MIT EECS 6.815/6.865: Assignment 3:

Convolution and the Bilateral Filter

Due Wednesday Sep. 25 at 9pm

1 Summary

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

2 Blur

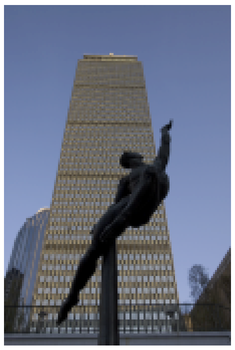
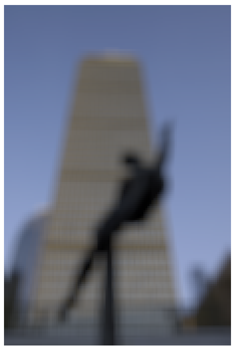
Note: numpy and scipy have built-in convolution implementations. You are **not** allowed to use them. You should write your own for loops.

We provide some test code a3_test.py to make sure your input/output type satisfy our requirement. Please uncomment the test accordingly.

2.1 Box blur

Implement a box filter `boxBlur(im, k)` that outputs for each pixel the average of its k*k neighbors, where k is an integer. Make sure it is centered (we only test for odd `k`.)

For all this assignment, we ask that you handle boundary effects using edge padding. The three color channels are treated the same, which is transparent using numpy: just use array operations.

 $\rightarrow$ `boxBlur(im, 7)` $\rightarrow$ 

For testing purposes, we provide you with a function `impulse` (in a3_test.py) that generates an image that is black everywhere except in the center.

2.2 General kernel

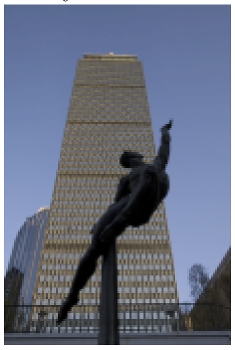
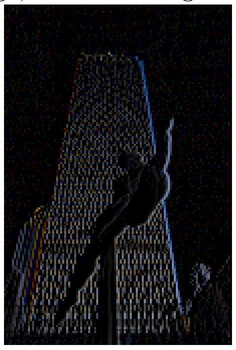
We represent convolution kernels using 2D arrays along y, x. Weights will be the same for R, G and B. See a3_test.py for examples of kernels, such as `box3, gauss3, deriv`, or `Sobel`.

Implement a function `convolve(im, kernel)` that computes the convolution of image `im` by kernel `kernel`. Assume that the kernel has already been flipped

In most cases, it won't matter because our kernels tend to be symmetric.

Pay attention to indexing: in our arrays, 0, 0 denotes the upper left corner whereas we want the center of our kernels to be in the middle. This means that you might need to shift indices by half the kernel size.

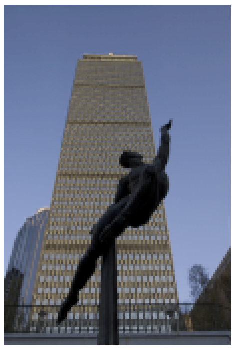
Test your function with `impulse`, a constant image, and real images.

 $\rightarrow$ `convolve(im, deriv)` $\rightarrow$ 

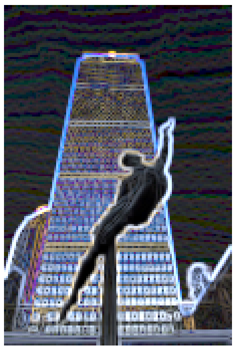
2.3 Gradient

Write a function `gradientMagnitude(im)` that uses the provided Sobel kernel to compute the horizontal and vertical components of the gradient of an image and returns the gradient magnitude. The gradient should be the square root of the sum of square of the gradient in the two directions.

Hint: the `transpose` function is quite useful to rotate kernels.



$\rightarrow$ `gradientMagnitude(im)` $\rightarrow$

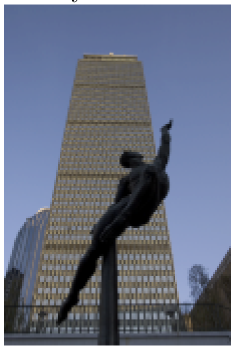


2.4 Horizontal Gaussian filtering

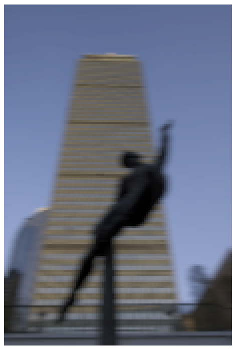
write a function `horiGaussKernel(sigma, truncate=3.0)` that returns a horizontal Gaussian kernel of standard deviation sigma. In theory, Gaussians have an infinite support, but they fall off so rapidly that we can truncate them at `truncate` times the standard deviation σ. Make sure that your kernel is normalized. Your output length should be `int(sigma*truncate)` and normalized. `sigma` should b a float number.

When computing the Gaussian, make sure you cast numbers to floats, since pixel coordinates tend to be integers, which could lead to problems when performing a division.

Try this kernel with your `convolve` function.



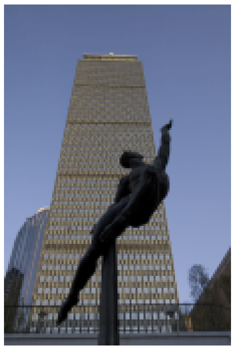
$\rightarrow$`convolve(im, horiGaussKernel(2.0, 3.0))` $\rightarrow$



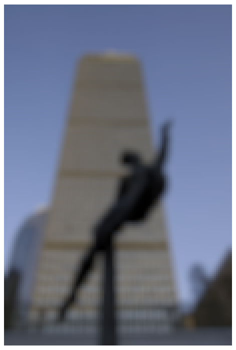
2.5 Separable Gaussian filtering

Implement separable Gaussian filtering `gaussianBlur(im, sigma, truncate=3)`, using a horizontal Gaussian followed by a vertical one.

Hint: the `transpose` function is quite useful to rotate kernels.



$\rightarrow$ `gaussianBlur(im, 2, 3)` $\rightarrow$



2.6 Verify separability and its usefulness

Next implement a function gauss2D(sigma=2, truncation=3) that returns a full 2D rotationally symmetric Gaussian kernel. Hint: there is a simple matrix algebra way involving the 1D kernel and its transpose that will give you the 2D blur. Or you can write the full expression as a function of the radius.

Verify that you get the same result with full 2D filtering and with separable Gaussian filtering.

Measure the running time with the separable vs. 2D version for sigma=3 and the zebra image. Report your findings.

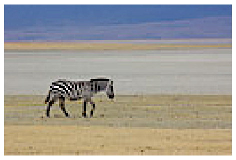
To measure running time, use:

```
import time
t=time.time()
(your code here)
print time.time()-t, 'seconds'
```

2.7 Sharpening

Write a `unsharpMask(im, sigma, truncate, strength)` function that sharpens an image using a Gaussian of standard deviation `sigma` to extract a high pass version of the image and adding it `strength` times.

$\rightarrow$ `unsharpMask(im, 1,3,1)` $\rightarrow$



3 Denoising using bilateral filtering

Implement `bilateral(im, sigmaRange, sigmaDomain)` that filters an image using the bilateral filter:

$$bila(y,x) = \frac{1}{k} \sum_{y',x'} G(y-y', x-x', \sigma_{Domain}) G(I(y,x)-I(y',x'), \sigma_{Range})\ I(y',x')$$

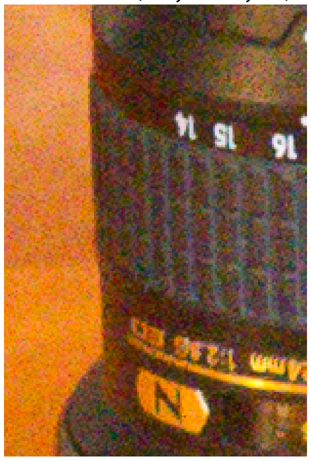
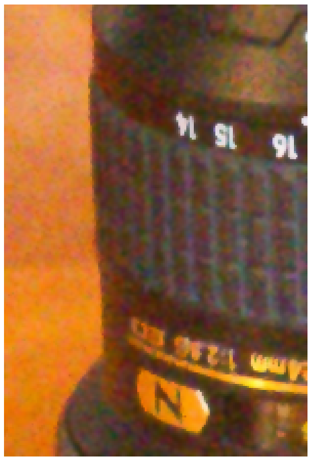
$$k = \sum_{y',x'} G(y-y', x-x', \sigma_{Domain}) G(I(y,x)-I(y',x'), \sigma_{Range})$$

where I is the input image, G are Gaussians, and k is a normalization factor. For speed, hardcode the `truncate=2`. This should be pretty much convolution. The only difference is that now the weight for neighborhood pixel depends on the color difference.

Hint: k here is the normalization factor. Hence, you compute the weight on neighborhood pixel, and then normalize by the sum of them.

The range Gaussian on $I(y,x)-I(y',x')$ should be computed using Euclidean distances in RGB.

Try your filter on the provided noisy images `lens3` and its crops, as well as on simple test cases. We recommend
`bilateral(im, 0.1, 1)` or `bilateral(im, 0.1, 1.5)` .


$\rightarrow$ `bilateral(im, 0.1, 1)` $\rightarrow$


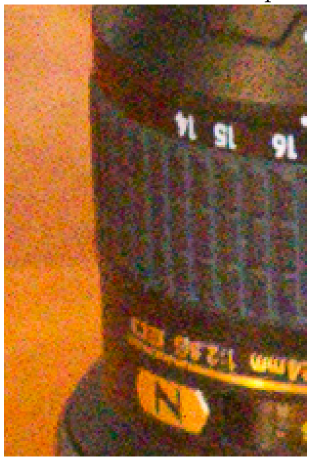
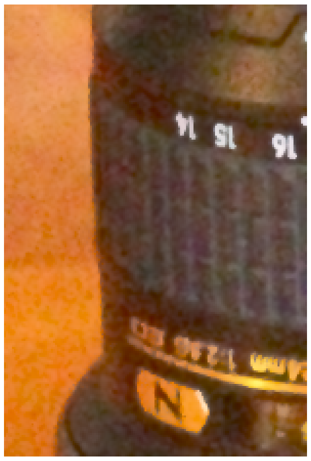
3.1 6.865 only: YUV version

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.

Implement `bilaYUV(im, sigmaRange, sigmaY, sigmaUV)` 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.

 → `bilaYUV(im, 0.1, 1, 4)` → 

4 Extra credit

Median filter

Fast incremental and separable box filter

Faster code: special case for boundary pixels in order to avoid checking array bounds everywhere.

Fake miniature tilt shift . You can use the segments from pset 2 and the magical UI to specify where focus should be.

Fast convolution using recursive Gaussian filtering
`http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.35.5904`

Use a look-up table to accelerate the computation of Gaussians for bilateral filtering.

Denoising using NL means (probably very slow in Python)
`http://www.math.ens.fr/culturemath/maths/mathapli/imagerie-Morel/Buades-Coll-Morel-movies.pd`

5 Submission

Turn in your python files and make sure all your functions can be called from a module `a3.py`. Put everything into a zip file and upload to the submission website.

For image result, we generate your results and display them as fail. It doesn't mean it's failed. You have to look at it and see if it's as expected. We use the same setting as in the `a3_test.py` The test is only to make sure your code can run our machine.

Include a `README.txt` containing the answer to the following questions:

- Q1:How long did the assignment take?

- Q2:Potential issues with your solution and explanation of partial completion (for partial credit)
- Q3:Any extra credit you may have implemented
- Q4:Collaboration acknowledgement (but again, you must write your own code)
- Q5:What was most unclear/difficult?
- Q6:What was most exciting?
- Q7:Timing for the two versions of Gaussian filtering.

Following this format: {`A1:_yout answer_`} {`A2:_yout answer_`}