Hieu Nguyen
GTID: 903185448
Email: hieu@gatech.edu
10/26/15

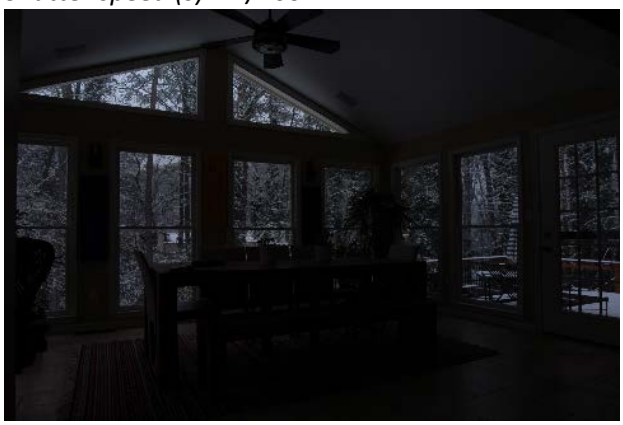
CS-6475 Computational Photography Assignment 9

High Dynamic Range (HDR) imaging is a technique used to reproduce a greater dynamic range of luminosity than is possible with standard digital imaging or photographic techniques. By using a set of images taken across an exposure spectrum, we can re-map the pixel intensities into a single image that represents the dynamic range of the scene.

Here is an example input/output with the provided "home" scene with an indoor foreground and a snowy, outdoor background.

Input Images:

Shutter speed (s) = 1/160



1/125

1/80

1/60

1/40

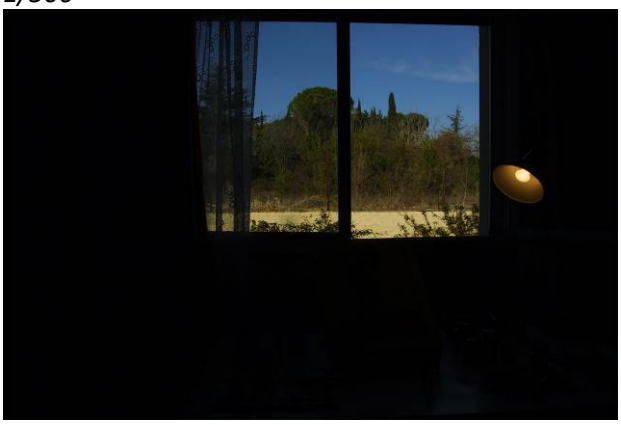
1/15

Output Image:

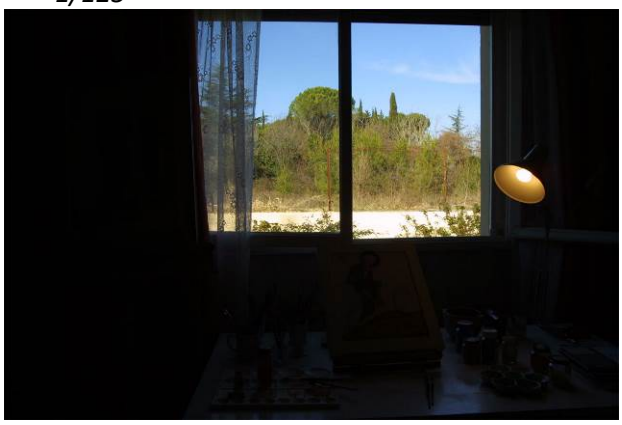
Here is the HDR result from a collection of “studio” images taken from
http://www.hdrsoft.com/examples2.html

Input Images:

1/500



1/125



1/30

1/8

1/2

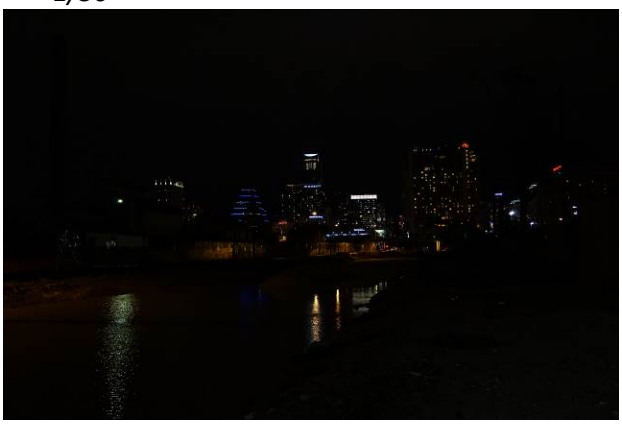
Output Image:

Lastly, here is the HDR result from images that I captured with my own camera. They are images of the downtown skyline in Austin, at night. I know that HDR is normally best-suited for indoor/outdoor environments with a far and natural source of illumination (e.g. our Sun), but I wanted to see how this technique would perform in a dark scene with several artificial lights. The result doesn't look very natural, but you can make out a good amount of detail without being overexposed in certain regions.

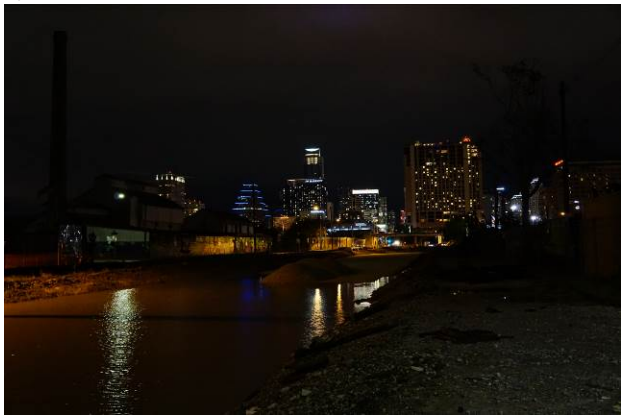
Input Images:

1/125

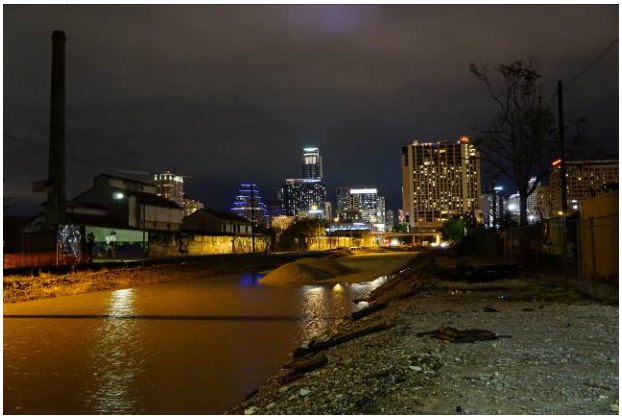
1/50



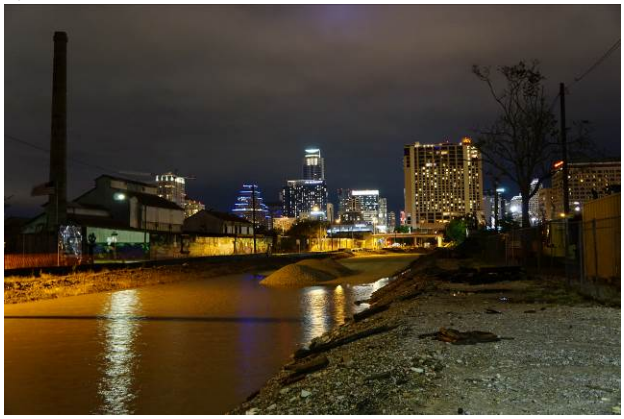
1/15



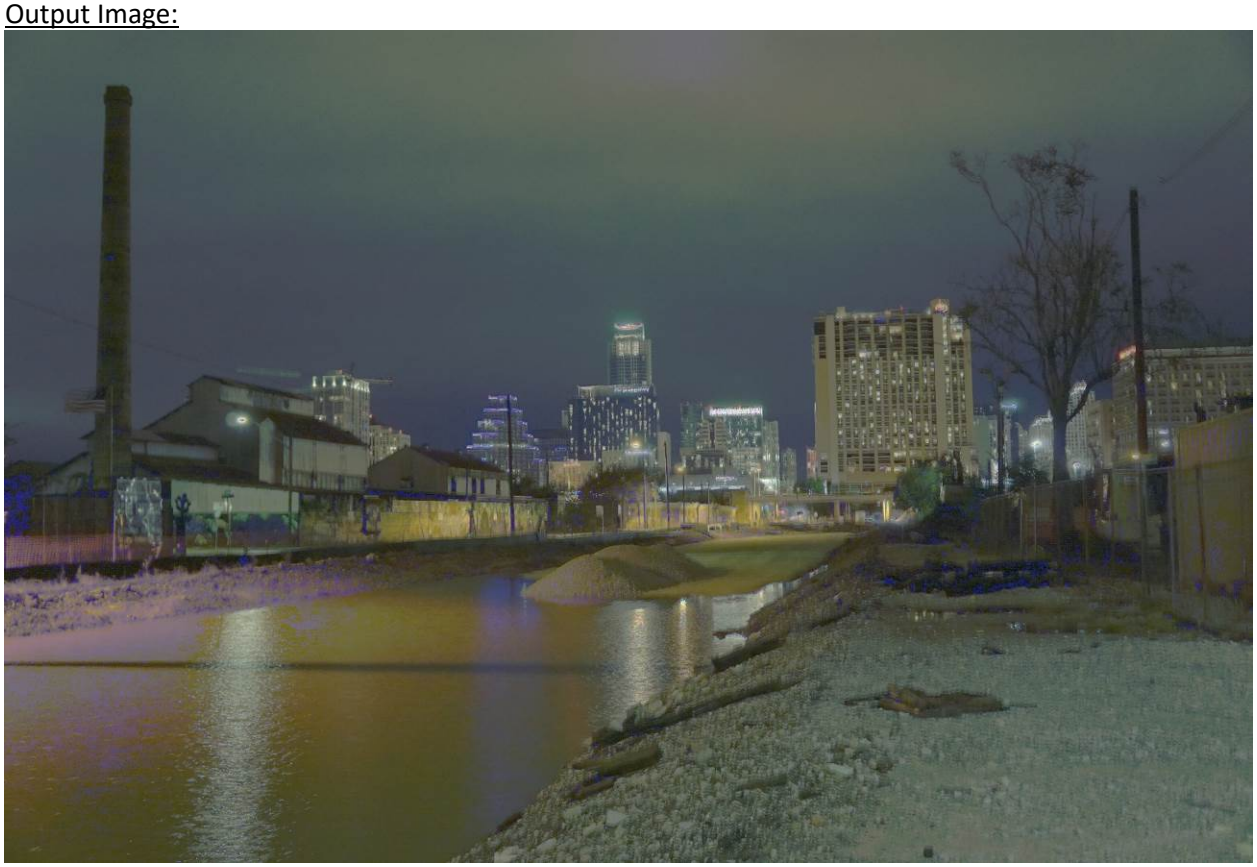
1/5



1/2



Output Image:



The following functions were implemented for this assignment.

1. normalizeImage()
   This function normalizes an image from any range to 0-255.

```
def normalizeImage(img):
    out = np.zeros(img.shape, dtype=np.float)
    out = img - np.amin(img)
    out = out * (255/np.amax(out))
    return np.uint8(out)
```

2. linearWeight()
   This function returns a linear weight based on an input pixel value.

```
def linearWeight(pixel_value):
    pixel_range_min = 0.0
    pixel_range_max = 255.0
    pixel_range_mid = 0.5 * (pixel_range_min + pixel_range_max)
    weight = 0.0
    if (pixel_value > pixel_range_mid):
        weight = pixel_range_max - pixel_value
    else:
        weight = np.float(pixel_value)
    return weight
```

3. getYXLocations()
   This function returns the Y, X locations of an image at a certain intensity value.

```
def getYXLocations(image, intensity_value):
    return np.int64(np.where(image == intensity_value))
```

4. computeResponseCurve()
   This function returns the camera response curve for one color channel by performing a single value decomposition (SVD), given images at different exposures for a single channel.

```
def computeResponseCurve(pixels, log_exposures, smoothing_lambda, weighting_function):
    pix_range = pixels.shape[0]
    num_images = len(log_exposures)
    # pix_range * 2 equates to range of pixels + number of unique pixels.
    mat_A = np.zeros((num_images * pix_range + pix_range - 1, pix_range * 2),
                     dtype=np.float64)
    mat_b = np.zeros((mat_A.shape[0], 1), dtype=np.float64)

    # Create data-fitting equations
    idx_ctr = 0
    for i in xrange(pix_range):
        for j in xrange(num_images):
            wij = weighting_function(pixels[i, j])
            mat_A[idx_ctr, pixels[i,j]] = wij
            mat_A[idx_ctr, pix_range+i]  = -wij
            mat_b[idx_ctr, 0] = wij * log_exposures[j]
            idx_ctr = idx_ctr + 1

    # Apply smoothing lambda throughout the pixel range.
    idx = pix_range * num_images
    for i in xrange(pix_range - 1):
        mat_A[idx + i, i] = smoothing_lambda * weighting_function(i)
        mat_A[idx + i, i + 1] = -2 * smoothing_lambda * weighting_function(i)
        mat_A[idx + i, i + 2] = smoothing_lambda * weighting_function(i)

    # Adjust color curve by setting its middle value to 0
    mat_A[-1, (pix_range / 2) + 1] = 0

    # Solve the system using Single Value Decomposition
```

```
    mat_A_inv = np.linalg.pinv(mat_A)
    x = np.dot(mat_A_inv, mat_b)
    g = x[0:pix_range]
    return g[:,0]
```

The following functions were already implemented for me as part of the assignment, but they are included in this report for completeness.

5. readImages()
   This function reads in images from a source directory. It can also reduce the images' size by 4 to improve execution time.

```
def readImages(image_dir, resize=False):
    file_extensions = ["jpg", "jpeg", "png", "bmp", "tif", "tiff"]
    # Get all files in folder.
    image_files = sorted(os.listdir(image_dir))
    # Remove files that do not have the appropriate extension.
    for img in image_files:
        if img.split(".")[-1].lower() not in file_extensions:
            image_files.remove(img)

    # Read in the gray and color images.
    num_images = len(image_files)
    images = [None] * num_images
    images_gray = [None] * num_images
    for image_idx in xrange(num_images):
        images[image_idx] = cv2.imread(os.path.join(image_dir,
                                                    image_files[image_idx]))
        images_gray[image_idx] = cv2.cvtColor(images[image_idx],
                                              cv2.COLOR_BGR2GRAY)
        if resize:
            images[image_idx] = images[image_idx][::4,::4]
            images_gray[image_idx] = images_gray[image_idx][::4,::4]
    return images, images_gray
```

6. computeHDR()
   This function performs the HDR computation. First, it reads in the input images and defines some defaults. Then gets random YX locations for each intensity value. Response curves are computed for each channel, which can then be used to build an image radiance map. This map is normalized and returned as an HDR image. Note that this function does not include "tone mapping", which allows the output image to have brighter tones.

```
def computeHDR(image_dir, log_exposure_times, smoothing_lambda = 100,
               resize = False):
    # STEP 1: Read in images from provided directory.
    images, images_gray = readImages(image_dir, resize)
    num_images = len(images)

    # STEP 2: Define default values.
    pixel_range_min = 0.0
    pixel_range_max = 255.0
    pixel_range_mid = 0.5 * (pixel_range_min + pixel_range_max)
    num_points = int(pixel_range_max + 1)
    image_size = images[0].shape[0:2]

    # Obtain the number of channels from the image shape.
    if len(images[0].shape) == 2:
        num_channels = 1
        logging.warning("WARNING: This is a single channel image. This code " +\
                        "has not been fully tested on single channel images.")
    elif len(images[0].shape) == 3:
        num_channels = images[0].shape[2]
    else:
        logging.error("ERROR: Image matrix shape is of size: " +
                      str(images[0].shape))
```

```
locations = np.zeros((256, 2, 3), dtype=np.uint16)
# STEP 3: For each channel:
for channel in xrange(num_channels):
    for cur_intensity in xrange(num_points):
        # Choose middle image for YX locations.
        mid = np.round(num_images / 2)
        # STEP 3a: Choose random pixels for each intensity value.
        y_locs, x_locs = getYXLocations(images[mid][:,:,channel],
                                        cur_intensity)
        if len(y_locs) < 1:
            # This is okay, finding every single intensity is not
            # necessary and probably not available in some images.
            logging.info("Pixel intensity: " + str(cur_intensity) +
                         " not found.")
        else:
            # Random y, x location.
            random_idx = random.randint(0, len(y_locs) - 1)

            # Pick a random current location for that intensity.
            locations[cur_intensity, :, channel] = y_locs[random_idx], \
                                                   x_locs[random_idx]

# Pixel values at pixel intensity i, image number j, channel k
intensity_values = np.zeros((num_points, num_images, num_channels),
                            dtype=np.uint8)
for image_idx in xrange(num_images):
    for channel in xrange(num_channels):
        intensity_values[:, image_idx, channel] = \
            images[image_idx][locations[:, 0, channel],
                              locations[:, 1, channel],
                              channel]

# Compute Response Curves
response_curve = np.zeros((256, num_channels), dtype=np.float64)

for channel in xrange(num_channels):
    response_curve[:, channel] = \
        computeResponseCurve(intensity_values[:, :, channel],
                             log_exposure_times,
                             smoothing_lambda,
                             linearWeight)

# Compute Image Radiance Map
# This maps a specific pixel value (from 0->255 to its new radiance value).
img_rad_map = np.zeros((image_size[0], image_size[1], num_channels),
                       dtype=np.float64)

for row_idx in xrange(image_size[0]):
    for col_idx in xrange(image_size[1]):
        for channel in xrange(num_channels):
            pixel_vals = np.uint8([images[j][row_idx, col_idx, channel] \
                                   for j in xrange(num_images)])
            weights = np.float64([linearWeight(val) \
                                  for val in pixel_vals])
            sum_weights = np.sum(weights)
            img_rad_map[row_idx, col_idx, channel] = np.sum(weights * \
                (response_curve[pixel_vals, channel] - log_exposure_times))\
                / np.sum(weights) if sum_weights > 0.0 else 1.0

hdr_image = np.zeros((image_size[0], image_size[1], num_channels),
                             dtype=np.uint8)

for channel in xrange(num_channels):
    hdr_image[:, :, channel] = \
        np.uint8(normalizeImage(img_rad_map[:, :, channel]))

return hdr_image
```