6.815/6.865: Assignment 3:

Convolution and the Bilateral Filter

Due Wednesday March 4th at 9pm

1 Summary

- 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

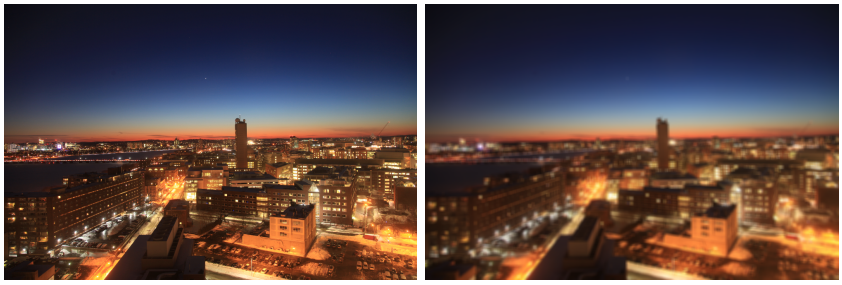
2 Blurring

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 by using the smartAccessor from problem set 2: `float smartAccessor(int x, int y, int z, bool clamp=false) const`. Also, process each of the three color channels in an image independently.

We have provided you with a function `Image impulseImg(const 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 `testGradient()` (in `a3_main.cpp`).

2.1 Box blur

2.1.1. Implement the box filter `Image boxBlur(const Image &im, const int &k, bool clamp=true)` in `filtering.cpp` . This function takes in an Image and an integer as input and outputs for each pixel the average of its $k \times k$ neighbors, where k is an integer. Make sure the average is centered. We will only test you on odd k.



(a) Original Image (b) Box Blurred Image

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

2.2 General kernels

Now, you will implement a more general convolution function that uses an arbitary kernel. The kernel is an instance of the `Filter` class. To create a kernel use the constructor `Filter(const vector<float> &fData, const int &fWidth, const 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 (width and height must be odd). See `a3_main.cpp` for example kernels.

2.2.1. 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 you must flip your kernel. Hint: To access the `(x,y)` location in a kernel from within the class type `operator()(x, y)`.

2.2.2. 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 `impulse`, a constant image and real images.

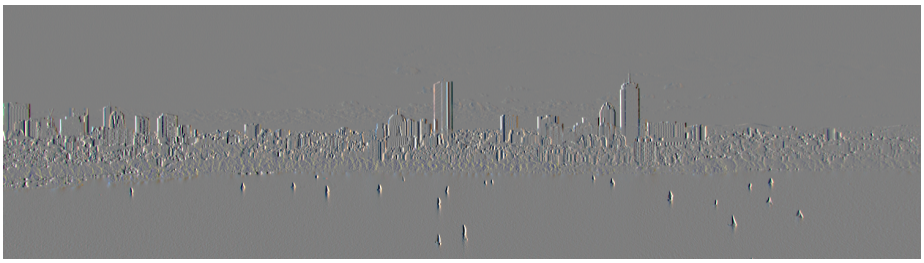
2.3 Gradients

2.3.1. Write a function `Image gradientMagnitude(const Image &im, bool clamp=true)` that uses the provided Sobel kernel in `testGradient()` (in `a3_main.cpp`) to compute the gradient magni-

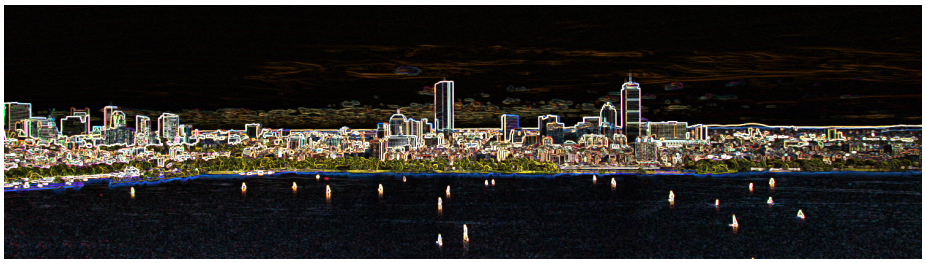
tude from the horizontal and vertical components of the gradient of an image. The gradient magnitude is defined as the square root of the sum of the squares of the two components. The Sobel kernels for the horizontal and vertical components of the gradient are respectively

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

(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.

2.4 Gaussian Filtering

2.4.1 1D Horizontal Gaussian filtering

2.4.1. 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, 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)`.

2.4.2. 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)`

2.4.2 2D Gaussian Filtering

2.4.3. 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.

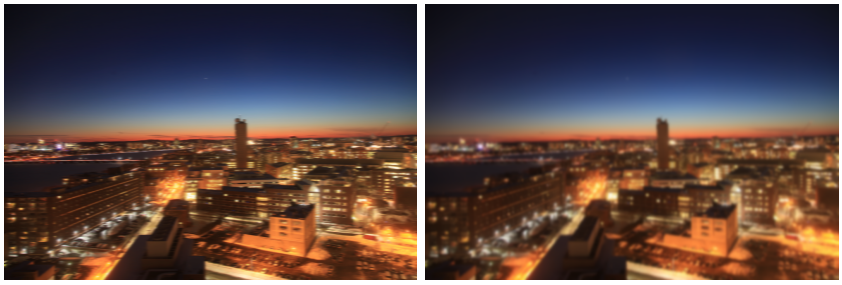
2.4.4. Implement `Image gaussianBlur_2D(const Image &im, float sigma, float truncate=3.0, bool clamp=true)` that uses the kernel from `gauss2DFilterValues` to filter an image.

2.4.3 Separable 2D Gaussian Filtering

2.4.5. 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. Measure the running times of the separable filtering vs. 2D filtering using `testGaussianFilters()` in `a3_main.cpp`.

2.5 Sharpening



(a) Horizontal Gaussian Kernel (b) 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`

2.5.1. 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.

3 Denoising using bilateral filtering

3.0.1. 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. The filter is defined as

$$I_{\text{out}}(x,y) = \frac{1}{k}\sum_{x,y} G(x-x', y-y', \sigma_{\text{Domain}}) G(I_{\text{in}}(x,y) - I_{\text{in}}(x',y'), \sigma_{\text{Range}}) I_{\text{in}}(x',y')$$

$$k = \sum_{x,y} G(x-x', y-y', \sigma_{\text{Domain}}) G(I_{\text{in}}(x,y) - I_{\text{in}}(x',y'), \sigma_{\text{Range}})$$

where I_{in} is the input image, G are Gaussian kernels and k is a normalization factor. The bilateral filter is very similar to convolution, but the kernel varies spatially and depends on the color difference between a pixel and its neighbors.

The range Gaussian on $I_{\text{in}}(x,y) - I_{\text{in}}(x',y')$ should be computed using the Euclidean distance in RGB. Try your filter on the provided noisy image `lens` as well as on simple test cases.

(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);`

3.1 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.

3.1.1. 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.

4 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
- Fast incremental and separable box filter

- Fast convolution using recursive Gaussian filtering (Deriche or Vliet-Young-Verbeek approximation):http://www.ipol.im/pub/art/2013/87/
- Use a look-up table to accelerate the computation of Gaussians for bilateral filtering.
- Denoising using NL means http://www.math.ens.fr/culturemath/maths/mathapli/imagerie-Morel/Buades-Coll-Morel-movies.pdf

5 Submission

Turn in your files to the online submission system (link is on Stellar) 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.

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

- How long did the assignment take?
- 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 acknowledgement (but again, you must write your own code)
- What was most unclear/difficult?
- What was most exciting?