MIT EECS 6.815/6.865: Assignment 2:

Convolution and the Bilateral Filter

Due Wednesday September 25th at 9pm

1 Summary

Important: This problem set is longer than the previous two.
Reading the slides carefully can easily save you some headaches!

The functions from the previous psets have been move to `basicImageManipulation.cpp`.

- box filter
- convolution
- gradient magnitude
- separable Gaussian blur
- comparison between 2D and separable Gaussian blur
- unsharp mask
- denoising with the bialteral filter
- *6.865 only*: YUV denoising

2 Smart Accessor

When resampling images or performing neighborhood operations, we might end up accessing a pixel at `x,y` coordinates that are outside the image. To handle such cases gracefully, it is good practice to write accessors that take `x,y` as inputs and check them against the bounds of the image before retrieving a value.

Multiple options are possible when a pixel is requested outside the bounds; but the most common are to return a black pixel or the value of the closest valid pixel (i.e. of the closest edge). For the latter, just clamp the pixel coordinates to `[0..height-1]` and `[0..width-1]` and perform the lookup there.

Black padding (a.k.a. zero-padding) is more appropriate for applications such as scale and rotation (which we'll see in the next assignment) whereas edge-pixel padding generally looks better for convolution.

1 Implement `float Image::smartAccessor(int x, int y, int z, bool clamp=false) const` in `Image.cpp`. The boolean `clamp` allows you to switch between clamping the coordinates and returning a black pixel.

3 Blurring

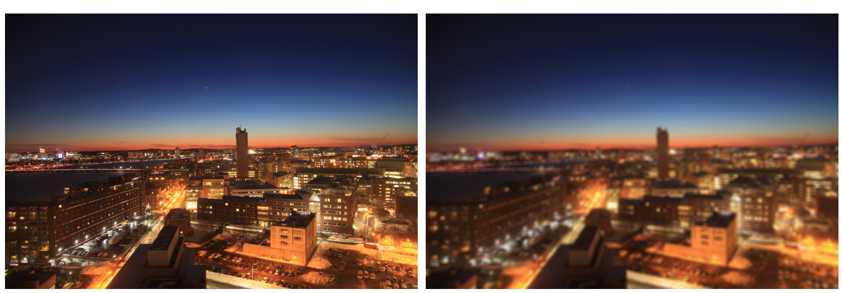
In the following problems, you will implement several functions in which you will convolve an image with a kernel. This will require that you index out of the bounds of the image. Handle these boundary effects with the smart accessor from Section 2 Also, process each of the three color channels independently.

We have provided you with a function `Image impulseImg(int k)` that generates a $k \times k \times 1$ grayscale image that is black everywhere except for one pixel in the center that is completely white. If you convolve a kernel with this image, you should get a copy of the kernel in the center of the image. An example of this can be seen in in `a2_main.cpp`. It can also be useful to have simple images, such as an all white Image e.g. `Image::set_color(1.0, 1.0,1.0)`.

3.1 Box blur

2.a Implement the box filter `Image boxBlur(const Image &im, int k, bool clamp=true)` in `filtering.cpp` . Each pixel of the output is the average of its $k \times k$ neighbors in the input image, where k is an integer. Make sure the average is centered. We will only test you on odd k.

2.b Can you think of ways to make the Box Blur filter computationnally more efficient? (Answer in the submission form)



(a) Original Image (b) Box Blurred Image

Figure 1: Result of blurring an image with a box width of `k=9` and `clamp=true`

3.2 General kernels

Now, let us implement a more general convolution function that uses an arbitary kernel. From now on, we'll perform using the `Filter` class. This class contains a buffer of float that represent the filtering kernel, together with its spatial footprint.

To create a filter use the constructor `Filter(const vector<float> &fData, int fWidth, int fHeight)`. This takes in a row-major vector containing the values of the kernel (just like in the `Image` class), and the width and height of the kernel respectively (to avoid alignment and coordinate round-off issues, we'll use odd width and height). See `a2_main.cpp` for an example.

3.a Implement the function `Image Convolve(const Image &im, bool clamp=true)` in the `Filter` class inside `filtering.cpp`. This function should compute the *convolution* of an input image by its kernel. Make sure the convolution is centered. Remember that in convolution (as opposed to correlation) you must flip your kernel. **Tip**: To access the `(x,y)` location in a kernel from within the class type `operator()(x, y)`.

3.b Implement the box filter using the `Convolve` method of the `Filter` class in `Image boxBlur_filterClass(const Image &im, const int &k, bool clamp=true)`. Check that you get the same answer as before with `boxBlur`.

Pay attention to indexing, $(0,0)$ denotes the upper left corner of the Image, but for our kernels we want the center to be in the middle. This means you might need to shift indices by half the kernel size. Test your function with `impulseImg`, a constant image and real images of your choice.

3.3 Gradients

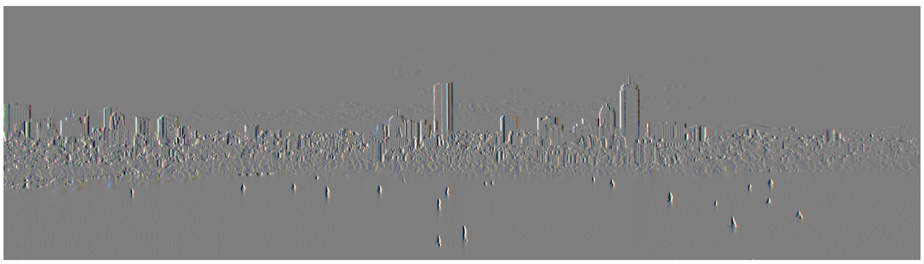
Image gradients measure the variation in intensity between neighboring pixels. We estimate gradients at each of pixel in the image using finite differences. Since this operation is identical regardless of the spatial location, we can write it as a convolution. A kernel often used to estimate the gradient is the Sobel kernel. Here are their weights for the horizontal and vertical components of the gradient are respectively:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \text{ and } \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

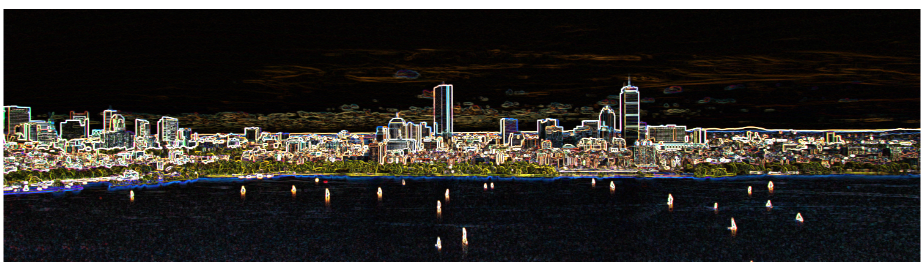
The gradient magnitude is defined as the square root of the sum of the squares of the two components:

$$||\nabla I|| = \sqrt{\left|\frac{\partial I}{\partial x}\right|^2 + \left|\frac{\partial I}{\partial y}\right|^2}\,. \tag{1}$$

(a) Original Image



(b) Image Filtered Using a Horizontal Sobel Kernel



(c) Gradient Magnitude of the Image

Figure 2: Result of filtering with a horizontal Sobel kernel and computing the gradient magnitude.

4.a Write a function `Image gradientMagnitude(const Image &im, bool clamp=true)` that uses the Sobel kernel to compute the gradient magnitude from the horizontal and vertical components of the gradient of an image.

4.b What kind of image structure does the gradient reveal? Can you imagine an application that would leverage this information? (Answer in the submission form)

3.4 Gaussian Filtering

The Gaussian filter is ubiquitous in image processing. It is a key building block of many algorithms. We'll start by implementing a one-dimensional version of the Gaussian filter, then the 2D extension. The mathematical expression for a continuous normalized 1D Gaussian is:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{x^2}{2\sigma^2}\right) \tag{2}$$

Notice that $\frac{1}{\sqrt{2\pi\sigma^2}}$ is the normalization factor.

In practice, we will not care about this exact normalization factor since we are working with truncated and discrete Gaussians. Instead, we'll make sure our *discrete* filter is normalized, i.e. its weights sum to one.

3.4.1 1D Horizontal Gaussian filtering

5.a Implement `vector<float> gauss1DFilterValues(float sigma, float truncate)` that returns the kernel values of a 1 dimensional Gaussian of standard deviation `sigma`. Gaussians have infinite support (they accept any x from $-\infty$ to ∞), but their energy falls off so rapidly that you can truncate them at `truncate` times the standard deviation `sigma`. Make sure that your truncated kernel is normalized to sum to 1. Your kernel's output length should be `1+2*ceil(sigma * truncate)`.

5.b Use the returned vector from `gauss1DFilterValues` to generate a 1D horizontal Gaussian kernel using the `Filter` class. Create and use this `Filter` to blur an image horizontally in `Image gaussianBlur_horizontal(const Image &im, float sigma, float truncate=3.0, bool clamp=true)`

5.c What is the effect on the output of varying sigma? (Answer in the form)

5.d How does `truncate` impact the computational cost? (Answer in the form)

3.4.2 2D Gaussian Filtering

Let us now extend our skinny 1D Gaussian to a the full-blown 2D version. e.g.

$$f(x, y) \propto \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right) . \tag{3}$$

6.a Implement a function `vector<float> gauss2DFilterValues(float sigma, float truncate)` that returns a full 2D rotationally symmetric Gaussian kernel. The kernel should have standard deviation `sigma` corresponding to a size of `1+2*ceil(sigma * truncate)` $\times$ `1+2*ceil(sigma * truncate)` pixels.

6.b Implement `Image gaussianBlur_2D(const Image &im, float sigma, float truncate=3.0, bool clamp=true)` that uses the kernel from `gauss2DFilterValues` to filter an image.

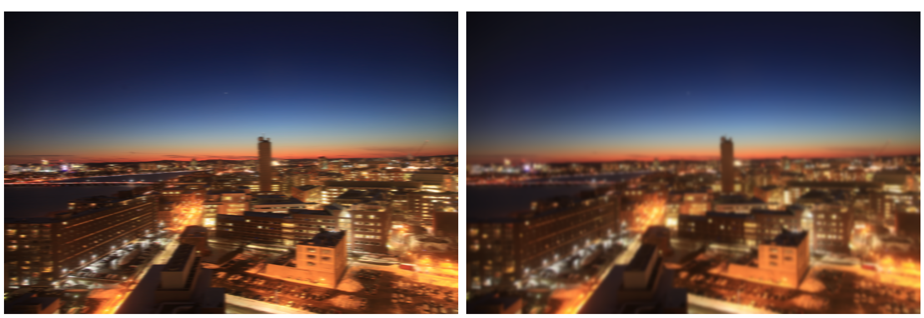
6.c How does the output differ qualitatively from the BoxBlur filter? (Answer in the form)

3.4.3 Separable 2D Gaussian Filtering

Filtering our images with a 2D Gaussian kernel gives us a nice blur effect, but it is computationnally costly. Fortunately we can exploit the rotational symmetry of the Gaussian kernel, and make the 2D convolution almost as fast as its 1D counterpart.

7 Implement separable Gaussian filtering in `Image gaussianBlur_seperable(const Image &im, float sigma, float truncate=3.0, bool clamp=true)`, using a 1D horizontal Gaussian filter followed by a 1D vertical one.

Verify that you get the same result with the full 2D filtering as with the separable Gaussian filtering. You can compare the running time of your two implementations following the example in the starter code in `a2_main.cpp`.



(a) Image Filtered Using a Horizontal Gaussian Kernel (b) Image Filtered Using a 2D Gaussian Kernel

Figure 3: Result of blurring an image with a horizontal and 2D Gaussian kernel for `sigma = 3.0, truncate=3.0, clamp=true`

3.5 Sharpening

8.a Implement `Image unsharpMask(const Image &im, float sigma, float truncate=3.0, float strength=1.0, bool clamp=true)` to sharpen an image. Use a Gaussian of standard deviation `sigma` to extract a lowpassed version of the image. Subtract that lowpassed version from the original image to produce a highpassed version of the image and then add the highpassed version back to it `strength` times.

8.b What is the influence of the strength parameter? How about sigma? How would you set sigma to emphasize very small-scale details? (Answer in the form)

4 Denoising using Bilateral Filtering

The Bilateral filter is defined as:

$$I_{\text{out}}(x,y) = \frac{1}{Z}\sum_{x',y'} G([x-x', y-y'], \sigma_D) \times G(\|I(x,y) - I(x',y')\|, \sigma_R) \times I(x',y')$$

$$\text{with } Z = \sum_{x',y'} G([x-x', y-y'], \sigma_D) \times G(\|I(x,y) - I(x',y')\|, \sigma_R)$$

where I is the input image, Z is a normalization factor and G is a Gaussian kernel. The notation $G([x-x', y-y'], \sigma_D)$ is shorthand for $G(x-x', \sigma_D) \times G(y-y', \sigma_D)$. The bilateral filter is very similar to a convolution, but the "kernel" varies spatially and depends on the color difference between a pixel and its neighbors. The different standard deviations trade-off the spatial (domain), σ_D, and color differences (range), σ_R. It is important to note that unlike other kernels, the normalization here depends on the pixel values and cannot be computed ahead of time. The range Gaussian should be computed using the Euclidean distance in RGB, that is $\|I(x,y) - I(x',y')\|$ above is a shorthand for $\sqrt{\sum_c (I(x,y,c) - I(x',y',c))^2}$.

Warning: Because this is not a straightforward convolution, you will not be able to use the `Filter` class implementation for this part.

9.a Implement `Image bilateral(const Image &im, float sigmaRange=0.1, float sigmaDomain=1.0, float truncateDomain=3.0, bool clamp=true)`, that filters an image using the bilateral filter.

9.b **(extra)** The Bilateral filter is often called an "edge-preserving" filter, do you have any intuition why that is? (Answer in the submission)

Try your filter on the provided noisy image `lens` as well as on simple test cases (e.g. an image with a rectangle).

(a) Original Image `im` (b) RGB Bilaterial Filtering (c) YUV Bilaterial Filtering

Figure 4: Results of denoising using a bilaterial filter. RGB Bilaterial Filtering: `bilateral(im, 0.1,1.0,3.0,true);` YUV Bilaterial Filtering: `bilaYUV(im, 0.1,1.0,4.0,3.0,true);`

4.1 6.865 only (or 5% Extra Credit): YUV version

We want to avoid chromatic artifacts by filtering chrominance more than luminance. This is because the human visual system is more sensitive to low frequencies in the chrominance components.

10 Implement `Image bilaYUV(const Image &im, float sigmaRange=0.1, float sigmaY=1.0, float sigmaUV=4.0, float truncateDomain=3.0, bool clamp=true)` that performs bilateral denoising in YUV where the Y channel gets denoised with a different domain sigma than the U and V channels.

In all cases, make sure you compute the range Gaussian with respect to the full YUV coordinates, and not just for the channel you are filtering. We recommend a spatial sigma four times bigger for U and V as for Y.

5 Extra credit

Here are ideas for extensions you could attempt, for 5% each. At most, on the entire assignment, you can get 10% of extra credit:

- Median filter
- Summed Area tables (i.e. integral images) for faster box blur
- Different edge padding (e.g. mirroring)
- Fast incremental and separable box filter
- Use a look-up table to accelerate the computation of Gaussian values for bilateral filtering.
- Edge detection
- Smart sharpen (e.g. based on pixel value or edge detection)
- Difference of Gaussian with stylistic modifications https://hpi.de/fileadmin/user_upload/fachgebiete/doellner/publications/2012/WKO12/winnemoeller-cag2012.pdf
- Fast convolution using recursive Gaussian filtering:http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.35.5904
- Denoising using NL means http://www.math.ens.fr/culturemath/maths/mathapli/imagerie-Morel/Buades-Coll-Morel-movies.pdf
- Correcting for dark value bias in denoising (denoised images tend to be too bright because the negative component of noise is missing)

6 Submission

Turn in your files to the online submission system and make sure all your files are in the `asst` directory under the root of the zip file. If your code compiles on the submission system, it is organized correctly. The submission system will run code in your main function, but we will not use this code for grading. The submission system should also show you the image your code writes to the `./Output` directory

In the submission system, there will be a form in which you should answer the following questions:

- How long did the assignment take? (in minutes)
- Potential issues with your solution and explanation of partial completion (for partial credit)
- Any extra credit you may have implemented and their function signatures if applicable
- Collaboration acknowledgment (you must write your own code)
- What was most unclear/difficult?

- What was most exciting?