Lab 4 - Morphological Image Processing

Task 1: Dilation and Erosion

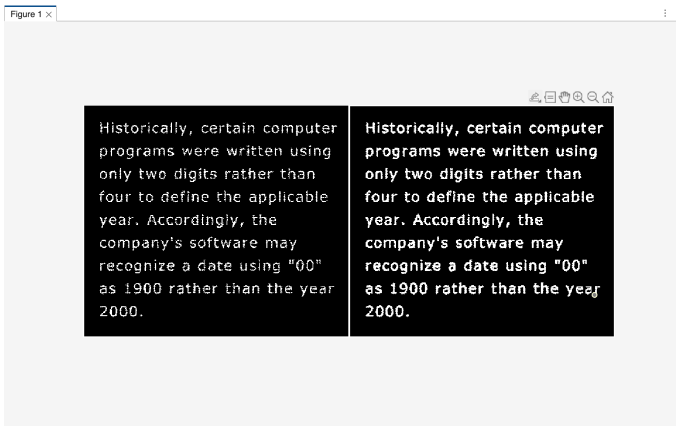
Matlab provides a collection of morphological functions. Here is a list of them:

`imerode`	Erode image
`imdilate`	Dilate image
`imopen`	Morphologically open image
`imclose`	Morphologically close image
`imtophat`	Top-hat filtering
`imbothat`	Bottom-hat filtering
`imclearborder`	Suppress light structures connected to image border
`imkeepborder`	Retain light structures connected to image border *(Since R2023b)*
`imfill`	Fill image regions and holes
`bwhitmiss`	Binary hit-miss operation
`bwmorph`	Morphological operations on binary images
`bwmorph3`	Morphological operations on binary volume
`bwperim`	Find perimeter of objects in binary image
`bwskel`	Reduce all objects to lines in 2-D binary image or 3-D binary volume
`bwulterode`	Ultimate erosion

Task 1.1 Code:

```
A = imread('assets/text-broken.tif');
B1 = [0 1 0;
1 1 1;
0 1 0];     % create structuring element
A1 = imdilate(A, B1);
montage({A,A1})
```

Output:



Historically, certain computer programs were written using only two digits rather than four to define the applicable year. Accordingly, the company's software may recognize a date using "00" as 1900 rather than the year 2000.

Historically, certain computer programs were written using only two digits rather than four to define the applicable year. Accordingly, the company's software may recognize a date using "00" as 1900 rather than the year 2000.

Observation:

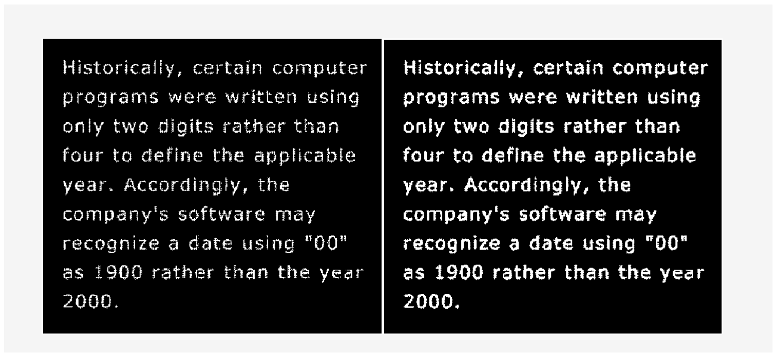
The original image contains broken white text with small gaps in the character strokes. Using a 3×3 cross-shaped structuring element, dilation expands white pixels in the up/down/left/right directions. As a result, the letters become thicker, and the small gaps are filled, effectively repairing the broken text.

Task 1.2: Change the structuring element (SE) to all 1's. Instead of enumerating it, you can do that with the function *ones*

Code:

```
B2 = ones(3,3);      % generate a 3x3 matrix of 1's
```

Output:



Historically, certain computer programs were written using only two digits rather than four to define the applicable year. Accordingly, the company's software may recognize a date using "00" as 1900 rather than the year 2000.

Historically, certain computer programs were written using only two digits rather than four to define the applicable year. Accordingly, the company's software may recognize a date using "00" as 1900 rather than the year 2000.

Observation:

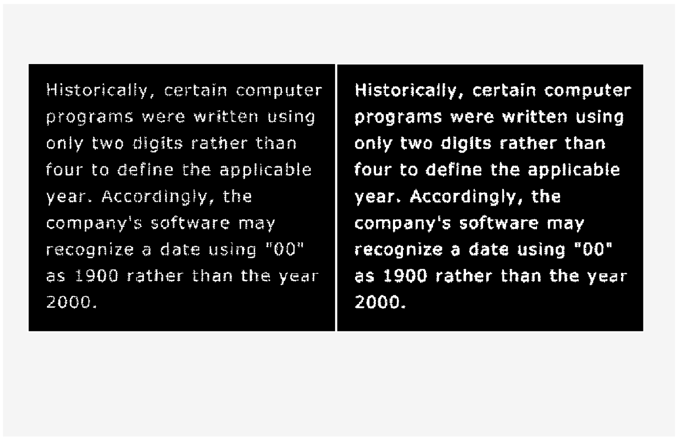
The dilation effects from B1 and B2 are visually very subtle in the full montage. Clear differences only appear when zooming closely into the strokes. This is because the text strokes are only 1–2 pixels thick, so adding one pixel in diagonal directions (B2) produces only minor changes. Despite the small visual difference, B2 does perform dilation in all 8 directions, making the text slightly bolder and more rounded.

Task 1.3: Try making the SE larger. Try to make the SE diagonal cross

Code:

```
Bx = [1 0 1;
      0 1 0;
      1 0 1];
```

Output:



Historically, certain computer programs were written using only two digits rather than four to define the applicable year. Accordingly, the company's software may recognize a date using "00" as 1900 rather than the year 2000.

Historically, certain computer programs were written using only two digits rather than four to define the applicable year. Accordingly, the company's software may recognize a date using "00" as 1900 rather than the year 2000.

Observation:

The diagonal-cross structuring element produces little to no visible change in the broken text image. This is because the text is composed mainly of horizontal and vertical strokes, while Bx only expands pixels along diagonal directions. Since there are no diagonal gaps to repair, dilation with Bx has a very minimal effect.

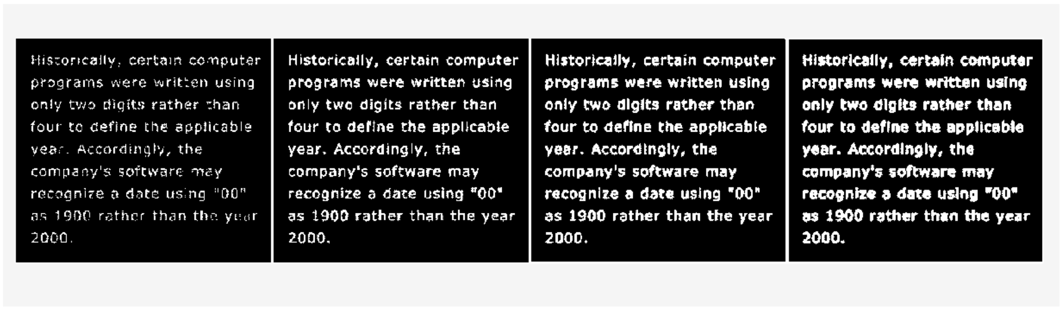
Challenge 1:

What happens if you dilate the original image with B1 twice (or more times)?

Code:

```
A1 = imdilate(A, B1);
A2 = imdilate(A1, B1);
A3 = imdilate(A2, B1);
montage({A, A1, A2, A3}, "Size", [1 4])
```

Output:


Historically, certain computer programs were written using only two digits rather than four to define the applicable year. Accordingly, the company's software may recognize a date using "00" as 1900 rather than the year 2000.

Historically, certain computer programs were written using only two digits rather than four to define the applicable year. Accordingly, the company's software may recognize a date using "00" as 1900 rather than the year 2000.

Historically, certain computer programs were written using only two digits rather than four to define the applicable year. Accordingly, the company's software may recognize a date using "00" as 1900 rather than the year 2000.

Historically, certain computer programs were written using only two digits rather than four to define the applicable year. Accordingly, the company's software may recognize a date using "00" as 1900 rather than the year 2000.

Observation:

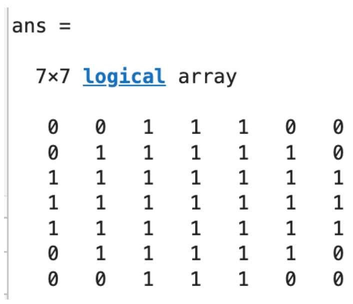
When `imdilate` is applied multiple times using B1, each iteration adds another pixel layer in the four cardinal directions. With 2 or more dilations, the text becomes progressively thicker, small gaps close completely, and adjacent letters may begin to merge. Repeated dilation, therefore, produces stronger smoothing and more aggressive expansion of white regions.

Task 1.4: Generation of a structuring element

Code:

```
SE = strel('disk',4);
SE.Neighborhood          % print the SE neighborhood contents
```

Output:



```
ans =

  7×7 logical array

   0   0   1   1   1   0   0
   0   1   1   1   1   1   0
   1   1   1   1   1   1   1
   1   1   1   1   1   1   1
   1   1   1   1   1   1   1
   0   1   1   1   1   1   0
   0   0   1   1   1   0   0
```

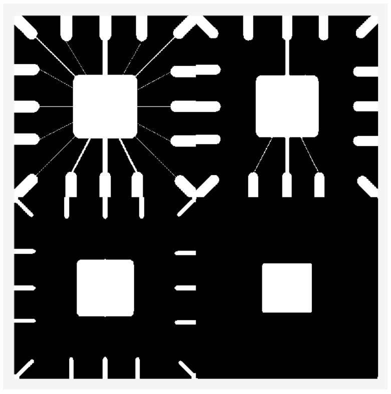
Observation:
The `strel` function generates this shape, and `SE.Neighborhood` displays the binary mask of the structuring element. Like here, `strel('disk',4)` produces a circular mask of 1's, which causes objects in the image to expand or shrink with smooth, rounded boundaries.

Task 1.5 Erosion Operation:

Code:

```
clear all
close all
A = imread('assets/wirebond-mask.tif');
SE2 = strel('disk',2);
SE10 = strel('disk',10);
SE20 = strel('disk',20);
E2 = imerode(A,SE2);
E10 = imerode(A,SE10);
E20 = imerode(A,SE20);
montage({A, E2, E10, E20}, "size", [2 2])
```

Output:



Observation:

Erosion gradually removes white pixels from the image, causing white objects to shrink. Thin white wires disappear first because they are too narrow. Bigger structuring elements remove even more, leaving only the thick white squares. So, the larger the SE, the more the white parts shrink.

Task 2 - Morphological Filtering with Open and Close

I decided to split the task into subtasks, starting with:

Task 2.1:

1. **Read the image file 'finger-noisy.tif' into *f*.**

```
clear all
close all
f = imread('assets/finger-noisy.tif');
imshow(f)
```

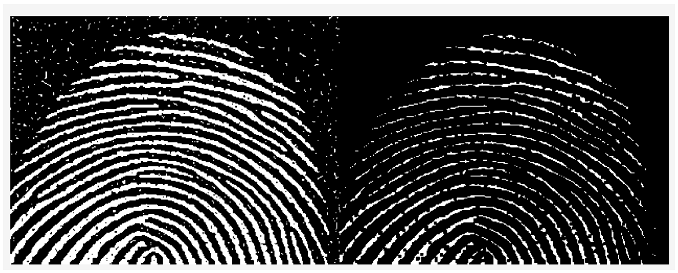
2. **Generate a 3×3 structuring element SE.**

```
>> SE = strel('square',3);
imshow(SE.Neighborhood)
>> SE.Neighborhood

ans =

  3×3 logical array

   1   1   1
   1   1   1
   1   1   1
```

SE.Neighborhood shows the structuring element's internal pixel mask. This 3×3 structuring element will be used for erosion and dilation. Since it's square, it will shrink/expand image objects equally in all directions.

3. **Erode *f* to produce *fe*.**

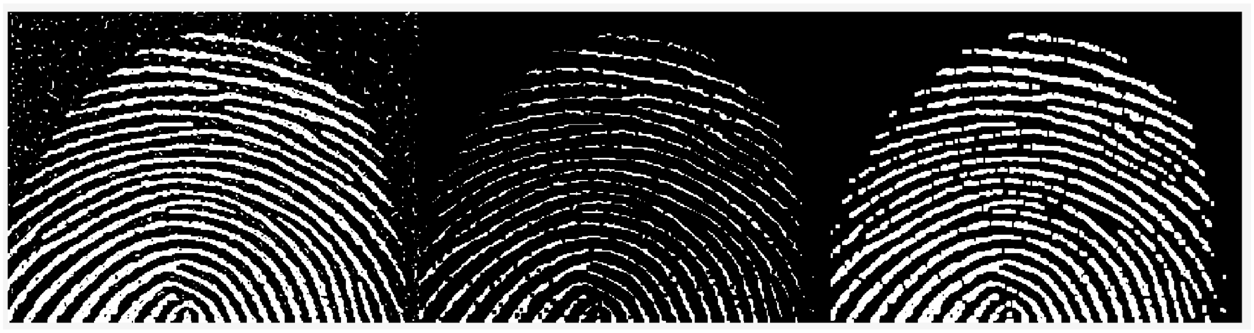
```
SE = strel('square',3);
fe = imerode(f, SE);
imshow(fe)
montage({f, fe}, "Size", [1 2])
```



Erosion (3×3 SE) shrinks the white fingerprint ridges and removes small white noise. Compared to the original image, the eroded image shows thinner ridges, more gaps, and reduced noise.

4. **Dilate *fe* to produce *fed*.**

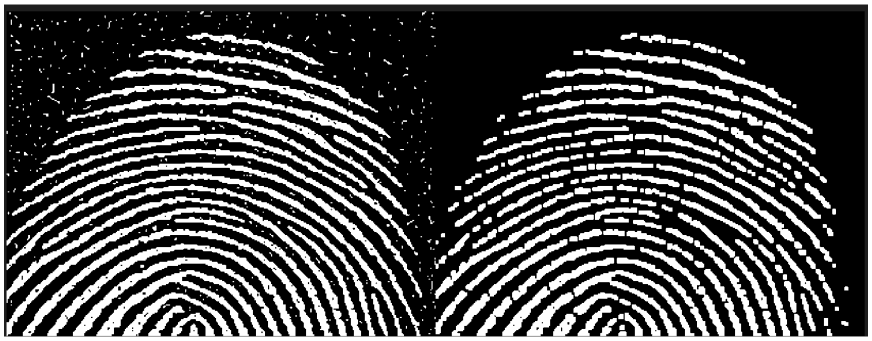
```
fed = imdilate(fe, SE);
montage({f, fe, fed}, "Size", [1 3])
```



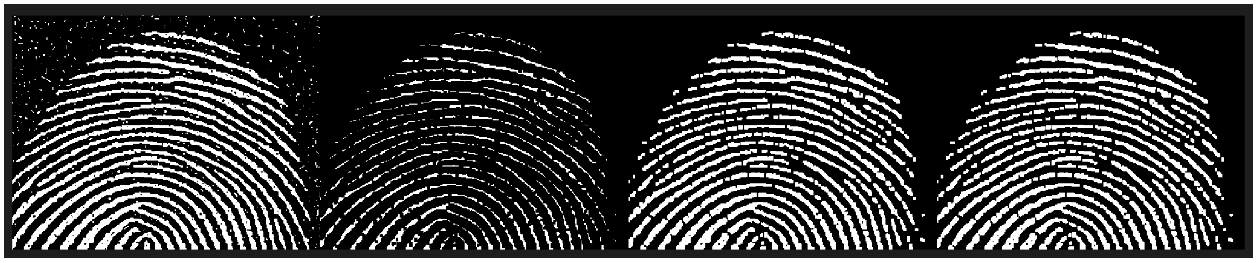
When I dilated the eroded image, the main ridges grew back to normal thickness, but the small noise did not come back. So fed looks cleaner than the original: the fingerprint structure is kept, but the random white specks are gone.

5. **Open *f* to produce *fo*.**

```
fo= imopen (f, SE);
montage({f, fe, fed, fo}, “Size”, [1 4])
```



6. **Show *f*, *fe*, *fed* and *fo* as a 4 image montage.**



With the help of the collection of functions at the beginning of the page and putting all four comparisons together, the montage shows all four stages side-by-side:

- **f (original):** fingerprint ridges with lots of small white noise dots.
- **fe (eroded):** ridges become thinner and most of the small noise disappears.
- **fed (eroded then dilated):** the ridges grow back to normal thickness but the noise does not return, giving a cleaner image.
- **fo (opened with imopen):** very similar to `fed`, with smooth ridges and reduced noise. Using `imopen` produces the same overall effect more consistently.

The progression clearly shows how opening removes noise while preserving the main

Task 2.2: Explore what happens with other sizes and shapes of the structuring element.

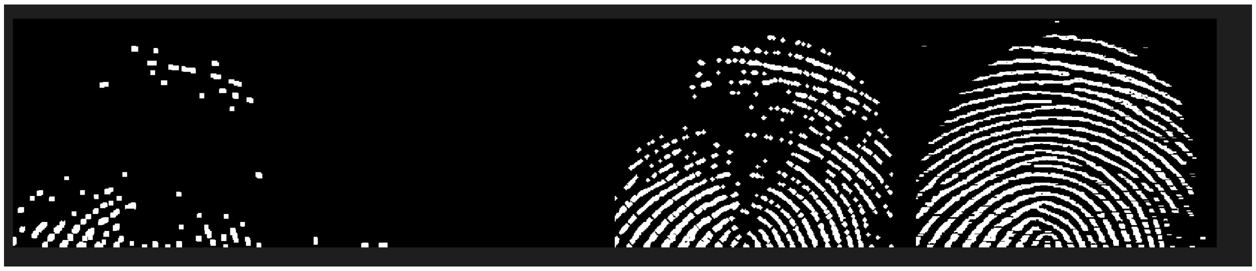
To understand how opening depends on the structuring element, we tried playing around with different sizes and shapes:

Code:

```
SE5    = strel('square',5);
SE7    = strel('square',7);
SEdisk = strel('disk',2);
SEline = strel('line',5,0);

fo5     = imopen(f, SE5);
fo7     = imopen(f, SE7);
fodisk = imopen(f, SEdisk);
```

```
foline = imopen(f, SEline);

montage({fo5, fo7, fodisk, foline}, "Size", [1 4])
```

Output:



Observation:
From this output, we attempted to explore with varied shapes (square, disk, line) and sizes (5,7). Here, in the output different structuring elements cleaned the fingerprint differently. Square SEs remove lots of noise but can erase thin ridges (leftmost & the next)(especially SE7). Disk SE gives the smoothest and most natural fingerprint. Line SE removes noise in one direction but damages ridges in other directions.

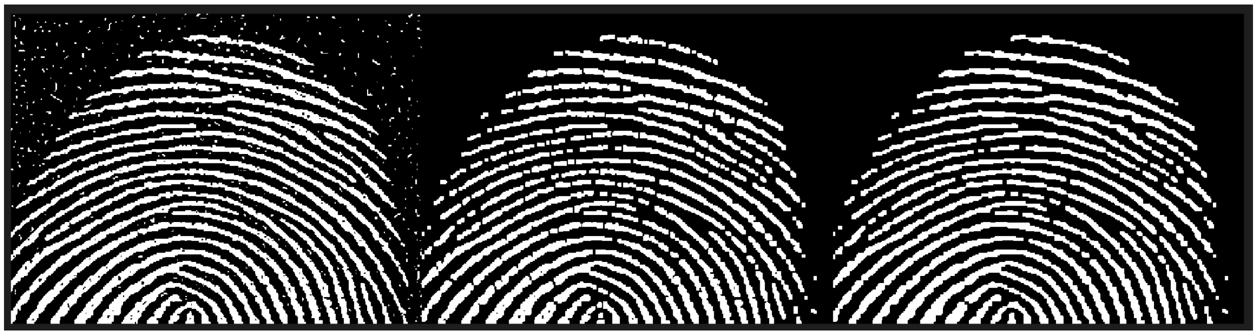
Task 2.3: Improve the image *fo* with a close operation.

Closing = Dilation → then Erosion

Code:

```
fc = imclose(fo, SE);
montage({f, fo, fc}, "Size", [1 3])
```

Output:



Observation:

Closing = **dilation, then erosion**.

It is used to:

- **fill small gaps** in the fingerprint lines
- **connect broken ridges**
- **make the lines look smoother**

So, after closing, the fingerprint looks **cleaner and more continuous**, without bringing the noise back. After applying `imclose` to `fo`, the fingerprint looks smoother and more connected.

Task 2.4: Compare morphological filtering using Open + Close to the spatial filter with a Gaussian filter.

In our understanding in simple language, opening + closing removes noise while keeping the fingerprint ridges sharp and connected. While Gaussian filtering smooths the whole image and makes the ridges blurry, it reduces important details.

Code:

```
clear all
close all
```

```
% Read the image
f = imread('assets/finger-noisy.tif');
```

```
% If f is already binary (logical), keep it as-is; otherwise
binarize
if islogical(f)
fb = f;
else
fb = imbinarize(f);
end
```

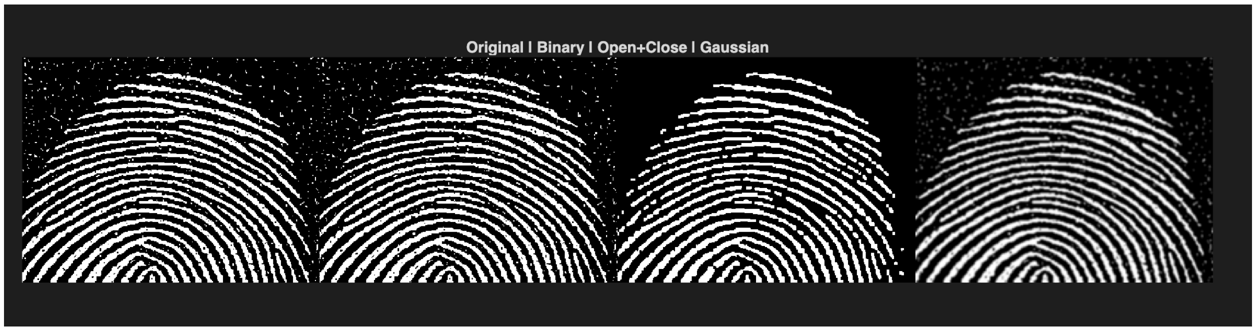
```
% Structuring element
SE = strel('square',3);
```

```
% Morphological filtering: Open then Close
fo = imopen(fb, SE);
fc = imclose(fo, SE);
```

```
% Gaussian spatial filter (needs numeric grayscale)
% If the image is binary, convert to uint8 first so Gaussian
makes sense
fg = imgaussfilt(uint8(f) * 255, 1);   % sigma = 1
```

```
% Show all results
figure
montage({f, fb, fc, fg}, "Size", [1 4])
title('Original | Binary | Open+Close | Gaussian')
```

Output:



Observation:

Morphological Open + Close removes noise while keeping the fingerprint ridges sharp and connected. Gaussian filtering smooths the image by blurring everything, which reduces detail in the ridges. Morphology clearly preserves structure better, while Gaussian blur makes the fingerprint fuzzier.

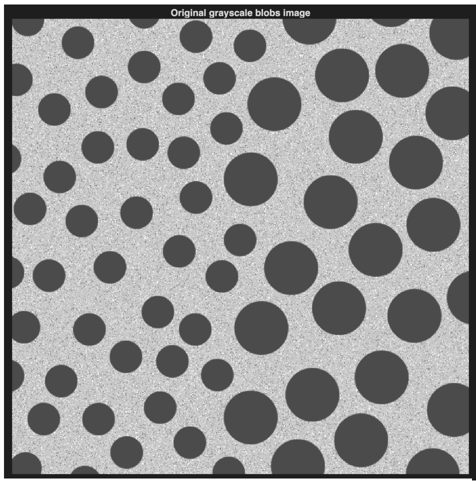
Task 3 - Boundary detection

Task 3.1: First, we turn this "inverted" grayscale image into a binary image with white objects (blobs) and a black background.

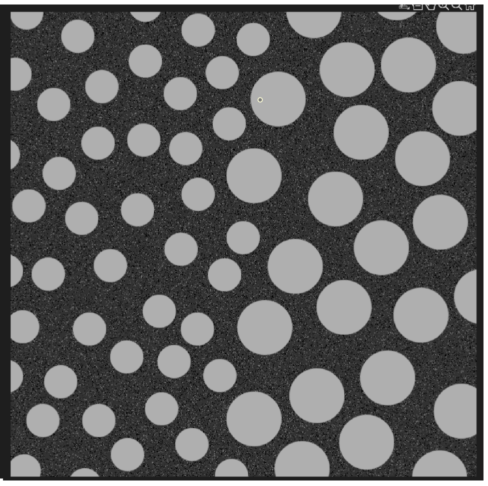
Code:

```
I_inv = imcomplement(I);
figure;
imshow(I_inv);
title('Inverted image (blobs now bright)');
```

Output:



Original Image

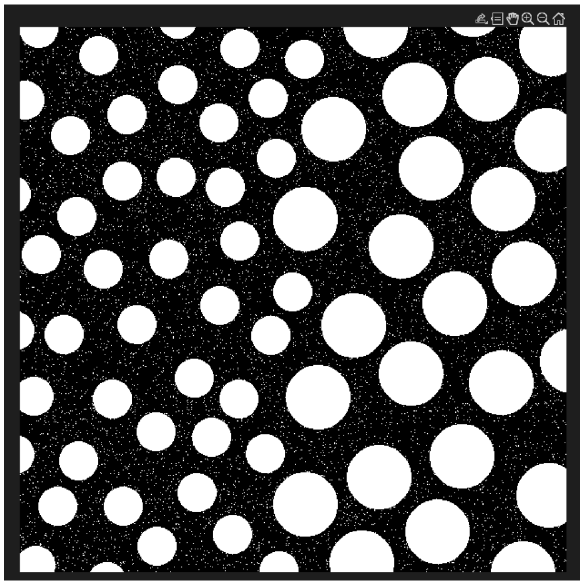

Inverted blobs now bright

Task 3.2: Convert the inverted grayscale image to binary

Code:

```
clear all
close all
I = imread('assets/blobs.tif');
I = imcomplement(I);
level = graythresh(I);
BW = imbinarize(I, level);
```

Output:



Observation:

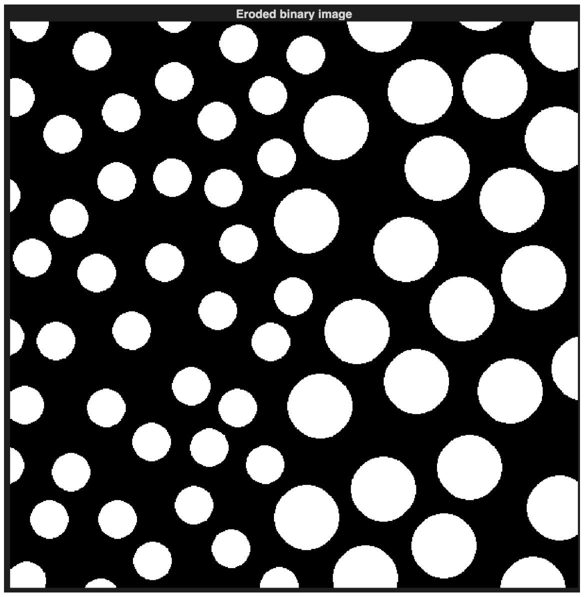
After inverting the grayscale image, the dark blobs become light/white and the background becomes darker. Using Otsu thresholding (`graythresh`) and `imbinarize` converts it into a binary image where blobs are white, and the background is black. A few small white specks remain due to noise in the original image.

Task 3.3(a): Erode BW

Code:

```
SE = ones(3,3);          % 3×3 structuring element
BW_e = imerode(BW, SE);    % eroded binary image
figure;
imshow(BW_e);
title('Eroded binary image');
```

Output:



Observation:

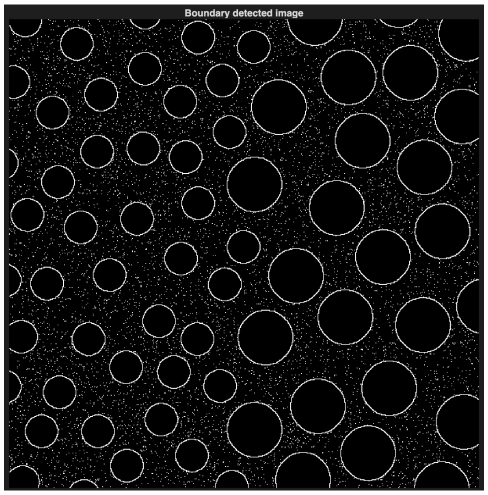
Here, the blobs become slightly smaller with thin arms/edges shrinking inward and tiny specks of noise disappear entirely

Task 3.3(b): Compute the boundary image

Code:

```
Boundary = BW - BW_e;
figure;
imshow(Boundary);
title('Boundary detected image');
```

Output:



Observation:

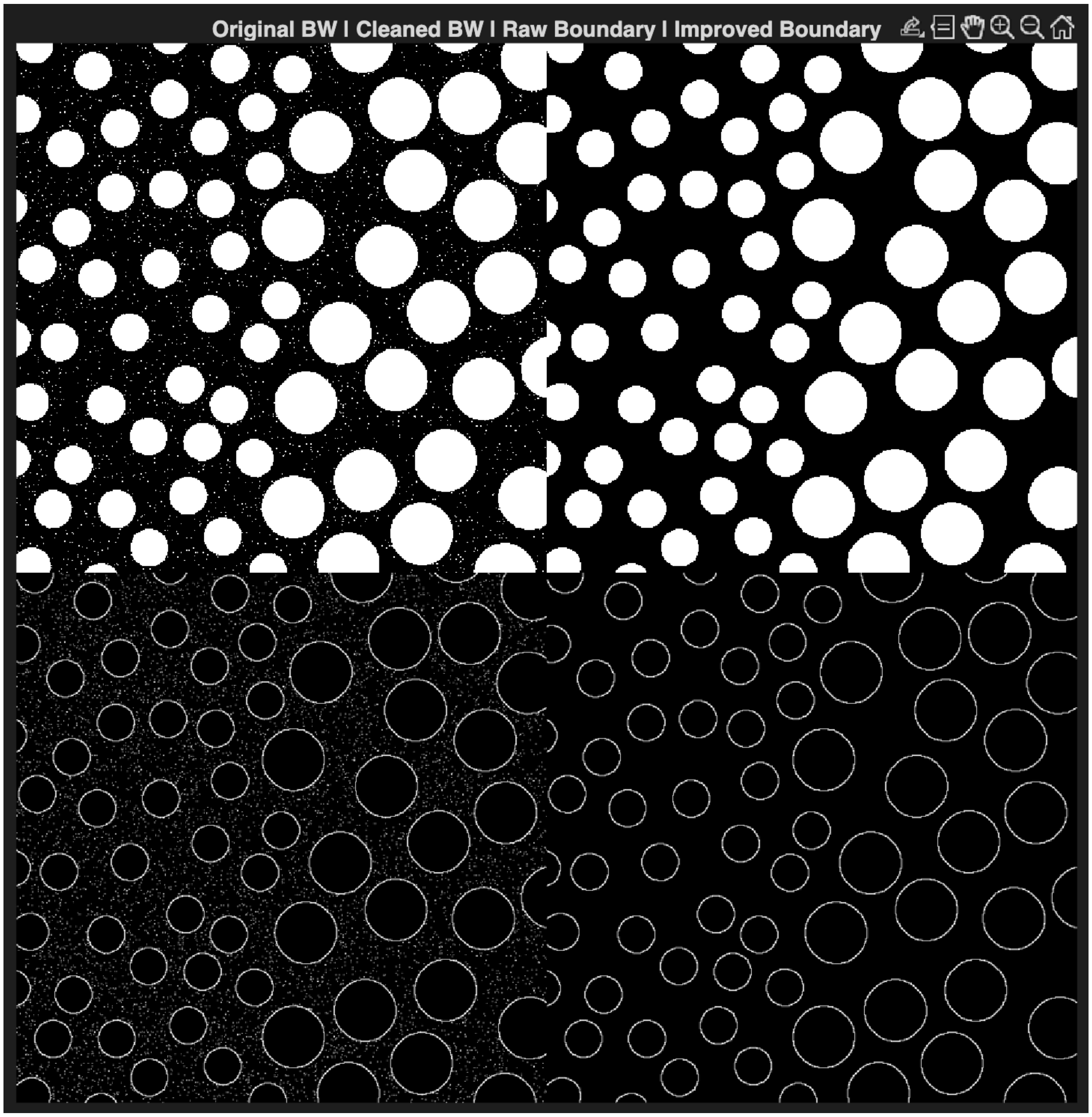
Thin white outlines around each blob while the interior disappears. Additionally, the noise is mostly gone, and the boundaries of blobs stand out clearly.

Task 3.4: "How can we improve this?" ~ By using Opening

Code:

```
BW_clean = imopen(BW, SE);    % clean binary mask
% Now do boundary on the cleaned image
BW_clean_e = imerode(BW_clean, SE);
Boundary_clean = BW_clean - BW_clean_e;

montage({BW, BW_clean, Boundary, Boundary_clean}, "Size", [2
2])
title('Original BW | Cleaned BW | Raw Boundary | Improved Bou
ndary');
```

Output:



Observations:

To improve the boundary result, I applied an opening operation to the binary image before extracting the boundaries. Opening removes small noise and smooths the blob shapes. The improved boundary image has cleaner, smoother outlines and fewer unwanted edges.

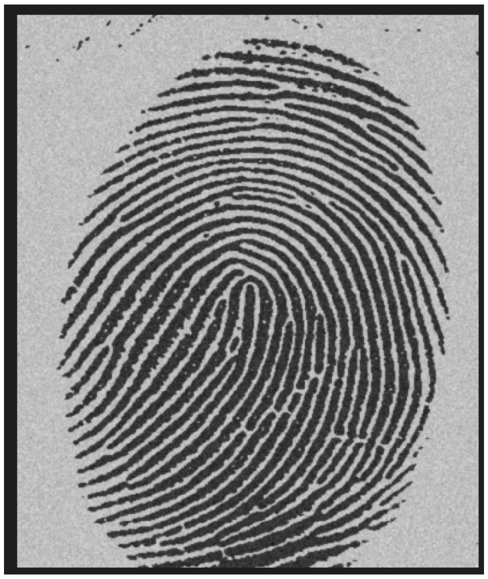
Task 4 - Function bwmorph - Thinning and Thickening

Step 1: Read the image file 'fingerprint.tif' into *f*.

Code:

```
f = imread('assets/fingerprint.tif');
imshow(f);
```

Output:



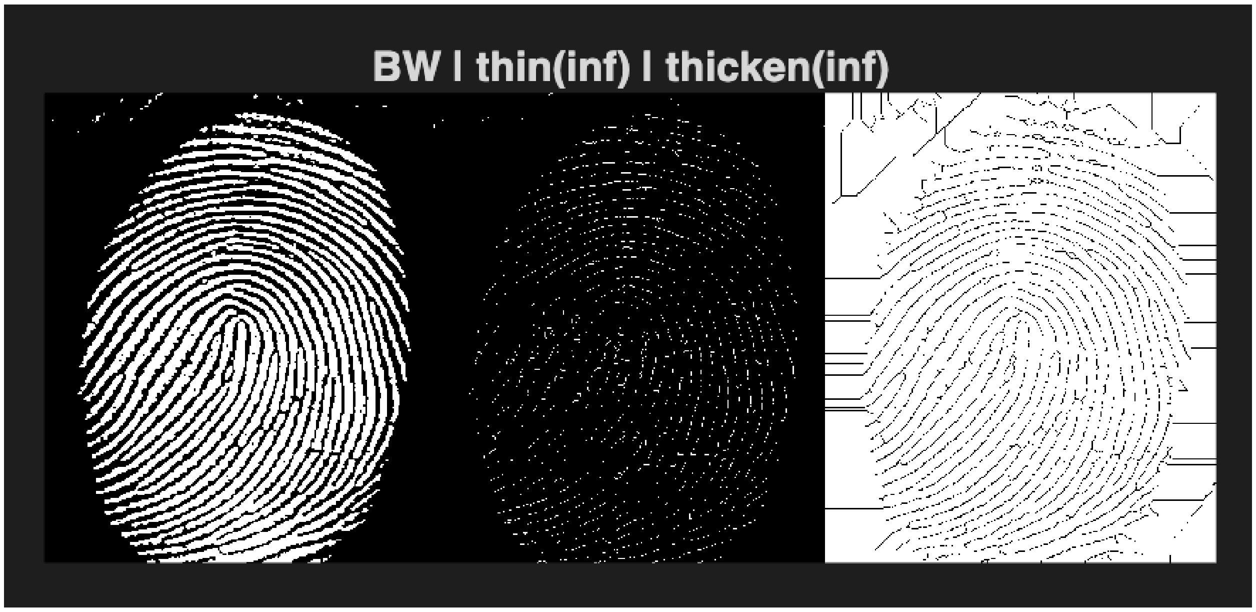
Step 2: Turn this into a good binary image using the method from the previous task > perform thinning operations and montage everything together:

Code:

```
clear all
close all
clc

% 1) Read image
f = imread('assets/fingerprint.tif');

% 2) Convert to a good binary image (Otsu threshold)
```

```
if ~islogical(f)
    level = graythresh(f);
    BW = imbinarize(f, level);
else
    BW = f;
end

% If background is mostly white, invert so ridges become whit
e
if mean(BW(:)) > 0.5
    BW = ~BW;
end

% 3) Thinning 1..5 times + until stable (inf)
g1 = bwmorph(BW, 'thin', 1);
g2 = bwmorph(BW, 'thin', 2);
g3 = bwmorph(BW, 'thin', 3);
g4 = bwmorph(BW, 'thin', 4);
g5 = bwmorph(BW, 'thin', 5);
ginf = bwmorph(BW, 'thin', inf);

% 4) Montage: original + thinned
figure
montage({BW, g1, g2, g3, g4, g5, ginf}, "Size", [1 7])
title('BW | thin(1) | thin(2) | thin(3) | thin(4) | thin(5) |
thin(inf)')

% 5) Display fingerprint as black lines on white background
BW_blackOnWhite = ~BW;
gInf_blackOnWhite = ~ginf;

figure
montage({BW_blackOnWhite, gInf_blackOnWhite}, "Size", [1 2])
title('Black-on-white: BW vs thin(inf)')

% 6) Show relationship between thinning and thickening
% Thickening white ridges is equivalent to thinning the inver
ted image
thick_inf = bwmorph(BW, 'thicken', inf);

figure
montage({BW, ginf, thick_inf}, "Size", [1 3])
title('BW | thin(inf) | thicken(inf)')
```

Output:



Observation:

On thinning the fingerprint image multiple (five) times, we noticed that the fingerprint ridges gradually became thinner until they turned into very fine lines. Using `n = inf` keeps thinning the image until it cannot change anymore, resulting in a skeleton-like version of the fingerprint. On the other hand, thickening does the opposite — it makes the ridges grow and eventually merge together. This helped me understand that thinning and thickening are opposite operations that affect the structure in reverse ways.

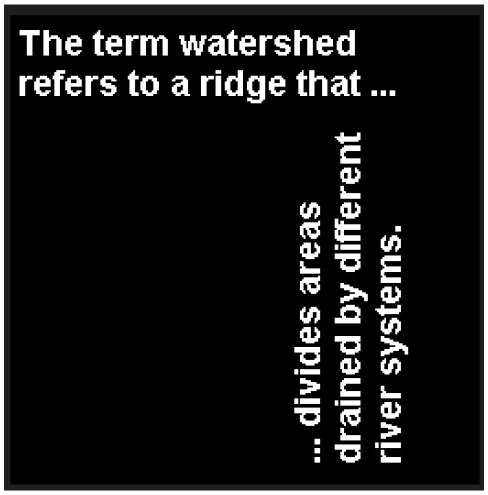
Task 5 - Connected Components and labels

Step 1: Read + view the image

Code:

```
t = imread('assets/text.png');
imshow(t)
CC = bwconncomp(t)
```

Output:



Observation:

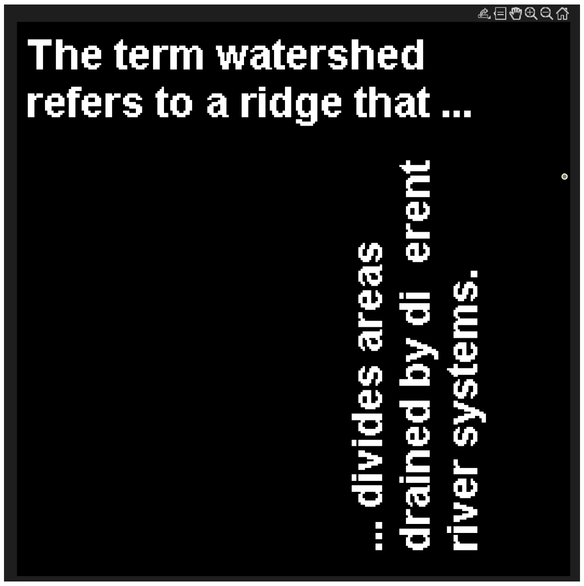
After loading the image using `imread`, I observed a binary text image with white characters on a black background. The text appears in two regions: a horizontal sentence at the top and a vertically oriented sentence on the right. Spaces in the top sentence separate the letters but appear connected in the vertical text block.

Step 2: Connecting Compponents

Code:

```
numPixels = cellfun(@numel, CC.PixelIdxList);
[biggest, idx] = max(numPixels);
t(CC.PixelIdxList{idx}) = 0;
figure
imshow(t)
```

Output:



Observation:

After computing connected components, I measured the size (pixel count) of each component and found the largest one. Setting its pixels to 0 removed that entire component from the image. In my case, the largest component corresponded to the large connected text block, which disappeared while the remaining characters stayed

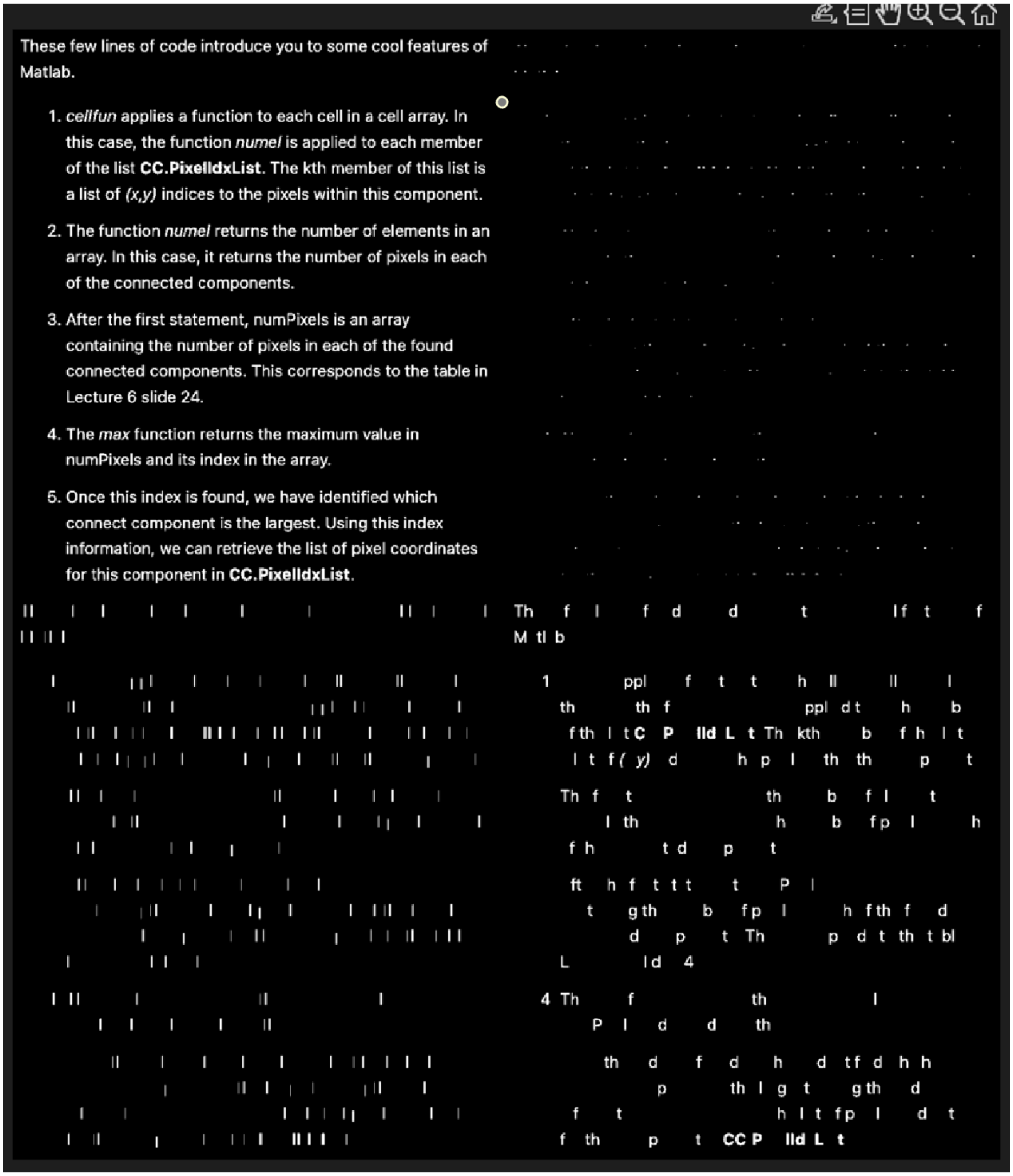
Task 6 - Morphological Reconstruction

Task 6.1: Comparison of Opening and Morphological Reconstruction

Code:

```
clear all
close all
f = imread('assets/text_bw.tif');
se = ones(17,1);
g = imerode(f, se);
fo = imopen(f, se);     % perform open to compare
fr = imreconstruct(g, f);
montage({f, g, fo, fr}, "size", [2 2])
```

Output:



These few lines of code introduce you to some cool features of Matlab.

1. *cellfun* applies a function to each cell in a cell array. In this case, the function *numel* is applied to each member of the list **CC.PixelIdxList**. The kth member of this list is a list of *(x,y)* indices to the pixels within this component.
2. The function *numel* returns the number of elements in an array. In this case, it returns the number of pixels in each of the connected components.
3. After the first statement, numPixels is an array containing the number of pixels in each of the found connected components. This corresponds to the table in Lecture 6 slide 24.
4. The *max* function returns the maximum value in numPixels and its index in the array.
5. Once this index is found, we have identified which connect component is the largest. Using this index information, we can retrieve the list of pixel coordinates for this component in **CC.PixelIdxList**.

Observation:

In the montage, f is the original binary text image. The image g shows the result after erosion with a vertical 17×1 structuring element, where only letters with long vertical strokes remain.

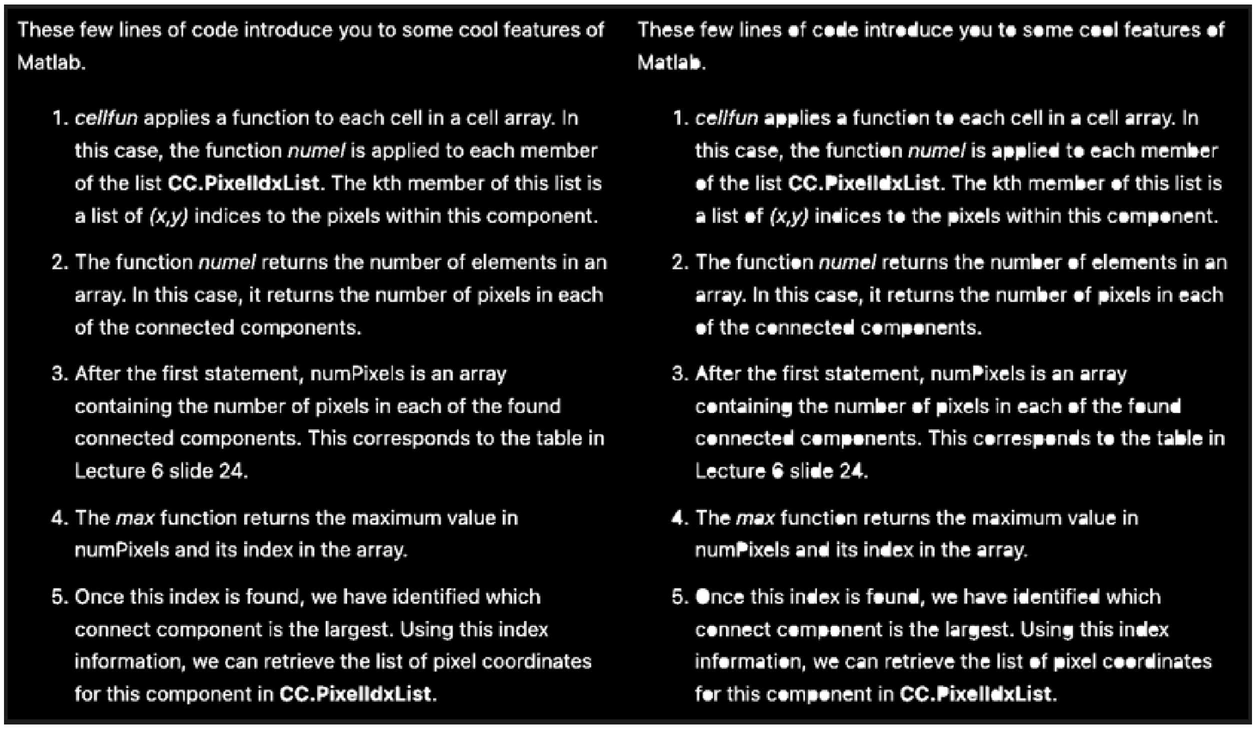
The image fo (normal opening) tries to restore the shapes but some letters appear distorted or incomplete. The image fr (morphological reconstruction) restores the remaining letters to their original shapes while removing all others. This shows that reconstruction preserves shapes much better than standard opening.

Task 6.2: imfill

Code:

```
ff = imfill(f);
figure
montage({f, ff})
```

Output:



These few lines of code introduce you to some cool features of Matlab.

1. *cellfun* applies a function to each cell in a cell array. In this case, the function *numel* is applied to each member of the list **CC.PixelIdxList**. The kth member of this list is a list of *(x,y)* indices to the pixels within this component.
2. The function *numel* returns the number of elements in an array. In this case, it returns the number of pixels in each of the connected components.
3. After the first statement, numPixels is an array containing the number of pixels in each of the found connected components. This corresponds to the table in Lecture 6 slide 24.
4. The *max* function returns the maximum value in numPixels and its index in the array.
5. Once this index is found, we have identified which connect component is the largest. Using this index information, we can retrieve the list of pixel coordinates for this component in **CC.PixelIdxList**.

These few lines of code introduce you to some cool features of Matlab.

1. *cellfun* applies a function to each cell in a cell array. In this case, the function *numel* is applied to each member of the list **CC.PixelIdxList**. The kth member of this list is a list of *(x,y)* indices to the pixels within this component.
2. The function *numel* returns the number of elements in an array. In this case, it returns the number of pixels in each of the connected components.
3. After the first statement, numPixels is an array containing the number of pixels in each of the found connected components. This corresponds to the table in Lecture 6 slide 24.
4. The *max* function returns the maximum value in numPixels and its index in the array.
5. Once this index is found, we have identified which connect component is the largest. Using this index information, we can retrieve the list of pixel coordinates for this component in **CC.PixelIdxList**.

Observation:

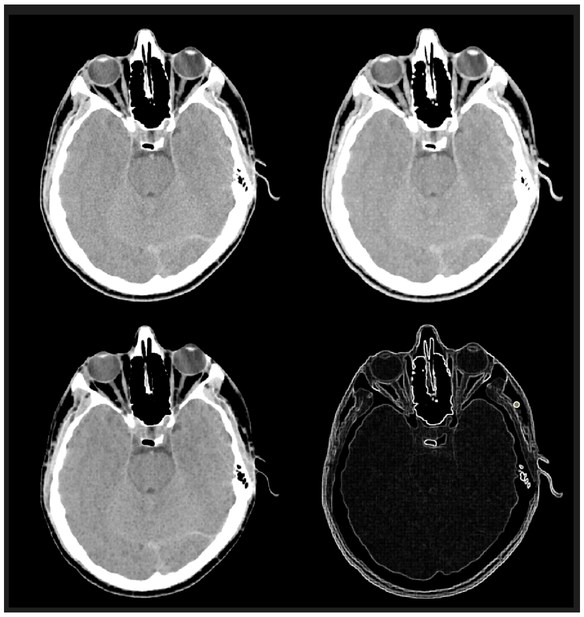
The function `imfill` fills black regions that are completely surrounded by white pixels. In the result image, holes inside letters such as “o” and “p” are filled, making the characters appear solid. This demonstrates how morphological filling can modify internal structures of binary objects.

Task 7 - Morphological Operations on Grayscale images

Code:

```
clear all;
close all;
f = imread('assets/headCT.tif');
se = strel('square',3);
gd = imdilate(f, se);
ge = imerode(f, se);
gg = gd - ge;
montage({f, gd, ge, gg}, 'size', [2 2])
```

Output:



Observation:

In this task, dilation made the bright areas in the CT image slightly thicker, while erosion made them thinner. When I subtracted the eroded image from the dilated image, the edges became clearly visible. This shows that the morphological gradient highlights boundaries and can be used for edge detection in grayscale images.