Graphics Programming

Assignment 03 - Kernels

Image 1

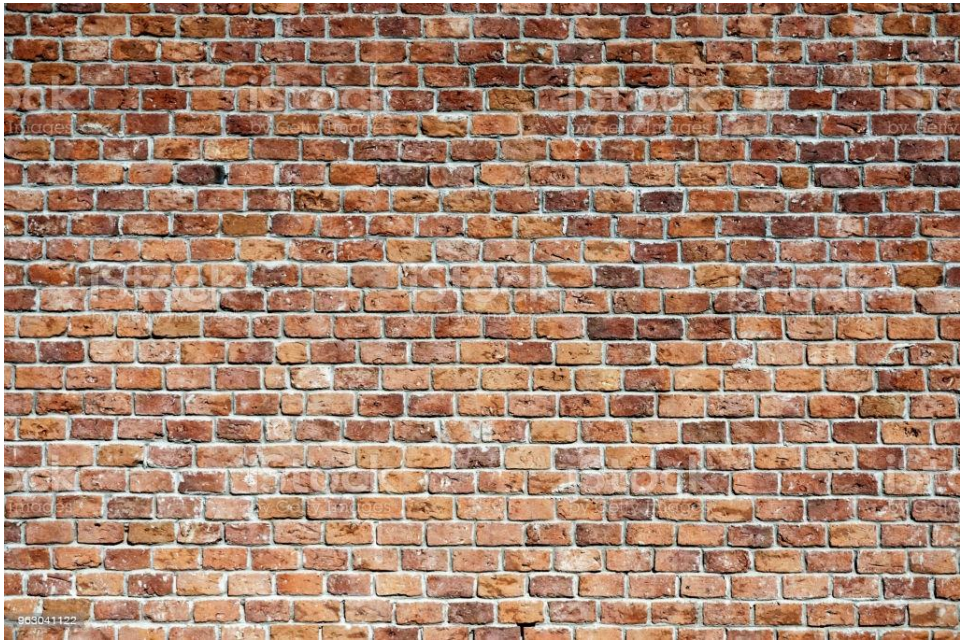


Image 2

Box Blur

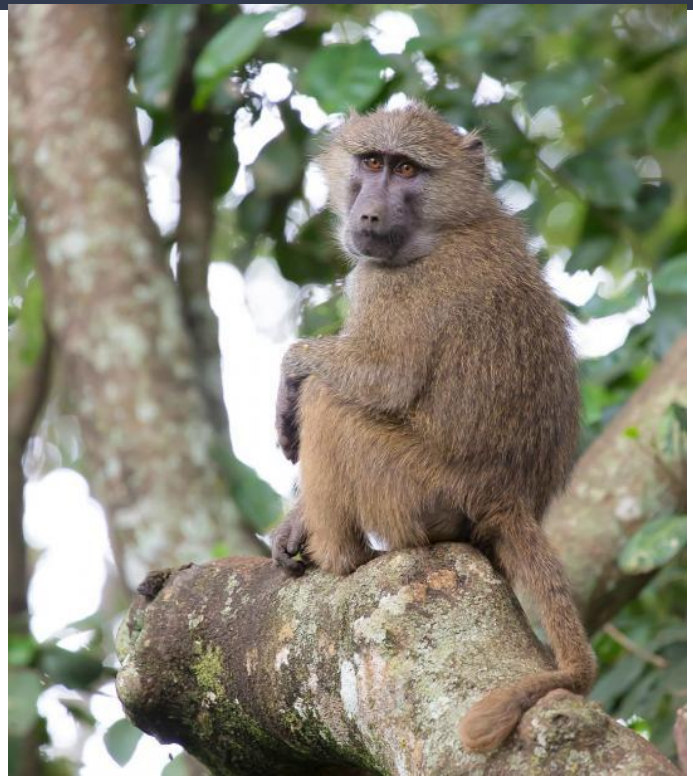
To perform a box blur, all we need is for the sum of the kernel to equal one. So for a NxN kernel, for instance, the entire thing would be divided by N^2.

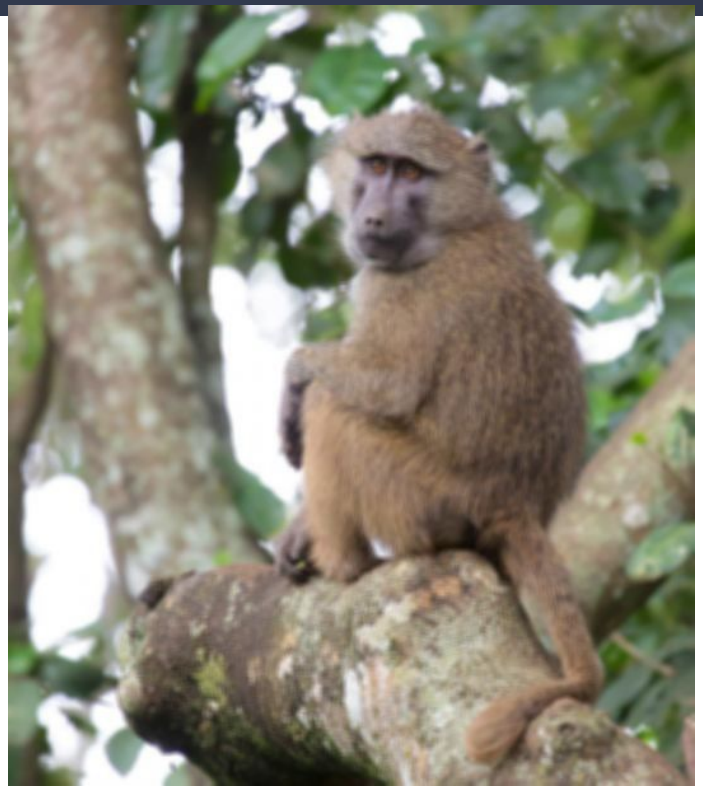
[1, 1, 1]
[1, 1, 1] * 1/9
[1, 1, 1]

That is simple to express in code, as OpenCV has a function which can create an array of all ones. All we have to do is divide it by N^2.

```
kernel = np.ones((n, n), dtype=np.float64) / n**2
```

Box blur (n=5)





Box Blur pt. 2

You can linearize box blurs by making two kernels. A Nx1 kernel and a 1xN kernel, and then applying this kernels on after the other. The output may have to be normalized, but in theory it should have the same effect as a normal box blur.

This can be tested by subtracting the original box blur from two-step blur. The average difference between the images should also be small. For box blur, the average absolute difference was about 0.5

```
    kernel1 = (np.ones((n, 1), dtype=np.float64)) / n
    kernel2 = kernel1.T
```

The first line makes an Nx1 kernel (the default is a 3x1), and the second kernel is the transpose of that. Note, here the kernel is divided by N rather than N^2.

```
    out1 = apply(img, kernel1)
    out2 = apply(out1, kernel2)
    out2 = normalize(out2)
```

```
    orig = boxBlur(img, n)
    diff = orig*1.0 - out2*1.0
```

Here id the difference code. The image was input as a float.

-

=

Blur 1

Blur 2

Gaussian Blur

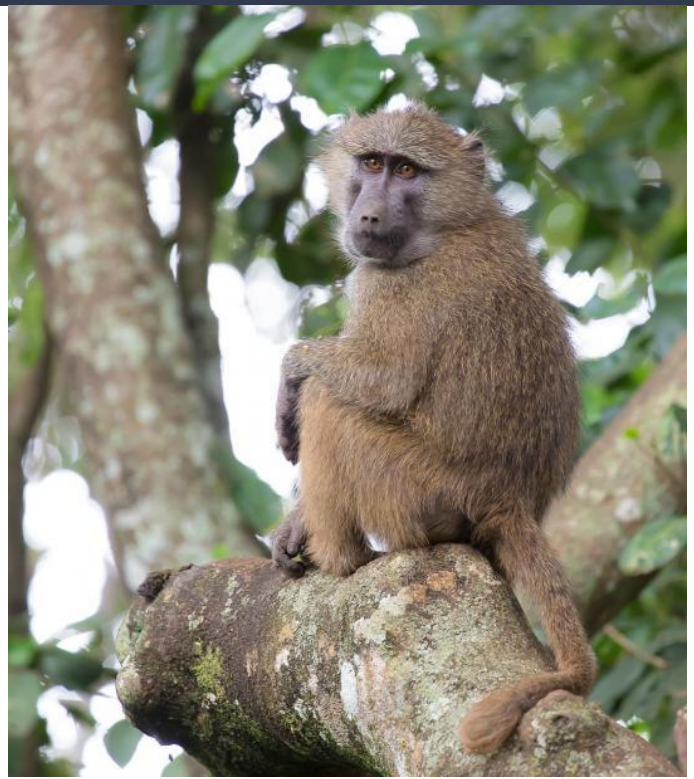
The Gaussian Blur filter is a filter that tries to make a more ‘smooth’ blurring effect. Instead of being a kernel of all 1s, you can think of it as a kernel with numbers from Pascal’s triangle. It is still divided by the sum of its values. So, for example, a 3x3 Gaussian Blur would be…

[1, 2, 1]
[2, 4, 2] * 1/16
[1, 2, 1]

```
kernel = np.zeros((n, n), dtype=np.float64)
    for row in range(len(kernel[0])):
        triangle = pascalsTriangle(n)
        kernel[row] = triangle*int(triangle[row])
    kernel /= kernel.sum()
```

A kernel with the NxN dimensions is made (default 3), then each row of that kernel is replaced with the corresponding part of Pascal’s triangle. That row is then multiplied by the entry in a certain row to create this kind of ‘circular’ pattern.

Gaussian Filter



Gaussian Blur pt. 2

You can linearize gaussian blurs by making two kernels. A Nx1 kernel and a 1xN kernel, and then applying this kernels on after the other. It should work similarly to linearizing the box blurs.

This can be tested by subtracting the original box blur from two-step blur. The average difference between the images should also be small. For this, the avg absolute difference is about 2.

```
    kernel1 = np.zeros((n, 1), dtype=np.float64)
    kernel1 = pascalsTriangle(n)
    kernel1 /= kernel1.sum()

    kernel2 = kernel1.T

    out1 = apply((img), kernel1)
    out2 = apply(out1, kernel2)
    out2 = normalize(out2)

    orig = gaussianBlur(img, n)
    diff = orig - out2
```

Gaussian Blur pt.2

Orig

Le double blur

The diff

Edge Filter

```
    kernel = np.array(([0, 1, 2], \
                       [-1, 0, 1], \
                      [-2, -1, 0]))
```

My logic was to take a normal Sobel filter, like

[1, 0, -1]

[2, 0, -2]

[1, 0, -1]

Except do that in the corners. That way it would find the diagonal edges instead of the horizontal or vertical ones.

Edge Filter

Sharpen

```
kernel = np.array(([0, 0, -1, 0, 0], \
                   [0, -1, -2, -1, 0], \
                   [-1, -2, 17, -2, -1], \
                   [0, -1, -2, -1, 0], \
                   [0, 0, -1, 0, 0], ))
```

Here is my sharpen kernel. The image was normalized after I applied the sharpen kernel. You can change the size of the matrix and values of the numbers, too.

Corners

Apparently there is no code required for this part.

But, to find corners, here is how I would approach that.

1. Find horizontal and vertical edges. I'd use two sobel filters for this.

```
    kernel1 = np.array(([-1, 0, 1], \
                        [-2, 0, 2], \
                        [-1, 0, 1]))

    kernel2 = np.array(([-1, -2, -1], \
                        [0, 0, 0], \
                        [1, 2, 1]))
```

Something like these.
2. Intersections of the edges area where the corners is. So making an image of where the two filtered images equal each other should be the corners of the image.