BBM 444 – Programming Assignment 2: HDR Imaging and Tonemapping

Due date: Monday, 30.03.2026, 23:59

Overview

The goal of this assignment is to explore high dynamic range (HDR) imaging, color calibration, and tonemapping[1]. As we discussed in class, HDR imaging can be used to create floating-point precision images that linearly map to scene radiance values. Color calibration ensures that the colors you see in the image match some groundtruth RGB values. Tonemapping algorithms compress the dynamic range of HDR images to an 8-bit range, so that they can be shown on a display. To get full credits, you will need to apply all these steps to an exposure stack.

Throughout the assignment, we refer to a number of key papers that were also discussed in class. While the assignment and class slides describe most of the steps you need to perform, we highly recommend that you read the associated papers.

As always, there is a "Hints and Information" section at the end of this document that is likely to help. It is highly recommended that you read that section in full before starting to work on the assignment. The Python packages required for this assignment are `rawpy`, `numpy`, `skimage`, `matplotlib`, and `cv2` (OpenCV, to read and write HDR files), and you can use the functions provided in the `./src/cp_assng2.py` file of the assignment ZIP archive.

1. HDR imaging (60 points)

For this and the following two parts (color correction and tonemapping), you will use an exposure stack of an office using a Nikon D2X camera. The image files are in the `./data/willy_desk` directory of the assignment ZIP archive. Figure 1 shows two exposures, as well as a (tonemapped) HDR composite.

While not particularly beautiful, the scene has a number of features that make it a good example for HDR: First, the light from the window lightens the scene around, while the rest are darker. There are two areas with very different illumination and dynamic range that no single exposure can simultaneously capture correctly. Second, both areas include various items (Books and colorful bottles on the desk in the bright area, under-the-desk cables, dark folders, and book covers in the dark area) that you can use to evaluate the color rendition of your results. Third, the in-focus area has high-detail features (lettering on the book covers and computer screen) that you can use to evaluate the resolution of your results. Finally, the scene includes a color checker that you can use for color calibration in the bonus question.

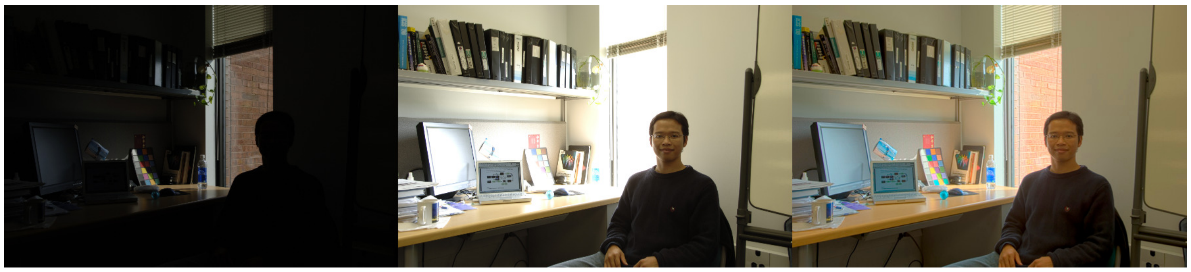

Figure 1: From left to right: Two LDR exposures, and an HDR composite tonemapped using the photo- graphic tonemapping.

You will notice that in the data folder, there are two sets of images, RAW (`.NEF`) and rendered

[1]Adapted from the programming assignment developed by Ioannis Gkioulekas for his computational photography class.

(`.JPG`). As discussed in class, the procedure for merging many low dynamic range (LDR) exposures into an HDR image is different for RAW and rendered images. To appreciate the difference, in this assignment, you will create HDR images from both sets of images. For reference, we captured both exposure stacks with fixed aperture and ISO, and with shutter speeds equal to $\frac{1}{320} \cdot 2^{k-40}$, where $k \in \{36, \dots, 44\}$ is the index in an image's file name.

Develop RAW images (5 points). Use `rawpy` (See Assignment 1) to convert the RAW `.NEF` images into linear 16-bit `.TIFF` images. For this, you should direct `rawpy` to do white balancing using the camera's profile for white balancing, do demosaicing using high-quality interpolation, and use sRGB as the output color space. Read through `rawpy`'s documentation, and figure out what the correct set of parameters for this conversion are.

Linearize rendered images (25 points). Unlike the RAW images, which are linear, the rendered images are non-linear. As we saw in class, before you can merge them into an HDR image, you first need to perform radiometric calibration in order to undo this non-linearity. You will do this using the method by Debevec and Malik [1]. We describe how this works below, but you are strongly encouraged to read at least Section 2.1 of this paper, which explains the method.

An intensity $I_{ij}^k \in \{0, \dots, 255\}$ at pixel $\{i, j\}$ of image k relates to some unknown scene flux value L_{ij} as

$$I_{ij}^k = f(t^k L_{ij}) \; , \tag{1}$$

where t^k is the (known) exposure of image k and f is the unknown non-linearity applied by the camera. If we knew f^{-1}, we could convert I_{ij}^k back to linear measurements.

Instead of f^{-1}, you will recover the function $g := \log(f^{-1})$ that maps the pixel values I_{ij}^k to $g(I_{ij}^k) = \log(L_{ij}) + \log(t^k)$. This is motivated by the fact that the human visual systems responds to logarithmic, instead of linear, intensity. As the domain of g are the discrete intensity values $\{0, \dots, 255\}$, g is essentially just a 256-dimensional vector.

Solving for these 256 values may seem impossible, because we know neither g nor L_{ij}. However, if the imaged scene remains static while capturing the exposure stack, we can take advantage of the fact that the value L_{ij} is constant across all LDR images. Then, we can recover g by solving the following least-squares optimization problem,

$$\min_{g, L_{ij}} \sum_{i,j} \sum_{k} \{w(I_{ij}^k/255)\,[g(I_{ij}^k) - \log(L_{ij}) - \log(t^k)]\}^2 + \lambda \sum_{z=0}^{255} \{w(z/255)\nabla^2 g(z)\}^2 \; . \tag{2}$$

The first term in Equation (2) is the *data term*, and encourages values g and L_{ij} to be such that intensities in linear images would scale linearly with exposure time. As we discussed in class, the weights w have to do with the fact that the linear estimates should rely more on well-exposed pixels than on under-exposed or over-exposed pixels. See later in Problem 1 ("Weighting schemes") about what weights exactly you should use. Note that the input to w is divided by 255, because the definitions of the weights assume that the input intensities are in the range [0, 1], whereas the ones you use are in the range $\{0, \dots, 255\}$.

The second term in Equation (2) is a *regularization term*, and encourages g to be smooth by penalizing solutions g which have large second-order derivative magnitudes. Given that g is discrete, the second order derivative can be approximated using a Laplacian filter, which can be defined as $\nabla^2 g(z) = g(z+1) - 2g(z) + g(z-1)$. The parameter λ controls how strongly this regularization affects the final result; it is a hyper-parameter that you will need to experiment with. Note that, when using the photon-optimal weights w_{photon} that require knowing exposure time, you can set the weights of the *regularization term* only to a constant (e.g., $w(z) = 1$).

Solve the *least-squares* optimization problem of Equation (2) by expressing it in matrix form:

$$||\mathbf{A}\mathbf{v} - \mathbf{b}||^2 \,, \tag{3}$$

where $\mathbf{A}$ is a matrix, $\mathbf{v} = [g; \log(L_{ij})]$ are the unknowns, and b is a known vector. Then, use one of NumPy's solvers to recover the unknowns. (See `numpy` function `numpy.linalg.lstsq`.)

While Debevec and Malik [1] recover a different g for each color channel, for this homework we recommend that you process pixels from all three channels simultaneously to recover a single g for all channels. This helps reduce color artifacts in the final HDR composite.

Plot the function g you recovered, then use it to convert the non-linear images I_{ij}^k into linear ones,

$$I_{ij,\text{lin}}^k = \exp\left(g(I_{ij}^k)\right) \tag{4}$$

Note that you will not use the values L_{ij} you recover from solving Equation (2).

Merge exposure stack into HDR image (30 points). Now that we have two sets of (approximately) linear images, coming from the RAW and rendered files, it is time to merge each one of them into an HDR image. This part will be common for both sets of linear images. Make sure that each HDR image you create only uses images from one or the other set.

Given a set of k LDR linear images corresponding to different exposures t^k, we can merge them into an HDR image either in the linear or in the logarithmic domain. The motivation for linear merging is physical accuracy, whereas the motivation for logarithmic merging is, as mentioned above, human visual perception.

We first introduce some notation. We use $I_{ij,\text{LDR}}^k$ to refer to the intensity value of pixel $\{i, j\}$ of the k-th *original* LDR input image, read directly from either a `.JPG` or a `.TIFF` file. We use $I_{ij,\text{lin}}^k$ to refer to the intensity of pixel $\{i, j\}$ of the k-th *linear* LDR input image; this is either the intensity from Equation (4) when using `.JPG` files, or the same as $I_{ij,LDR}^k$ when using `.TIFF` files. Additionally, from this point on we assume that $I_{ij,\text{LDR}}^k \in [0, 1]$. Therefore, you need to normalize the original LDR input images to the [0,1] range, which you can do by dividing with 255 when using `.JPG` files, and by $2^{16} - 1$ when using `.TIFF` files.

With this notation at hand, when using linear merging, we form the HDR image as:

$$I_{ij,\text{HDR}} = \frac{\sum_k w(I_{ij,\text{LDR}}^k) I_{ij,\text{lin}}^k / t^k}{\sum_k w(I_{ij,\text{LDR}}^k)} \tag{5}$$

When using logarithmic merging, we form the HDR image as:

$$I_{ij,\text{HDR}} = \exp\left(\frac{\sum_k w(I_{ij,\text{LDR}}^k)\left(\log\left(I_{ij,\text{lin}}^k + \epsilon\right) - \log(t^k)\right)}{\sum_k w(I_{ij,\text{LDR}}^k)}\right) \tag{6}$$

where ϵ is a small constant to avoid the singularity of the logarithm function at 0. As before, the weights w in Equations (5) and (6) can be used to place more emphasis on well-exposed pixels, and less emphasis on under-exposed or over-exposed ones. See below about what weights to use.

Implement both linear and logarithmic merging for each of the two exposure stacks. Then, store the resulting HDR images as `.HDR` files, which is an open source high dynamic range file format. (See the provided function `writeHDR` in `./src/cp_assgn2.py`)

Weighting schemes. There are many possible weighting scheme choices [2]. You will implement four:

$$w_{\text{uniform}} = \begin{cases} 1, & \text{if } Z_{\min} \leq z \leq Z_{\max}, \\ 0, & \text{otherwise} \end{cases}, \tag{7}$$

$$w_{\text{tent}} = \begin{cases} \min(z, 1-z), & \text{if } Z_{\min} \leq z \leq Z_{\max}, \\ 0, & \text{otherwise} \end{cases},$$

$$w_{\text{Gaussian}} = \begin{cases} \exp\left(-4\frac{(z-0.5)^2}{0.5^2}\right), & \text{if } Z_{\min} \leq z \leq Z_{\max}, \\ 0, & \text{otherwise} \end{cases},$$

$$w_{\text{photon}} = \begin{cases} t^k, & \text{if } Z_{\min} \leq z \leq Z_{\max}, \\ 0, & \text{otherwise} \end{cases},$$

All the above weighting schemes assume that the intensity values are in the range $z \in [0, 1]$. You can experiment with different clipping values $Z_{\min}$ and $Z_{\max}$, but we recommend $Z_{\min} = 0.05$, $Z_{\max} = 0.95$. Unlike the other schemes, the weights w_{photon} also depend on the exposure under which a pixel was captured.

Note that, when creating an HDR image from the `.JPG` stack, you need to use the same weighting scheme in both Equations (2) (linearization) and (5)-(6) (merging).

Implement all the above weighting schemes, and use them to create HDR images. In total, you will create 16 HDR images: 2 sets of images (RAW and rendered) $\times$ 2 merging schemes (linear and logarithmic) $\times$ 4 weighting schemes (uniform, tent, Gaussian, and photon-noise optimal).

Make your pick. Select one out of the sixteen HDR images you created. You can select, for example, the one that you find the most aesthetically pleasing. Make sure to comment on why you selected the image you did. Note that, as you have not yet tonemapped your HDR images, if you display them directly they will not look very nice; see "Hints and Information".

2. Color correction and white balancing (20 points)

For this part, you will use the HDR image you selected at the end of Part 1. As shown to the left of Figure 2, your tonemapped images will tend to have an orange cast in the dark parts of the room. This is because the very low light inside the room and the large contrast with the light outside the room are throwing the camera's automatic white balancing off. Additionally, even if the white balancing worked perfectly, we have not been very careful about the color space the various image composites reside in.

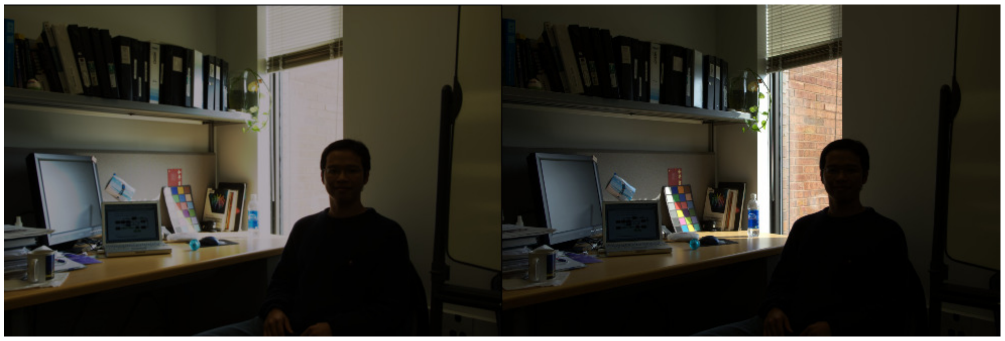

Figure 2: Tonemapped HDR image without (left) and with (right) color correction.

You could apply any of the automatic white balancing algorithms we discussed in Programming Assignment 1 to ameliorate the issue. But, given that the images include a color checker (Figure 3), it is possible to do better than that and perform accurate color correction.

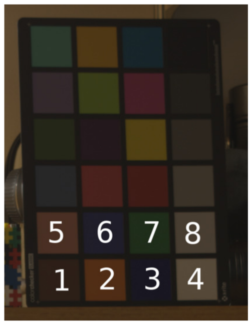

Figure 3: Color checker and patch numbering.

In particular, the color checker is created in such a way that its patches have a specific set of RGB coordinates in the (linear) sRGB color space, when the color checker is viewed under a standard illuminant (so-called "D65" illumination, roughly corresponding to daylight at noon). The function `read_colorchecker_gm`, provided in the `./src/cp_assgn2.py` file of the homework ZIP archive, returns these ground-truth RGB coordinate values, with the patches numbered as shown in Figure 3.

Then, in order to have your HDR image show color correctly, you can apply a linear transform on its three channels, so that the color checker's RGB coordinates in the image match the ground-truth coordinates as closely as possible. You can do this as follows.

1. For each color checker patch, crop a square that is fully contained within the patch. (See matplotlib function `matplotlib.pyplot.ginput` for interactively recording image coordinates.) Make sure to store the coordinates of these cropped squares, so that you can reuse them. Use the resulting 24 crops to compute average RGB coordinates for each of the color checker's 24 patches.

2. Convert these computed RGB coordinates into *homogeneous* 4×1 coordinates, by appending a 1 as their fourth coordinate.

3. Solve a least-squares problem to compute an affine transformation, mapping the measured to the ground-truth homogeneous coordinates.

4. Apply the computed *affine transform* to your original RGB HDR image. Note that the transformed image may have some negative values, which you should clip to 0.

5. Finally, apply an additional white balancing transform (i.e., multiply each channel with a scalar), so that the RGB coordinates of patch 4 are equal to each other. This is analogous to the manual white balancing in Programming Assignment 1, where now we use patch 4 as the white object in the scene.

Store the color corrected and white balanced HDR image in an `.HDR` file. You should now have two HDR images total: The one from Part 1 that has not been color-corrected, and the one you just created. Compare the color-corrected image with the original, and discuss which one you like the best.

3. Photographic tonemapping (20 points)

Now that you have a couple of HDR images, you need to tonemap them so that you can display them. For this part, you can use whichever of the two HDR images at the end of Part 2 you liked the best. You will implement the tonemapping operator proposed by Reinhard et al. [3], which is a good baseline to start from when displaying HDR images. We describe how to do this below, but you are strongly encouraged to read at least Sections 2 and 3 of this paper, which explain the rationale behind the specific form of this tonemapping operator, the effect of the various parameters, and its relationship to the zone system used when developing film.

Given pixel values $I_{ij,\text{HDR}}$ of a linear HDR image, photographic tonemapping is performed as where[2]

$$\begin{aligned}
\tilde{I}_{\text{white}} &= B \cdot \max_{i,j}\left(\tilde{I}_{ij,\text{HDR}}\right) , && (8)\\
\tilde{I}_{ij,\text{HDR}} &= \frac{K}{I_{m,\text{HDR}}} I_{ij,\text{HDR}} , && (9)\\
I_{m,\text{HDR}} &= \exp\left(\frac{1}{N}\sum_{i,j}\log(I_{ij,\text{HDR}}+\epsilon)\right) , && (10)
\end{aligned}$$

The parameter K is the key, and determines how bright or dark the resulting tonemapped rendition is. The parameter B is the *burn*, and can be used to suppress the contrast of the result. Finally, N is the number of pixels, and ϵ is a small constant to avoid the singularity of the logarithm function at 0.

Implement the photographic operator and apply it to your RGB HDR images in two ways: First, apply it by tonemapping all color channels simultaneously in the same way. Second, apply it only to the luminance channel Y. For the latter, you can use the provided function `lRGB2XYZ` to convert the HDR image from RGB to XYZ, and then convert it to xyY using the definition discussed in class. While in xyY, tonemap the luminance Y while leaving the chromaticity channels x, y untouched. Then, invert the color transform to go back to RGB using the provided function `XYZ2lRGB`.

Experiment with different key and burn values. Some reasonable starting values for the parameters are $K = 0.15$ and $B = 0.95$, but to get good tonemaps you will need to explore different values. Plot representative tonemaps for both the RGB and luminance methods, and discuss your results. Make sure to mention which tonemap you like the most.

4. (Bonus) Create and tonemap your own HDR photos (50 points)

It is now time to apply what you implemented above to your own pictures. To create results which are clearly better than any single exposure, you should use the pictures of a scene that actually has a high dynamic range! Good examples include: scenes that have both indoor and outdoor elements (a room with windows), indoor scenes with two different illuminations (like the data you used in parts 1-3), scenes with very strong backlighting, or outdoors scenes during a sunny day with strong shadows. Please make sure to read the suggestions in Hints and Information on what camera settings to use.

Once you select the scene, capture exposure stacks in RAW and JPEG formats. We suggest using exposures that are equally spaced in the *logarithmic domain*. For example, start with some very low base exposure, and then use exposures that are 2× the base, 4×, 8×, and so on. You can either exhaust the exposure range (i.e., start from the lowest shutter speed possible, and go all the way to the maximum shutter speed in 2x steps), or select an exposure range that works for your scene.

Use the exposure stacks you captured to create two HDR images, one from the RAW and one from the JPEG images. You can use whichever of the HDR variants you implemented in Part 1 you prefer—or you can try out all of them and decide which one looks the best. Store these two images in `.HDR` format. Since you do not have a color checker, you can skip the color calibration step.

[2]Equation (10) is different from the corresponding Equation (1) in Reinhard et al. [3]. The version given here is correct, and the version in the paper is incorrect.

Then, process these images using the tonemapping algorithms you implemented in Part 3 (photographic, in RGB or luminance-only). Experiment with different parameters, show a few representative tonemaps, discuss your results, and determine which result you like the most. The total number of points you will get for this part will depend on how visually compelling the final tonemapped image you create is.

What to Hand In

Your submitted solution should include the following:

- The filled-in Jupyter Notebook as both your source code and report. The notebook should include (1) markdown cells reporting your written answers alongside any relevant figures and images and (2) well-commented code cells with reproducible results.
- Any figure you want to appear in any markdown cell in the notebook.
- You need to submit the HDR images that you create in parts 1 (only the one you pick at the end), and 2. If you do the bonus part 4, you also need to submit the HDR images that you create as well as at least one RAW and corresponding `.JPG` LDR image you capture. You can also include additional image files, LDR or HDR, for various experiments (e.g., tonemapping with different values) other than your final ones, if you think they show something important.
- **Important Note:** If the file size exceeds the limit allowed by the submission system, please upload your images to a cloud storage service (e.g., Google Drive, OneDrive, Dropbox, etc.) and include the shared link in your notebook file. Make sure the link remains accessible after the submission deadline.

You should prepare a ZIP file named `name-surname(s)-assgn2.zip` containing the files stated above and submit it to `submit.cs.hacettepe.edu.tr`, where you will be assigned a submission. The file hierarchy is up to you as long as your Jupyter notebook works fine.

Late policy

You may use up to five *extension* days (in total) over the course of the semester for the programming assignments. Late submission will not be allowed.

Academic Integrity

All work on assignments must be done individually unless stated otherwise. You are encouraged to discuss with your other classmates about the given assignments, but these discussions should be carried out in an abstract way. That is, discussions related to a particular solution to a specific problem (either in actual code or in the pseudocode) will not be tolerated. In short, turning in someone else's work, in whole or in part, as your own will be considered as a violation of academic integrity. Please note that the former condition also holds for the material found on the web as everything on the web has been written by someone else.

Hints and Information

- Download and install the latest version of `LibRaw` from `https://www.libraw.org/download`, then install the Python binding by running `pip install rawpy`. To verify the installation, check both the `rawpy` and bundled `LibRaw` versions by inspecting `rawpy.__version__` and `rawpy.libraw_version`.

- When working with the provided and captured exposure stacks, you will notice that your algorithms will be using a lot of memory. This is a common issue when processing photographs captured with modern cameras, due to the very large number of pixels these cameras have. At 12 Megapixels, the Nikon D2X used for this assignment is at the mid-range of megapixels. Still, at this resolution, a 3-channel HDR image takes up more than 0.5 GB of memory.

  This has two implications. First, you should be careful about how many of these images you create in your Python code, as otherwise you run the risk of filling up your memory and crippling your computer. Second, when processing an image, you need to make sure you use vectorized code that processes all of its pixels in parallel, as trying to process all 12 million pixels one-by-one with a double for loop will take ages.

  In particular, when performing HDR merging, note that Equations (5)-(6) can be applied to each of the k exposure images independently. Therefore, instead of loading the entire exposure stack at once, you can load its images and process them one by one. Additionally, within each image, Equations (5)-(6) apply to each pixel in a completely parallel way. Therefore, you can process each image with a single vectorized call, instead of a double `for` loop.

  One place where, no matter how careful you are, you will run out of memory is when solving the linear system (3) to recover the non-linear map g. As suggested by Debevec and Malik [1], you should greatly downsample the input images before forming the linear system. Note that you should *not* resize the image with `skimage.transform.resize`, or try to blur it before downsampling. For the purposes of inferring g, all you have to do is downsample an input image `I` with `I[::N, ::N]`, for some N. We recommend using $N = 200$.

  More generally, when you are still debugging your code, we strongly recommend that you work on downsampled images to accelerate the development process. Once you know your code is correct, you can run it one more time on the full-resolution image, to produce your final results.

- When merging many LDR images to HDR ones, you may end up with pixels for which there are not any well-exposed values (i.e., the sum of weights in the denominators of Equations (5)-(6) is exactly 0). You can set those pixels to equal the maximum or minimum valid pixel value of your HDR image, respectively for problematic pixels that are always over-exposed or always under-exposed.

- Even with tonemapping, your images may appear too dark. In practice, after tonemapping, you still need to apply gamma encoding for images to be displayed correctly. As a reminder from Programming Assignment 1, gamma encoding is the following non-linear operator:

$$C_{\text{non-linear}} = \begin{cases} 12.92 \cdot C_{\text{linear}}, & C_{\text{linear}} \leq 0.0031308, \\ (1 + 0.055) \cdot C_{\text{linear}}^{\frac{1}{2.4}} - 0.055, & C_{\text{linear}} > 0.0031308, \end{cases} \tag{11}$$

  You should implement this in a script, and use it to always apply gamma encoding to tonemapped or HDR images before displaying them. Applying gamma encoding will also be helpful for displaying intermediate results (see below).

- As in Programming Assignment 1, you will likely find it helpful to display intermediate results. If you directly display the HDR images you create, they may appear very bright (potentially fully-white) or very dark (potentially fully-black). This is *not* a problem: as we discussed in class, HDR images are linear with respect to incident flux, but are scaled by a (somewhat) arbitrary scaling factor. All you have to do is multiply your image with an appropriate scaling factor of your own (smaller than 1 if the image is very bright, larger than 1 otherwise), apply gamma encoding, and then use the `clip` and `imshow` functions as in

Programming Assignment 1. You will likely need to experiment with a few different values for the scaling factor you apply, until you find the one that correctly exposes your image.

- If you want to view the `.HDR` files you create, note that you cannot do so using a standard image viewer. Instead, you should use a dedicated viewer for `.HDR` files, such as `OpenHDR`[3]. This viewer provides interactive sliders for controlling exposure (the scaling factor you apply to the image) and gamma encoding, making it easier to find good settings for examining your HDR image. Alternatively, you can use the function `readHDR` in the code we provide to open and load the `.HDR` in Python, then display it as we describe in the previous step.

- When applying photographic tonemapping to each RGB channel separately, you may get better results by using the same scalars $I_{m,\text{HDR}}$ and $\tilde{I}_{\text{white}}$ for all three channels. You can do this by using pixels from all three channels in Equations (8) and (10).

  Additionally, evaluating Equation (10) as written (i.e., by first computing the average of logarithms, and then exponentiating) may result in zero, NaN, or Inf values due to finite numerical precision. You may get more stable results by recognizing that Equation (10) is equivalent to computing the geometric mean of all pixels $I_{i,j,\text{HDR}}$, and changing your implementation accordingly. For more information about this type of numerical issues, look up "log-average form of geometric mean".

- When using your camera to capture your own exposure stack, you should make sure to set its white balancing option to a fixed preset, as appropriate for the lighting in the scene you selected, and its output color space to sRGB. Additionally, if your camera supports this, set it to store both RAW and `.JPG` files for each image you capture (the Nikon D2X has this option). That way, you will have perfectly paired RAW and `.JPG` exposure stacks, and you can use them to compare doing HDR with one or the other.

- While capturing your exposure stack, it is critical that no camera parameters other than shutter speed change. Therefore, you should set the camera to manual mode, and disable auto-focus for the duration of the capture. If you do not take these steps, then the camera may automatically change parameters such as aperture, ISO, and focus, making your data unusable.

  Regarding aperture, you should use an aperture setting that gives you good depth of field for the scene you selected for your exposure stack.

  Regarding focusing, you can use autofocus while framing the scene you will use for your exposure stack, to make sure that your captured images will be sharp. Once the lens has been focused, you can then disable autofocus, switch to manual, and start capturing your exposure stack.

- As discussed both in class and above in the assignment, it is very important that both your camera and your scene remain static while capturing your exposure stack. Given this, we strongly recommend that you mount your camera on a tripod, or at the very least on a very stable surface (e.g., a table) when taking images.

  While capturing your exposure stack, you will need to adjust the camera's shutter speed several times. Doing this manually requires touching the camera to rotate the shutter speed dial. You will also need to activate the shutter release, which means further touching the camera and pressing buttons. All of these manual actions can result in considerable camera movement, and therefore in your captured LDR images being misaligned. Using a tripod does not protect you from this type of camera motion.

[3]You can download it via `https://viewer.openhdr.org/`.

Therefore, we strongly recommend that you *tether*, i.e., connect, the camera to your laptop, so that you can control its settings and shutter release electronically, without touching the camera. Each of the class cameras comes with a USB cable you can use for this purpose.

To control the camera, you can try using the software provided by each manufacturer on their website.

As an alternative, we recommend that you try `gphoto2`[4]. This is a very powerful command-line tool that can be used to script your camera and implement very complicated capture procedures. For example, the following lines auto-detect a connected camera, capture an image at shutter speed 1/2048, and then download the images from the camera to your computer and store them with filename `exposure1`. If your camera is set to capture both RAW and `.JPG`, this excerpt will download both images and store them as `exposure1.nef` and `exposure.jpg`, respectively.

```
gphoto2 --auto-detect
gphoto2 --set-config-value /main/capturesettings/shutterspeed=1/2048
gphoto2 --capture-image-and-download --filename exposure1.%C
```

References

[1] P. E. Debevec and J. Malik. Recovering high dynamic range radiance maps from photographs. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 369–378, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[2] K. Kirk and H. J. Andersen. Noise characterization of weighting schemes for combination of multiple exposures. In *BMVC*, volume 3, pages 1129–1138, 2006.

[3] E. Reinhard, M. Stark, P. Shirley, and J. Ferwerda. Photographic tone reproduction for digital images. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '02, pages 267–276, New York, NY, USA, 2002. ACM.

[4] You can download it via `http://gphoto.org/proj/gphoto2/`.