Extended Introduction to Computer Science CS1001.py

Lecture 24-25: Introduction to Digital Image Processing

Instructors: Elhanan Borenstein, Amir Rubinstein

Teaching Assistants: Michal Kleinbort, Noam Parzanchevski, Ori Zviran

School of Computer Science
Tel-Aviv University
Spring Semester 2019-20
http://tau-cs1001-py.wikidot.com

Lectures 23-24: Highlights

- Error detection and correction
  - Codes and codewords
  - Hamming distance between words
  - Hamming distance of a code
  - Examples:
    - Israeli control digit in ID numbers
    - “Card magic” (2D parity bit)
    - Repetition code
    - Parity bit code
    - Hamming (7,4,3) code

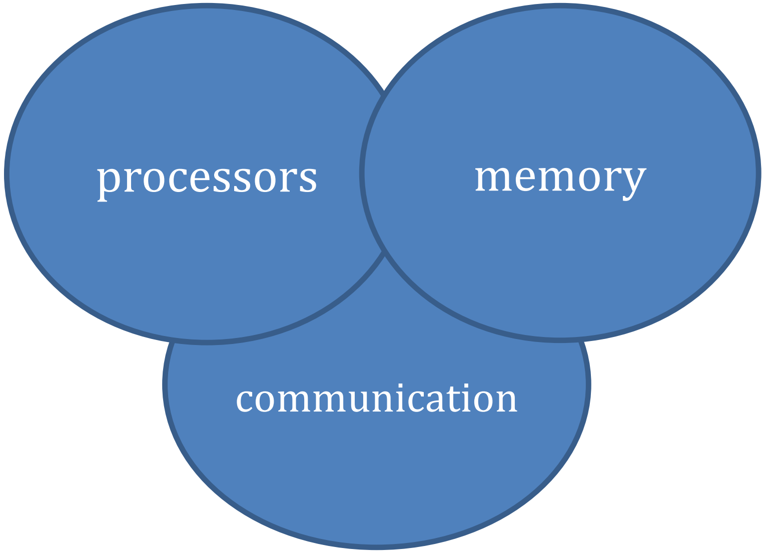
Lecture 24-25: Plan

- Introduction to Digital Image Representation.
  - Grayscale and color images
  - Bit depth, resolution
  - Class Matrix
  - Generating synthetic images

- Basics of Digital Image Processing
  - Noise models: Gaussian, Salt & Pepper
  - Noise reduction: local means, local medians

Brief "Historical" Technological Context

Number of transistors	speed
29 K	4.77 MHz
1.4 G	3.7 GHz



RAM	Hard Disk
640 KB	5 MB
4 GB	500 GB

- early 1980's
- today

e-mail, simple text (128 ascii chars)

tons of data, inc. images (next slide)

A Brief Historical Context, 30 Years Later

- With the proliferation of

  (1) larger and faster memory,

  (2) strong, inexpensive processors,

  (3) faster internet,

  it became possible to efficiently

  (1) store,

  (2) process, and

  (3) transmit large digital images.

- Facebook stores about 350 million photos DAILY (reported Sep 2013).

  1.1 billion photos where uploaded on 2013 New Years Eve.

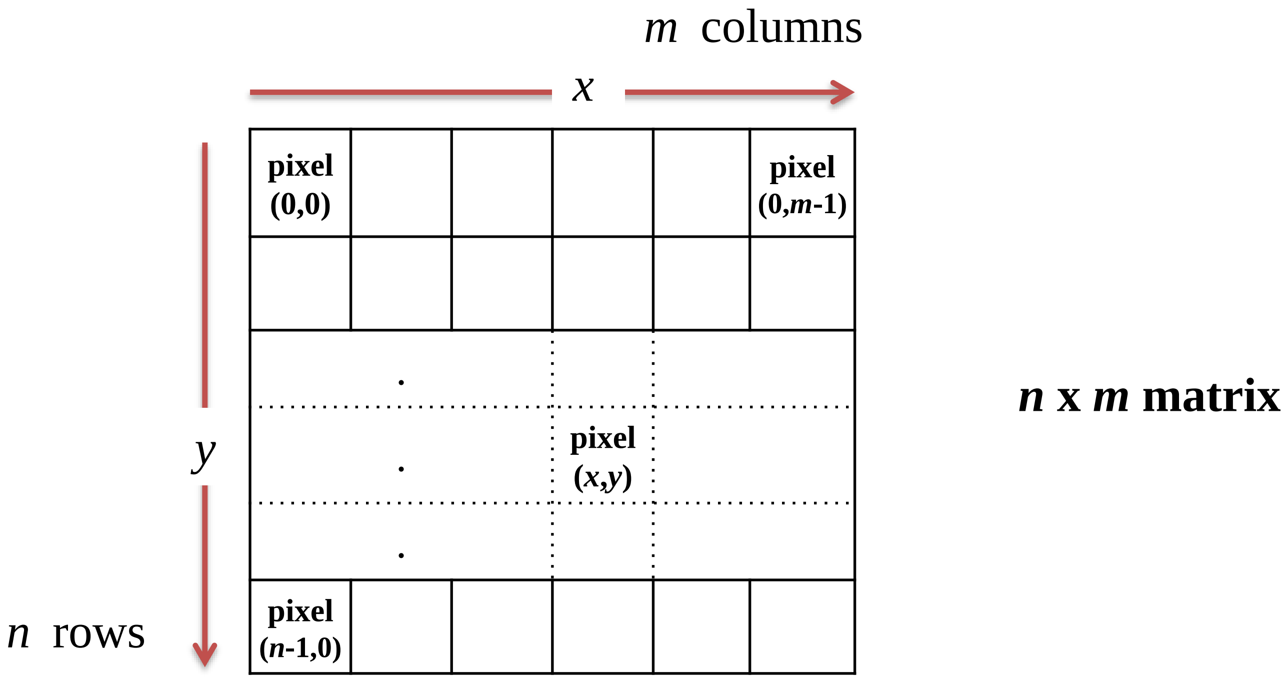
- The total number of photos shared on Instagram is 16 billion. On average, 55 million photos are posted daily (reported Dec 2013). (Instagram was launched on Oct 2010 !!).

- On Flickr the average upload of images per MONTH in 2012 was about 43 million.

- This dramatic technological progress is reflected by the following saying, often attributed (apparently incorrectly) to Bill Gates, in 1981: "640KB ought to be enough for anybody".

Basic Model of a Digital Image

- A digital image is typically encoded as a *n*-by-*m* rectangle, or matrix, *M*, of either grey-level or color values.
- Each element $M[x, y]$ of the image is called a pixel, shorthand for picture element.



m columns

x

y

n rows

pixel (0,0)

pixel (0,*m*-1)

pixel (*x*,*y*)

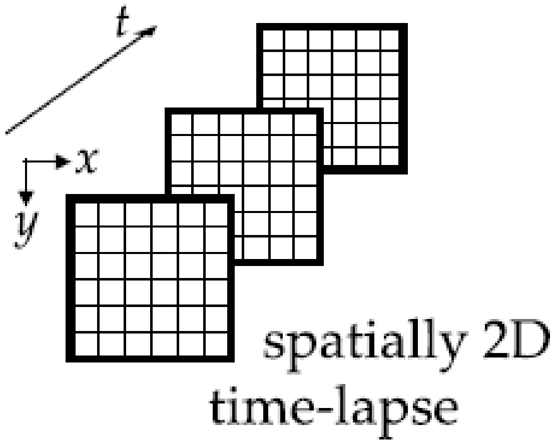
pixel (*n*-1,0)

***n* x *m* matrix**

Video

- A 2D image is encoded as a *n*-by-*m* matrix *M*

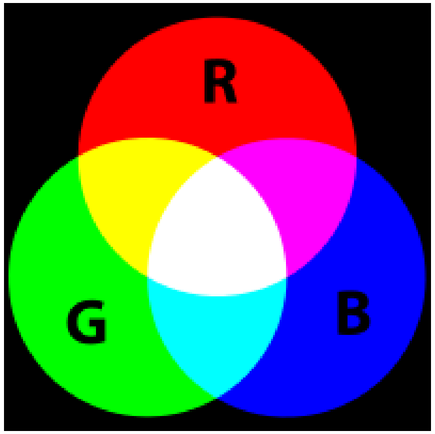
- For videos (movies), there is a third dimension, "time". For each point *t* sampled in time, the frame at time *t* is nothing but a "regular" image.

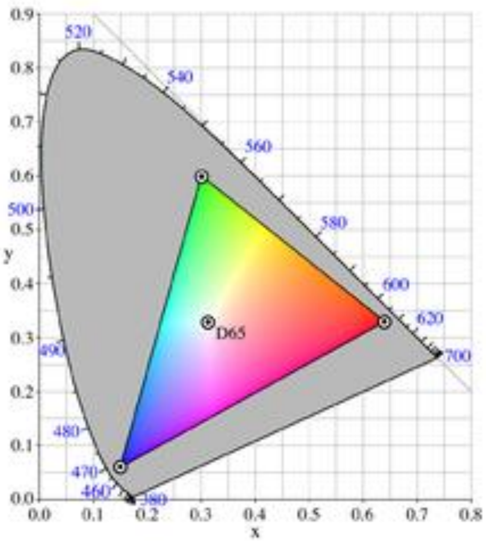


RGB format

- For grey level images, $M[x, y]$ is a non negative number, representing the light intensity at the pixel.
-
- For standard (RGB) color images, $M[x, y]$ is a triplet of values, representing the red, green, and blue components of the light intensity at the pixel.

(images from Wikipedia)





Some fun with color representation

- https://csfieldguide.org.nz/en/chapters/data-representation/images-and-colours/

Grey Level format

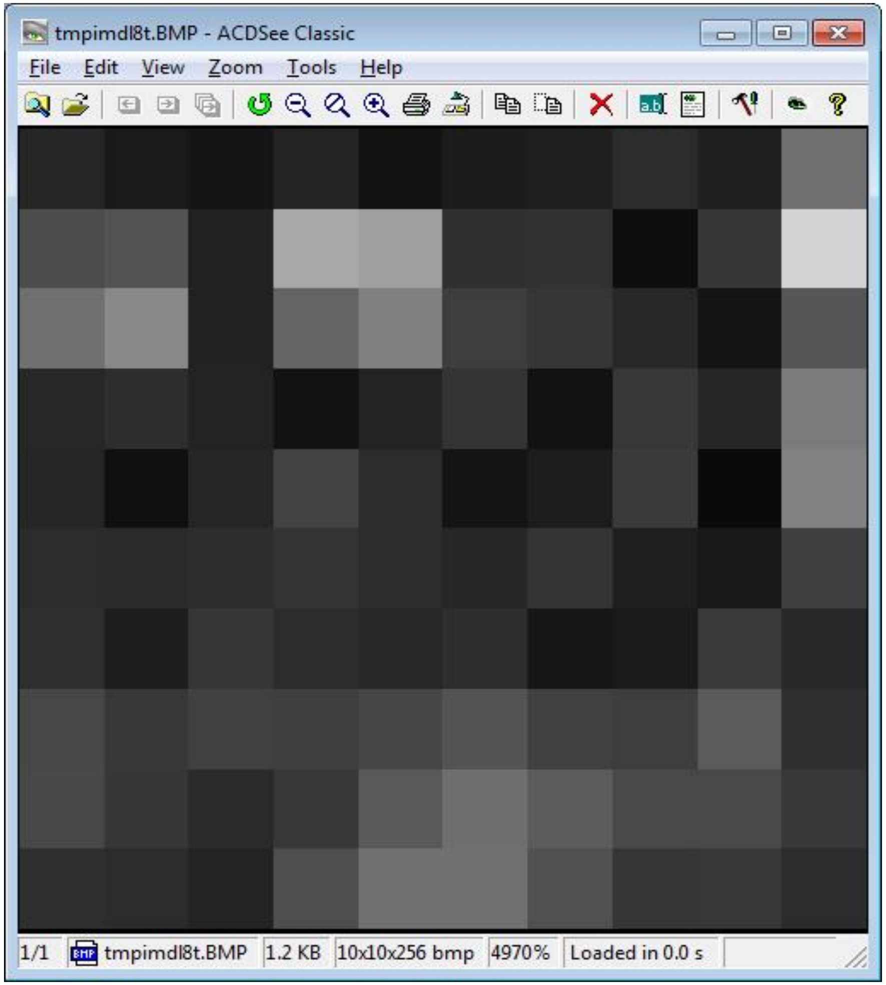
- For the sake of simplicity, the remainder of this presentation will deal with greyscale images only. However, what we will do is applicable to color images as well.
- Real numbers expressing grey levels (or colors) have to be discretized in order to enable their representation on bounded precision digital devices.

- A good quality greyscale photograph (that is, good by human visual inspection) has 256 grey-level values (8 bits) per pixel.
- The value 0 represents black, while 255 represents white. For each pixel, the closer its value is to 0, the blacker it is. So 128 is considered a "perfect" grey.

- We remark that in some applications, such as medical imaging, 4096 grey levels (12 bits) are used.

Grey Level format - example

- 8 bits per pixel (2^8=256 gray levels): 0 = black, 255 = white


tmpimdl8t.BMP - ACDSee Classic

1/1 | tmpimdl8t.BMP | 1.2 KB | 10x10x256 bmp | 4970% | Loaded in 0.0 s

```
38,   26,   21,  36,   19,   28,   33,  44,  31,  112,
77,   83,   34,  168,  159,  48,   50,  14,  55,  211,
112,  137,  34,  101,  129,  62,   54,  40,  21,  86,
41,   46,   35,  19,   35,   52,   18,  57,  39,  123,
38,   16,   38,  67,   45,   21,   29,  59,  10,  130,
45,   43,   46,  51,   44,   39,   53,  31,  24,  64,
47,   30,   54,  45,   40,   46,   23,  26,  58,  40,
71,   57,   66,  63,   70,   84,   65,  62,  91,  49,
72,   55,   43,  57,   90,   111,  92,  73,  74,  56,
47,   45,   36,  78,   114,  113,  81,  54,  57,  44
```

Bit Depth

- Number of bits per pixel.

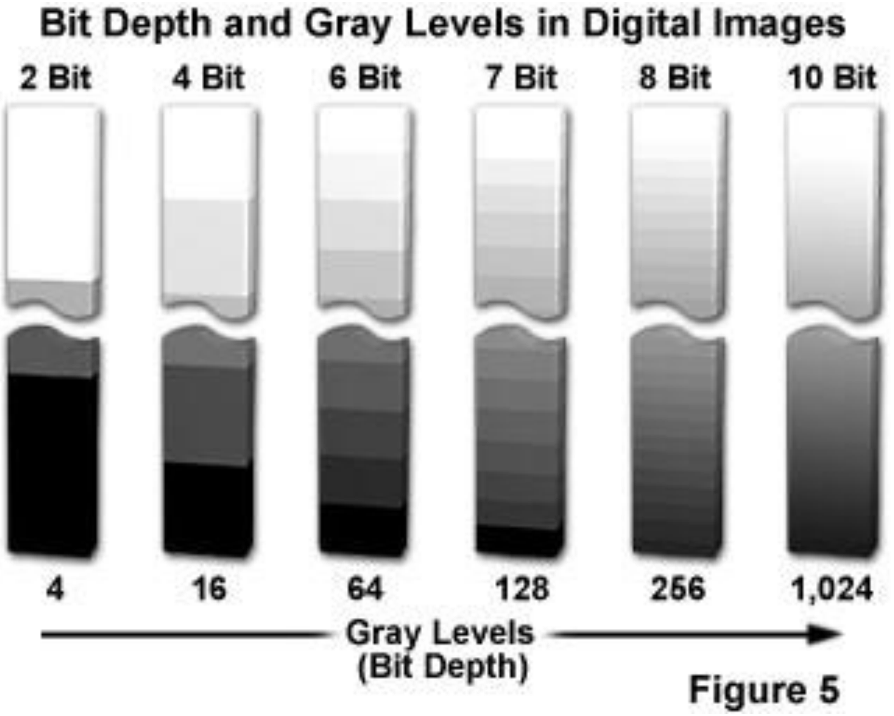
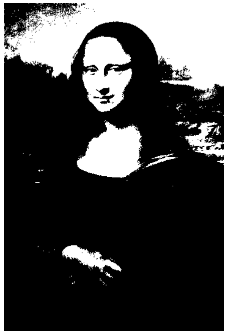


Image from:
http://micro.magnet.fsu.edu/

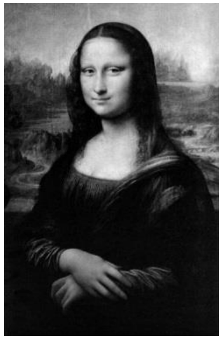
- A human observer is able to discriminate between at most a few hundreds shades of gray in optimal conditions (some estimations are lower, depending also on the background, distance from the image etc.).
- Higher bit depths images are sometimes aimed for an automated analysis by a computer.

BW / Grayscale / RGB - summary

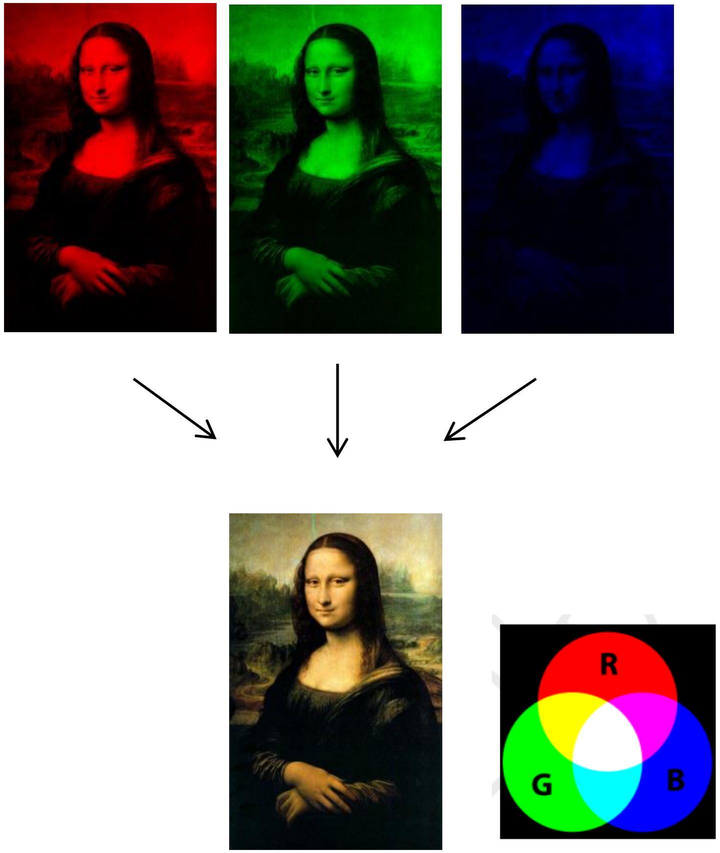
- B&W / gray-level / RGB



B&W
(1 bpp)



256 gray level image
(8 bpp)



"true color" image
(8+8+8 = 24 bpp)

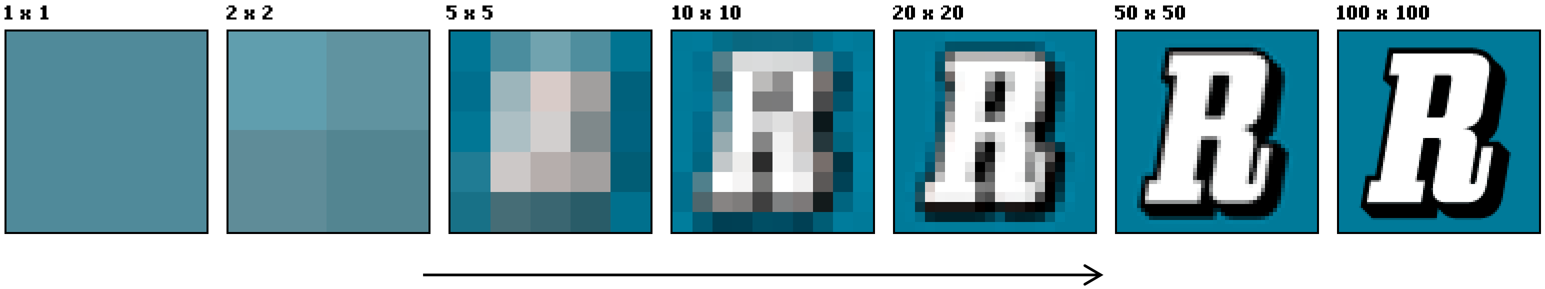
Images from: http://www.csse.uwa.edu.au/~wongt/matlab.html

Resolution and Pixel Physical Size

- Resolution is the capability of the sensor to observe or measure the smallest object clearly with distinct boundaries.

- Resolution depends upon the physical size of a pixel.

  Higher resolution = lower pixel size.



Increasing resolution

Source: wikipedia

Some Technicalities

- class Matrix

- working with "real" images using the external package PILLOW

The Class Matrix

- Please welcome our home-made class Matrix.

- We will briefly go over its main functionalities

- It is implemented as a list of lists:

```
class Matrix:
    def __init__(self, n, m, val=0):
        assert n>0 and m>0
        self.rows = [[val]*m for i in range(n)]
```

The Class Matrix (2)

```
class Matrix:
      …

    def dim(self):
        return len(self.rows), len(self.rows[0])


    def __repr__(self):
        if len(self.rows)>10 or len(self.rows[0])>10:
            return "Matrix too large, specify submatrix"
        return "<Matrix {}>".format(self.rows)


    def __eq__(self, other):
        return isinstance(other, Matrix) and \
               self.rows == other.rows
```

Calls __eq__ of class list

The Class Matrix (3)

- Additional methods:
  - **copy**

  - **Arithmetical** operations, e.g. `mat1 + mat2`

  - **__getitem__** : receives a tuple (i,j)
  - **__setitem__** : receives a tuple (i,j) and val

    i and j can be both integers or both slices

  - **display**: shows the image represented by a matrix, uses the Python standard (no installation needed) package tkinter

  - **save** and **load**: enable storing and reading images from files

class Matrix - item access and assignment

```
>>> mat = Matrix(10, 10)  # 10x10 matrix of zeros
>>> mat[4,5]              # same as m.__getitem__((4,5))
0
>>> mat[4,5] = 99         # same as m.__setitem__((4,5),99)
>>> mat[4,5]
99
```

Note: the code in the matrix.py file contains an additional feature: accessing and assignment of a whole slice.

```
>>> mat[3:5, 4:8]         # here i and j are both slices
<Matrix [[0, 0, 0, 0], [0, 99, 0, 0]] >
```

class Matrix – Indexing (read)

```
# cell/sub-matrix access
def __getitem__(self, ij):
    #ij is a tuple (i,j). Allows m[i,j] instead m[i][j]
    i,j = ij
    if isinstance(i, int) and isinstance(j, int):
        return self.rows[i][j]
    elif isinstance(i, slice) and isinstance(j, slice):
        M = Matrix(1,1) # to be overwritten
        M.rows = [row[j] for row in self.rows[i]]
        return M
    else:
        return NotImplemented
```

class Matrix – Indexing (write)

```
# cell/sub-matrix assignment
def __setitem__(self, ij, val):
    #ij is a tuple (i,j). Allows m[i,j] instead m[i][j]
    i,j = ij
    if isinstance(i, int) and isinstance(j, int):
        assert isinstance(val, (int, float, complex))
        self.rows[i][j] = val
    elif isinstance(i, slice) and isinstance(j, slice):
        assert isinstance(val, Matrix)
        n,m = val.dim()
        s_rows = self.rows[i]
        assert len(s_rows) == n and len(s_rows[0][j]) == m
        for s_row, v_row in zip(s_rows,val.rows):
            s_row[j] = v_row
    else:
        return NotImplemented
```

display

```
n = 500
m = 500
mat = Matrix(n,m)

for i in range(n):
    for j in range(m):
        mat[i,j] = random.randint(0,255)

>>> mat
Matrix too large, specify submatrix

>>> mat[3:5, 4:8]
<Matrix [[216, 213, 114, 208], [2, 4, 245, 149]]>

>>> mat.display()
>>> mat.display(zoom=2)
```

Note: You do NOT need to understand the display method

save to and load from a file

```
>>> mat.save("./rand_image.bitmap")
```

A new file rand_image.bitmap will be created.
This is merely a text file:

rand_image.bitmap

```
500 500
230 168 178 213 28 159 121 …
222 133 165 152 8 236 188  …
51 152 152 93 120 117 208  …
…
```

```
>>> mat2 = Matrix.load("./rand_image.bitmap")
```

Note: You do NOT need to understand the load and save methods

From "real" image formats to .bitmap

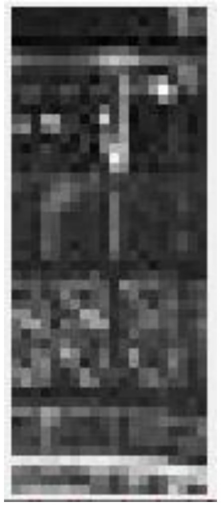
- We provide a way to work with "real" images in known formats such as jpg, bmp, tif etc.
- The file format_conversion.py contains the transformation in both directions.
- To have it work, you first need to install an external Python package called PILLOW – Python Imaging Library, see: https://pypi.python.org/pypi/Pillow

```
>>> image2bitmap("./my_image.jpg") #creates my_image.bitmap

>>> bitmap2image("./my_image.bitmap") #vice versa
```

Note: You do NOT need to understand the image2bitmap and bitmap2image methods

Guess what's in the image

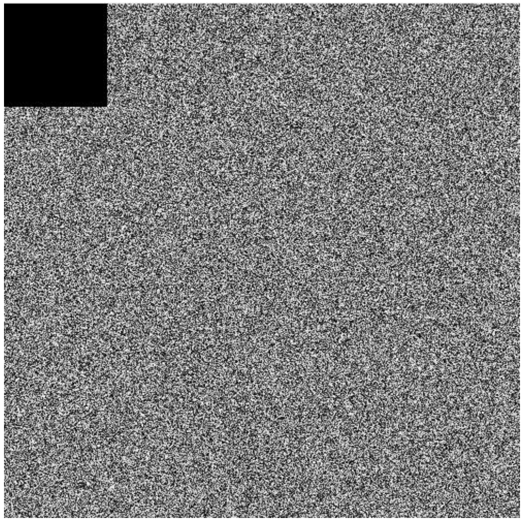
```
>>> image2bitmap("./guess.bmp")

>>> img = Matrix.load("./guess.bitmap")

>>> img.dim()
(541, 388)

>>> img[400:450, 200:220].display(zoom=4)
```



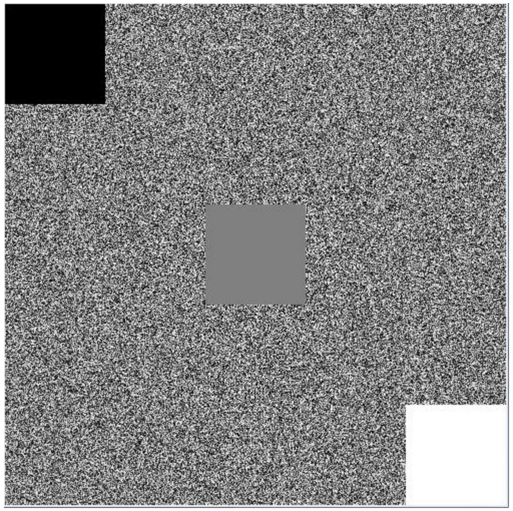
And now for some more interesting stuff

```
def black_square(mat):
    ''' add a black square at upper left corner '''
    n,m = mat.dim()
    if n<100 or m<100:
        return None
    else:
        new = mat.copy()
        for i in range(100):
            for j in range(100):
                new[i,j] = 0
    return new
```

```
mat = Matrix(500,500)

for i in range(500):
    for j in range(500):
        mat[i,j] = random.randint(0,255)

black_square(mat).display()
```



```
def three_squares(mat):
    ''' add a black square at upper left corner, grey at
    middle, and white at lower right corner'''
    n,m = mat.dim()
    if n<500 or m<500:
        return None
    else:
        new = mat.copy()
        for i in range (100):
            for j in range (100):
                new[i,j] = 0 # black square
        for i in range (200,300):
            for j in range (200,300):
                new[i,j] = 128 # grey square
        for i in range (400,500):
            for j in range (400,500):
               new[i,j ]= 255 # white square
    return new
```



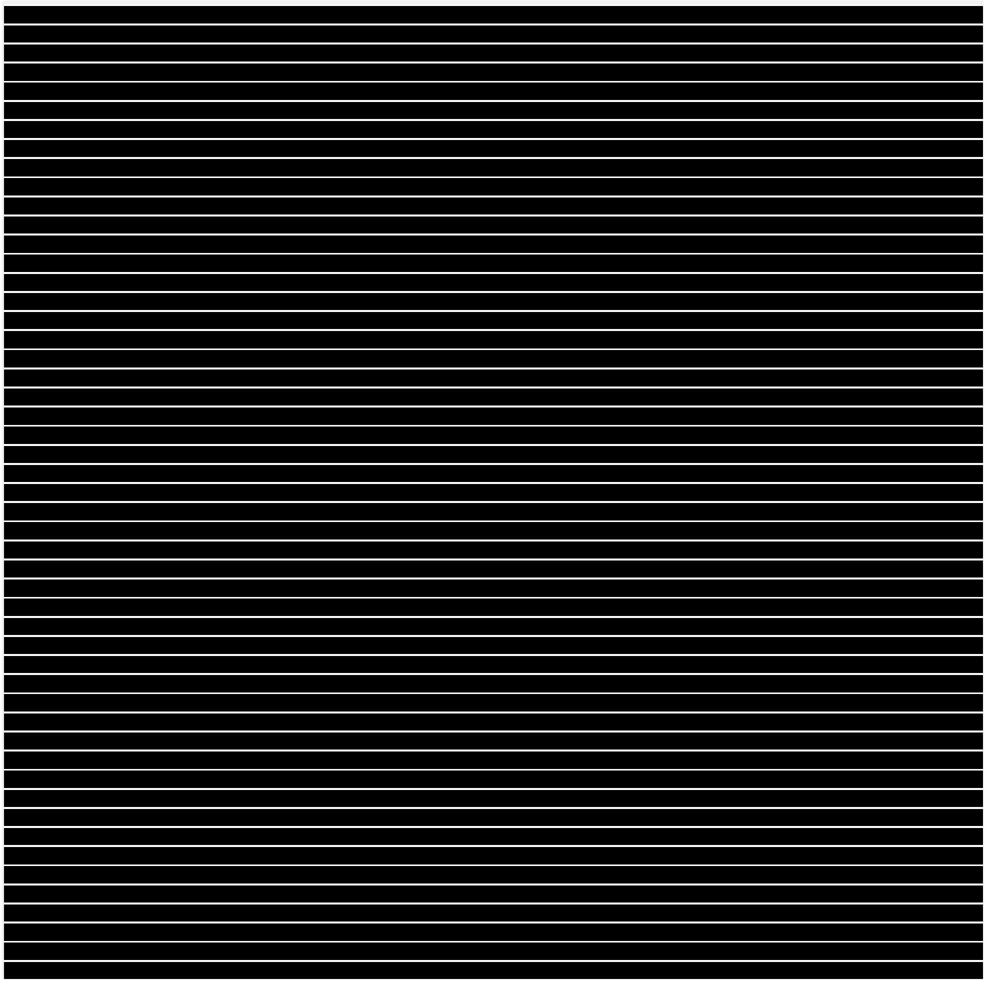
Simple Synthetic Images: Lines and More

```
def horizontal_lines(n, c):
    mat = Matrix(512,512)

    for i in range(n):
        if i%c == 0:
            for j in range(n):
                mat[i,j] = 255

    return mat
```

```
>>> img = horizontal_lines(256, 10)
>>> img.display(zoom=2)
```

Displaying Synthetic Images: Lines and More

```
>>> img = horizontal_lines(256, 10)
>>> img.display(zoom=2)
```



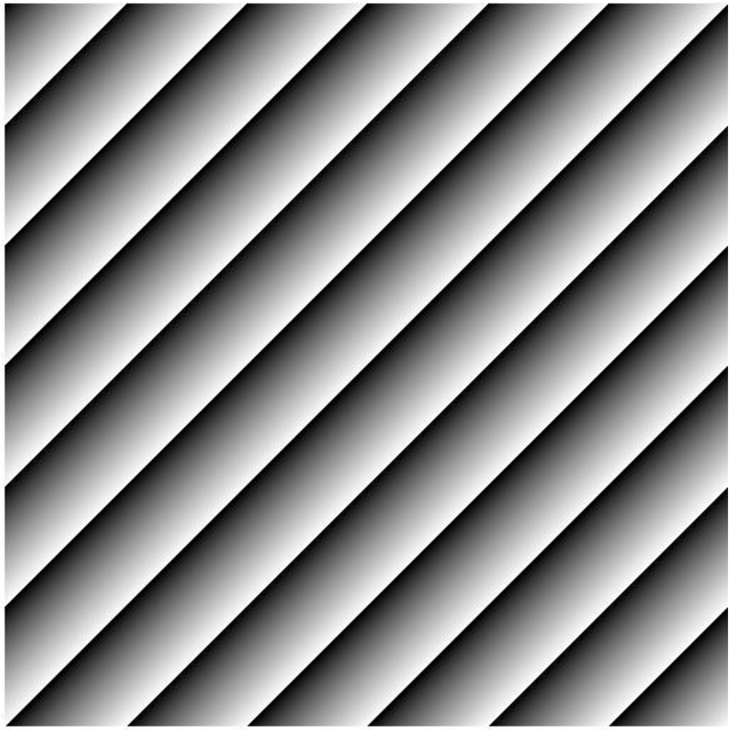
Simple Synthetic Images: Diagonal Lines

```
def diagonals(n, c=1):
    mat = Matrix(n,n)
    for i in range(n):
        for j in range(n):
            mat[i,j] = (c*(i+j)) % 256
    return mat
```

```
>>> img = diagonals(256)
>>> img.display()
```

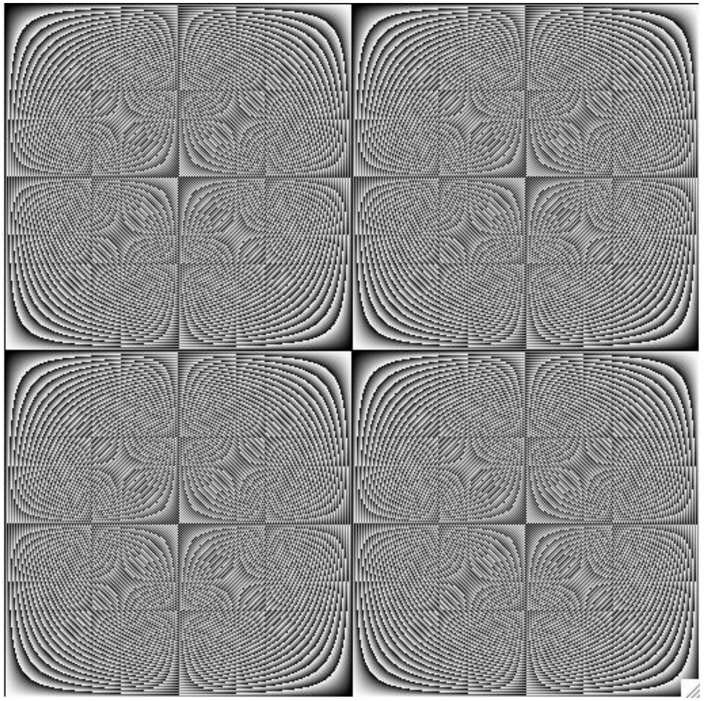
compare to:

```
>>> img = diagonals(256, c=3)
>>> img.display()
```



Simple Synthetic Images: Product and Circles

```
def product(n, c=1):
    surprise = Matrix(n,n)
    for i in range(n):
        for j in range(n):
            surprise[i,j] = (c*(i*j))% 256
    return surprise
```



```
>>> img = product(256)
>>> img.display()
```

compare to:

```
>>> img = product(256, c=2)
>>> img.display()
```

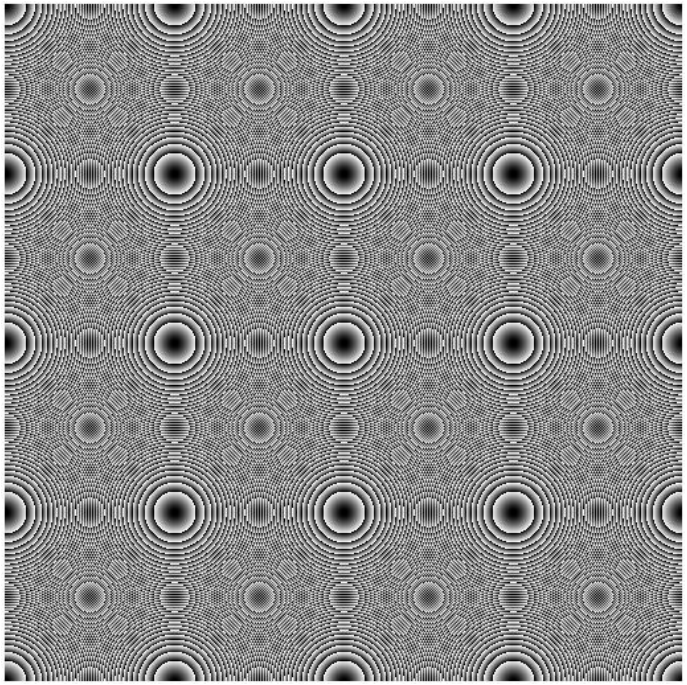
Simple Synthetic Images: Product and Circles

```
def circles(n, c=1):
    surprise = Matrix(n,n)
    for i in range(n):
        for j in range(n):
            surprise[i,j] = (c*(i**2+j**2)) % 256
    return surprise
```

```
>>> img = circles(256)
>>> img.display()
```

compare to:

```
>>> img = circles(256, c=2)
>>> img.display()
```



Simple Synthetic Images: high order function

```
def synthetic(n, m, func):
  """ produces a synthetic image "upon request" """
  mat = Matrix(n,m)
  for i in range(n):
      for j in range(m):
          mat[i,j] = func(i,j)%256
  return mat
```

Simple Synthetic Images: Miscellaneous

```
>>> a = synthetic(512, 512, lambda x,y: random.randint(0,255))
>>> a.display()
```

```
>>> b = synthetic(512, 512, lambda x,y: \
                       math.sin(16*(x**2 + y**2)))
>>> b.display()
```

```
>>> c = synthetic(512, 512, lambda x,y: \
        100*math.sin(32*cmath.phase(complex(x, y))))
>>> c.display ()
```

We urge you to try these (and other) functions by yourself.

Tiling images: join horizontally

```
def join_h(mat1, mat2):
    """ joins 2 mats, side by side with some separation """
    n1,m1 = mat1.dim()
    n2,m2 = mat2.dim()
    m = m1+m2+10 #+10 to separate between the images
    n = max(n1,n2)
    new = Matrix(n, m, val=255)  # fill new matrix white

    new[:n1,:m1] = mat1
    new[:n2,m1+10:m] = mat2

    return new
```

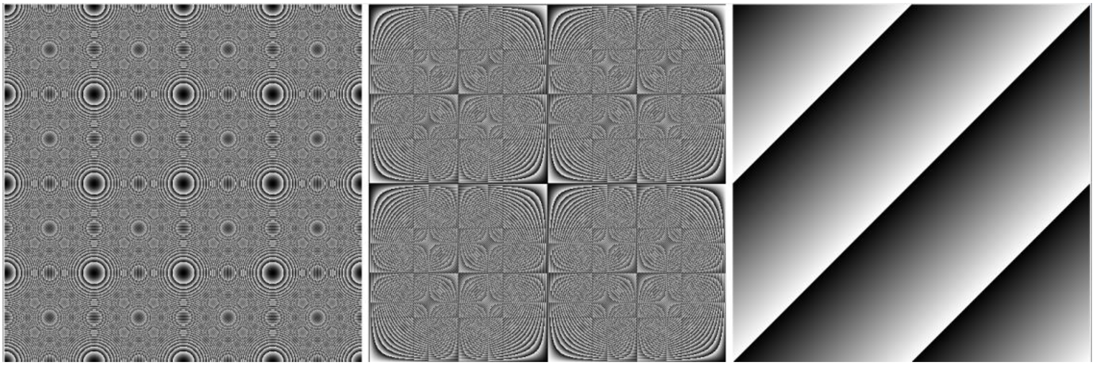
Tiling images: join vertically

```
def join_v(mat1, mat2):
    """ joins 2 mats, vertically with some separation """
    n1,m1 = mat1.dim()
    n2,m2 = mat2.dim()
    n = n1+n2+10 #+10 to separate between the images
    m = max(m1,m2)
    new = Matrix(n, m, val=255)  # fill new matrix white

    new[:n1,:m1] = mat1
    new[n1+10:n,:m2] = mat2

    return new
```

Tiling multiple images

```
def join(*mats, direction):
    ''' *mats enables a variable number of parameters.
        direction is either 'h' or 'v',
        for horizontal or vertical join, respectively '''

    func = join_v if direction == 'v' else join_h
    res = mats[0] #first matrix parameter
    for mat in mats[1:]:
        res = func(res, mat)
    return res
```

```
>>> a = circles()
>>> b = product()
>>> c = diagonals()
>>> abc = join(a, b, c, direction='h')
>>> abc.display()
```



Digital Image Processing

- Image processing is any form of signal processing for which the input is an image, such as a photograph or video frame.

- The output of image processing may be either an image or a set of characteristics or parameters related to the image.

- Most image-processing techniques involve treating the image as a two-dimensional signal and applying standard signal-processing techniques to it .

(text and figure taken from Wikipedia).

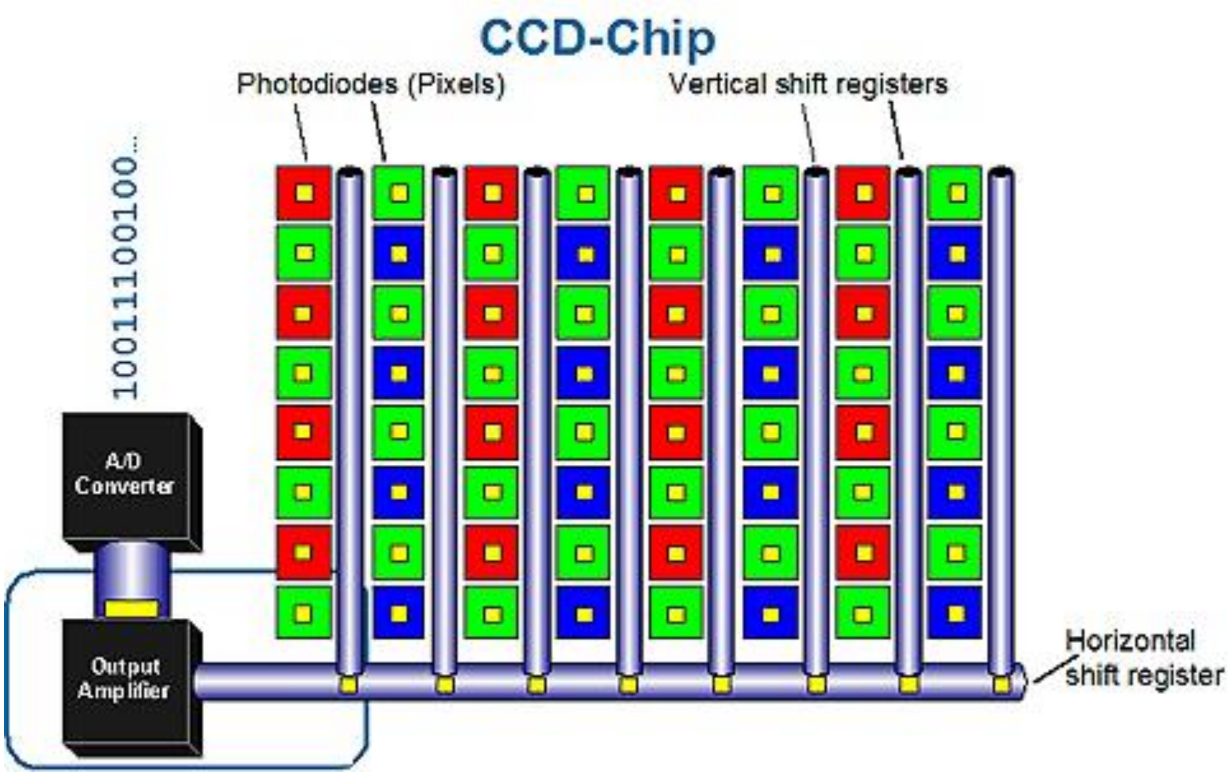
Digital Image Processing

- Common problems:
  - Noise reduction (denoising) - removing noise from an image.
  - Segmentation - partitioning a digital image into segments (e.g. background and foreground)
  - Tracking – identifying relate objects in subsequent frames of a film
  - Edge detection – detecting discontinuities in the image
  - Registration - transforming different images into one coordinate system (*e.g*. minor shifts in the camera position in subsequent frames
  - Color correction.

- Typical applications:
  - Machine vision
  - Medical / biological image analysis
  - Face detection
  - Object recognition
  - Augmented reality
  - …

CCD (for reference only)

- CCD (charge coupled device): transforming light (photons) to electrical voltage
- Each captor of the CCD is roughly a square area, in which the number of incoming photons is being counted for a fixed period.



(image and text taken from http://www.axis.com/edu/axis/)

Capturing Images

- Consider a specific pixel with coordinates x, y.
- Denote T(x, y) the "true" value at pixel x, y. This is the value which would be observed by averaging the photon count on a long period of time, assuming the image source is constant over time.

- The observed value, M(x, y), is the result of the light intensity measurement, usually made by a CCD, together with an optical light focusing system (lens or lenses).

Blur (for reference only)

- The two major effects hampering image accuracy are termed blur and noise.
- Blur is an intrinsic phenomenon to digital image acquisition. Blur most often occurs on out of focus objects or due to camera motion. While these kinds of blur can be prevented by adequate photography skills, there is a permanent intrinsic blur caused by the optics of image formation.
- To really understand it, a non negligible knowledge about signal processing is required. It is completely outside the scope of this course (and, unfortunately, of general CS studies as well).

California State University
Hello
San Bernardino
World
Department of Computer Science

An original image (left) and a blurred version thereof (right). Taken from Wikipedia.

Noise and Denoising

- The observed value at pixel x, y, M(x, y), equals the sum of the true value T(x, y) plus noise N(x, y).

$$M(x, y) = T(x, y) + N(x, y)$$

$$(0 \le x < n,\ 0 \le y < m)$$

- The goal of denoising algorithms is, given the observed image M to produce a new image, $\hat{T}$, which should be close to the original image T.

- Obviously such goal is not well defined, and thus cannot be solved, if there are no constraints on the image and on the noise.

Assumptions on the images

- We assume the image is piecewise smooth:

  Most of the image's area consists of large, smooth regions where light intensity varies continuously - if x_1, y_1 and $x_{2,}$ y_2 are neighbors, then $M[x_1,y_1]$ and $M[x_2,y_2]$ attain close enough values.

Gaussian Noise Model

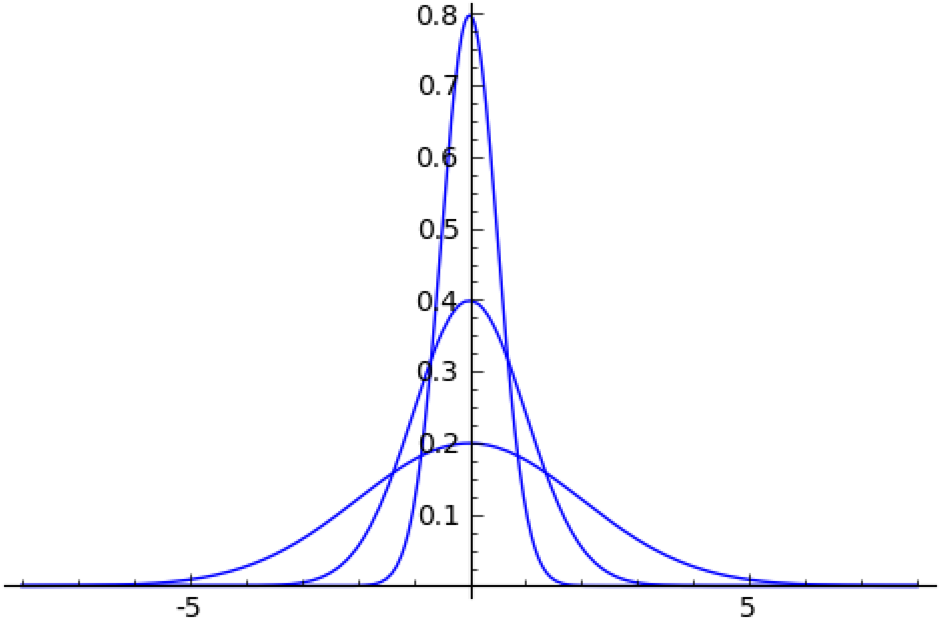
- **Gaussian noise model**: The noise at pixel x, y, N(x, y), is a random variable.

- It is usually assumed that N(x, y) is "white noise", distributed independently of the noise at other pixels.

Gaussians

- The probability density function $$G_\sigma(x) = \frac{e^{-x^2/2\sigma^2}}{\sigma\sqrt{2\pi}}$$

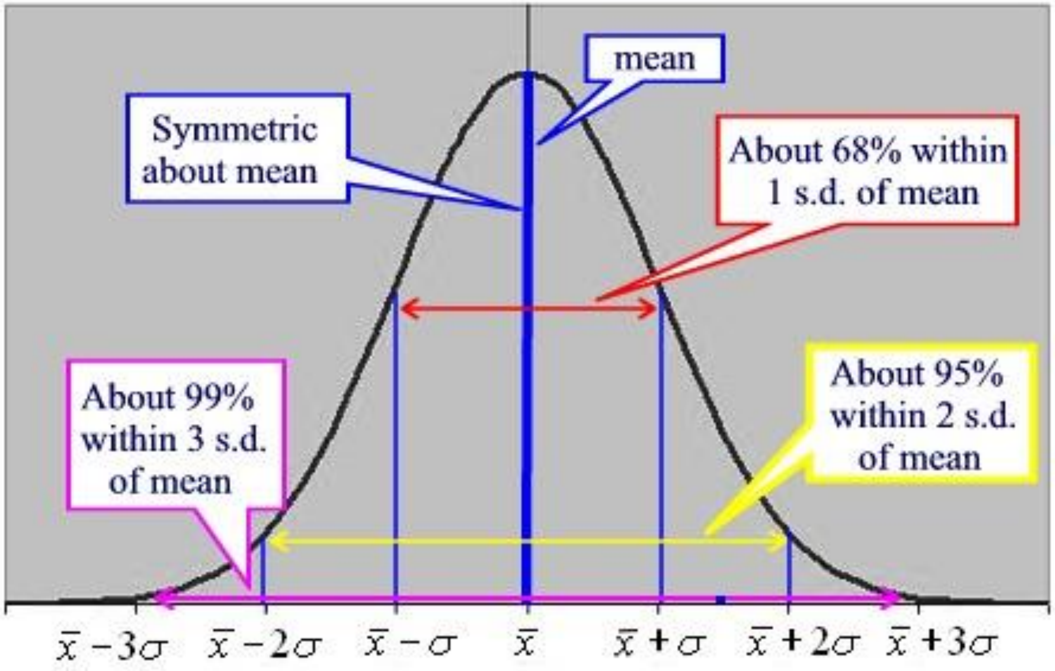
is called the Gaussian, or normal, distribution. It has mean 0 and standard deviation σ. This is a continuous function, which is the limit of the Binomial distribution, as the number of events tends to infinity. The Gaussian has the well known bell curve shape.



Three Gaussians, with σ = 0.5, 1, 2 (σ = 0.5 is the narrowest).

More on Gaussians

68% of the distribution lies within one standard deviation of the mean. **95%** of the distribution lies within two standard deviations of the mean. **99.7**% of the distribution lies within three standard deviations of the mean. These percentages are known as the "empirical rule".



http://www.regentsprep.org/regents/math/algtrig/ats2/normallesson.htm

Gaussian Noise: Python Code

- The function random.gauss(mu, sigma) returns a floating point number, distributed according to a Gaussian distribution with expected value (mean) μ and standard deviation σ.
- We will use μ = 0, and a default value σ = 10. When added to pixel values, we will round the noise and make sure the outcome falls within 0 to 255.

```
>>> import random
>>> random.gauss(0,10)
0.36121514047571907
>>> random.gauss(0,10)
21.643048694527852
>>> lst = [round(random.gauss(0,10)) for i in range(20)]
>>> lst
[-8, 22, 12, 4, -1, 2, 11, 6, -16, -1, 4, -9, -3, 1, -5, -3, 5, 18, 19, 1]
>>> sorted(lst)
[-16, -9, -8, -5, -3, -3, -1, -1, 1, 1, 2, 4, 4, 5, 6, 11, 12, 18, 19, 22]
```

14 out of 20 values between -10 and 10.
19 out of 20 between -20 and 20

Gaussian Noise: Python Code

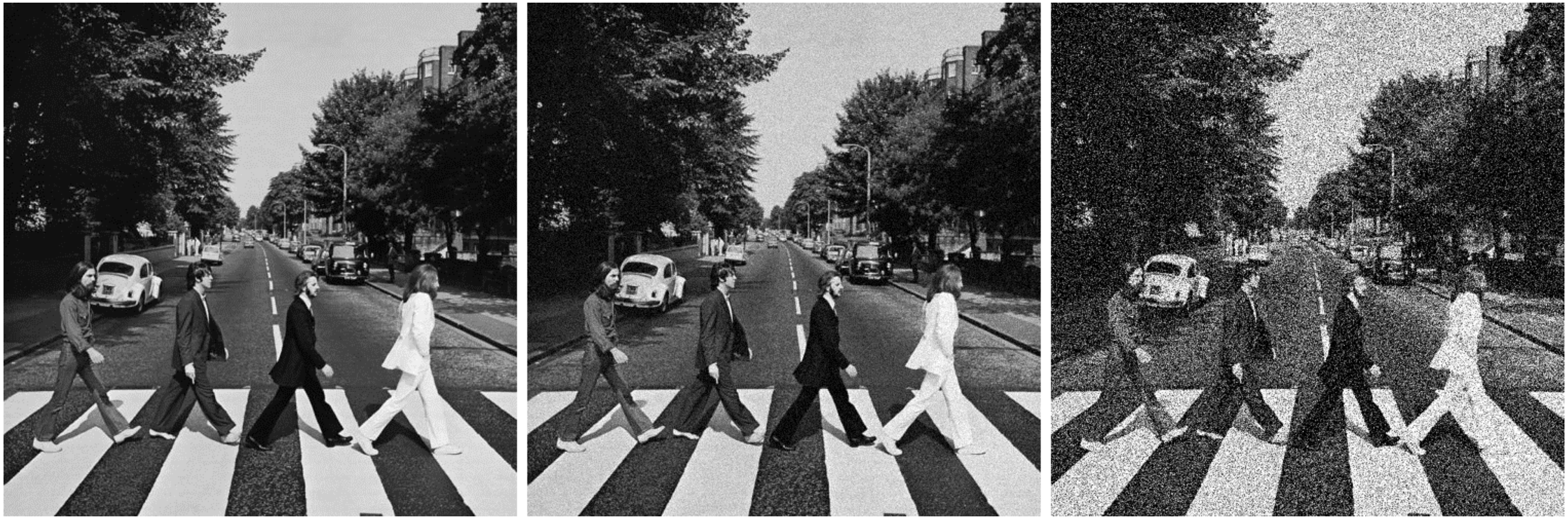
- To observe how Gaussian noise looks like, the following function adds such noise to a given image:

```
def add_gauss(mat, sigma=10):
       ''' Generates Gaussian noise with mean 0 and SD sigma.
           Adds indep. noise to pixel,
           keeping values in 0..255'''
       n,m = mat.dim()
       new = copy(mat)
       for i in range(n):
           for j in range(m):
               noise = round(random.gauss(0,sigma))
               if noise > 0:
                   new[i,j] = min(mat[i,j] + noise, 255)
               elif noise < 0:
                   new[i,j] = max(mat[i,j] + noise, 0)
       return new
```

Adding Gaussian Noise to an Image: Examples

We add Gaussian noise at different levels, and show pairs of results.

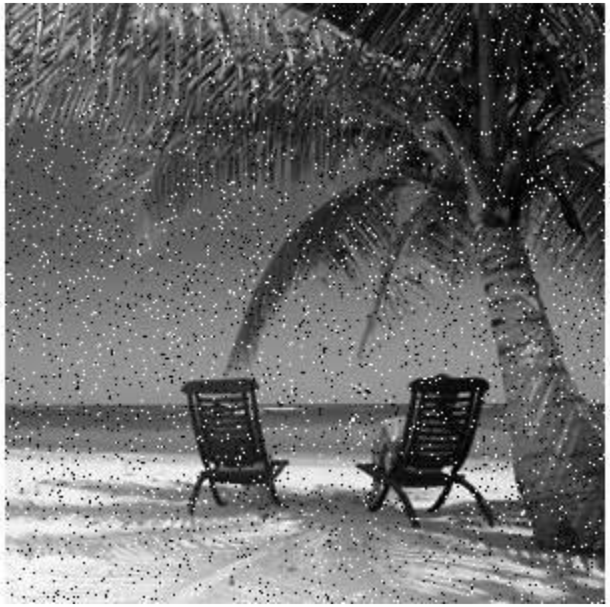
```
>>> img = Matrix.load("abbey_road.bitmap")
>>> new10 = add_gauss(img)
>>> new50 = add_gauss(img, sigma =50)
>>> joined = join(img, new10, new50, direction='h')
>>> joined.display()
```



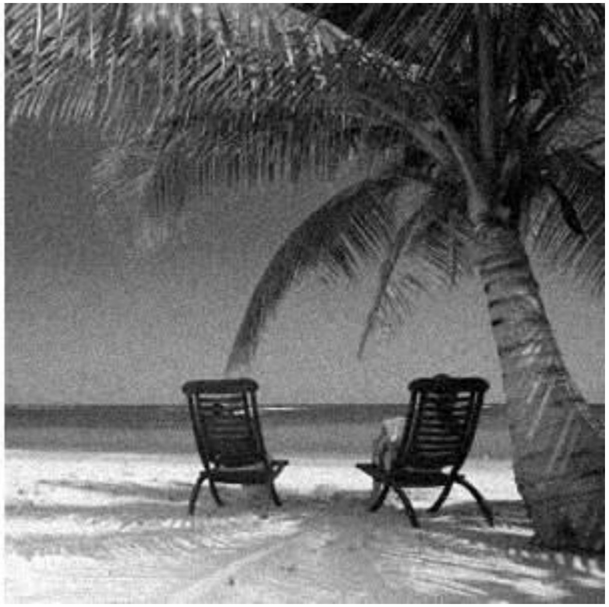
Salt and Pepper Noise Model

A different type of noise is the so called salt and pepper noise - extreme grey levels (white and black), or bursts, appearing at random and independently in a small number of pixels.

Original image



Salt & pepper noise



Gaussian noise

Salt and Pepper Noise: Python Code

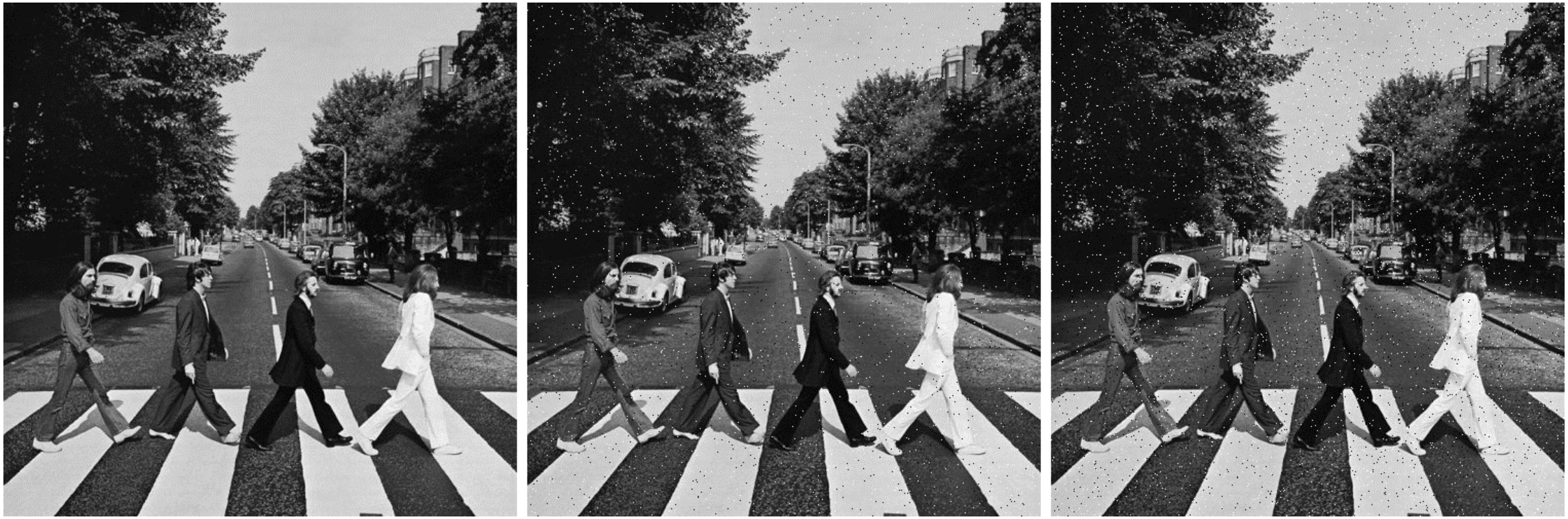
- Salt and pepper noise is not as fundamental in statistics as Gaussian noise is, so we'll write the code for it ourselves. The default parameter p=0.01 is the probability of a pixel being "hit" by the SP noise.

```
def add_SP(mat, p=0.01):
    ''' Generates salt and pepper noise: Each pixel is "hit" indep.
        with prob. p. If hit, it has fifty fifty chance of becoming
        white or black. '''
    n,m = mat.dim()
    new = copy(mat)
    for i in range(n):
        for j in range(m):
            rand = random.random() #a random float in [0,1)
            if rand < p:
                if rand < p/2:
                    new[i,j] = 0
                else:
                    new[i,j] = 255
    return new
```

Adding S&P Noise to an Image: Examples

We add Salt and Pepper noise at diffeent levels.

```
>>> img = Matrix.load("abbey_road.bitmap“)
>>> sp1 = add_SP(img)
>>> sp2 = add_SP(a,p =0.02)
>>> joined = join(img, sp1, sp2, direction = 'h')
>>> joined.display()
```



Denoising Algorithms

We will discuss two approaches to denoising, and implement them:

- Denoising by Local Means.
- Denoising by Local Medians.

We will also mention denoising by Non local means.

Of course, these three approaches are only the tip of the iceberg.

Local denoising

- We define a neighborhood of the pixel whose coordinates are x, y as the set of all pixels whose coordinates are close to x, y.
- A neighborhood commonly considered is the (2k+1)-by-(2k+1) square matrix of coordinates centered at x, y, where k is a small integer - typically 1 or 2.

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

- Local denoising changes the center pixel according to some function of its neighborhood.

Denoising by Local Means

- In denoising by local means, we replace the observed value at (x, y), M(x, y), by the average (or mean) of the observed values in a neighborhood of (x, y), for example, the 3-by-3 neighborhood above, $N_{3x3}(x, y)$.

- In writing the code, we should make sure we do not modify the original matrix of observed values.

Local denoising: Auxiliary Code

- items(mat) returns a list whose elements are the matrix elements.

```
def items(mat):
  '''flatten mat elements into a list'''
  n,m = mat.dim()
  lst = [mat[i,j] for i in range(n) for j in range(m)]
  return lst
```

- local_operator applies op on every pixel (except the boundaries of the image: pixels not in the center of a 2k+1-by-2k+1 window are left intact.)

```
def local_operator(mat, op, k=1):
  n,m = mat.dim()
  res = mat.copy()
  for i in range(k, n-k):
     for j in range(k, m-k):
       res[i,j] = op(items(mat[i-k:i+k+1, j-k:j+k+1]))
  return res
```

Local Means

```
def average(lst):
    n = len(lst)
    return round(sum(lst)/n)


def local_means(mat, k=1):
    return local_operator(mat, average, k)
```

Local Means: A Synthetic Example

```
>>> a = Matrix (4 ,4)
>>> for i in range (4):
        for j in range (4):
            a[i,j] = i + (j**2)
>>> for i in range (4):
      print ([a[i,j] for j in range (4)])

[0, 1, 4, 9]
[1, 2, 5, 10]
[2, 3, 6, 11]
[3, 4, 7, 12]
```

```
>>> b = local_means(a)
>>> for i in range (4):
      print ([b[i,j] for j in range (4)])

[0, 1, 4, 9]
[1, 3, 6, 10]
[2, 4, 7, 11]
[3, 4, 7, 12]
```

The example uses the default "frame radius" k=1 (9 pixels in frame).

Local Means: A Second Synthetic Example

```
>>> a[2,2] = 1000
>>> for i in range (4):
        print ([a[i,j] for j in range (4)])

[0, 1, 4, 9]
[1, 2, 5, 10]
[2, 3, 1000 , 11]
[3, 4, 7, 12]

>>> b = local_means (a)
>>> for i in range (4):
          print ([b[i,j] for j in range (4)])

[0, 1, 4, 9]
[1, 113 , 116 , 10]
[2, 114 , 117 , 11]
[3, 4, 7, 12]
```

- As you see, averaging is highly affected by "outliers".

Denoising by Local Means: Motivation

- If the pixel x, y resides in a smooth portion of the image, the light intensity in its neighborhood is about the same, so averaging will not change it significantly.

- On the other hand, it is known that averaging $(2k + 1)^2$ independent random variables with standard deviation σ, the standard deviation of the average decreases to

$$\sigma / \sqrt{(2k+1)^2}$$

which equals $\sigma/3$ for the $N_{3x3}(x, y)$ neighborhood.

- So in smooth areas, averaging preserves the signal component of the pixel, yet substantially decreases Gaussian noise contribution.

Local Means: Weighted Variants

- Uniform averaging based on the whole neighborhood, as discussed before, can be expressed as the matrix Frobenius inner product, namely sum of element by element product $\sum A_{i,j} \cdot B_{i,j}$

$$\begin{pmatrix} S[x-1,y-1] & S[x,y-1] & S[x+1,y-1] \\ S[x-1,y] & S[x,y] & S[x+1,y] \\ S[x-1,y+1] & S[x,y+1] & S[x+1,y+1] \end{pmatrix} \begin{pmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{pmatrix}$$

- A common variant puts more weight close to the central pixel. For example, in our case of the 3-by-3 neighborhood, replacing the 1/9 matrix by:

$$\begin{pmatrix} 1/12 & 1/12 & 1/12 \\ 1/12 & 1/3 & 1/12 \\ 1/12 & 1/12 & 1/12 \end{pmatrix}$$

- We point out that while this maintains more of the original signal, the noise reduction here is smaller.

Denoising by Local Means: Limitations

X When the pixel x,y does not reside in a smooth portion of the image, averaging does not preserve the signal component of the image. The outcome is an image with blurred edges.

X An additional disadvantage of averaging is its sensitivity to spurious extreme values (a general problem with average, not just in the images context), like those originating by salt and pepper noise.

- For example, suppose the original area of the image is fairly light, say intensity level around 240. Yet in the $N_{3x3}(x, y)$ neighborhood, one pixel, e.g. x-1, y-1, is observed as very dark, e.g. intensity level around 20, due to noise.
- Indeed, $\hat{T}$(x-1, y-1) will be corrected to 216. But each of the other 8 pixels containing x-1, y-1 in their neighborhood, will also exhibit such "correction", which is undesirable.

Denoising by Local Medians

- In denoising by local medians, we replace the observed value at (x, y), M(x, y), by the median of the observed values in a neighborhood of (x, y) (for example, the 3-by-3 neighborhood, $N_{3x3}(x, y)$, above).

✓ The median does preserve edges (a big plus).

✓ The median is not sensitive to spurious extreme values, so it withstands salt and pepper noise easily.

X However, the median tends to eliminate small, fine features in the image, such as thin contours.

X It also takes more time to compute median than mean.

Local Medians: Code

```
def median(lst):
   sort_lst = sorted(lst)
   n = len(sort_lst)
   if l%2==1: # odd number of elements. well defined median
      return sort_lst[l//2]
   else:    # even number of elements. average of middle two
      return (int(sort_lst[-1+n//2]) + int(sort_lst[n//2])) // 2


def local_medians(mat, k=1):
   return local_operator(mat, median, k)
```

- Comment: Median is computed by first sorting the values in the local window, and taking the middle element.

Local Medians: A Synthetic Example

```
>>> a = Matrix (4 ,4)
>>> for i in range (4):
        for j in range (4):
            a[i,j] = i + (j**2)
>>> for i in range (4):
        print ([a[i,j] for j in range (4)])

[0, 1, 4, 9]
[1, 2, 5, 10]
[2, 3, 6, 11]
[3, 4, 7, 12]

>>> b = local_medians(a)
>>> for i in range (4):
        print ([b[i,j] for j in range (4)])

[0, 1, 4, 9]
[1, 2, 5, 10]
[2, 3, 6, 11]
[3, 4, 7, 12]
```

Local Medians: A Second Synthetic Example

```
>>> a[2,2] = 1000
>>> for i in range (4):
        print ([a[i,j] for j in range (4)])

[0, 1, 4, 9]
[1, 2, 5, 10]
[2, 3, 1000 , 11]
[3, 4, 7, 12]

>>> b = local_means (a)
>>> for i in range (4):
        print ([b[i,j] for j in range (4)])

[0, 1, 4, 9]
[1, 113 , 116 , 10]
[2, 114 , 117 , 11]
[3, 4, 7, 12]
```

```
>>> c = local_medians (a)
>>> for i in range (4):
      print ([c[i,j] for j in range (4)])

[0, 1, 4, 9]
[1, 2, 5, 10]
[2, 3, 7, 11]
[3, 4, 7, 12]
```

- The medians essentially ignore outliers.

Time Complexity of Local Means and Local Medians

- Suppose the image dimensions are n-by-m.

- The number of windows that are wholly contained in the image is $(n - 2k)(m - 2k) = O(n \cdot m)$. Assuming k is significantly smaller than m, n, this upper bound is also tight.

- For every such window, we either compute the average of the values in the window, or find their median.

- The number of pixels in a window is $(2k + 1)^2 = 4k^2 + 4k + 1 = O(k^2)$.
  - This is the time complexity to compute the average.
  - For median, we employed sorting, taking $O(k^2 \log k^2) = O(k^2 \log k)$ steps. But faster median finding, running in time which is linear in the number of items, is known (you may see it in the data structures / algorithms courses). Computing median can therefore be done in $O(k^2)$ steps too. (The hidden constant for the local means will be smaller than the hidden constant for the local medians.)

- All in all, $O(k^2nm)$ steps (but not in our median implementation).

Putting Local Means/medians to the Test

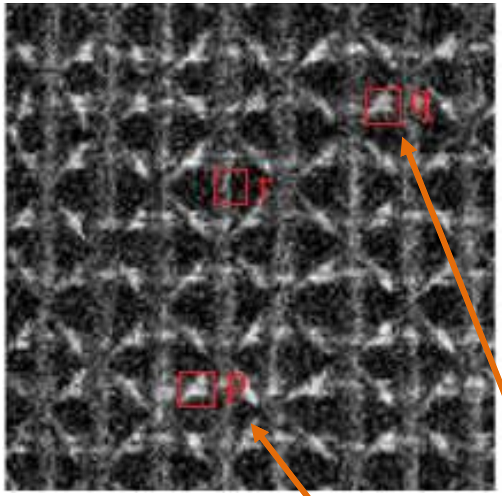
We will explore different local denoising methods on-line in class, and display results back to back with original or each other.

Any conclusions? Which method is better? Where is it better?

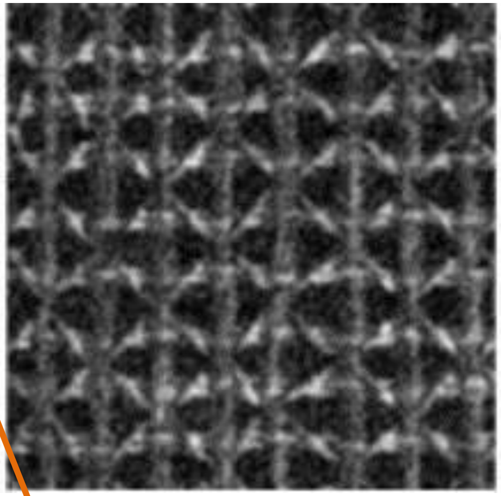
Time (and energy) permitting, we will also explore variants with larger local windows (specifically, $k=2$).

Towards Non-Local Means: Regularity in Natural Images

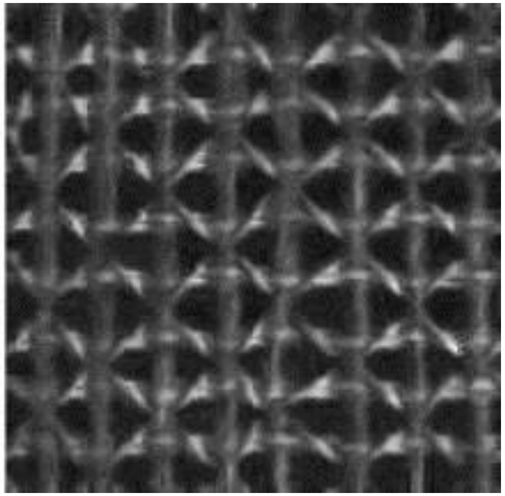
- Many natural images have a high degree of redundancy. Specifically, this means that for most small windows in the original image, the window has many similar windows in the same image.



Gaussian noise means



Local means



Non-local means

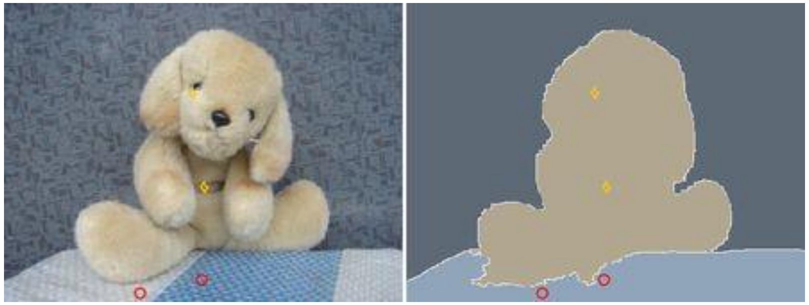
- Windows centered at p and q are similar, but not to the one centered at r.

Denoising by Non-Local Means

- The non-local (NL) means algorithm of (A. Buades, B. Coll, and J. M. Morel, 2005) heavily employs the notion of non-local, similar windows. Given a window centered at (x, y), we search for all windows in the image that are similar to it.
- In other words, we look for all (x', y') such that the "distance" between the windows centered at x, y and x', y' is below some fixed threshold h.
- We compute the weighted average value of all those similar center pixels (including (x, y) itself), with higher weights assigned to windows that are more similar. The corrected value, $\hat{T}$(x, y), equals this average.
- The method is called non-local since the windows that effect the corrected value $\hat{T}$(x, y) are not necessarily in close proximity to (x, y).
- Remark: This is a fairly simplified version of NL means. For reasons of efficiency, one usually scans only a subset of all possible windows.

Segmentation

- The process of partitioning a digital image into multiple segments (sets of pixels, also known as superpixels).



Source:
http://www.sonycsl.co.jp/person/nielsen/applets.html

- The goal of segmentation is to simplify and/or change the representation of an image into something that is more meaningful and easier to analyze.

- Image segmentation is critical for many subsequent processes, such as object recognition, shape analysis and tracking. It is typically used to locate objects and boundaries (lines, curves, etc.).

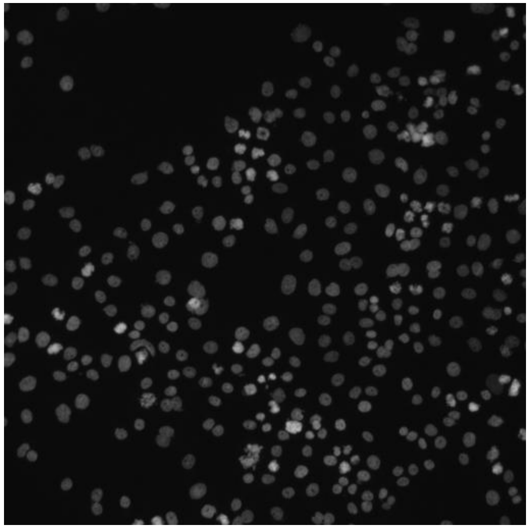
- Examples: locating tumors or anatomical structures in medical images; face detection; identifying objects in satellite images (roads, forests, crops, etc.).

Binary Segmentation by Thresholding

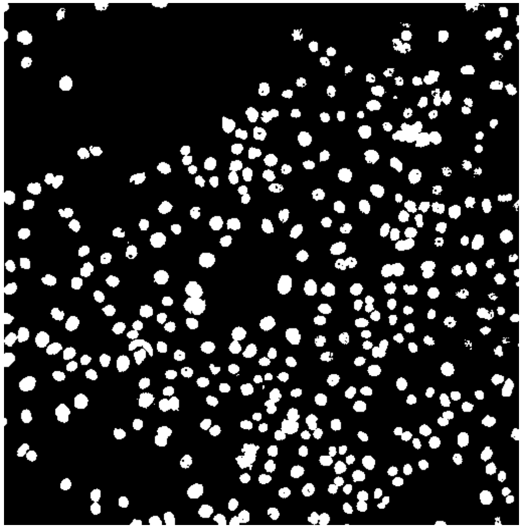
- Simplest segmentation method.

- Apply a threshold to turn a gray-scale image into a binary image (BW) – this is called binary segmentation.

- Assumes the image contains two classes of pixels denoted foreground and background, and these two classes have distinct, different light intensities: the background is much darker than the foreground.



Human HT29 colon-cancer cells
http://www.broadinstitute.org/bbbc/image_sets.html



Binary segmentation, threshold = 40

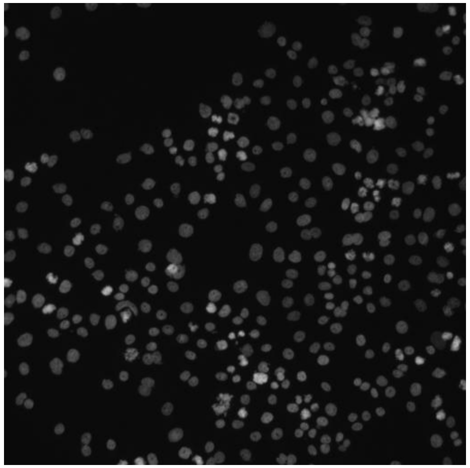
- Generally, one can apply more than one threshold, creating >2 segments

Picking a Threshold

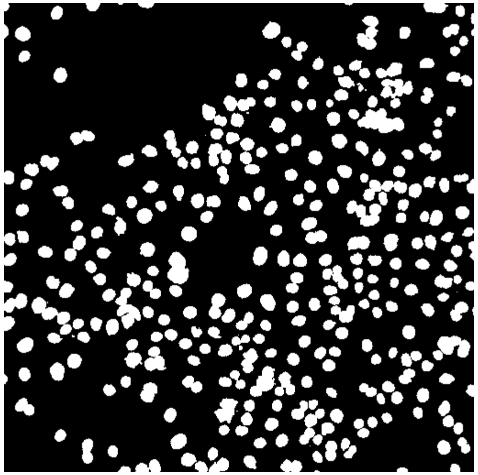
- The key is to select the appropriate threshold
- Which one is the best here?

- When the threshold is too low (20 in this case) areas in the image where cells are densely populated become bulbs.
- When it is too high (60) some cells are lost (those whose brightness was low in the original image).

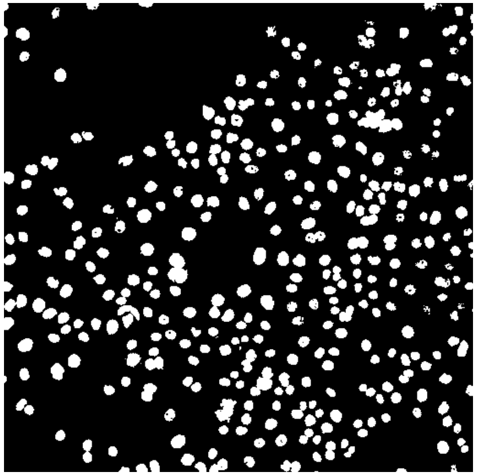
Original



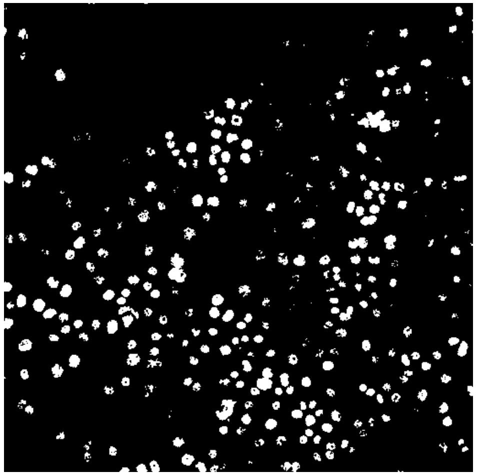
Threshold = 20



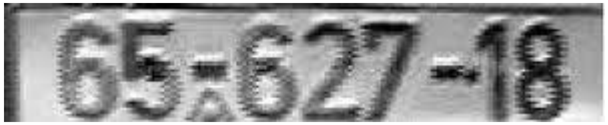
Threshold = 40



Threshold = 60

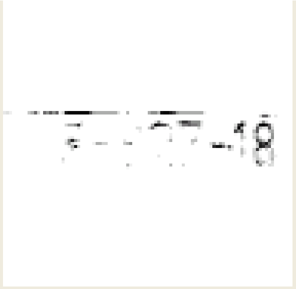


Binary Segmentation – another example

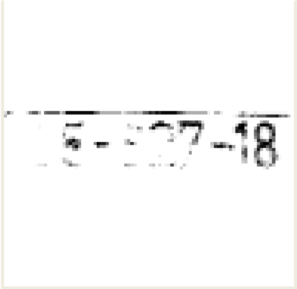


- Below are the results of binary segmentation with increasing thresholds (out_20 for example uses threshold 20).

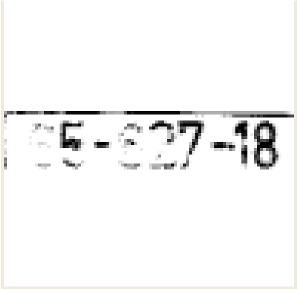
out_20.bmp



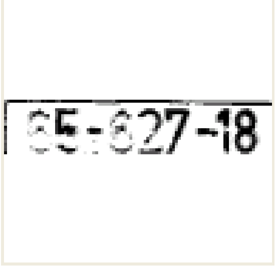
out_40.bmp



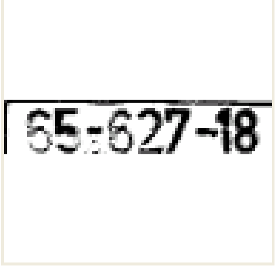
out_60.bmp



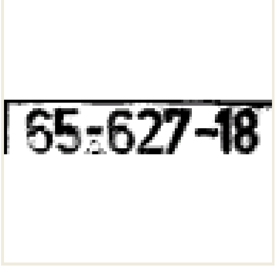
out_80.bmp



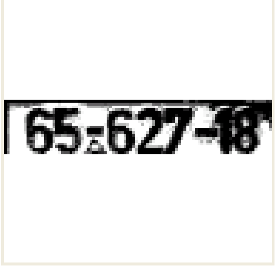
out_100.bmp



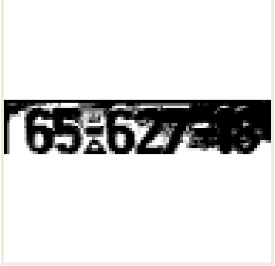
out_120.bmp



out_140.bmp



out_160.bmp



out_180.bmp

More on Image Processing (for reference only)

- Otsu segmentation
- Edge detection
- Erosion and dilation
- Compression, the jpg format

Otsu method for threshold calculation

- A good threshold for segmentation:
  - minimizes differences within each segment, and
  - maximizes differences between segments.

- Otsu's method finds an optimal threshold for segmentation.

- Uses image histogram: grey level values distribution.
  - x-axis – grey hues
  - y-axis – number of pixels with a particular hue

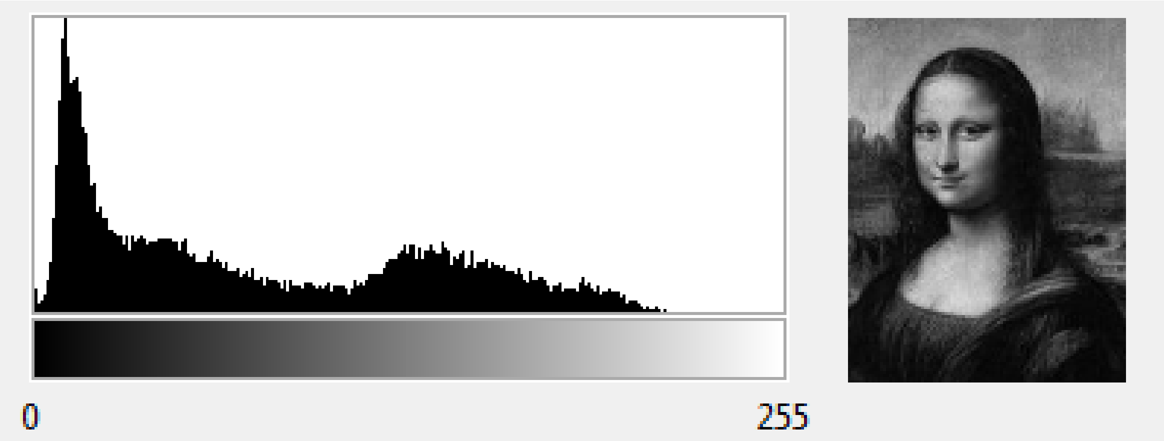
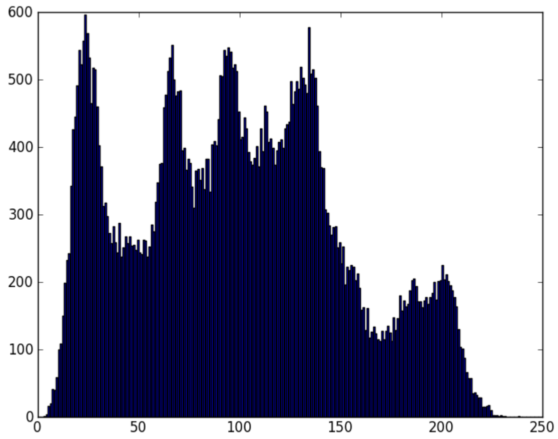


Image Histogram - Code

```
def histogram(im):
    ''' Return a histogram as a list,
        where index i hold the number of pixels with value I
    '''
    mat = im.load()
    width, height = im.size
    hist = [0]*256

    for x in range(width):
       for y in range(height):
            gray_level= mat[x,y]
            hist[gray_level] += 1

    return hist #hist[i] = number of pixels with gray level=i
```

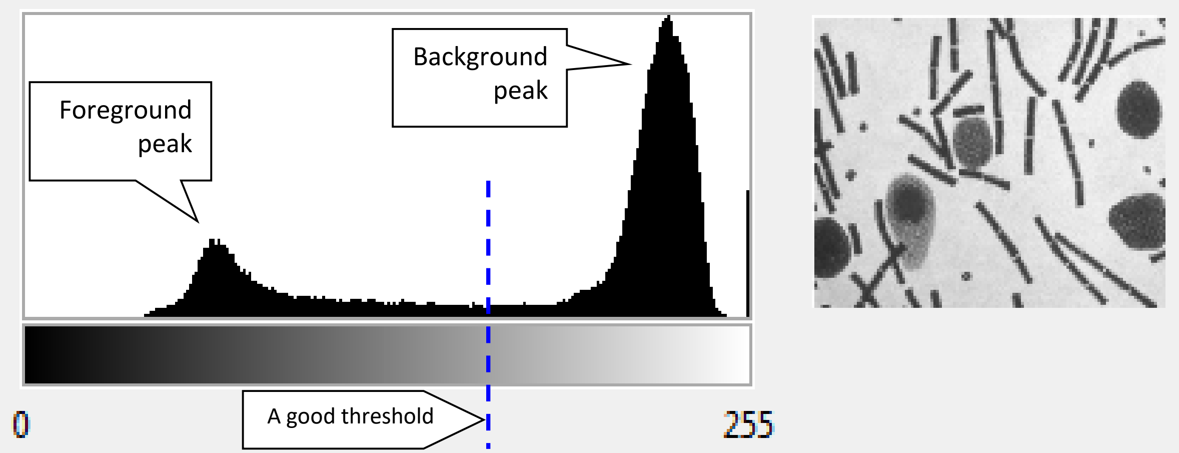


Otsu method for threshold calculation

- Otsu's method relies on the assumption that the foreground and the background of the image differ substantially in their brightness.

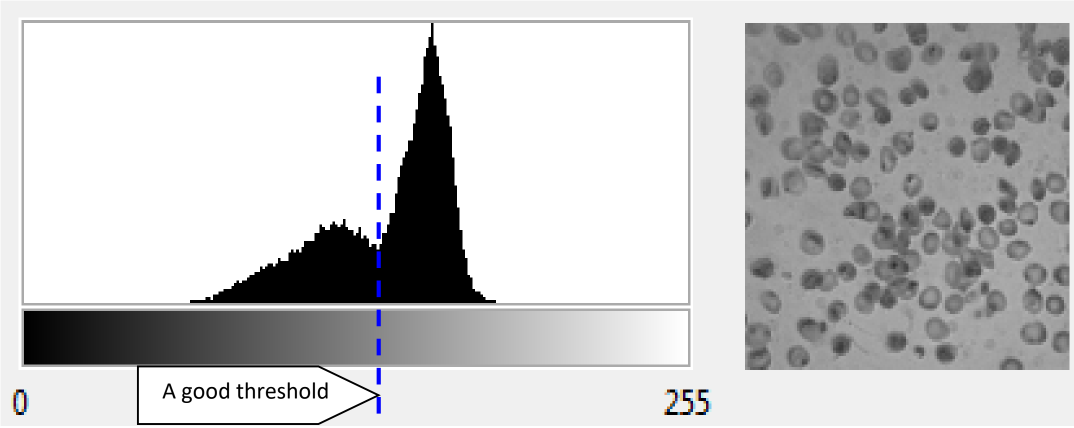
- This assumption is not true in many cases, as in the Mona Lisa example.

- However, when this assumption holds, there are expected to be two peaks in the gray values of an image’s histogram (such image histograms are called bi-modal).
- In this case the lowest mid-point between these two peaks would be a good choice for a threshold.

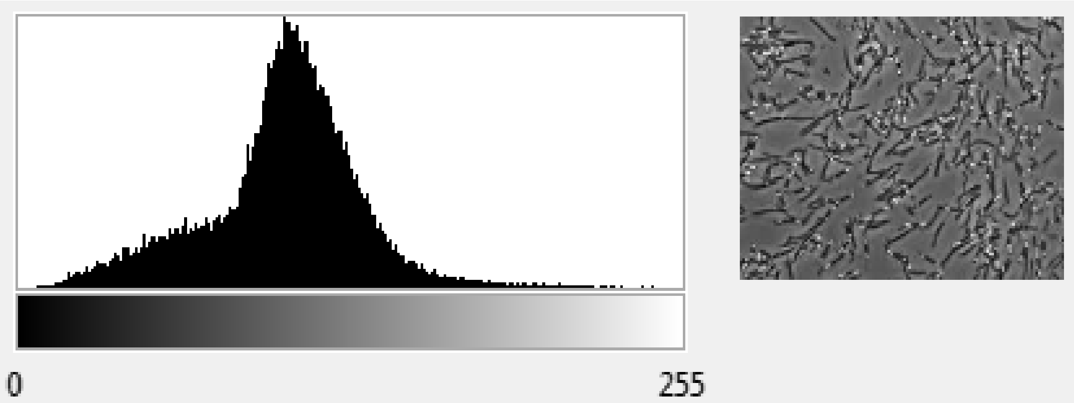


Otsu method for threshold calculation

- When the difference between foreground and background are less sharp, the peaks may be partly overlapping:



- Furthermore, when the image is rather uniform, there will be no such two peaks at all (in which case Otsu's method will be inapplicable):



Otsu's Formula

For every threshold *t* denote:

back – number of background pixels (<= t)
fore – number of foreground pixels (> t)

mean_back – mean value of the background pixels
mean_fore – mean value of the foreground pixels

var_between(t) = *back * fore * (mean_back - mean_fore)*^{2}

- Otsu threshold is the one that maximizes the *var_between* among all possible thresholds *t*.
- What is the effect of the difference between the means?
- What is the effect of the relative sizes of the background and foreground?

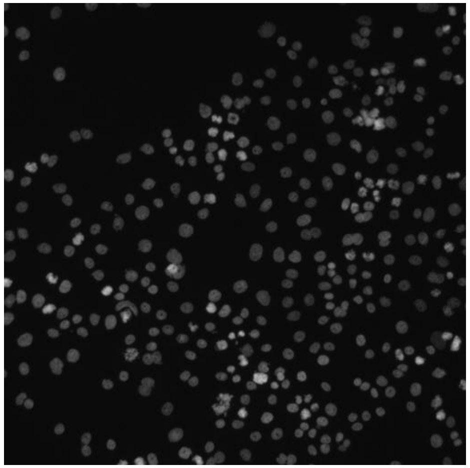
Otsu threshold - Run

We will not show code for Otsu’s method (HW?).

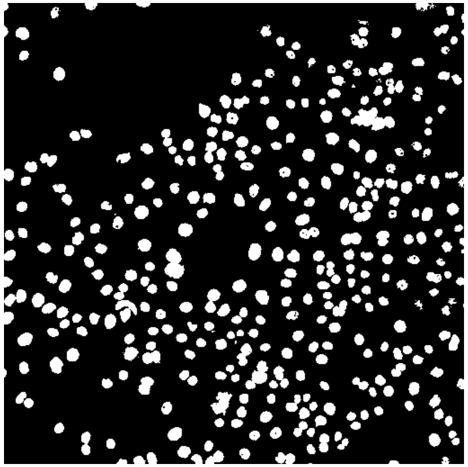
But here is an execution:

```
>>> im = Matrix.load("./HT29.bitmap")
>>> th = otsu(im)
38
>>> segment(im, th).display()
```

Original: Human HT29 colon-cancer cells

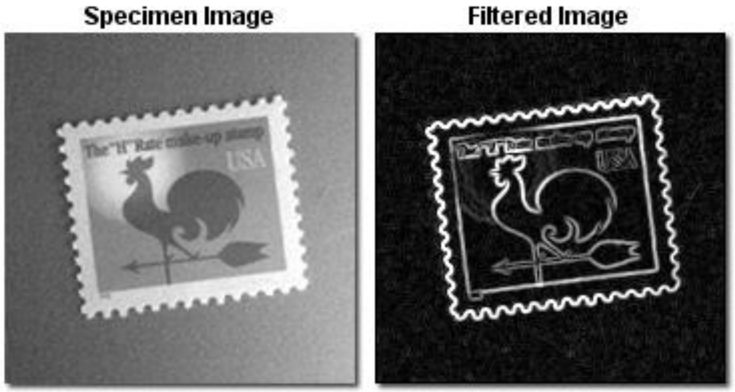


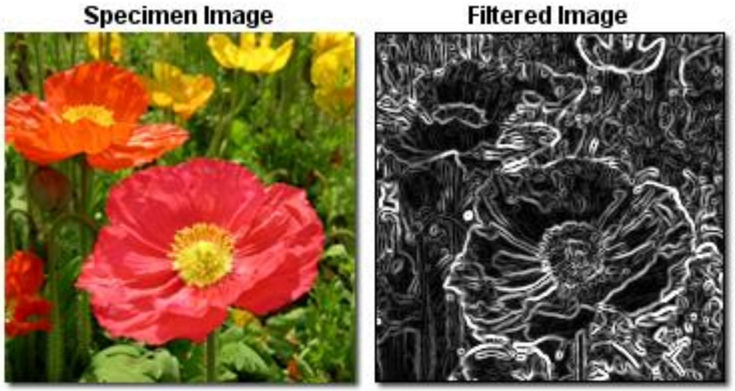
Otsu's Threshold = 38



Edge Detection

- Edge - sharp change in intensity between close pixels
- Usually captures much of the meaningful information in the image





images extracted using Sobel filter from:

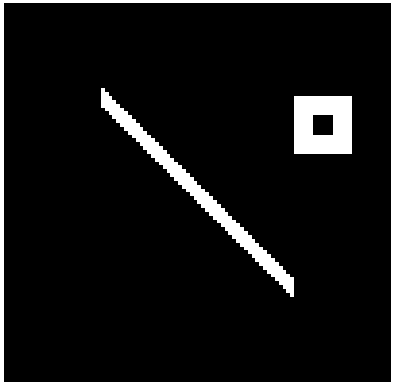
http://micro.magnet.fsu.edu/primer/java/digitalimaging/russ/sobelfilter/index.html

Erosion and Dilation

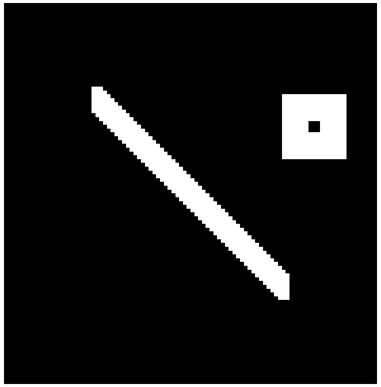
Assume features in the foreground are bright and background is dark.

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

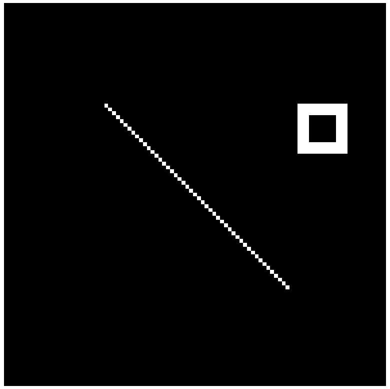
- Dilation - the addition of pixels to the periphery of features.
  - enlarges foreground areas, and holes shrink.



Original



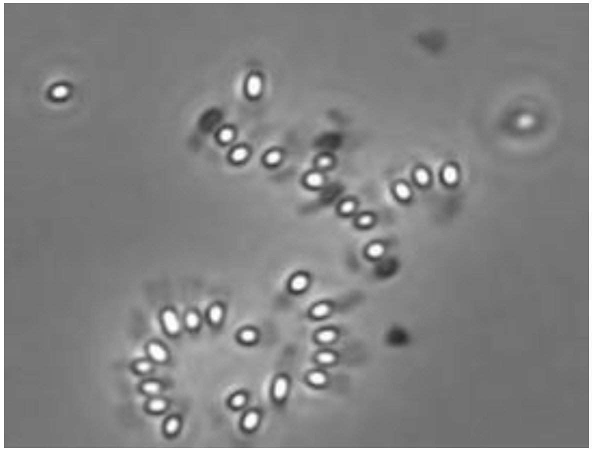
Dilation



Erosion

- Like segmentation, these basic operators are often used to pre-process or post-process images to facilitate analysis.

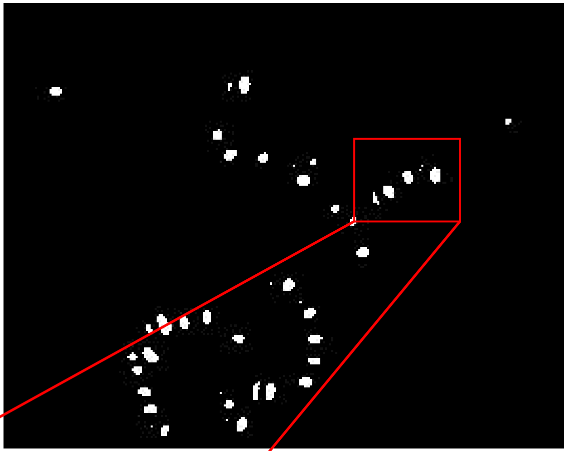
Erosion and Dilation - Example

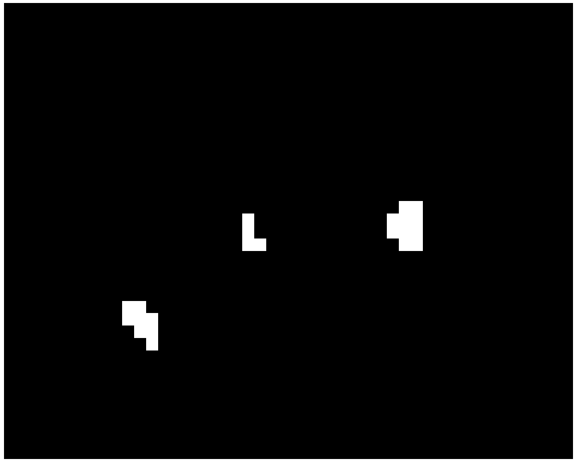


A microscope slide containing Clostridium botulinum cells and spores. Spores appear bright with dark boundaries (the spore coat). Vegetative cells were stained to provide contrast, and thus appear dark

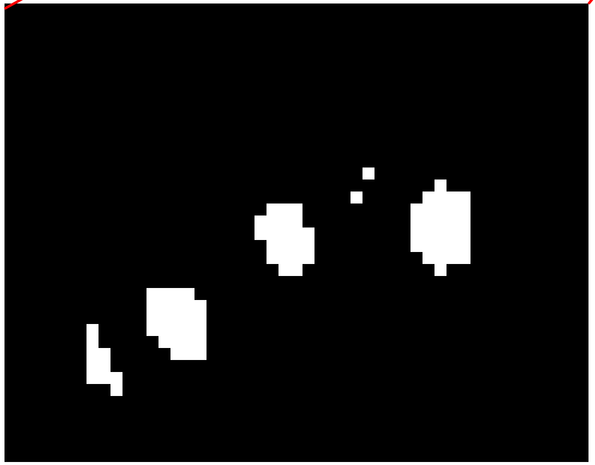
Source: Martin, M.D., *Phase contrast image of germinating spores of a non-pathogenic clostridia that grows at low temperatures*. 2013.

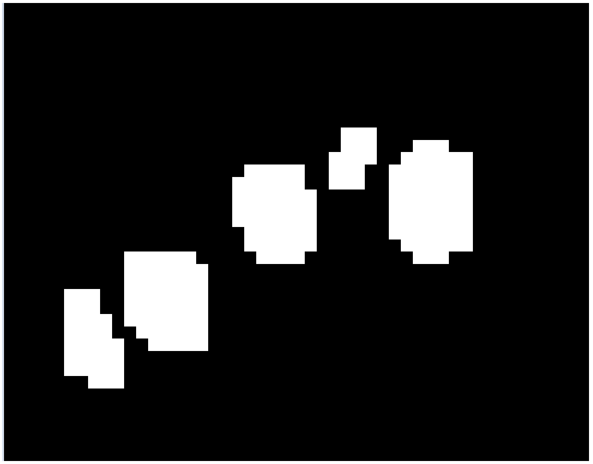
Binary segmentation
(th=200)





Erosion





Dilation

Compression and Image Formats

- Digital images with high pixel resolution and bit depth take up lots of computer memory.

- This motivates the need for compressing images.

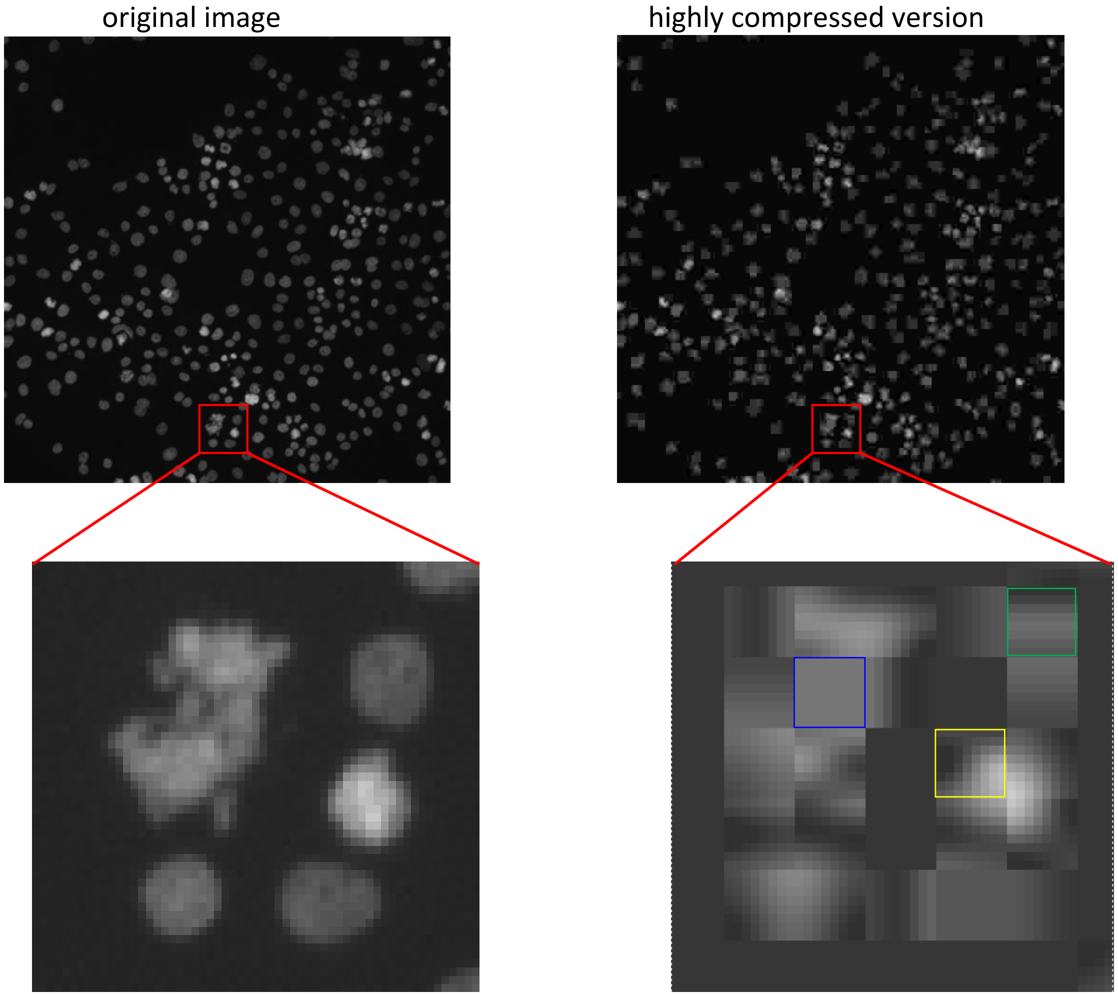
- During compression, some of the information in the image may be lost, in which case the compression is termed lossy. Otherwise, we call it lossless.

- jpg, tiff, png, bmp, gif etc., differ by the type of compression applied to the original image.

  The bmp format is lossless, while the other formats are lossy (tiff can be both, depending on some parameter settings).

The example of jpg

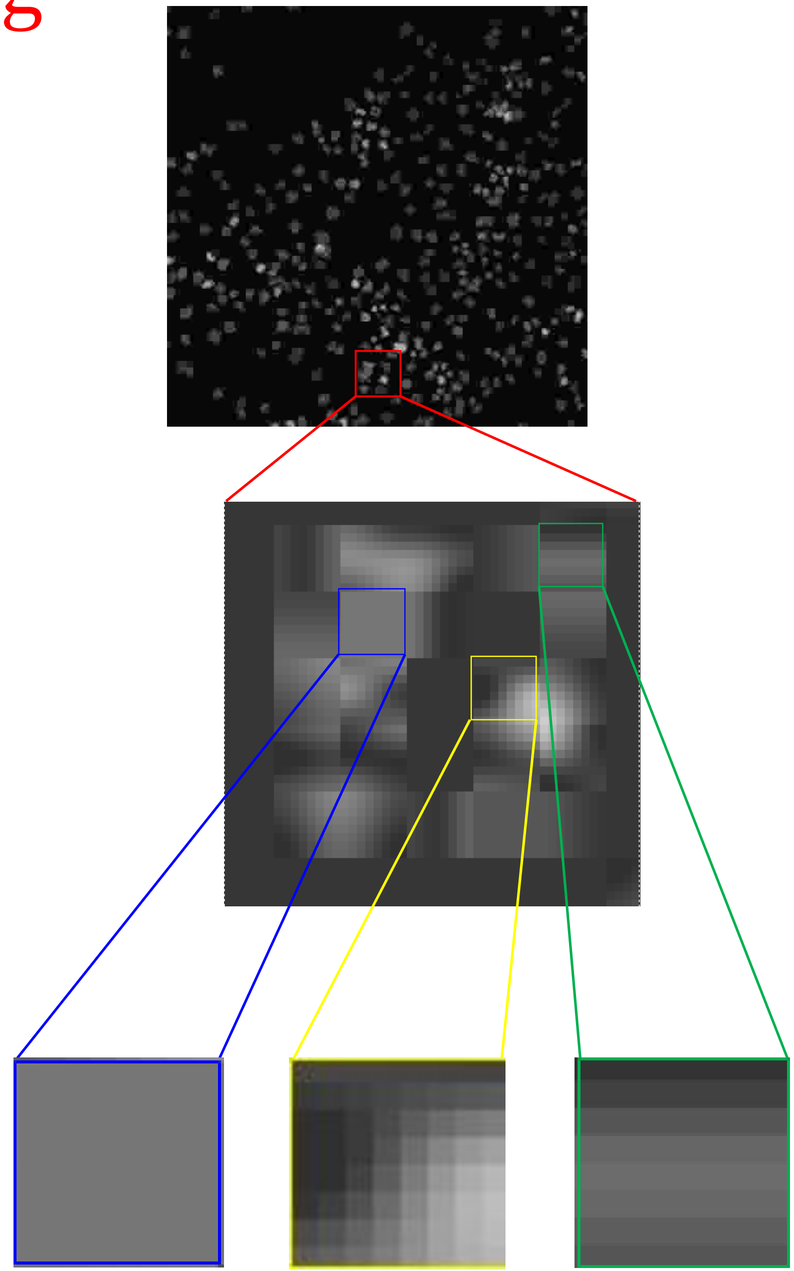
- jpg format partitions the image into squares of 8-by-8 pixels.
- Most such squares will exhibit only gradual, moderate changes, especially in smooth areas of the image.



- These gradual changes can be well approximated by far fewer bits than the 8·8·8 = 512 bits in the original representation.
- A factor of 10 (or even more) saving in space can be achieved.

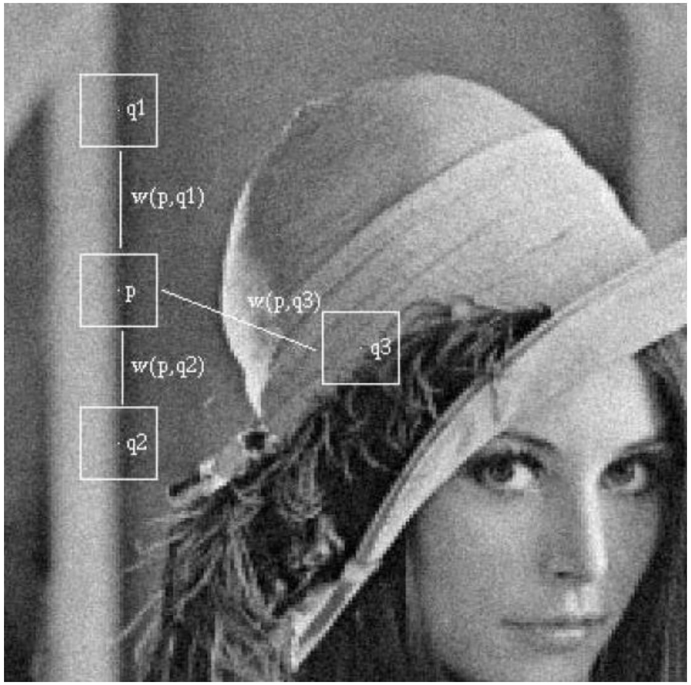
Human HT29 colon-cancer cells.
In the compressed image on the right, In the blue square all pixels are identical. In the green square, pixels only change from top to bottom. In the yellow square, pixels change in both directions.

The Example of jpg



Human HT29 colon-cancer cells.
In the compressed image on the right, all the pixels in the blue square are identical. In the green square, pixels only change from top to bottom. In the yellow square, pixels change in both directions.

Some non-CS issues



From Wikipedia: Lenna or Lena is the name given to a standard test image widely used in the field of image processing since 1973. It is a picture of Lena Sderberg, shot by photographer Dwight Hooker, cropped from the centerfold of the November 1972 issue of Playboy magazine. Given the nature of the image and its source, several academics have criticized its continued use in scientific publications and higher education as both sexist and unprofessional.

The course staff joins this view. We do our best to avoid objectification of women in the course or the course material.