Image Processing 2 – COMP4423 Computer Vision

Xiaoyong Wei (魏驍勇)

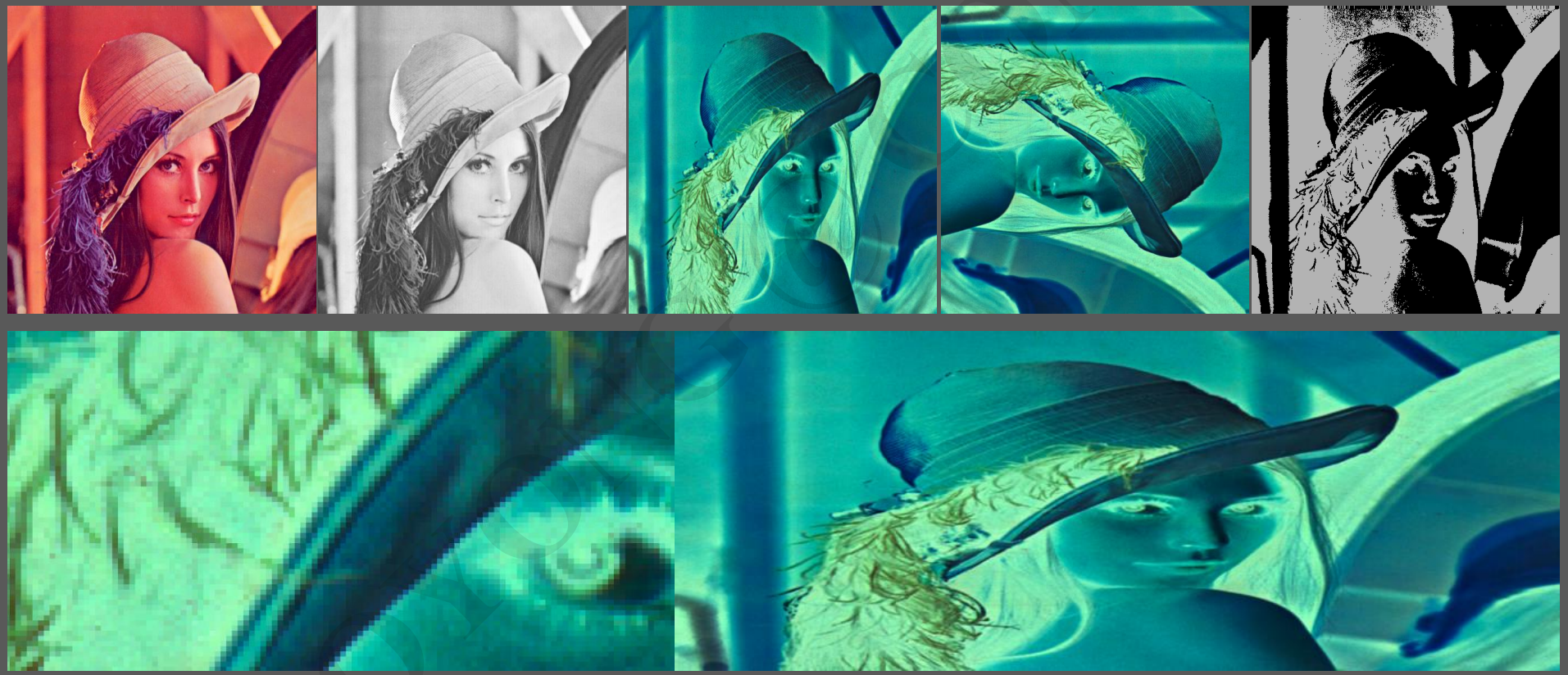
x1wei@polyu.edu.hk

Department of Computing
電子計算學系

The Hong Kong Polytechnic University
香港理工大學

Opening Minds • Shaping the Future
啟迪思維 • 成就未來

Image Processing 1



Those are for dealing with the images themselves.

Let's play with the content!

Outline

- >Filters and Convolutions
- >Edge Filters
- >Noise Reduction
- >Morphological Operations

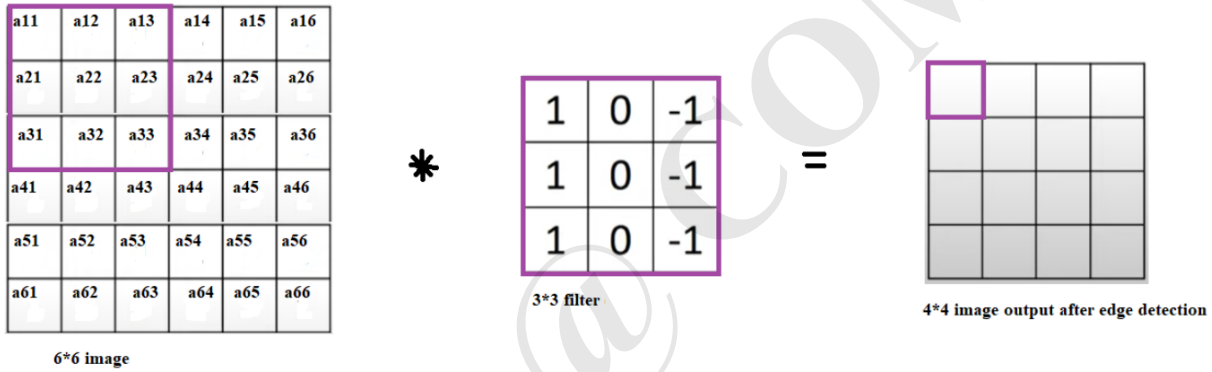
Edge Detection

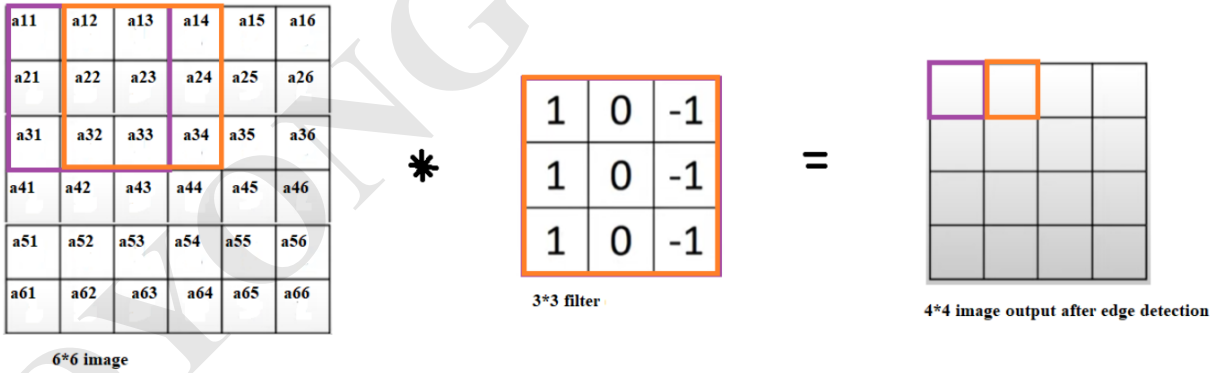
Edges in image brightness may result from the change of

1. depths,
2. surface orientations,
3. materials,
4. and/or illumination.

https://en.wikipedia.org/wiki/Edge_detection/

Filters and Convolutions





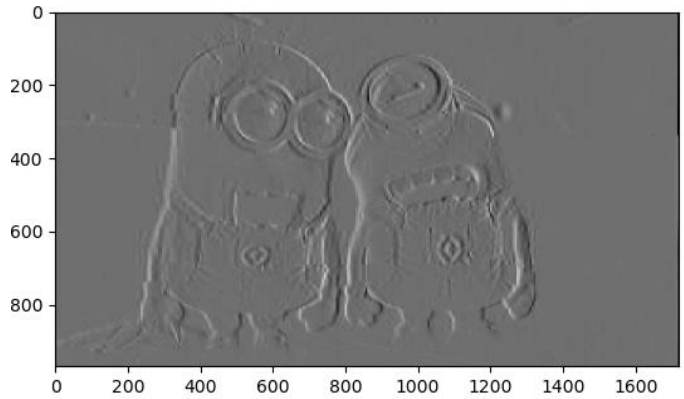
https://www.mygreatlearning.com/blog/introduction-to-edge-detection/

Prewitt Edge Detection

Prewitt is a commonly used edge detector for detecting the horizontal and vertical edges in images. Its edge filters are as follows:

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \qquad G_y = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

Prewitt filter for vetical edge detection **Prewitt filter for horizontal edge detection**



https://www.mygreatlearning.com/blog/introduction-to-edge-detection/

Sobel Edge Detection

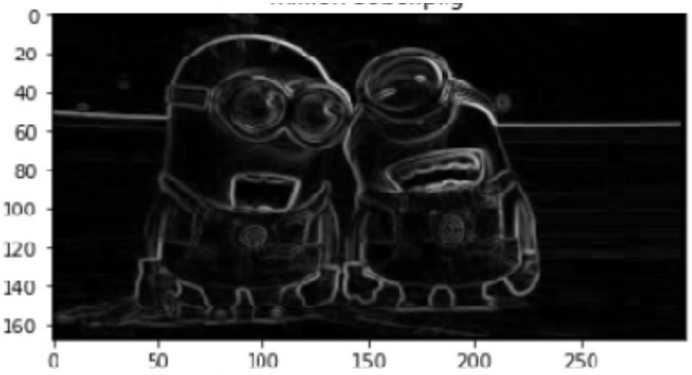
The filter pays more attention to the center of the window. It is one of the most commonly used edge detectors, which reduces noise, differentiates regions, and responses to edges simultaneously.

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

Sobel filter for vertical edge detection

$$G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Sobel filter for horizontal edge detection



https://www.mygreatlearning.com/blog/introduction-to-edge-detection/

Sobel Edge Detection

The filter pays more attention to the center of the window. It is one of the most commonly used edge detectors, which reduces noise, differentiates regions, and responses to edges simultaneously.

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

Sobel filter for vertical edge detection

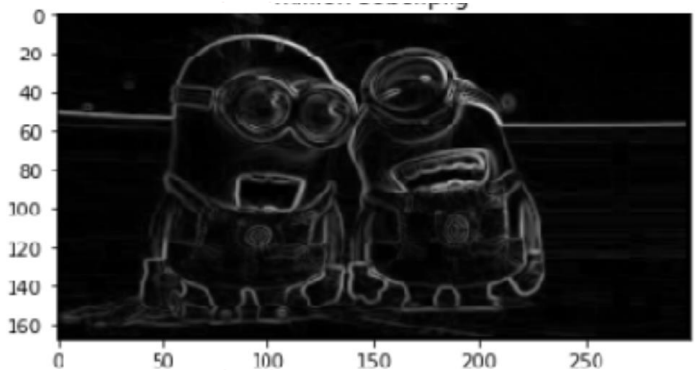
$$G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Sobel filter for horizontal edge detection

Magnitude: $|G| = \sqrt{Gx^2 + Gy^2}$

Approximation: $|G| = |Gx| + |Gy|$

Orientation: $\theta = \arctan(Gy/Gx)$



https://www.mygreatlearning.com/blog/introduction-to-edge-detection/

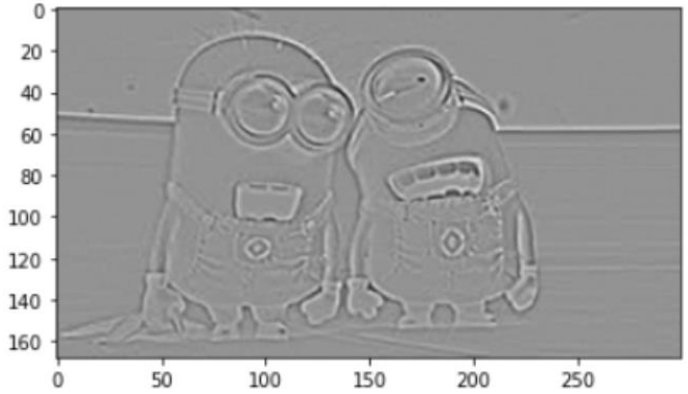
Laplacian Edge Detection

The Laplacian edge detectors uses only one filter. In a single pass, Laplacian edge detection performs second-order derivatives and hence are sensitive to noise. To avoid this sensitivity to noise, before applying this method, Gaussian smoothing can be done on the image.

−1	−1	−1
−1	8	−1
−1	−1	−1

0	−1	0
−1	4	−1
0	−1	0

Common Laplacian edge detection filters



https://www.mygreatlearning.com/blog/introduction-to-edge-detection/

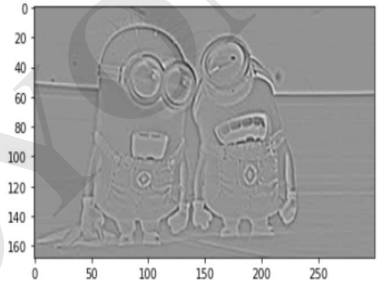
Laplacian Edge Detection

The Laplacian edge detectors uses only one filter. In a single pass, Laplacian edge detection performs second-order derivatives and hence are sensitive to noise. To avoid this sensitivity to noise, before applying this method, Gaussian smoothing can be done on the image.

−1	−1	−1
−1	8	−1
−1	−1	−1

0	−1	0
−1	4	−1
0	−1	0

Common Laplacian edge detection filters



$$\nabla f_x(x,y) = \frac{\partial f(x,y)}{\partial x}$$
$$= \lim_{\epsilon \to 0} \frac{f(x,y) - f(x+\varepsilon,y)}{\epsilon}$$
$$\approx f(x,y) - f(x+1,y)$$

$$\nabla f^2(x,y) = \nabla f_x^2(x,y) + \nabla f_y^2(x,y)$$
$$\nabla f_x^2(x,y) = \nabla f_x(x,y) - \nabla f_{x'}(x,y)$$
$$\approx f(x,y) - f(x+1,y) - \big(f(x-1,y) - f(x,y)\big)$$
$$\approx -f(x-1,y) + 2f(x,y) - f(x+1,y)$$
$$\nabla f_y^2(x,y) \approx -f(x,y-1) + 2f(x,y) - f(x,y+1)$$

https://www.mygreatlearning.com/blog/introduction-to-edge-detection/

Laplacian Edge Detection

The Laplacian edge detectors uses only one filter. In a single pass, Laplacian edge detection performs second-order derivatives and hence are sensitive to noise. To avoid this sensitivity to noise, before applying this method, Gaussian smoothing can be done on the image.

−1	−1	−1
−1	8	−1
−1	−1	−1

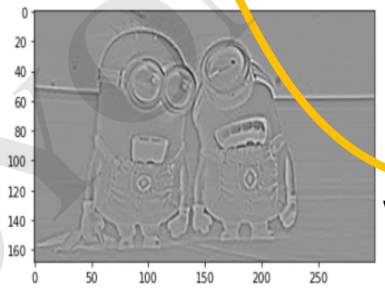
0	−1	0
−1	4	−1
0	−1	0

Common Laplacian edge detection filters

$$\nabla f_x(x,y) = \frac{\partial f(x,y)}{\partial x}$$
$$= \lim_{\epsilon \to 0} \frac{f(x,y) - f(x+\varepsilon,y)}{\epsilon}$$
$$\approx f(x,y) - f(x+1,y)$$

$$\nabla f^2(x,y) = \nabla f_x^2(x,y) + \nabla f_y^2(x,y)$$
$$\nabla f_x^2(x,y) = \nabla f_x(x,y) - \nabla f_{x'}(x,y)$$
$$\approx f(x,y) - f(x+1,y) - \big(f(x-1,y) - f(x,y)\big)$$
$$\approx -f(x-1,y) + 2f(x,y) - f(x+1,y)$$

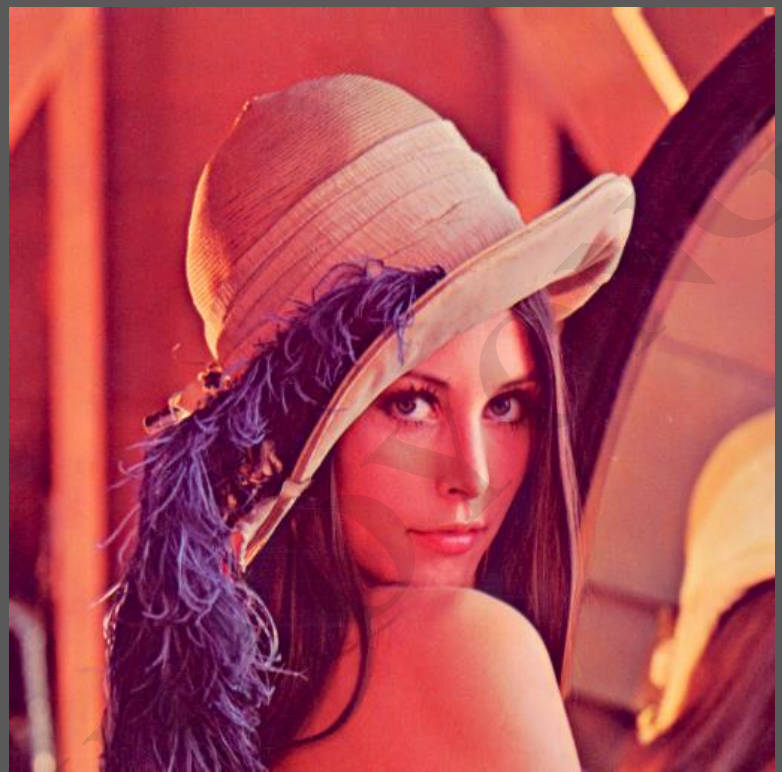
$$\nabla f_y^2(x,y) \approx -f(x,y-1) + 2f(x,y) - f(x,y+1)$$



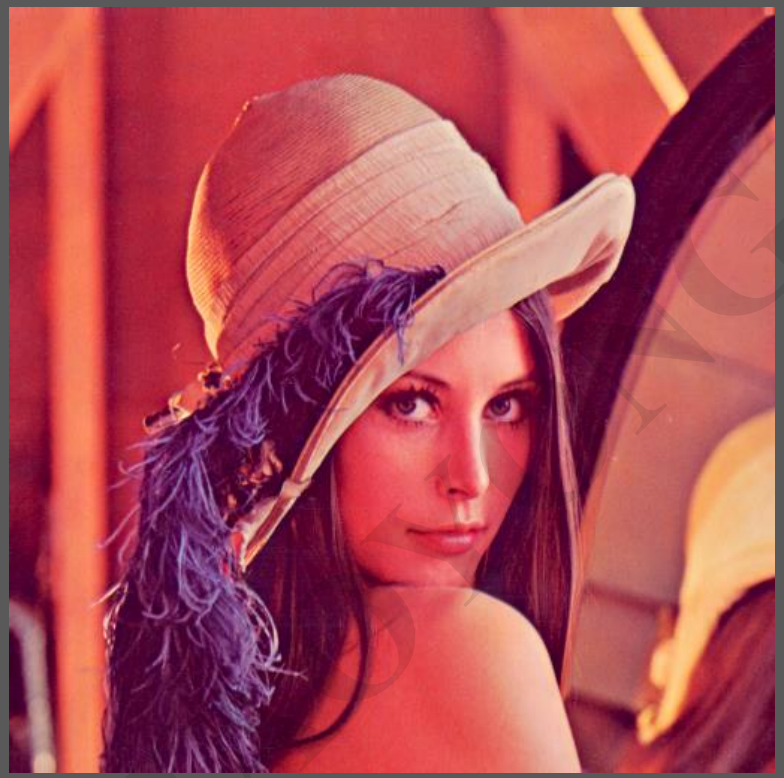
https://www.mygreatlearning.com/blog/introduction-to-edge-detection/


TALK
IS CHEAP SHOW ME
THE CODE

```python
from PIL import Image,ImageFilter
import urllib.request
import skimage.io

# save the image from the url
urllib.request.urlretrieve('https://upload.wikimedia.org/wikipedia/en/7/7d/Lenna_%28test_image%29.png',
                           'lenna.png')

# load the image using PIL.Image
img_lenna = Image.open('lenna.png')
# convert the image into gray scale
img_lenna_gray = img_lenna.convert("L")
# detect the edges using the argumented filter
image_edge = img_lenna_gray.filter(ImageFilter.FIND_EDGES)
display(image_edge)
```



```
# detect the edges using the Laplacian filter
image_edge = img_lenna_gray.filter(ImageFilter.Kernel((3, 3), (-1, -1, -1, -1, 8,
                                    -1, -1, -1, -1), 1, 0))
display(image_edge)
```

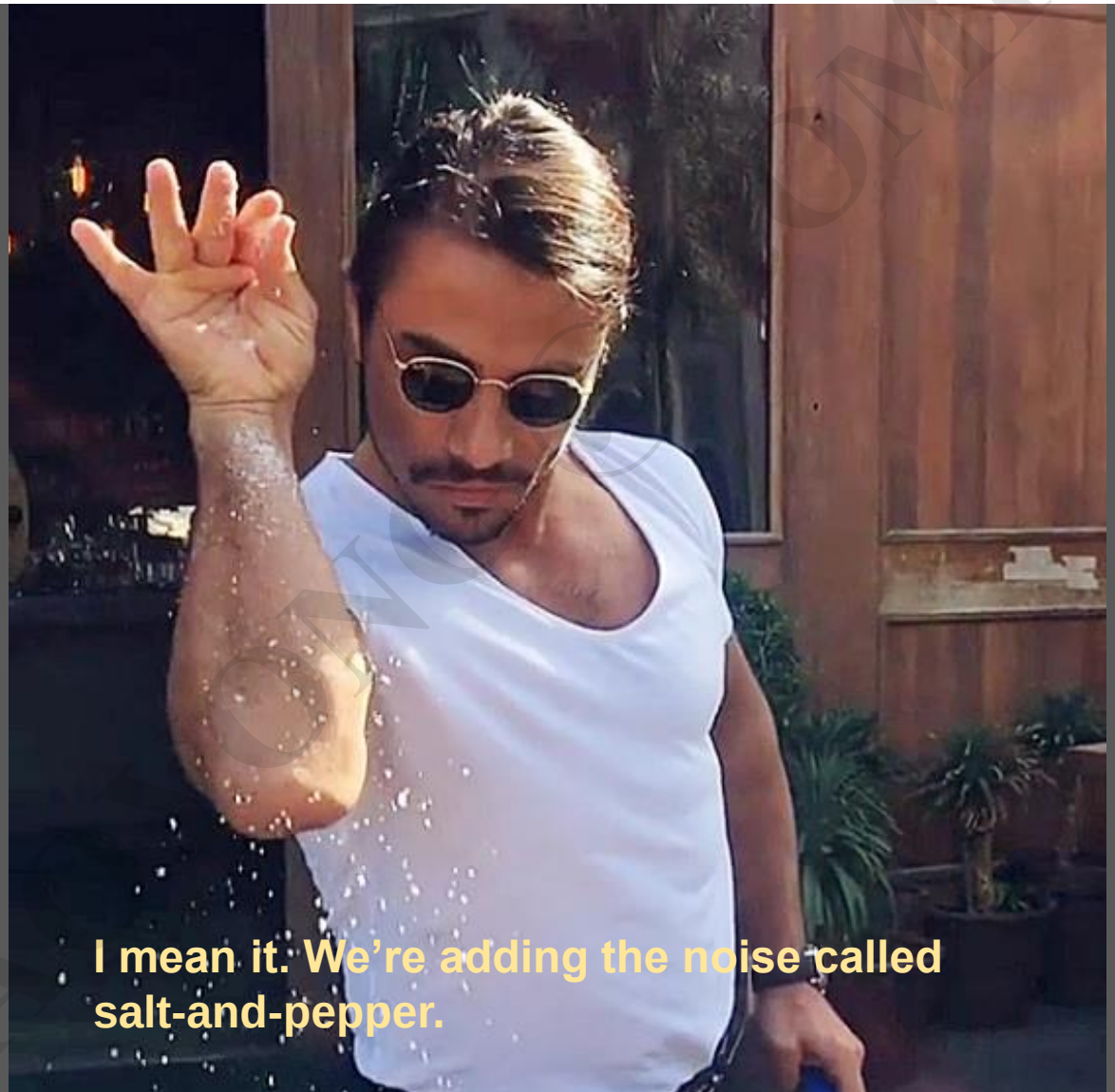


```
# detect the edges using the Sobel filters
image_edge_x = img_lenna_gray.filter(ImageFilter.Kernel((3, 3), (1, 2, 1, 0, 0,
                                                0, -1, -2, -1), 1, 0))
display(image_edge_x)
image_edge_y = img_lenna_gray.filter(ImageFilter.Kernel((3, 3), (1, 0, -1, 2, 0,
                                                -2, 1, 0, -1), 1, 0))
display(image_edge_y)
```

Filters is a useful tool not only for **Edge Detection**.

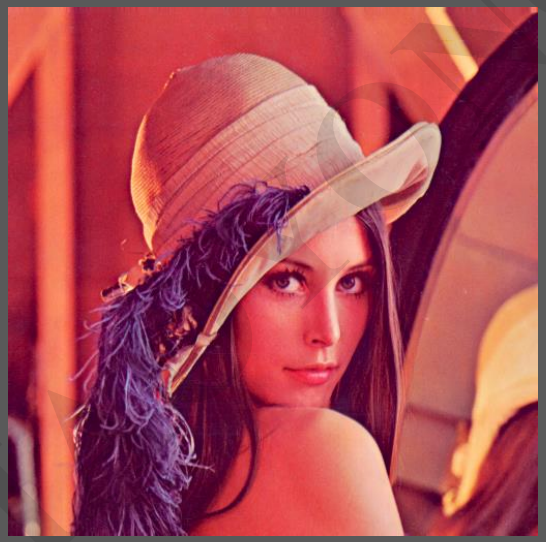
To begin with, let's try Noise Reduction!

But we have to add some “salt” first



But we have to add some noise first

```
from skimage.util import random_noise
import cv2
import numpy as np

# load the image
img_lenna = cv2.imread('lenna.png')
# Add salt-and-pepper noise
lenna_noise_array = random_noise(img_lenna, mode='s&p',amount=0.05)
print(type(lenna_noise_array))
print((lenna_noise_array.dtype))
# we have to convert the array into uint8 before composing the image
lenna_noise_array = np.array(255*lenna_noise_array, dtype = 'uint8')
lenna_noise_img=Image.fromarray(lenna_noise_array)
display(lenna_noise_img)
```



Mean Filter

The idea of the Mean Filter is to "average out" the noise using the surrounding pixels. However, it will "blur" the normal pixels at the same time.

$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$

Mean Filter

```
image_denoise = lenna_noise_img.filter(ImageFilter.Kernel((3, 3), (1/9, 1/9, 1/9, 1/9, 1/9,
                                                                  1/9, 1/9, 1/9, 1/9), 1, 0))
display(image_denoise)
```

Median Filter

Instead of using the average-out, the Median filters reduce the noise by “replacing” the central pixel by the median of pixels in the neighborhood. The assumption is that noise appear as the “outliers” on which the *median* works better than the *mean*.

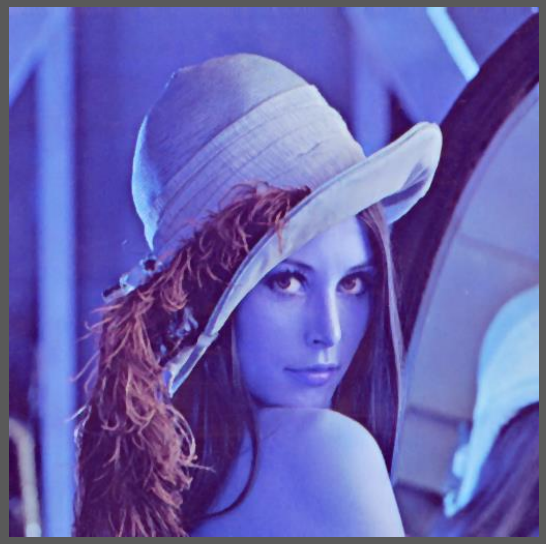
	10	5	20					
	14	80	11					
	8	3	22					

(3,5,8,10,11,14,20,22,80)

Mean Filter

```
image_denoise = lenna_noise_img.filter(ImageFilter.MedianFilter)
display(image_denoise)
```

Mean Filter



Median Filter

Noise vs. Filters

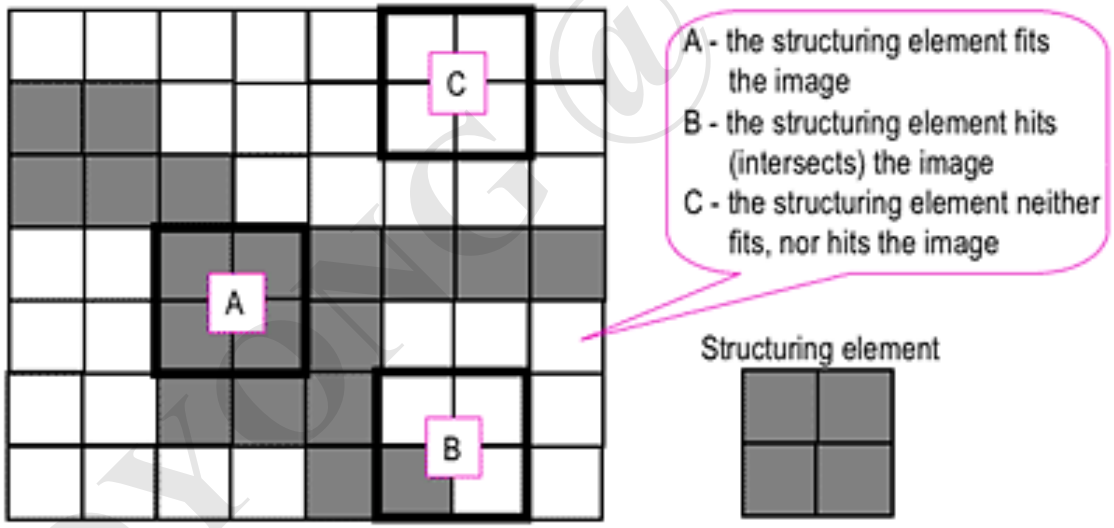
Median filters seem worked better than Mean filters for salt-and-pepper noise. However, it doesn't mean it's a better choice in a general sense. The performance of filters are dependent on the type of noise. Please find below the noise and filters and try them out by yourself.

Noise	Recognized Filters
Salt and pepper	Median
Poisson	Mean
Gaussian	Gaussian
Speckle	Weiner

Another set of tools, which is similar to Filters, is **Morphological Operations.**

Morphological Operations

Morphological techniques probe an image with a small shape or template called a **Structuring Element (SE).** The structuring element is positioned at all possible locations in the image and it is compared with the corresponding neighborhood of pixels. Some operations test whether the element "fits" within the neighborhood, while others test whether it "hits" or intersects the neighborhood:



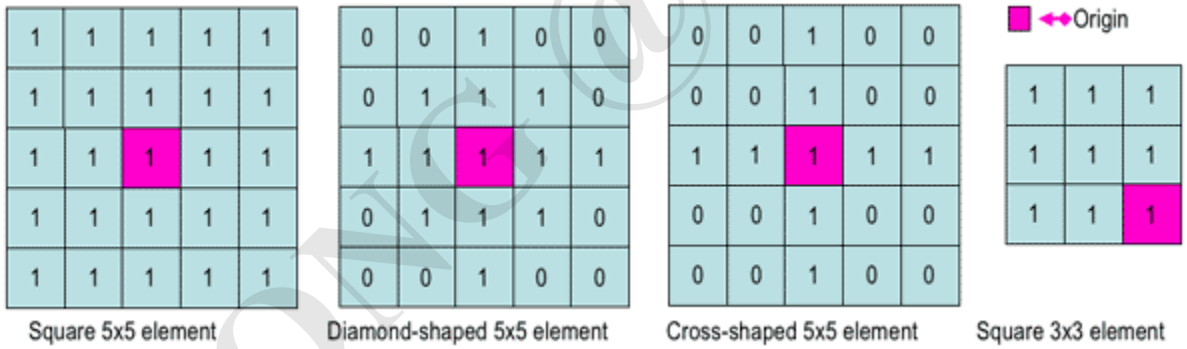
https://www.cs.auckland.ac.nz/courses/compsci773s1c/lectures/ImageProcessing-html/topic4.htm

Structing elements can be considered as filters with which a different way of filtering (i.e., Morphological Operations) is defined (rather than using the weighted averages).

In addition, it is usually applied to the **binary images**, on which it's easier to determine the fitness of a structing element to image positions.

Morphological Operations

- The **structuring element** is a small binary image, i.e. a small matrix of pixels, each with a value of zero or one:
  - The matrix dimensions specify the size of the structuring element.
  - The pattern of ones and zeros specifies the shape of the structuring element.
  - An origin of the structuring element is usually one of its pixels, although generally the origin can be outside the structuring element.



A common practice is to have odd dimensions of the structuring matrix and the origin defined as the centre of the matrix. Structuring elements play in morphological image processing the same role as convolution kernels in linear image filtering.

https://www.cs.auckland.ac.nz/courses/compsci773s1c/lectures/ImageProcessing-html/topic4.htm

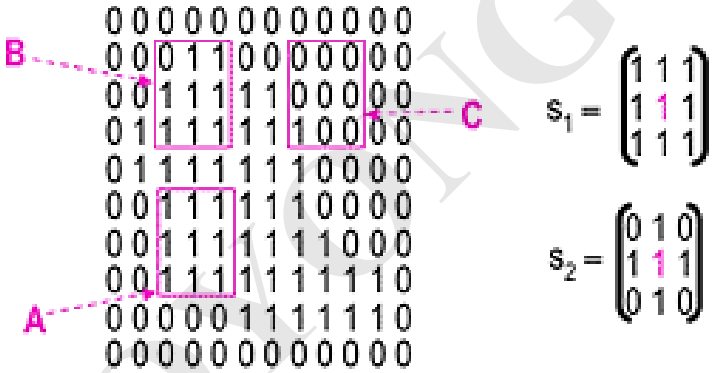
Morphological Operations

> **Fit:**

- The structuring element is said to **fit** the image if, **for each of** its pixels set to 1, the corresponding image pixel is also 1.
- $sum(Region \cap S) = sum(S)$

> **Hit:**

- Similarly, a structuring element is said to **hit,** or intersect, an image if, **at least for one** of its pixels set to 1 the corresponding image pixel is also 1.
- $sum(Region \cap S) > 0$



$$s_1 = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \qquad s_2 = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

		A	B	C
fit	s_1	yes	no	no
	s_2	yes	yes	no
hit	s_1	yes	yes	yes
	s_2	yes	yes	no

https://www.cs.auckland.ac.nz/courses/compsci773s1c/lectures/ImageProcessing-html/topic4.htm

Morphological Operations

- Basic morphological operations include erosion, dilation, opening and closing.
- **Erosion**
- The erosion of a binary image f by a structuring element s (denoted $f \ominus s$) produces a new binary image $g = f \ominus s$ with ones in all locations (x, y) of a structuring element's origin at which that structuring element s **fits** the input image f, i.e. $g(x, y) = 1$ if s fits f and 0 otherwise, repeating for all pixel coordinates (x, y).

Greyscale image

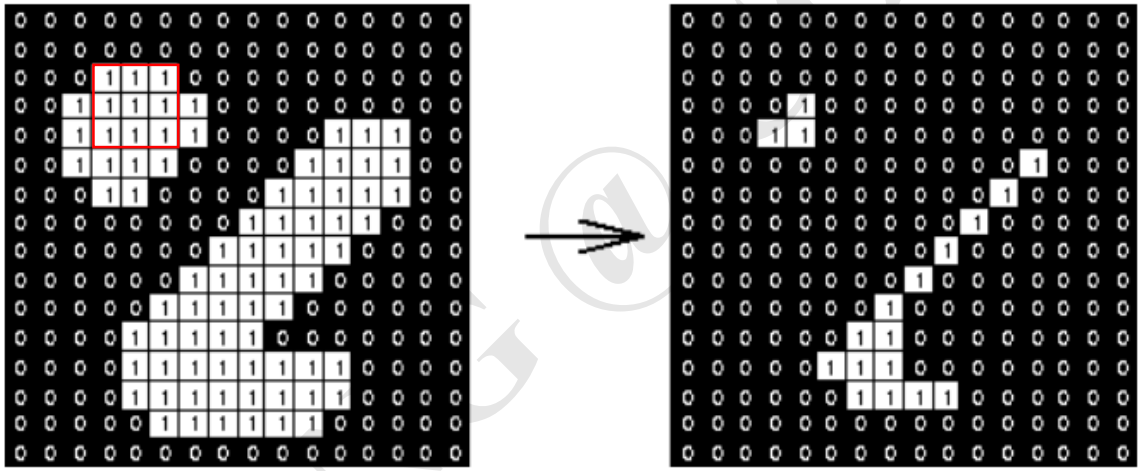
Binary image by thresholding

Erosion: a 2×2 square structuring element

https://www.cs.auckland.ac.nz/courses/compsci773s1c/lectures/ImageProcessing-html/topic4.htm
http://documents.wolfram.com/applications/digitalimage/UsersGuide/Morphology/ImageProcessing6.3.html

Morphological Operations

Erosion with small (e.g. 2×2 - 5×5) square structuring elements shrinks an image by stripping away a layer of pixels from both the inner and outer boundaries of regions.



Erosion: a 3×3 square structuring element

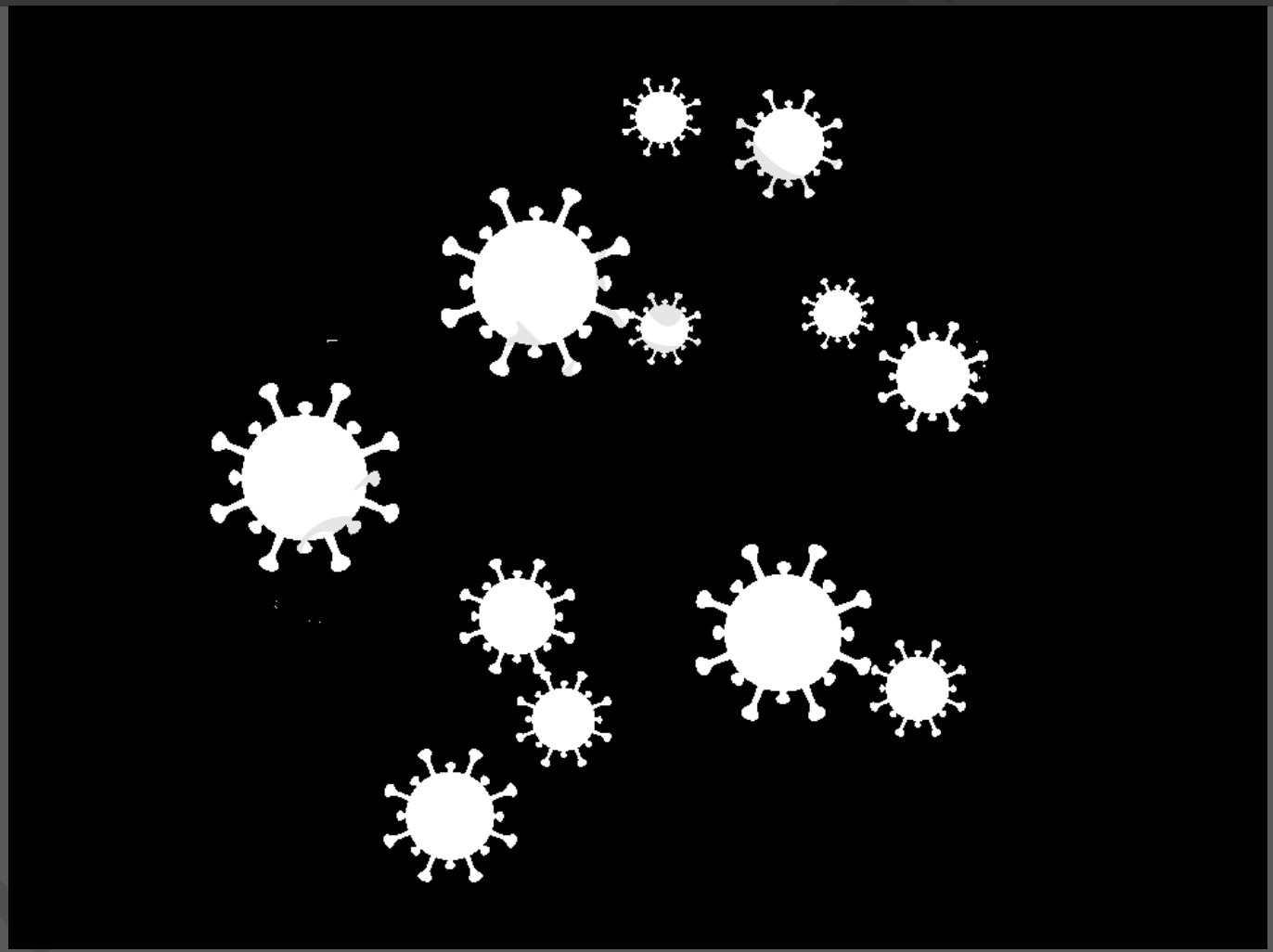
Erosion removes small-scale details from a binary image but simultaneously reduces the size of regions of interest, too. By subtracting the eroded image from the original image, boundaries of each region can be found: $b = f - (f \ominus s)$ where f is an image of the regions, s is a 3×3 structuring element, and b is an image of the region boundaries.

https://www.cs.auckland.ac.nz/courses/compsci773s1c/lectures/ImageProcessing-html/topic4.htm
www.cs.princeton.edu/~pshilane/class/mosaic/

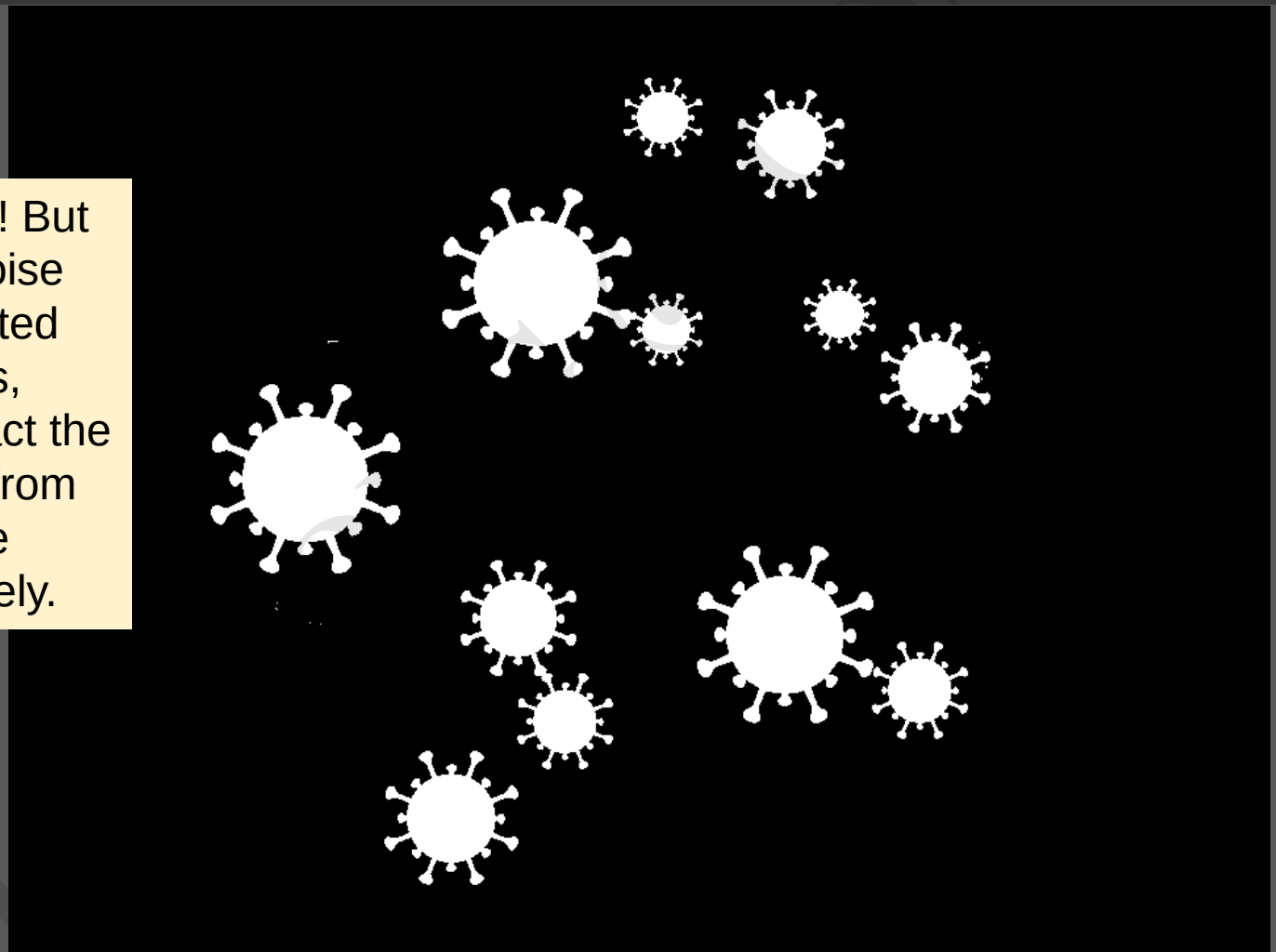
Let's learn this through a new example of "Find the Coronaviruses"

How many coronavirus are there?

```
import cv2

# load the image containing coronavirus
img_virus=cv2.imread("coronvirus-mask.png")
# convert to grey image
img_vir_grey = cv2.cvtColor(img_virus, cv2.COLOR_BGR2GRAY)
# get the binary image after thresholding
ret,img_th = cv2.threshold(img_vir_grey,160,255,cv2.THRESH_BINARY)
bw_img=Image.fromarray(255-img_th)
display(bw_img)
```



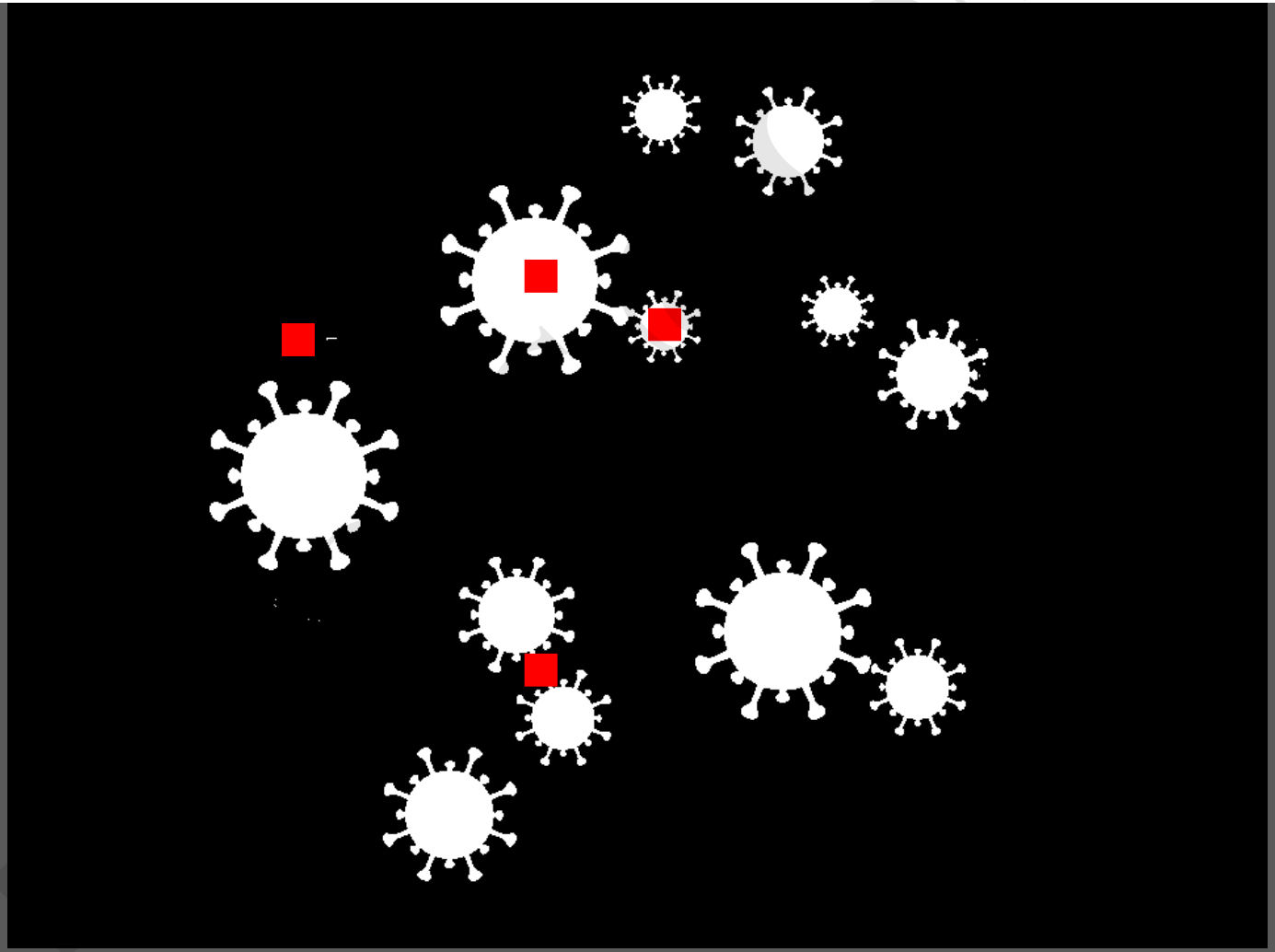
```python
import cv2

# load the image containing coronavirus
img_virus=cv2.imread("coronvirus-mask.png")
# convert to grey image
img_vir_grey = cv2.cvtColor(img_virus, cv2.COLOR_BGR2GRAY)
# get the binary image after thresholding
ret,img_th = cv2.threshold(img_vir_grey,160,255,cv2.THRESH_BINARY)
bw_img=Image.fromarray(255-img_th)
display(bw_img)
```



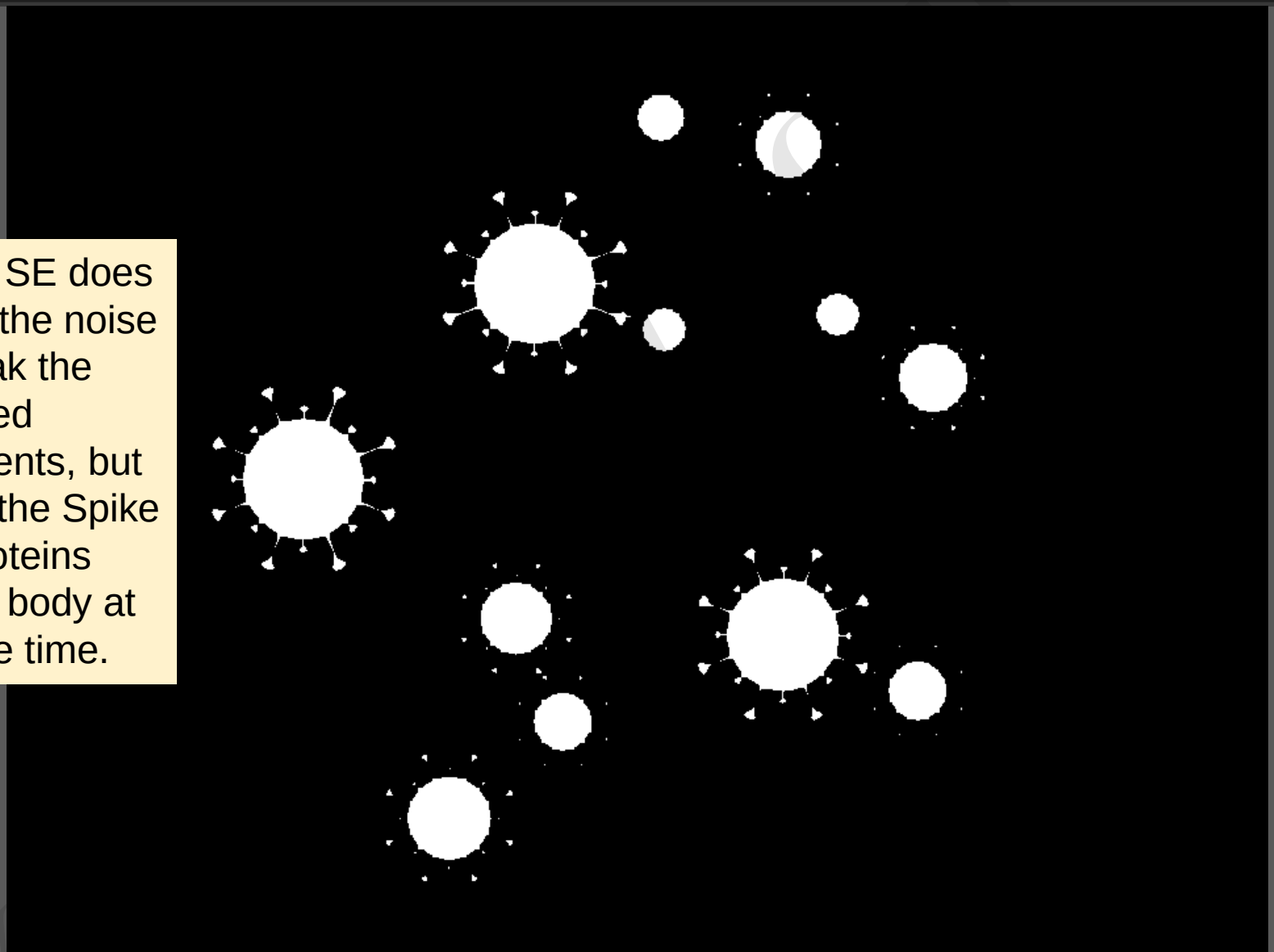
Looks good! But there are noise and connected components, which distract the computers from counting the virus precisely.

Use Erosion to exclude the distractions

- **Erosion** excludes the positions that the Structing Element (SE) dose not fits. We can design a SE that fits into the bodies of viruses but not the noise and connections.

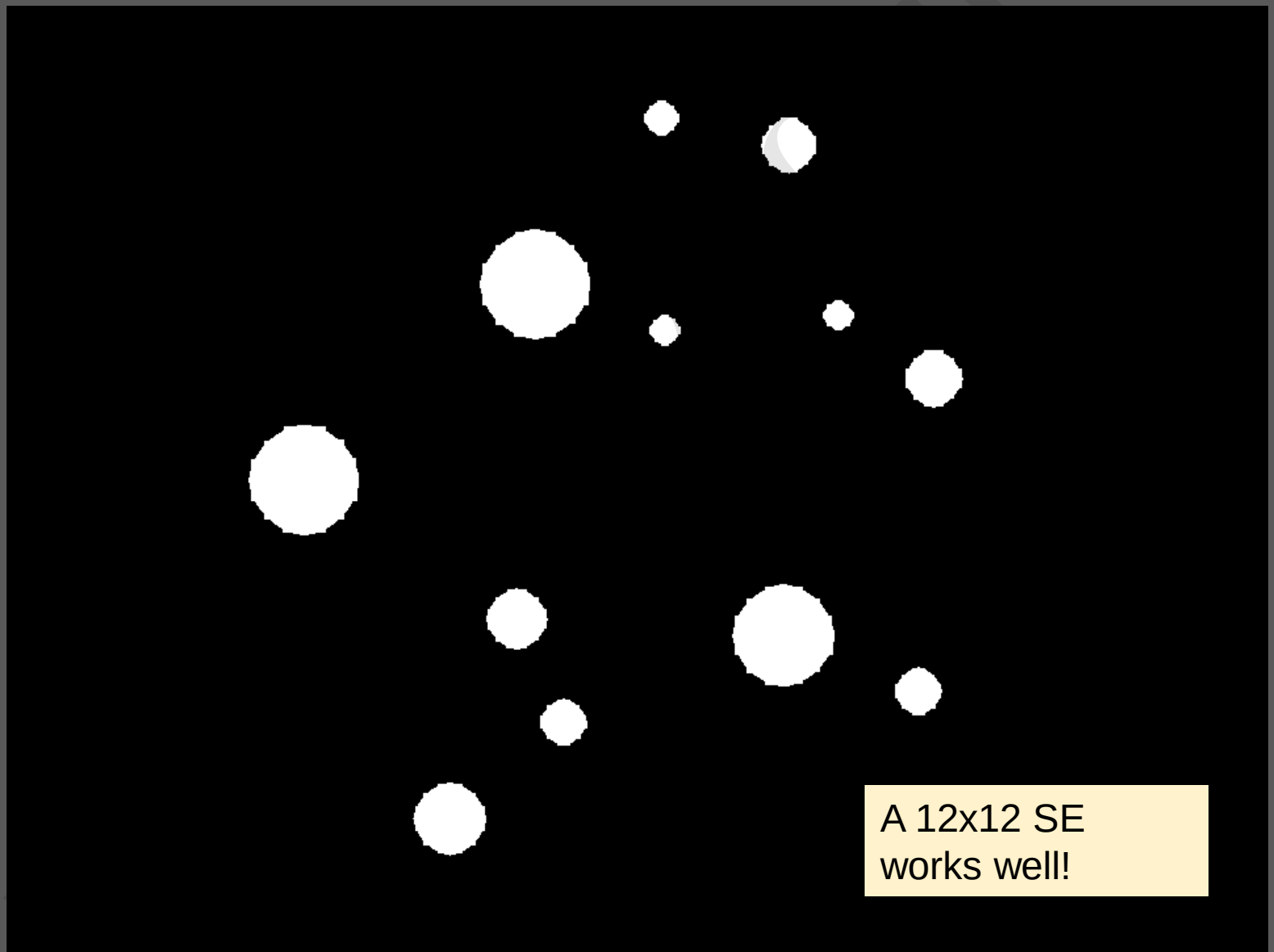


```
import numpy as np
#define a 5x5 SE
SE = np.ones((5,5),np.uint8)
img_erosion = cv2.erode(255-img_th,SE,iterations = 1)
bw_img=Image.fromarray(img_erosion)
display(bw_img)
```



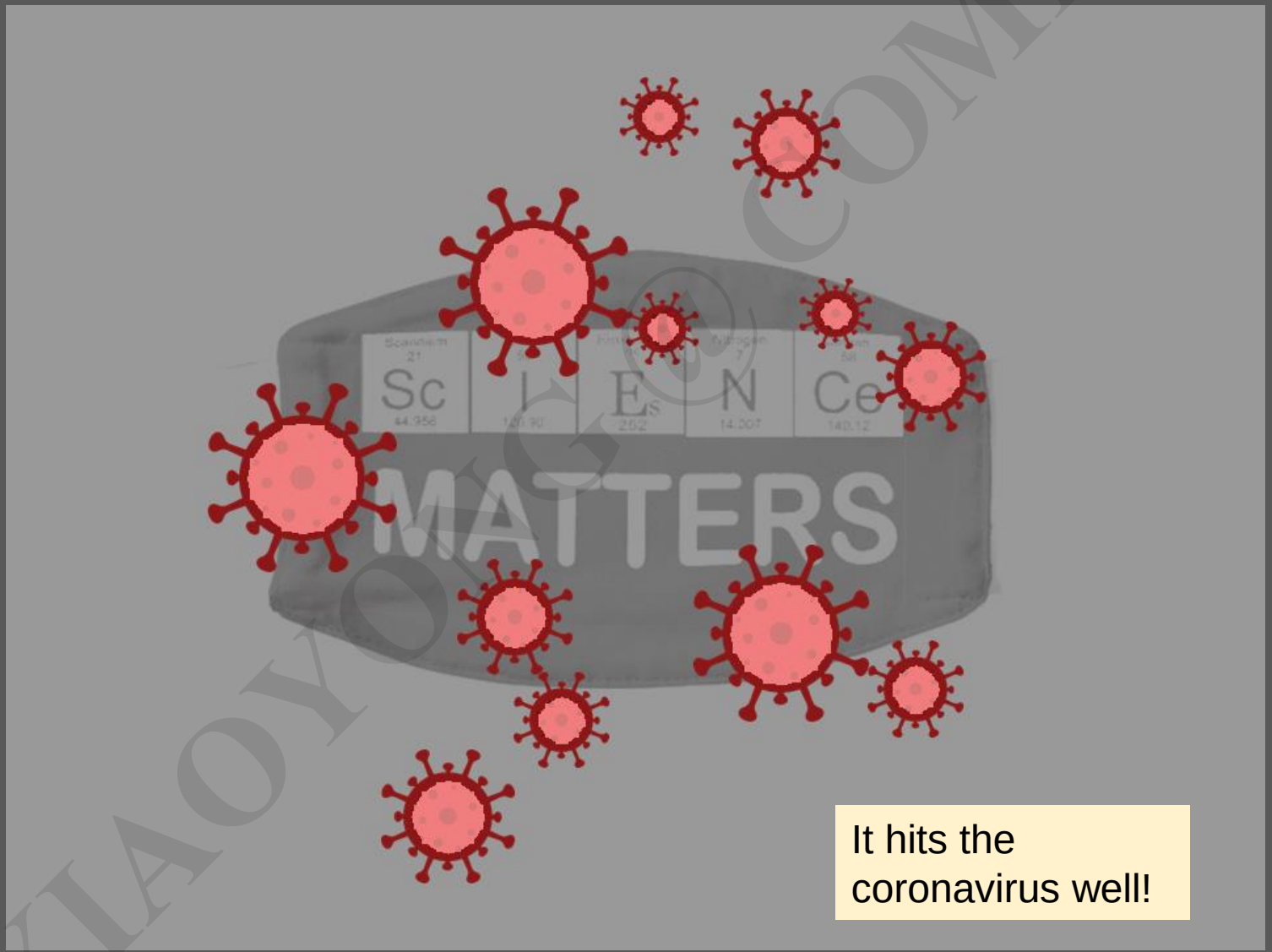
The 5x5 SE does exclude the noise and break the connected components, but isolates the Spike Glycoproteins from the body at the same time.

```
SE = np.ones((12,12),np.uint8)
img_erosion = cv2.erode(255-img_th,SE,iterations = 1)
bw_img=Image.fromarray(img_erosion)
display(bw_img)
```



A 12x12 SE works well!

```
# let's merage it back to the orginal image and see the result of localization
np_merge=cv2.merge((img_erosion,img_erosion,img_erosion))*0.4+np_virus*0.6
display(Image.fromarray(cv2.cvtColor(np_merge.astype(np.uint8), cv2.COLOR_BGR2RGB)))
```



It hits the coronavirus well!

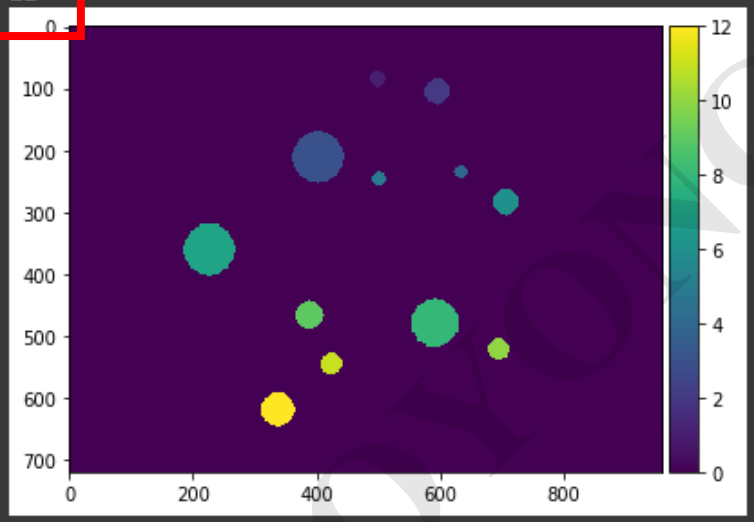
Now, we can count the viruses by the number of the white dots

The white dots hit the virus well, which meams we can count the virus by the number of dots.

This can be done using the ***skimage.measure.label*** and ***skimage.measure.regionprops***.

```
from skimage.measure import label,regionprops
from skimage import io
np_labeled=label(img_erosion)
io.imshow(np_labeled)
regions = regionprops(np_labeled)
print(len(regions))
```

```
/usr/local/lib/python3.7/dist-packages/skimage/io/_plugins/matplotlib_plugin.py:150: UserWarning: Low image data range; displaying
  lo, hi, cmap = _get_display_range(image)
12
```



There are 12 dots (viruses)!

Studies show that the bigger a virus is, the more infectious it is.
(I made it up. It's not necessarily true.)

Can you measure how infectious each of the virus is?

We can measure the area of each virus. Let's build a mask for each of them first. This can be done using **Dilation**.

Morphological Operations

- **Dilation:** The dilation of an image f by a structuring element s (denoted $f \oplus s$) produces a new binary image $g = f \oplus s$ with ones in all locations (x, y) of a structuring element's origin at which that structuring element s **hits** the the input image f, i.e. $g(x, y) = 1$ if s hits f and 0 otherwise, repeating for all pixel coordinates (x, y).

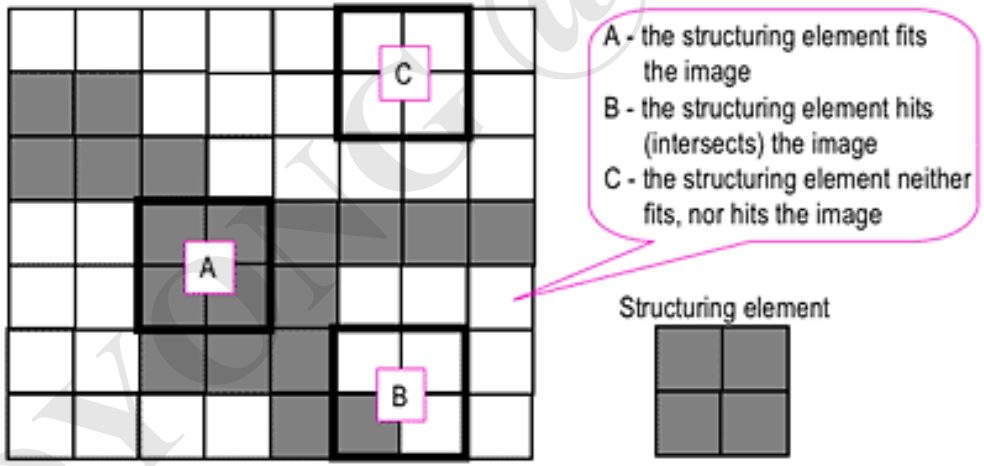
Binary image

Dilation: a 2×2 square structuring element

https://www.cs.auckland.ac.nz/courses/compsci773s1c/lectures/ImageProcessing-html/topic4.htm
http://documents.wolfram.com/applications/digitalimage/UsersGuide/Morphology/ImageProcessing6.3.html

Morphological Operations

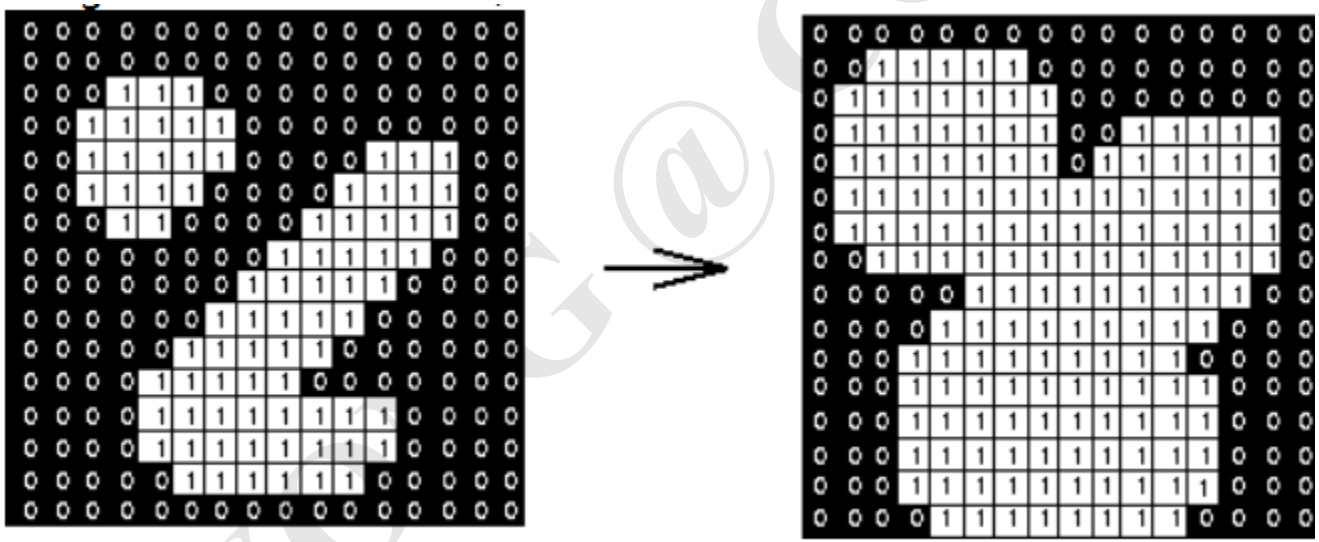
Morphological techniques probe an image with a small shape or template called a structuring element. The structuring element is positioned at all possible locations in the image and it is compared with the corresponding neighborhood of pixels. Some operations test whether the element "fits" within the neighborhood, while others test whether it "hits" or intersects the neighborhood:



https://www.cs.auckland.ac.nz/courses/compsci773s1c/lectures/ImageProcessing-html/topic4.htm

Morphological Operations

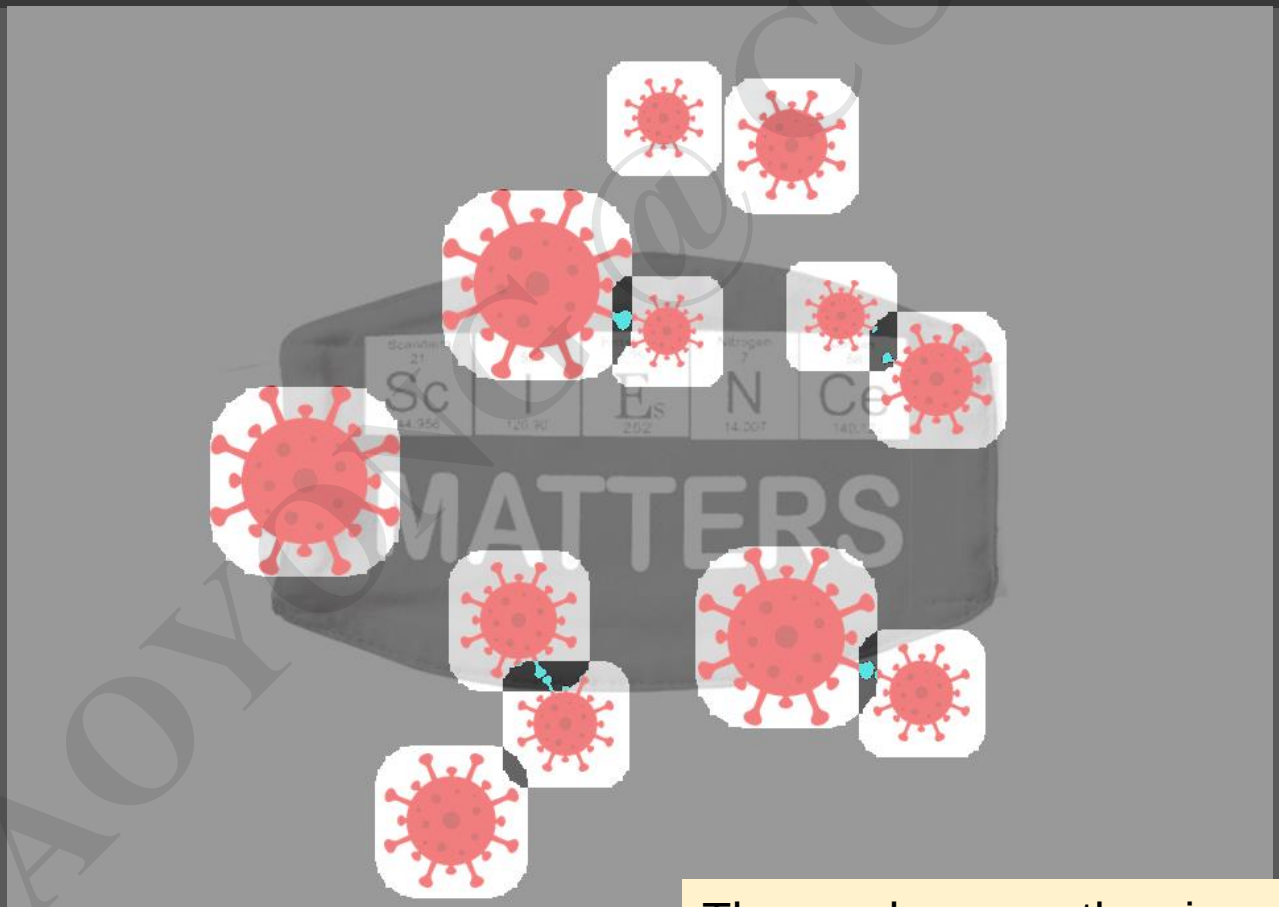
> Dilation has the opposite effect to erosion -- it adds a layer of pixels to both the inner and outer boundaries of regions.



Dilation: a 3×3 square structuring element

https://www.cs.auckland.ac.nz/courses/compsci773s1c/lectures/ImageProcessing-html/topic4.htm
www.cs.princeton.edu/~pshilane/class/mosaic/

```
np_regions=[]
np_maskes=[]
merged_mask=np.zeros(np_labeled.shape)
for index in range(1, np_labeled.max()+1):
    np_regions.append((((np_labeled==index)+0)*255).astype(np.uint8))
    np_dilation = cv2.dilate(np_regions[-1],SE,iterations = 15)
    np_maskes.append(np_dilation)
    merged_mask=merged_mask+np_dilation
    #display(Image.fromarray(np_dilation))

np_merge=cv2.merge((merged_mask,merged_mask,merged_mask))*0.4+np_virus*0.6
display(Image.fromarray(cv2.cvtColor(np_merge.astype(np.uint8), cv2.COLOR_BGR2RGB)))
```

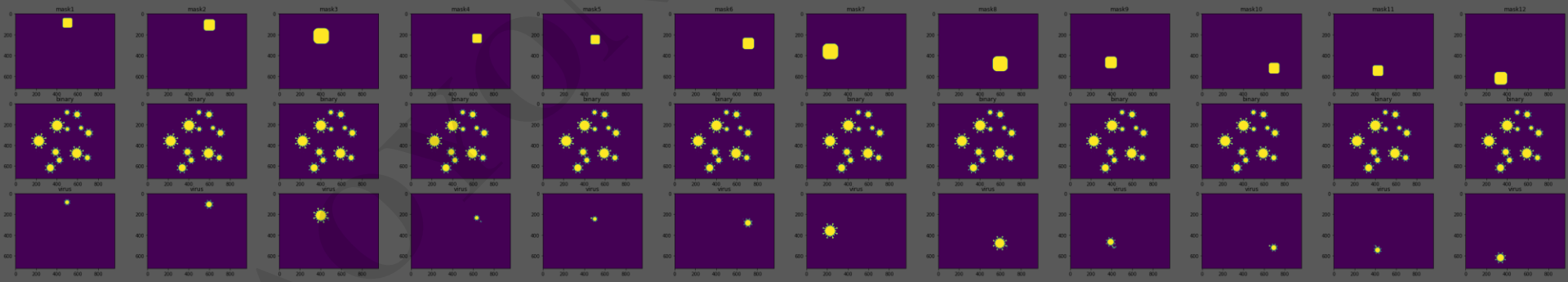


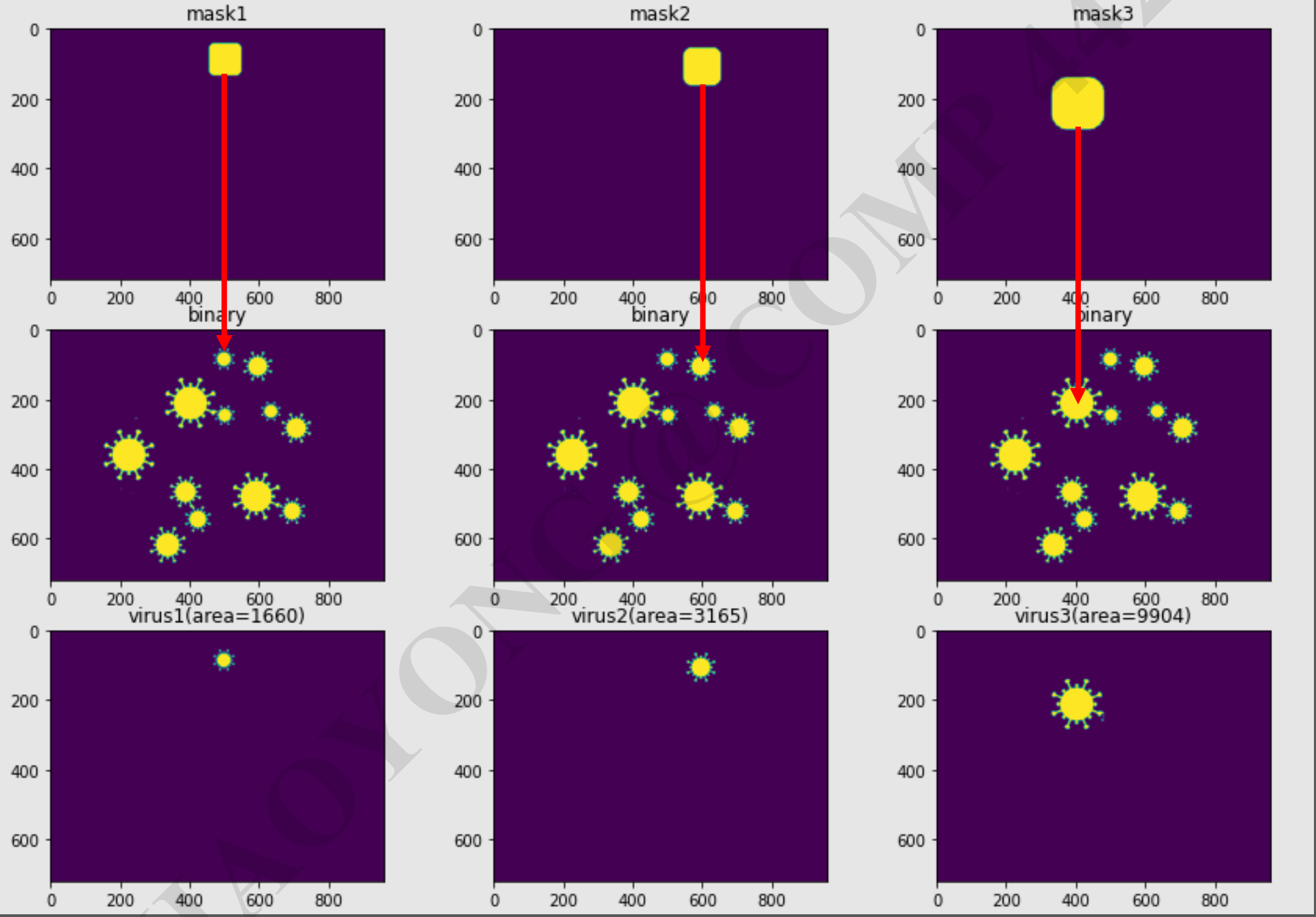
The masks cover the viruses well

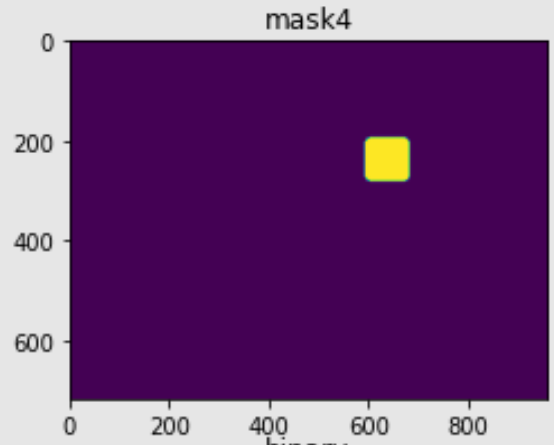
Now we can use the masks to separate the viruses from each other. This can be done by calculating the **AND** of a mask and the original binary image.

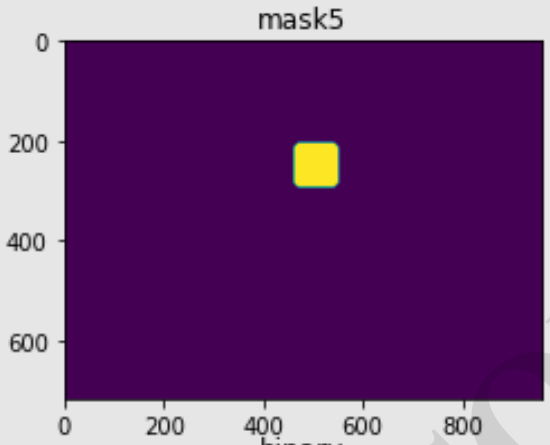
```python
import matplotlib.pyplot as plt
fig = plt.figure(figsize=(60, 10))
binary_np=255-img_th
idx=1
plts=[]
for mask in np_masks:
  binary_virus=((mask==255)*(binary_np==255))
  area=np.sum(binary_virus)
  #display(Image.fromarray(cv2.cvtColor((binary_virus*255).astype(np.uint8), cv2
  plts.append(fig.add_subplot(3, 12, idx))
  plts[-1].set_title('mask'+str(idx))
  plt.imshow(mask)
  plts.append(fig.add_subplot(3, 12, idx+12))
  plts[-1].set_title('binary')
  plt.imshow(binary_np)
  plts.append(fig.add_subplot(3, 12, idx+24))
  plts[-1].set_title('virus')
  plt.imshow(binary_virus)
  print('Area of virus '+str(idx)+':\t'+str(area))
  idx=idx+1
```

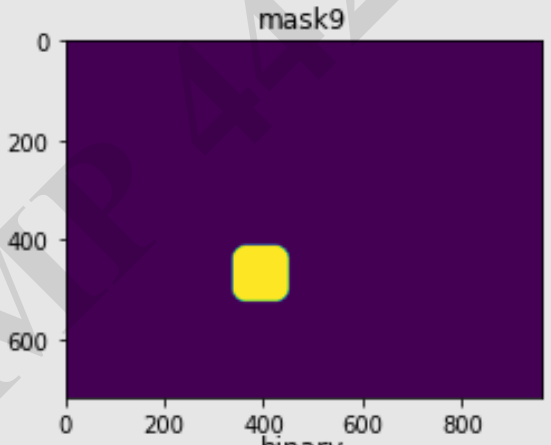
```
Area of virus 1:        1660
Area of virus 2:        3165
Area of virus 3:        9904
Area of virus 4:        1546
Area of virus 5:        1636
Area of virus 6:        3453
Area of virus 7:        9912
Area of virus 8:        8716
Area of virus 9:        4038
Area of virus 10:       2666
Area of virus 11:       2607
Area of virus 12:       4877
```

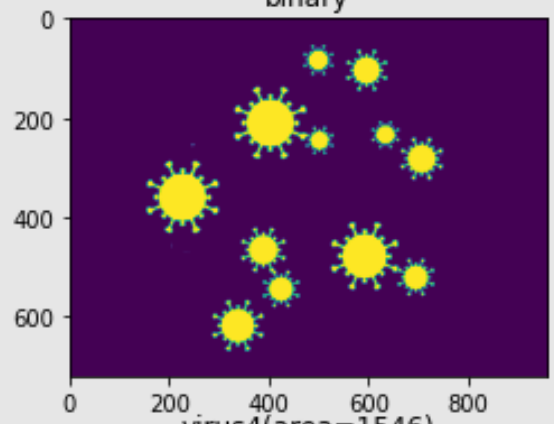


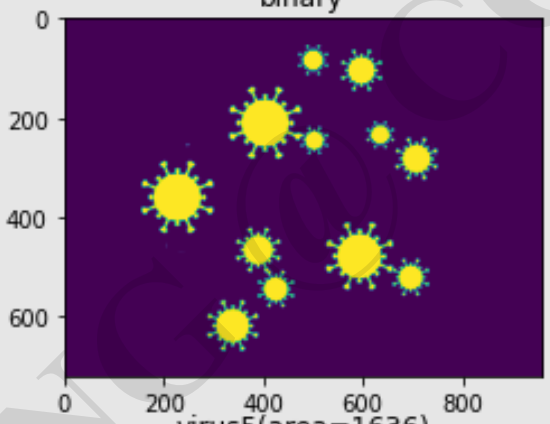

mask1
mask2
mask3
binary
binary
binary
virus1(area=1660)
virus2(area=3165)
virus3(area=9904)

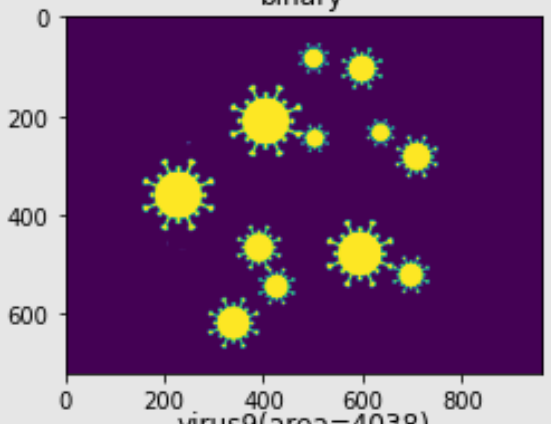

mask4
0
200
400
600
0
200
400
600
800

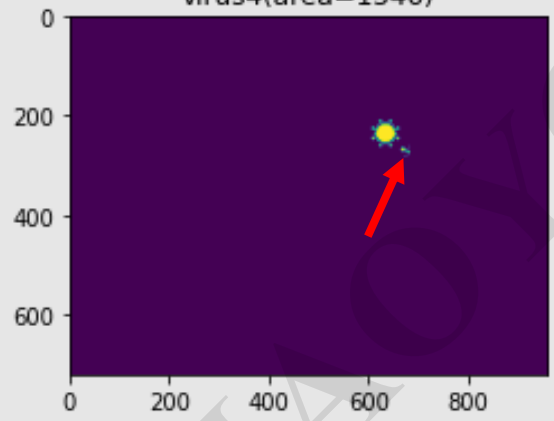

mask5
0
200
400
600
0
200
400
600
800

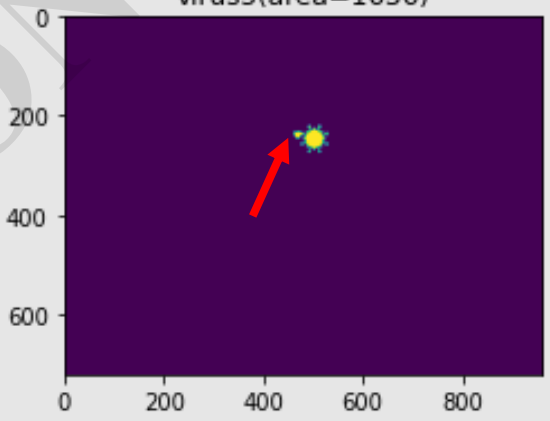

mask9
0
200
400
600
0
200
400
600
800

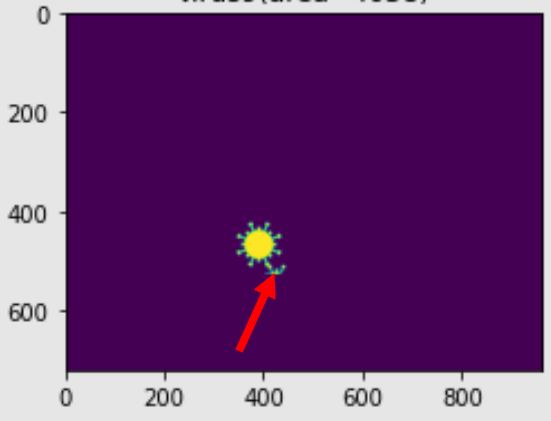

binary
0
200
400
600
0
200
400
600
800


binary
0
200
400
600
0
200
400
600
800


binary
0
200
400
600
0
200
400
600
800


virus4(area=1546)
0
200
400
600
0
200
400
600
800


virus5(area=1636)
0
200
400
600
0
200
400
600
800

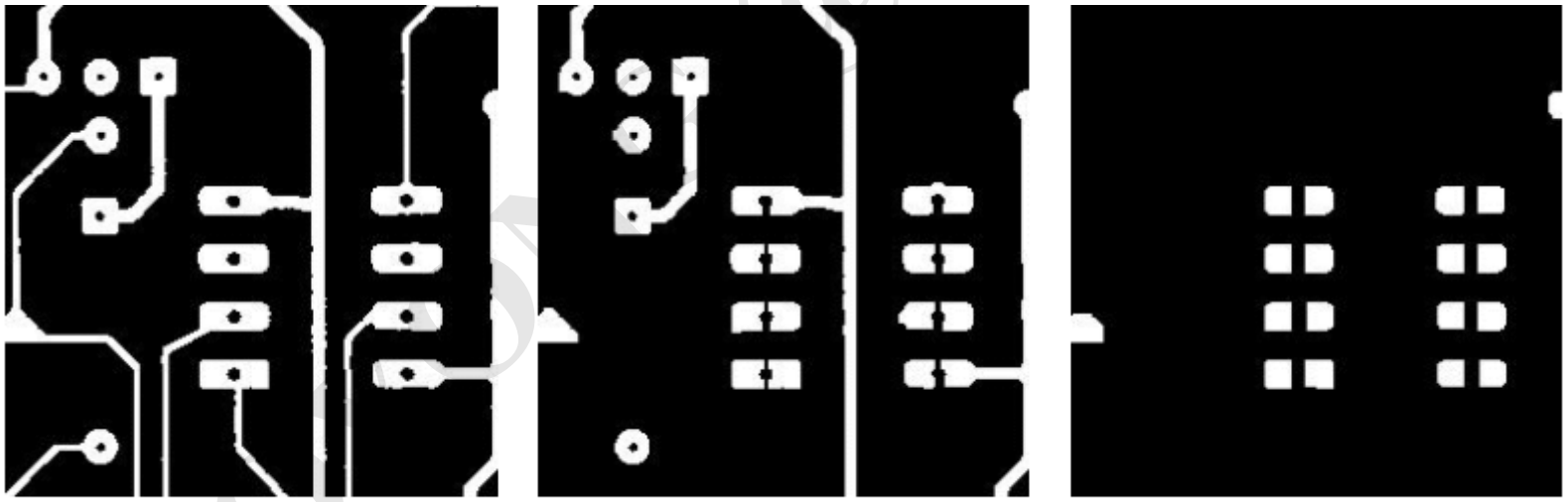

virus9(area=4038)
0
200
400
600
0
200
400
600
800

Not perfect, but OK.

Is there a better way?

Morphological Operations

- **Opening:** The opening of an image f by a structuring element s (denoted by $f \circ s$) is **an erosion followed by a dilation**:
- $f \circ s = (f \ominus s) \oplus s$
- Opening is so called because it can open up a gap between objects connected by a thin bridge of pixels. Any regions that have survived the erosion are restored to their original size by the dilation:

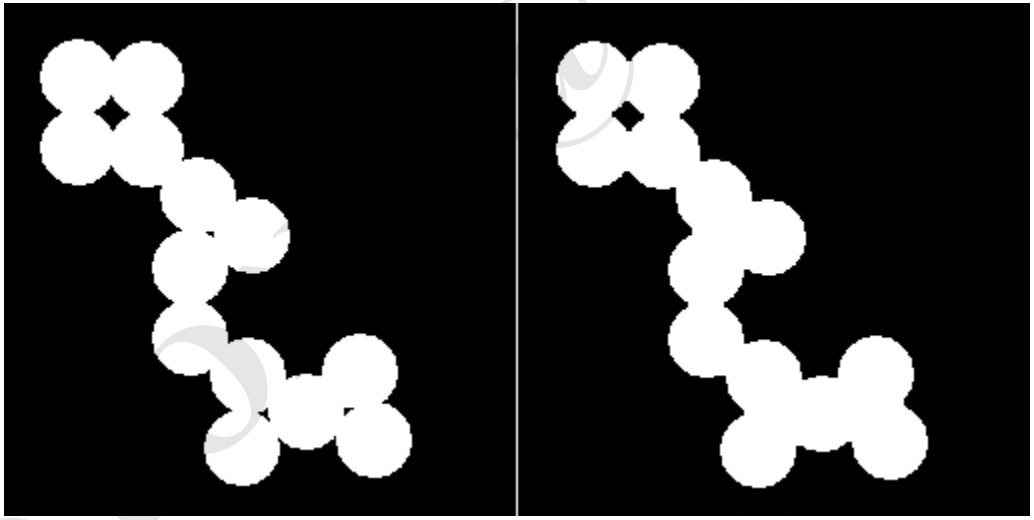


Binary image f | $f \circ s$ (5×5 square) | $f \circ s$ (9×9 square)

https://www.cs.auckland.ac.nz/courses/compsci773s1c/lectures/ImageProcessing-html/topic4.htm
www.mmorph.com/html/morph/mmopen.html/
https://en.wikipedia.org/wiki/Mathematical_morphology#Basic_operators

Morphological Operations

- **Closing:** The closing of an image f by a structuring element s (denoted by $f \bullet s$) is **a dilation followed by an erosion**:
- $f \bullet s = (f \oplus s) \ominus s$
- Closing is so called because it can fill holes in the regions while keeping the initial region sizes.



https://www.cs.auckland.ac.nz/courses/compsci773s1c/lectures/ImageProcessing-html/topic4.htm
https://www.mathworks.com/help/images/morphological-dilation-and-erosion.html

Thank you!