Lecture 3 - Filtering and Morphological Operations

February 1, 2021

1 Lecture 3 - Filtering and Morphological Operations

In this notebook, we will discuss how we will detect and clean objects of interests in images. Specifically, we will discuss: 1. Spacial Filters 1. Morphological Operations

The contents of this notebook is compiled from the following references:

- C Alis, *"Introduction to Digital Image"*, IIP 2018

Organized by: Benjur Emmanuel L. Borja

1.1 1. Spatial Filters

Spatial filters assign the value of a pixel based on their neighbors. Filters are defined as matrices, known as **kernels**, and are applied to an image through an operation known as a convolution. The traditional kernels are initialized below. For more information on the different kernels, and their effects, visit this link.

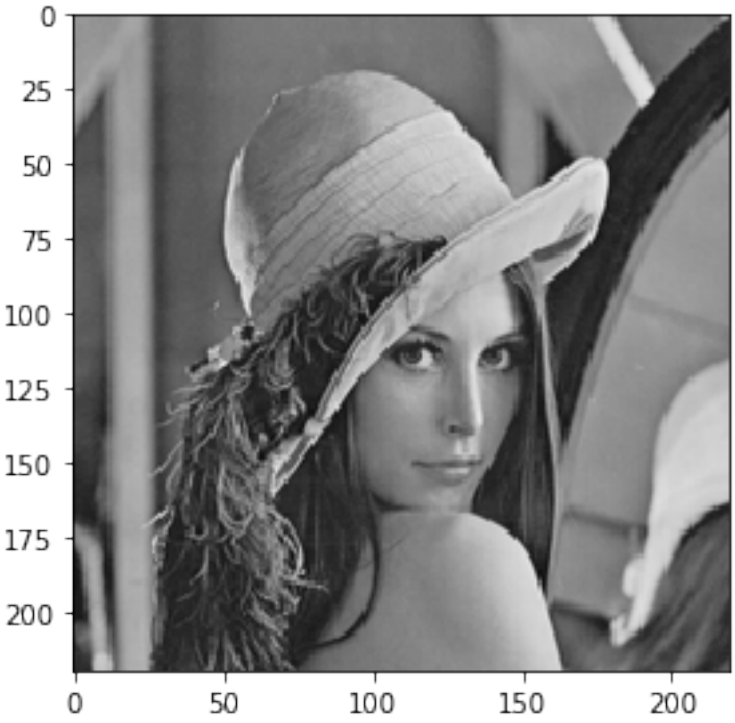
[76]:
```
import numpy as np

# Sobel Operators
# Horizontal Sobel Filter
kernel1 = np.array([[1, 2, 1],
                    [0, 0, 0],
                    [-1, -2, -1]])

# Vertical Sobel Filter
kernel2 = np.array([[1, 0, -1],
                    [2, 0, -2],
                    [1, 0, -1]])

# Left Diagonal Filter
kernel3 = np.array([[1, -1, -1],
                    [-1, 1, -1],
                    [-1, -1, 1]])

# Right Diagonal Filter
kernel4 = np.array([[-1, -1, 1],
```

```
                    [-1, 1, -1],
                    [1, -1, -1]])

# Edge Detection
kernel5 = np.array([[-1, -1, -1],
                    [-1, 8, -1],
                    [-1, -1, -1]])

# Sharpen
kernel6 = np.array([[0, -1, 0],
                    [-1, 5, -1],
                    [0, -1, 0]])

# Box Blur
kernel7 = (1 / 9.0) * np.array([[1., 1., 1.],
                                [1., 1., 1.],
                                [1., 1., 1.]])

# Gaussian Blur
kernel8 = (1 / 16.0) * np.array([[1., 2., 1.],
                                 [2., 4., 2.],
                                 [1., 2., 1.]])
```

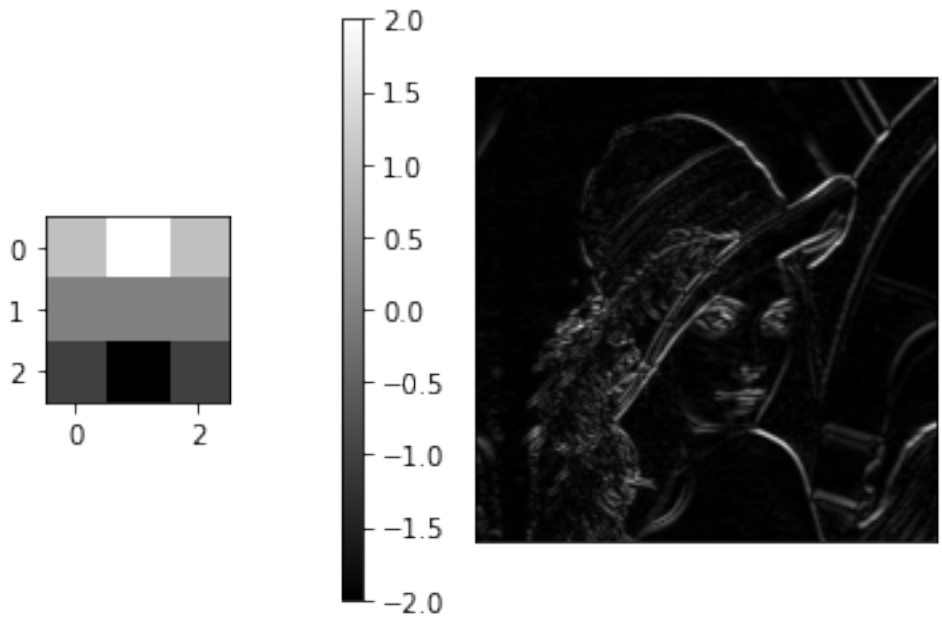
Let's check how these filters affect our image. First we try our horizontal sobel kernel, then we try the vertical one. From the effects of each on the image, we can see that different edges of the image are highlighted by the choice of kernel. This is useful to know, as the appropriate choice of kernel/filter will impact the what features of the image will be brought into focus.

[77]:
```
from skimage.io import imread, imshow
from skimage.color import rgb2gray

lena = rgb2gray(imread('lena.png'))
imshow(lena)
```

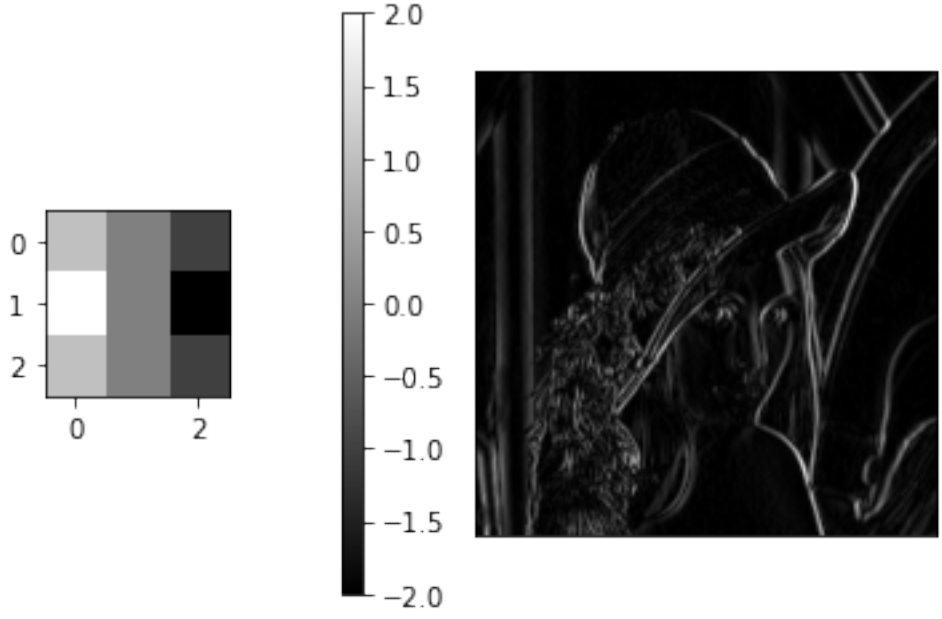
[77]:
```
<matplotlib.image.AxesImage at 0x7f8a1d94f400>
```



[78]:
```
import matplotlib.pyplot as plt
from scipy.signal import convolve2d

conv_im1 = convolve2d(lena, kernel1, 'valid')
fig, ax = plt.subplots(1,3, gridspec_kw=dict(width_ratios=(8,1,20), wspace=0.5))
kern = ax[0].imshow(kernel1, cmap='gray')
fig.colorbar(kern, cax=ax[1])
ax[2].imshow(abs(conv_im1), cmap='gray')
ax[2].set_xticks([]);
ax[2].set_yticks([]);
```

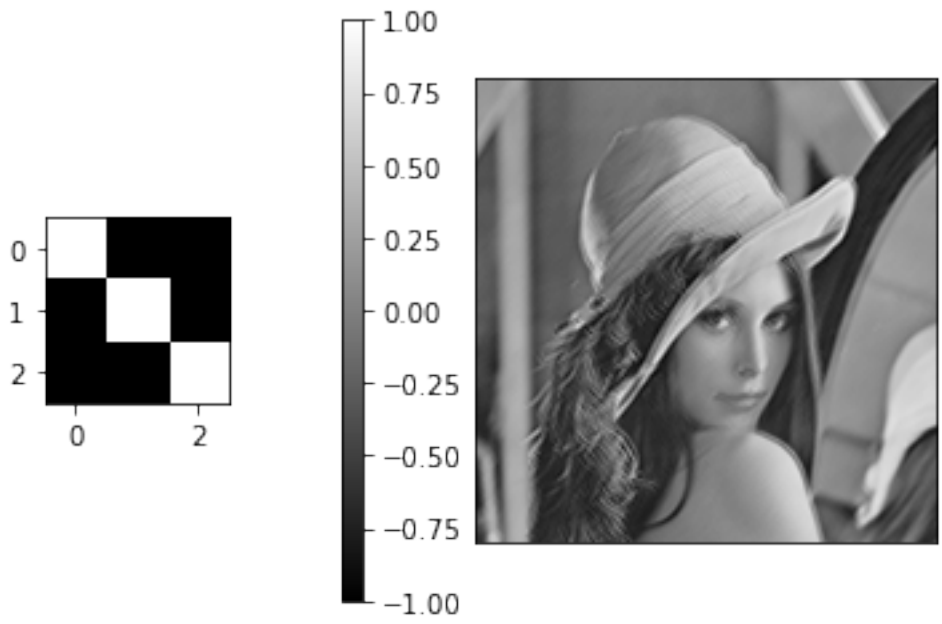


[79]:
```
conv_im1 = convolve2d(lena, kernel2, 'valid')
fig, ax = plt.subplots(1,3, gridspec_kw=dict(width_ratios=(8,1,20), wspace=0.5))
kern = ax[0].imshow(kernel2, cmap='gray')
fig.colorbar(kern, cax=ax[1])
ax[2].imshow(abs(conv_im1), cmap='gray')
ax[2].set_xticks([]);
ax[2].set_yticks([]);
```

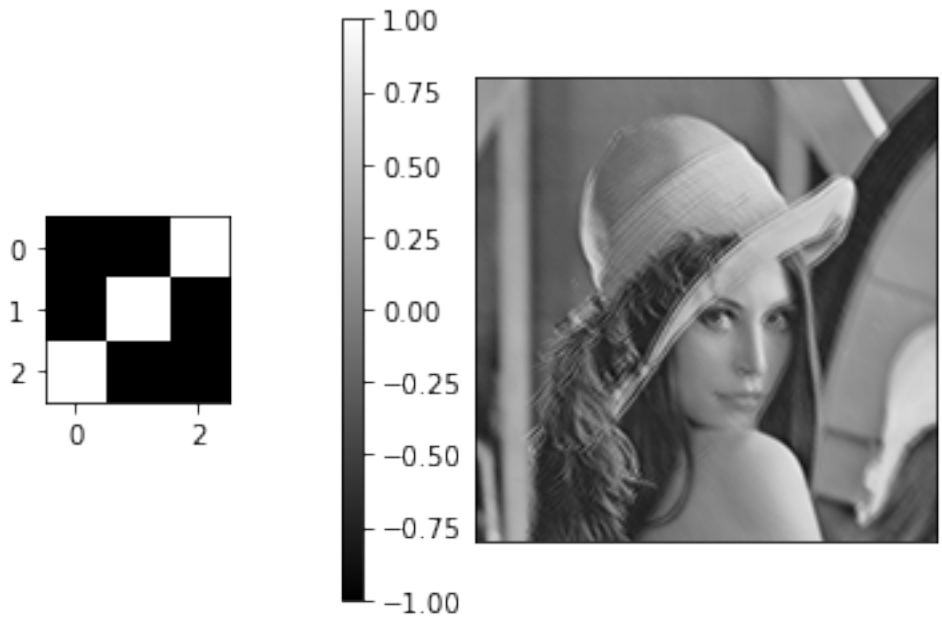


Exercise: Apply the rest of the kernels on the image then describe what the filter does on the image.

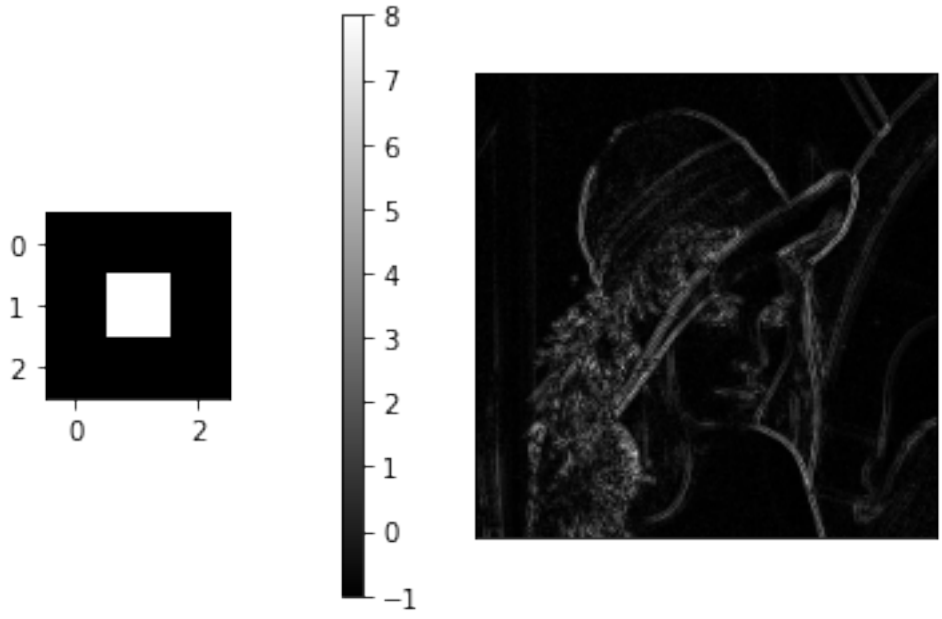
[80]:
```
conv_im1 = convolve2d(lena, kernel3, 'valid')
fig, ax = plt.subplots(1,3, gridspec_kw=dict(width_ratios=(8,1,20), wspace=0.5))
kern = ax[0].imshow(kernel3, cmap='gray')
fig.colorbar(kern, cax=ax[1])
ax[2].imshow(abs(conv_im1), cmap='gray')
ax[2].set_xticks([]);
ax[2].set_yticks([]);
```



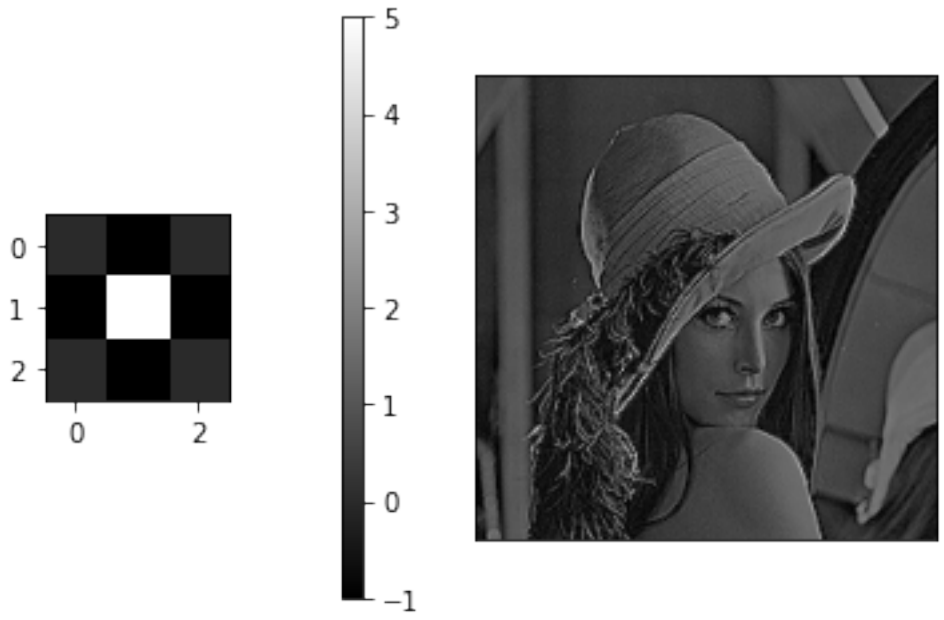
[81]:
```
conv_im1 = convolve2d(lena, kernel4, 'valid')
fig, ax = plt.subplots(1,3, gridspec_kw=dict(width_ratios=(8,1,20), wspace=0.5))
kern = ax[0].imshow(kernel4, cmap='gray')
fig.colorbar(kern, cax=ax[1])
ax[2].imshow(abs(conv_im1), cmap='gray')
ax[2].set_xticks([]);
ax[2].set_yticks([]);
```



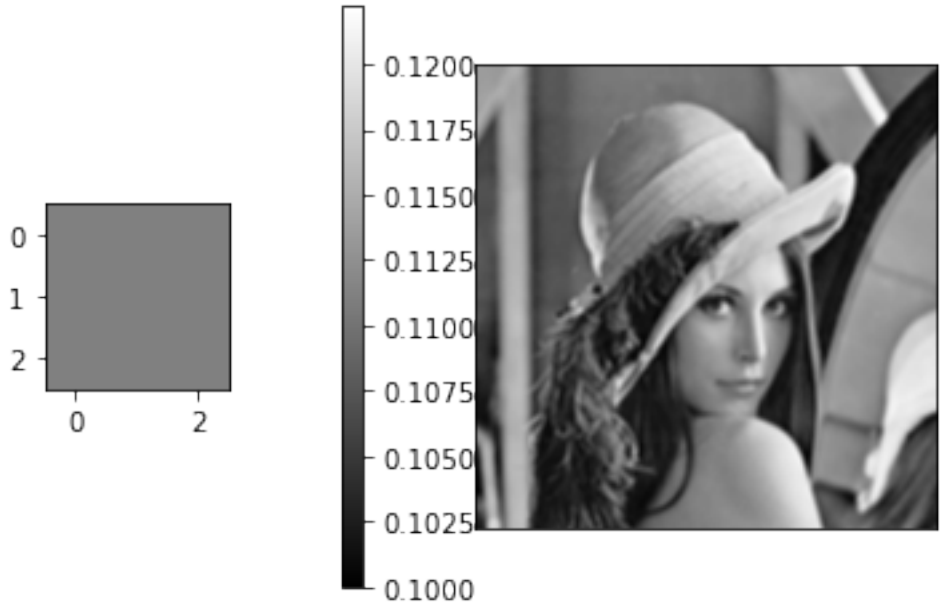
[82]:
```
conv_im1 = convolve2d(lena, kernel5, 'valid')
fig, ax = plt.subplots(1,3, gridspec_kw=dict(width_ratios=(8,1,20), wspace=0.5))
kern = ax[0].imshow(kernel5, cmap='gray')
fig.colorbar(kern, cax=ax[1])
ax[2].imshow(abs(conv_im1), cmap='gray')
ax[2].set_xticks([]);
ax[2].set_yticks([]);
```



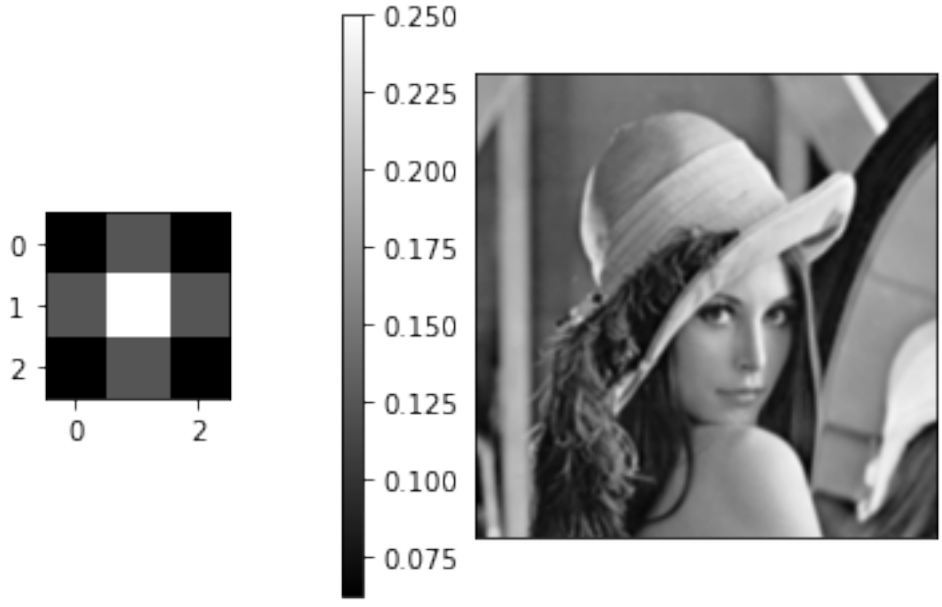
[83]:
```
conv_im1 = convolve2d(lena, kernel6, 'valid')
fig, ax = plt.subplots(1,3, gridspec_kw=dict(width_ratios=(8,1,20), wspace=0.5))
kern = ax[0].imshow(kernel6, cmap='gray')
fig.colorbar(kern, cax=ax[1])
ax[2].imshow(abs(conv_im1), cmap='gray')
ax[2].set_xticks([]);
ax[2].set_yticks([]);
```



[84]:
```
conv_im1 = convolve2d(lena, kernel7, 'valid')
fig, ax = plt.subplots(1,3, gridspec_kw=dict(width_ratios=(8,1,20), wspace=0.5))
kern = ax[0].imshow(kernel7, cmap='gray')
fig.colorbar(kern, cax=ax[1])
ax[2].imshow(abs(conv_im1), cmap='gray')
ax[2].set_xticks([]);
ax[2].set_yticks([]);
```



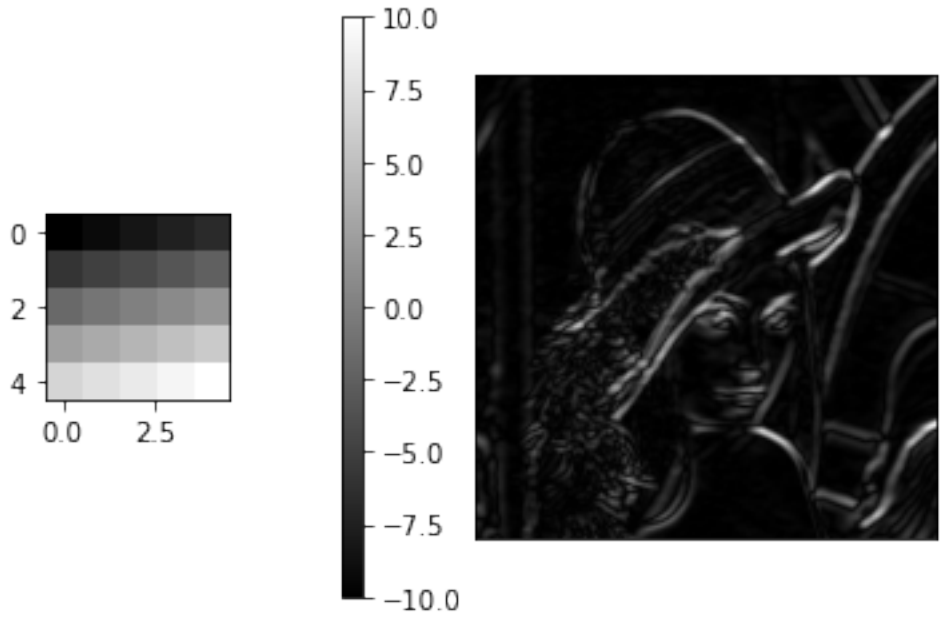
[85]:
```
conv_im1 = convolve2d(lena, kernel8, 'valid')
fig, ax = plt.subplots(1,3, gridspec_kw=dict(width_ratios=(8,1,20), wspace=0.5))
kern = ax[0].imshow(kernel8, cmap='gray')
fig.colorbar(kern, cax=ax[1])
ax[2].imshow(abs(conv_im1), cmap='gray')
ax[2].set_xticks([]);
ax[2].set_yticks([]);
```



From the experiments above, we underscore that each kernel serves a different purpose when applying a filter to an image. In many of the examples, the kernel results in a blurring effect on the image, while the sharpen kernel (kernel 6) appears to merely shift the intensity values darker altogether. Other kernels appear to change the contrast/saturation of the values, while (of course) the edge kernel (#5) reduces the image to its edges.

Exercise: It is important to note that, as the kernel reacts directly with the pixels of the image (read: matrix operations) it is necessary to match the size of the kernel with the resolution of the image. To small a kernel, and you may end up picking out more features than you intended; to large a kernel and you may miss relevant features altogether.

[86]:
```
kernel_ = np.linspace(-10,10, 25).reshape(5,5)

import matplotlib.pyplot as plt
from scipy.signal import convolve2d

conv_im1 = convolve2d(lena, kernel_, 'valid')
fig, ax = plt.subplots(1,3, gridspec_kw=dict(width_ratios=(8,1,20), wspace=0.5))
kern = ax[0].imshow(kernel_, cmap='gray')
fig.colorbar(kern, cax=ax[1])
ax[2].imshow(abs(conv_im1), cmap='gray')
ax[2].set_xticks([]);
ax[2].set_yticks([]);
```



1.2 2. Morphological Operations

Once we have transformed our images using spatial filters and thresholding methods, there are some cases where we need to clean our image in terms of the "correctness" or "completeness" of our objects. For these cases, we will use **Morphological Operations** to enhance our image.

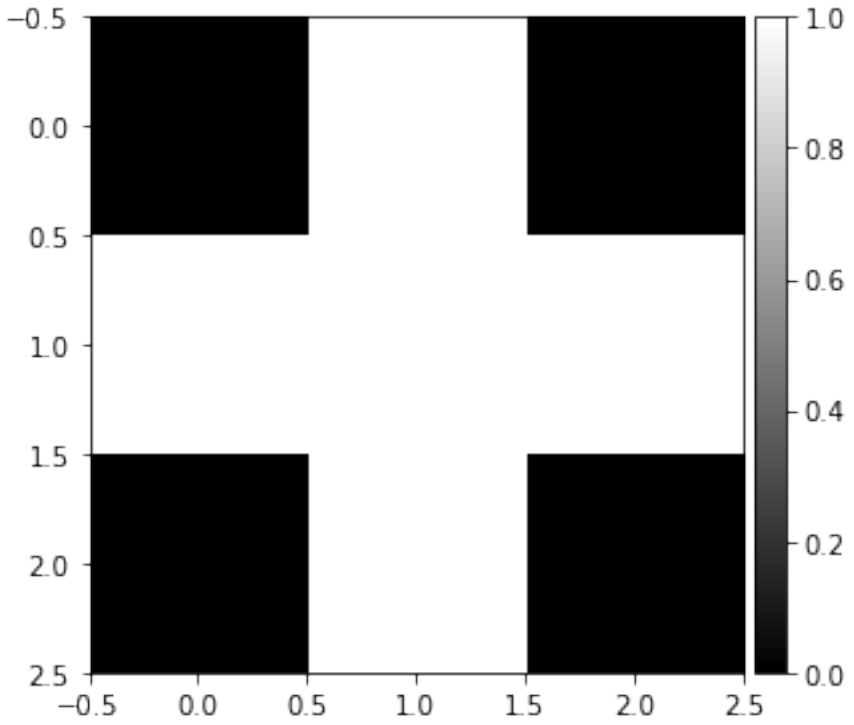
Simillar to spatial filters, morphological operations transform each pixel of an image to a value based on the values of its neighbors selected by a smaller matrix known as a **structuring element**. These operations are useful for cleaning images especially in preparation for segmentation. One way to think about it is that th structuring element tries to correct (either by appending or removing pixels) the structure of a segment of an image.

1.2.1 2.1 Basic morphological operations

To see how morphological operations work, let's discuss there two basic morphological operations: erosion and dilation. Erosion is the process of shaving away at the edges of an image based on the defined structuring element; conversely, dilation is padding at the edges of an image based on the structuring element. It is important to note that, similar to the kernels, the choice of size of the structuring element should be proportional to the image segment that it is working on.

Let's first define our structuring element:

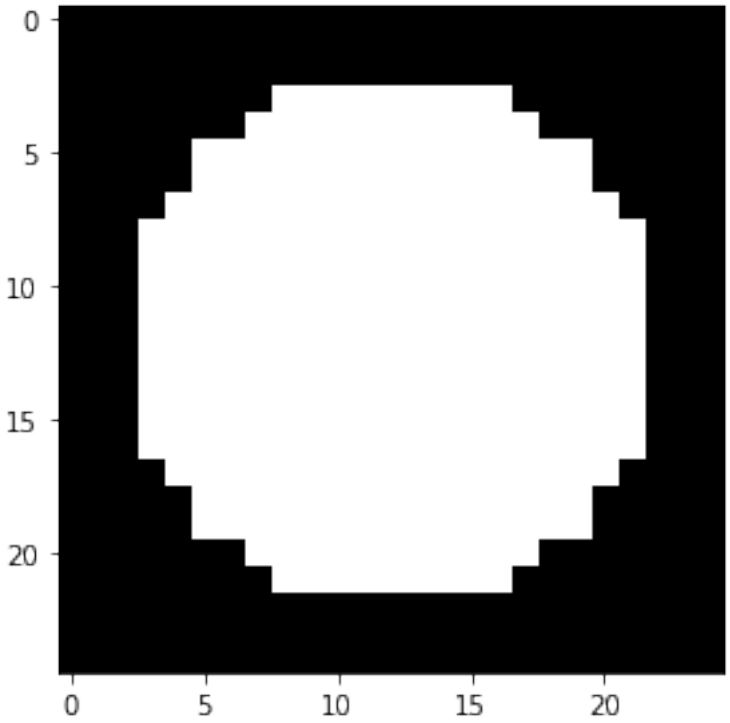
[87]:
```
selem_circ = np.array([[0,1,0],
                       [1,1,1],
                       [0,1,0]])
imshow(selem_circ, cmap='gray');
```



Then let's define a circular image:

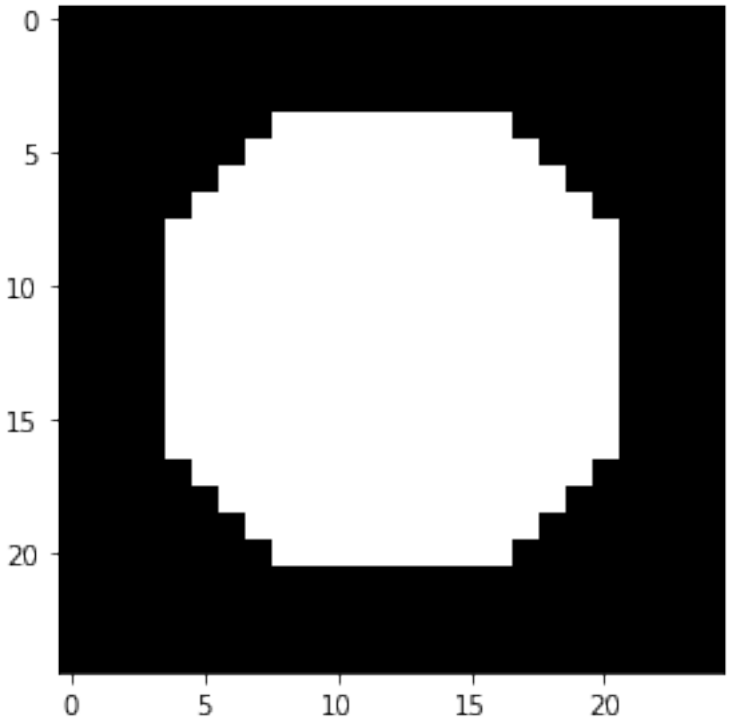
[88]:
```
from skimage.draw import circle

circ_image = np.zeros((25, 25))
circ_image[circle(12, 12, 10)] = 1
imshow(circ_image);
```

```
<ipython-input-88-bf03ab98c584>:4: FutureWarning: circle is deprecated in favor
of disk.circle will be removed in version 0.19
  circ_image[circle(12, 12, 10)] = 1
```



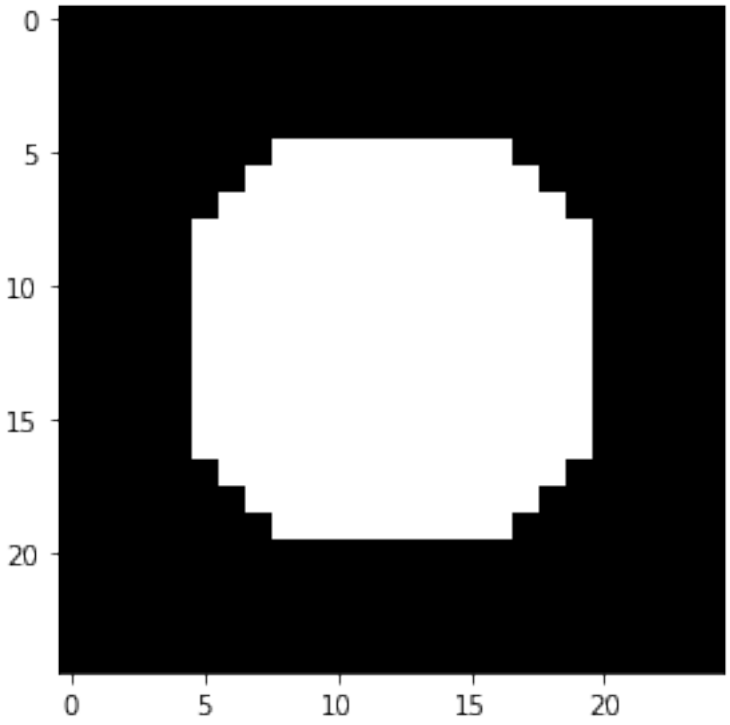
Applying erosion:

[89]:
```
from skimage.morphology import erosion

imshow(erosion(circ_image, selem_circ));
```



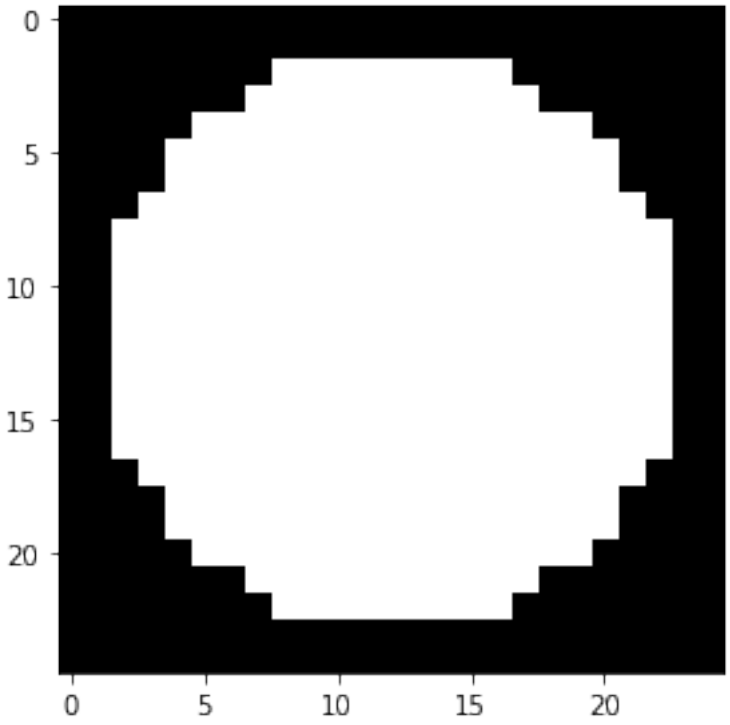
As we can see, the image has shrunk slightly from the original. We can easily imagine how an image segment can get progressively smaller over multiple erosions.

[90]:
```
circ_image2 = erosion(circ_image, selem_circ)
imshow(erosion(circ_image2, selem_circ));
```



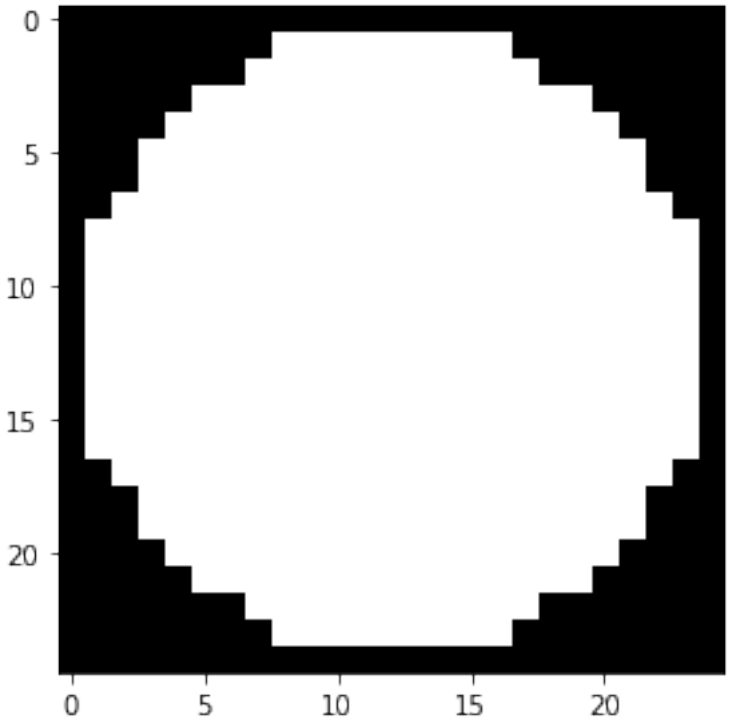
In contrast, as we will see below, the dilation process has enlarged the effective image. Here also we can easily the effects multiple dilations will have on the image.

[91]:
```
from skimage.morphology import dilation

imshow(dilation(circ_image, selem_circ));
```



Now, what happend to our image? Again, try to apply it multiple times to see the effect more clearly:

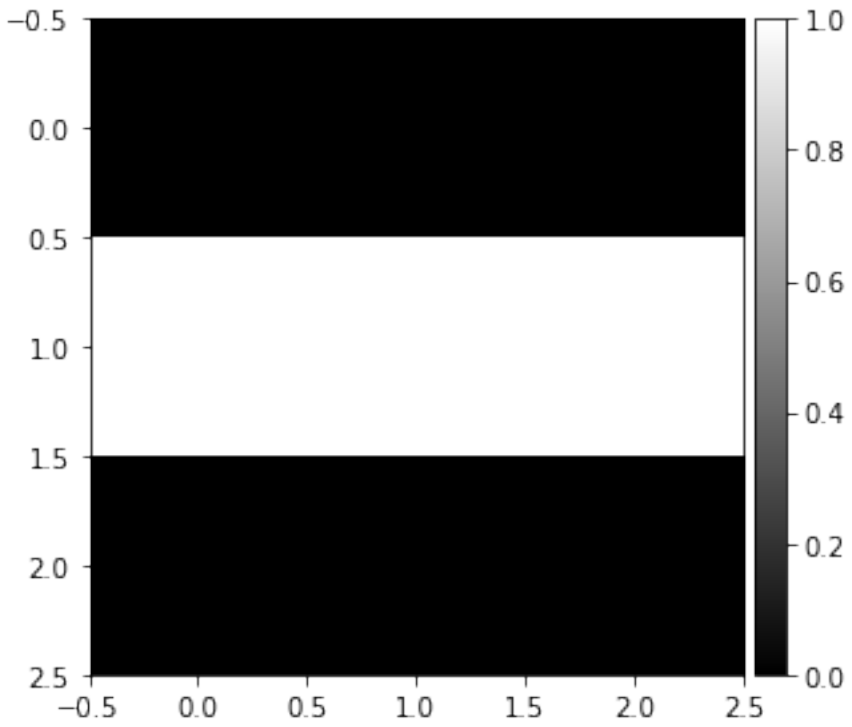
[92]:
```
circ_image2 = dilation(circ_image, selem_circ)
imshow(dilation(circ_image2, selem_circ));
```



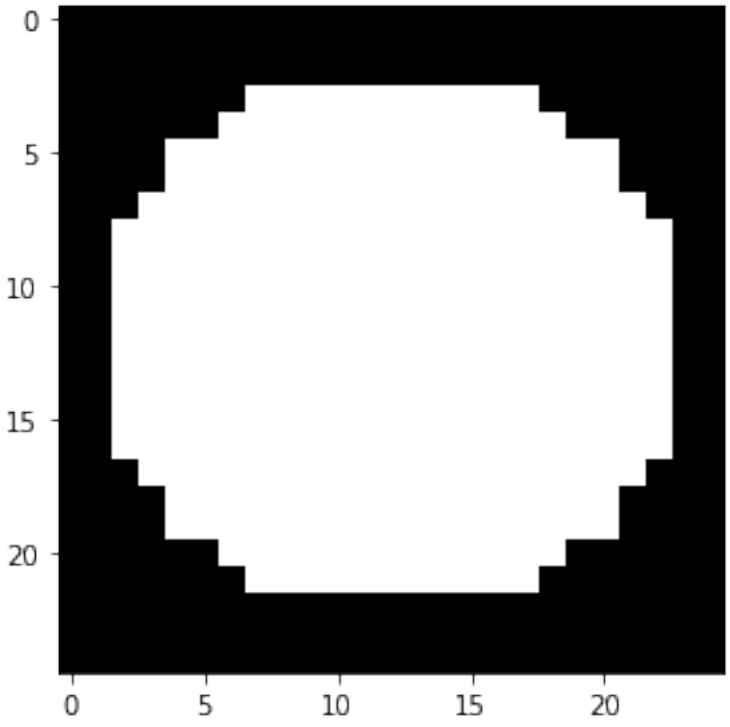
Now, one may wonder, why have such operations at all? It all ties back to how computers simply read images as matrices upon matrices. While the delineation of objects is very clear to us, it is not the same with computer vision. When having the computer identify different components of an image (e.g. fruits, people, roads, forest cover, etc.) the different image processing can result in too many or too little detail being retained as a result of our filtering/image enhancement/segmentation/etc. Morphological operations come in to help either clean an image further, or to reconstruct lost portions.

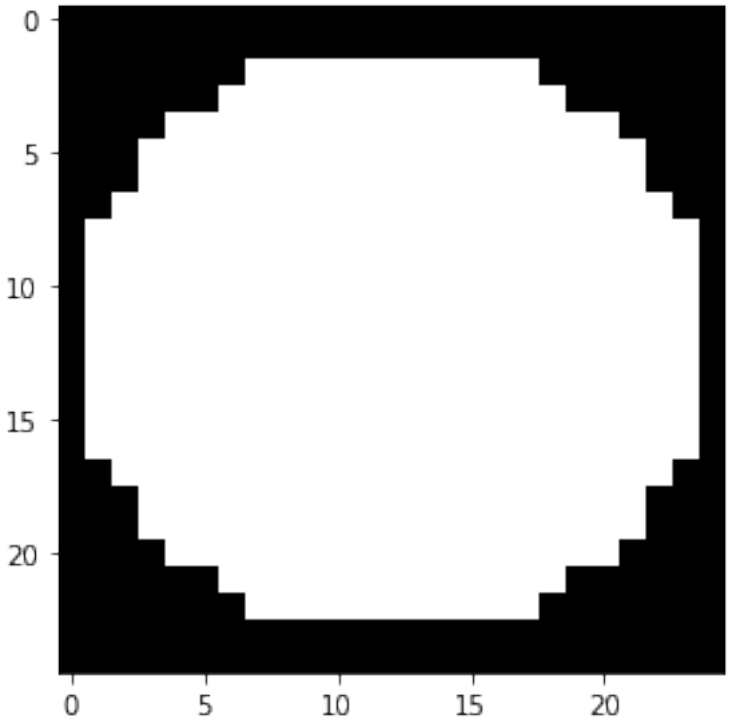
Exercise: In the series of examples below, we see that using dilation expands the image in the direction of the kernel (wider when horizontal, taller when vertical); while, inversely, the image appears compressed along the axis of the kernel when using erosion (narrower for horizontal, shorter for vertical).

[93]:
```
selem_hor = np.array([[0,0,0],
                      [1,1,1],
                      [0,0,0]])
imshow(selem_hor, cmap='gray');
```

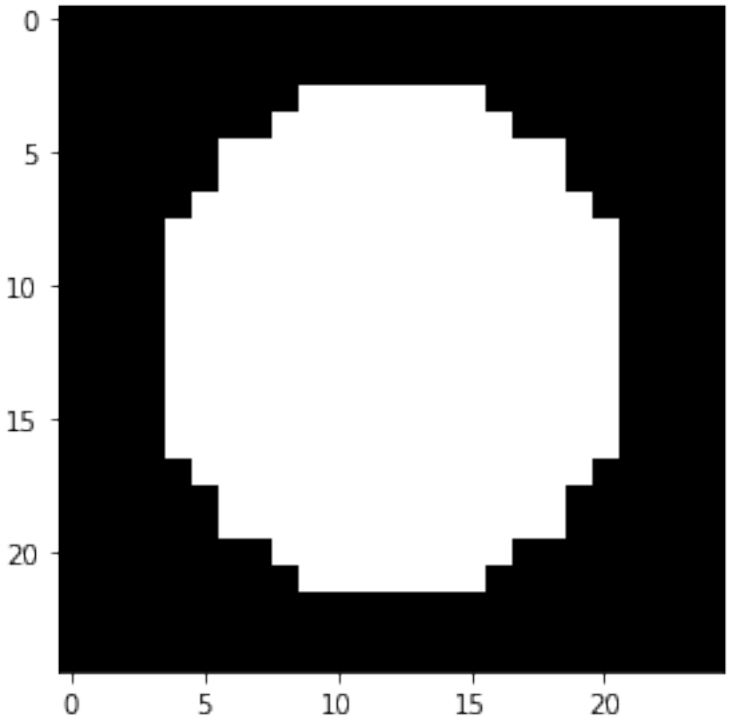


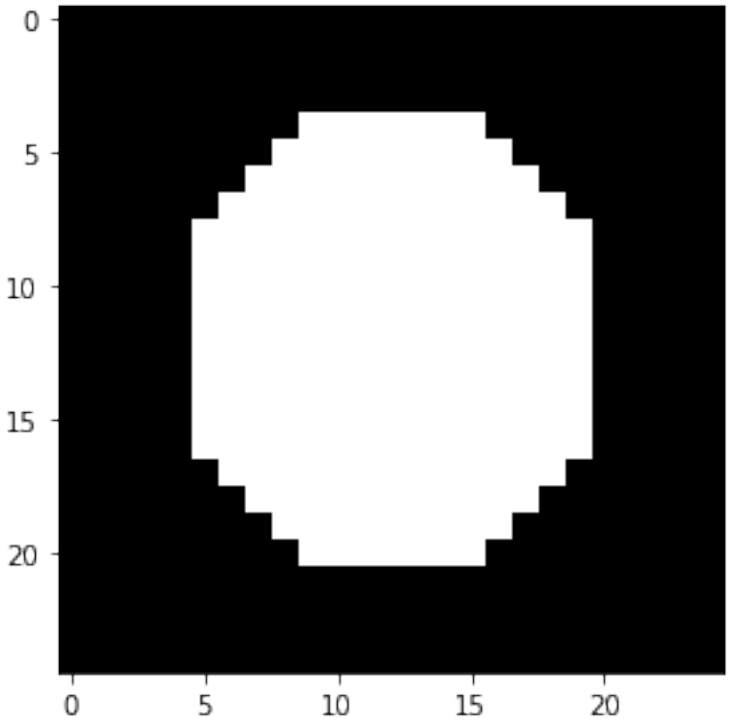
[94]:
```
imshow(dilation(circ_image, selem_hor))
plt.show()
imshow(dilation(dilation(circ_image, selem_hor)));
```


0
5
10
15
20
0
5
10
15
20

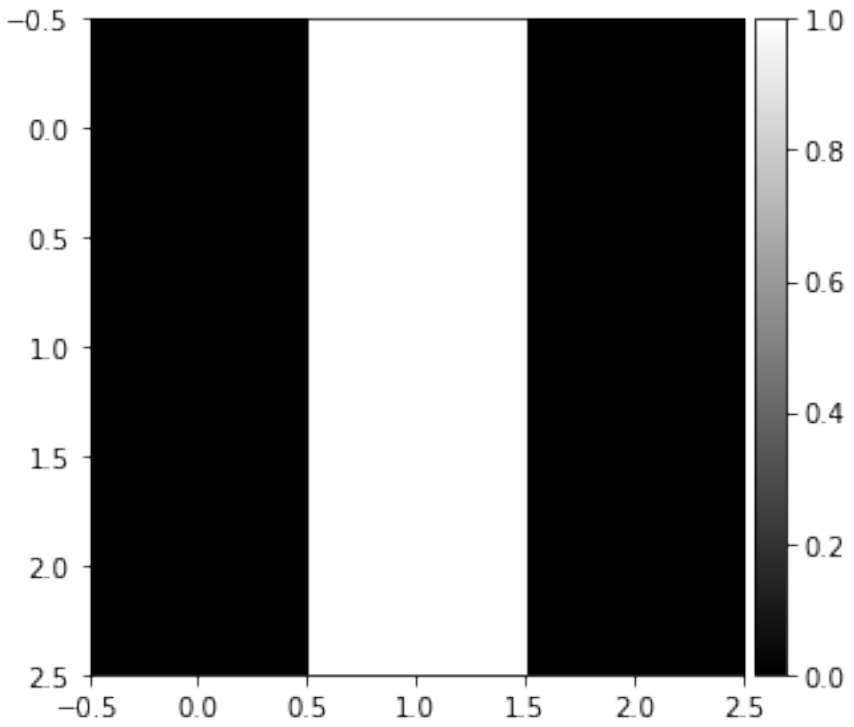

0
5
10
15
20
0
5
10
15
20

[95]:
```
imshow(erosion(circ_image, selem_hor))
plt.show()
imshow(erosion(erosion(circ_image, selem_hor)));
```

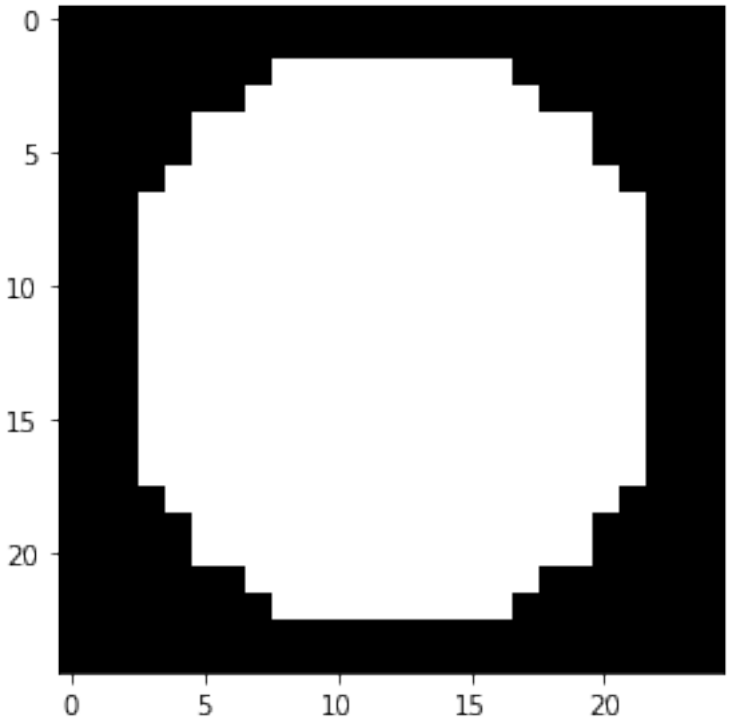


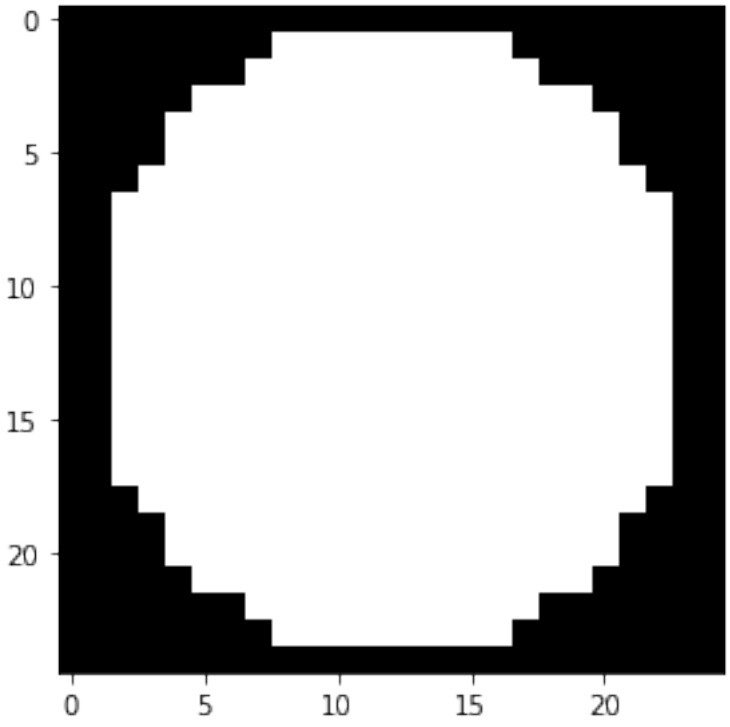


[96]:
```
selem_ver = np.array([[0,1,0],
                      [0,1,0],
                      [0,1,0]])
imshow(selem_ver, cmap='gray');
```

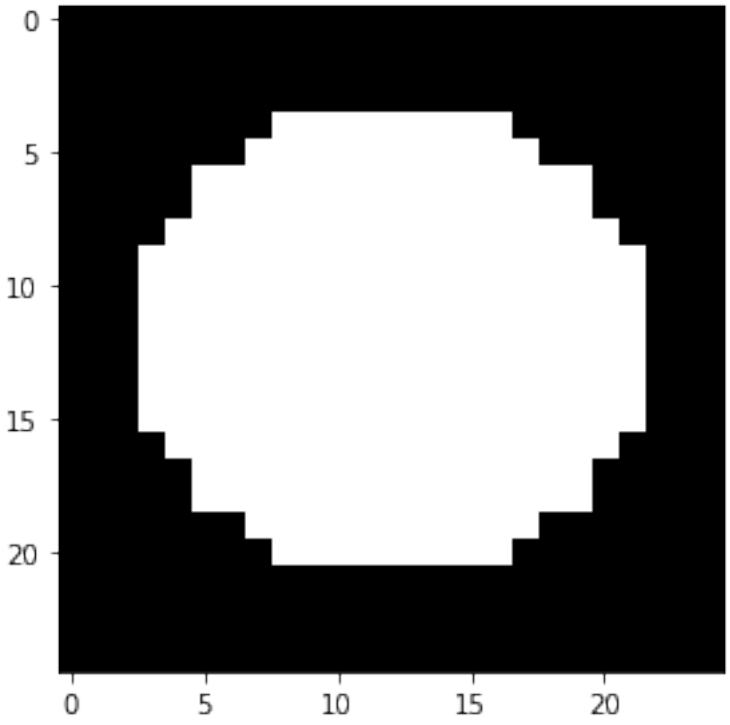


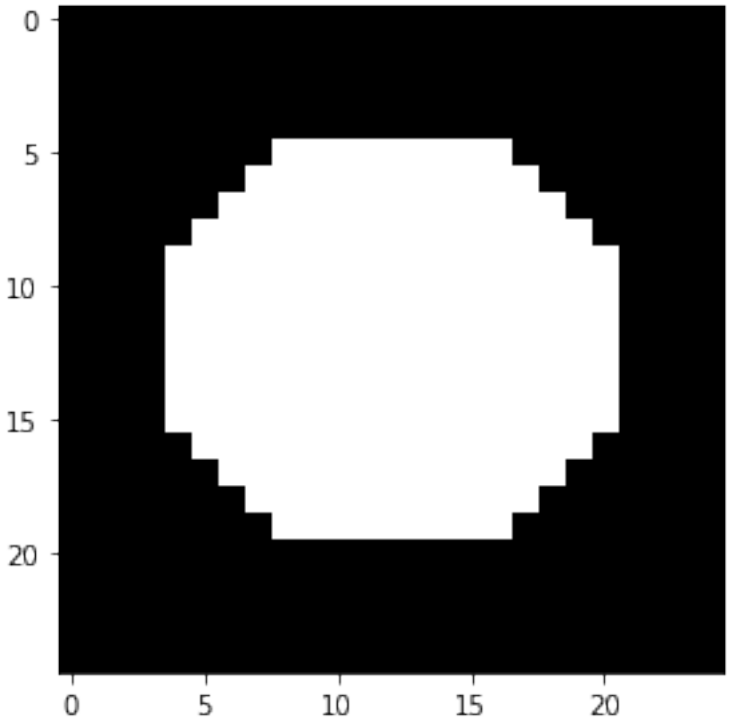
[97]:
```
imshow(dilation(circ_image, selem_ver))
plt.show()
imshow(dilation(dilation(circ_image, selem_ver)));
```


0
5
10
15
20
0
5
10
15
20


0
5
10
15
20
0
5
10
15
20

[98]:
```
imshow(erosion(circ_image, selem_ver))
plt.show()
imshow(erosion(erosion(circ_image, selem_ver)));
```

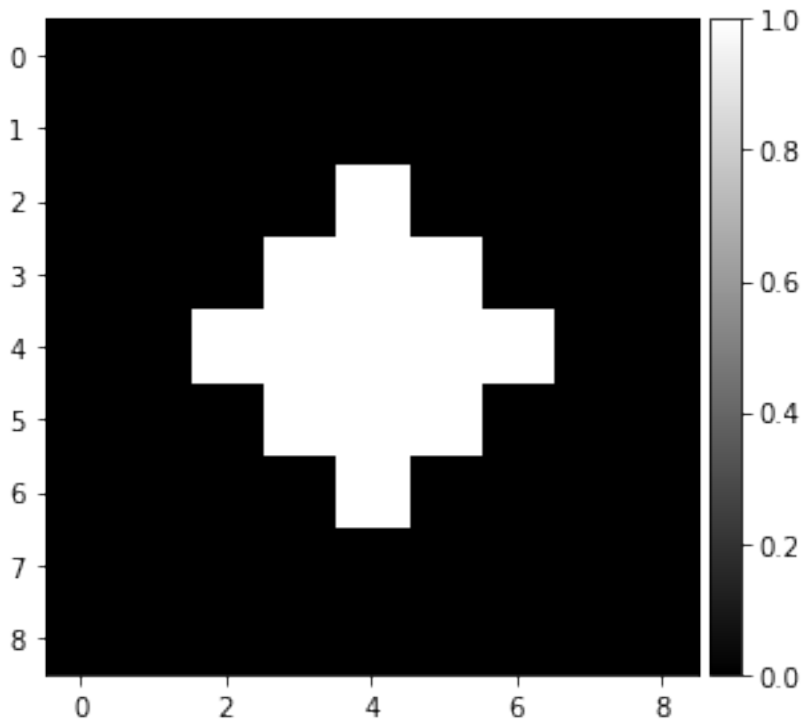




Exercise: Finally, try to find a structuring element that will transform the following image:

[99]:
```
im1 = np.array([[0,0,0,0,0,0,0,0,0],
                [0,0,0,0,0,0,0,0,0],
                [0,0,0,0,1,0,0,0,0],
                [0,0,0,1,1,1,0,0,0],
                [0,0,1,1,1,1,1,0,0],
                [0,0,0,1,1,1,0,0,0],
                [0,0,0,0,1,0,0,0,0],
                [0,0,0,0,0,0,0,0,0],
                [0,0,0,0,0,0,0,0,0]])
imshow(im1, cmap='gray')
```

[99]:
```
<matplotlib.image.AxesImage at 0x7f8a1d78e310>
```

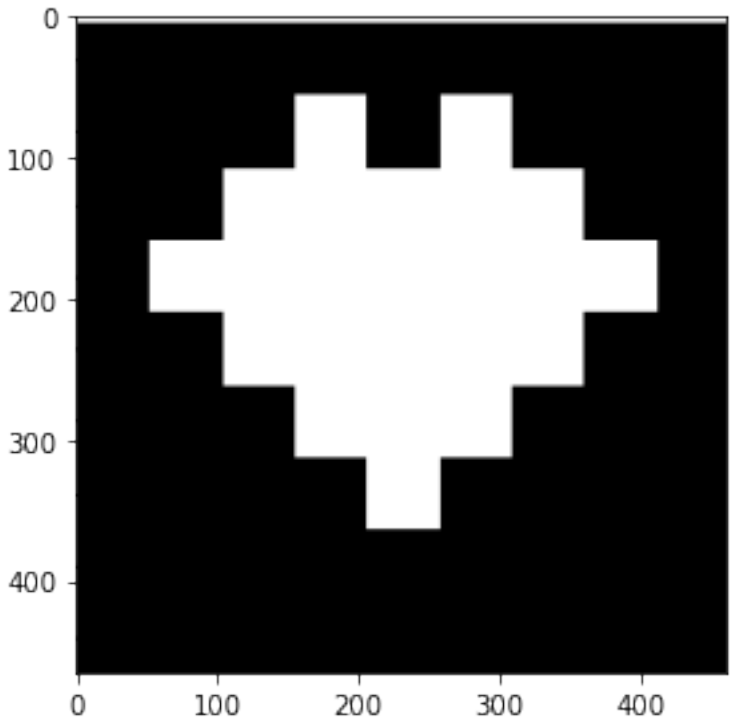


Into this:

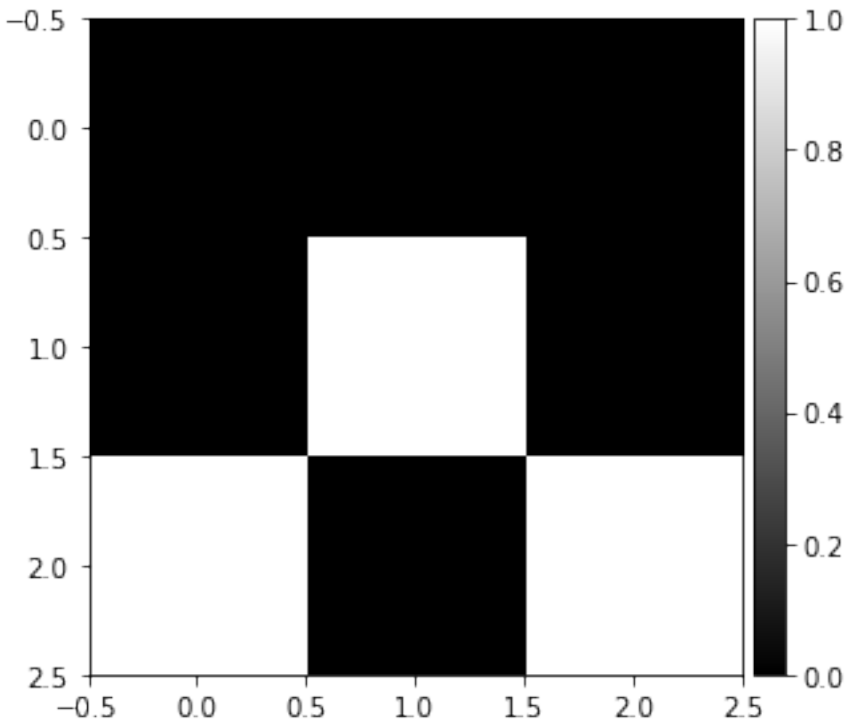
[100]:
```
heart = rgb2gray(imread('heart.PNG'))
imshow(heart)
```

```
<ipython-input-100-a5de21cd8742>:1: FutureWarning: Non RGB image conversion is
now deprecated. For RGBA images, please use rgb2gray(rgba2rgb(rgb)) instead. In
version 0.19, a ValueError will be raised if input image last dimension length
is not 3.
  heart = rgb2gray(imread('heart.PNG'))
```

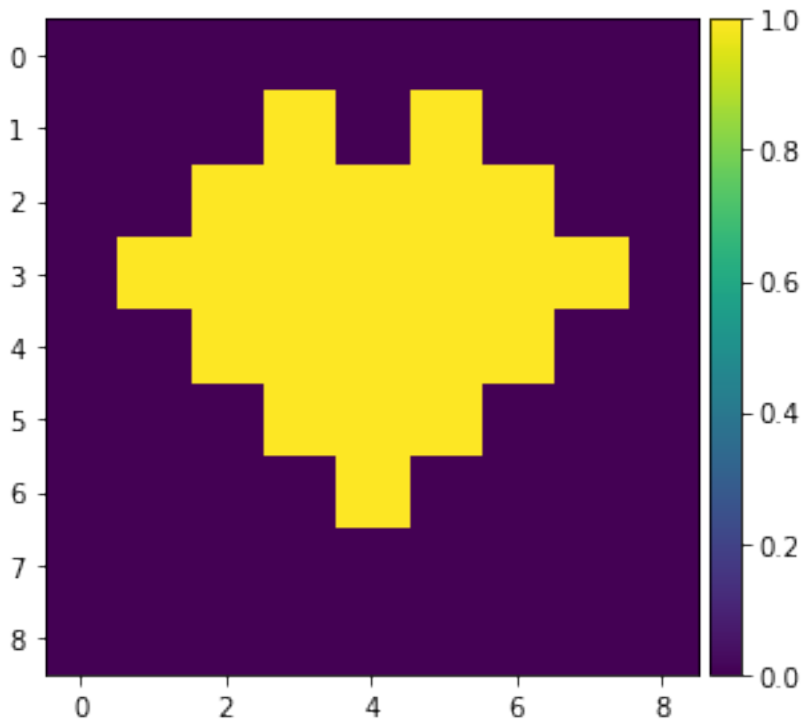
[100]:
```
<matplotlib.image.AxesImage at 0x7f8a17bd1d30>
```



[101]:
```
selem_circ = np.array([[0,0,0],
                       [0,1,0],
                       [1,0,1]])
imshow(selem_circ, cmap='gray');
```



[102]:
```
from skimage.morphology import dilation

imshow(dilation(im1, selem_circ));
```



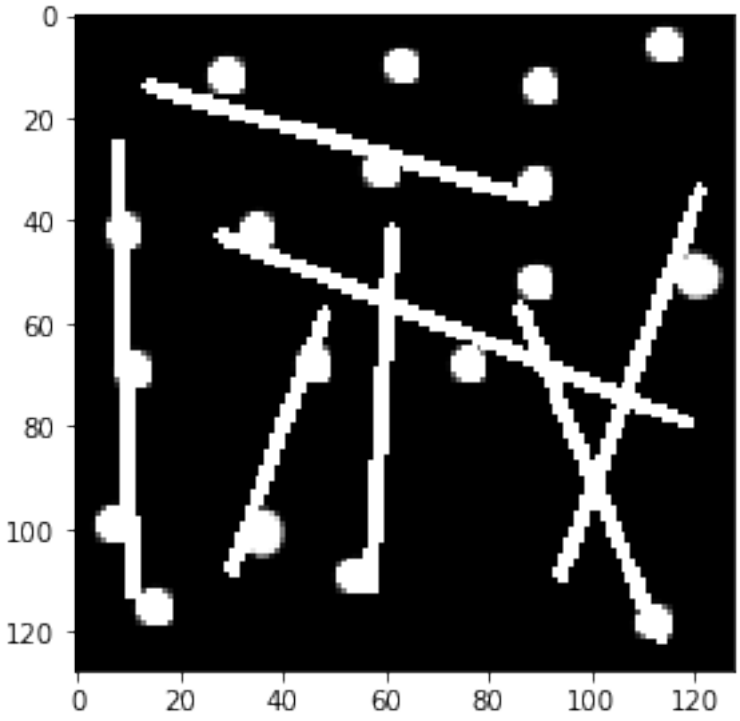
From the above we see that, especially at key ratios (and lower resolutions) the shape of the kernel has very specific effects on the starting image. This is important to note for future applications as the results of the morphological operations are dependent on the appropriate kernel design.

1.2.2 2.2 Other morphological operations (Breakout)

Given the erosion and dilation, we can perform iterations of these two operations to do opening and closing. Opening and closing are successive iterations of erosion and dilation. To better appreciate this, let's look at the following images:

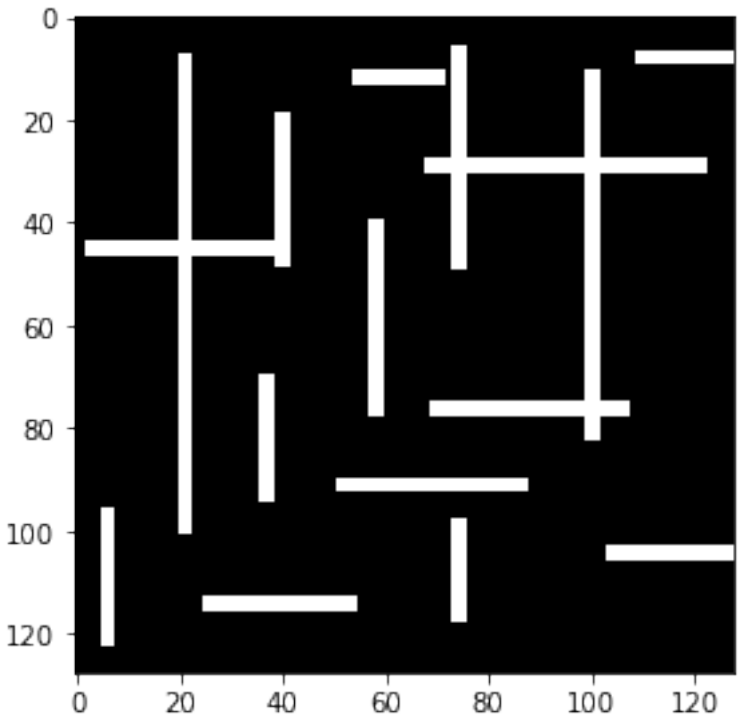
[103]:
```
art1 = rgb2gray(imread('art1.PNG'))
imshow(art1)
```

[103]:
```
<matplotlib.image.AxesImage at 0x7f8a1d4b7f10>
```



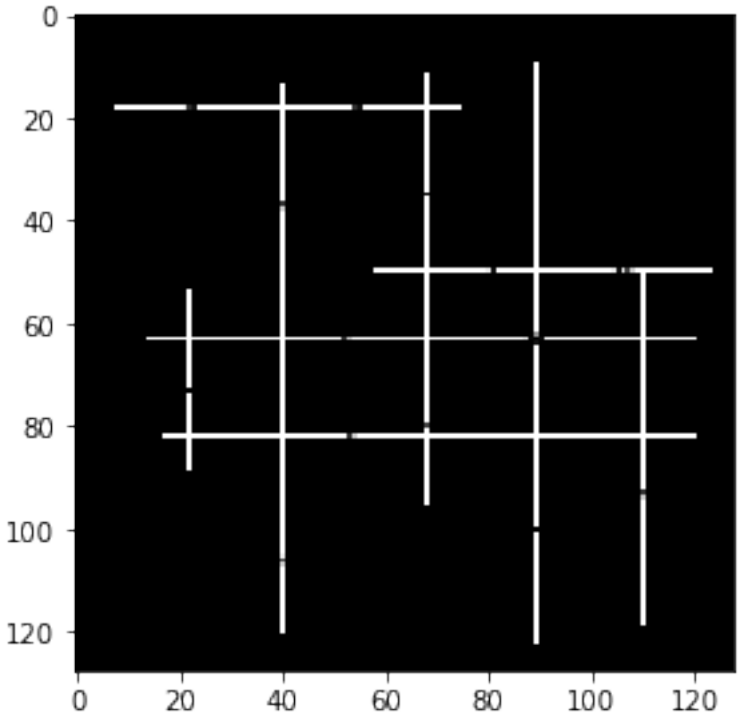
[104]:
```
art2 = rgb2gray(imread('art2.png'))
imshow(art2)
```

[104]: <matplotlib.image.AxesImage at 0x7f8a1d8a7ee0>



[105]:
```
art3 = rgb2gray(imread('art3.png'))
imshow(art3)
```

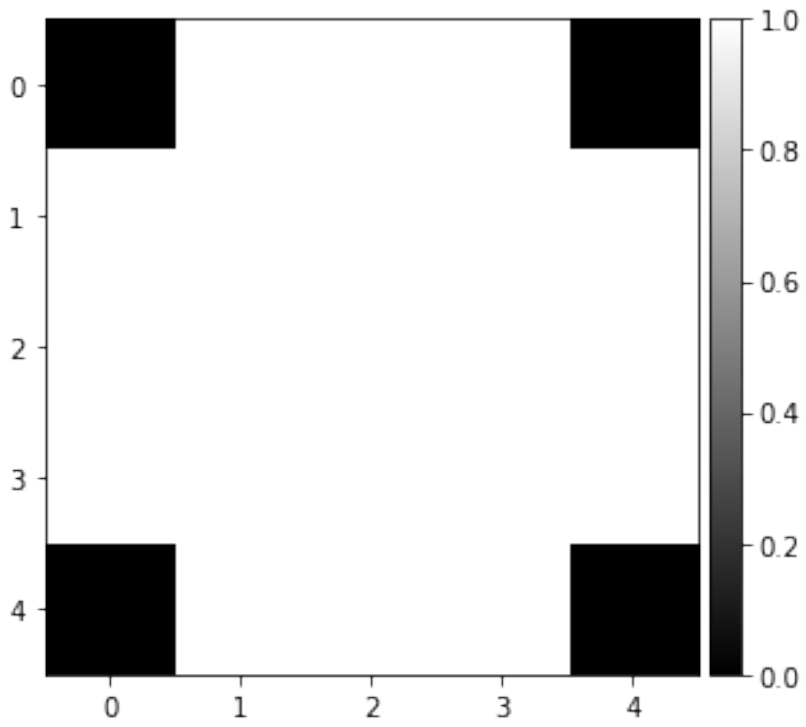
[105]: <matplotlib.image.AxesImage at 0x7f8a1d8e6d30>



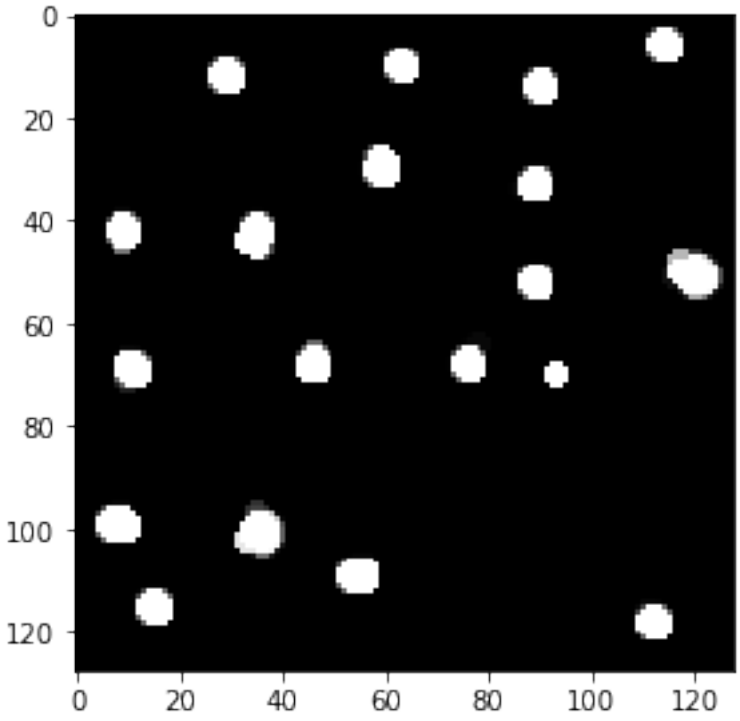
[106]:
```
from skimage.morphology import opening, closing
# it follows the same syntax as dilation and closing, i.e. operation(image,␣
 ↪structuring element)
```

Produce the following images: 1. Art 1 with only the circles visible 1. Art 2 with only either the horizontal or vertical lines visible. 1. Art 3 with the gaps connected.

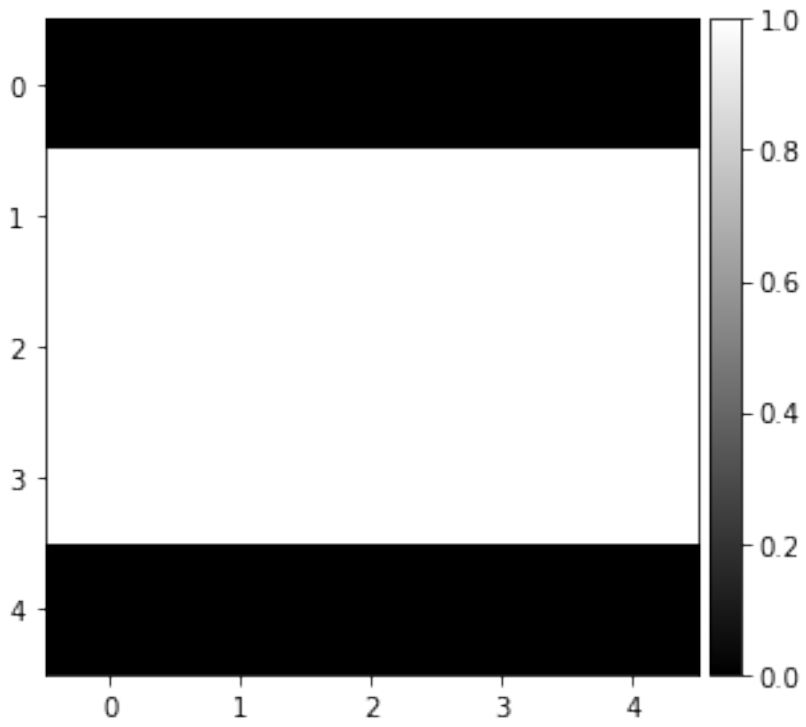
[107]:
```
k_art1 = np.array([[0,1,1,1,0],
                   [1,1,1,1,1],
                   [1,1,1,1,1],
                   [1,1,1,1,1],
                   [0,1,1,1,0]])
imshow(k_art1, cmap='gray');
```



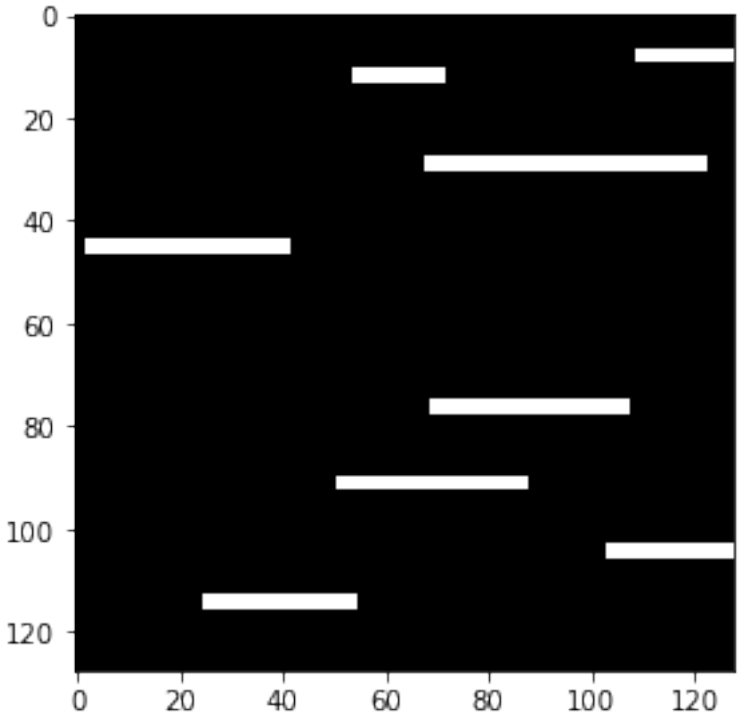
[108]:
```
art3 = rgb2gray(imread('art1.PNG'))

imshow(opening(art1, k_art1));
```



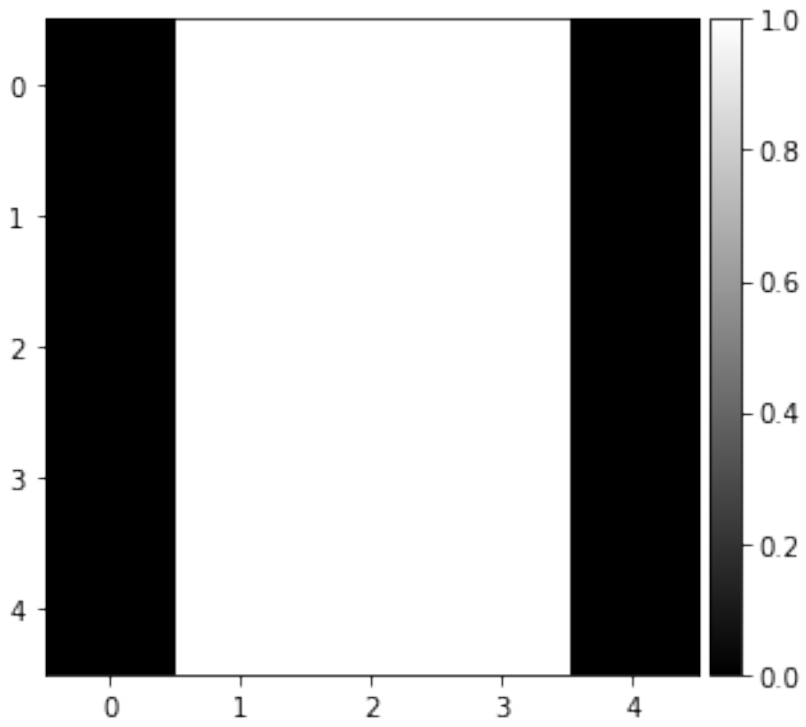
[109]:
```
k_art2h = np.array([[0,0,0,0,0],
                    [1,1,1,1,1],
                    [1,1,1,1,1],
                    [1,1,1,1,1],
                    [0,0,0,0,0]])
imshow(k_art2h, cmap='gray');
```



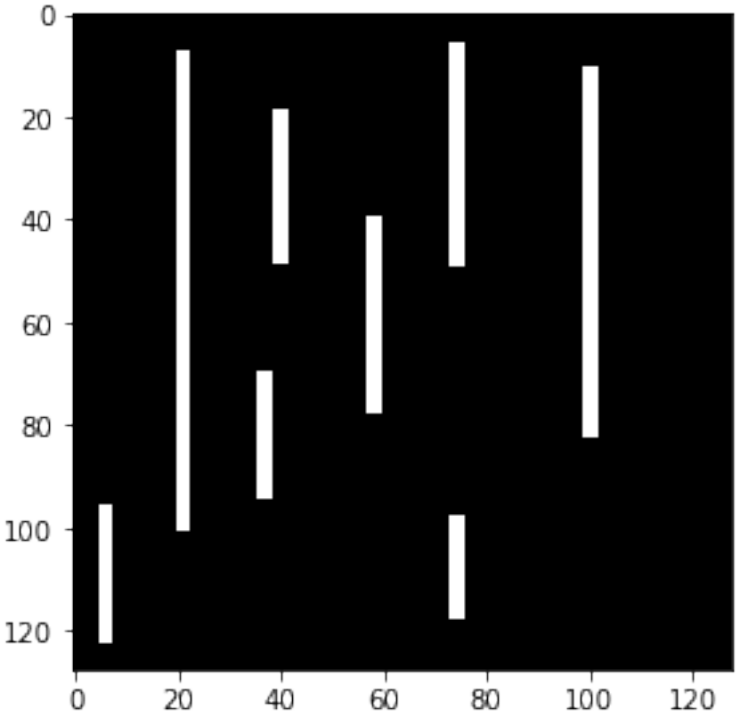
[110]:
```
art2 = rgb2gray(imread('art2.png'))

imshow(opening(art2, k_art2h));
```



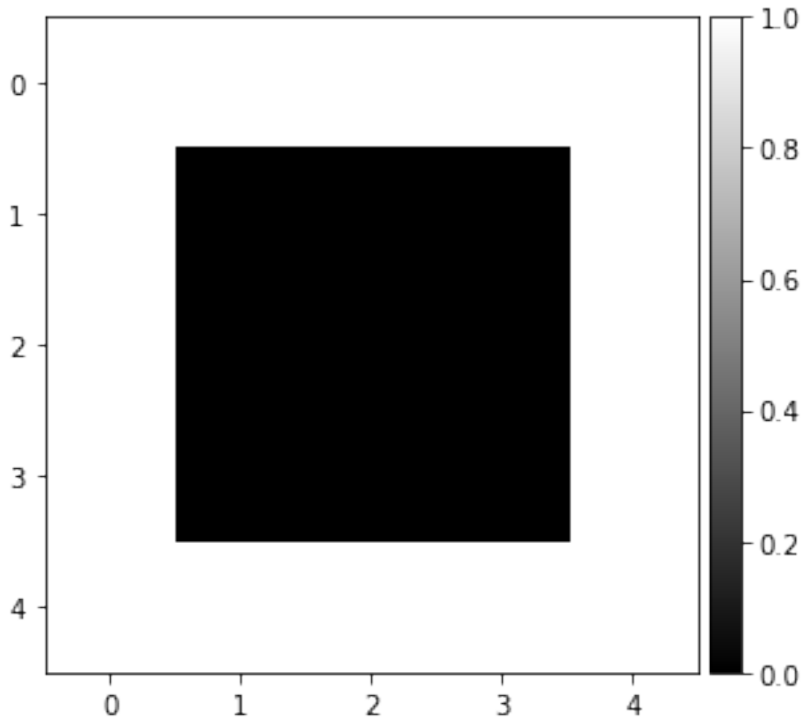
[111]:
```
k_art2v = np.array([[0,1,1,1,0],
                    [0,1,1,1,0],
                    [0,1,1,1,0],
                    [0,1,1,1,0],
                    [0,1,1,1,0]])
imshow(k_art2v, cmap='gray');
```



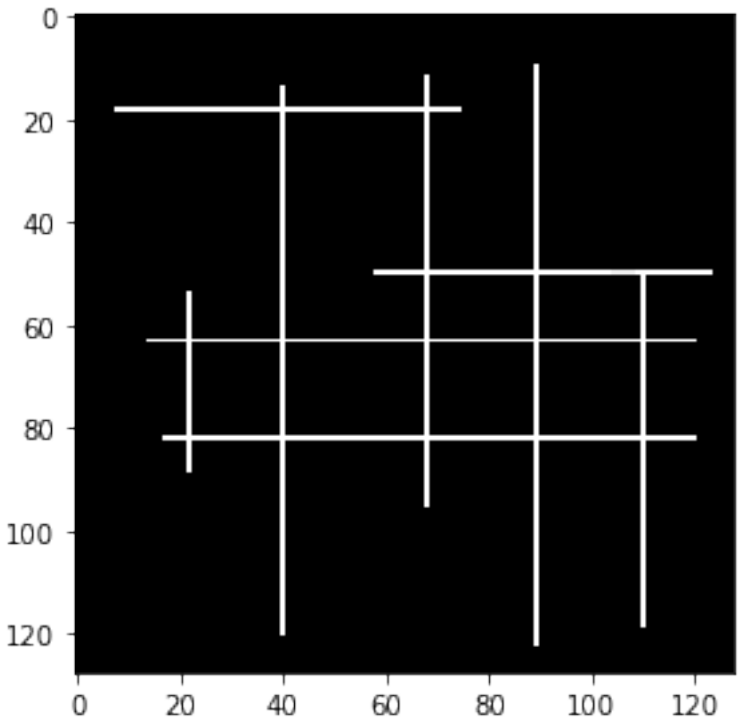
[112]:
```
imshow(opening(art2, k_art2v));
```



[113]:
```
k_art3 = np.array([[1,1,1,1,1],
                   [1,0,0,0,1],
                   [1,0,0,0,1],
                   [1,0,0,0,1],
                   [1,1,1,1,1]])
imshow(k_art3, cmap='gray');
```



[114]:
```
art3 = rgb2gray(imread('art3.png'))

imshow(art3)
imshow(closing(art3, k_art3));
```



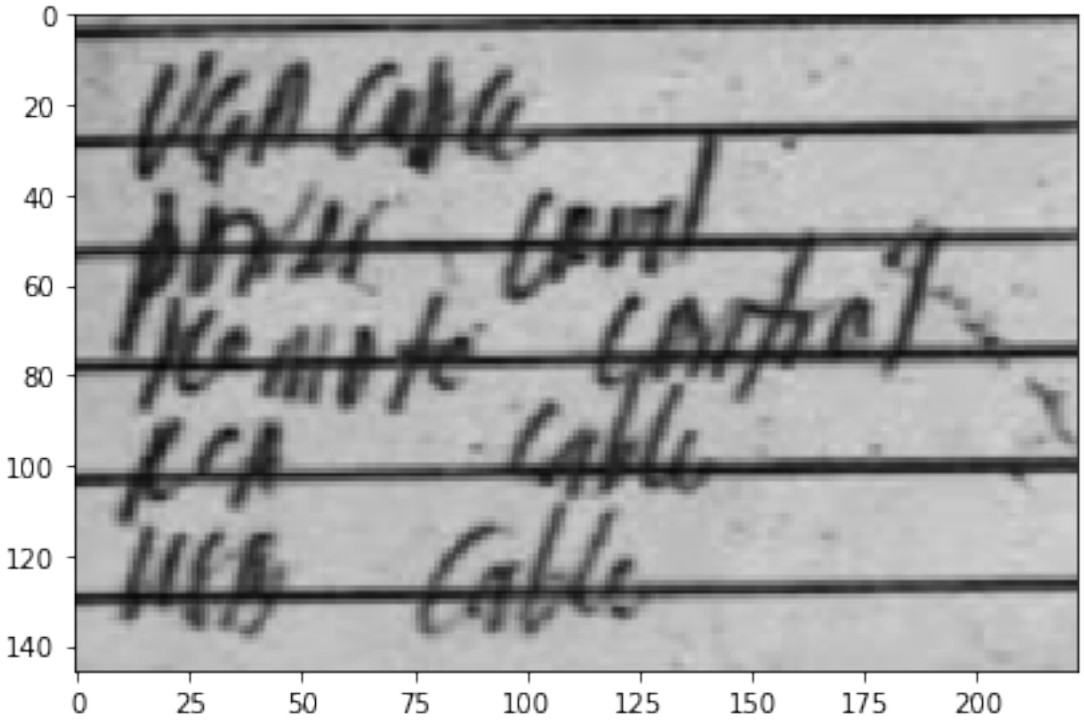
1.2.3 2.3 Applications of Morphological Operations (Asynchronous)

Given our discussion, let's apply it to an optical character recognition (OCR) application. Supposed that we need to digitize the following document:

Use the skills we learned in class to prepare the snippet of the image such that only the texts are visible.

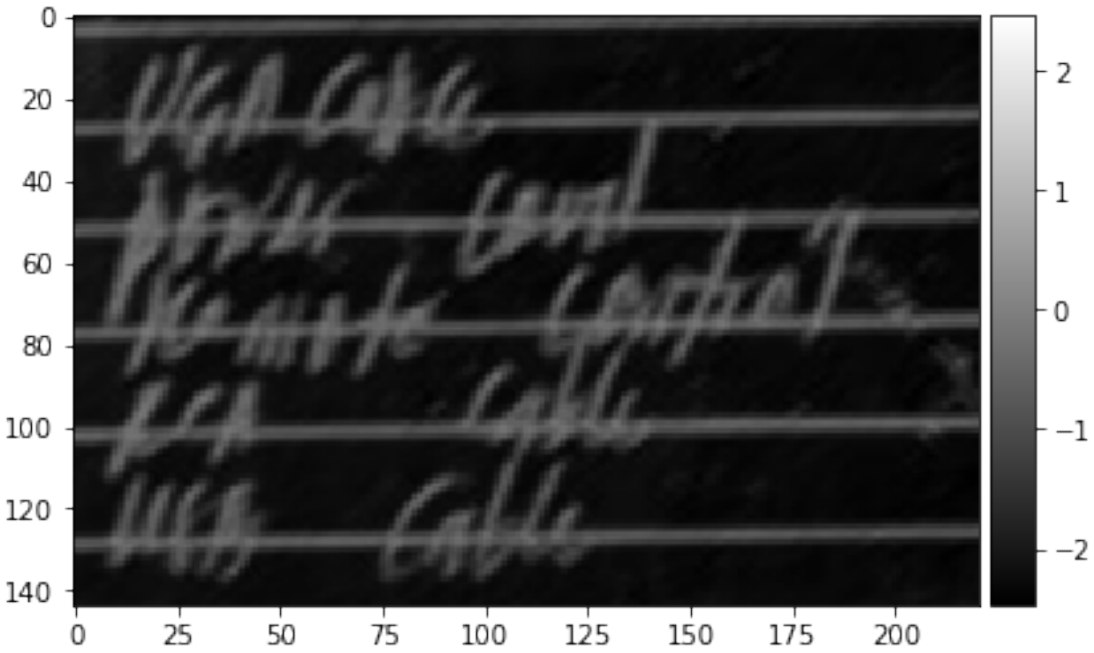
[144]:
```
receipt = rgb2gray(imread('receipt2.PNG'))

imshow(receipt);
```

```
<ipython-input-144-ca1c5c86b730>:1: FutureWarning: Non RGB image conversion is
now deprecated. For RGBA images, please use rgb2gray(rgba2rgb(rgb)) instead. In
version 0.19, a ValueError will be raised if input image last dimension length
is not 3.
  receipt = rgb2gray(imread('receipt2.PNG'))
```

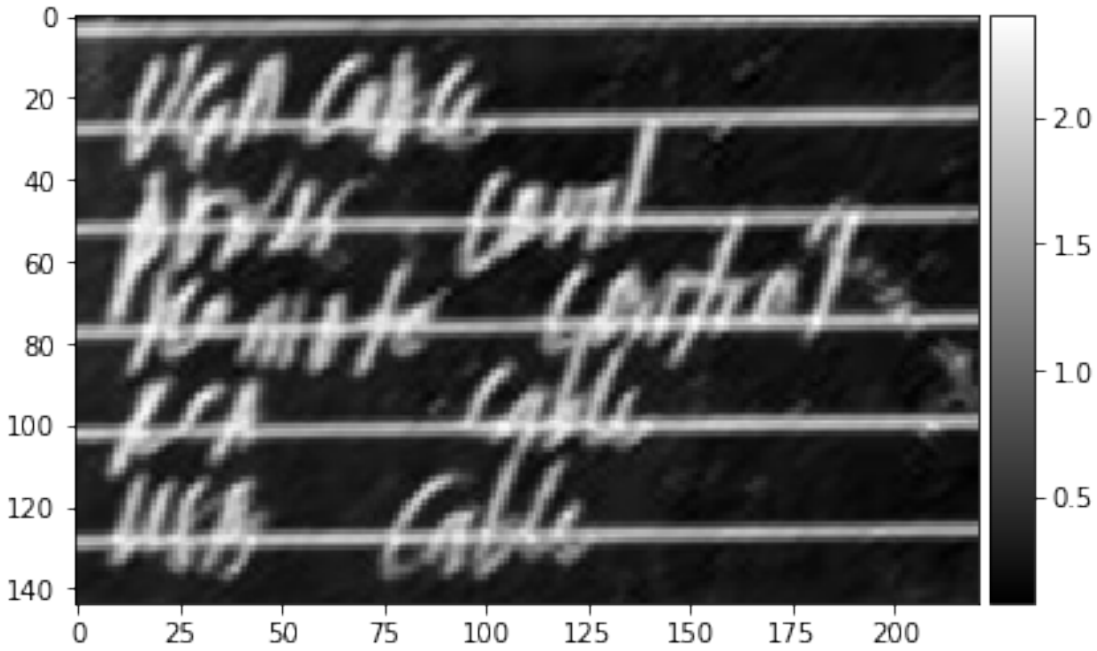


[154]:
```
# receipt_i = convolve2d(receipt, kernel3, 'valid')
# imshow(receipt_i, cmap='gray')
```

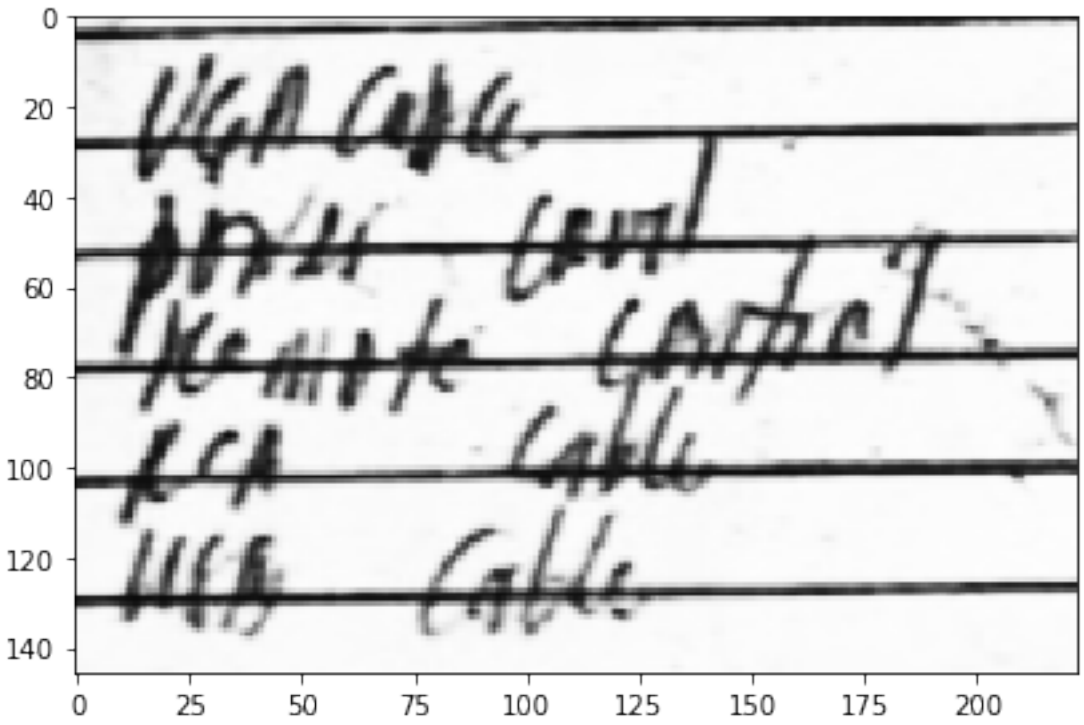
[154]: <matplotlib.image.AxesImage at 0x7f8a171c0dc0>



[227]:
```
# imshow(receipt_i+2.55, cmap='gray')
# receipt2 = receipt_i+4
```

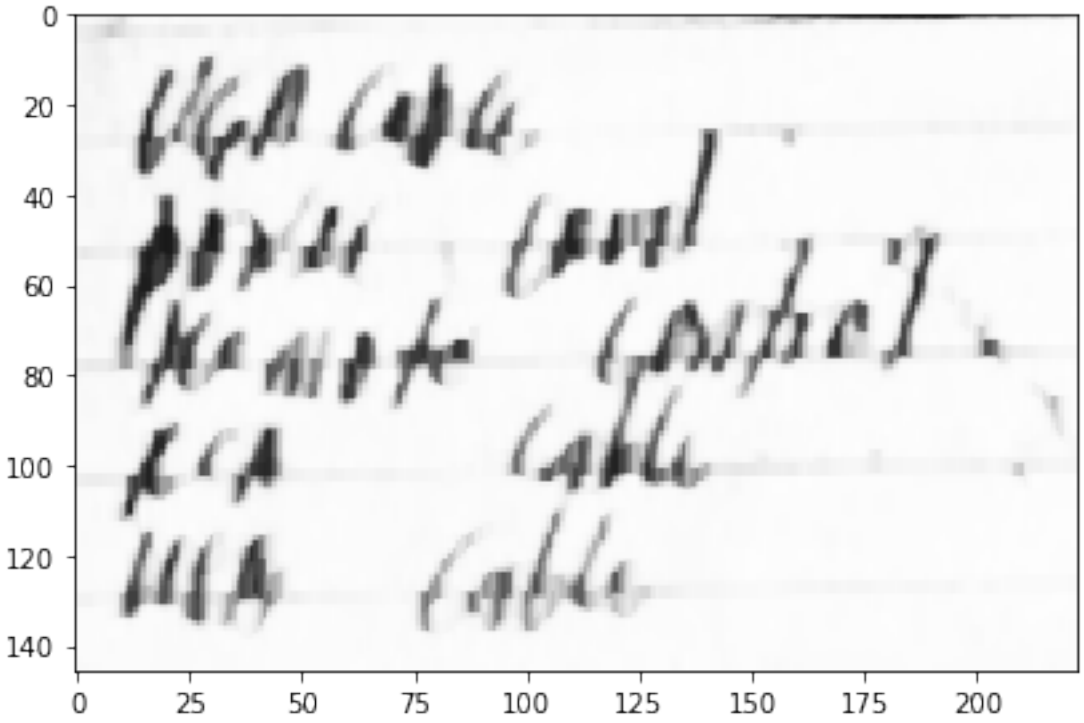


[229]:
```
from skimage.exposure import adjust_sigmoid, rescale_intensity
# receipt3 = adjust_sigmoid(receipt2, 1.4)
receipt3 = adjust_sigmoid(receipt, 0.35)
imshow(receipt3, cmap='gray');
```



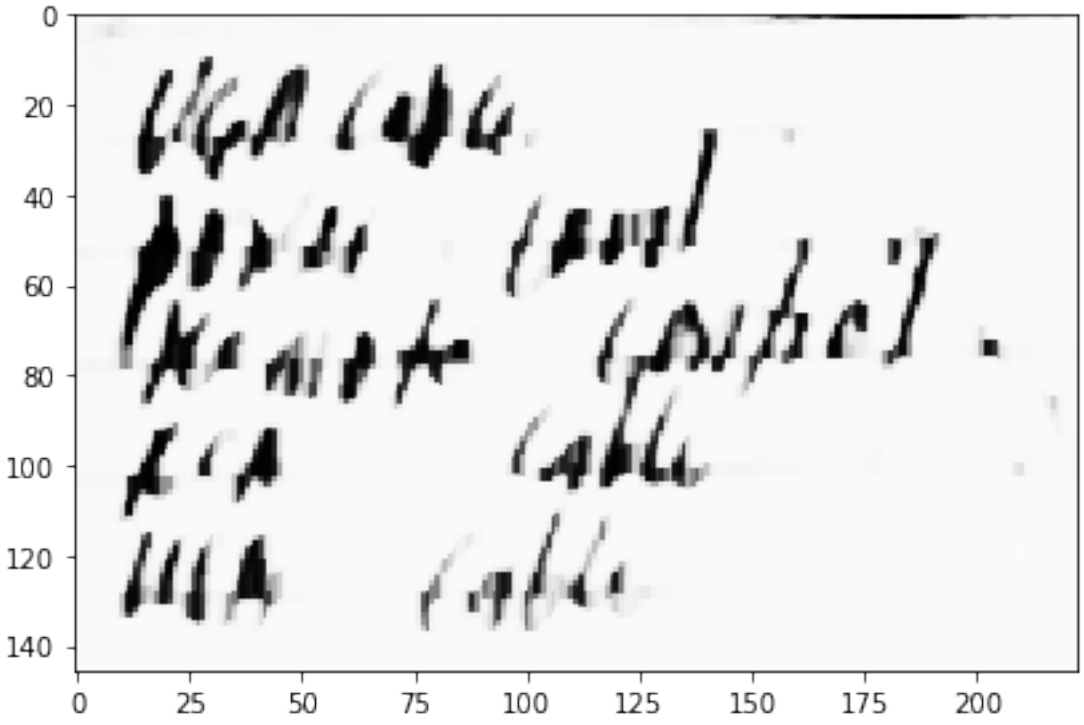
[254]:
```
k_ = np.zeros((3,3))
k_[1,:] = 1

def morph(im, f=dilation, selem=k_.T, times=1):
    for i in range(times):
        im = f(im, selem)
    return im

receipt4 = morph(morph(receipt3), closing, times=3)
imshow(receipt4, cmap='gray')
```

[254]: <matplotlib.image.AxesImage at 0x7f8a154eae50>



[259]:
```
imshow(adjust_sigmoid(receipt4, 0.6), cmap='gray')
```

[259]: <matplotlib.image.AxesImage at 0x7f8a152c9220>


0
20
40
60
80
100
120
140
0
25
50
75
100
125
150
175
200

[ ]: