

Image analysis for biology

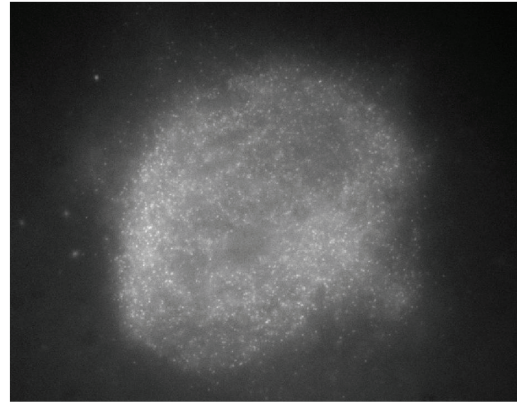
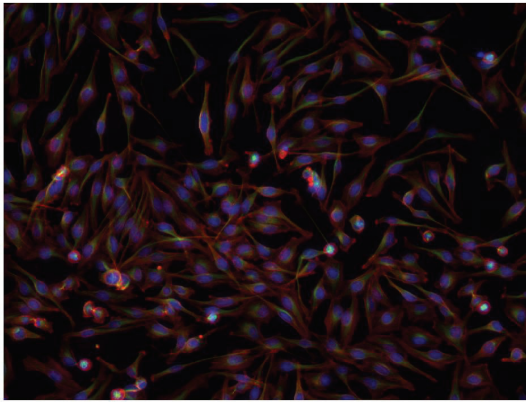
Thanks to

Hao Yuan Kueh, Eugenio Marco, and Sivaraj Sivaramakrishnan

1) Why Image Analysis?	2
Image Analysis Strategies	2
Layout of this tutorial	4
2) Basics	5
Image types, data classes and image classes	7
Basic Segmentation using Thresholding	9
Image histograms	9
Metamorph Stack Files	13
EXERCISES	14
3) Contrast adjustments	15
4) Spatial Filtering	18
Smoothing filters	20
Edge detection filters	21
Laplacian filter	22
Median filter	23
EXERCISES	23
5) Morphological image processing	24
Dilation	24
EXERCISE	26
Erosion	28
EXERCISE	29
Opening and closing	29
EXERCISE	29
Additional useful image processing tools	30
Filling holes	30
Clearing border objects	31
6) Image Segmentation	31
Edge detection	31
Morphological Watersheds	34
EXERCISES	37
7) Analysis of motion in biological images	40
Designing Graphical User Interfaces in MATLAB	40
Kymographs	45
Difference Images, Maximum Intensity Projections	46
Image Cross-correlation	48
EXERCISES	49
Particle Tracking	50
8) REFERENCES	51

1) Why Image Analysis?

Biological images contain a wealth of objects and patterns, which may convey information about underlying mechanism in biology. Take a look at the following microscopy images:



The left microscopy image shows a field of view of tissue-culture cells. One can ask: how many cells are there in this field of view? What is the average size? How much DNA is in each of the cells? How are the microtubule and actin cytoskeletons organized spatially? For the movie of the speckled spindle on the right, one can ask: What is the distribution of polymer mass in the spindle? What is the flux rate? Does it depend on the position along the spindle? Where is monomer getting incorporated and lost?

Image processing and analysis provides a means to extract and quantify objects and patterns in image data and obtain answers to meaningful biological questions. It offers two advantages over traditional more manual methods of analysis: 1) Human vision, while highly sensitive, can be easily biased by pre-conceived notions of objects and concepts; automated image analysis provides an unbiased approach to extracting information from image data and testing hypotheses. 2) Once an image-analysis routine is devised, it can be applied to a large number of microscopy images, facilitating the collection of large amounts of data for statistical analysis.

Image Analysis Strategies

Image analysis involves the conversion of features and objects in image data into quantitative information about these measured features and attributes. Microscopy images in biology are often complex, noisy, artifact-laden and consequently require multiple image processing steps for the extraction of meaningful quantitative information. An outline of a general strategy for image analysis is presented below:

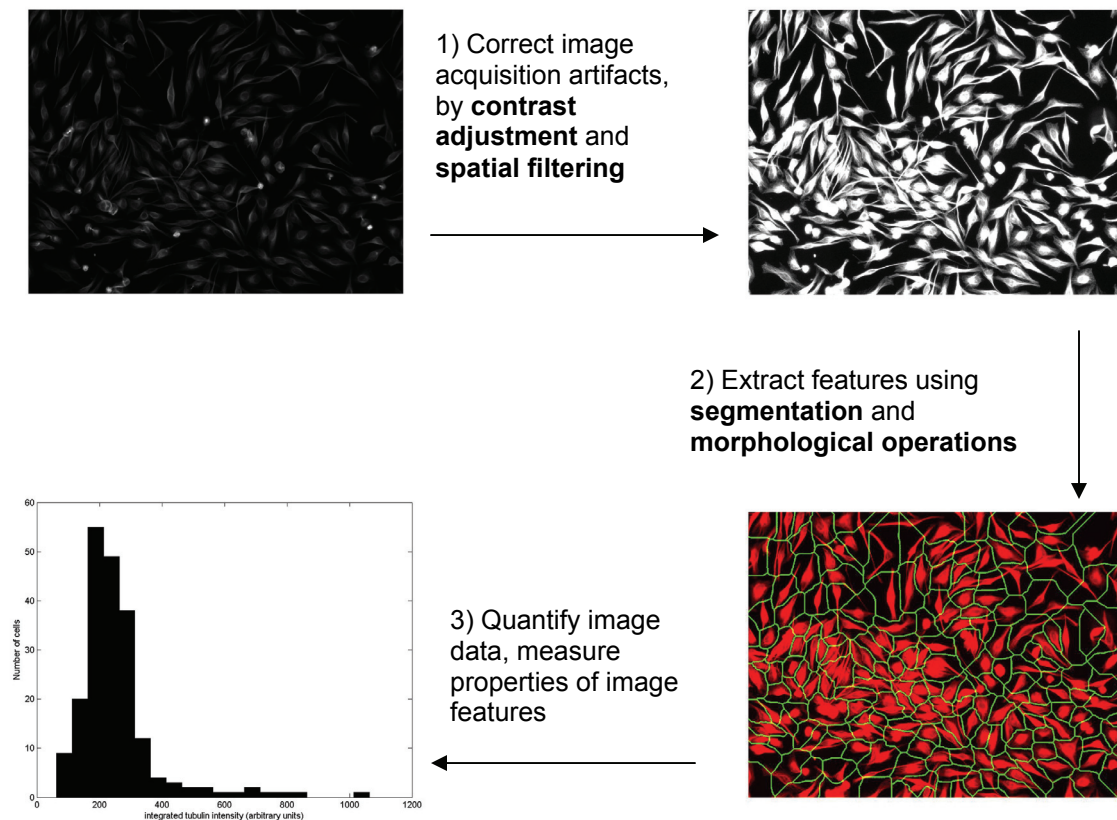
1) The starting point in image analysis typically involves a digital image acquired using a CCD camera. Raw microscopy images obtained on digital CCD cameras are subject to various imperfections of the image acquisition setup, such as noise at low light levels, uneven illumination, defective pixels, etc... We often need to first process the image to correct for such defects and also

to enhance the contrast to accentuate features of interest in the image for subsequent analysis. In section II, we introduce various image transformation and spatial filtering techniques that can be used for this purpose.

2) Having corrected artifacts and enhanced contrast in the images, we can apply various computational techniques to extract features and patterns from the images. In the following section, we describe various tools of morphological image processing and image segmentation that can be used for this purpose.

3) After biological important features have been segmented from images, we can then derive quantitative information from these features and objects. MATLAB provides a set of tools that can be used to measure the properties of regions; the matrix representation of images in MATLAB also allows for easy manipulation of data and calculation of quantities from microscopy images.

Here is an outline of the process:



Layout of this tutorial

The goal of this tutorial is to take you through the aforementioned process of extracting data from raw microscopy images in MATLAB. In Section I, you will learn to load and save images and convert between data types in MATLAB. You will also go through a simple example of segmentation and data extraction from segmentation, using as an example an image of rice grains over a dark background.

The simple example of segmentation covered in Section I will highlight some image artifacts and imperfections that will prevent accurate extraction of data from raw images. Sections II-IV cover various techniques that can correct for defects in either raw images or segmented images. Section II describes techniques of contrast adjustment, which can be used to improve the contrast of features of interest to facilitate the segmentation process. Section III discusses the basic principles and some simple applications of spatial filtering. Spatial filters serve many different purposes in image analysis, from smoothing to sharpening boundaries to detecting edges. Section IV discusses morphological image processing, which is a powerful technique for correcting defects in segmented images. Each section discusses both the basic math and ideas behind these techniques, and also gives practical implementation of the tools in MATLAB.

Section V describes two additional segmentation techniques – 1) segmentation based on edge detection, and 2) segmentation by the watershed algorithm. These techniques are complementary to threshold-based segmentation and may become useful for different types of images. Section V then closes with a set of problems based on biological images that give you the opportunity to review all the techniques you have learnt and also integrate them in a simple image processing routine. They are not easy, but you are strongly encouraged to try them and perhaps bang your heads against the wall in the process! They will probably give you a good taste of what a real-life image analysis problem is like. Good luck!

2) Basics

In this section we present the basics of working with images in Matlab. We will see how to read, display, write and convert images. We will also talk about the way images are represented in Matlab and how to convert between the different types.

The Matlab command for reading an image is

```
imread('filename')
```

Note that we suppress the output with a semicolon, otherwise we will get in the output all the numbers that make the image. **'filename'** is the name of a file in the current directory or the name of a file including the full path. Try

```
>> f = imread('chest-xray.tif');
```

We now have an array **f** where the image is stored

```
>> whos f
```

Name	Size	Bytes	Class
f	494x600	296400	uint8 array

Grand total is 296400 elements using 296400 bytes

f is an array of class **uint8**, and size 494x600. That means 494 rows and 600 columns. We can see some of this information with the following commands

```
>> size(f)  
ans =  
  
    494    600
```

```
>> class(f)  
ans =  
  
uint8
```

We will talk later on about the different image classes.

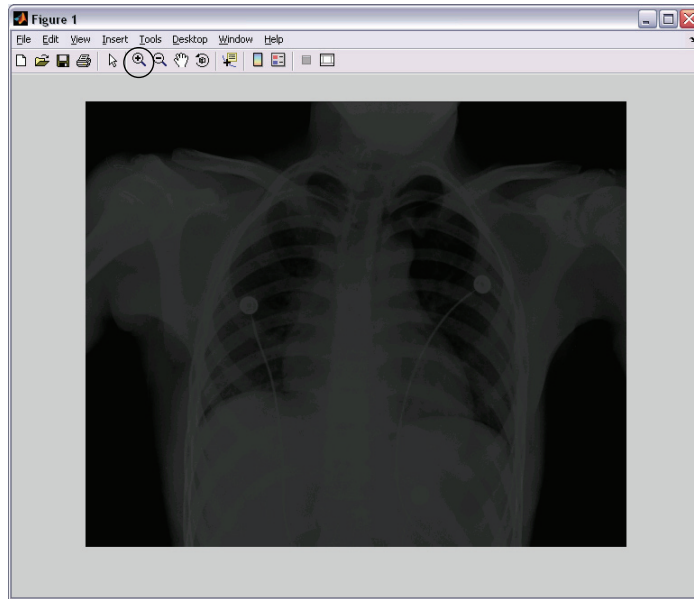
Sometimes it is useful to determine the number of rows and columns in an image. We can achieve this by means of

```
>> [M, N] = size(f);
```

To display the image we use **imshow**

```
>> imshow(f)
```

You will get a window similar to this



Note that in the figure toolbar we have buttons that allow us to zoom parts of the image. The syntax `imshow(f, [low high])` displays all pixels with values less than or equal to **low** as black, all pixels with values greater or equal to **high** as white. Try

```
>> imshow(f, [10 50])
```

Finally,

```
>> imshow(f, [])
```

sets the variable **low** to the minimum value of array **f** and **high** to its maximum value. This is very useful for displaying images that have a low dynamic range. This occurs very frequently with 16-bit images from a microscope.

We can also display portions of an image by specifying the range

```
>> imshow(f(200:260, 150:220))
```

Another matlab tool available to display images and do simple image manipulations is `imtool`. Try

```
>> imtool(f)
```

In the figure window we have now available the following tools: overview, pixel region, image information, adjust contrast and zoom. Try them.

Images can be written to disk using the function **imwrite**. Its format is

```
imwrite(f, 'filename')
```

with this syntax, **filename** must include the file format extension. Alternatively

```
imwrite(f, 'filename', format)
```

saves **f** using **format**. For example

```
>> imwrite(f, 'test', 'jpeg', 'quality', 25)
```

In the help you can find more information about available formats and their options.

Image types, data classes and image classes

There are different image types and image classes available in MATLAB. The two primary image types you will be working with are as follows

- Intensity images
 - **uint16** [0, 65535] (CCD cameras on microscopes)
 - **uint8** [0, 255] (From your standard digital camera)
 - **double** [-10^{308} , 10^{308}]
- Binary images (black and white)
 - **logical**, 0 or 1

Raw images typically come in the form of an unsigned integer (**uint16** denotes 16-bit unsigned integer, and **uint8** denotes 8-bit unsigned integer). However floating-point operations (mathematical operations that involve a decimal point, such as **log(a)**) can only be done with arrays of class **double**.

Hence, to work on a raw image, first convert it from **uint16** or **uint8** to **double** using the **double** function:

```
>> f = imread('actin.tif');  
>> g = double(f);
```

Now type

```
>> whos;
```

to see the different data types associated with each variable. Note that while the data type changes, the actual numbers after the conversion remain the same.

Many MATLAB image processing operations operate under the assumption that the image is scaled to the range [0,1]. For instance, when **imshow** displays an **double** image, it displays an intensity of 0 as black and 1 as white. You can automatically create a scaled **double** image using **mat2gray**:

```
>> h = mat2gray(g);
```

Question: How would you write a function to manually scale a **double** image?
Certain image processing commands only work with scaled double images.

Finally, we can convert an intensity image into a binary image using the command **im2bw(f, T)**, where **T** is a threshold in the range [0, 1]. Matlab converts **f** to class **double**, and then sets to 0 the values below **T** and to 1 the values above **T**. The result is of class **logical**. See the following example.

We wish to convert the following **double** image

```
>> f = [1 2; 3 4]
```

f =

1	2
3	4

to binary such that values 1 and 2 become 0 and the other two values become 1. First we convert it to the range [0, 1]

```
>> g = mat2gray(f)
```

g =

0	0.3333
0.6667	1.0000

We can convert the previous image to a binary one using a threshold, say, of value 0.6:

```
>> gb = im2bw(g, 0.6)
```

gb =

0	0
1	1

Note that we can obtain the same result using relational operators

```
>> gb = f > 2
```

gb =

0	0
1	1

Binary images generated by thresholding often form the basis for extracting morphological features in microscopy images. In the next section, we will extract some basic quantitative information about objects in an image by first using thresholding to generate a binary image and then using the **regionprops** command to extract quantitative information from the binary image.

Question: How might you represent a color image?

Basic Segmentation using Thresholding

Many biological images comprise of light objects over a constant dark background (especially those obtained using fluorescence microscopy), in such a way that object and background pixels have gray levels grouped into two dominant modes. One obvious way to extract the objects from the background is to select a threshold T that separates these modes:

$$\begin{aligned} g(x,y) &= 1 && \text{if } f(x,y) > T \\ &= 0 && \text{otherwise} \end{aligned}$$

where $g(x,y)$ is the thresholded binary image of $f(x,y)$. We can implement the thresholding operation in MATLAB by the following function:

$$\mathbf{g} = \mathbf{im2bw}(\mathbf{f}, \mathbf{T})$$

The first argument \mathbf{f} gives the input image, and the second argument \mathbf{T} gives the threshold value.

Image histograms

We need to choose a threshold value T that properly separates light objects from the dark background. Image histograms provide a means to visualize the distribution of grayscale intensity values in the entire image. They are useful for estimating background values, determining thresholds, and for visualizing the effect of contrast adjustments on the image (next section). The matlab function to visualize image histograms is **imhist**

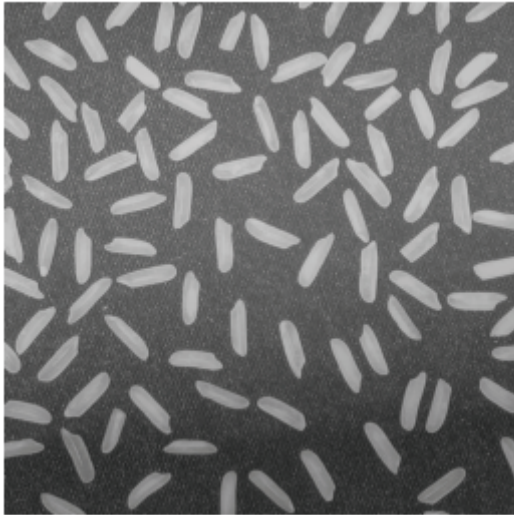
```
>> f = imread('chest-xray.tif');  
>> imhist(f);
```

The histogram has 256 bins by default. The following command makes 20 bins

```
>> imhist(f,20);
```

A good value for T can be obtained by visually inspecting the image histogram obtained using the **imhist** command:

```
>> im = imread('rice.png');
```



```
>> imhist(im);
```

Based on the histogram, pick a grayscale value manually that separates the light rice grains from the dark background. Then threshold the image and display the results.

MATLAB provides a function **graythresh** that automatically computes a threshold value:

$$\mathbf{T} = \text{graythresh}(\mathbf{im})$$

where **im** is the input image and **T** is the resulting threshold. **graythresh** calculates the threshold value by essentially maximizing the weighted distances between the global mean of the image histogram and the means of the background and foreground intensity pixels.

EXAMPLE

In this example, we threshold the image of rice grains opened above:

```
>> im = imread('rice.png');  
>> im = mat2gray(im);
```

Calculate the threshold value:

```
>> level = graythresh(im);
```

and create a new binary image using the obtained threshold value:

```
>> imb = im2bw(im,level);
```

Note that the thresholding operation segments the rice grains quite well. However, a problem in this image is that the rice grains near the bottom of the image aren't segmented well – the background is uneven and is low at the bottom, leading to incorrect segmentation. We'll see a way to correct for this uneven background using image processing later.

Using the binary image, we can then calculate region properties of objects in the image, such as area, diameter, etc... An object in a binary image is a set of white pixels (ones) that are connected to each other. We can enumerate all the objects in the figure using the **bwlabel** command:

```
[L, num] = bwlabel(f)
```

where L gives the labeled image, and num gives the number of objects. To label the binary image of the rice grains, type:

```
>> [L, N] = bwlabel(imb);
```

Now look at the labeled image **L** using **imtool**. What are the values of the objects in the pixels? Adjust the contrast to see the range of intensity values in the image.

Once the image has been labeled, use the **regionprops** command to obtain quantitative information about the objects:

```
D = regionprops(L, properties)
```

There's a lot of useful statistical information about objects that can be extracted using **regionprops**. Here's a list:

' Area '	' EulerNumber '	' Orientation '
' BoundingBox '	' Extent '	' Perimeter '
' Centroid '	' Extrema '	' PixelIdxList '
' ConvexArea '	' FilledArea '	' PixelList '
' ConvexHull '	' FilledImage '	' Solidity '
' ConvexImage '	' Image '	'SubarrayIdx'
' Eccentricity '	' MajorAxisLength '	
' EquivDiameter '	' MinorAxisLength '	

Extract the area and perimeter of individual objects in the labeled image as follows:

```
>> D = regionprops(L, 'area', 'perimeter');
```

NOTE:

The information in **D** is stored in an object called a structure array. A structure array is a variable in MATLAB that contains multiple fields for storage of information. You can access the field information in **D** as follows:

```
>> D
```

```
D =
```

```
151x1 struct array with fields:  
    Area  
    Perimeter
```

Access an individual element in the structure array by referring to its index in parenthesis:

```
>> D(1)
```

```
ans =
```

```
        Area: 145  
    Perimeter: 58.2843
```

TIP: Using structure arrays in MATLAB

Access an individual field within in the structure array by referring to its name after a dot (.). For instance:

```
>> D(1).Area
```

You can assign values to structure arrays using the assignment operator (=):

```
>> D(1).Area = 888;    (n.b. we do not condone data manipulation)
```

And define new fields by using a new field name during assignment:

```
>> D(1).test = 1
```

To get an array of values from one field in the structure **D**, type:

```
>> w = [D.Area]
```

You can get the mean, standard deviation and the full histogram of the Areas from **w** as follows:

```
>> meanarea = mean(w)  
>> stdarea = std(w)
```

```
>> hist(w)
```

Now look at the histogram. Did the histogram look the way you expected? How do you account for the features in the histogram? (Why are there so many values less than 50? Why are there values around 450? Why is there a spread of values between 50 and 200?)

Metamorph Stack Files

One popular format for microscopy movies produced by Metamorph are `.stk` files, which are similar to tiff files, but contain additional information like the exposure time, timestamp, stage position, etc. François Nedelec has created a matlab function called `stkread` extracts the information from the files (found in the `materials` directory). Try this:

```
>> S = stkread('yeastcdc42.stk')
```

```
S =
```

```
1x31 struct array with fields:
```

```
    filename  
    NewSubfiletype  
    width  
    height  
    bits  
    photo_type  
    info  
    x_resolution  
    y_resolution  
    resolution_unit  
    software  
    datetime  
    MM_private1  
    MM_stack  
    MM_wavelength  
    MM_private2  
    data
```

`S` is a structure array with the number of elements equal to the number of frames in the stack. The images are stored in the field `data`. To see the first frame we can do

```
>> imshow(S(1).data, [])
```

the image acquisition times (in ms) are stored in the field `MM_stack`. We can extract it for all frames using

```
for index = 1:length(S)  
    S(index).timestamp=S(index).MM_stack(4)-S(1).MM_stack(4);  
end
```

EXERCISES

1. Load the images `pollen.tif` and `bone-scan-GE.tif`. Display them with `imtool`. Look at the pixel values in some area. Convert them into `double`. Display and look at the values again.
2. Use thresholding and `regionprops` to count the nuclei in the following images:

```
nuclei1.tif  
nuclei2.tif
```

How many nuclei are there in the picture?
What is the average area of a nucleus?

Which image was easier? Why? How can the segmentation be improved? We'll see different ways shortly.

Try to write your programs in a generic fashion.

Comment and maybe even include functions.

We will expand the program you are writing now in future exercises.

Good habits now will make changing the program later easy!

3a. Eliminate “bad” rice granules from the `rice.png` picture.

Use `regionprops` to identify the rice you want to eliminate.

Use the `find` function to change eliminate these grains from the labeled image.

(You will probably want to write a loop that goes through each object in the labeled file)

Compare the original image to your new image (`imshow` and `hist`).

How well did you do? Qualitatively, how many real grain got eliminated/how many bad grains remain?

b. Relabel your resulting image from 3a so that your objects are label 1 to total objects.

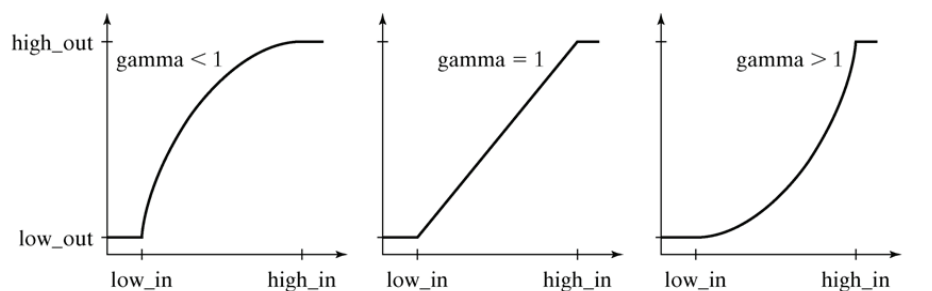
Compare the mean and std of the area of these object versus the original objects.

c. Now calculate the total intensity of each rice grain. You will have to use a combination of the original rice image and your new labeled image.

3) Contrast adjustments

Often, images have a low dynamic range and many of its features are difficult to see. We will present different intensity transformations that will improve the appearance of the images. Improving the appearance of an image does not merely serve an aesthetic role – often, it can help improve the performance of image segmentation algorithms and feature recognition.

During contrast adjustment, the intensity value of each pixel in the raw image is transformed using a transfer function to form a contrast-adjusted image. The most common transfer function is the gamma contrast adjustment:



Here **low_in** and **low_high** give the low and high grayscale intensity values for the contrast adjustment, and **gamma** gives the exponent for the transfer function.

The basic matlab function for contrast adjustment is `imadjust`. It has the syntax:

```
g = imadjust(f,[low_in high_in],[low_out high_out],gamma)
```

Without the optional arguments (`g = imadjust(f)`), the output image has 1% of the data in `f` is saturated at low and high intensities.

Try the following commands (negative image)

```
>> f = imread('chest-xray.tif');  
>> imshow(f)  
>> g1 = imadjust(f);  
>> imshow(g1)  
>> g2 = imadjust(f, [0 1], [1 0]);  
>> figure,imshow(g2)
```

How does changing the value of gamma affect the image? What types of images are different values of gamma good for?

EXERCISE

Check out the DNA and tubulin staining of these mammalian cancer cell lines:

```
U2OS Tubulin.tif
U2OS Dapi.tif
```

Find also the maximum and minimum of the images, and try to apply different gamma adjustments to the images. What features of the images are accentuated during each transformation?

The negative of an image can be obtained with

```
>> g = imcomplement(f);
```

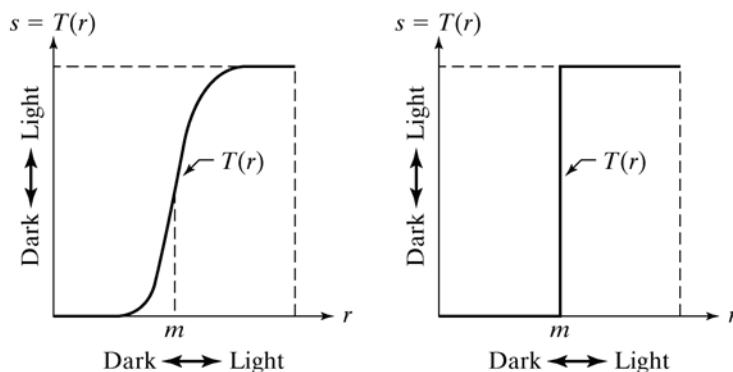
Try these other commands

```
>> g3 = imadjust(f, [0 0.2], [0 1]);
>> figure, imshow(g3)
>> g4 = imadjust(f, [], [], .2);
>> figure, imshow(g4)
```

Two other useful intensity transformations are the logarithmic and contrast-stretching. They can be implemented in matlab with the functions:

```
g = c*log(1+double(f))
g = 1./(1+(m./(double(f) + eps)).^E)
```

In the following figure we can see the shape of the contrast-stretching transformation. The value of the threshold given by m and the steepness of the threshold is given by E



In the limiting case of $E \gg 1$, the image is thresholded at the intensity m . Try the following commands

```
>> f = imread('spectrum.tif');
```



```
>> imshow(f)

>> g = im2uint8(mat2gray(log(1+double(f))));

>> figure, imshow(g)
```

EXERCISES

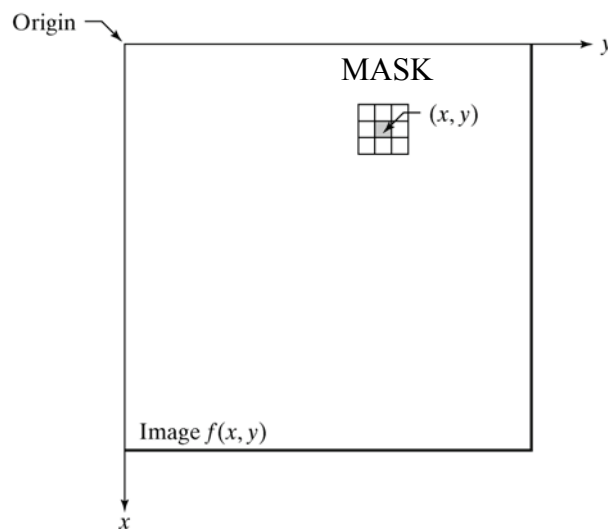
1. Create a function that does a contrast stretching transformation for different values of **m** and **E**. Apply it to **spectrum.tif**. Also try a logarithmic transformation.
2. Load the images **pollen.tif** and **bone-scan-GE.tif**. Display them with **imtool**. Transform the images and display them again.

4) Spatial Filtering

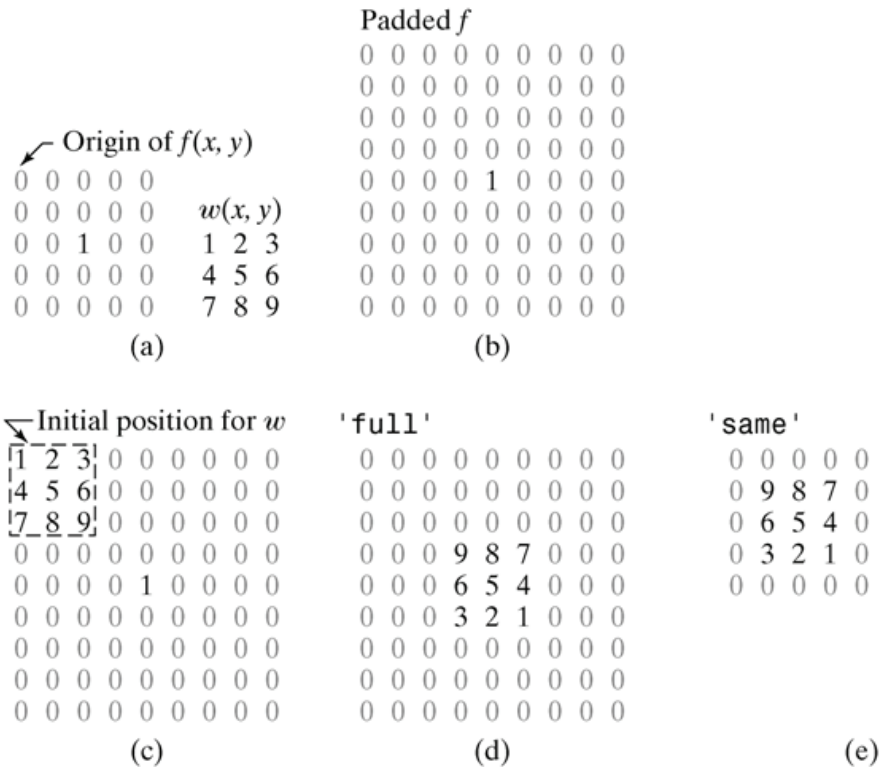
A large variety of image processing tasks can be accomplished by a technique called spatial filtering. Spatial filtering involves a mask, consists of an array of values ($a-i$) and has a center (gray):

a	b	c
d	e	f
g	h	i

The mask is placed on the image of interest:



and translated across all possible pixel positions on the image. A new (filtered) image is produced by replacing the intensity value at the center by a linear combination of the intensity values of the center pixel and all neighboring pixels covered by the mask. The coefficients ($a-i$) in the mask array give the weights of each pixel in the linear combination. Here is an example:



Filters can perform many different functions. The function of the filter is essentially determined by the value of the mask coefficients ($a-i$). In this tutorial, we consider primarily two classes of filters: 1) Smoothing filters, which reduce noise in an image, and 2) edge detection (or derivative) filters, which can detect object borders.

The basic command in matlab is:

```
g = imfilter(f, w)
```

The toolbox supports a number of predetermined 2-D linear spatial filters. The command to generate them is fspecial:

```
w = fspecial(type, parameters)
```

Type	Syntax and Parameters
'average'	<code>fspecial('average', [r c])</code> . A rectangular averaging filter of size $r \times c$. The default is 3×3 . A single number instead of $[r c]$ specifies a square filter.
'disk'	<code>fspecial('disk', r)</code> . A circular averaging filter (within a square of size $2r + 1$) with radius r . The default radius is 5.
'gaussian'	<code>fspecial('gaussian', [r c], sig)</code> . A Gaussian lowpass filter of size $r \times c$ and standard deviation <code>sig</code> (positive). The defaults are 3×3 and 0.5. A single number instead of $[r c]$ specifies a square filter.
'laplacian'	<code>fspecial('laplacian', alpha)</code> . A 3×3 Laplacian filter whose shape is specified by <code>alpha</code> , a number in the range $[0, 1]$. The default value for <code>alpha</code> is 0.5.
'log'	<code>fspecial('log', [r c], sig)</code> . Laplacian of a Gaussian (LoG) filter of size $r \times c$ and standard deviation <code>sig</code> (positive). The defaults are 5×5 and 0.5. A single number instead of $[r c]$ specifies a square filter.
'motion'	<code>fspecial('motion', len, theta)</code> . Outputs a filter that, when convolved with an image, approximates linear motion (of a camera with respect to the image) of <code>len</code> pixels. The direction of motion is <code>theta</code> , measured in degrees, counterclockwise from the horizontal. The defaults are 9 and 0, which represents a motion of 9 pixels in the horizontal direction.
'prewitt'	<code>fspecial('prewitt')</code> . Outputs a 3×3 Prewitt mask, <code>wv</code> , that approximates a vertical gradient. A mask for the horizontal gradient is obtained by transposing the result: <code>wh = wv'</code> .
'sobel'	<code>fspecial('sobel')</code> . Outputs a 3×3 Sobel mask, <code>sv</code> , that approximates a vertical gradient. A mask for the horizontal gradient is obtained by transposing the result: <code>sh = sv'</code> .
'unsharp'	<code>fspecial('unsharp', alpha)</code> . Outputs a 3×3 unsharp filter. Parameter <code>alpha</code> controls the shape; it must be greater than 0 and less than or equal to 1.0; the default is 0.2.

The best way to get an idea for what a given filter is doing is to look at its shape.

Smoothing filters

The following is a 3x3 average filter:

$1/9$	$1/9$	$1/9$
$1/9$	$1/9$	$1/9$
$1/9$	$1/9$	$1/9$

which can be obtained using the following command:

```
>> fspecial('average',3)
```

The average filter replaces a value at the center of the mask by an average of all grayscale intensity values in the mask.

The average filter tends to smooth out sharp image features smaller than the size of the filter. It is important to pick a filter size that smooths out the noise in your image, but at the same time preserves the features of interest.

In the following example we will apply an average filter of size 31 to a pattern and see the results of using different options.

```
>> f = imread('original_test_pattern.tif ');
>> f2 = im2double(f);

>> imshow(f2)
>> w = ones(31);
>> gd = imfilter(f2, w);
>> figure,imshow(gd, [])
```

Note the presence of blurring at the image boundaries. When the filter reaches a boundary, it does not have enough pixel information to filter near the image boundaries. By default, MATLAB pads the boundaries of the image with zeros and runs the filter across this expanded image, giving rise to the blurring observed. It is possible to specify alternate padding conditions. For instance, try the options 'replicate' or 'symmetric' in the filtering:

```
>> gd2 = imfilter(f2, w, 'symmetric');
>> figure,imshow(gd2, [])
```

A Gaussian filter is one that has peak at the center of the mask, and has a Gaussian decay away from center. Check out, for instance:

```
fspecial('gaussian', 10, 5)
```

The Gaussian filter also performs averaging, but performs a weighted average to give more emphasis on pixels near the center of the mask. The Gaussian filter is more commonly used for smoothing than the average filter; it is radially symmetric and is hence less likely to introduce image artifacts such as sharp directional discontinuities.

GENERAL RULE #1: In general, the sum of all values in mask for a smoothing filter must equal one. Why?

Edge detection filters

Let's take a look at the Prewitt filter:

```
>> f = fspecial('prewitt')
```

1	1	1
0	0	0
-1	-1	-1

What does it do? Where does it give maximum contrast? Try out the filter on the rice grain example of Section I. Where is the grayscale intensity maximum? minimum? zero?

GENERAL #2: Generally speaking, a filter provides maximum contrast in regions of the image that resemble itself.

Now flip the filter around 90 degrees by transposition:

```
>> f = f'
```

Try applying this filter to the rice grain example again.

The Prewitt filter and the Sobel filter belong to a class of filters called ‘first-derivative’ filters, which essentially calculate the slope of grayscale intensities in an image in a given direction. They give a maximum value (or minimum value) at regions with discontinuities in grayscale values and are frequently used in edge detection. In section V, we will combine use edge detection as a means to recognize closed boundaries of objects in an image, and use that information to segment the image.

GENERAL RULE #3: In general, the sum of all values in mask for a smoothing filter must equal zero. Why?

Laplacian filter

```
>> f = fspecial('laplacian',0);
```

The Laplacian filter looks like this:

1	1	1
1	-8	1
1	1	1

What features does this filter pick up? The sum of all values in this mask is zero. What does this mean? Run this filter on **moon.tif**. Where do the regions of maximum contrast / minimum contrast occur?

The Laplacian filter mathematically calculates the local curvature of grayscale intensities in an image and serves as a method for picking up edges that is independent of edge orientation. Use the

imtool command to examine the edges of the filtered image of **moon.tif**. Note that when you approach the edge of a bright object from the outside, the pixel intensity values first increase, then decrease. This change reflects the change in curvature expected along a discontinuity between a bright object and a dark background.

Now form a new image by subtracting the laplacian filtered image from the original image of **moon.tif**. How does the resultant image differ from the original? This combination of filtering operations is a commonly used technique and is (counterintuitively) referred to as an 'unsharp mask'. Why does it produce this effect? HINT: Think about what happens along an edge when you do the image subtraction.

Median filter

All the previous filters are called linear, because the final pixel value is the result of a linear combination of pixel values from the original image. Nonlinear filters, on the contrary, assign nonlinear combinations of the pixel values to the filtered image. MATLAB has functions to implement nonlinear filters, but we won't study them in detail. We consider, an important nonlinear filter, the median filter, which has its own implementation in matlab. Matlab also has a function to generate noise in an image. We will see in the following example how to corrupt an image (with **imnoise**) and how to try to restore it with a median filter.

```
g = medfilt2(f, [m n])
```

Try the following

```
>> f = imread('ckt-board-orig.tif');  
>> imshow(f)  
>> fn = imnoise(f, 'salt & pepper', 0.2);  
>> figure, imshow(fn)  
>> gm = medfilt2(fn);  
>> figure, imshow(gm)
```

EXERCISES

1. Try adding different types of noise to **ckt-board-orig.tif**. Look in the help for the different options in **imnoise**. Then choose the appropriate filter to remove the noise.
2. Repeat for **pollen.tif**.

5) Morphological image processing

Mathematical morphology is a powerful tool that can be used to extract features and components from an image. It is often used to pre-process or post-process images to facilitate analysis. In morphology, a small shape (structuring element) is translated across the image during the course of processing. Certain mathematical logic operations are performed on the image using the structuring element to generate the processed image.

In this section, we first introduce dilation and erosion, two fundamental operations in mathematical morphology. We then describe more complex morphological operations obtained by combining erosion and dilation. We describe a handful of morphological tools that are especially useful in image analysis.

Dilation

Dilation is an operation that grows objects in a binary image. The thickening is controlled by a small structuring element. In the following figure you can see the structuring element on the right and the result after applying dilation on a rectangle.


```

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 0
0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 0
0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

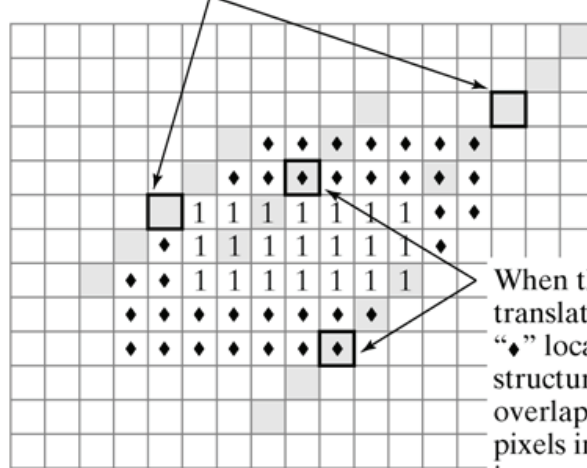
```

```

      1
     1
    1
   1
  1
 1
1

```

The structuring element translated to these locations does not overlap any 1-valued pixels in the original image.



When the origin is translated to the “♦” locations, the structuring element overlaps 1-valued pixels in the original image.

```

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0
0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0
0 0 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0
0 0 0 0 1 1 1 1 1 1 1 1 1 1 0 0 0
0 0 0 1 1 1 1 1 1 1 1 1 1 1 0 0 0
0 0 0 1 1 1 1 1 1 1 1 1 1 0 0 0 0
0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

```

Try the following example

```
>> A = imread('broken-text.tif');
```

```
>> B = [0 1 0; 1 1 1; 0 1 0];  
>> A2 = imdilate(A, B);  
>> imshow(A), figure, imshow(A2)
```

1. Generate different structuring elements and apply them to `broken-text.tif`. Try also applying several dilations consecutively.

The **strel** function can be used to build structuring elements in a variety of shapes. They are stored in a special format that makes use of decompositions of the elements into simpler structuring elements. Its syntax is:

```
se = strel(shape, parameters)
```

See the table for possible structuring elements.

```
>> se = strel('diamond', 5)
```

EXERCISE

1. Load '`broken-text.tif`'. Dilate the image with different structuring elements. And see the different results.
2. Do the same with `noisy-fingerprint.tif`

Syntax Forms	Description
<code>se = strel('diamond', R)</code>	Creates a flat, diamond-shaped structuring element, where <i>R</i> specifies the distance from the structuring element origin to the extreme points of the diamond.
<code>se = strel('disk', R)</code>	Creates a flat, disk-shaped structuring element with radius <i>R</i> . (Additional parameters may be specified for the disk; see the <code>strel</code> help page for details.)
<code>se = strel('line', LEN, DEG)</code>	Creates a flat, linear structuring element, where <i>LEN</i> specifies the length, and <i>DEG</i> specifies the angle (in degrees) of the line, as measured in a counterclockwise direction from the horizontal axis.
<code>se = strel('octagon', R)</code>	Creates a flat, octagonal structuring element, where <i>R</i> specifies the distance from the structuring element origin to the sides of the octagon, as measured along the horizontal and vertical axes. <i>R</i> must be a nonnegative multiple of 3.
<code>se = strel('pair', OFFSET)</code>	Creates a flat structuring element containing two members. One member is located at the origin. The second member's location is specified by the vector <i>OFFSET</i> , which must be a two-element vector of integers.
<code>se = strel('periodicline', P, V)</code>	Creates a flat structuring element containing $2 \times P + 1$ members. <i>V</i> is a two-element vector containing integer-valued row and column offsets. One structuring element member is located at the origin. The other members are located at $1 \times V$, $-1 \times V$, $2 \times V$, $-2 \times V$, ..., $P \times V$, and $-P \times V$.
<code>se = strel('rectangle', MN)</code>	Creates a flat, rectangle-shaped structuring element, where <i>MN</i> specifies the size. <i>MN</i> must be a two-element vector of nonnegative integers. The first element of <i>MN</i> is the number rows in the structuring element; the second element is the number of columns.
<code>se = strel('square', W)</code>	Creates a square structuring element whose width is <i>W</i> pixels. <i>W</i> must be a nonnegative integer scalar.
<code>se = strel('arbitrary', NHOOD)</code> <code>se = strel(NHOOD)</code>	Creates a structuring element of arbitrary shape. <i>NHOOD</i> is a matrix of 0s and 1s that specifies the shape. The second, simpler syntax form shown performs the same operation.

Erosion

Erosion shrinks or thins objects in a binary image. After erosion the only pixels that survive are those where the structuring element fits entirely in the foreground.

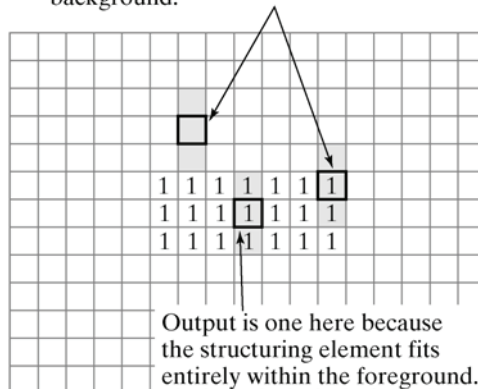
```

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0
0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0
0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

```

$$\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

Output is zero in these locations because the structuring element overlaps the background.



```

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

```

In the following example one can see how different erosions affect the features of an image

```

>> A = imread('wirebond-mask.tif');
>> imshow(A)
>> se = strel('disk',10);
>> A2 = imerode(A,se);

```

```
>> figure, imshow(A2)
```

Try disks of radius 5 and 20.

EXERCISE

1. Load `noisy-fingerprint.tif` and `broken-text.tif`. Apply different erosions.

Opening and closing

Combinations of morphological operations can be very useful and help remove many artifacts present in images. This will become very useful after segmenting an image.

The first operation we will see is opening, which is an erosion followed by dilation. Opening smoothes object contours, breaks thin connections and removes thin protrusions. After opening, all objects smaller than the structuring element will disappear.

Closing is a dilation followed by erosion. Closing smoothes object contours, joins narrow breaks, fills long thin gulfs and fills holes smaller than the structuring element.

The following example will show the utility of opening and closing.

```
>> f = imread('shapes.tif');
>> imshow(f)
>> se = strel('square',20);
>> fo = imopen(f, se);
>> figure, imshow(fo)
>> fc = imclose(f, se);
>> figure, imshow(fc)
>> foc = imclose(fo, se);
>> figure, imshow(foc)
```

EXERCISE

1. Reopen `rice.png`

Use `imopen` to eliminate all the rice grains.

Convert all numbers so that they are between 0 and 1

Use `mesh` to view image (see details of 'mesh' command in help section)

Use this new array to modify the original `rice.png`. Plug this back into your code from exercise 3 (the previous rice grain exercise). Is it easier or harder to correctly pick all the rice grains.? How does the quantitation compare?

2. Load and apply opening and closing to **noisy-fingerprint.tif** and **broken-text.tif**.

Additional useful image processing tools

A very useful morphological transformation to subtract the background from an image is the so called tophat. Tophat is the subtraction of an opened image from the original. One can do opening in gray images, removing all features smaller than the structuring element. In the following example we will show its usefulness

```
>> I = imread('rice.png');
>> imshow(I)
>> background = imopen(I,strel('disk',15));
>> figure, imshow(background)
>> figure
>> surf(double(background(1:8:end,1:8:end))),zlim([0 255]);
>> set(gca,'ydir','reverse');
>> I2 = imsubtract(I,background);
>> figure, imshow(I2)
```

The same result can be obtained using

```
>> I2 = imtophat(I, strel('disk',15));
>> figure, imshow(I2)
```

What happens when you change the size of the structuring element? What criteria should you use to choose the appropriate size?

Filling holes

Frequently, after some morphological operation we need to fill the holes in a binary image. For example, we detect the boundary of a cell and want to obtain an object which is filled and covers the cell. In this example we will see its effect in text.

```
g = imfill(f, 'holes')

>> f = imread('book-text.tif');
>> imshow(f)
>> f2 = imfill(f, 'holes');
>> figure, imshow(f2)
```

Clearing border objects

After segmenting an image there are usually objects touching the border of the image. Since we can not obtain complete information about them it is usually the best to remove them.

`imclearborder` is the matlab function to remove objects touching the border.

```
g = imclearborder(f)

>> f = imread('book-text.tif');
>> imshow(f)
>> f2 = imclearborder(f);
>> figure, imshow(f2)
```

6) Image Segmentation

Segmentation refers to the process in which an image is subdivided into constituent regions or objects. These objects can be further processed or analyzed for the extraction of quantitative information. Biological image data is usually messy and noisy, and as a result difficult to segment properly. Multiple image processing steps are often required in the process of segmentation. We often combine segmentation with various morphological processing and filtering techniques described above to achieve accurate and robust segmentation of an image.

We looked at a basic segmentation technique earlier – thresholding. In this section, we look at other complementary segmentation techniques that may be useful in different situations.

Edge detection

One way to find boundaries of objects is to detect discontinuities in intensity values at the edge of a region. These discontinuities can be found by calculating the first and/or second order derivatives of an image. The first derivative of choice in image processing is the gradient, defined as the vector:

$$\text{grad } f = [G_x \ G_y]$$

where $G_x = df/dx$ and $G_y = df/dy$ are the partial derivatives in the horizontal and vertical directions of the image. The magnitude of this vector is

$$|\text{grad } f| = (G_x^2 + G_y^2)^{1/2}$$

The gradient vector points in the direction of steepest ascent. The angle of steepest ascent is given by:

$$a(x,y) = \tan^{-1}(G_y/G_x)$$

We can estimate the derivatives G_x and G_y digitally by linearly filtering the image with the following 3 by 3 kernels:

z_1	z_2	z_3
z_4	z_5	z_6
z_7	z_8	z_9

Image neighborhood

-1	-2	-1
0	0	0
1	2	1

$$G_x = (z_7 + 2z_8 + z_9) - (z_1 + 2z_2 + z_3)$$

-1	0	1
-2	0	2
-1	0	1

$$G_y = (z_3 + 2z_6 + z_9) - (z_1 + 2z_4 + z_7)$$

Sobel

-1	-1	-1
0	0	0
1	1	1

-1	0	1
-1	0	1
-1	0	1

Prewitt

The Prewitt and Sobel operators are among the most used in practice for computing digital gradients. The Prewitt masks are simpler to implement than the Sobel masks, but the latter have slightly superior noise-suppression characteristics.

To gain an intuition for what a gradient image looks like, try opening some grayscale images into **im** and creating a gradient image as follows:

```
>> h = fspecial('sobel');
>> Gy = imfilter(im, h);
>> Gx = imfilter(im, h');
```

We can also visualize the magnitude of the gradient:

```
>> G = (Gx.^2 + Gy.^2).^(1/2)
```


Note that the magnitude of the gradient is high at an edge, where the intensity of a pixel changes rapidly between neighboring pixels. A thresholding operation is then applied to the image **G** to segment the boundary pixels. Objects can then be identified by filling in the holes of detected closed contours using the **imfill** function (see previous section)

MATLAB provides a general edge detection command that is sensitive to horizontal or vertical edges or both. The general syntax for this function is

$$\mathbf{g} = \text{edge}(\mathbf{f}, \text{'method'}, \text{parameters})$$

Where **f** is the input image, **method** gives to the edge detection algorithm used, and **parameters** gives the parameters that accompany the algorithm. **g** is binary image where edges have a value of 1.

EXAMPLE

Let's apply edge detection to the segmentation of actin filaments:

```
>> im = imread('actin2.tif')
```

First try to use thresholding (in the first section) to segment the actin filaments.

Let's run the two edge detectors on the microtubules image for comparison:

```
>> edge_sobel = edge(im, 'sobel');  
>> edge_prewitt = edge(im, 'prewitt');
```

The most powerful edge detector is the Canny edge detector provided by the function **edge**. The Canny edge detector incorporates information about the direction of the image gradient in determining whether a given pixel is an edge pixel. It also has two thresholds, a weak threshold T_1 and a strong threshold $T_2 > T_1$. Pixels above the weak threshold count as an edge only if there are pixels above the strong threshold adjacent to them. The Canny edge detector gives good edge detection for biological images, which are typically noisy and have less well-defined edges.

$$\mathbf{BW} = \text{edge}(\mathbf{I}, \text{'canny'}, \mathbf{THRESH}, \mathbf{SIGMA})$$

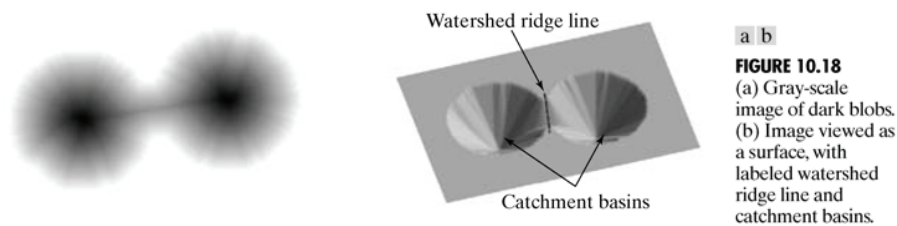
Where **THRESH** is a two element vector in which the first element is the low threshold, and the second element is the high threshold. **SIGMA** is the standard deviation of a Gaussian filter that is applied to the image prior to edge detection. Try applying the canny edge detector the microtubule image, and play around different values of the threshold and smoothing.

Edge detection can be used in conjunction with hole filling (**imfill**) to recognize and segment objects with well defined boundaries.

Morphological Watersheds

In the previous sections, we discussed segmentation based on 1) detection of discontinuities in an image, and 2) thresholding. Morphological watersheds provide a complementary approach to the segmentation of objects. It is especially useful for segmenting objects that are touching one another.

To understand the watershed transform – we view a grayscale image as a topological surface, where the values of $\mathbf{f}(\mathbf{x}, \mathbf{y})$ correspond to heights:



Consider the topographic surface on the right. Water would collect in one of the two catchment basins. Water falling on the watershed ridge line separating the two basins would be equally likely to collect into either of the two catchment basins. Watershed algorithms then find the catchment basins and the ridge lines in an image.

The algorithm works as follows: Suppose a hole is punched at each regional local minimum and the entire topography is flooded from below by letting the water rise through the holes at a uniform rate. Pixels below the water level at a given time are marked as flooded. When we raise the water level incrementally, the flooded regions will grow in size. Eventually, the water will rise to a level where two flooded regions from separate catchment basins will merge. When this occurs, the algorithm constructs a one-pixel thick dam that separates the two regions. The flooding continues until the entire image is segmented into separate catchment basins divided by watershed ridge lines.

The watershed algorithm is implemented in the MATLAB image processing toolbox as:

$$\mathbf{L} = \text{watershed}(\mathbf{f})$$

where \mathbf{f} is the input image and \mathbf{L} is a labeled matrix image having positive integer values at different regions and 0 at the watershed ridge lines.

The key behind using the watershed transform for segmentation is this: ***Change your image into another image whose catchment basins are the objects you want to identify.*** In the following examples, we consider different ways to pre-process images to make them amenable to watershed segmentation.

EXAMPLE – watershed segmentation by the distance transform

Consider the task of separating two touching objects circles in this binary image.

```
>> im = imread('circles.tif');
```



How can we modify this image so its catchment basins are two circular objects? To do this we'll use another new tool in the Image Processing Toolbox: **bwdist**, which computes the *distance transform*. The distance transform of a binary image is the distance from every pixel to the nearest nonzero-valued pixel, as this example shows.

1	1	0	0	0	0.00	0.00	1.00	2.00	3.00
1	1	0	0	0	0.00	0.00	1.00	2.00	3.00
0	0	0	0	0	1.00	1.00	1.41	2.00	2.24
0	0	0	0	0	1.41	1.00	1.00	1.00	1.41
0	1	1	1	0	1.00	0.00	0.00	0.00	1.00

A small binary image (left) and its distance transform (right).

Calculate the image distance transform of image:

```
>> A = bwdist(im)
```

This image is not very useful, because there is only one catchment basin spanning the entire image. Instead, try computing the distance transform of the image's complement:

```
>> B = bwdist(~im);
```

This image is closer, but we need to negate the distance transform to turn the two bright areas into catchment basins.

```
>> C = -B
```

Now there is one catchment basin for each object, so we call the watershed function.

```
>> L = watershed(C) ;
```

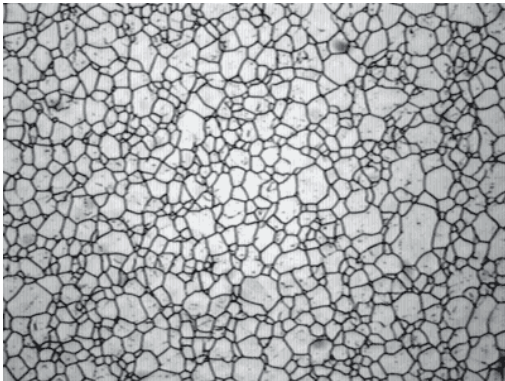
L is called a *label matrix*, and it contains positive integers corresponding to the locations of each catchment basin. We can use the zero-valued elements of L , which are located along the watershed lines, to separate the objects in the original image.

```
>> im(L == 0) = 0;
```

EXAMPLE – oversegmentation

Consider the following microscopy image of steel grains:

```
>> im = imread('steel.tif')
```



try running the watershed algorithm on the complement of this image:

```
>> L = watershed(imcomplement(im)) ;
```

As you can see, there are too many watershed ridge lines that do not correspond to objects in which we are interested. Oversegmentation is common problem when using watersheds. One way to counter the problem of oversegmentation is to remove the minima that are too shallow using the following command:

```
g = imhmin(f, H)
```

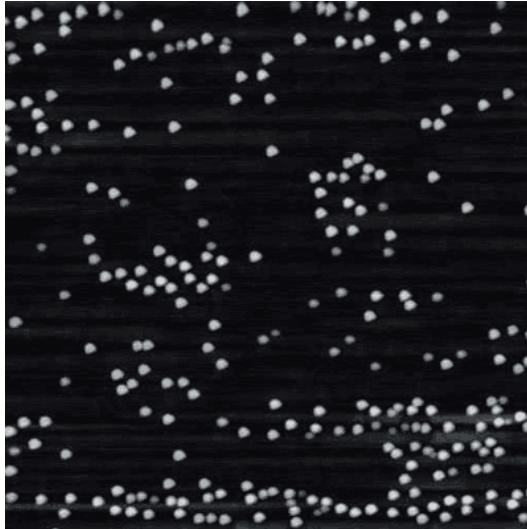
which takes an input image \mathbf{f} and suppresses the minima whose depth is less than \mathbf{H} . Try this:

```
>> im2 = imcomplement(im) ;  
>> im3 = imhmin(im2, 20) ;  
>> L = watershed(im3) ;
```

The result is much improved.

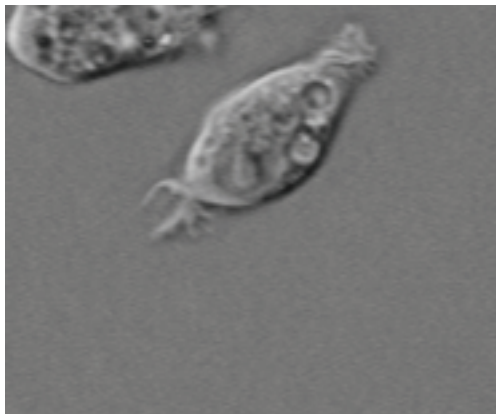
EXERCISES

1. **Quantum dots.** Devise a segmentation strategy to segment the following image of quantum dots (quantumdots.tif):



Extract the following parameters from the image: 1) number of dots, 2) histogram of areas and 3) histogram of integrated intensities

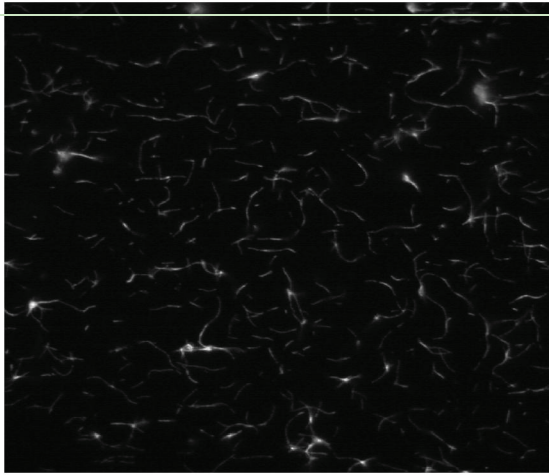
2. **DIC images of cells.**



Find the boundary of this cell using any of your favourite image segmentation techniques. (dic_cell.tif)

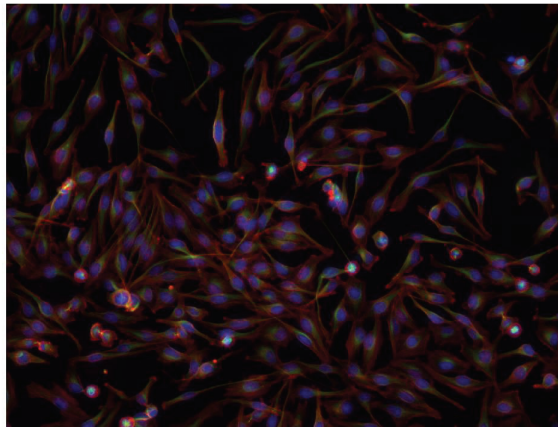
3. **Actin filaments**

Devise an image processing strategy to obtain the distribution of filaments from this image, and subsequently calculate 1) the mean filament length, 2) the variance in filament length,



and 3) the number of filaments. Note: image segmentation may be complicated by the low light levels in the image. (actin.tif)

4. **Fixed cells.** Here are some cells that are fixed and stained for actin (red), tubulin (green) DNA (blue), and a histone marker (not shown). Devise an image processing strategy to segment the cells. You may operate on any of the color channels in the image (or a multiple of them). This problem is especially tricky because many of the cells are touching. (4color_cells.stk)



5. **Additional Exercises.** Develop a criteria to distinguish between the DNA and tubulin morphologies of these cells treated with different drugs:

```
U2OS DAPI Taxol.tif
U2OS DAPI VX680.tif
U2OS DAPI.tif
U2OS Tubulin Taxol.tif
U2OS Tubulin VX680.tif
```

6. Measuring transferrin endocytosis

The images in the `\transferrin\` directory are in matched pairs. The first image shows DAPI staining and the second image shows staining of transferrin that has been endocytosed. Write a program to segment the cells (most likely taking advantage of the nuclei staining) and quantify how much transferrin is being endocytosed. Your quantification can be just the sum of the total amount of transferrin endocytosed or take into account the number/qualities of individual vesicles (more advanced).

7. Finding cell borders in a sheet:

The images in the `\sheet\` directory are in matched pairs of DAPI and membrane staining. Write a program to identify each cell and its borders.

7) Analysis of motion in biological images

The previous sections focused on extracting information from individual images using a variety of filtering, morphological processing and segmentation techniques. In this section, we turn our attention to the analysis of dynamics and motion in a movie, which comprises a sequence of images taken at successive time intervals. From the movies, we can extract information about dynamical processes in biology, from the kinetics of biochemical reactions to the movements of proteins and cellular structures.

We begin by describing ways of visualizing motion in a sequence of images. A common approach to visualizing dynamics involves converting the dimension of time in a series of images into a spatial dimension. Kymographs are essentially two-dimensional plots with time running along one of the dimensions. We next discuss ways of extracting quantitative information by performing mathematical manipulations between images in a stack. Cross-correlation and projection operations fall into this category. Finally we discuss various techniques for tracking particles and their movements over time.

Designing Graphical User Interfaces in MATLAB

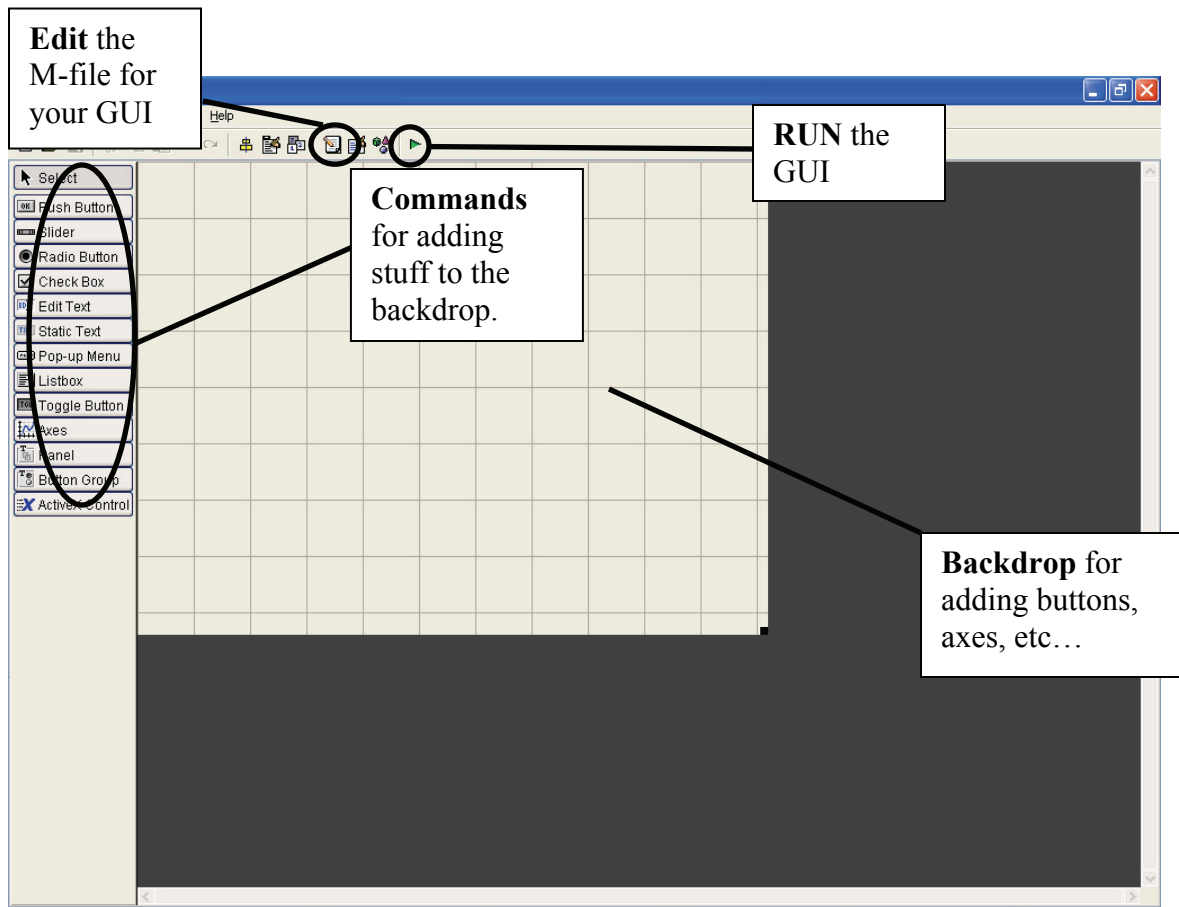
Before learning how to analyze motion in microscopy movies, we will first learn how to create graphical user interfaces (GUIs) in MATLAB, which will make it easier to interact with your functions and change and test parameters. It will also make it easier for other people to use the image processing routines that you have written.

We here go through the process of a creating a GUI for viewing Metamorph stacks. You can easily customize and extend this GUI to incorporate the image processing routines that you developed.

The basic MATLAB interface for creating GUIs in MATLAB is the GUI developer's environment, which is run by the command **guide**:

```
>> guide
```

Create a new Blank GUI (Default) by pressing OK. You will see something like this:



Here are the key components:

1. The **GUI Backdrop** is the background on which you add GUI components such as buttons, text, axes, text boxes and all sorts of other bells and whistles.
2. The **Commands** allow you to add various bells and whistles to the Backdrop.
3. Once you have designed the basic layout for your GUI, you can attach various functions to your GUI components by pressing the button to **Edit** your M-file.
4. Once everything is complete, run the GUI by pressing the **Run** button!

Before adding any GUI components onto the backdrop, let's first take a look at the M-file behind the GUI. Press the button 'M-file editor' to edit your M-file. When prompted, save the GUI file as 'mygui'.

When the file opens, scroll down to:

```
function mygui_OpeningFcn(hObject, eventdata, handles, varargin)
```

This is the function that executes before the GUI is made visible. Any initialization code is added below here. For the meantime, let's add a variable that acts as a flag as to whether a stack is loaded:

```
handles.loaded = 0;
```

Handles is a structure that acts a global variable that can be seen by all functions inside the GUI M-file. To save the changes to handles made by a function, we run the following line:

```
guidata(hObject, handles);
```

So the value `handles.loaded = 0` can now be seen by all components inside the GUI.

When you add components to your GUI, new functions will appear in your M-file. Let's add a button for loading a Metamorph stack image into MATLAB. Add a **Push Button** onto the backdrop. Double click on it to edit its properties.

Set the 'Tag' of the Push Button to `loadstack`. A unique tag identifies each component in the GUI, and the tag will appear in the names of GUI component functions in the M-file. Also, set the string to 'Load Stack' and set its tag as `loadstack`.

Check the MATLAB help for explanations of the properties of GUI components.

Go to the editor for the M-file. You should see the following function appear:

```
function loadstack_Callback(hObject, eventdata, handles)
```

As you can see, the name of the function is determined by its tag. The callback function executes whenever the button is pressed. Add the following lines of code:

```
% Call up a GUI to load to choose the Metamorph stack
CurrentDir=pwd;
[filename, path] = uigetfile('*.stk', 'Choose stack:');

%load the stack, if it exists
if(filename)
    [stk,stacklength]=stkread([path filename]);
    handles.stk = stk;           % entire stack structure written by metamorph
    handles.loaded = 1;
else
    cd(CurrentDir);
    return
end

figure(1); imshow(stk(1).data,[]);

guidata(hObject, handles);
```

The code calls up **uigetfile** to retrieve the filename of the Metamorph stack file to be read, and reads the stack using the command **stkread**. The stack is then added to the handles structure, which is then saved using the command `guidata(hObject, handles)`. We also set the flag `handles.loaded = 1` to show that a file has been loaded.

Now let's add a slider to the backdrop that can be used to cycle through different planes in the stack. Add a **slider**. The slider has arrow buttons and a box that can be moved, and is set to assume the value of the current image plane.

Now double click on the slider to modify its properties. Set:

```
'Tag'           = planeslider
'Min'           =      1
'Max'           =     10
'Value'         =      1
```

The above commands initialize the minimum, maximum and current values of the slider. The current value gives the current image plane that is to be displayed in the figure window. When we load a Metamorph stack, we will set the maximum value of the slider to equal the number of image planes in the stack.

The following function appears in the M-file:

```
function planeslider_Callback(hObject, eventdata, handles)
```

The function executes whenever the slider is touched in the GUI. The lower value of the slider is 1 and the upper value of the slider is N, which is the number of planes in the stack. We can set these values after loading the stack by adding this to loadstack_Callback:

```
set(handles.planeslider,'Min',1,'Max',stacklength);
```

Note that the handles structure also holds information regarding the status of various GUI components, which can be get or set in the manner shown above. For instance, the current position of the slider can be obtained as follows:

```
get(handles.planeslider, 'Value');
```

To get a comprehensive list of GUI component properties that can be get or set, refer to the MATLAB help. We now add the following code to planeslider_callback, to basically update the image in the figure window every time the slider is moved:

```
if (handles.loaded == 1)
    % get the value of the current plane
    slideval = get(handles.planeslider, 'Value');
    plane = round(slideval);

    % retrieve image from stack and find minimum and maximum for contrast
    % adjustment
    im = handles.stk(plane).data;
    imin = min(min(im));
    imax = max(max(im));

    % display image (optimized to prevent flickering)
    figure(1)
```

```
imagesc('cdata', handles.stk(plane).data, 'erasemode','none', [imin imax]);  
colormap('gray');  
children = get(gca,'children');  
delete(children(end));  
xlabel(['plane ' num2str(plane)]);  
end
```

This code retrieves the current value of the slider, retrieves the image from the stack and displays the image in figure(1). So by pressing the arrows in the slider, we can scroll through all the planes in the stack.

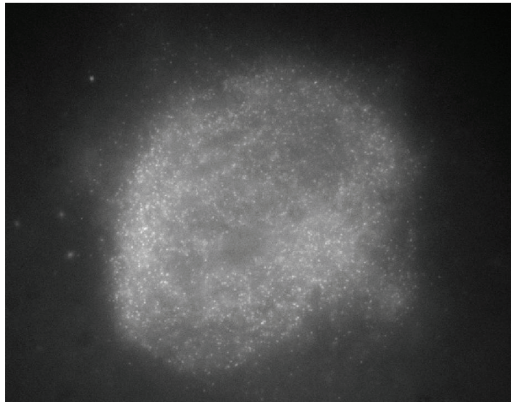
There are many features that can be added to the above GUI. For instance, we can add text boxes to display the plane number or the timestamp of the current frame of the movie inside the GUI. We can further extend this basic GUI by adding image processing buttons and functions for processing images and stacks and to save the results of the image analysis. We can even add edit text boxes to allow us to vary the segmentation parameters easily. Efficient GUI design can enhance productivity and allow for other people to use your routines more easily.

Kymographs

One way to visualize dynamical behavior is to plot time along a spatial axis. Time traces are one-dimensional graphs where time t is placed on a single axis and a dynamical tracked $F(t)$ is tracked over time. Kymographs are a two-dimensional analog of time traces, where time t occupies one axis, space along a one-dimensional contour x occupies another axis, and the dynamical variable $F(x,t)$ is visualized as an image.

Kymographs provide a fast and convenient way to visualize motion and dynamics in microscopy images. Load the following image of a spindle labeled with speckle amounts of tubulin using the GUI you just created.

```
>> stk = stkread('fluxingspindle.stk');
```



Play the movie a few times. Note that the tubulin speckles flux towards the both spindle poles. One way to measure the rate of flux is to create a kymograph along a straight line that runs from one spindle pole to the other.

We start off by creating a scan of intensities along a straight line connecting the spindle poles. One way to do this in MATLAB is to use the command **improfile**, which returns the pixel-value cross-sections along line segments:

```
c = improfile(I,xi,yi)
```

where **I**, the input image, and **xi**, **yi**, which are equal-length vectors specifying the spatial coordinates of the endpoints of the line segments. To find the x and y coordinates for the endpoints, we can use the command

```
[y,x,p] = impixel
```

which allows the user to interact with the current figure with the mouse to obtain coordinate values. Each mouse-click adds an coordinate value to **x** and **y** and a right mouse click terminates the interactive part of the command.

We take the intensity profiles along the line segment for all images in the plane, and then concatenate all the intensity profiles to form a two dimensional image:

```
stk = handles.stk;
imshow(stk(2).data,[]); % display image
[y,x,p] = impixel;      % use impixel to interactively find endpoints of the spindle

% build kymograph

kymo = [];
% loop through all image plans, find the linescan, and concatenate to form
% the kymograph
for i = 1:length(stk)
    im = im2double(stk(i).data);
    c = improfile(im, x, y);
    kymo = [kymo c];      % concatenation
end
% transpose kymograph
kymo = kymo'
```

Note that the resultant image has many diagonal lines. These lines correspond to moving tubulin speckles, and the velocity of motion is given by the slope of these lines. Note that image analysis can be performed on the kymographs for the extraction of quantitative information, just like any other regular image.

EXERCISES

1. Add the kymograph as an additional tool in the GUI you created above. Provide a means to save the kymograph into a file.
2. Instead of simply taking an intensity profile along a line, it is common to take an average (or maximum) of intensities of the image in the direction orthogonal to the line segment when making a kymograph. How would you implement this?
3. Implement a kymograph that takes intensity values along the circumference of a circle of a given radius and center. Can you think of any biological situations when this might be useful?

Difference Images, Maximum Intensity Projections

It is often possible to visualizing dynamics in image stacks by performing mathematical manipulations involving neighboring images in a stack, or even by performing manipulations on the

entire stack. In this section, we discuss two possible manipulations; difference images and maximum intensity projections.

Difference Images

A difference image of successive frames in a movie is given by:

$$D_j(x,y) = I_{j+1}(x,y) - I_j(x,y)$$

where D is the difference image, and j is a given plane in the stack. Difference images highlight features of the image that change rapidly over time, much in the same way that a spatial gradient enhances contrast around an edge. The change is usually brought about by movement of an object in the image, or by a kinetic process that has not reached steady-state (like photobleaching).

EXERCISE

1. **Photobleaching of CDC-42.** Load **yeastcdc42.stk**. Take difference images between successive frames. What do the peaks and troughs in intensity correspond to?

Maximum Intensity Projections

A maximum intensity projection of an entire image stack is given by

$$M(x,y) = \max_j(I_j(x,y))$$

The maximum on the right hand side is the maximum in intensity value at a given pixel value over all stacks in an image. Maximum projections collapse the entire dynamics of the stack onto a single plane, and is especially useful for visualizing entire trajectories of moving particles on a single plane.

EXERCISE

1. **Visualizing listeria trajectories.** *Listeria* is an intracellular pathogen that moves inside the cell by polymerizing actin at its rear surface. Load **listeriacells.stk** and use a maximum intensity projection to visualize the entire trajectory of the moving bug. Can you apply image processing tools to trace and segment the trajectory?

Image Cross-correlation

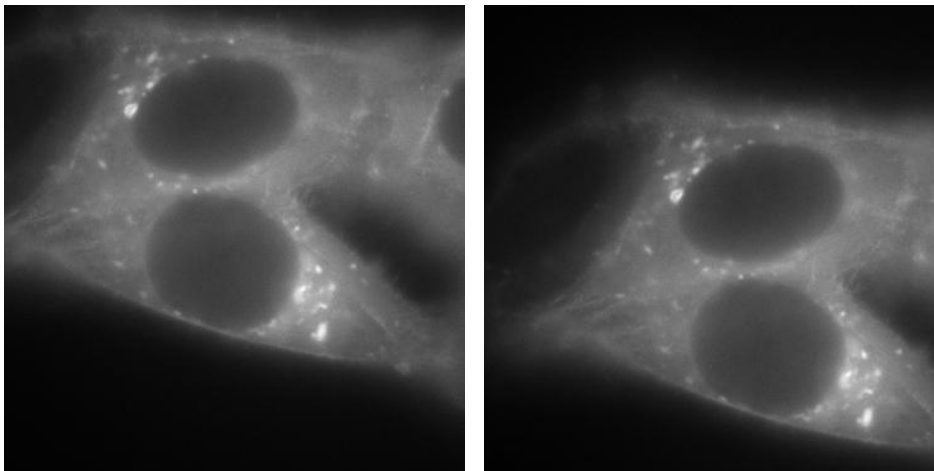
A common way to detect motion of objects over successive frames in a movie is to perform a correlation of successive images with one another. When there is concerted motion between two image frames, the cross-correlation of the two images will exhibit a peak at a non-zero location, and the offset of the peak from zero will then give the distance of movement over the period of time.

Cross-correlation is most effective when there is concerted motion over large regions of the microscopy image. In this section, we first go through an example of registering an image using cross-correlation. We then cover some biological examples where cross-correlation may be useful in extracting dynamical information in the system of interest.

First, let's load the image

```
>> im1 = imread('cells1.tif');  
>> im2 = imread('cells2.tif');
```

The images are out of register:



Create a superimposed color image of the two images to simultaneously visualize both images. The command **cat** is used to concatenate matrices in the third dimension and can be used to create RGB color images:

```
>> blank = zeros(size(im1));  
>> cim = cat(3, mat2gray(im1), mat2gray(im2), blank);  
>> imshow(cim, []);
```

As you can see, the two images are translated with respect to each other. To determine how much one image is translated with respect to the other, you can perform a cross-correlation on the two images. MATLAB has a command for performing a normalized cross-correlation:


```
>> b = normxcorr2(im1,im2);
```

Now visualize the cross-correlation function using the **mesh** command:

```
>> figure; mesh(b);
```

The output of this cross-correlation function is a matrix **b** that is the sum of the size of the two input images. **b** has a single maximum, which is offset from the center of the matrix by a small amount. This offset corresponds to the translation of **im1** with respect to **im2**. To determine the location of this peak, we use the **find** command, which returns the locations of nonzero indices in a given matrix. We can couple this command to conditional statement to find the maximum point of this matrix:

```
>> [y,x] = find(b == max(b(:)));
```

Note that, in matrix subscript notation, rows (**y**) are addressed before columns (**x**). Now we determine the offset of these positions from the origin of the matrix:

```
>> [yc,xc] = size(im2);  
>> yoff = y - yc;  
>> xoff = x - xc;
```

Now let's create a large color image to visualize the superposition of the two frames:

```
>> cim2 = zeros( yc+yoff, xc+xoff, 3);  
>> cim2(yoff:(yc+yoff-1), xoff:(xc+xoff-1), 1) = mat2gray(im1);  
>> cim2(1:yc, 1:xc, 2) = mat2gray(im2);  
>> imshow(cim2);
```

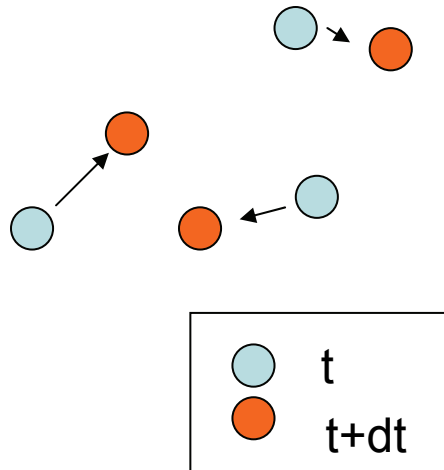
EXERCISES

1. Perform cross-correlation between successive frames of the spindle image **fluxingspindle.stk**. Cross-correlate the first image (2) with images that are further apart in time. What does the cross-correlation plot look like? What does this tell us about the biology?
2. A common way of extracting flows in fluid dynamics is to perform cross-correlation on subregions of a movie to obtain a flow vector field. Write a program that calculates flow vector fields by performing cross-correlation in a localized region near the vector to be calculated. Treat the size the neighborhood as a free parameter and test the program on **fluxingspindle.stk**

Particle Tracking

In particle-tracking, we attempt to extract the trajectories of moving point-like objects from a stack of images. There are two main steps:

- 1) **Segmentation.** We first need to identify the particles in all of the images. The segmentation techniques taught in the previous sections can be used to segment these particles and extract their locations in the microscopy images.
- 2) **Motion correspondence.** Given the locations of segmented particles in successive microscopy images, we need to know which particles in one frame correspond to which particles in a frame taken a small time-interval later. We essentially need to match particles at given time t with particles at a time $t+dt$ later, and do this for all image frames:



It is often the case in image acquisition that the time interval between successive frames is so small that these moving particles move only small distances during this time interval. In such a case, it is often adequate to match particles with the closest ones nearby. In such a case, we find a matching that minimizes a cost function determined by the distance traveled by particles. A cost function that works well for diffusive motion is given by the sum of squares of the distances:

$$C = \sum d_i^2$$

Here the summation is over all matched particles and the cost function is minimized over all possible matching pairs. In practice it is computationally too expensive to go through all possible matching pairs to find the minimum. One typically imposes a threshold distance beyond which no pair of particles in successive frames can be matched.

A particle tracker in MATLAB that solves the motion correspondence problem for the above cost function can be found at:

<http://www.deas.harvard.edu/projects/weitzlab/matlab/tutorial.html>

There are other problems and difficulties that may arise when solving the motion correspondence problem. For instance, particles can appear or disappear from the field of view during the course of the movie. Moreover, particles may become temporarily occluded from the field of view. It is important to modify the motion correspondence algorithms to address such potential problems.

Given a set of particles in a stack of segmented images and a correspondence between particles in successive images, it is possible to obtain a set of trajectories that describe the motion of these individual particles.

EXERCISES

1. **Kinetochores.** In a spindle, microtubules attach to chromosomes through structures called kinetochores. In the stack **kin.stk**, kinetochores are labeled with a fluorescent marker. Write a particle tracking algorithm to track the movement of individual kinetochores. Is there anything you can say about their dynamics?

Acknowledgements

Thanks to Mary Badon, Onn Brandman, Aaron Groen, Dan Needleman, Jade Shi and Puck Ohi for images.

8) REFERENCES

We relied on the following books to put together parts of the tutorial:

Gonzalez R.C., Woods R.E. (2001) Digital Image Processing. 2nd edition, Upper Saddle River, Prentice Hall

Gonzalez R.C., Woods R.E., Eddins S.L. (2004) Digital Image Processing using MATLAB, Upper Saddle River, Prentice Hall

Check out the following website for lots of image processing examples:

<http://www.imageprocessingplace.com/>

Some of the examples were pulled of from:

http://www.imageprocessingplace.com/DIPUM/dipum_tutorials/tutorials.htm

Useful particle tracking references:

Vallotton P., Ponti A., Waterman-Storer, C.M., Salmon E.D and Danuser G. (2003) Recovery, Visualization, and Analysis of Actin and Tubulin Polymer Flow in Live Cells: A Fluorescent Speckle Microscopy Study. *Biophys J.* 85:1289-1306

Veenman, C. J., M. J. T. Reinders, and E. Backer. 2001. Resolving motion correspondence for densely moving points. *IEEE Trans. Pattern Analysis and Machine Intelligence.* 23:54–72.