

Lab 4 Log Book

Task 1: Dilation and Erosion

Matlab provides a collection of morphological functions:

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

```
A = imread('assets/text-broken.tif'); % Read a damaged text image
B1 = [0 1 0;
      1 1 1;
      0 1 0]; % A 3×3 structuring element has been defined, with the central
%pixel and its top, bottom, left, and right pixels set to 1.
A1 = imdilate(A, B1);
montage({A,A1})
```

%imdilate(A, B1) performs an dilation operation on A, expanding the white (foreground) area and filling in small gaps. montage({A, A1}) displays the original image A and the dilated image A1 side by side, making it convenient to observe the effect.

Output: The expansion operation extends the white foreground area, making character connections closer together, reducing breakpoints, and making the text easier to recognize.

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.

```
B2 = ones(3,3); % generate a 3x3 matrix of 1's
A2 = imdilate(A, B2);
montage({A,A2})
```

Output: Although not very obvious, compared to B1 (cross-shaped structural element), B2 (3x3 all-1 structural element) extends more and more evenly. The character feels thicker.

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.

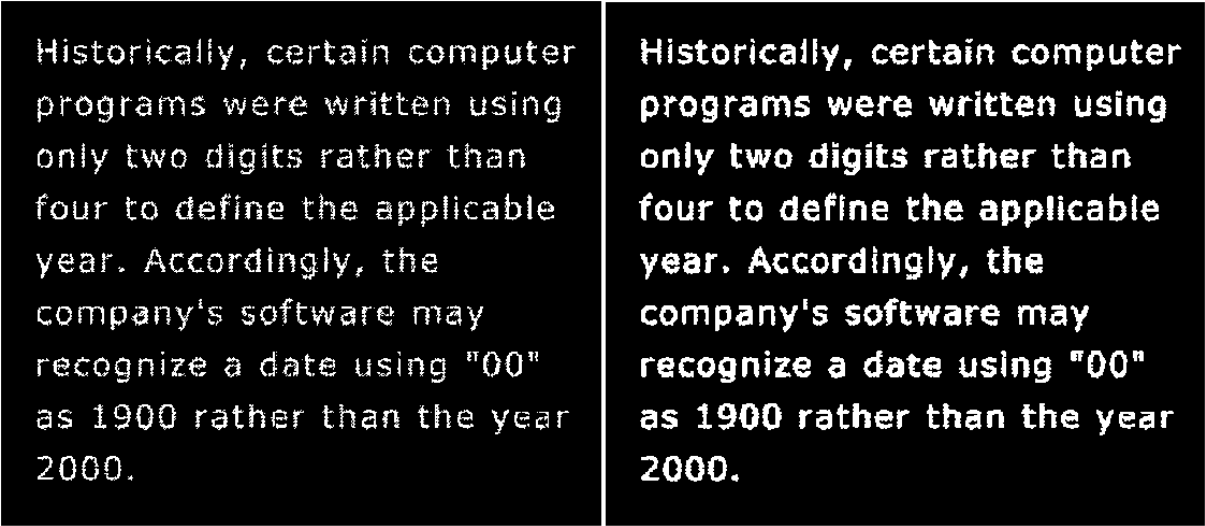
Historically, certain computer programs were written using only two digits rather than four to define the applicable year. Accordingly, the company's software may recognize a date using "00" as 1900 rather than the year 2000.

```

Bx = [1 0 1;
      0 1 0;
      1 0 1]; %Generate a 3x3 structural element with diagonal elements and the center
Ax = imdilate(A, Bx);
montage({A,Ax})

```

Output: Bx (diagonal structural element), causing the dilation to expand only in the diagonal direction, making the character smoother but not significantly thicker. The text characters on the right are smoother compared to those on the left, but the inflation effect is not as strong as before.



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.

```

A1 = imdilate(A, B1); % first dilation
A2 = imdilate(A1, B1); % second dilation
A3 = imdilate(A2, B1); % third dilation
montage({A, A1, A2, A3});

```

Output: Three expansions were conducted, with each expansion having a higher degree than the previous one.

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.

Generation of structuring element

```
SE = strel("diamond",r)
SE = strel("disk",r)
SE = strel("disk",r,n)
SE = strel("octagon",r)
SE = strel("line",len,deg)
SE = strel("rectangle",[m n])
SE = strel("square",w)
```

Structuring Element	Code Example	Description
Square	<code>strel('square', w)</code>	A <code>w×w</code> square structuring element, expands uniformly.
Rectangle	<code>strel('rectangle', [m n])</code>	A <code>m×n</code> rectangular element, suitable for non-uniform dilation.
Diamond	<code>strel('diamond', r)</code>	A diamond-shaped element with radius <code>r</code> , suitable for diagonal expansion.
Disk	<code>strel('disk', r)</code>	A disk-like structuring element, providing smoother results.
Octagon	<code>strel('octagon', r)</code>	Similar to a disk but with more regular edges.
Line	<code>strel('line', len, deg)</code>	A line with length <code>len</code> and angle <code>deg</code> .
Custom (Cross Shape, etc.)	<code>strel('arbitrary', [0 1 0; 1 1 1; 0 1 0])</code>	A user-defined structuring element, such as a <code>+</code> shape.

```
SE = strel('disk',4); % creates a disk-shaped structuring element with a radius
SE.Neighborhood      % print the SE neighborhood contents
```

Output: The matrix displayed is a 7×7 binary matrix,

- The 1s represent the pixels that are part of the **disk-shaped structuring element**.
- The 0s represent pixels **outside the disk**.

```
ans =  
  
7×7 logical 数组  
  
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
```

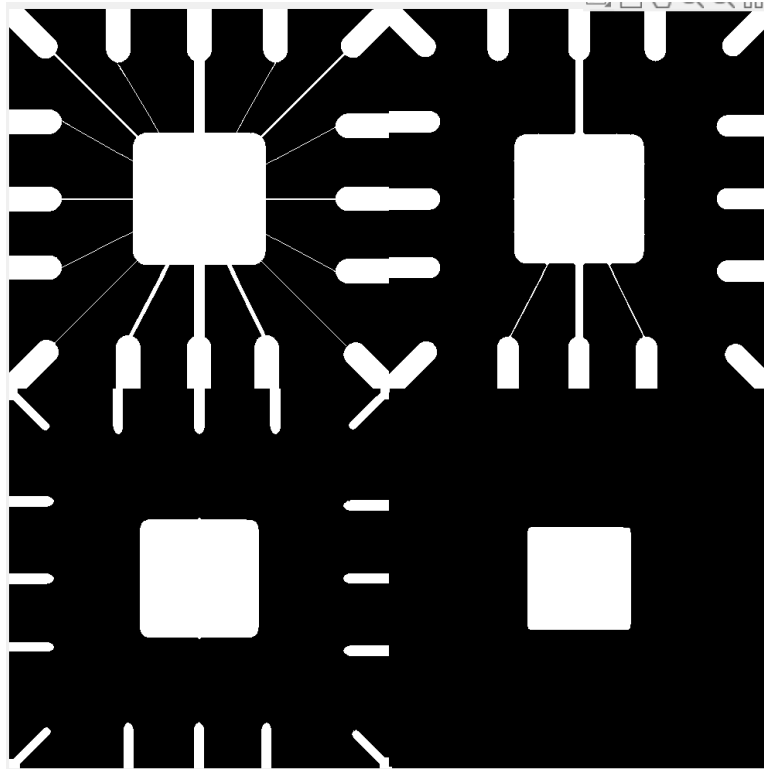
```
clear all  
close all  
A = imread('assets/wirebond-mask.tif');  
SE2 = strel('disk',2); % Disk structure element with a radius of 2  
SE10 = strel('disk',10); %Disk structure element with a radius of 10  
SE20 = strel('disk',20); %Disk structure element with a radius of 20  
E2 = imerode(A,SE2); % Using a structural element with a radius of 2 for erosion  
E10 = imerode(A,SE10); % Using a structural element with a radius of 10 for erosion  
E20 = imerode(A,SE20); %Using a structural element with a radius of 20 for erosion  
montage({A, E2, E10, E20}, "size", [2 2])
```

Output: Erosion remove foreground pixels (white parts), retaining only the pixels completely covered by the structuring element,

Small structuring element (disk, 2) → The white area is slightly contracted.

Large structuring element (disk, 10) → The white area is contracted more, and finer connections may be removed.

Larger structuring element (disk, 20) → Fine lines are completely removed, leaving only larger white blocks.



Task 2 - Morphological Filtering with Open and Close

Opening = Erosion + Dilation

```
f = imread('assets/fingerprint-noisy.tif'); %Read the fingerprint image f with n
SE = strel('square', 3) ; %Create a 3x3 square structural element for morphologic
fe = imerode(f, SE); %Erode f to reduce noise and obtain fe.
fed = imdilate(fe, SE); %Expand "fe", restore some structure, and obtain "fed."
fo = imopen(f, SE); %Opening operation (Opening = Erosion followed by Dilation)
montage({f, fe, fed, fo}, 'Size', [2 2]);
```

Output: It can be seen that after the `imerode` operation, the fine structures (noise) have become smaller, but the overall size has shrunk, and breakpoints have appeared in the fingerprint part. After the `fed` operation, the structure has recovered, but some breakpoints still appear. The `fo` opening operation first removes noise and then performs dilation, with an effect similar to `fe+fed`.



```
fo_closed = imclose(fo, SE); % Apply closing operation to restore structures
montage({f, fo, fo_closed}, 'Size', [1 3]);
```

Output: After the close+open processing, the final fingerprint image presents a better effect, it fills in some broken points, and repairs the image.



```
fo_closed = imclose(fo, SE); % Apply closing operation to restore structures
f = im2double(f); %convert f to double
f_gaussian = imgaussfilt(f, 2); % Apply Gaussian filter with sigma = 2
montage({f, fo_closed, f_gaussian}, 'Size', [1 3]);
```

Output: Compared to the opening and closing operations, Gaussian blur does not preserve obvious edges, and the image details are also lost. Fine noise can still be seen in the image. Since the edges of my image are very distinct after binarization, Gaussian blur is not as effective as morphological filtering.



Task 3 - Boundary detection

```
clear all
close all
I = imread('assets/blobs.tif');
I = imcomplement(I); % Invert the colors, making the bubbles white and the background black
level = graythresh(I);
BW = imbinarize(I, level); % Calculate the global Otsu threshold and perform binary thresholding

SE = strel('square', 3); % Create a 3x3 square structural element for morphological erosion
Es = imerode(BW, SE);
f_operated = BW - Es; % Calculate boundary (Original image - Eroded image)
montage({I, BW, Es, f_operated}, 'size', [2 2]);
```

The Matlab function `graythresh` computes a global threshold *level* from the grayscale image *I*, by finding a threshold that minimizes the variance of the thresholded black and white pixels. The function `imbinarize` turns the grayscale image to a binary image **BW**: those pixels above or equal to *level* are made foreground (i.e. 1), otherwise they are background (0).

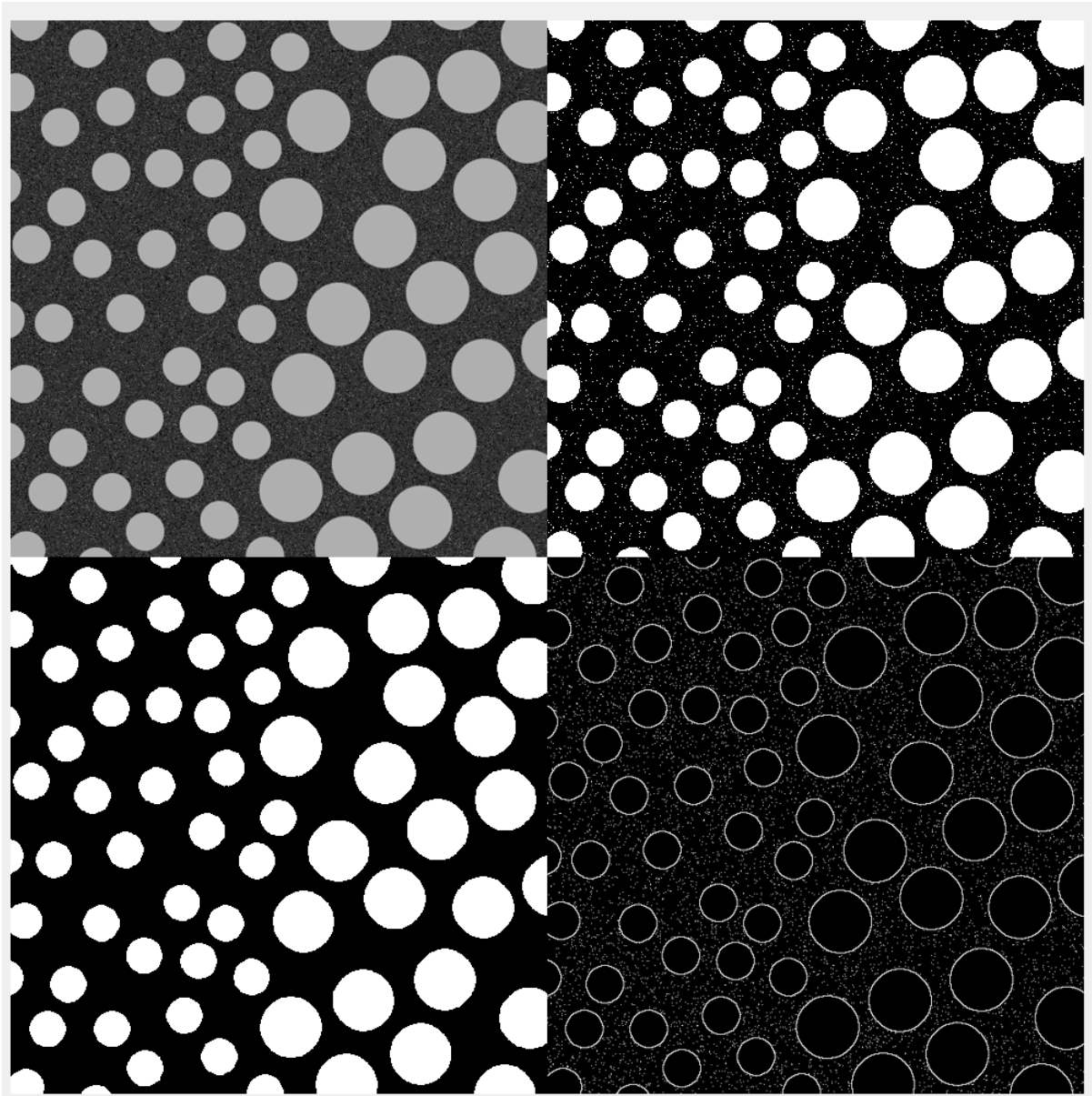
Output:

Original Image (I): The bubbles are white, the background is black, and there is noise.

Binary Image (BW): Clearly separates the bubbles from the background, but still retains some noise.

Eroded Image (Es): The bubble boundaries are weakened, making it easier to extract the edges.

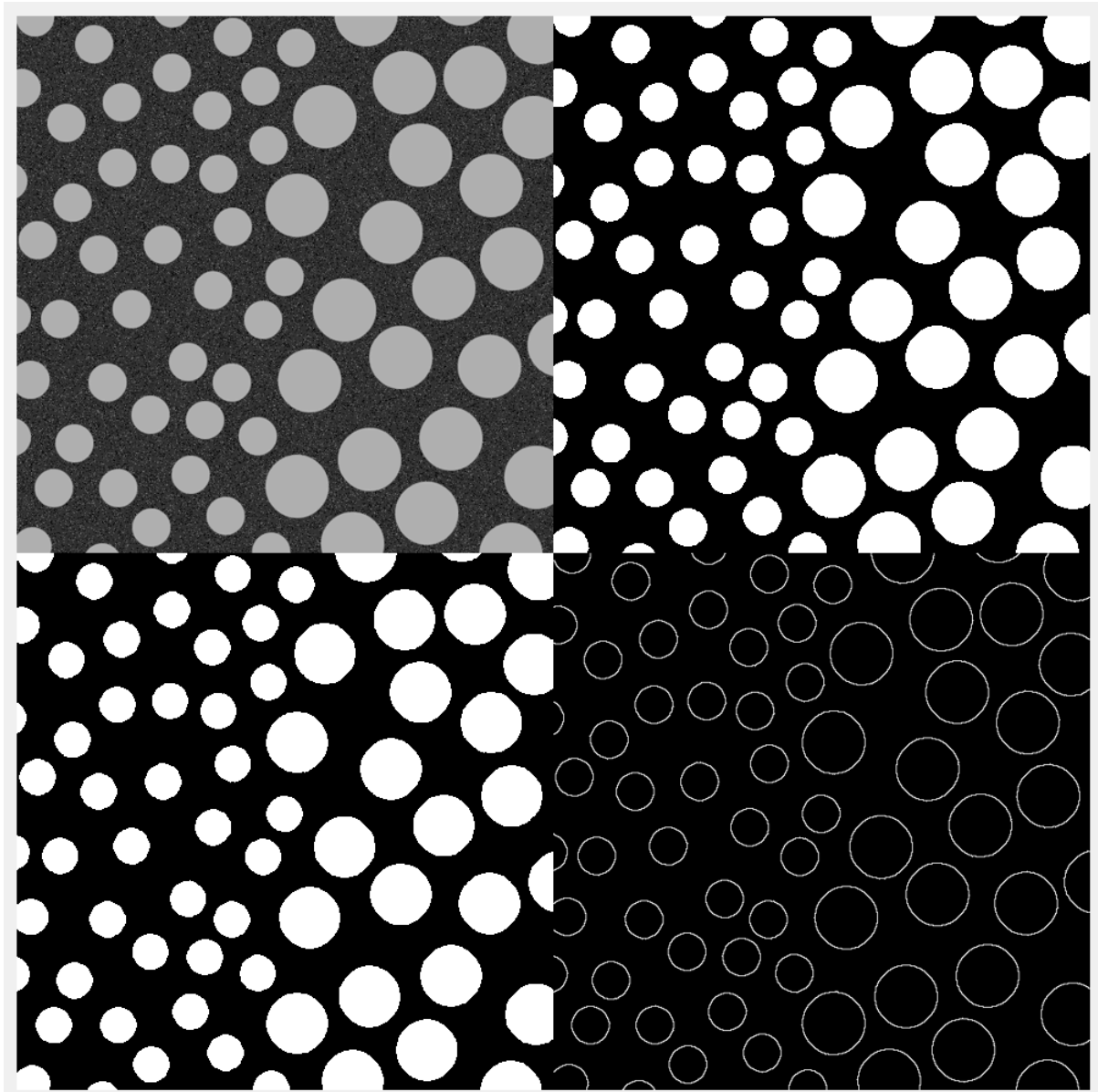
Boundary Detection (f_operated): Successfully extracted the bubble boundaries! The edge contours are clear and meet expectations.



Further improvement:

```
SE = strel('disk', 2);
BW_clean = imopen(BW, SE);
SE = strel('square', 3);
Es = imerode(BW_clean, SE);
f_operated = BW_clean - Es;
montage({I, BW_clean, Es, f_operated}, 'Size', [2 2]);
```

Output: The current issue is that there are still many noise points, so I first used morphological opening operation to remove small noise points, first removing the noise, so that the final result is clearer and more visible.



Task 4 - Function `bwmorph` - thinning and thickening

Matlab's Image Processing Toolbox includes a general morphological function `bwmorph` which implements a variety of morphological operations based on combinations of dilations and erosions. The calling syntax is:

```
g = bwmorph(f, operations, n)
```

where f is the input binary image, *operation* is a string specifying the desired operation, and n is a positive integer specifying the number of times the operation should be repeated. ($n = 1$ if omitted.)

The morphological operations supported by `bwmorph` are:

Operation	Description
bothat	"Bottom-hat" operation using a 3×3 structuring element; use <code>imbothat</code> (see Section 10.6.2) for other structuring elements.
bridge	Connect pixels separated by single-pixel gaps.
clean	Remove isolated foreground pixels.
close	Closing using a 3×3 structuring element of 1s; use <code>imclose</code> for other structuring elements.
diag	Fill in around diagonally-connected foreground pixels.
dilate	Dilation using a 3×3 structuring element of 1s; use <code>imdilate</code> for other structuring elements.
erode	Erosion using a 3×3 structuring element of 1s; use <code>imerode</code> for other structuring elements.
fill	Fill in single-pixel "holes" (background pixels surrounded by foreground pixels); use <code>imfill</code> (see Section 11.1.2) to fill in larger holes.
hbreak	Remove H-connected foreground pixels.
majority	Make pixel p a foreground pixel if at least five pixels in $N_4(p)$ (see Section 10.4) are foreground pixels; otherwise make p a background pixel.
open	Opening using a 3×3 structuring element of 1s; use function <code>imopen</code> for other structuring elements.
remove	Remove "interior" pixels (foreground pixels that have no background neighbors).
shrink	Shrink objects with no holes to points; shrink objects with holes to rings.
skel	Skeletonize an image.
spur	Remove spur pixels.
thicken	Thicken objects without joining disconnected 1s.
thin	Thin objects without holes to minimally-connected strokes; thin objects with holes to rings.
tophat	"Top-hat" operation using a 3×3 structuring element of 1s; use <code>imtophat</code> (see Section 10.6.2) for other structuring elements.

```

clear all
close all
I = imread('assets/fingerprint.tif');
I = imcomplement(I); % Flip, making the fingerprint white and the background
% Calculate Otsu threshold & perform binarization
level = graythresh(I);
BW = imbinarize(I, level);
% Perform Thinning operation
g = bwmorph(BW, "thin", 1);
g1 = bwmorph(BW, "thin", 2);
g2 = bwmorph(BW, "thin", 3);
g3 = bwmorph(BW, "thin", 4);
g4 = bwmorph(BW, "thin", 5);
g_i = bwmorph(BW, "thin", inf);
% Reverse all results, making the fingerprint black and the background white.
BW = imcomplement(BW);
g = imcomplement(g);
g1 = imcomplement(g1);
g2 = imcomplement(g2);

```

```
g3 = imcomplement(g3);  
g4 = imcomplement(g4);  
g_i = imcomplement(g_i);  
  
montage({BW, g, g1, g2, g3, g4, g_i}, 'Size', [2 4]);
```

Output: By using the "thin" operation, the fingerprint gradually becomes thinner, and the noise is also gradually eliminated. However, after more than three operations, the fingerprint appears with obvious breakpoints, and the inf converges to the final image, making the outline invisible, with only some black dots on the image.



Task 5 - Connected Components and labels

CC is a data structure returned by *bwconncomp* as described below.

Connected components, returned as a structure with four fields.

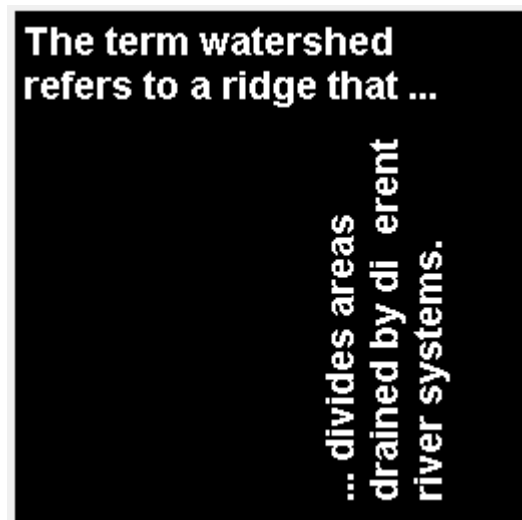
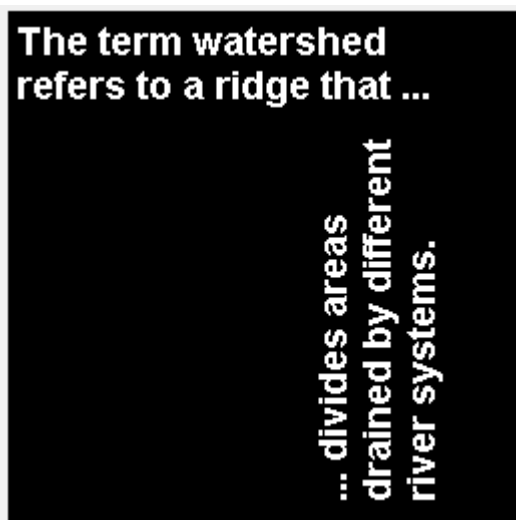
Field	Description
Connectivity	Connectivity of the connected components (objects)
ImageSize	Size of BW
NumObjects	Number of connected components (objects) in BW
PixelIdxList	1-by-NumObjects cell array where the k -th element in the cell array is a vector containing the linear indices of the pixels in the k -th connected component.

```
t = imread('assets/text.png');
imshow(t)
CC = bwconncomp(t)
numPixels = cellfun(@numel, CC.PixelIdxList);
[biggest, idx] = max(numPixels);
t(CC.PixelIdxList{idx}) = 0;
figure
imshow(t)
```

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

1. `cellfun` applies a function to each element in an array. In this case, the function `numel` is applied to each member of the list `CC.PixelIdxList`. The k th member of this list is itself a list of (x,y) indices to the pixels within this component.
2. The function `numel` returns the number of elements in an array or list. In this case, it returns the number of pixels in each of the connected components.
3. The first statement returns `numPixels`, which is an array containing the number of pixels in each of the detected connected components in the image. 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 the largest connect component. Using this index information, we can retrieve the list of pixel coordinates for this component in `CC.PixelIdxList`.

Output: Successfully monitored two f connections together, which are the largest connected component, and successfully eliminated



Task 6 - Morphological Reconstruction

In morphological opening, erosion typically removes small objects, and subsequent dilation tends to restore the shape of the objects that remains. However, the accuracy of this restoration relies on the similarity between the shapes to be restored and the structuring element.

Morphological reconstruction (MR) is a better method that restores the original shapes of the objects that remain after erosion.

```
clear all
close all
f = imread('assets/text_bw.tif');
se = ones(17,1); %Define a 17x1 structural element (SE).
g = imerode(f, se); %Corrosion the text image vertically along the 17-pixel direction
fo = imopen(f, se); % perform open to compare
fr = imreconstruct(g, f); %Using g as the seed image, perform morphological reconstruction
montage({f, g, fo, fr}, "size", [2 2])
```

Output: Different from expansion, it only restores the structures that originally exist in f and does not introduce new noise. The second image likely shows the result of **morphological opening** (`imopen`) or **erosion** (`imerode`). Many small, isolated pixels (noise) have been removed. The presence of numbers and

scattered text elements suggests an attempt to label connected components in the binary image (Third image) .

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`.

```

%% ...
numPixels = cellfun('numel', CC.PixelIdxList, 'UniformOutput', false);
[~, maxIndex] = max(numPixels);
largestCC = CC.PixelIdxList{maxIndex};
% ...

```

```

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

```

Output: It can be clearly seen that the enclosed part has been filled in.

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 *k*th 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 *k*th 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**.

Task 7 - Morphological Operations on Grayscale images

In this task, we will explore the effect of erosion and dilation on grayscale images.

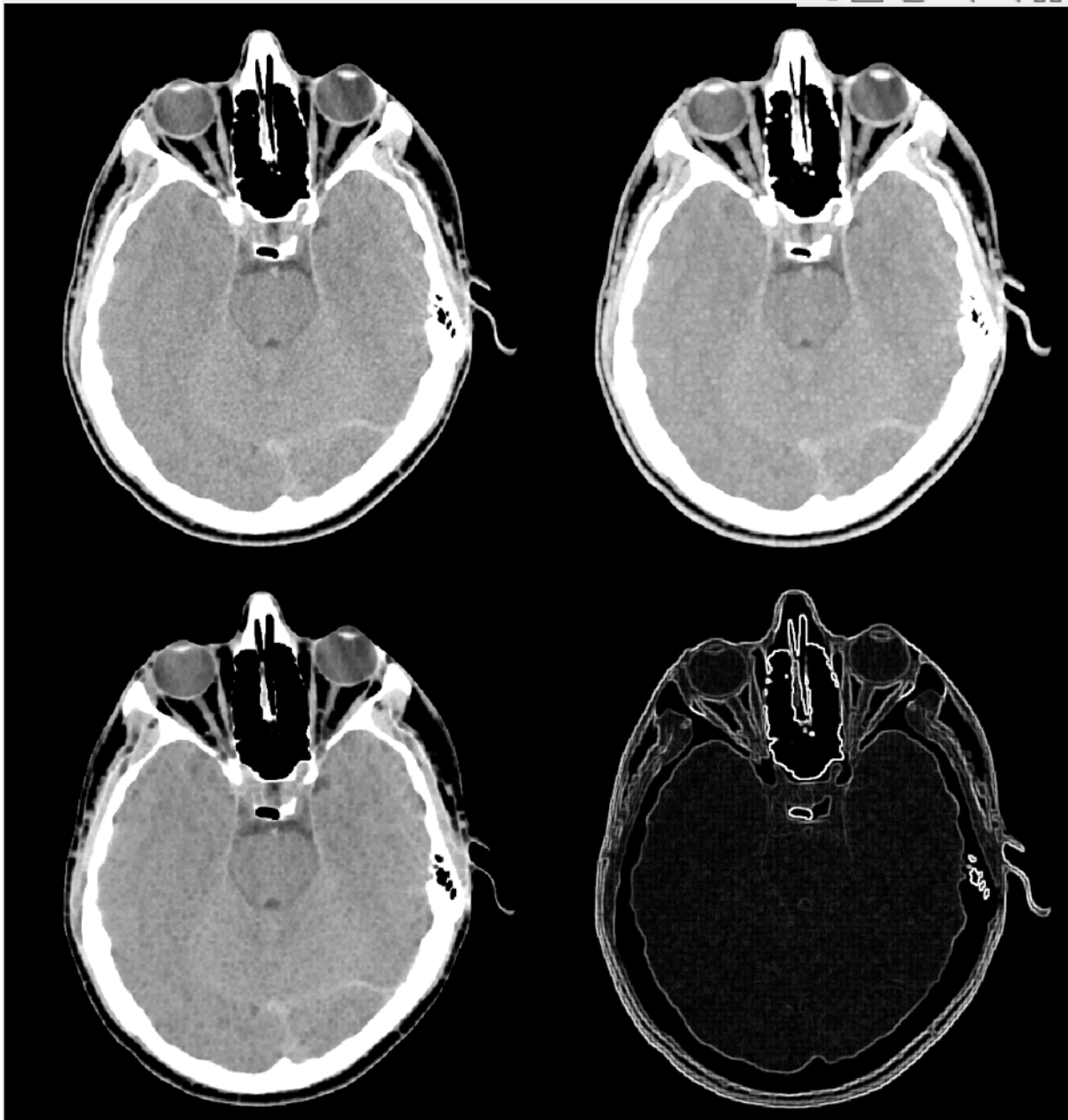
```
clear all; close all;
f = imread('assets/headCT.tif');
se = strel('square',3);
gd = imdilate(f, se);
ge = imerode(f, se);
gg = gd - ge;
montage({f, gd, ge, gg}, 'size', [2 2])
```

Output:

gd: Morphological dilation enhances bright regions, making white structures (such as bones) appear thicker.

ge: Morphological erosion shrinks bright regions, making structures appear thinner while enhancing darker areas.

gg: Highlights edges and transitions between different tissue densities, useful for edge detection and feature extraction in medical imaging.



Challenges

1. The grayscale image file '*assets/fillings.tif*' is a dental X-ray corrupted by noise. Find how many fills this patient has and their sizes in number of pixels.

When inspecting dental fillings, I found that direct binarization would cause confusion between the teeth and the fillings, so I first examined the histogram of the image and manually selected an appropriate threshold (`manual_threshold = 240`) to ensure that only the brightest filling areas were extracted. Then, I

used morphological closing to remove small black holes and connect the filling areas, making the detection results more complete. Finally, I calculated the number and size of the fillings through connected components analysis and visualized the entire processing process, ultimately obtaining accurate and clear filling detection results.

```
clear; close all; clc;
f = imread('assets/fillings.tif'); % Read dental X-ray images

figure;
imhist(f); title('Histogram of X-ray Image'); % manually select an appropriate th

manual_threshold = 240; % Manually selected grayscale threshold
bw = f > manual_threshold; % Perform binarization (only extract areas brighte

% Morphological closing operation (removing small black holes, connecting fi
se = strel('disk', 5);
bw_clean = imclose(bw, se);

% Calculate connected regions (number and size of fillers)
CC = bwconncomp(bw_clean);
stats = regionprops(CC, 'Area');
num_fillings = CC.NumObjects;
areas = [stats.Area];

montage({f, bw, bw_clean}, 'Size', [1 3]);
title('Original | Manual Threshold | Morphological Closing');

% Output filling material statistical information
fprintf('Detected Fillings: %d\n', num_fillings);
fprintf('Sizes of Fillings (in pixels): %s\n', mat2str(areas));
```

Detected Fillings: 2

Sizes of Fillings (in pixels): [10553 6069]



1. The file '*assets/palm.tif*' is a palm print image in grayscale. Produce an output image that contains the main lines without all the underlining non-characteristic lines.

In this task, my goal is to extract the main texture features of the palm while removing smaller noise and irrelevant details. To achieve this, I adopted a morphological erosion + morphological reconstruction + further denoising process to ensure that the final result can retain the main features while removing redundant interference information.

```
clear; close all; clc;

f = imread('assets/palm.tif');
f_gray = im2gray(f);

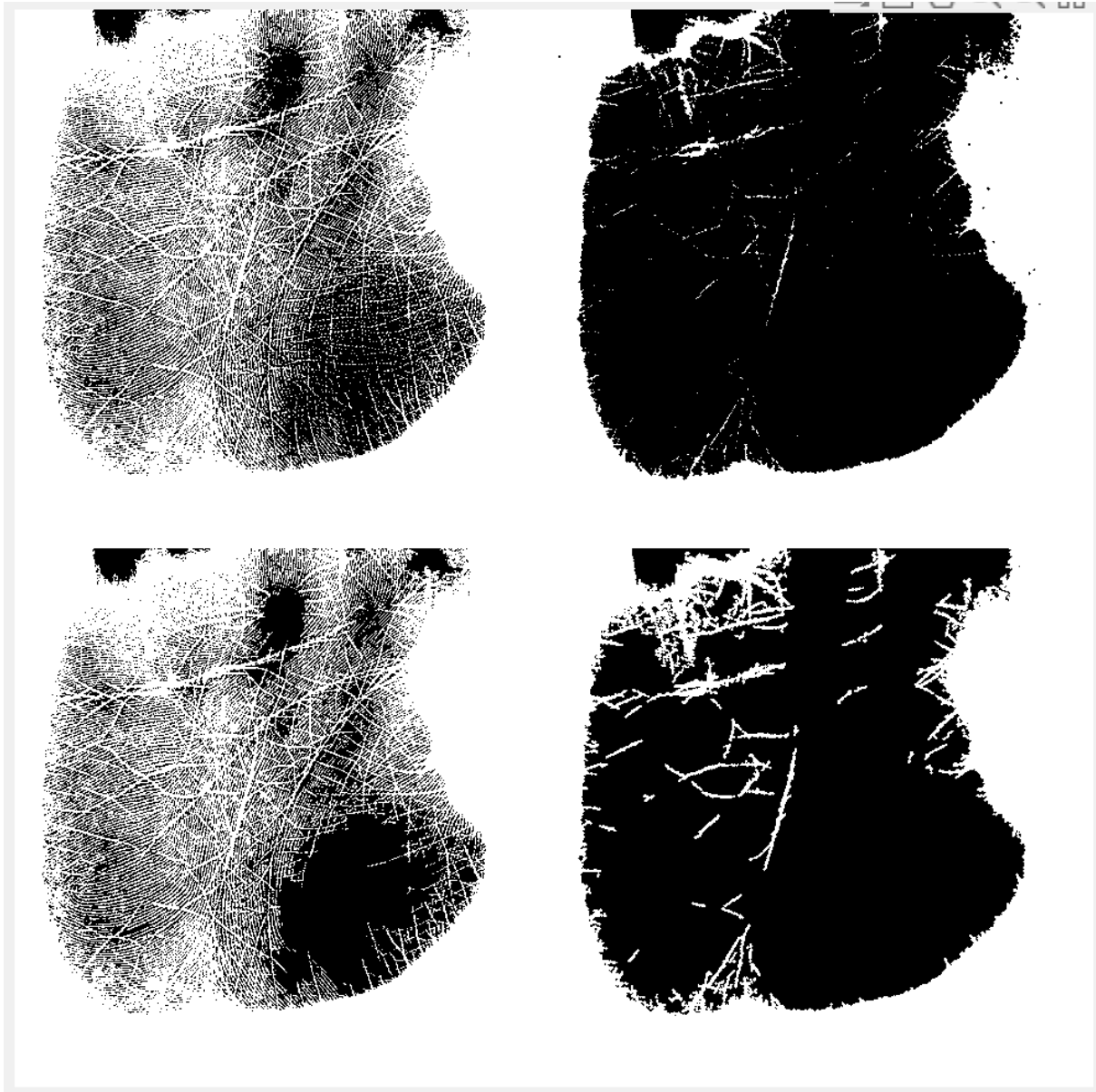
bw = imbinarize(f_gray); % Perform binarization (extract bright area features)

%Morphological Erosion - First remove small features, leaving large features
se = strel('diamond', 5); % diamond's structural elements
marker = imerode(bw, se);

% Morphological Reconstruction
bw_reconstruct = imreconstruct(marker, bw);

% 5. Further cleaning of small noise
bw_clean = imopen(bw_reconstruct, se);
bw_clean = bwareaopen(bw_clean, 500); % Only retain white areas larger than 500 pixels

montage({bw, marker, bw_reconstruct, bw_clean}, 'Size', [2 2]);
```



1. The file '*assets/normal-blood.png*' is a microscope image of red blood cells. Using various techniques you have learned, write a Matlab .m script to count the number of red blood cells.

I first convert the color image to a grayscale image and enhance the contrast. Then, I perform binarization using the Otsu threshold. To ensure the correct connection of red blood cells, I use dilation to expand the black areas, and then I calculate the number of connected regions to obtain the final red blood cell count. And I considered the possibility of cell overlap, so I manually set a threshold. When the pixel value is greater than 10,000, it is judged as two.

```
clear; close all; clc;  
f = imread('assets/normal-blood.png');
```

```

% Convert to grayscale image
f_gray = im2gray(f);
f_eq = adapthisteq(f_gray);

% Perform binarization (Otsu threshold)
bw = imbinarize(f_eq);

% 5. Morphological dilation (expanding the black area)
se = strel('disk', 5);
bw_inverted = ~bw;
bw_dilated = imdilate(bw_inverted, se);

% Calculate connected regions
CC = bwconncomp(bw_dilated);
stats = regionprops(CC, 'Area');

% Set area threshold and calculate cell count
min_area = 200; % Minimum pixel area (discard values less than this)
max_area = 10000; % Maximum pixel area of a single red blood cell (values g

areas = [stats.Area];
num_single_cells = sum(areas >= min_area & areas <= max_area);
num_large_cells = sum(areas > max_area);
num_cells = num_single_cells + 2 * num_large_cells;

montage({f_gray, f_eq, bw, bw_dilated}, 'Size', [2 2]);
% Detected number of red blood cells
fprintf('Detected number of red blood cells:%d\n', num_cells);

```

Detected number of red blood cells: 42