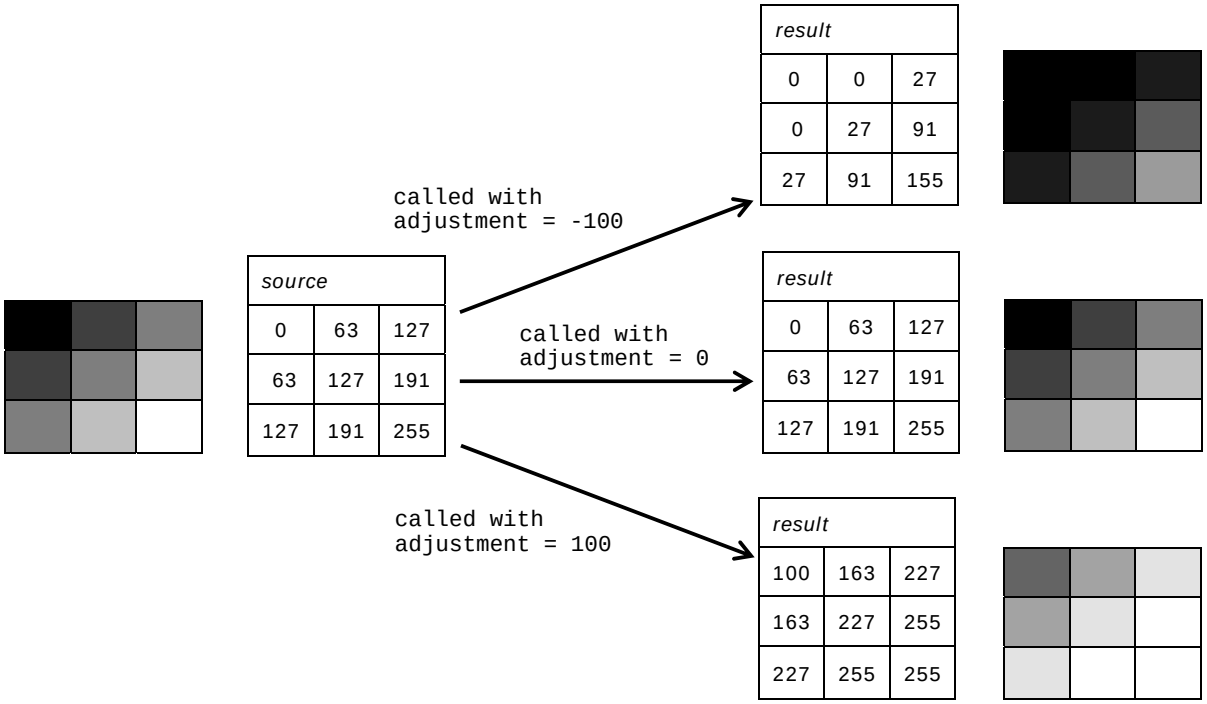
Single pixel operations

In single pixel operations, the value of a pixel in the *result* image depends on the value of a single pixel in the *source* image.

Adjusting Brightness (Task 2)

In general, as noted in the assignment sheet, increasing the value of a pixel will make it lighter, and decreasing its value will make it darker. The effect of calling the *adjust_brightness()* function will be as follows: if the adjustment value is negative, the *result* will represent a darkened copy of the *source*; if the adjustment is positive, the *result* will represent a lightened copy of the *source*; and if the adjustment is 0, the *result* will be identical to the *source*.

For example:



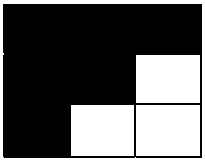
The first illustration, with an adjustment value of -100, corresponds to the example given in the assignment sheet.

Converting to monochrome (Task 3)

This illustration corresponds to the example given in the assignment sheet for *convert_to_bw()*:

source		
0	63	127
63	127	191
127	191	255

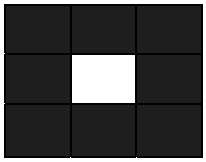
result		
0	0	0
0	0	255
0	255	255



Neighbourhood operations

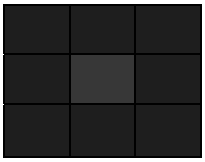
In contrast to the operations in Tasks 2 and 3, many common image manipulations use the values of several *source* pixels to generate the value of a pixel in the *result*. In a typical operation of this kind, the *result* pixel value is computed from the values of a neighbourhood of pixels centred on the equivalent pixel in the *source* image.

Consider a white pixel introduced by mistake into an otherwise dark area of a *source* image (this is an example of *noise* and could arise from, for example, a stuck or hot pixel in a digital camera's sensor. Another example of noise could be dark pixels appearing by error in a light area of an image). One way to reduce visible noise is to generate each value in a *result* image by averaging the values of source pixels *and* their neighbours. For example, for each pixel in the *source*, we can average its value with the values of its 8 neighbours to generate a pixel value for the equivalent position in the *result*. This is the 3x3 neighbourhood averaging approach used in Task 4.



source		
30	30	30
30	255	30
30	30	30

result (for central pixel only)		
30	30	30
30	55	30
30	30	30



In this example, the value of the central pixel in *result* is calculated as the average of the 9 pixels in the 3x3 *source* neighbourhood shown.

The pixel value in *result* = (sum of pixel value of central *source* pixel and its 8 neighbours) / 9

= (255 + 30 + 30 + 30 + 30 + 30 + 30 +30 +30) / 9
= 495 / 9
= 55

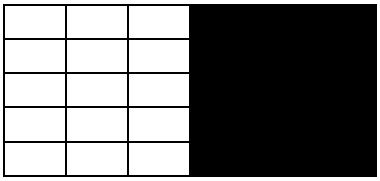
The effect in this example is to reduce the brightness of the noisy pixel so that it more closely matches its neighbours.

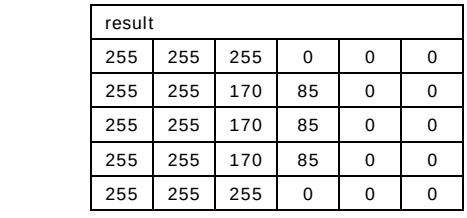
The final issue to consider when performing neighbourhood averaging is how to calculate values for pixels at the boundaries of the *result* image. For example, a corner pixel of the *source* has only 4 of the required 9 neighbourhood pixels and we can't calculate a 3x3 average. There are many ways we could

deal with this. One of the simplest, as described in Task 4, is to copy pixel values directly from *source* image to *result* image for locations where we can't calculate the required average. For a 3x3 neighbourhood, this means that a 1-pixel wide border of values from the *source* will be copied to the *result* unchanged. The values of all other pixels lying within that border can be calculated using neighbourhood-averaging.

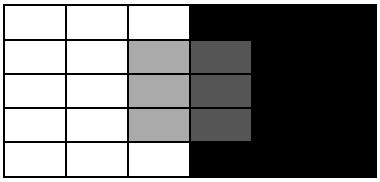
Example:

source					
255	255	255	0	0	0
255	255	255	0	0	0
255	255	255	0	0	0
255	255	255	0	0	0
255	255	255	0	0	0





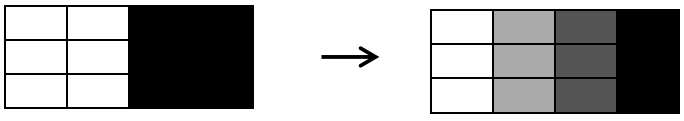
result					
255	255	255	0	0	0
255	255	170	85	0	0
255	255	170	85	0	0
255	255	170	85	0	0
255	255	255	0	0	0



Note that the values in the border of result have been copied, unchanged, from *source*, while all other values have been calculated by applying neighbourhood-averaging. Thus *result(0, 0)* is identical to *source(0, 0)*, while *result(2, 2)* = (255 + 255 + 0 + 255 + 255 + 0 + 255 + 255 + 0) / 9
= 1530 / 9
= 170.

Blurring an image (Task 4)

The above example also shows the effect of applying 3x3 neighbourhood-averaging to an *edge* in an image. An edge of a feature or object depicted in an image is indicated by a change in pixel value. In the example, there is a distinct edge between the white and the black regions of the *source*. In the *result* image, the edge is no longer so distinct – we pass from white to black through two additional shades of grey. The effect has been to blur the edge. This can be seen more clearly by considering only the pixels lying within the border:



So, in addition to reducing noise, neighbourhood-averaging provides a simple way to blur an image.

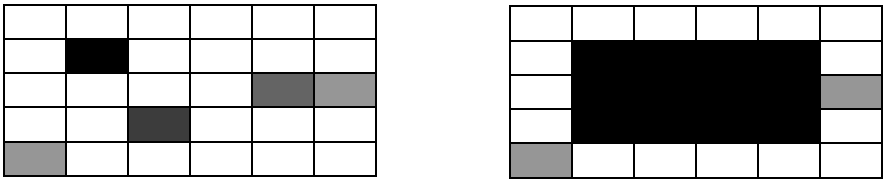
We decompose the blurring operation into two sub-operations, each implemented as a function. The first function, *copy_border()*, copies pixel values from the border of the *source* image, to the border of the *result* image. The second function, *average()*, applies 3x3 neighbourhood-averaging to calculate the values of each non-border pixel of *result*. The following are illustrations corresponding to the examples given in the assignment sheet for each function:

Copying the border (Task 4a)

In this example, the *source* image is white with some random noise in the form of dark pixels. Assuming the *copy_border()* function is called with a *result* image array initialized to PURE_BLACK, we obtain the following:

source					
255	255	255	255	255	255
255	0	255	255	255	255
255	255	255	255	100	150
255	255	60	255	255	255
150	255	255	255	255	255

result					
255	255	255	255	255	255
255	0	0	0	0	255
255	0	0	0	0	150
255	0	0	0	0	255
150	255	255	255	255	255

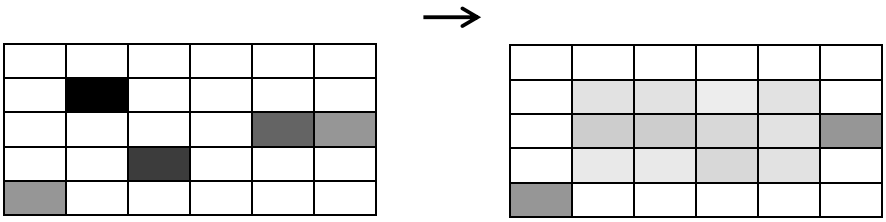


Applying 3x3 neighbourhood-averaging after copying the border (Task 4b)

The second step in blurring is to calculate the values of the non-border pixels in the result image:

source					
255	255	255	255	255	255
255	0	255	255	255	255
255	255	255	255	100	150
255	255	60	255	255	255
150	255	255	255	255	255

result					
255	255	255	255	255	255
255	226	226	237	226	255
255	205	205	216	226	150
255	221	233	216	226	255
150	255	255	255	255	255



You can see that the noisy pixels have been blurred. That is, they are less visible than before. The visual effect will be easier to appreciate when you apply your filter to real photographic samples supplied in the Appendix of the assignment sheet.

Sharpening an image (Task 5)

In Task 4 we blurred images by making the edges of features and objects indistinct. To sharpen an image we do the opposite and try to *emphasize* the edges of features and objects in the image. If we could obtain an image containing *only* the edges that are present in the *source*, then we could add it to the *source* to emphasize the edges in the image and, hence, sharpen the image. The problem then, is how to obtain an image containing only the pattern of edges.

The technique we'll use has a strange name: it is known as an *unsharp masking*. It solves the problem of obtaining the edges by using an unsharp (that is, a blurred) version of the *source*. The idea is simple. When we blur an image, we *de-emphasize* the edges in the source while leaving other parts of the image relatively unchanged. That is, the difference between an original and a blurred version is concentrated in the edges of features and objects in the image. Thus, if we subtract a blurred version of an image from the original, we should obtain an image containing only the edges. We can then combine this with the original to sharpen it. Usually, when applying this technique, we let the user control the proportion of the edge image added to the original by specifying the amount of sharpening to apply.

For example, let's start with an image *S*. Apply your `blur()` function to generate *B*, a blurred version of *S*.

Subtract *B* from *S* to obtain an image, *E*, of the edges present in *S*.

$$E = S - B$$

E is known as an unsharp mask. You will then add some percentage of *E* back to *S* to obtain a sharpened result, *R*. This percentage is the *amount* of sharpening specified by the user. Typical values are between 0% and 200%, with up to 500% often allowed.

$$R = S + E * amount / 100$$

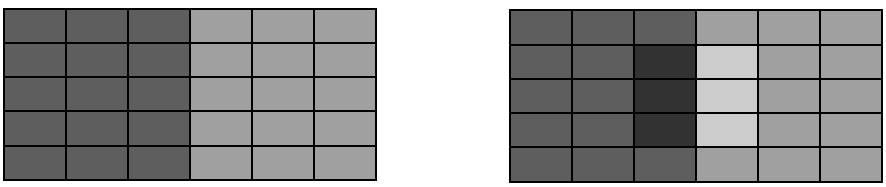
$$= S + (S - B) * amount / 100$$

When *amount* is 0%, there is no sharpening and the *result* is identical to the original *source*.

The following illustrates the example in the assignment sheet:

source					
94	94	94	160	160	160
94	94	94	160	160	160
94	94	94	160	160	160
94	94	94	160	160	160
94	94	94	160	160	160

result					
94	94	94	160	160	160
94	94	50	204	160	160
94	94	50	204	160	160
94	94	50	204	160	160
94	94	94	160	160	160



Although the effect will be easier to appreciate in a real-world image, even in this small example it is clear that the edge within the border has been emphasized by applying the sharpening technique.