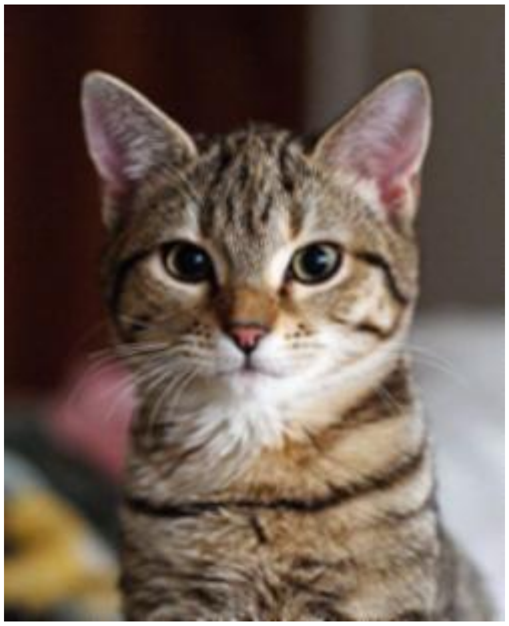
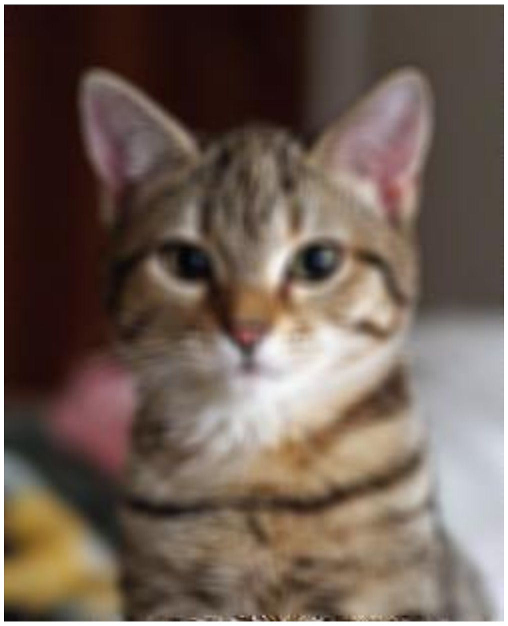
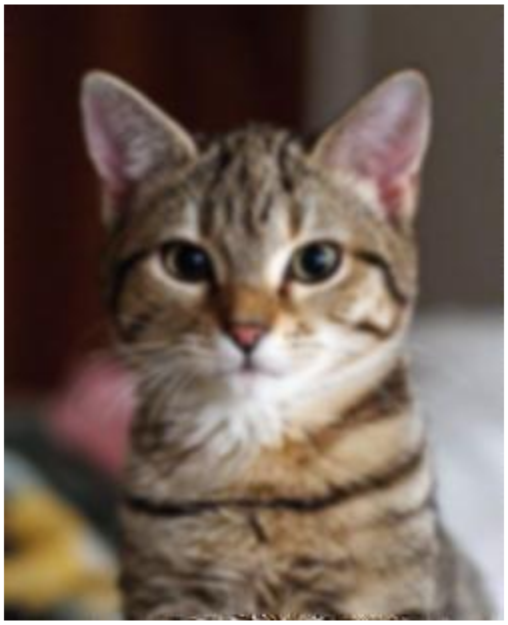
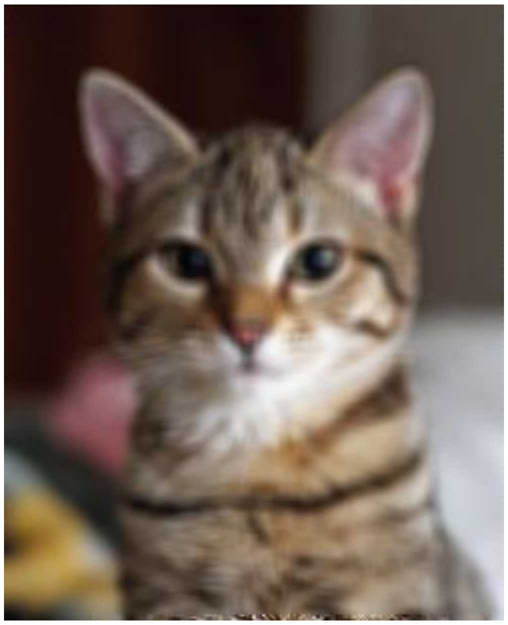
The size of the kernel seemed to affect the filter amount. A larger size kernel resulted in a greater degree of gaussian blur. Here's a comparison where the sigma value is held constant, but the size of kernel is changed:


σ=7, k=3

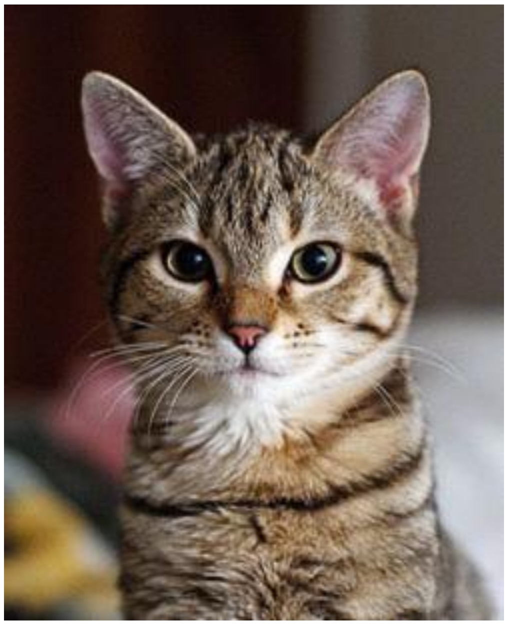

σ=7, k=7

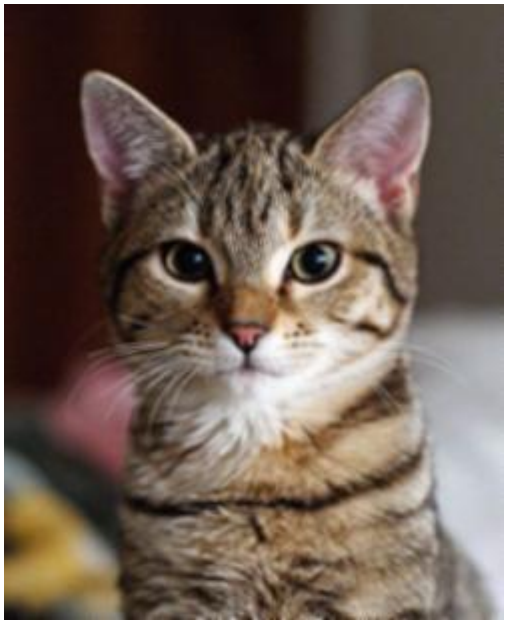
The increased blur from the larger kernel sizes removes more noise but appears to trade for image quality. The second image looks more pixelated. I also noticed that due the larger kernel size, during the convolution the kernel will go out of bounds farther away from the edge, so the non-blurred edges are more noticeable than with a smaller kernel. Rather than skipping the pixels that require values beyond the edge, using a different method of edge handling could be used to fix this such as extending, wrapping, or mirroring the edges. Using the current implementation, the output could be cropped slightly smaller to hide the edge cases. I also noticed increase sigma exponentially increases the runtime of the convolution. This makes sense because the multiplication has to be done on more elements per pixel. However, without implementing a separated kernel, this made it difficult to test kernel sizes over 7.


σ=1, k=7

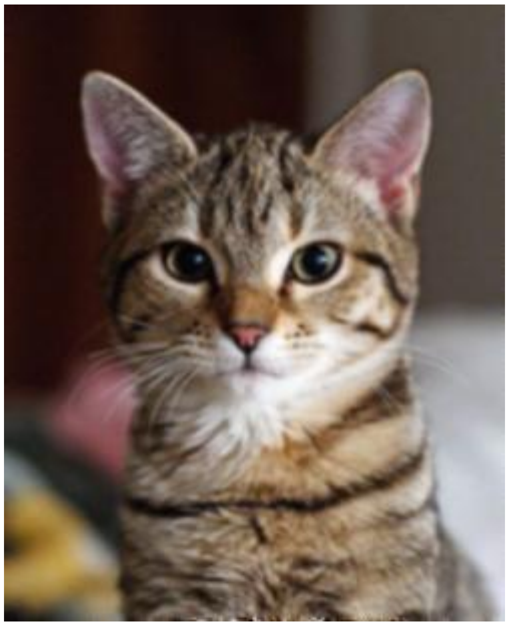

σ=7, k=7

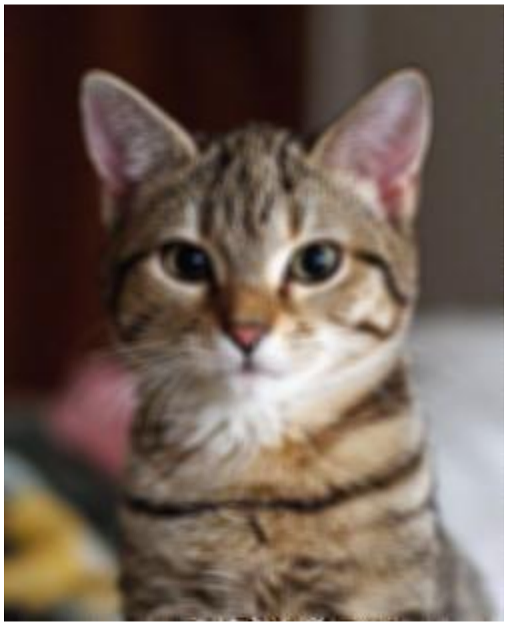
Here, the kernel size value is held constant while the sigma is changed to compare the effect of sigma on the image. A higher value of sigma increases the level of blur, without adding additional computation time.


σ=1, k=3


σ=7, k=3

The above shows, using a lower kernel size, the effect of the sigma isn't as significant, but the sigma will still increase the blurriness. The experiment below follows up on this, trying a higher k value.

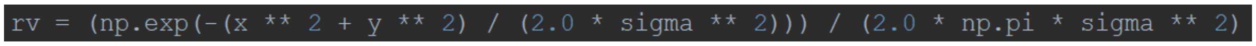

σ=1, k=5


σ=7, k=5

Comparing this set of images with the last, the value of sigma was changed the same amount, but the k value was increased. The effect of sigma is more apparent with a higher k value.

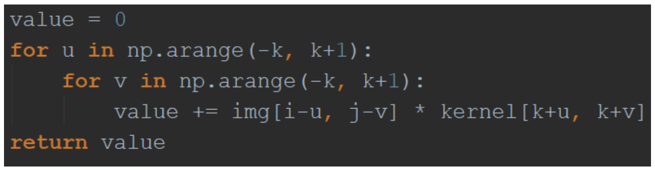
Overall, implementing the gaussian blur was not difficult. The programming aspect of it is trivial, especially with prior knowledge of numpy. The difficult aspect of the blur is the high-level concepts. Once I read through the skeleton, it made sense what needed to be done, and from there, the scariest parts were producing the kernel, and the convolution function. I tried to mimic the mathematical functions as close as possible:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$



```python
rv = (np.exp(-(x ** 2 + y ** 2) / (2.0 * sigma ** 2))) / (2.0 * np.pi * sigma ** 2)
```

$$H * F = \sum_{u=-k}^{k} \sum_{v=-k}^{k} H[u,v]F[i-u,j-v]$$



```python
value = 0
for u in np.arange(-k, k+1):
    for v in np.arange(-k, k+1):
        value += img[i-u, j-v] * kernel[k+u, k+v]
return value
```

I struggled for a bit to realize that I needed to shift up kernel[u,v] by k to account of the negative k values, but other than that, things seemed to work out.