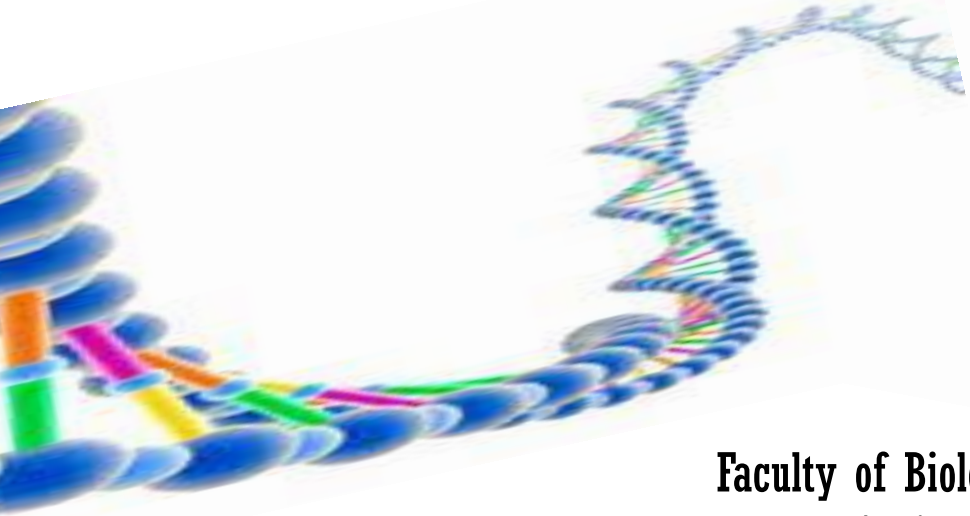
Computational Approaches for Life Scientists



Faculty of Biology, Technion
Spring 2017

Lecturer: **Dr. Amir Rubinstein**
Teaching Assistant: **Gur Hevroni**
Co-designer and advisor: **Prof. Benny Chor**

Biological Images –
Morphological operators – erosion and dilation

Outline

Introduction

- Representation of digital images in the computer
- Synthetic images
- Simple image manipulations

- **Binary segmentation**
  - Thresholding, Otsu segmentation

- **Labeling**

- **Morphological operators**
  - Erosion and dilation
  - Noise reduction (denoising) – maybe in the future

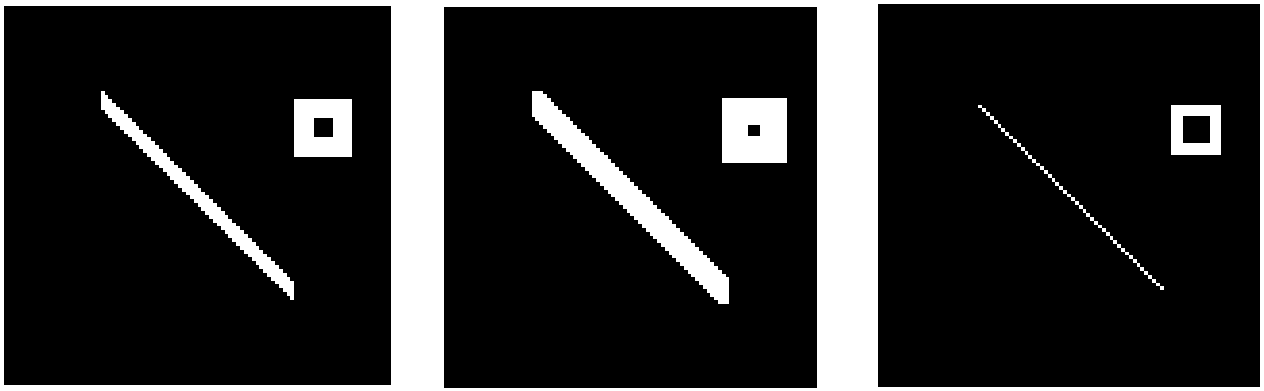
Last time

Erosion and Dilation

Assume features in the foreground are bright and background is dark.

- Erosion - the removal of pixels from the periphery of features.
  - shrinks foreground areas, and holes grow.

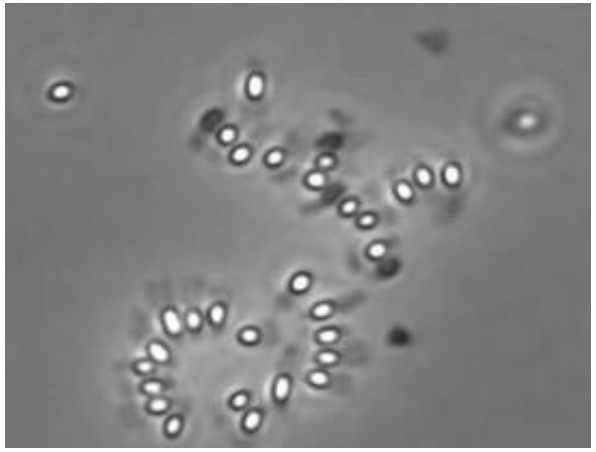
- Dilation - the addition of pixels to the periphery of features.
  - enlarges foreground areas, and holes shrink.



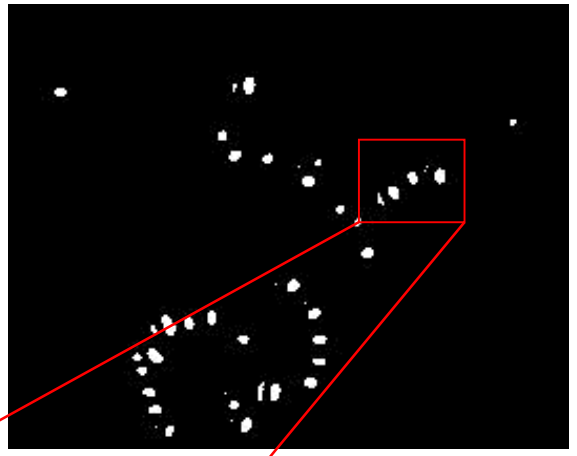
Original Dilation Erosion

- Like segmentation, these operators are often used to pre-process or post-process images to facilitate analysis (as we will see later).

Erosion and Dilation - Example

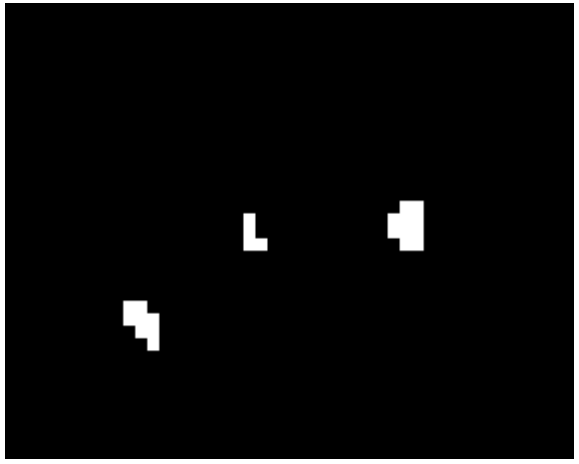


Binary segmentation
(th=200)

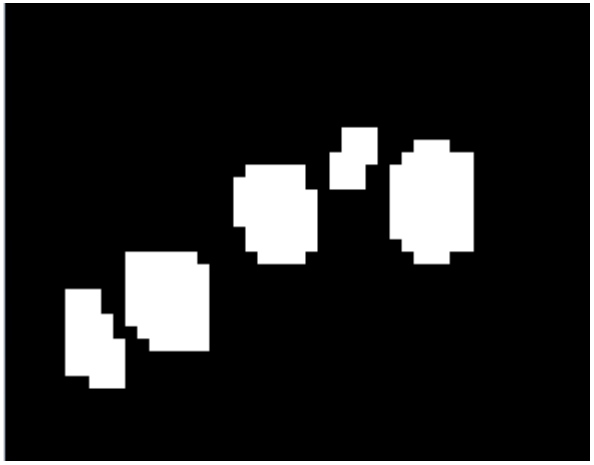


A microscope slide containing Clostridium botulinum cells and spores. Spores appear bright with dark boundaries (the spore coat). Vegetative cells were stained to provide contrast, and thus appear dark

Source: Martin, M.D., *Phase contrast image of germinating spores of a non-pathogenic clostridia that grows at low temperatures*. 2013.



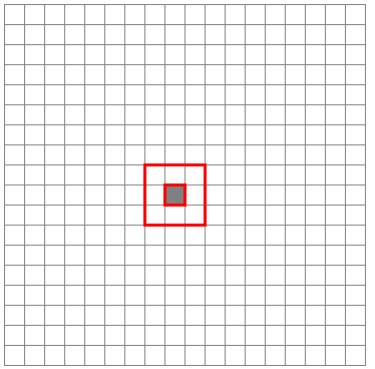
Erosion



Dilation

Erosion / Dilation as Morphological Operators

- In the context of image processing, a morphological operator is a manipulation of pixels related to the shape (morphology) of features in an image.
  Erosion and dilation are two basic such operators.



- For each pixel, we will look at its neighboring pixels.
  For example, a 3X3 square around the pixel.

  - For binary (black and white) images:
    - Dilation: pixels with at least one white neighbor will turn white
    - Erosion: pixels with at least one black neighbor will turn black

  - More generally, for 256 gray level images:
    - Dilation: pixels get the maximal gray level of their neighbors
    - Erosion: pixels get the minimal gray level of their neighbors

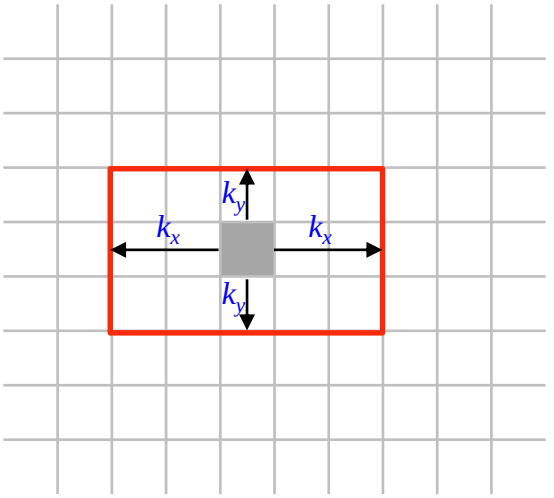
Neighborhood

- Neighborhood (or environment) of a pixel (x,y) is the set of all pixels close to it.
  - For example, a 3x3 square neighborhood:

$$N_{3x3}(x,y) = \begin{bmatrix} x-1, y-1 & x, y-1 & x+1, y-1 \\ x-1, y & x, y & x+1, y \\ x-1, y+1 & x, y+1 & x+1, y+1 \end{bmatrix}$$

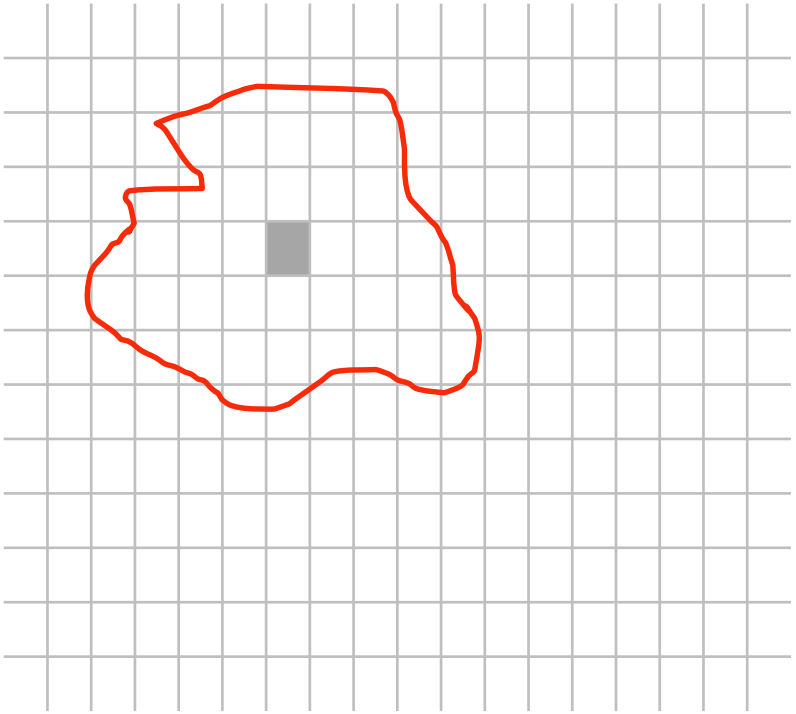
- More generally, a rectangular neighborhood of dimensions (k_x, k_y) is a $(2k_x+1)$-by-$(2k_y+1)$ rectangle.

(when $k_x = k_y = 1$ we get a 3x3 square)



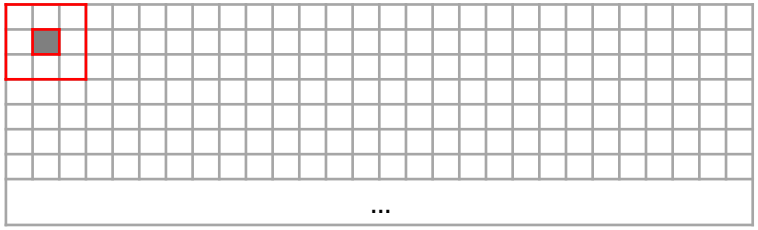
Neighborhood

- Even more generally, a neighborhood of a pixel can take any contiguous shape:



Abstract Morphological Operator

- Since erosion and dilation differ only in the function they invoke, a good approach would be to implement an abstract morph_operator function:



Abstract Morphological Operator

- Since erosion and dilation differ only in the function they invoke, a good approach would be to implement an abstract morph_operator function:

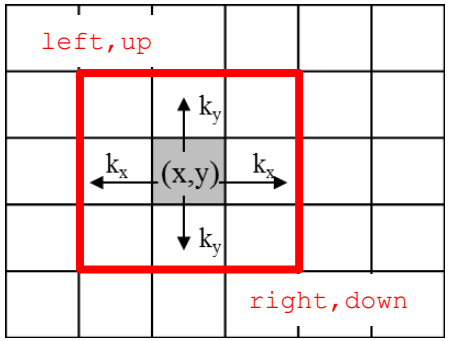
```
def erosion(im, kx=1, ky=1):
     return morph_operator(im, min, kx=1, ky=1)


def dilation(im, kx=1, ky=1):
     return morph_operator(im, max, kx=1, ky=1)




def morph_operator(im, op, kx, ky):
    ''' apply operator op on every pixel of im
        with a rectangular neighborhood of size (kx, ky) '''
   ...
```

A function!

Morphological Operator - Code

```
def morph_operator(im, op, kx=1, ky=1):
    ''' apply operator op on every pixel of im
        with a rectangular neighborhood of size (kx, ky) '''
    w, h = im.size
    out_im = im.copy()

    mat = im.load()
    out_mat = out_im.load()

    for x in range(w):
        for y in range(h):
            left  = max(0,x-kx)
            up    = max(0,y-ky)
            right = min(w-1, x+kx)
            down  = min(h-1, y+ky)

            neighbors_matrix = im.crop((left, up, right+1, down+1))
            neighbors_list = neighbors_matrix.getdata()
            out_mat[x,y] = op(neighbors_list)

    return out_im
```

Default: 3x3 square neighborhood



flatten into a 1D list
(pixel location is lost. Some operators take pixels location into consideration)

The operator is applied on the neighboring pixels

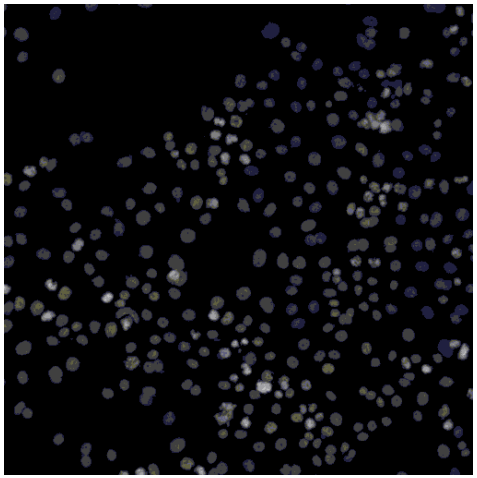
Using Erosion for Simple Edge Detection

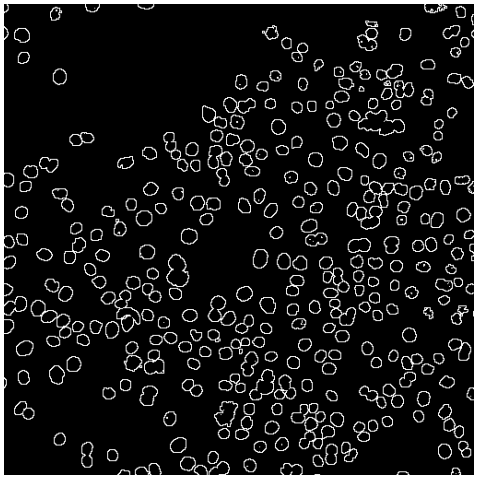
- Normally, an image is piecewise-smooth:
  - changes between neighboring pixels are moderate.
  - in some areas of the image, pixels do change extremely over small regions. Such areas are termed boundaries, or edges.

- Applying an edge detection algorithm to an image may significantly reduce the amount of data to be processed and may therefore filter out information that may be regarded as less relevant, while preserving the important structural properties of an image.

- In biological images, edges may be the boundaries of cells, the nucleus, various organelles inside a cell, etc.

- The visual neural system in our brain performs edge detection at the early stage of analyzing images.

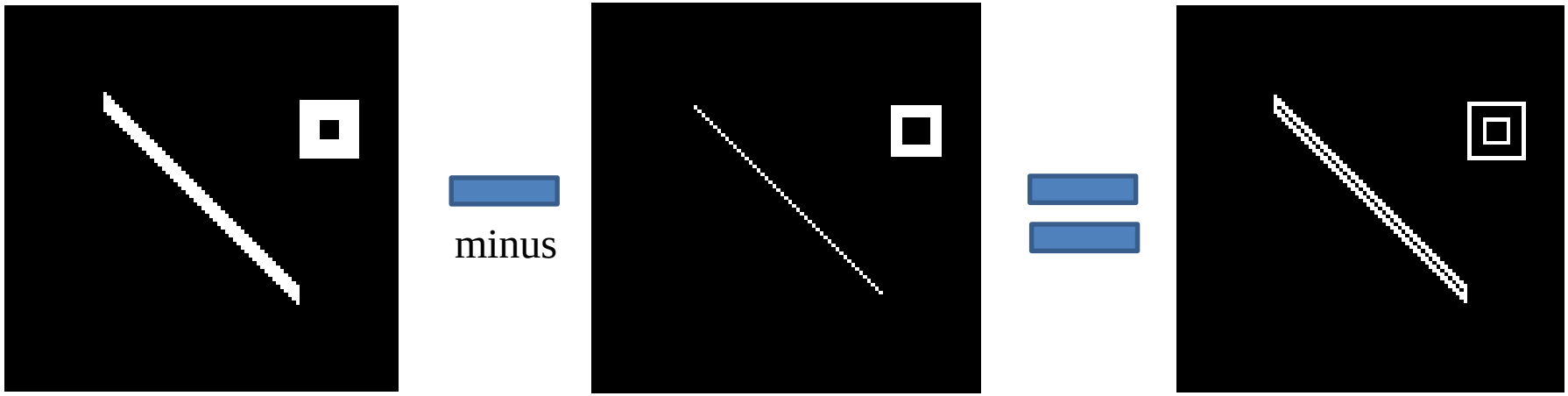




Using Erosion for Simple Edge Detection

- A trivial method for edge detection uses erosion (or dilation).

- This simple method is appropriate mostly for binary (black and white) images. In other cases, more sophisticated methods are needed.



Original image | Eroded image | Edges

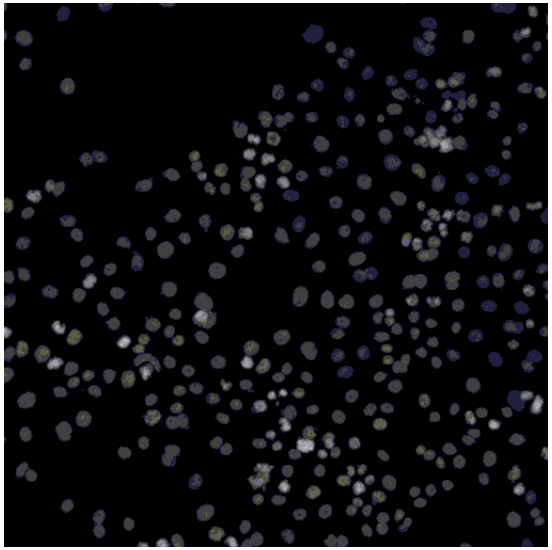
```
def edges(im):
    return minus(im, erosion(im))
```

Pixel-wise Minus

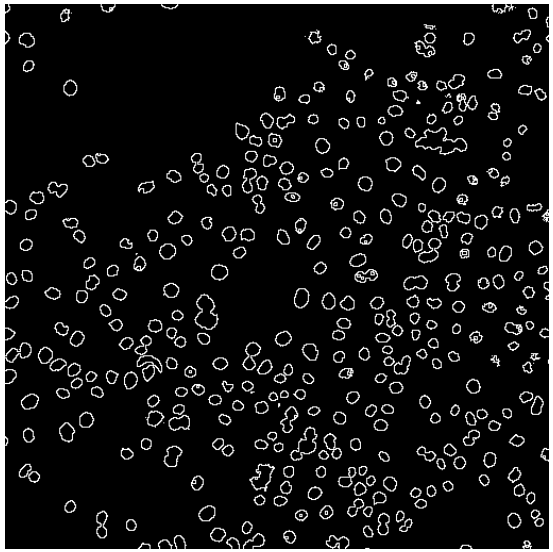
```
def minus(im1, im2):
    ''' creates the diff im1-im2 pixel-wise'''
    assert im1.size == im2.size

    w, h = im1.size
    mat1 = im1.load()
    mat2 = im2.load()
    out = im1.copy()
    out_mat = out.load()

    for x in range(w):
        for y in range(h):
            out_mat[x,y] = mat1[x,y] - mat2[x,y]

    return out
```

Edge Detection – Some Results

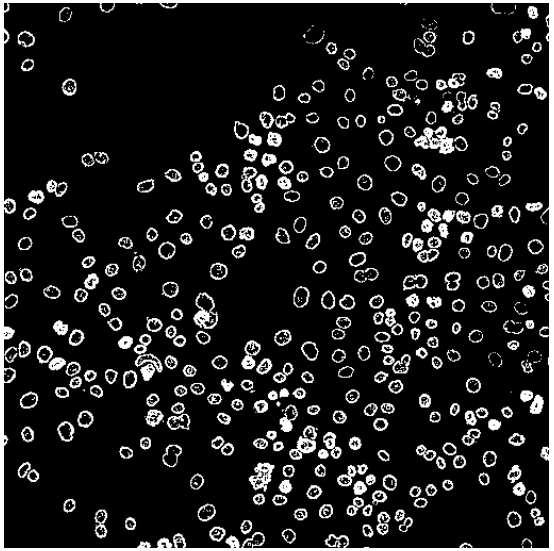
```
im = Image.open("./HT29.tif").convert('L')
im.show()
```



```
#segmentation → edge detection
th = otsu_thrd(im) #th is 38
seg = segment(im, th)
edges(seg).show()
```



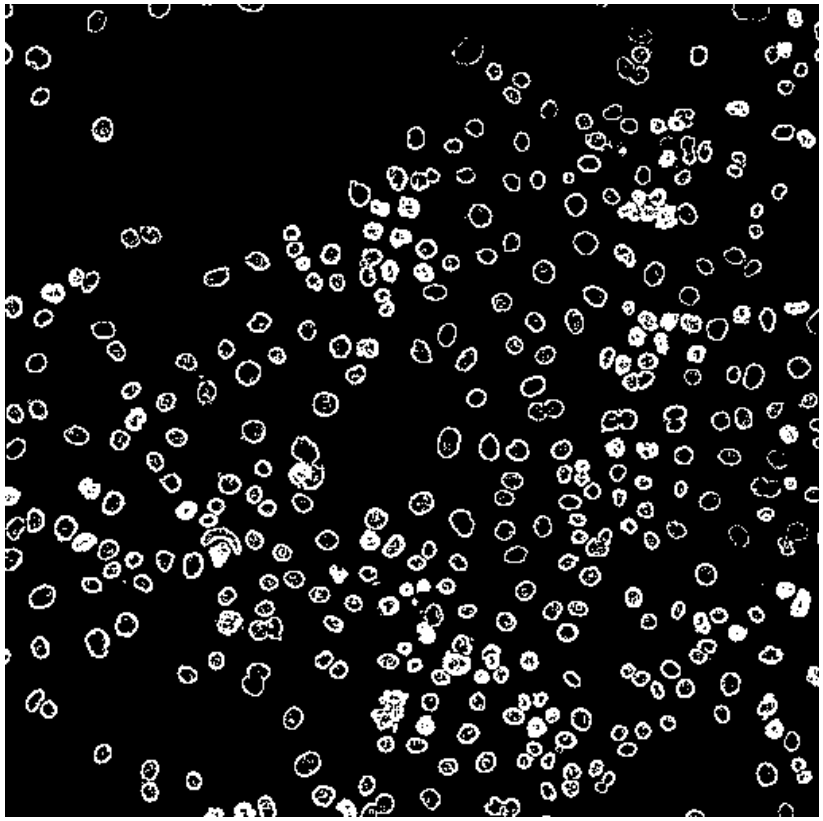
```
#edge detection → segmentation
im2 = edges(im)
th = otsu_thrd(im2)
seg = segment(im2, th)
seg.show()
```



PIL's Edge Detection

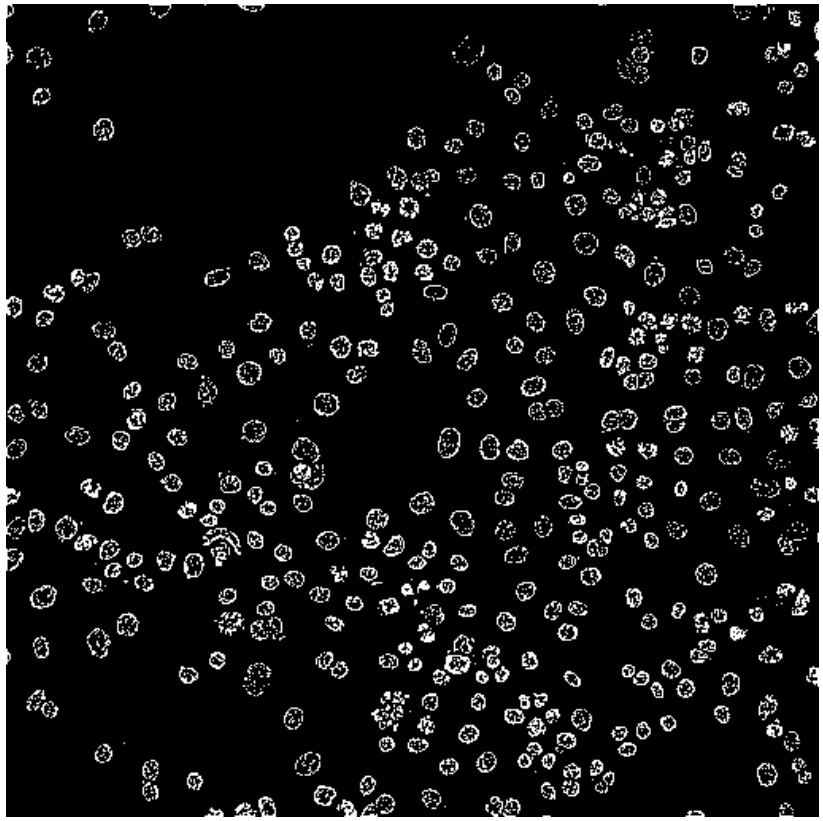
```
im2 = edges(im)
th = otsu_thrd(im2)
seg = segment(im2, th)
seg.show()
```

(from previous slide)



Vs.

```
from PIL import ImageFilter
im3 = im.filter(ImageFilter.FIND_EDGES)
th = otsu_thrd(im3)
seg = segment(im3, th)
seg.show()
```



- Which is better? You be the judges…

Reflection

- Obviously, some well-known existing tools, such as Photoshop and others, enable many image processing tasks to be performed without writing a single line of code.
- However, this course aims at exposing you to the interiors of the digital image world. In addition, the commercial tools and packages may lack the specific functionality that one may need to solve a specific problem.

- We used Python's Pillow imaging package, as well as the Scipy package (for labeling). These packages contain many more functionalities than we exposed here, and there are other packages as well.
- Fortunately, these all have a rich and informative documentation, as well as supporting forums, and thus are relatively easy to learn and use.

Reflection (2)

- Today, some image processing tasks still require human intervention, and thus are not conducted fully automatically.

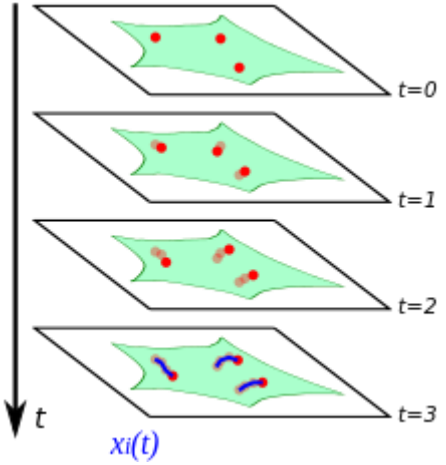
- Worth mentioning in this context autonomous vehicles

- In Biology, a major difficulty originates from the high diversity in imaging technology. For example, fluorescence microscopy produces images very different in nature from MRI images.

- Another reason is the high diversity in image content: an edge detection algorithm may work well on elongated shapes, but less so on round or elliptic ones, etc.

Reflection (3)

- Image processing goes way beyond the fundamental operations introduced in this chapter.
- One example is the problem of tracking: locating moving objects in consecutive images taken over time (*e.g.* video frames).
- Especially difficult when the objects are moving fast relative to the frame rate, when the tracked objects change orientation or shape over time, disappear from the frame, or divide into several objects (e.g. cell division).



Appendix: Tiling Images

```
def join(*images):
    ''' Join several images horizontally for easy display.
        Assume all images are of the same size
        The * before the parameter means a variable number of parameters '''
    w,h = images[0].size
    n = len(images) #number of images
    new = Image.new('L',(w*n+n,h), 'white') #+n for some space between images

    for i in range(len(images)):
        new.paste(images[i], (w*i+i,0)) #+i for some space between images
    return new
```

- Since this function may be useful in other scenarios beyond denoising, we will put it in a separate file util.py, and import it from where it is needed:

```
from util import tile
```

or

```
from util import *
```

(we can also simply write `import util`, but then its usage is `util.tile`)