**HW2: the image processing pipeline** CSx73 Computer Vision 19sp Johnstone due: 1.31.19

## Purpose

In HW2, you will:

- get better acquainted with your data, the image
- learn how a standard image is manipulated by the time it reaches you
- simulate the image processing pipeline
- start to learn Matlab

## Related reading

Szeliski 2.3

## Background

Digital SLR cameras allow access to the raw data from the image sensors, through a RAW image. (This is not the default mode, so you need to capture the image differently.) The pixels of a raw image reflect the original sensor data captured by the camera, without processing, plus some meta-data about how the image was captured. Put another way, it is the image before it has been linearized, demosaicked, white balanced, gamma corrected, and non-linearly compressed. Each pixel in a raw image is a scalar, carrying information about a single colour channel, not the 3 possible channels (RGB) in a standard image. To capture colour, the 3 colour channels are spread out using a Bayer mosaic. Typically, a raw image will pass through the image processing pipeline before it is stored as an image, cleaning up the image for better viewing, but also applying unpredictable nonlinear operations that prevent the effective use of some computer vision algorithms (e.g., shape from shading). In this homework, you will simulate the image processing pipeline of a raw image.

**early steps of the pipeline** linearize, discover the Bayer mosaic, white balance
**late steps of the pipeline** demosaic, brighten, gamma correct, compress

## x73 differentiation

**CS473** early steps of the pipeline
**CS673** entire pipeline
**CS773** entire pipeline + a 1-page review of one of the following journal papers:
R. Kimmel, 'Demosaicing: Image Reconstruction from Color CCD Samples', IEEE TIP, 1999 (find it at **Kimmel's website** along with **images**) (how are edges preserved by his demosaicing algorithm?) OR
P. Debevec and J. Malik, 'Recovering High Dynamic Range Radiance Maps from Photographs', SIGGRAPH 1997. (find it at **Debevec's website**)
**what you hand in** your Matlab script hw2.m; all of your intermediate and final images, in a tarball called 'hw2_images.tar'; 773 only: journal paper report

*A possible solution of this homework, 673/773 version: a standard RGB image computed from a RAW image (entire pipeline)*

You will start with a raw image in tiff format, rather than the camera's native format (e.g., .nef for Nikon, .cr2 for Canon), since Matlab cannot read the native format directly. You will massage the image, imitating steps of the image processing pipeline, yielding a viewable image similar to the image below, which will be similar to the image that is typically downloaded from a camera.

At each step, I will suggest Matlab functions that will help you solve that step. You should look at the Matlab documentation (doc foo) for these functions to learn how to apply them to the problem at hand. Some of these Matlab functions will be discussed in our Matlab lecture, some not.

You may prefer to display brightened images, by scaling, say by 4. But when writing an image to a file, use the original image.

## algorithm

- load your image
  - **steps**
    - read the image as-is (raw.tiff)
    - write its width, height, type (of each pixel), the minimum scalar value in the image, and the maximum scalar value, all to a file `hw1_x73sp19.txt`, one statistic per line
    - display the raw image
    - cast the image to double
  - **useful Matlab functions**: imread, fopen, fprintf, size, class, min, max, colon operator, double
- linearize
  - in this phase, you will shrink the range of the pixels, using a linear transform to map black to 0 and white to 1
  - **background**: the meta-data in a RAW image includes a black value b (all sensor values less than b correspond to black pixels, due to noise) and a saturation value s (all sensor values greater than s may be considered white pixels, since they represent pixels with a full well). I will provide b and s to you, since this metadata is lost as the raw image is converted to tiff.
  - **steps**
    - use a linear transform (translate and scale) to map the image to the range [0,1] so that the black value b becomes 0, the saturation value becomes 1, and values less than 0 or greater than 1 are clamped to [0,1] (that is, push values less than 0 to 0, values greater than 1 to 1)
    - display the linearized image
  - **useful matlab functions**: arithmetic operators, min, max

- visualize the Bayer mosaic
  - **background**: by zooming in on a small chunk of the image, you can see the underlying 2x2 patches caused by the Bayer mosaic
  - **steps**
    - extract a small chunk of the image that reveals the Bayer patterning: a good choice is the 100x100 patch at (1000,1000)
    - brighten this image, since it will be too dark to be visible: to crudely brighten any image at this stage, scale it, say by 5, while cropping to [0,1] (e.g., `img = min(1,min*5)`)
    - display a zoomed version of this image
  - **useful matlab functions**: imshow; see documentation on 'InitialMagnification' under Input Arguments; this will also show you how to use name-value pair arguments, a common feature in matlab

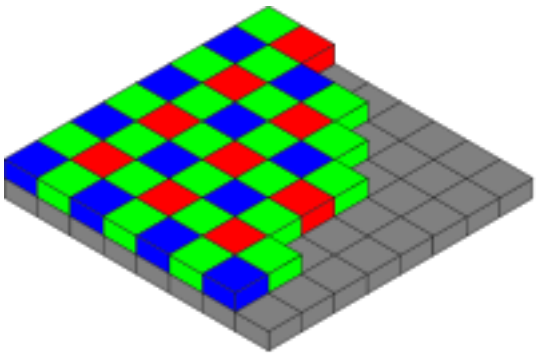

Figure 1: a Bayer mosaic

- discover the Bayer mosaic:
  - **background**: A Bayer mosaic is used to record colours (Fig 1), since a pixel can only record intensity of one colour. Twice as many greens are used since the human eye is more sensitive to green and green is a better indicator of the luminance of an image (the formula for mapping an RGB image to grayscale is grey = $0.2126R$ + *0.7152*G + 0.0722*B). There are four variants for the Bayer mosaic, depending on which colour starts (with two variants for green start): bggr, rggb, gbrg, grbg (row-major order of the top left 2x2 square of the Bayer mosaic).
  - **steps**
    - extract 4 quarter-resolution images from the linearized image, by extracting every other pixel
      - e.g., if the raw image uses bggr, im(1:2:end, 1:2:end) would be the blue channel (how would you extract the red component?)
      - suggested variable names for the 4 images: top_left, top_right, bot_left, bot_right
    - build 4 RGB images, one per Bayer mosaic candidate; an RGB image can be built from 3 1-channel images using cat (Lecture 2); hint: given a Bayer candidate, which subimage represents red? (draw a 2x2 box)
    - display the 4 RGB images resulting from each potential Bayer mosaic, labeled by their mosaic 4-tuple (e.g., bggr), in a 2x2 grid; brighten each image by scaling by 4, for better viewing

    * choose the best one as your encoding for downstream processing (hint: use the shirt), and write your chosen Bayer pattern to the file and to the screen (you have reverse engineered the camera sensor!)
  - in all downstream computation, please use the best Bayer mosaic
  - **useful Matlab functions**: colon operator, cat, subplot
- white balance:
  - in this phase, you will white-balance the image
  - **background**: since the colour of a pixel is a weighted combination of the colour of the object at that location and the colour of the impinging light, the colour of the pixel is not simply the colour of the object (as we might desire); in particular, white materials may not (probably will not) appear as white in the image (recall the Adelson experiment on the chessboard); to improve the image, you can (try to) force white materials to appear white in the image using white-balancing; there are two popular methods for white-balancing, grey-world and white-world; both scale the red and blue components to bring them into line with the green component.
  - note that the output of both white-balancings is still a Bayer-mosaic image (e.g., red stored in just 1 of every 4 pixels), but now white-balanced
  - in the suggested algorithm, you will extract the red, green, blue components from the mosaic, white-balance them, then reinsert them into a new mosaic
  - note that the 2 green pixels in a 2x2 Bayer block will not have equivalent intensities (do the same photons hit this pair of green sensors?); therefore, when extracting green, you should use the average of those 2 pixels (but vecctorize to average in one step, using mean)
  - white-balancing under the **grey-world assumption** is implemented as follows:
    * R_new = R * (G_avg / R_avg)
    * G_new = G
    * B_new = B * (G_avg / B_avg)
  - note that this leaves green alone, and pulls the others in agreement with it: the mean red/blue/green will now all agree.
  - white-balancing under the **white-world assumption** is implemented as follows:
    * R_new = R * (G_max / R_max)
    * G_new = G
    * B_new = B * (G_max / B_max)
  - note that the max red/blue/green will now all agree.
  - you will **implement both white-balancing options**, then decide which version is best *after you have completed the entire pipeline*

- **steps**
  - extract the red, green, blue channels from the Bayer mosaic into quarter-resolution images (using the Bayer pattern you learned in the previous step); for green, use the mean of the two green channels
  - build a white-balanced mosaic image using the grey-world assumption:
    - find the mean of each channel
    - inject the white-balanced channels back into a new mosaic (this is the opposite of your extraction step)
  - build a white-balanced mosaic image using the white-world assumption:
    - find the max of each channel
    - inject the white-balanced channels back into a new mosaic
  - display the white-balanced images, with labels 'grey-world' and 'white-world'
- write the white-balanced images to 'grey_world.jpg' /'white_world.jpg'
- **useful matlab functions**: colon operator, mean, zeros
- hint: in one step, you can inject the white-balanced red component into a Bayer mosaic, using matrix assignment
- note: white-balancing can be expressed using a scaling matrix (see lecture 3) but we have used a simpler component interpretation

- demosaic (673/773 only)
  - after white balancing, you are ready to demosaic the image: that is, to convert the partial RGB channels into full-resolution RGB channels; after demosaicing, the image will be a tensor with 3 channels in each pixel, rather than a matrix with one channel in each pixel
  - **background**: in the present image, consider the red channel. Many of the pixels of the raw image are missing a red component. However, the pixels that are missing red luckily have 4 red neighbours that can be used to define the missing red component. The mathematical technique to fill these gaps is called bilinear interpolation, and Matlab has a predefined function for bilinear interpolation (interp2), which you should use; take a look at Wikipedia for a definition of bilinear interpolation.
  - **steps**
    - use bilinear interpolation to fill in the gaps in red, defining the red channel
    - use bilinear interpolation to fill in the gaps in blue, defining the blue channel
    - use bilinear interpolation to fill in the gaps in green, defining the green channel
    - merge the red/green/blue channels into a single RGB image
  - **useful Matlab functions**: colon operator, interp2, meshgrid, zeros
  - hint, meshgrid can prepare some of the arguments needed by interp2; interp2 wants both coordinates and values (at those coordinates); meshgrid can prepare the coordinates; but notice that meshgrid's coordinate system is not image coordinates, but Euclidean coordinates, so flip x and y whenever using coordinates in meshgrid and interp2; see the documentation for interp2 and meshgrid; hint: [Ysize, Xsize] = size(raw); [Y,X] = meshgrid (1:2:Xsize, 1:2:Ysize);
  - suppose that grbg is your choice for mosaic pattern; then red values are in the top right slots (1:2:end, 2:2:end), and missing red values need to be added in the top left, bottom left, and bottom right slots, using bilinear interpolation;
  - note that construction of the red and blue channels are similar, while construction of the green channel is slightly different, since only 2/4 of the pixels are missing, rather than 3/4

- brightening and gamma correction (673/773 only):
  - **background**: The present image is a 16-bit linear RGB image (with each channel in the range [0,1]). An important characteristic is that it is a linear image, with values related to what the camera sensor recorded, which is very useful for many algorithms; however not for display. You will brighten the image and apply gamma-correction, to make it look better. This is a subjective computation, and depends on your monitor, so you can try different options. About gamma-correction (also read Szeliski 2.3): monitors are nonlinear with respect to input, and can be characterized by a gamma value $\gamma$, where an input intensity of $a$ is actually displayed with intensity $a^\gamma * \max$ (where max is the maximum intensity the monitor can generate). Gamma of a monitor can be estimated by finding the intensity $a$ that looks halfway between black and white (by comparing it to a black-white checkerboard at a distance) then solving for $\gamma$ in $a^\gamma = .5$: $\gamma = \frac{\ln 0.5}{\ln a}$. You may either try that technique for your monitor, or lock in a realistic choice of $\gamma = 2.2$. Another advantage of gamma correction (and one that is retained as monitors improve to reduce the difference between desired intensity and realized intensity) is that humans notice relative differences between intensities, rather than absolute differences, so a small difference at low intensity (near black) is more noticeable than the same small difference at high intensity (near white): mathematically, this means that perception of intensity is nonlinear. Gamma correction is a nonlinear transform that helps this perception.
  - **steps**
    - brightening: scale the image ($im \rightarrow \alpha$ * im), where $\alpha$ is a scalar that you determine as follows:
      - convert the image to grayscale
      - find the maximum (luminance) value $m$ in the grayscale image
      - choose $\alpha$ as a multiple of $m$
      - scale the image by $\alpha$ (remember to clip to max of 1)
      - report your brightening factor $\alpha$
    - gamma correction:
      - choose $\gamma$ (2.2 is a safe choice, but you can fool around with it)
      - set each intensity i to $i^{\frac{1}{\gamma}}$
  - **useful Matlab functions**: rgb2gray
- compression (673/773 only):
  - **background**: most image formats compress the image for efficient storage, but this may also affect its quality, depending on the amount of compression
  - **steps**
    - store the image in png format (an uncompressed format)
    - store the image in jpg format, using the default parameter settings
    - record the two sizes, and the compression ratio (ratio of two sizes)
    - adjust the quality of the jpg image until you start to notice a difference between the jpg and png image
    - report this quality and the compression ratio
  - **useful Matlab functions**: imwrite (and its Quality name-value pair)