C Lab, Assignment 3: Image Processing

IMPORTANT: Please read the assignment rules and submission procedures which are available in the assignments section of the module on surreylearn

`https://surreylearn.surrey.ac.uk/`

BEFORE starting the assignments.

The deadline for the assigment is shown on the Surreylearn submission system.

Your task in this assignment will be to implement some image processing algorithms. Note that in each part below, any output from your program should go to stdout (using printf statements) and NOT stderr.

When submitting the files, you MUST follow the name specification as described in the end of each part of the assignment. Failing to follow the name specification will result in **a penalty of 10 percent reduction** from your final mark.

Part 1: Read and Print an Image (40 marks)

In part 1 you should write a program that reads and prints an image, where the input image is run-length encoded.

Firstly, **cd** to your **eeclabs** directory. Then download the input files for part 1 into this directory from SurreyLearn. This should include the run-length encoded image file **panda.run** and the original image **panda.img**.

Then write your program in the file **image1.c**. It should take the run-length encoded image from the standard input, and print the image to standard output. Run-length encoding is a way of coding image data that takes advantage of the fact that large areas of images are homogeneous, i.e. have similar brightness locally. Following the image pixels in ‘‘`raster-scan`’’ order, illustrated in Figure 1:

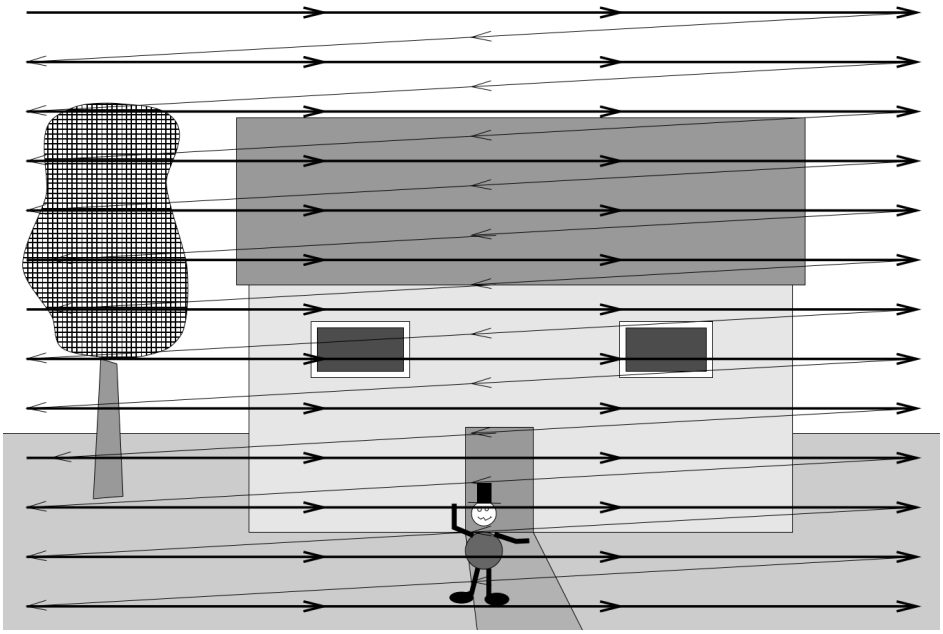

Figure 1: Raster scan order starts at the top-left of the image (row 1), and follows each row to the end, before dropping to the next row.

The format of the run-length encoded image is:

```
<width> <height> <grey-levels> <pix1> <len1> <pix2> <len2> .....
```

where all the `<...>`'s represent integer values. White-space (spaces and/or new-line characters) separate the values.

`<width>` is the width of the image in pixels. The limit on width is 80 pixels.

`<height>` is the height of the image in pixels. The limit on height is 100 pixels.

`<grey-levels>` is the number of grey levels in the image, for instance 2 for a black and white image. The grey-level pixel values can only take the values 0, 1,..., `<grey-levels>`−1. The maximum value for `<grey-levels>` is 4. The limits on `<width>`, `<height>` and `<grey-levels>` apply to all parts of the assignment.

`<pix1>` is the grey level pixel value of the first pixel on the first row of the image, i.e. the top-left pixel.

`<len1>` is the number of times that the first pixel is repeated, following the image pixels in raster-scan order.

`<pix2>` is the grey level pixel value of the next pixel, i.e. the first pixel different from `<pix1>`.

`<len2>` is the number of times that `<pix2>` is repeated.

`.....` the list of `<pix>`, `<len>` pairs continuing to the end of the image in the bottom-right hand corner.

The sum of the `<len>`s is equal to `<width>*<height>`. Once you have read the image into your program, print it to standard output. The format is as follows:

- One character per image pixel.
- Print a space (‘ ’) for pixel value `<grey-levels>`−1 (white), and a number character (‘#) for value 0 (“black”).
- If there are three grey levels, use a dot (‘.’) for value 1.
- If there are four grey levels, use a colon (‘:’) for 1 and a dot (‘.’) for 2.
- Print a new-line character at the end of every line.

For example, the run-length encoding for the image shown in Figure 2 is

```
6 5 4 0 2 3 2 0 5 1 2 2 2 3 1 2 4 1 3 0 6 3 2 1 1
```

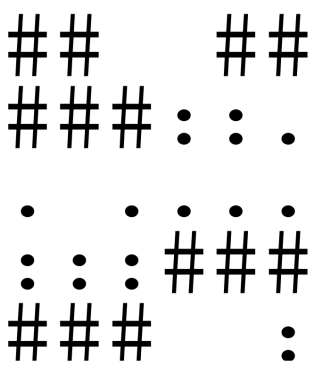

Figure 2: Example image (see text).

Compile your program using the command

```
gcc -ansi -Wall image1.c -o image1
```

Test it using the encoded test image panda.run:

```
./image < panda.run
```

You should obtain the output shown in figure 3.

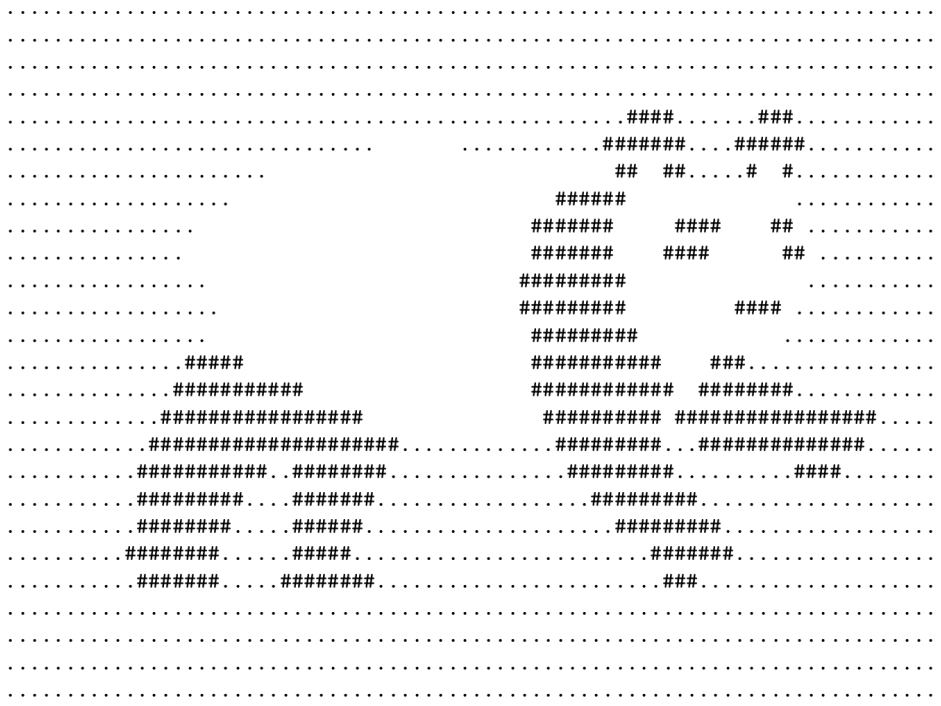

Figure 3: Output image for part 1 from file panda.run.

This corresponds to the same result as the original image in the file panda.img. To determine differences between your output and the correct output, perform the following test:

```
./image < panda.run > panda.new
diff panda.img panda.new
```

This will compare the two files and print out any differences. If nothing is printed it means that the two files are identical, which is what you want.

When you have thoroughly debugged and tested your program (image1.c), re-name your file according to the following specification and then submit the code to **SurreyLearn**.

SPEC: The name of the submitted file MUST be proceeded with your surname and initial followed by the name of the program. For example, if your surname is “Brown”, and your first name is “Peter”, you need to name the file as follows:

BROWNP-image1.c

The first part is your surname and initial followed by a hyphen, and then followed by the original filename (`image1.c`). If you also have middle names, ignore them in the file name.

Similarly, if your surname is "Jackson", and your first name is "Tony", you need to name the file as follows:

JACKSONT-image1.c

Part 2: Edge detection (30 marks)

Download the input files for part 2 from **SurreyLearn**.

This should include the run-length encoded image file **balloon.run**, the original image **balloon.img** and the edge detected image (i.e. the gradient image) **balloon.grad**. Copy the program **image1.c** from part 1 into a new file **image2.c** and modify the new program to apply an image processing algorithm, edge detection to the image, which locates boundaries in the image. You will only have to implement the first part of the edge detector, which computes the gradient of the image.

We shall compute the gradient as follows:

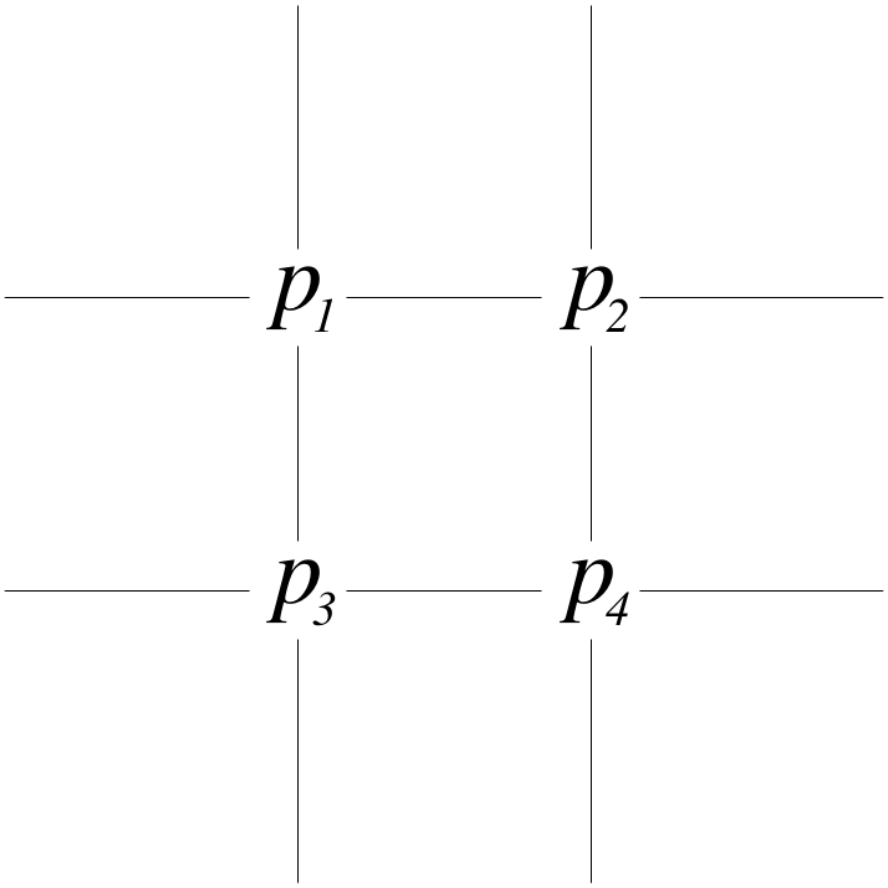

Figure 4: Configuration of adjacent pixels for gradient computation.

Let p_1, p_2, p_3, p_4 be adjacent pixels with p_2 to the right of p_1, p_3 below p_1, and p_4 below p_2, i.e. to the right of p_3, as in Figure 4. For each such set of four pixels, the gradient is computed along four directions: horizontal, vertical, positive diagonal (top-left to bottom-right) and negative diagonal (top-right to bottom-left). Then the gradient value is set to the highest of the gradient values computed along the four directions.

The formulae for the gradients is:

$$g_h = abs((p_1 - p_2 + p_3 - p_4)/2), \qquad g_p = abs(p_1 - p_4) \tag{1}$$

$$g_v = abs((p_1 - p_3 + p_2 - p_4)/2), \qquad g_n = abs(p_2 - p_3) \tag{2}$$

where `abs()` signifies the absolute value function.

Note that the resulting gradient "image" has one less row and column than the original image. For the sake of clarity, reverse the grey-levels when printing the result, so that ' ' corresponds to pixel value zero, '.' to one etc (this reversal applies only to this section). The original image (with 4 grey levels) is included in the file **balloon.img** so that you can check that you are decoding the image correctly. The gradient image is in **balloon.grad**. Check for differences using the diff command as above.

When you have thoroughly debugged and tested your program (image2.c), re-name your file according to the following specification and then submit the program to **SurreyLearn**.

SPEC: The name of the submitted file MUST be proceeded with your surname and initial followed by the name of the program. For example, if your surname is "Brown", and your first name is "Peter", you need to name the file as follows:

BROWNP-image2.c

The first part is your surname and initial followed by a hyphen, and then followed by the original filename (`image2.c`). If you also have middle names, ignore them in the file name.

Similarly, if your surname is "Jackson", and your first name is "Tony", you need to name the file as follows:

JACKSONT-image2.c

Part 3: Antialiasing (30 marks)

Download the input files for part 2 from **SurreyLearn**.

This should include the files **skull.run**, **skull.img**, **skull.double** and **skull.aa**.

Copy the program from part 1 (**image1.c**) into a new file **image3.c**. The goal here is to write a program that doubles the size of an image while smoothing out the boundaries in the image. First modify it to double the size of the image, in the following way.

The new image width will be $2 * $ `<width>` $- 1$ and the new height will be $2 * $ `<height>` $- 1$. It will have four grey levels instead of the two in the input image, so firstly convert pixels with value 1 to have value 3 (zero values stay at zero). The new pixels are then introduced between the old ones as shown in Figure 5.

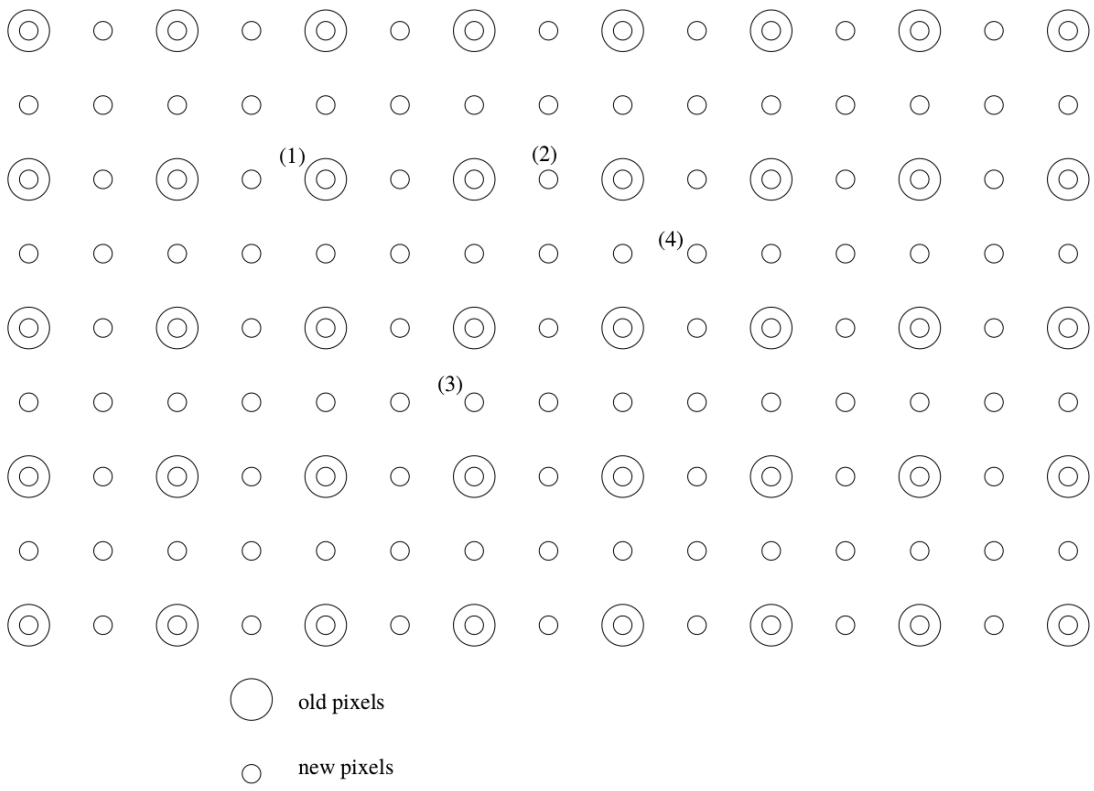

Figure 5: Arrangement of pixels in double-sized image.

The way the new pixels are computed is as follows:

- If a new pixel coincides with an old pixel (e.g. pixel (1) in Figure 5, set the new pixel value to the corresponding old pixel value.
- If a new pixel lies between two old pixels horizontally (e.g. pixel (2) in Figure 5) set the new pixel value to the average of the two corresponding old pixels. Here the "average" of two values a and b is defined as $(a + b)/2$ using the C language integer division.

- If a new pixel lies between two old pixels vertically (e.g. pixel (3) in Figure 5) set the new pixel value to the average of the two corresponding old pixels.
- If a new pixel lies between four old pixels (e.g. pixel (4) in Figure 5) set the new pixel value to the average of the four corresponding old pixels. Here the "average" of four values a, b, c and d is defined as $(a+b+c+d+1)/4$ using the C language integer division.

Once this is done, compare the output with the correct version in the file skull.double. The original image is in **skull.img**.

Finally, change the program to smooth the double-sized image before printing it. Smoothing is performed by repeated local averaging of pixel values. Each pixel p_{ij} in the (double-sized) image is replaced by the pixel value

$$s_{ij} = \frac{1}{16}\begin{pmatrix} & p_{i-1\,j-1} & + & 2p_{i-1\,j} & + & p_{i-1\,j+1} \\ + & 2p_{i\,j-1} & + & 4p_{i\,j} & + & 2p_{i\,j+1} \\ + & p_{i+1\,j-1} & + & 2p_{i+1\,j} & + & p_{i+1\,j+1} + 7 \end{pmatrix} \tag{3}$$

(NOTE: the term in brackets is not a vector or a matrix; merely the sum of ten numbers). Pixels at the edge of the image are copied directly into the smoothed image without smoothing. Finally the smoothed image pixels s_{ij} are copied back into the original p_{ij}, completing a single iteration of smoothing. Do three smoothing iterations in total. The output should be identical to the correct result in the file **skull.aa**.

When you have thoroughly debugged and tested your program (image3.c), re-name your file according to the following specification and then submit the program to **SurreyLearn**.

SPEC: The name of the submitted file MUST be proceeded with your surname and initial followed by the name of the program. For example, if your surname is "Brown", and your first name is "Peter", you need to name the file as follows:

BROWNP-image3.c

The first part is your surname and initial followed by a hyphen, and then followed by the original filename (`image3.c`). If you also have middle names, ignore them in the file name.

Similarly, if your surname is "Jackson", and your first name is "Tony", you need to name the file as follows:

JACKSONT-image3.c