C Tutorial
Image Processing

1 PART I – Image Processing

You will work with PPM and PGM images. In this tutorial an image is a 2D array with the appropiate width and height of a collection of pixels. PPM images utilise the RGB colour model, in which a colour is represented by the intensities of its red (R), green (G) and blue(B) components, in that order. PPM are considered 24-bit images, in which each pixel comprises 24 bits and thus allocates one byte to each colour component. The PGM images are 8-bit images, in which each pixel is repreesented by 8 bits as utilise GRAY colour model. In this exercise the C structure of an image defined in `loader.h` is as follows[1] :

```
 typedef struct {
  int width, height;
  int nChannels;
  int widthStep;
  int depth;
  uint8_t *pixelsData;
} image_t;
```

The `nChannels` represents the color model information of the image; e.g. a colour model requires 3 channels to represent the R,G and B components. A B/W or GRAY image requires only one channel as it has only one component. The `widthStep` represents the number of bytes between a pixel in one row and the same pixel position in the next row. Therefore, the `widthstep` is the multiplication of the `width` of the image per the number of color channels `nChannels`.The `depth` represents the maximum intensity value of the component, which in this exercise colour and gray images are always set to 255 (`DEPTH`), and in binary images (i.e. B/W images) the `depth` is set to 1 as it is its maximum value. [2]. `pixelsData` is a pointer to the collection of pixels. Following figure shows how a 3x2 image is structured in memory:

pixel0	pixel1	pixel2
pixel3	pixel4	pixel5

Figure 1: Image.

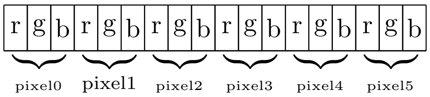

Figure 2: Memory layout of a 3x2 colour image.

pixel0	pixel1	pixel2	pixel3	pixel4	pixel5

Figure 3: Memory layout of a 3x2 b/w image.

[1] It is similar to the actual image structure in OpenCV

[2] see the enum type `DEPTH` defined in `loader.h`

1.1 Error handling

Many of the C functions you will be working with return a status code indicating whether or not they succeeded, or what happened in the event that they failed. Since C functions may only return one value, this means that such functions must accept *pointers* to which they can write their results.

As an example of this style, consider the `image_read` function (from `loader.c` or `loader.c`), which is used by `main.c` to read in an image. The `out` parameter is a pointer to an image pointer. The function will change the image pointer that `out` points to in order to pass back newly read image. `image_read` has the following structure:

```
image_error_t image_read(const char *filename, image_t **out) {
  /*
   * ... Open the file, returning if errors occur ...
   */

  /*
   * Allocate and read pixel data.
   */
  image->pixelsData = malloc(image->width * image->height *
  image->nChannels * sizeof(uint8_t));
  if (!image->pixelsData){
    return IMG_INSUFFICIENT_MEMORY;
 }

  /*
   * ... Read in the image, handling errors as necessary ...
   */

  /*
   * Update out to point to the new image.
   */
  *out = image;
  return IMG_OK;
}
```

`image_read` is called in `main.c` by passing a reference to an image pointer, as follows:

```
  /*
   * Create an image pointer, but don't allocate space for it
   * to point to.
   */
  image_t *img_in = NULL;
  image_error_t img_err;
  img_in_filename = argv[1];
  /*
   * Pass a pointer to the image pointer to image_read, which will
   * assign it a pointer which points to a piece of allocated
   * space.
   */
  img_err = image_read(img_in_filename,&img_in);
  /*
   * ... Check image_error, etc. ...
   */
```

Some of the functions you will write will require returning or checking status codes declared by the `image_error_t` enumeration, defined as follows in `loader.h`:

```
typedef enum{
  IMG_OK,
  IMG_OPEN_FAILURE,
  IMG_MISSING_FORMAT,
  IMG_INVALID_FORMAT,
  IMG_INSUFFICIENT_MEMORY,
  IMG_INVALID_SIZE,
  IMG_INVALID_DEPTH,
  IMG_READ_FAILURE,
  IMG_WRITE_FAILURE
  }image_error_t;
```

1.2 What to do

In this part you will have to convert from a colour image to a binary image. In order to do that, you will have to convert the colour image (3-bytes per pixel representation) to a gray image (1-byte per pixel representation) and then quantify the gray image with a threshold value 50 to obtained a b/w image.

1. Implement a function:

   ```
   image_error_t InitImage(image_t** dst, int width, int height, int nChannels, int depth);
   ```

   which will provide an image `dst` with the `width`, `height`, `nChannels` and `depth` given. Where the `width` corresponds to the width of an image and `height` to the height of an image and `nChannels` is the colour model `GRAY` or `RGB`.Bear in mind that in this exercise `depth` is always set to `DEPTH` enum value defined in `loader.h` if the image is gray or colour image, if you want to provide a binary image you must set the `epth` to 1 (as it is the maximum intensity value). Allocated memory for the `pixelsData` must be initialised to zero.

2. Implement a function:

   ```
   image_error_t ConvertColor(const image_t* src, image_t** dst);
   ```

   A colour model is a basic mathematical representation of how the colour should be perceived. To convert a colour space to gray space you should use the following equation:
   $Y = 0.2126 * r + 0.7152 * g + 0.0722 * b$
   where Y represents a gray intensity value from 0 to `DEPTH` in 8-bits.

   Create a new gray image (`dst`) using `InitImage` with the `width` and `height` of `src` image, `nchannels` set to `GRAY` and `depth` to `DEPTH`. Then, for each pixel of the `dst` image set the value as follows:

   pixel$_{\text{gray}}$[0] = 0.2126 * pixel$_{\text{colour\_r}}$[0] + 0.7152 * pixel$_{\text{colour\_g}}$[0] + 0.0722 * pixel$_{\text{colour\_b}}$[0];

   You can test the results by writting in `main.c`:

   (a) read the PPM input image from ../images/input.ppm using `image_read` in `loader.h`.
   (b) initiliase an image out to `GRAY` color format and `DEPTH`.
   (c) convert the color format input image to a gray format image.
   (d) write in ../images/outhreshold.pgm the gray image in PGM format.

   outhreshold.pgm should be the same image as input.ppm but in gray scale.

3. Implement a function:

   ```
   image_error_t ThresholdImage(const image_t* src, image_t** dst,int threshold);
   ```

which will transform a gray image `src` to a binary image `dst` using a `threshold` value of 50. Create a new binary image (`dst`) using `InitImage` with the `width` and `height` of `src` image, `nchannels` set to `GRAY` and `depth` to one. Then,per each pixel of the input gray image `src`, if its value(from 0...DEPTH)) is bigger than the `threshold`, set the pixel on the binary image `dst` to 1 otherwise the pixel will be set to 0.

You may want to test the results by writting in `main.c`:

(a) read the PGM input image from ../images/oouthreshold.pgm using `image_read` in `loader.h`.

(b) initiliase an image out to `GRAY` color format and value 1 in `depth`.

(c) threshold the gray format image to get a binary image.

(d) write in ../images/outbinary.pgm the binary image in PGM format.

outbinary.pgm should be the same image as input.ppm but in B/W.