Introduction to computer science in Python

Fall 2021-22

Department of Software and Information Systems Engineering

Ben-Gurion University of the Negev

Topic 13: Image representation and processing

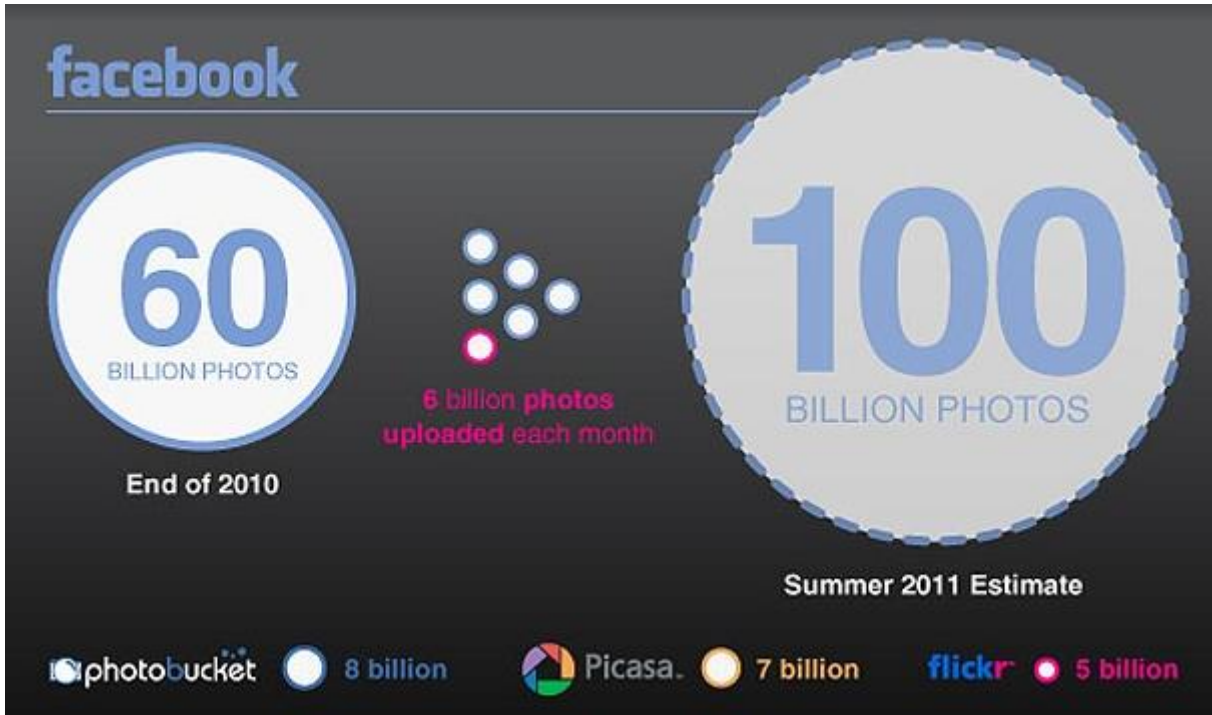
Signal processing

- In the physical world, any quantity measurable through time or over space can be taken as a signal
- Signals are or electrical representations of time-varying or spatial-varying physical quantities
- Signal processing: applying mathematical techniques for the extraction, transformation and interpretation of signals, in either discrete (digital) or continuous (analog) time
- Example signals: radio, telephone, radar, sound, images, video, sensor data

Digital images

- Digital image is a numeric representation of a two-dimensional image
- Examples: photos, microscopic, medical, astronomical

*Help me find a more updated visualization!

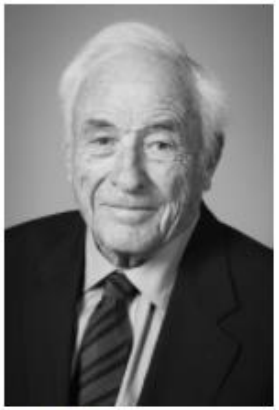


Charge Coupled Device (CCD)

Transforming light (photons) to electrical voltage

The Nobel Prize in Physics 2009

© The Nobel Foundation. Photo: U. Montan
Charles Kuen Kao
Prize share: 1/2



© The Nobel Foundation. Photo: U. Montan
Willard S. Boyle
Prize share: 1/4

© The Nobel Foundation. Photo: U. Montan
George E. Smith
Prize share: 1/4

The Nobel Prize in Physics 2009 was divided, one half awarded to Charles Kuen Kao "for groundbreaking achievements concerning the transmission of light in fibers for optical communication", the other half jointly to Willard S. Boyle and George E. Smith "for the invention of an imaging semiconductor circuit - the CCD sensor."

https://www.nobelprize.org/prizes/physics/2009/prize-announcement/

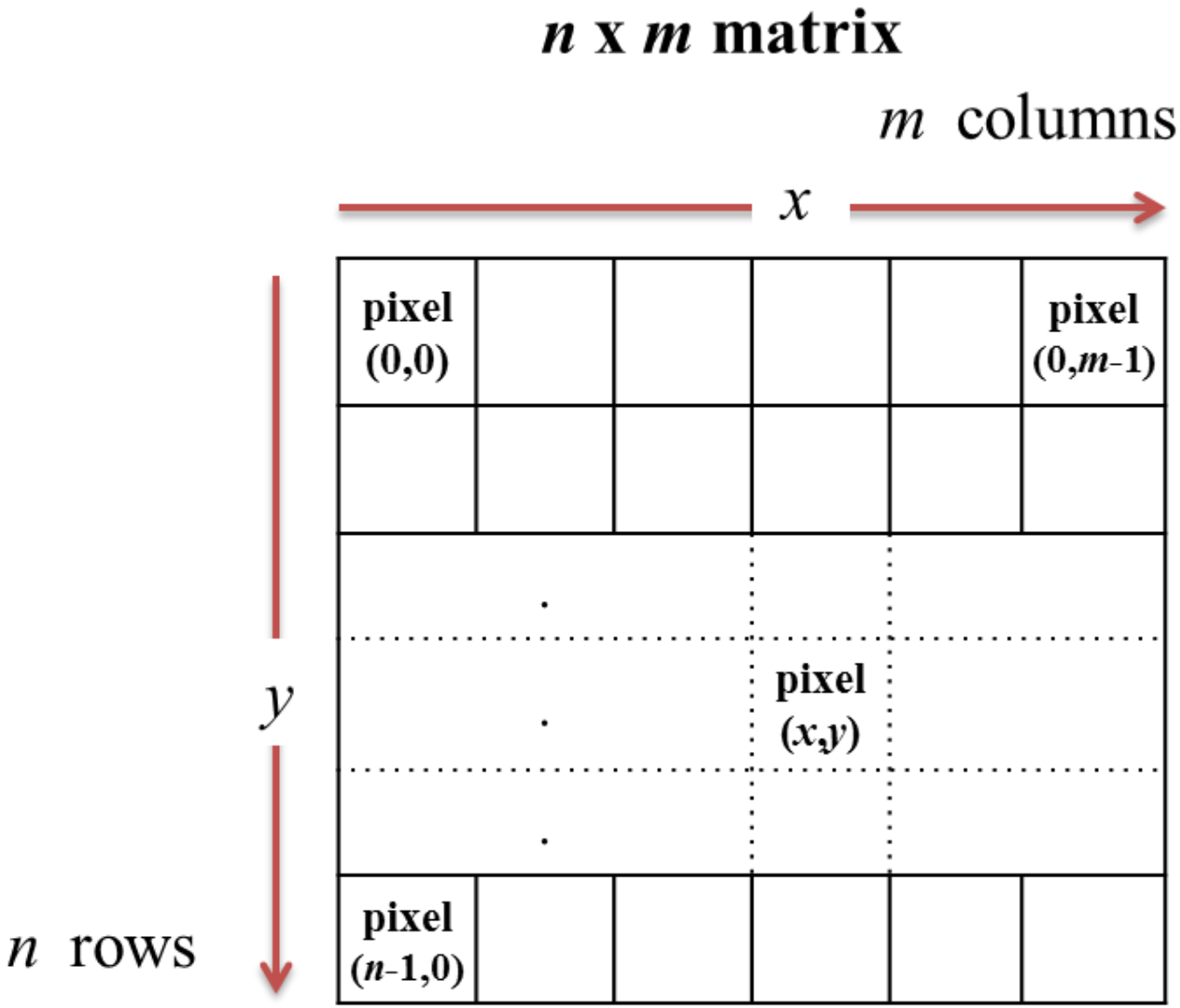
Popular science (Hebrew): https://davidson.weizmann.ac.il/online/sciencehistory/את-ששינה-החיישן- התמונה

Image representation and processing

- Digital image representation
- Generating synthetic images
- Basics of image processing
- Noise, and local noise reductions
- Image segmentation

Basic model of a digital image

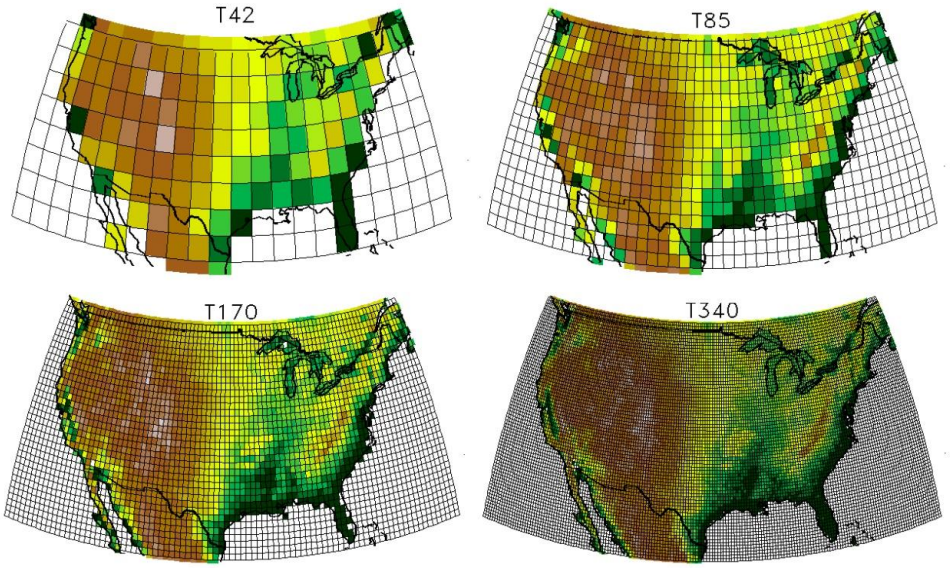
- Pixel: each element M[x, y] of the image

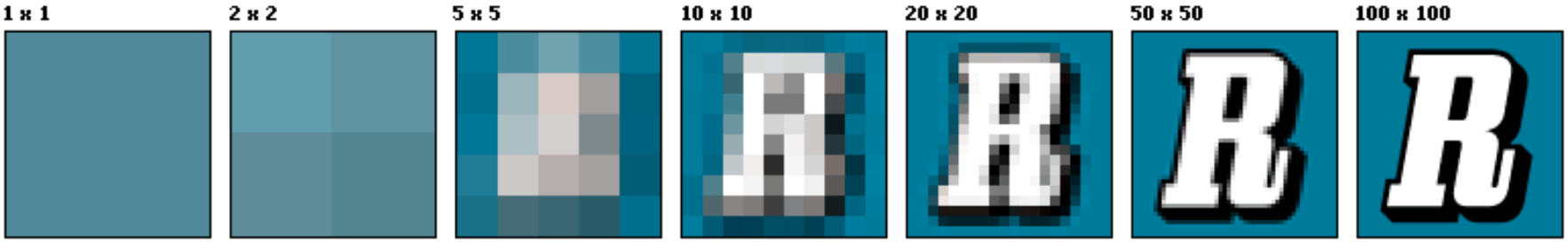


Resolution

- Resolution is the capability of the sensor to observe or measure the smallest object clearly with distinct boundaries
- Resolution depends on the physical size of a pixel

  Higher resolution = more pixels per area = **lower** pixel size

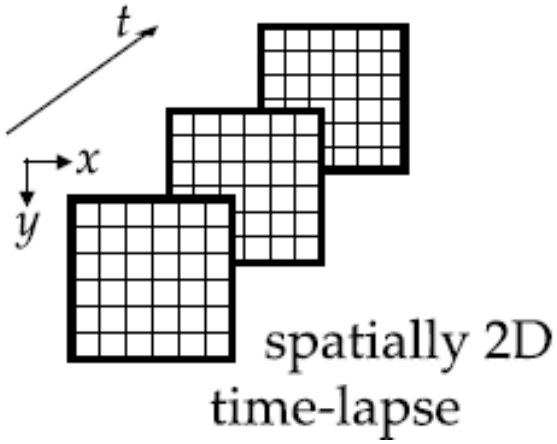




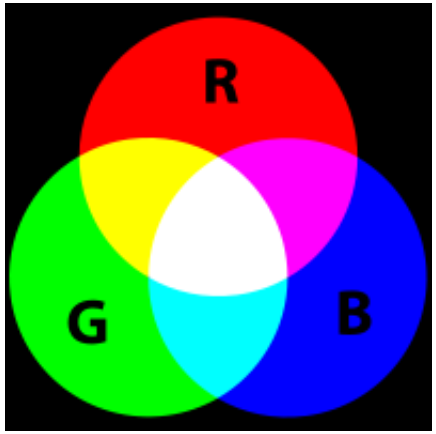
Source: wikipedia

2D, 2D + time (video), 3D, color

- A 2D image is encoded as a n-by-m matrix M
- Video – a 3rd "time“ dimension



- 3D image
- Color (e.g., RGB)



Gray scale images

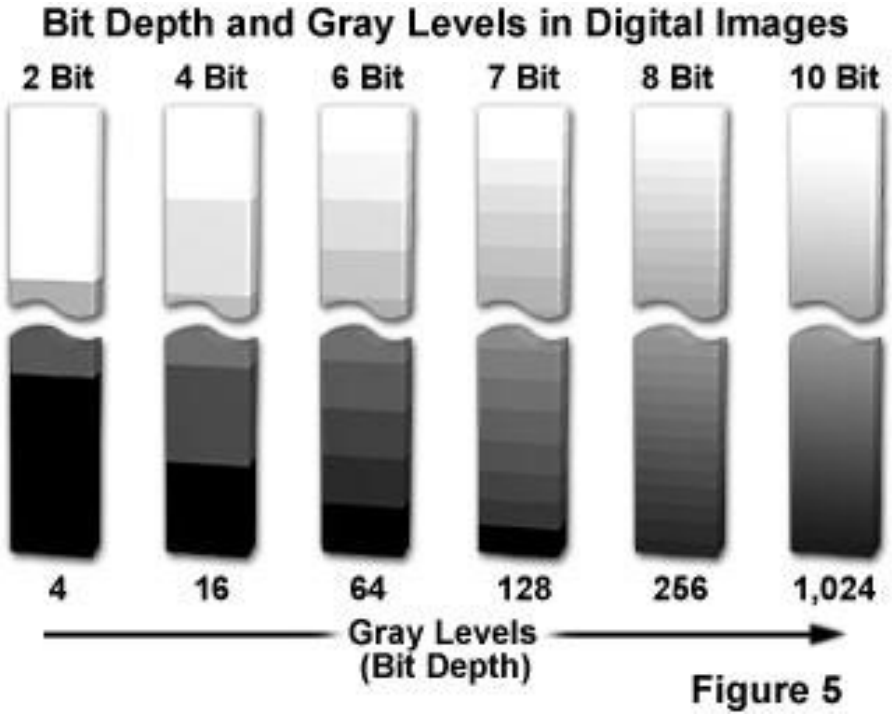
- We discuss gray scale images only (for simplicity)
- Real numbers expressing gray levels have to be discretized
- A good quality photograph (human visual inspection) has 256 gray-level values (8 bits) per pixel
- The value 0 represents black, 255 represents white
- In some applications (e.g., medical imaging) 4096 gray levels (12 bits) are used

Bit Depth

- Number of bits per pixel.

Image from:

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

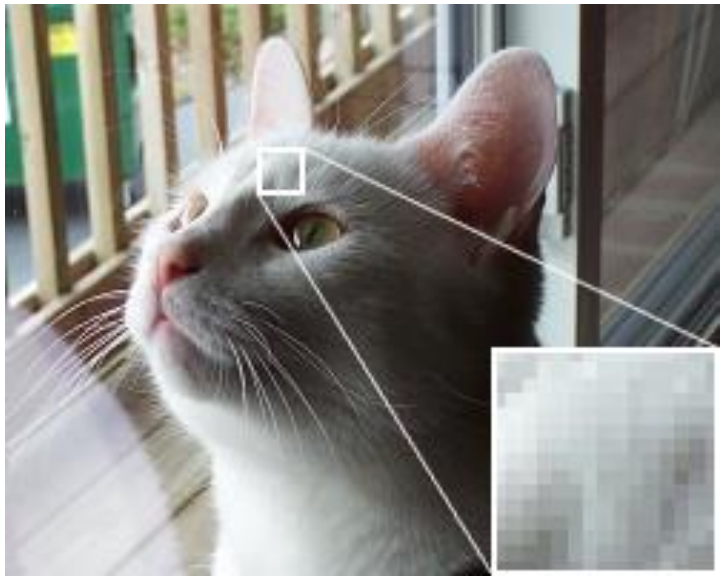


- A human observer sees at most a few hundreds shades of gray
- Higher bit depths images: typically for automated analysis by a computer

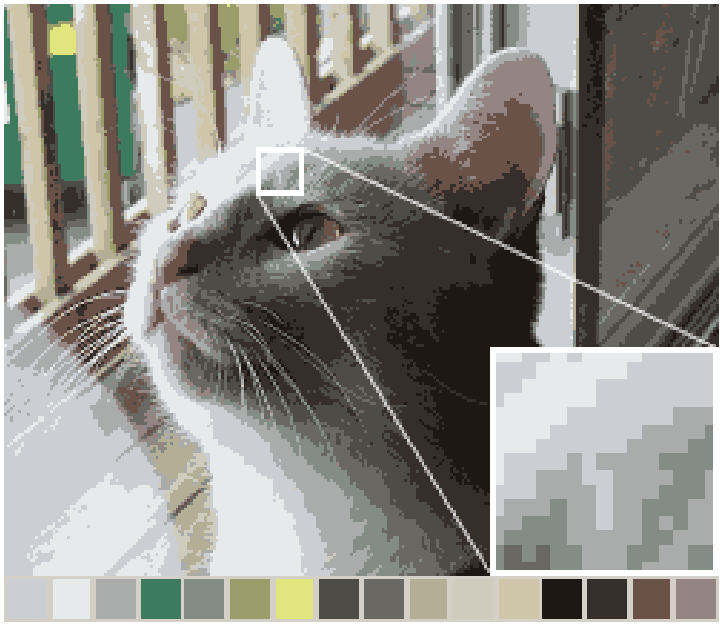
Image Quantization

Number of bits per pixel

24 bit RGB



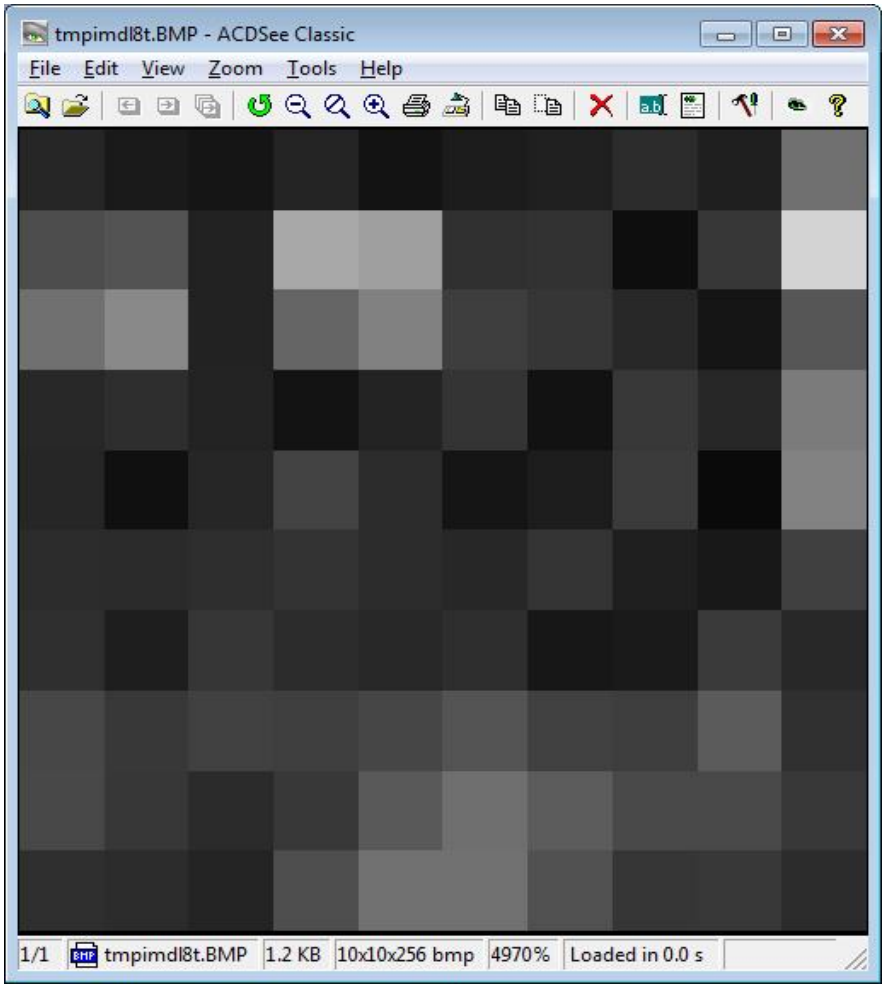
16 colors



Note that both images have the same pixel & spatial resolution

Gray scale image

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



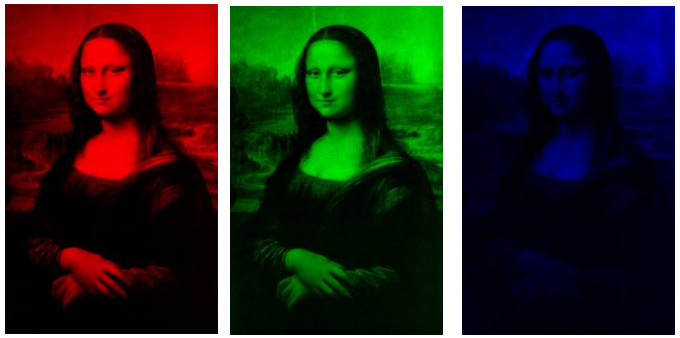

```
38,  26,  21, 36,  19,  28,  33, 44, 31, 112,
77,  83,  34, 168, 159, 48,  50, 14, 55, 211,
112, 137, 34, 101, 129, 62,  54, 40, 21, 86,
41,  46,  35, 19,  35,  52,  18, 57, 39, 123,
38,  16,  38, 67,  45,  21,  29, 59, 10, 130,
45,  43,  46, 51,  44,  39,  53, 31, 24, 64,
47,  30,  54, 45,  40,  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
```

BW / Grayscale / RGB

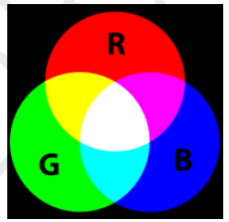
- Black & white / gray-level / RGB

Black & white
(1 bpp)

256 gray level image
(8 bpp)



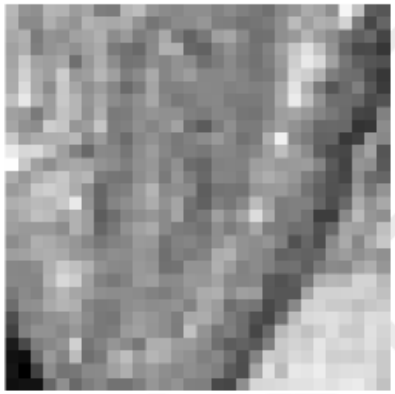
Images from: http://www.csse.uwa.edu.au/~wongt/matlab.html



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

Video tutorial on colors here: https://www.youtube.com/watch?v=syJo37AKaro

An image



Any guesses as to what this image is (or is part of)?

```
[183, 148, 143, 181, 178, 156, 165, 126, 123, 181, 189, 148, 139, 135, 142]
[178, 138, 138, 175, 158, 147, 179, 171, 168, 173, 147, 117, 127, 139, 139]
[168, 176, 123, 147, 142, 161, 165, 176, 140, 154, 125, 136, 169, 122, 99]
[161, 173, 127, 147, 154, 144, 161, 170, 122, 113, 105, 138, 177, 175, 104]
[164, 200, 162, 158, 167, 111, 151, 174, 140, 115, 117, 141, 133, 146, 140]
[170, 208, 171, 158, 204, 152, 158, 182, 159, 126, 138, 169, 134, 147, 157]
[171, 219, 170, 140, 199, 166, 143, 157, 134, 106, 123, 164, 129, 129, 133]
[192, 232, 180, 156, 200, 181, 149, 168, 140, 130, 144, 167, 135, 120, 125]
[160, 154, 122, 157, 194, 181, 145, 175, 122, 124, 141, 148, 144, 144, 128]
[165, 145, 140, 205, 197, 150, 157, 197, 124, 128, 144, 133, 145, 162, 111]
[199, 160, 172, 174, 175, 184, 136, 156, 125, 108, 135, 145, 133, 121, 129]
[234, 215, 218, 193, 159, 129, 104, 137, 135, 118, 141, 156, 135, 122, 126]
[254, 199, 160, 142, 163, 168, 163, 147, 140, 127, 144, 151, 127, 144, 109]
[173, 152, 173, 188, 199, 175, 182, 124, 117, 116, 141, 154, 122, 150, 126]
[154, 163, 200, 206, 197, 172, 142, 102, 124, 128, 155, 180, 138, 142, 139]
```

Our implementation: the class Matrix

- Class Matrix, 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)]
    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
```

The class Matrix

- Additional methods (we will skip most of these today):
  - **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 library matplotlib (remember graph visualization?)
  - **save** and **load**: enable storing and reading images from files

class Matrix - item access and assignment

```
>>> m = Matrix(10, 10)   # 10x10 matrix of zeros
>>> m[4,5]               # same as m.__getitem__((4,5))
0
>>> m[4,5] = 45          # same as m.__setitem__((4,5),45)
>>> m[4,5]
45
```

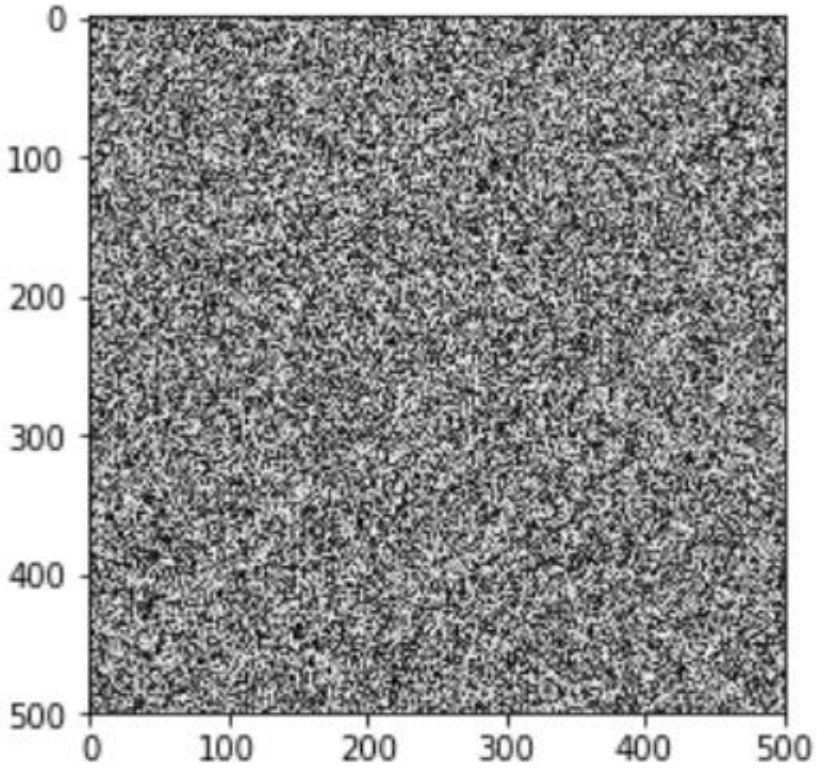
Note: the code file contains an additional feature: accessing and assignment of a whole slice.

-
```
>>> m[3:5, 4:8]          # here i and j are both slices
<Matrix [[0 , 0, 0, 0], [0, 45, 0, 0]] >
```

Loading, saving and displaying an image

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

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



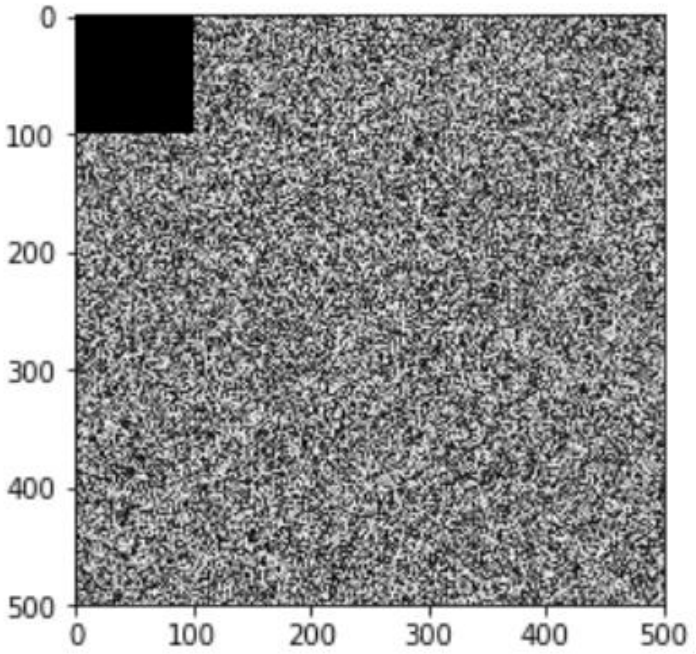
A few issues to discuss regarding image representation

- "Translating" our custom .bitmap format to standard image formats in next (hidden) slide (which I'll be skipping today)
- Any suggestions on how to improve the inner representation of an image?
  - numpy arrays (intro to DS)
- Of course there are existing Python libraries with image (and image processing) functionalities, e.g., https://scikit-image.org/

Generating simple synthetic images

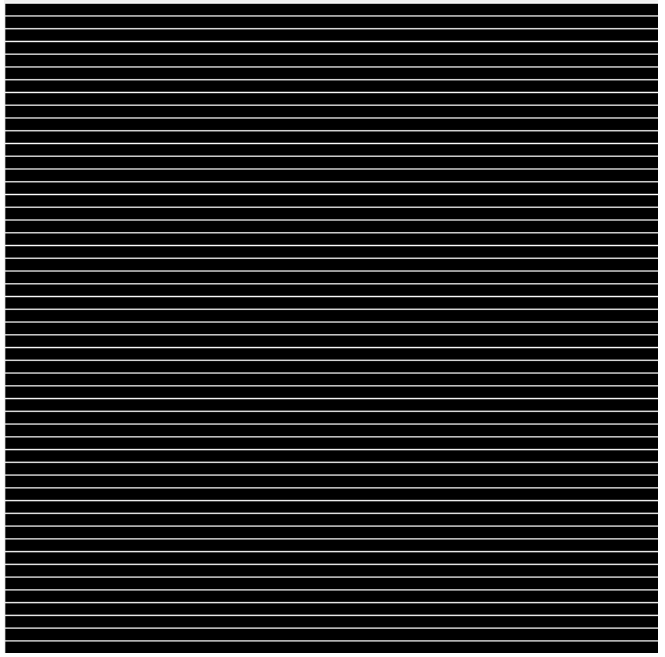
Drawing a black square

```
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

black_square(mat).display()
```



Horizontal lines

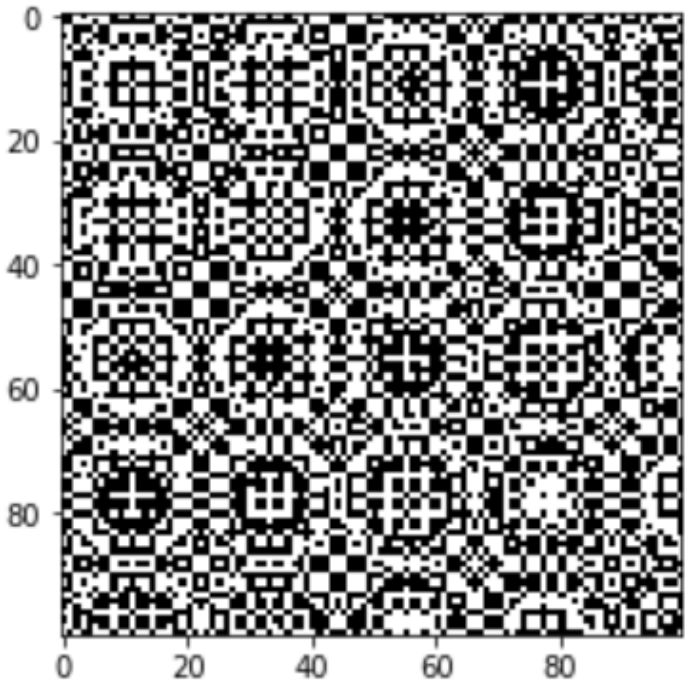
```
def horizontal(size=512):
    horizontal_lines = Matrix(size,size)

    for i in range(1,size):
        if i%10 == 0:
            for j in range(size):
                horizontal_lines[i,j] = 255

    return horizontal_lines

im = horizontal(512)
im.display( )
```



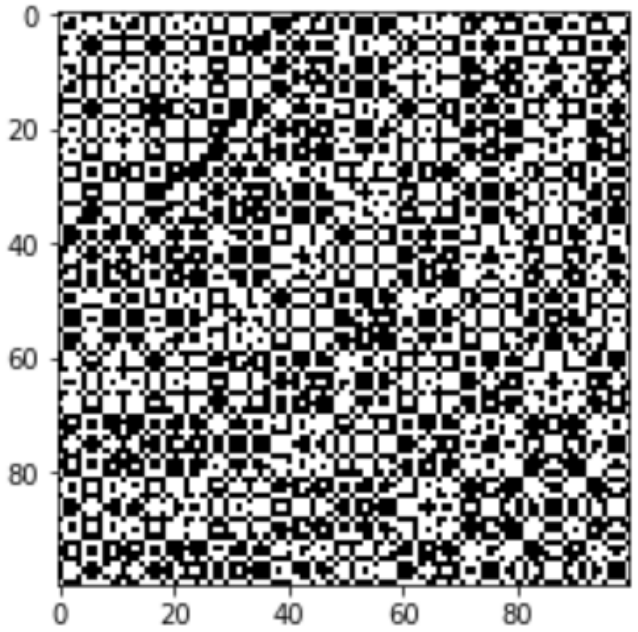
Functions as arguments

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

```
def sin_mul(i,j):
    return math.sin((i**2+j**2))%256
synthetic(100,100,sin_mul).display()
```




```
def cos_mul(i,j):
    return math.cos(4*(i**2 + j**2))%256
synthetic(100,100,cos_mul).display()
```



Try generating synthetic images yourself!

Digital Image Processing

Digital image processing

Image processing is any form of signal processing for which the input is an image, the output may be either an image or a set of characteristics or parameters related to the image

Source: Wikipedia

Digital image processing

- Common problems:
  - Noise reduction (denoising) - removing noise from an image.
  - Segmentation - partitioning a digital image into segments (*e.g.*, identify the boundaries of cells in a multi-cell image)
  - Tracking - 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
  - …

Blur

- Blur and noise are two major effects hampering image accuracy
- Blur is intrinsic to image acquisition. For example, out of focus due to camera motion or due to the optics
- Take a signal processing course (probably not in ISE) to understand more...

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)

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

- Denoising algorithms: given the observed image M, produce a new image, $\hat{T}$, which should be close to the original image T

- This goal is not well defined, and can be solved only with putting 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

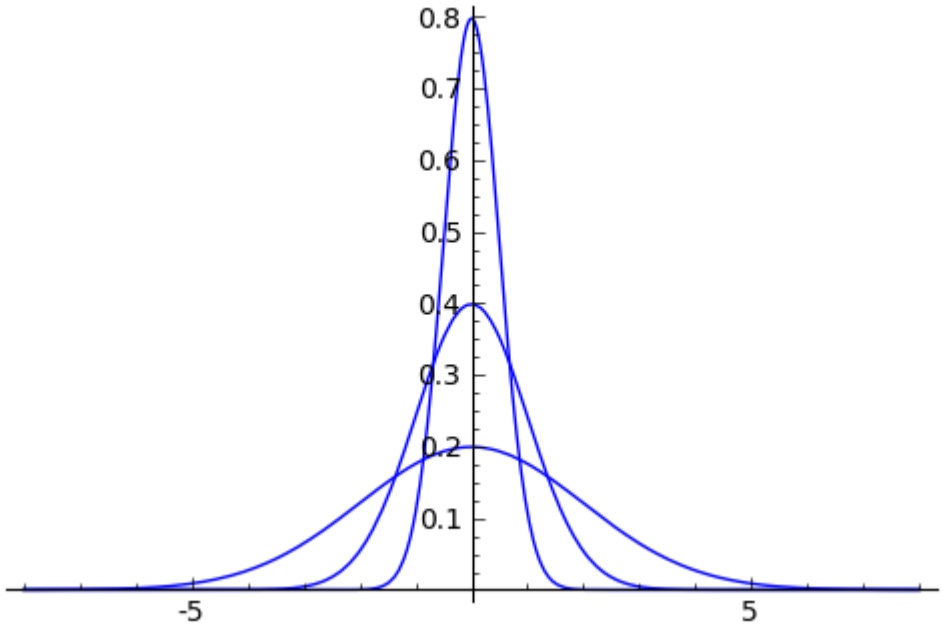
- **Gaussian noise model**: The noise at pixel x, y, N(x, y), is a random variable.

- Simplest assumption: N(x, y) is "white noise", distributed independently of the noise at other pixels.

Normal (Gaussian) distribution

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

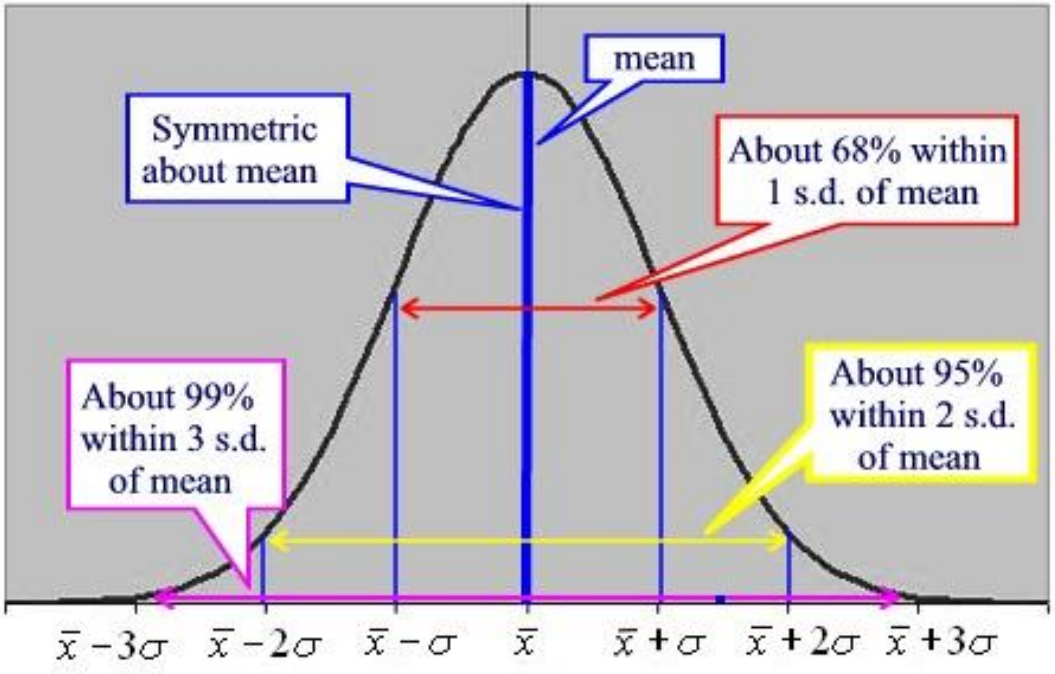
Gaussian, or normal, distribution
with mean 0 and standard deviation σ



Three Gaussians, with σ = 0.5, 1, 2 (σ = 0:5 is the narrowest).

More on normal distributions

68% of the distribution lies within one standard deviation (std) 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.



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

Modeling Gaussian noise

- random.gauss(mu, sigma) returns a number distributed according to a Gaussian distribution: mean μ and std σ

- We will use $\mu = 0$, and a default value $\sigma = 10$. 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]
```

Add Gauss noise to image

```
#####################################
## Adding noise to images,
## for testing noise reduction
#####################################

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 = mat.copy()
    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
```

Examples

```
new10 = add_gauss (abbey)
new20 = add_gauss (abbey, sigma =20)
new50 = add_gauss (abbey, sigma =50)
news = join( new10, new20, new50, direction='h' )
news.display()
```

Denoising algorithms

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

- Denoising by Local means
- Denoising by Local Medians
- Denoising by Non local means

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

Local denoising

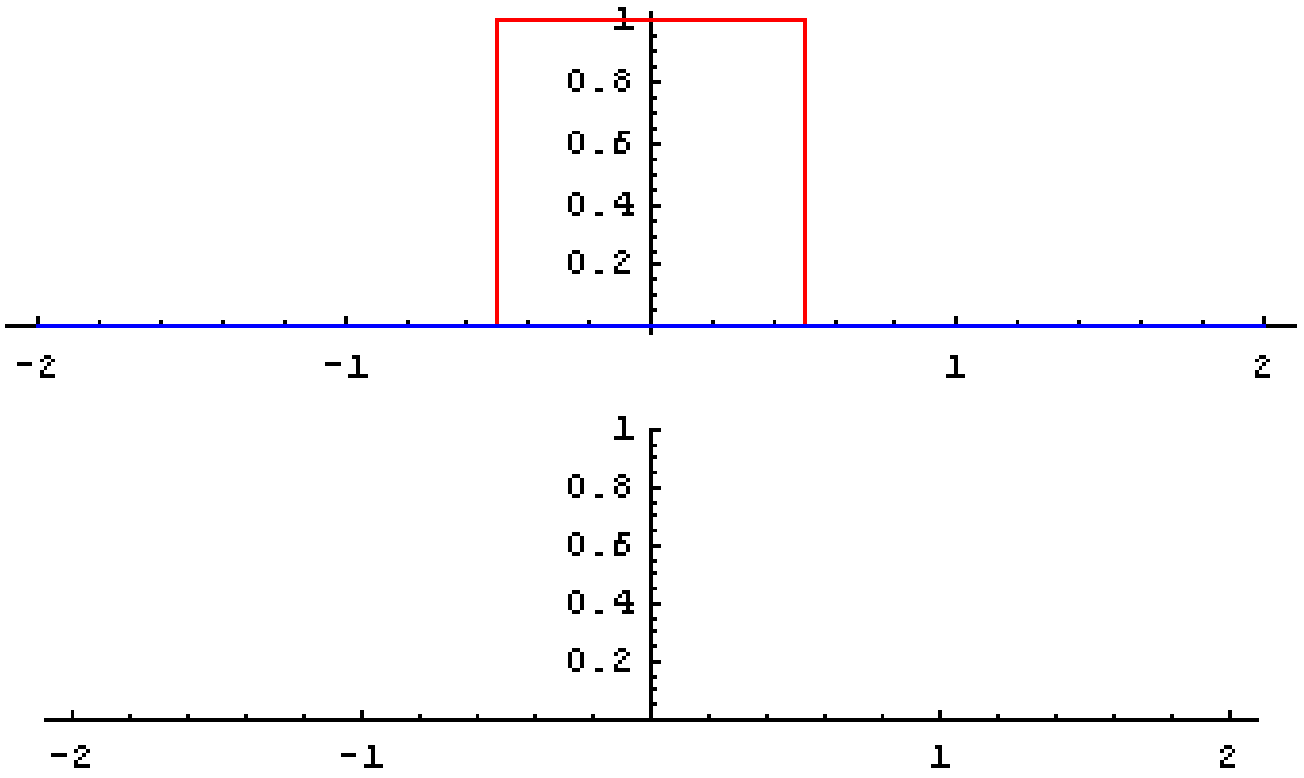
- Neighborhood of the pixel x, y is defined as the set of all pixels whose coordinates are close to x, y

- 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
- Called “convolution”

Convolution (brief detour)



Source: Wikipedia

Remember the peaks problem?

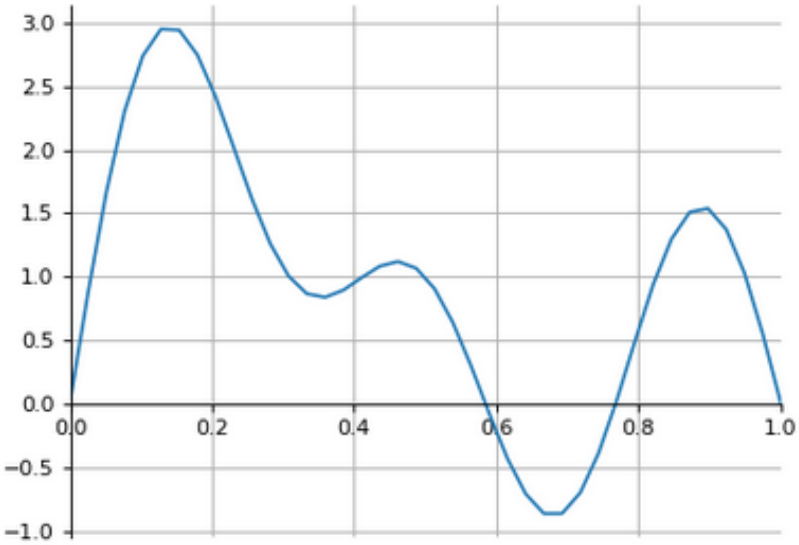
(1st week)

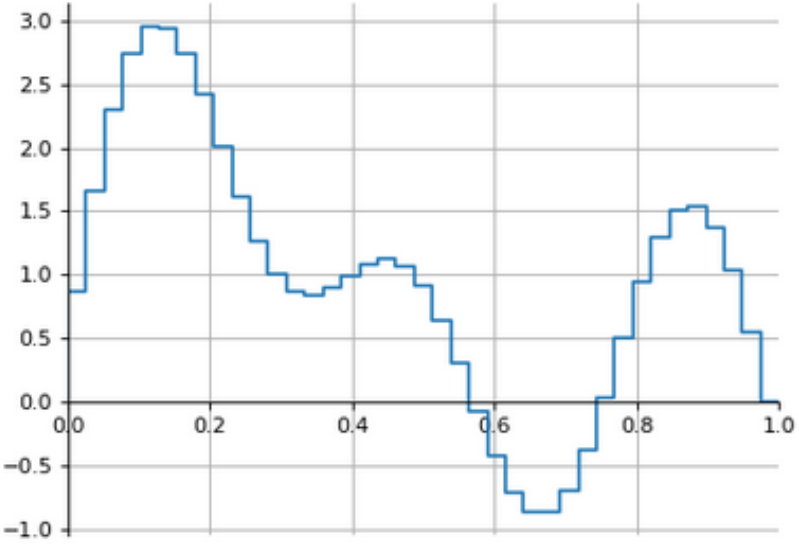
Quantization

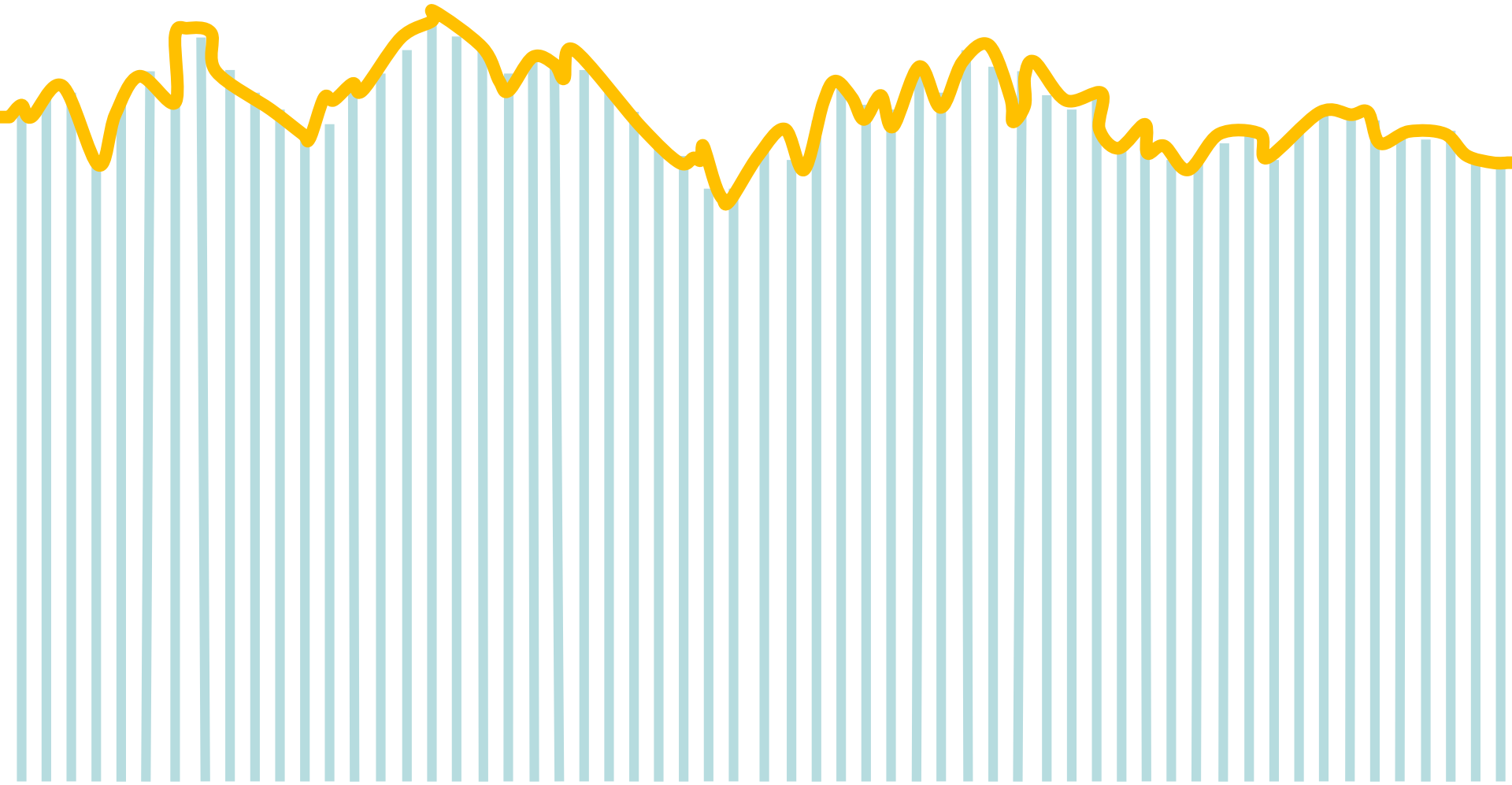
"Quantization is the process of constraining an input from a continuous or otherwise large set of values (such as the real numbers) to a discrete set (such as the integers)“ (Wikipedia)

Examples:

- Mathematical integration
- Signal processing (e.g., audio/image, time/space/color)







The Peaks Problem

Input: A sequence S of real numbers of length n.

Output: All the triples (subsequences of size 3) of S such that the middle number is the largest of the three.

The sliding window mechanism

40	50	22	29	55	43	69	41	44	47	65	73	62	59	71

The computer's
"point of view"

40	50	22	29	55	43	69	41	44	47	65	73	62	59	71

Moving sliding window from 1D to 2D

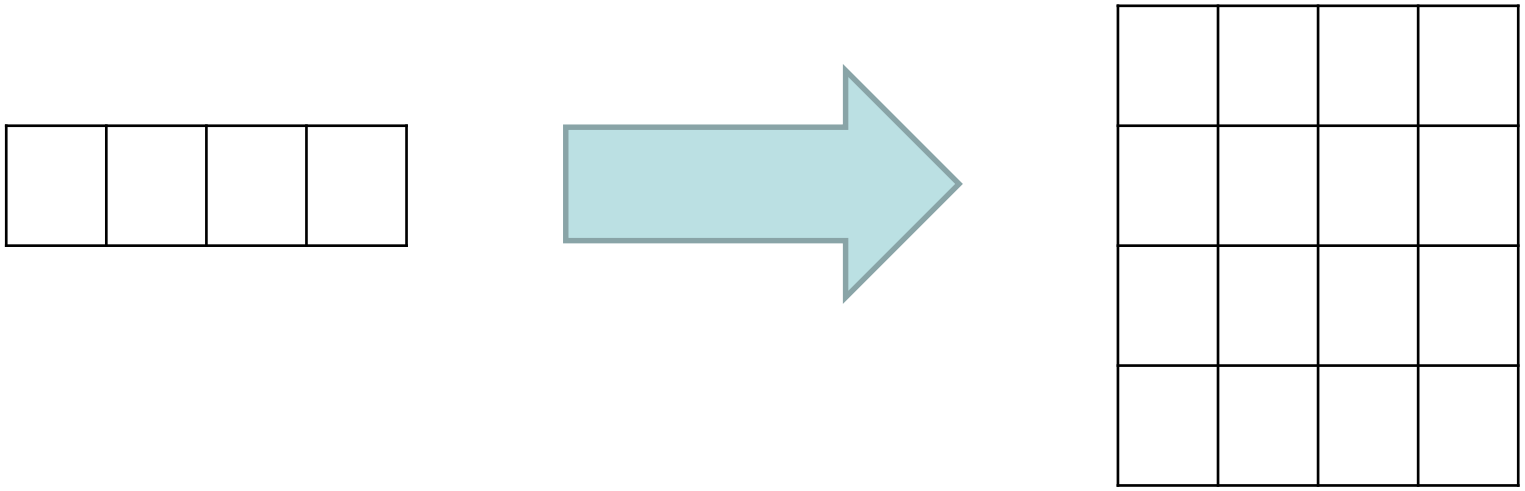
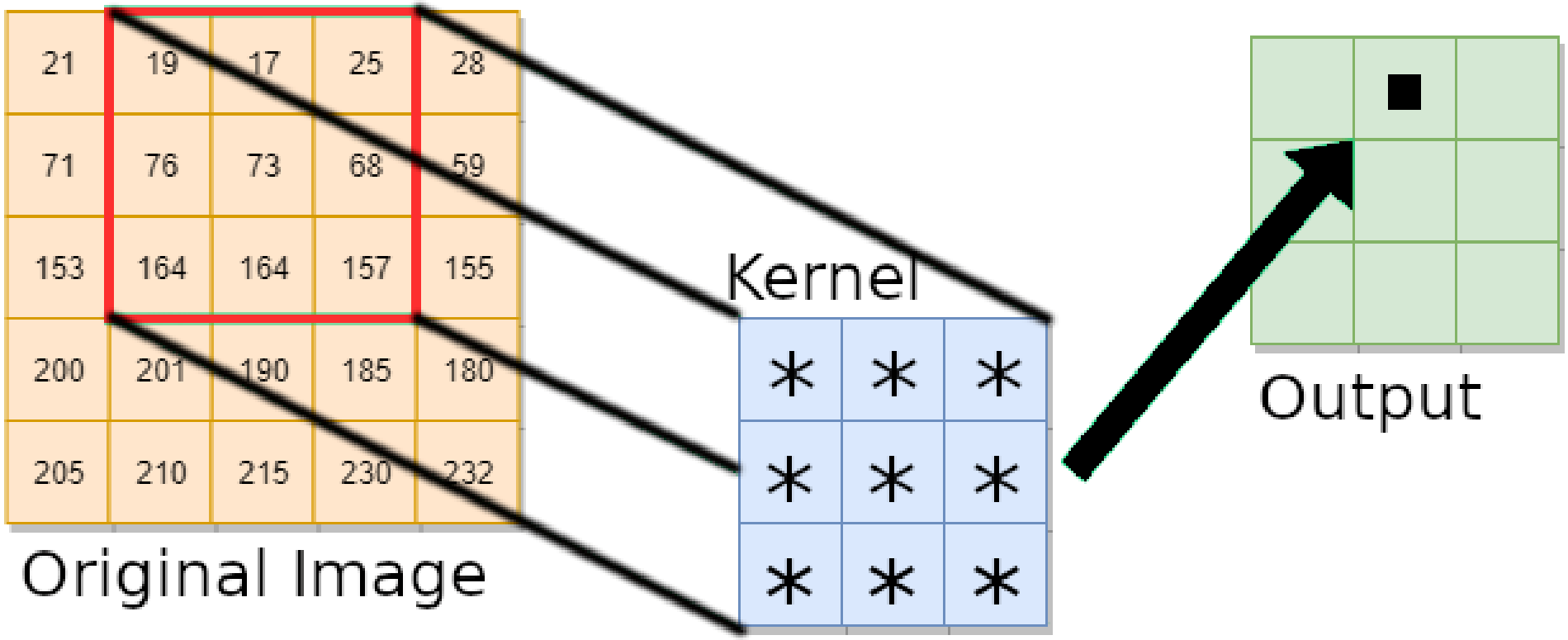


Image Processing &
Computer Vision

Image convolution



https://ulhpc-tutorials.readthedocs.io/en/latest/cuda/exercises/convolution/

Example

Input image

Convolution Kernel

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Feature map

https://developer.nvidia.com/discover/convolution

Example application: denoising

Original

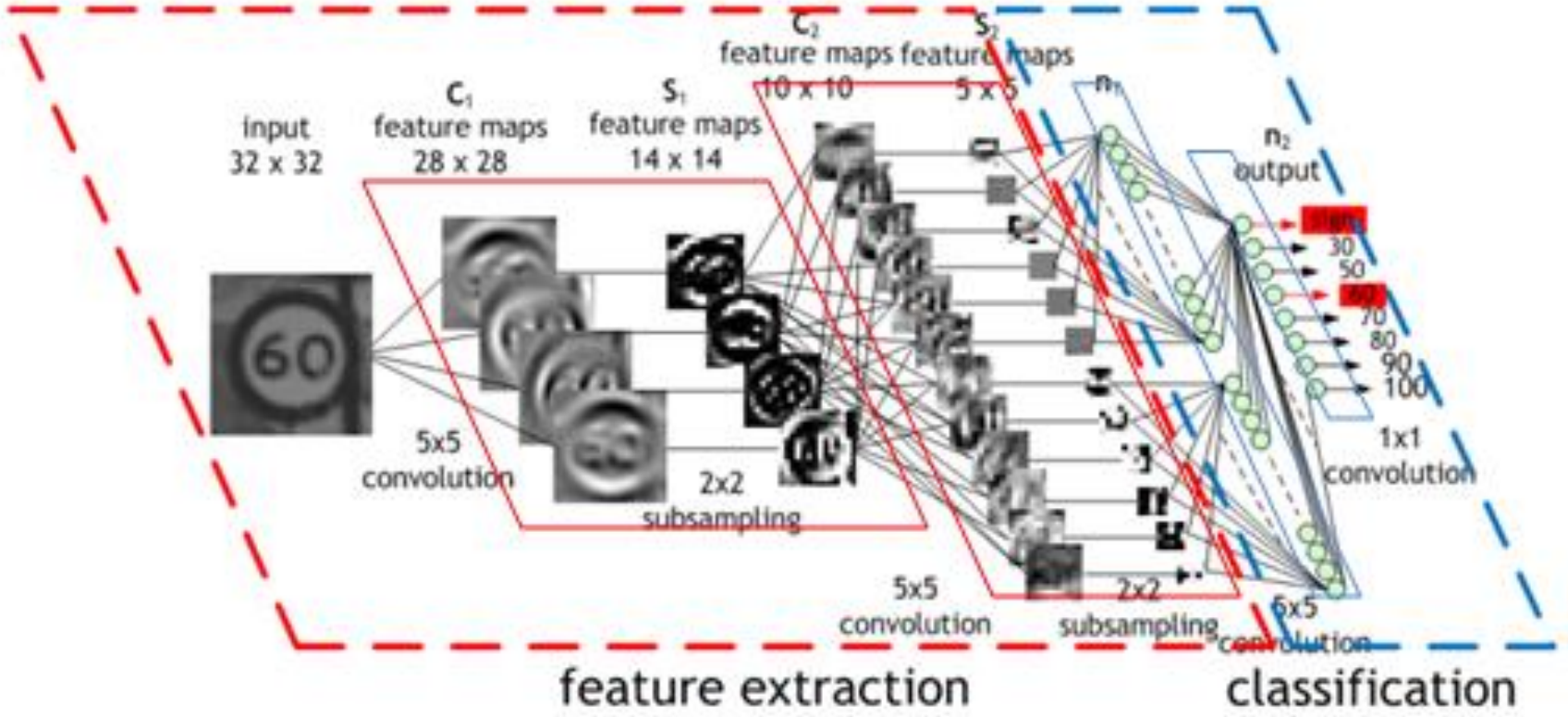
Result

Example application: edge detection

Original

Result

Convolutional Neural Network



https://developer.nvidia.com/discover/convolutional-neural-network

Back to local denoising

- Neighborhood of the pixel x, y is defined 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
- Called “convolution”

Denoising by local means

- Replace the observed value M(x, y), by the average of the observed values in its neighborhood

- Make sure not to 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):
    ''' Apply op to every pixel.
        op is a local operator on a square neighbourhood
        of size 2k+1 X 2k+1 around a pixel '''
    n,m = mat.dim()
    res = mat.copy()  # brand new copy of A
    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
```

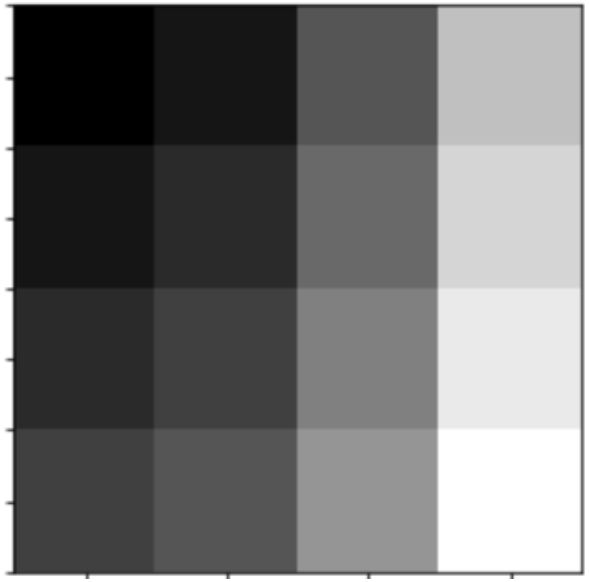
Local means

```
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

```
mat = Matrix (4 ,4)
for i in range (4):
    for j in range (4):
        mat[i,j] = i + (j**2)
for i in range (4):
    print ([mat[i,j] for j in range (4)])
mat.display()
```

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

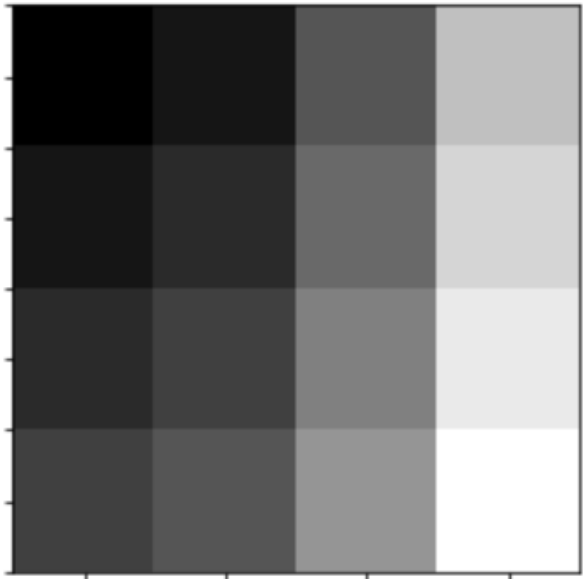


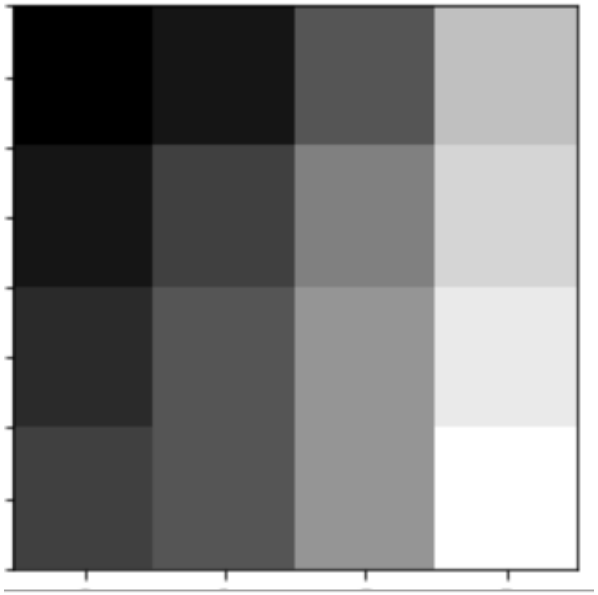
Local means: a synthetic example

```
mat2 = local_means (mat)
for i in range (4):
    print ([mat2[i,j] for j in range (4)])
mat2.display()
```

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

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

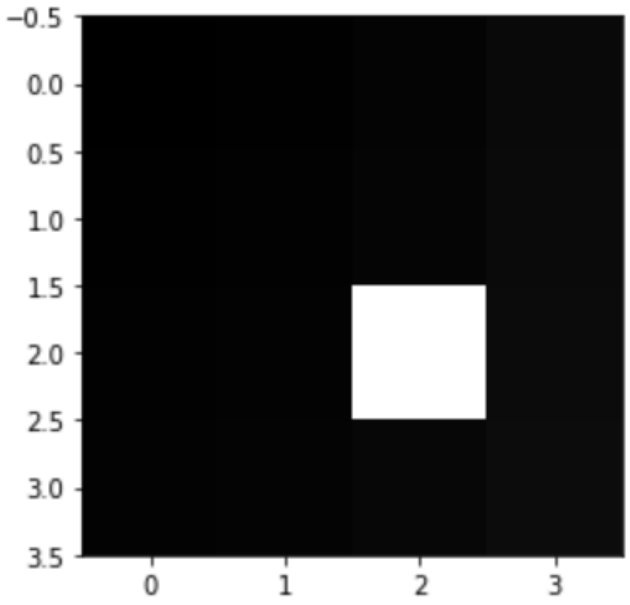




Local means: another synthetic example

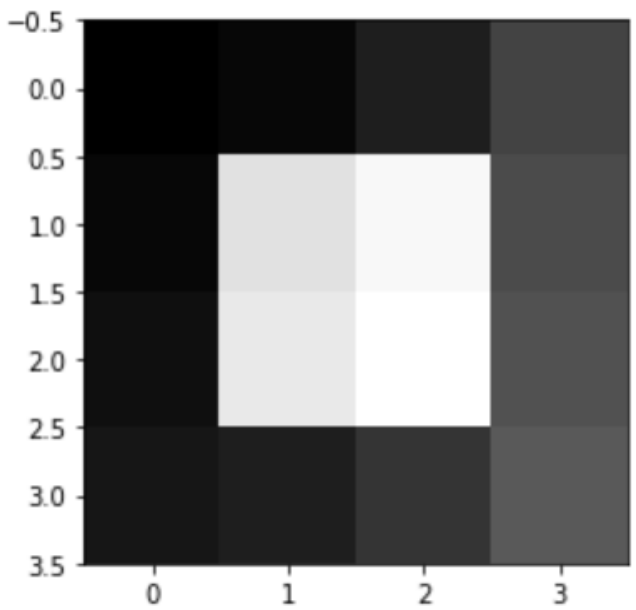
```
mat[2,2] = 255
for i in range (4):
   print ([mat[i,j] for j in range (4)])
mat.display()
```

```
[0, 1, 4, 9]
[1, 2, 5, 10]
[2, 3, 255, 11]
[3, 4, 7, 12]
```




```
mat2 = local_means (mat)
for i in range (4):
    print ([mat2[i,j] for j in range (4)])
mat2.display()
```

```
[0, 1, 4, 9]
[1, 30, 33, 10]
[2, 31, 34, 11]
[3, 4, 7, 12]
```



averaging is highly affected by "outliers"

Denoising by local means: motivation

- If the pixel x, y resides in a smooth portion of the image, then averaging will not change it significantly

- 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 the noise contribution

Local means: weighted variants

- Averaging can be expressed as the matrix *Frobenius inner product* - sum of element by element product

$$\sum A_{i,j} \cdot B_{i,j}$$

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

- Put more weight close to the central pixel:

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

- Maintains more of the original signal, with smaller noise reduction

Denoising by local means: limitations

- X Pixel that does not reside in a smooth portion of the image, does not preserve the signal → blurred edges

- X Sensitivity to spurious extreme values, example: salt & pepper noise

  - For example, suppose the original area of the image has intensity level of 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.

  - $\hat{T}$(x-1, y-1) will be corrected to 216. Each of the other 8 pixels containing x-1, y-1 in their neighborhood, will also exhibit such "correction", which is undesirable.

Salt & pepper noise model

Extreme gray levels (white and black) at random and independently in a small number of pixels

Original image

Salt & pepper noise

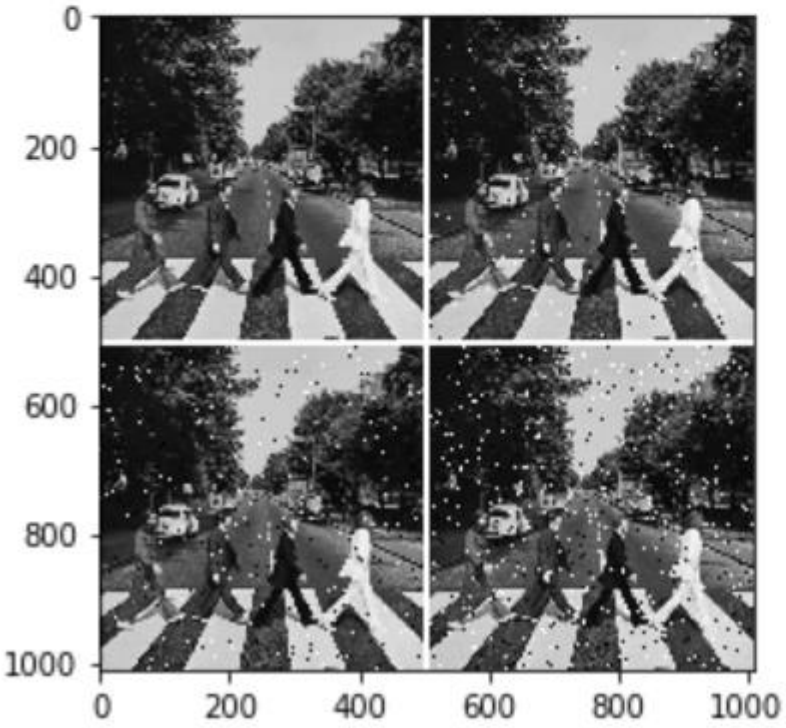
Gaussian noise

Implementing salt & pepper 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 = mat.copy()
    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

```
sp1 = add_SP(abbey)
sp2 = add_SP(abbey, p =0.02)
sp5 = add_SP(abbey, p =0.05)
joined1 = join(abbey, sp1, direction='h')
joined2 = join(sp2, sp5, direction='h')
joined = join(joined1, joined2, direction = 'v')
joined.display()
```



Denoising by local medians

- Replace the observed value M(x, y), by the median of the observed values in a neighborhood of (x, y).

✓ The median preserves **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

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

```
def median(lst):
    sort_lst = sorted(lst)
    n = len(sort_lst)
    if n%2==1:     # odd number of elements. well defined median
        return sort_lst[n//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)
```

Local Medians: A Synthetic Example

```
mat = Matrix (4 ,4)
for i in range (4):
    for j in range (4):
         mat[i,j] = i + (j**2)
for i in range (4):
    print ([mat[i,j] for j in range (4)])
mat.display()
```

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

```
matb = local_medians(mat)
for i in range (4):
    print ([matb[i,j] for j in range (4)])
matb.display()
```

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

Local medians: another synthetic example

```
mat[2,2]=255
print("Original:")
for i in range (4):
    print ([mat[i,j] for j in range (4)])
matb = local_means(mat)
print("Local Means:")
for i in range (4):
    print ([matb[i,j] for j in range (4)])
matc = local_medians(mat)
print("Local Medians:")
for i in range (4):
    print ([matc[i,j] for j in range (4)])
```

```
Original:
[0, 1, 4, 9]
[1, 2, 5, 10]
[2, 3, 255, 11]
[3, 4, 7, 12]

Local Means:
[0, 1, 4, 9]
[1, 30, 33, 10]
[2, 31, 34, 11]
[3, 4, 7, 12]

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

The median ignores outliers..

Complexity of local means and local medians

- Suppose the image dimensions are n-by-m.
- The number of windows to process: $O(n \cdot m)$ ($k << m, n$).
- For every window, compute the average or median.
- 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 the median**: sorting ➔ $O(k^2 \log k^2) = O(k^2 \log k)$ steps. Faster median finding exists - linear in k. Computing median can therefore be done in $O(k^2)$ steps too.
- All in all, $O(k^2nm)$ steps (with better median implementation).

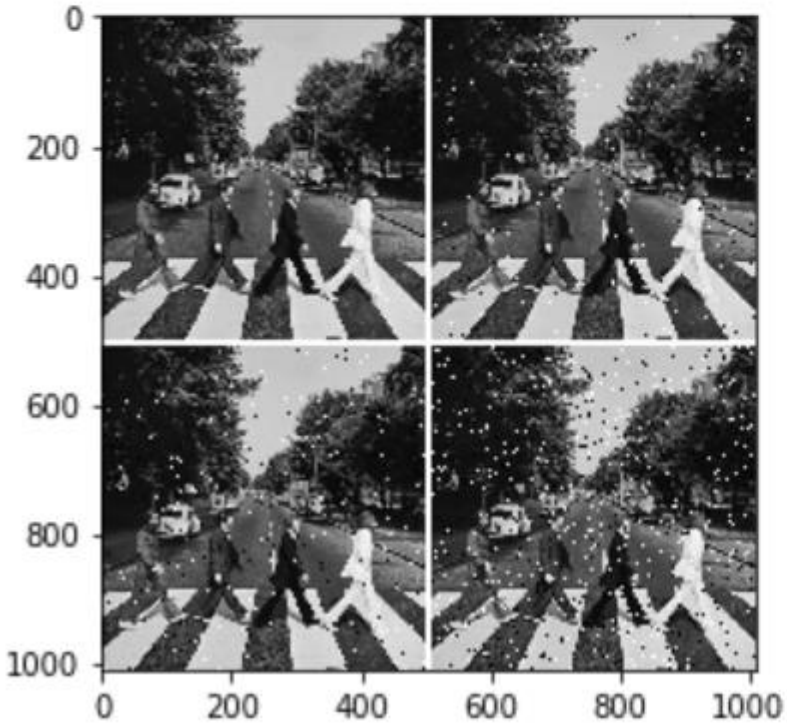
Putting local means/medians to the test

Let’s test our methods

Try to decide which is better (and when)

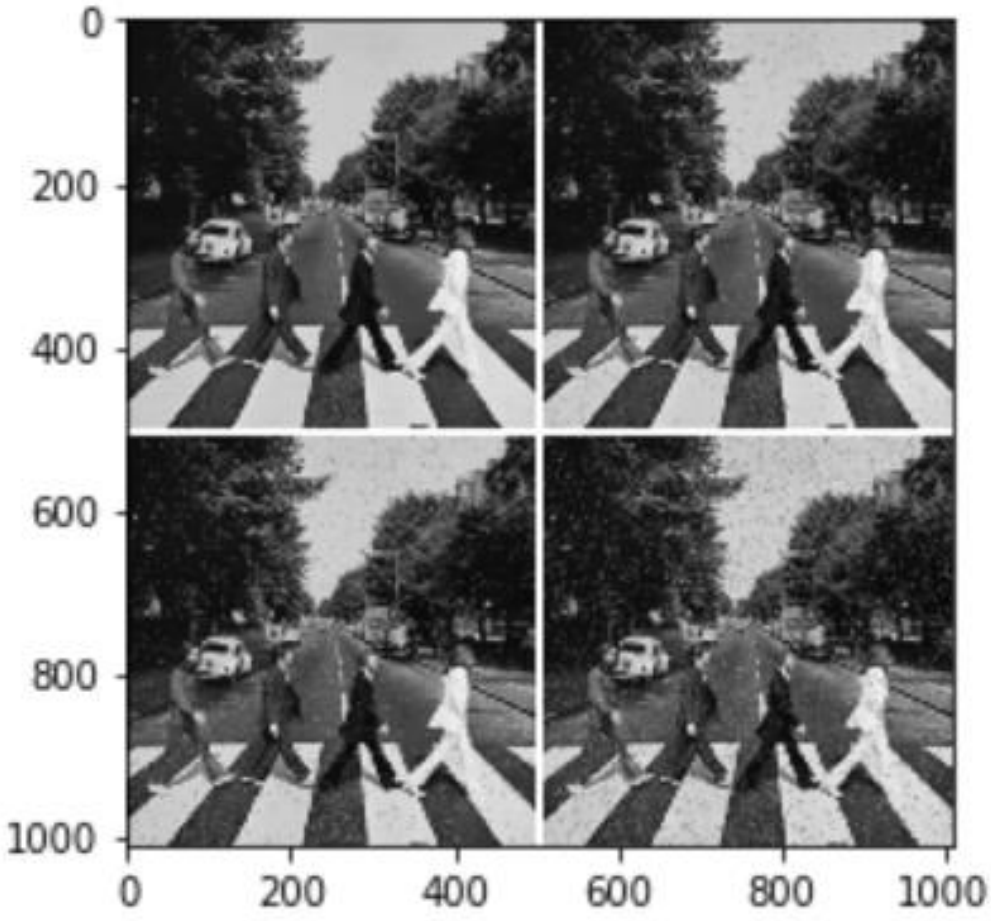
Testing local means/medians with SP noise

```
sp1 = add_SP(abbey)
sp2 = add_SP(abbey, p =0.02)
sp5 = add_SP(abbey, p =0.05)
joined1 = join(abbey, sp1, direction='h')
joined2 = join(sp2, sp5, direction='h')
joinedAbbeySP = join(joined1, joined2, direction = 'v')
joinedAbbeySP.display()
```



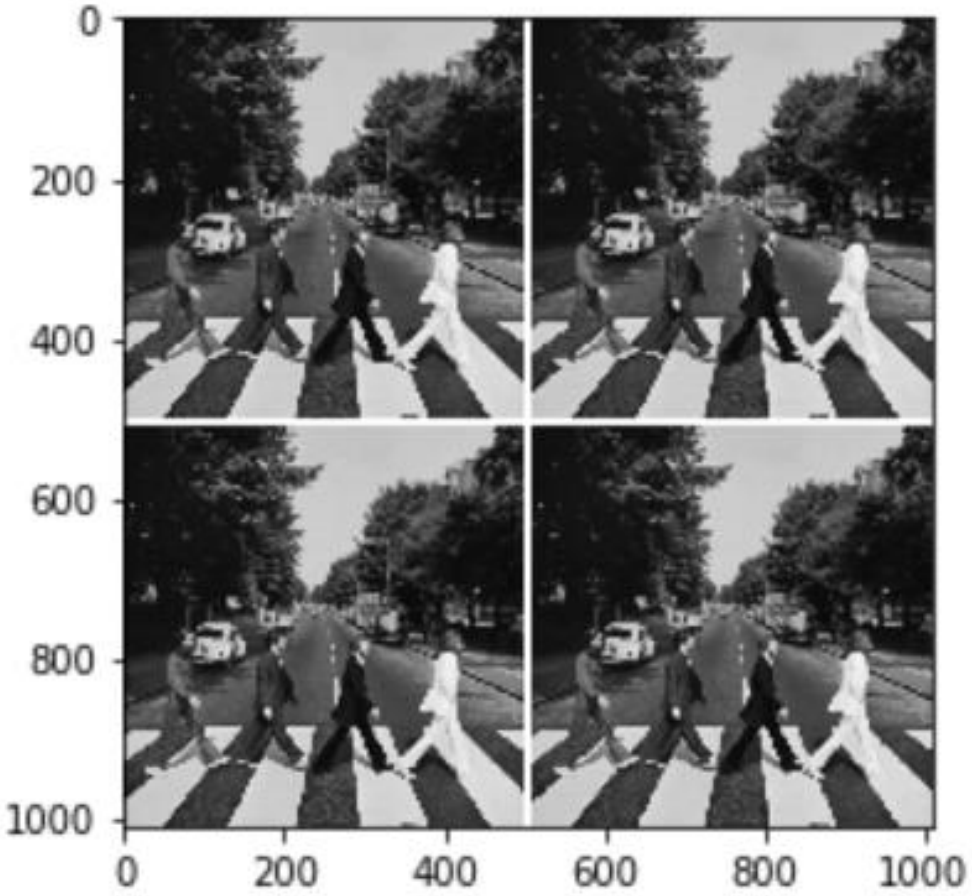
Testing local means with SP noise

```
#this code will take a few seconds to run
denoised_by_means = local_means(joinedAbbeySP)
denoised_by_means.display()
```



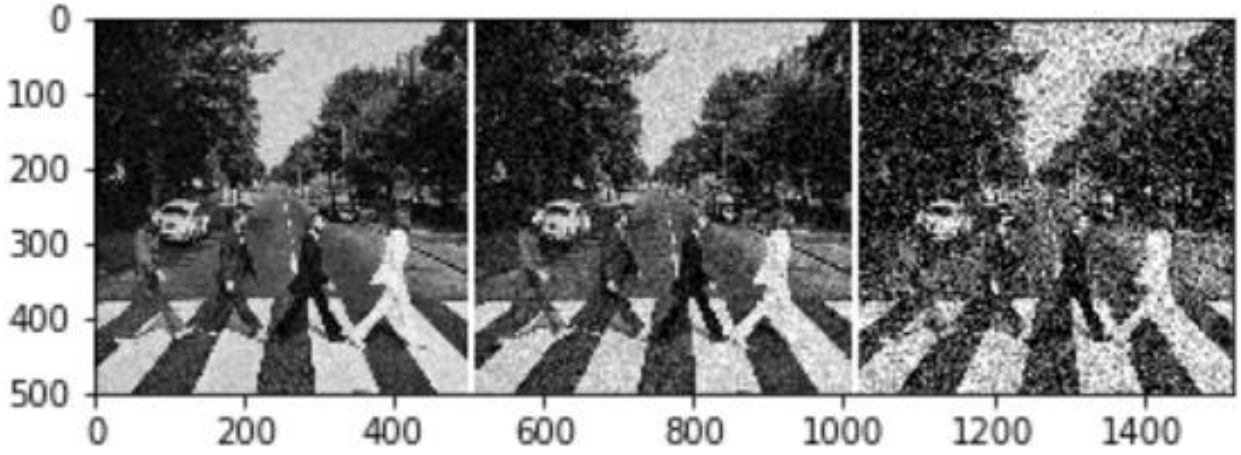
Testing local medians with SP noise

```
#this code will take a few seconds to run
denoised_by_medians = local_medians(joinedAbbeySP)
denoised_by_medians.display()
```



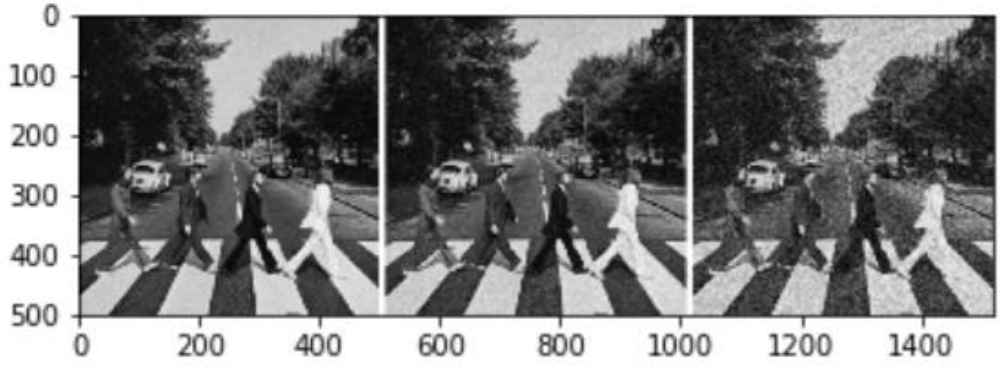
Testing local means/medians with Gaussian noise

```
new10 = add_gauss (abbey)
new20 = add_gauss (abbey, sigma =20)
new50 = add_gauss (abbey, sigma =50)
joinedAbbeyGauss = join( new10, new20, new50, direction='h' )
joinedAbbeyGauss.display()
```

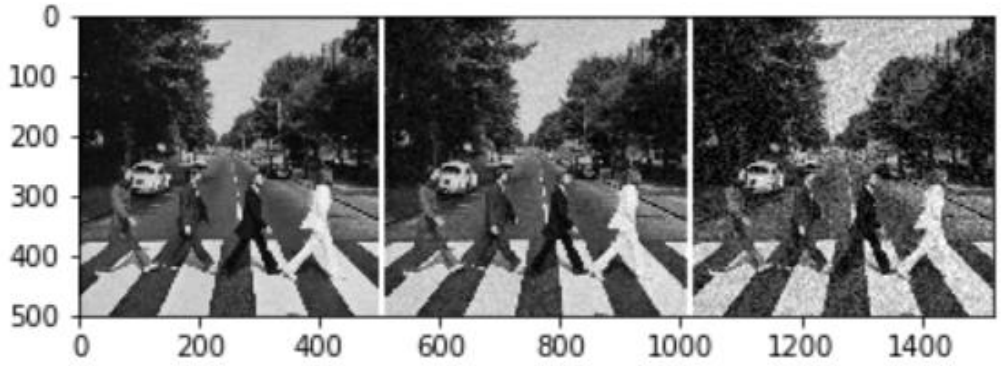


Testing local means/medians with Gaussian noise

```
#this code will take a few seconds to run
Gauss_denoised_by_means = local_means(joinedAbbeyGauss)
Gauss_denoised_by_means.display()
```

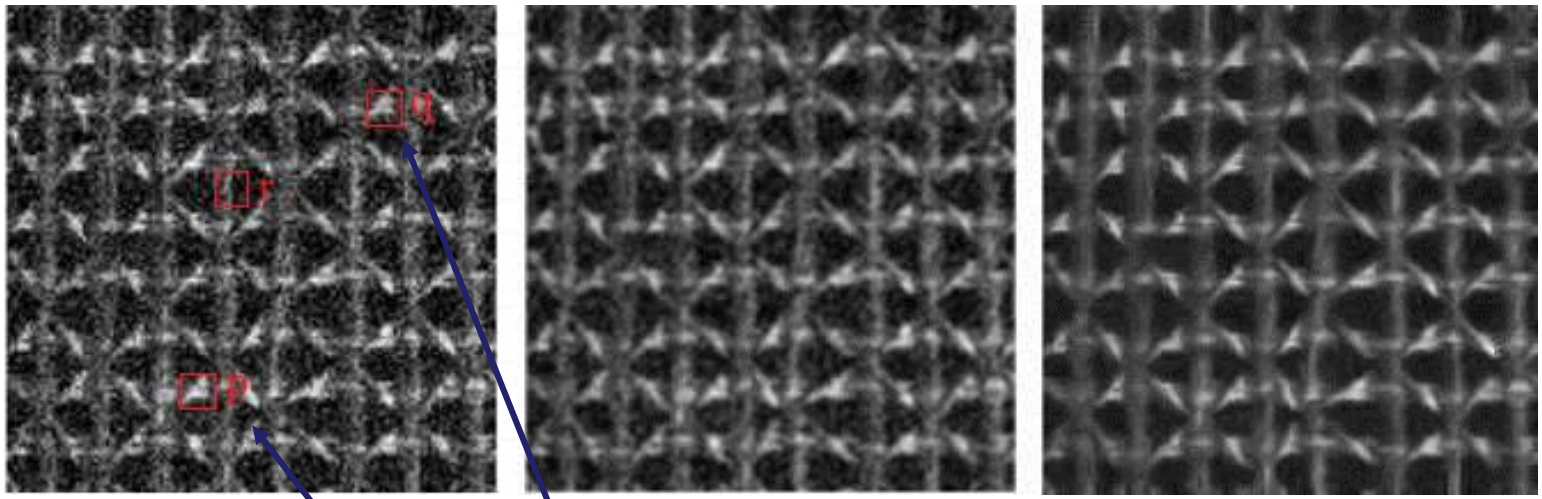



```
#this code will take a few seconds to run
Gauss_denoised_by_medians = local_medians(joinedAbbeyGauss)
Gauss_denoised_by_medians.display()
```



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 | Local means | Non-local means means

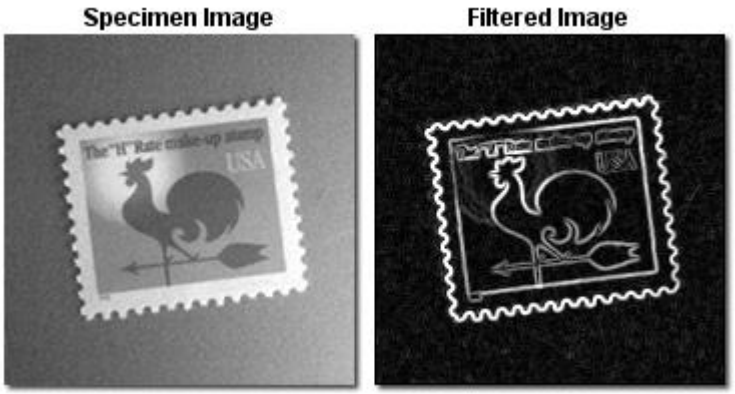
- Windows centered at p and q are similar, but not to the one centered at r.

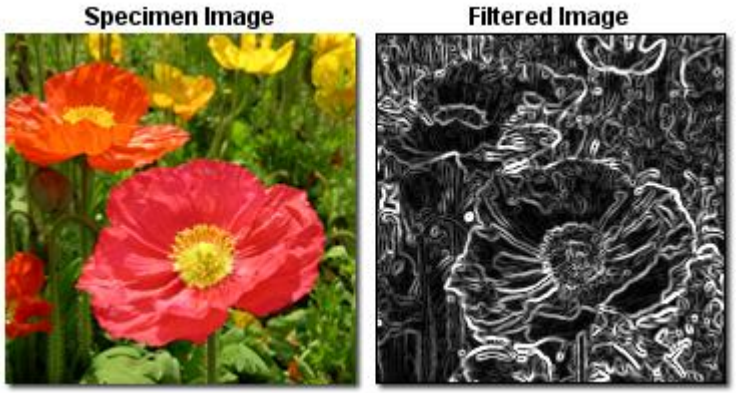
Denoising by non-local means

- 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.

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:

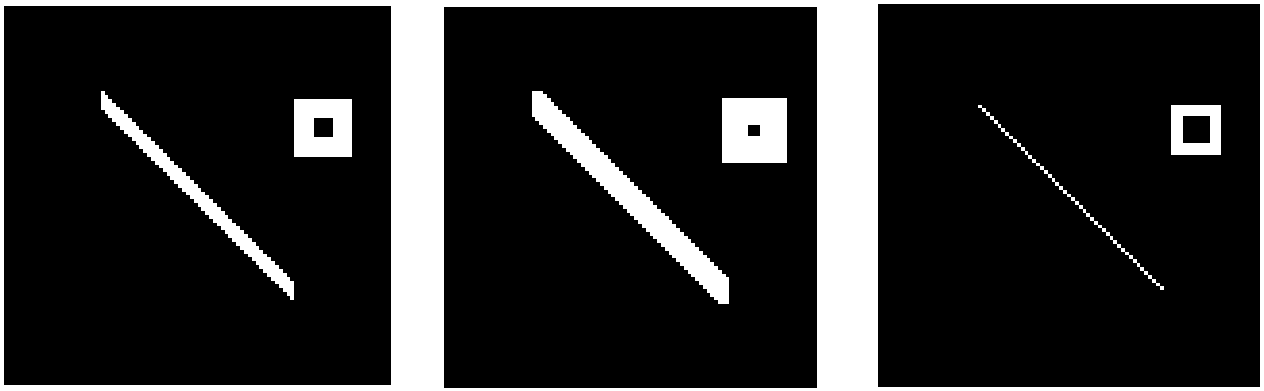
http://micro.magnet.fsu.edu/primer/java/digitalimaging/russ/sobelfilter/index.html

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.

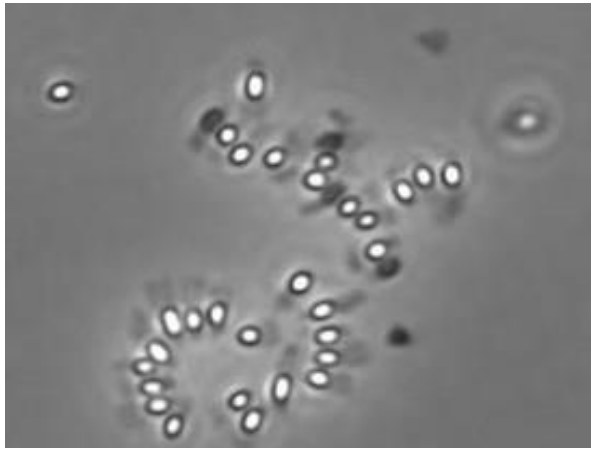
- Dilation - the addition of pixels to the periphery of features.
  - enlarges foreground areas, and holes shrink.



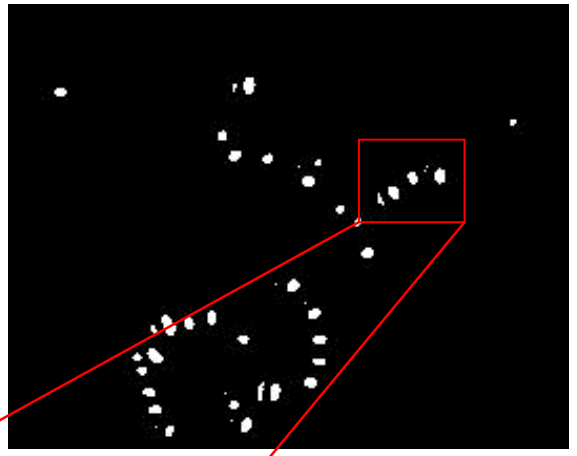
Original Dilation Erosion

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

Erosion and dilation - example

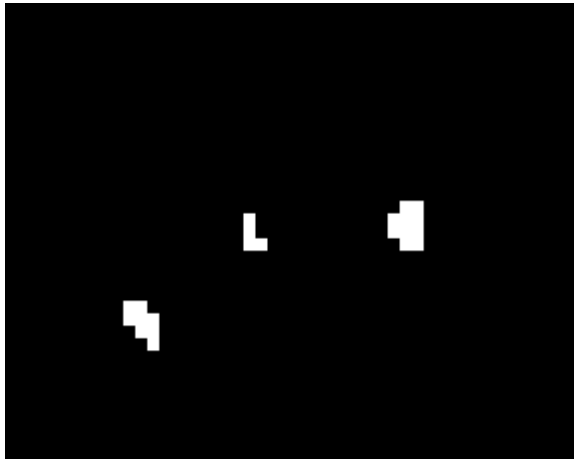


Binary segmentation
(th=200)

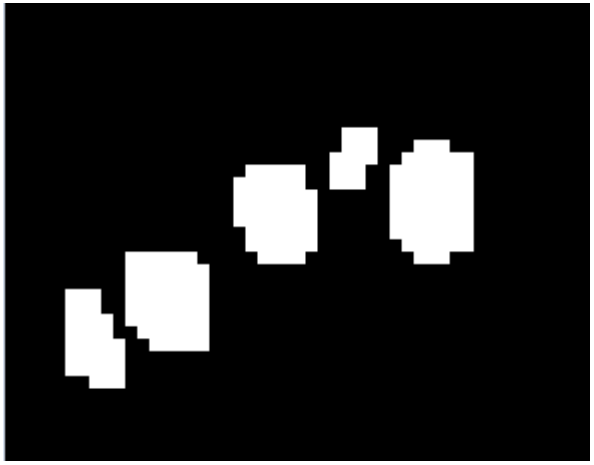


A microscope slide containing Clostridium botulinum cells and spores. Spores appear bright with dark boundaries (the spore coat). Vegetative cells were stained to provide contrast, and thus appear dark

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



Erosion



Dilation

Open CV

Open Source Computer Vision Library

https://opencv.org/

Install

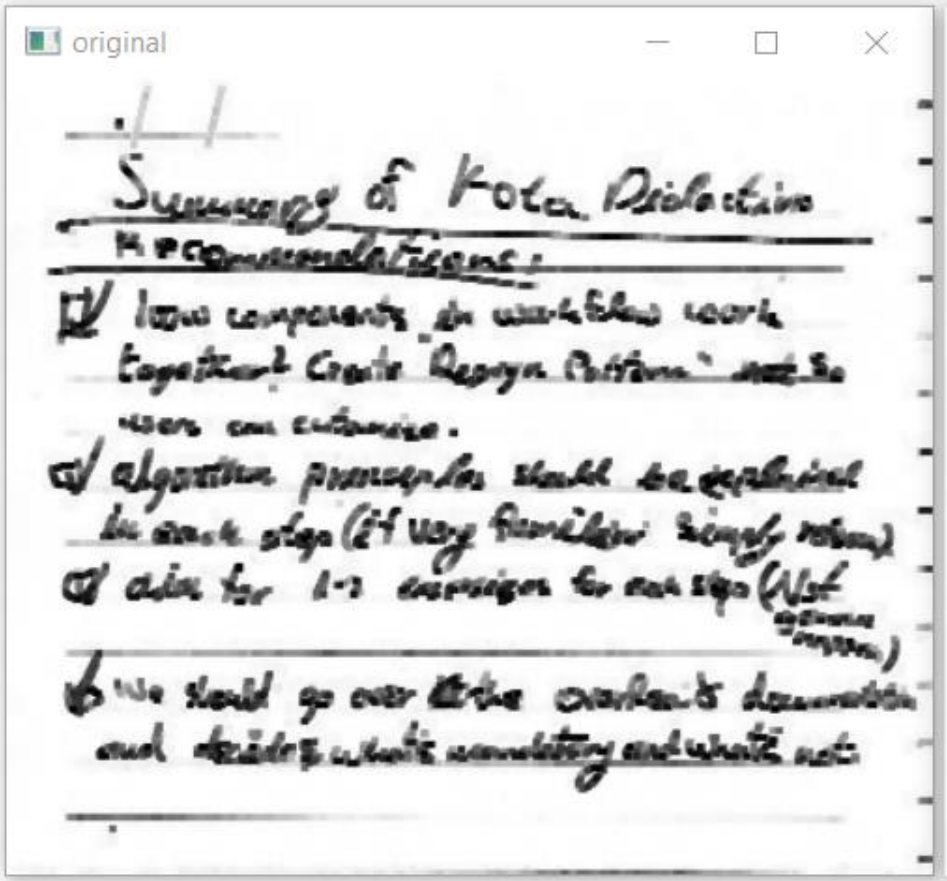
```
pip install opencv-python
```

```
Collecting opencv-python
  Downloading opencv_python-4.5.4.60-cp38-cp38-win_amd64.whl (35.1 MB)
Requirement already satisfied: numpy>=1.17.3 in c:\users\assaf\anacond
Installing collected packages: opencv-python
Successfully installed opencv-python-4.5.4.60
Note: you may need to restart the kernel to use updated packages.
```

Dilation with Open CV

```
import numpy as np
import cv2

circle_ker_3x3 = np.array([
    [0, 255, 0],
    [255, 255, 255],
    [0, 255, 0]
], np.uint8)

# fixing the bad text scan
bad_txt = cv2.imread('Scan1_eroded.jpg', 0)
fixed_text = cv2.dilate(bad_txt, circle_ker_3x3, iterations=1)
cv2.imshow("original", bad_txt)
cv2.imshow("fixed text", fixed_text)
cv2.waitKey(0)
```

Output


original
Summary of Kota Dislocation
Recommendations:
☑ low components in workflow work together & Create Design Pattern & so user can customize.
☑ algorithm principles should be explained in each step (if very familiar simply reload)
☑ aim for 1-3 corrections to our steps (Not grammar)
☐ We should go over the earlier documentation and decide what is mandatory and what not

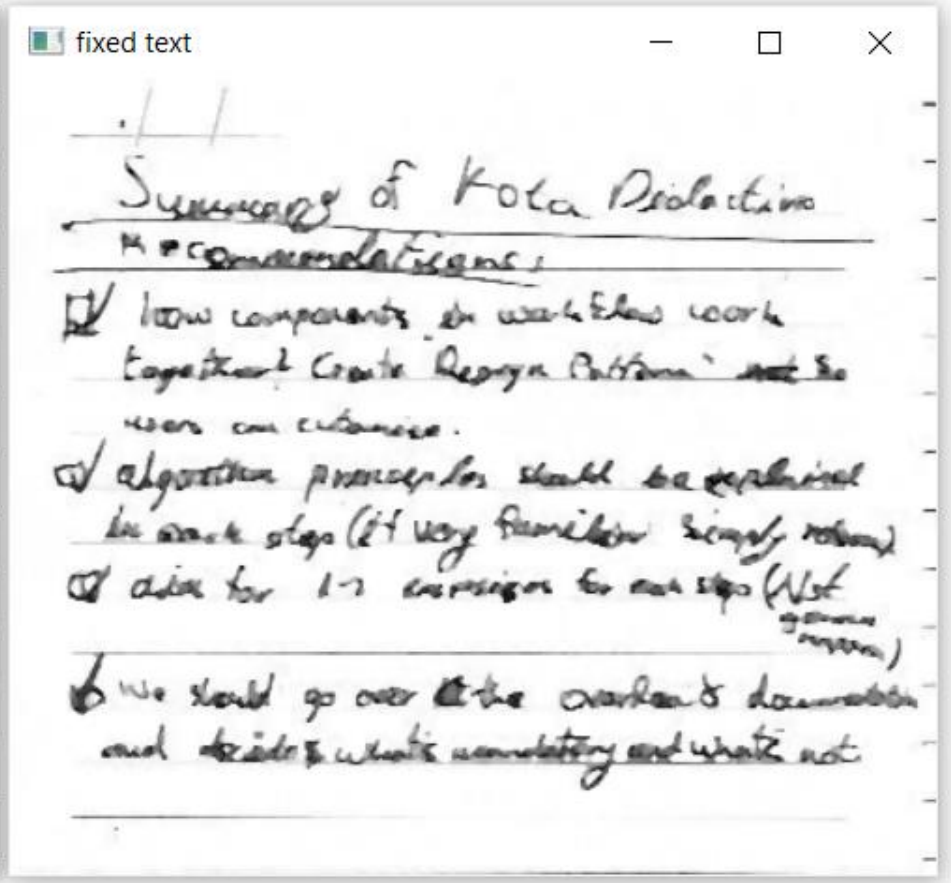
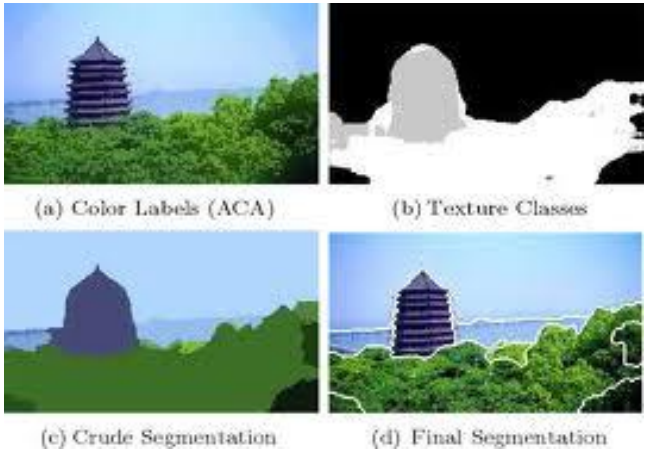
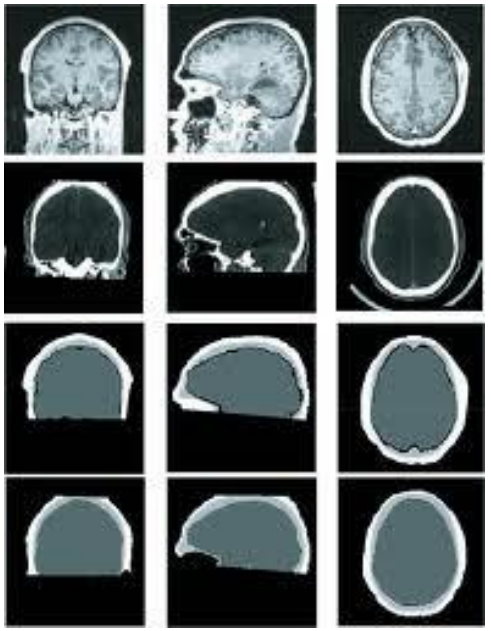

fixed text
Summary of Kota Dislocation
Recommendations:
☑ low components in workflow work together & Create Design Pattern & so user can customize.
☑ algorithm principles should be explained in each step (if very familiar simply reload)
☑ aim for 1-3 corrections to our steps (Not grammar)
☐ We should go over the earlier documentation and decide what is mandatory and what not

Image segmentation





Segmentation

- Partitioning a digital image into multiple segments.

Source:

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

- Goal: to simplify the representation for understanding and \ or analysis.
- Used to locate objects and boundaries (lines, curves, etc.). Usually, the first step in more complicated procedures: object recognition, shape analysis , tracking…
- Examples: locating tumors in medical images; identifying objects in satellite images (roads, forests, crops, etc.).

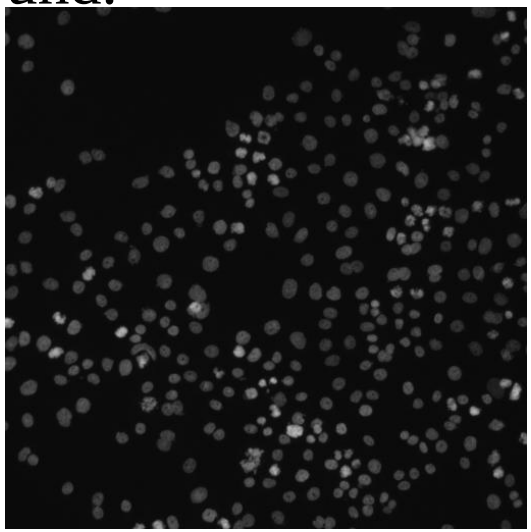
Image Segmentation Algorithms

- Thersholding
- Clustering
- Region growing
- Compression-based methods
- Histogram-based methods
- Model-based methods
- Etc.

Binary segmentation by thresholding

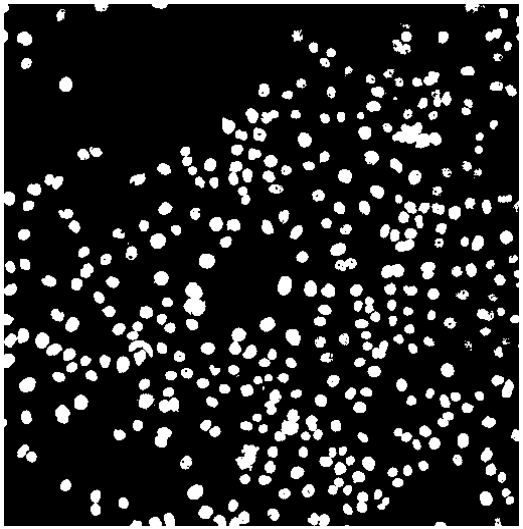
- The simplest segmentation method: apply a threshold to turn a gray-scale image into a binary image (BW)

- 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

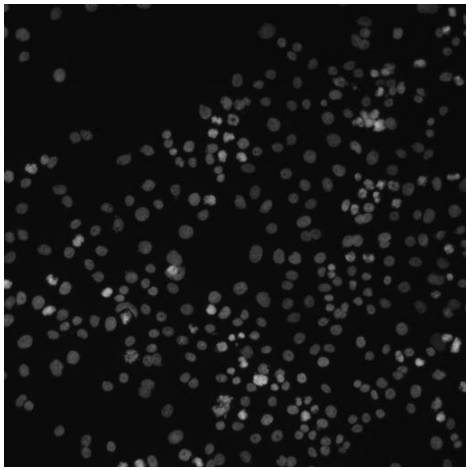
Implementing threshold-based segmentation

```
def segment(mat, threshold):
    ''' Binary segmentation of image (matrix)
        using a threshold
    '''
    n,m = mat.dim()
    out = Matrix(n,m)

    for x in range(n):
        for y in range(m):
            if mat[x, y] >= threshold:
                out[x,y] = 255 #white
            else:
                out[x,y] = 0 #black

    return out
```

Picking a threshold: example

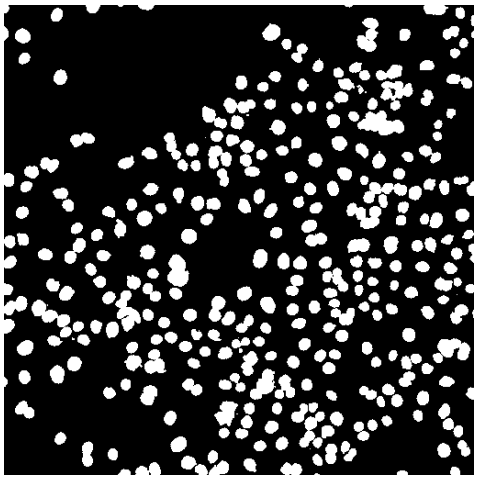
- The key is to select the appropriate threshold
- Which one is the best here?

Original

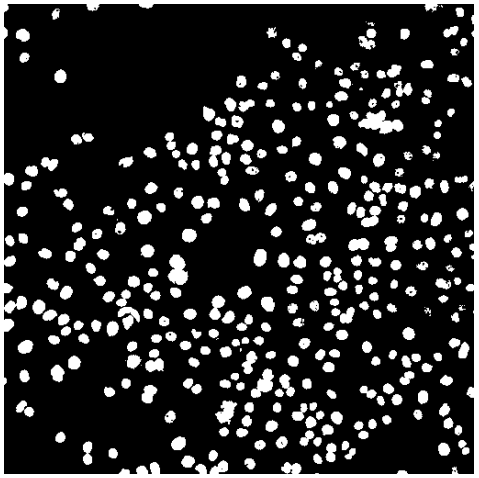


- When the threshold is too low (20 in this case) areas in the image where cells are densely populated become bulbs.
- When it is too high (60) some cells are lost.

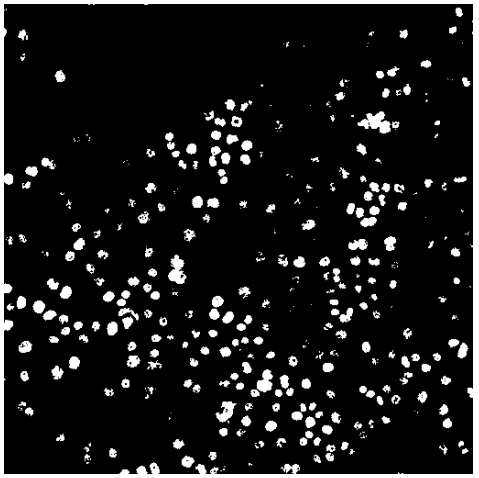
Threshold = 20



Threshold = 40

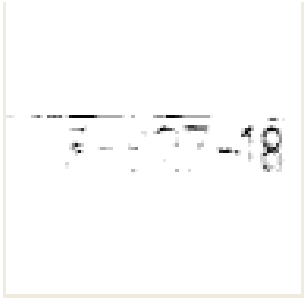


Threshold = 60

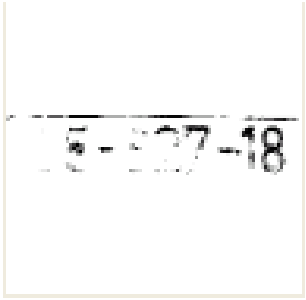


Picking a threshold: another example

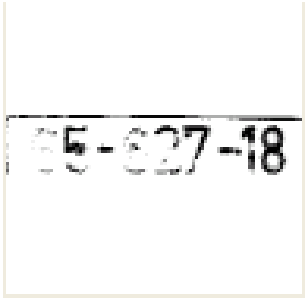
out_20.bmp



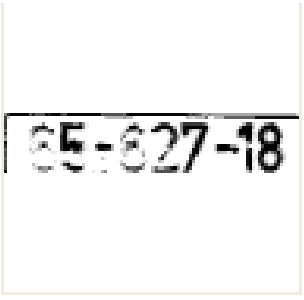
out_40.bmp



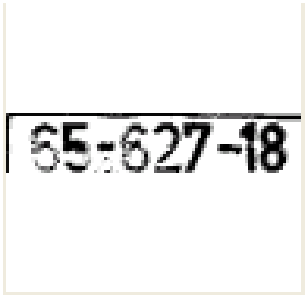
out_60.bmp



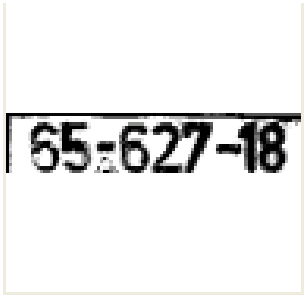
out_80.bmp



out_100.bmp



out_120.bmp



out_140.bmp

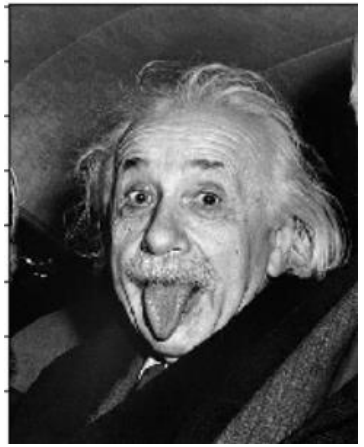
out_160.bmp

out_180.bmp

Picking a threshold: another example

- We want: white = Einstein, black = background
- When the threshold is too low (50) we cannot separate face from background.
- When it is too high (150) some areas are lost (e.g. parts of the hair), but still good.
- With 100 – some noise in the background (left)
- How can we pick a good threshold automatically?

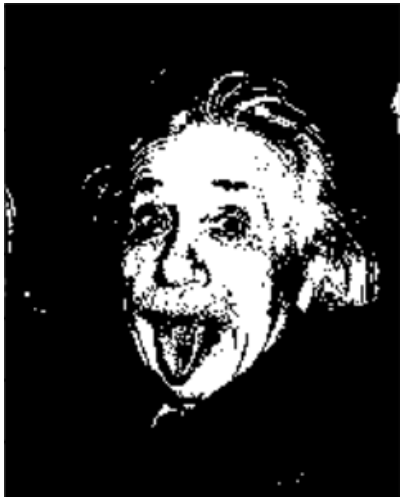
Original



Threshold = 50

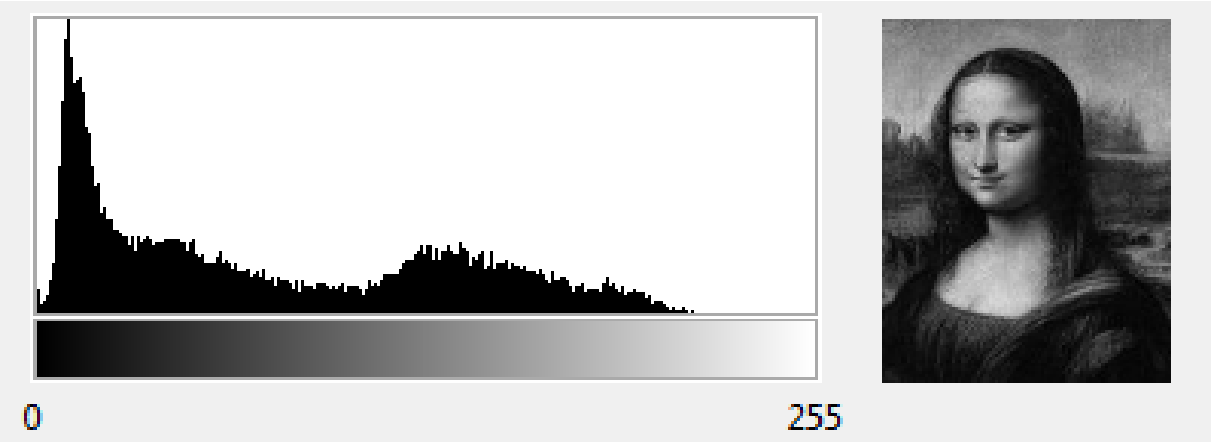
Threshold = 100

Threshold = 150



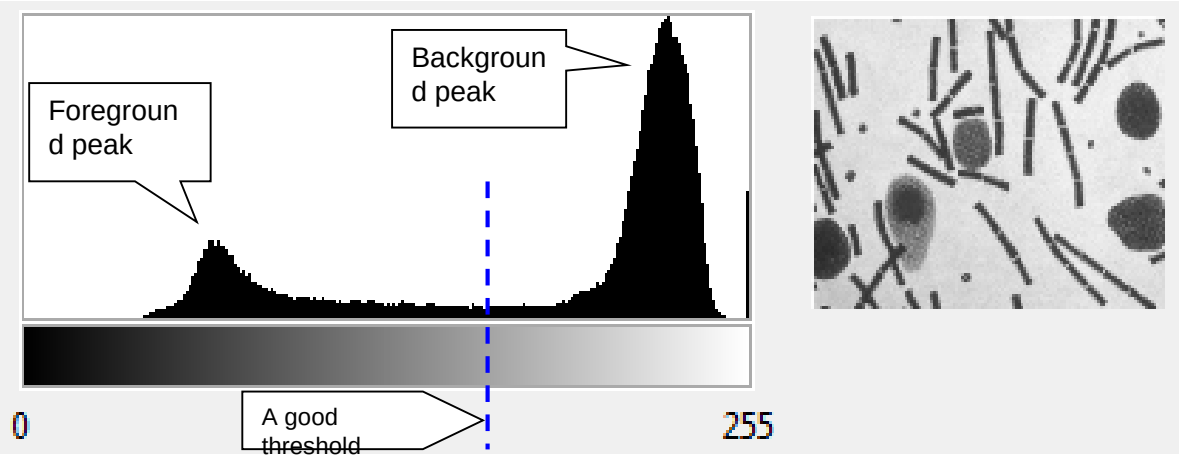
Otsu method for threshold calculation

- 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 values
  - y-axis – number of pixels with a particular value



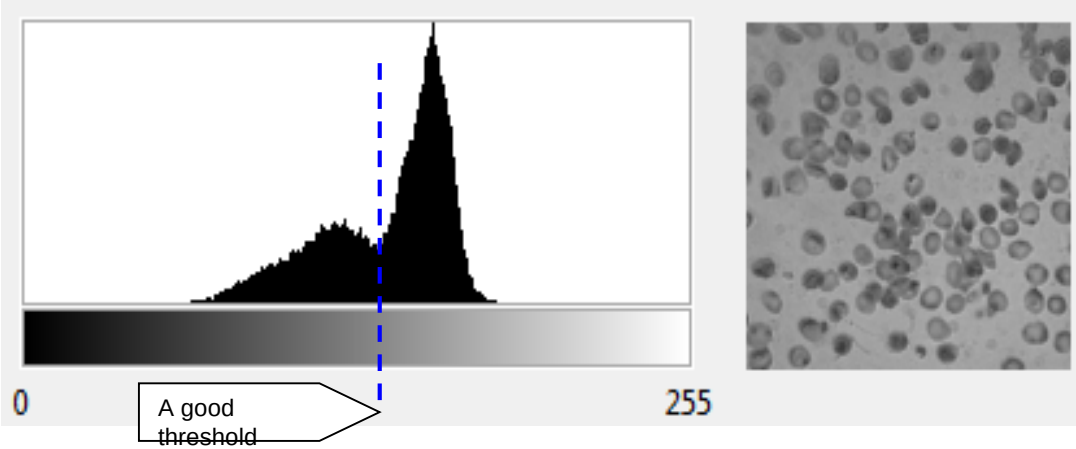
Otsu method for threshold calculation

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

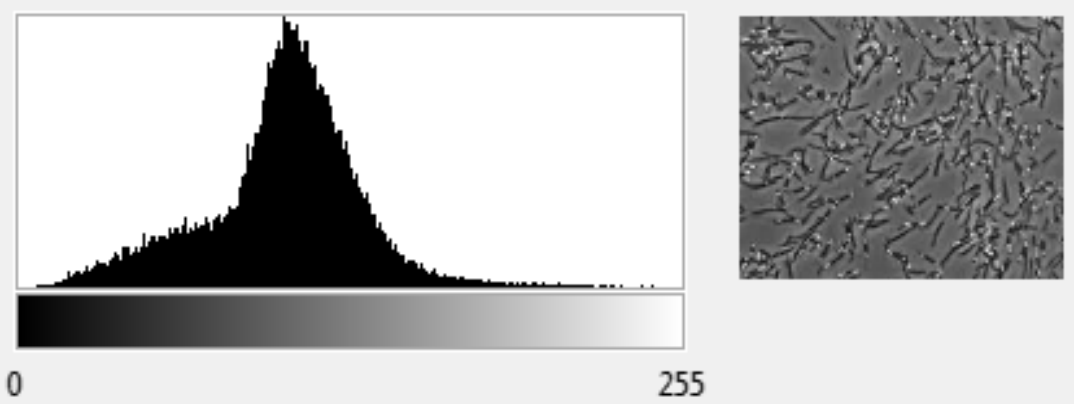


Otsu method for threshold calculation

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



- 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 – more formally

The algorithm exhaustively searches for the threshold that minimizes the intra-class variance, defined as a weighted sum of variances of the two classes:

$$\sigma_w^2(t) = \omega_0(t)\sigma_0^2(t) + \omega_1(t)\sigma_1^2(t)$$

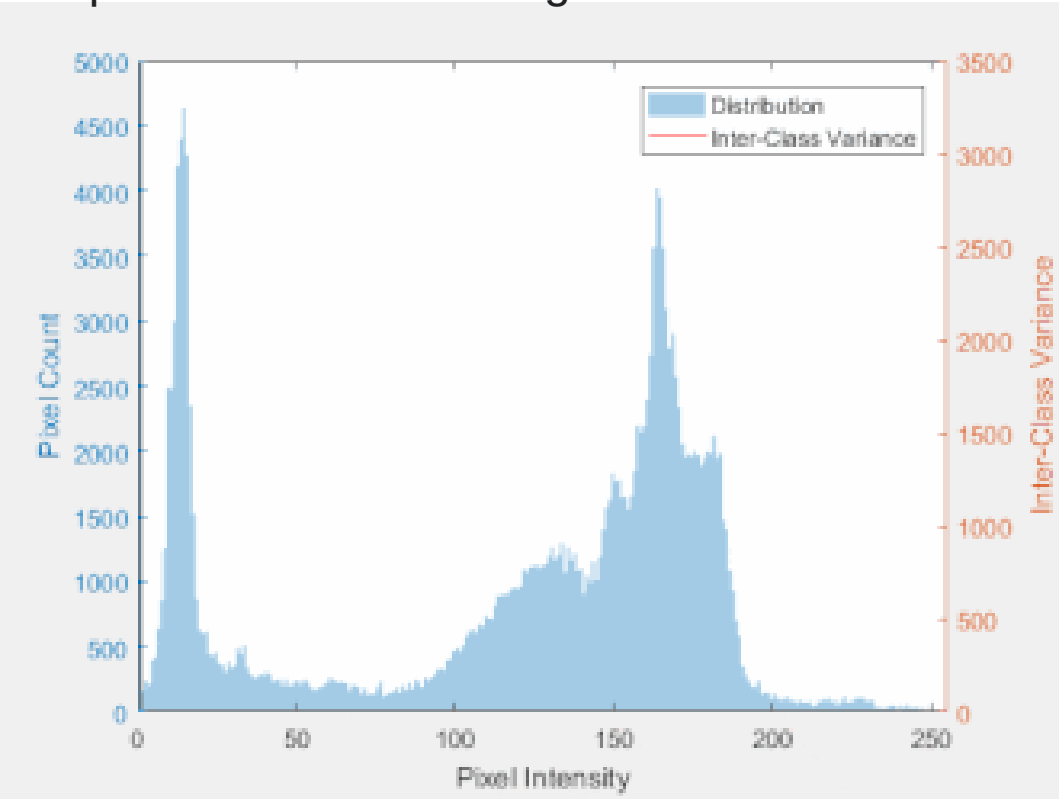
Weights ω_0 and ω_1 are the probabilities of the two classes separated by a threshold t ,and σ_0^2 and σ_1^2 are variances of these two classes.

The class probability $\omega_{0,1}(t)$ is computed from the L bins of the histogram:

$$\omega_0(t) = \sum_{i=0}^{t-1} p(i)$$

$$\omega_1(t) = \sum_{i=t}^{L-1} p(i)$$

For 2 classes, minimizing the intra-class variance is equivalent to maximizing inter-class variance

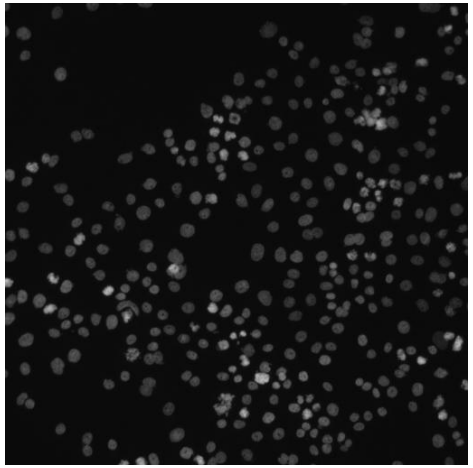


Source: Wikipedia

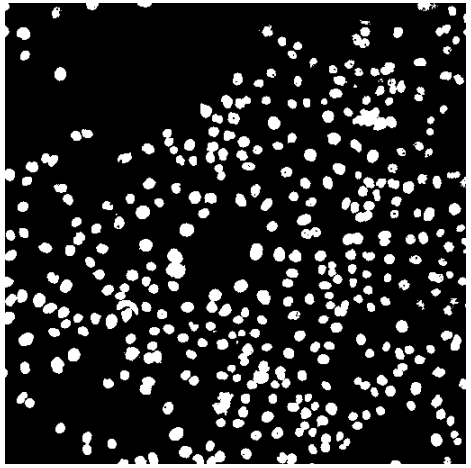
Otsu – example execution

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

Original: Human HT29 colon-cancer cells

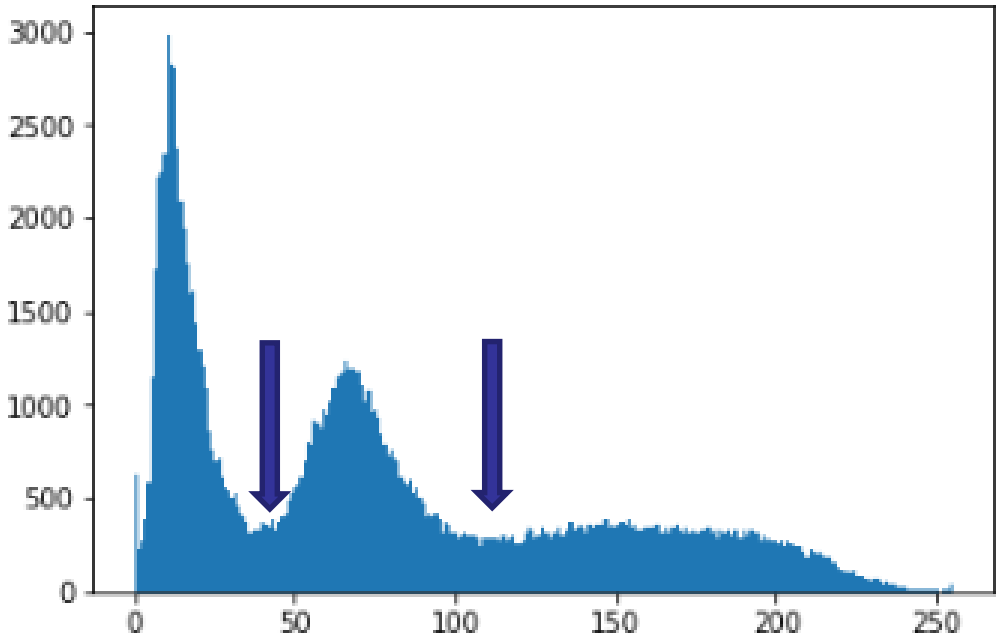


Otsu's Threshold = 38



Using Histograms

```
einstein=Matrix.load('albert-einstein-1951.bitmap')
plt.hist(items(einstein), bins=256 )

print()
```



Threshold = 45

Threshold = 110

Good candidates: 45, 110

However, 45 isn't good …

Histograms do not contain all information..

Thresholding Abbey Road

- Sometimes Thresholds are bad…
- Here we cannot say that white = people, black = background (or the other way around)
- **Thresholding is only good for certain types of images**

Original

Threshold = 50

Threshold = 100

Threshold = 150

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).

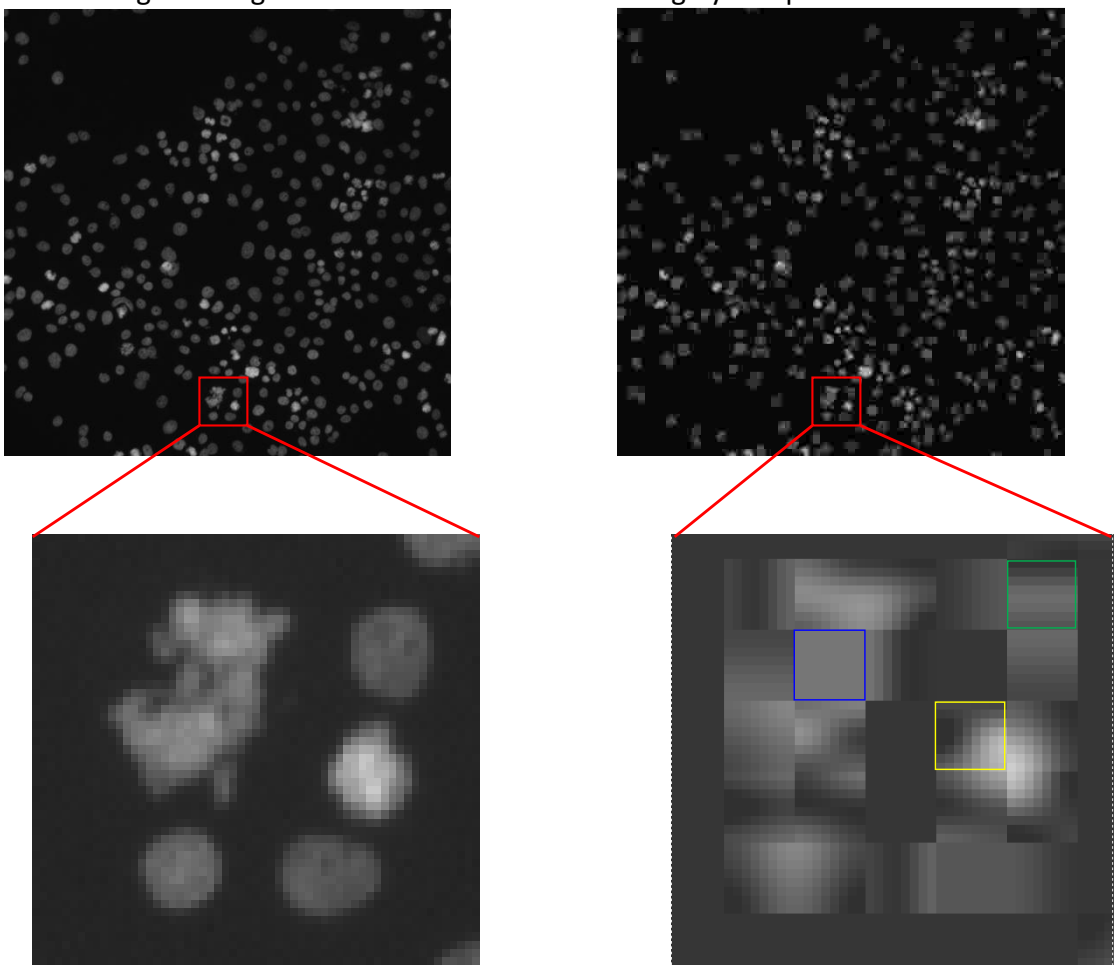
The example of jpg

- 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 $8 \cdot 8 \cdot 8 = 512$ bits in the original representation.
- A factor of 10 (or even more) saving in space can be achieved.

original image

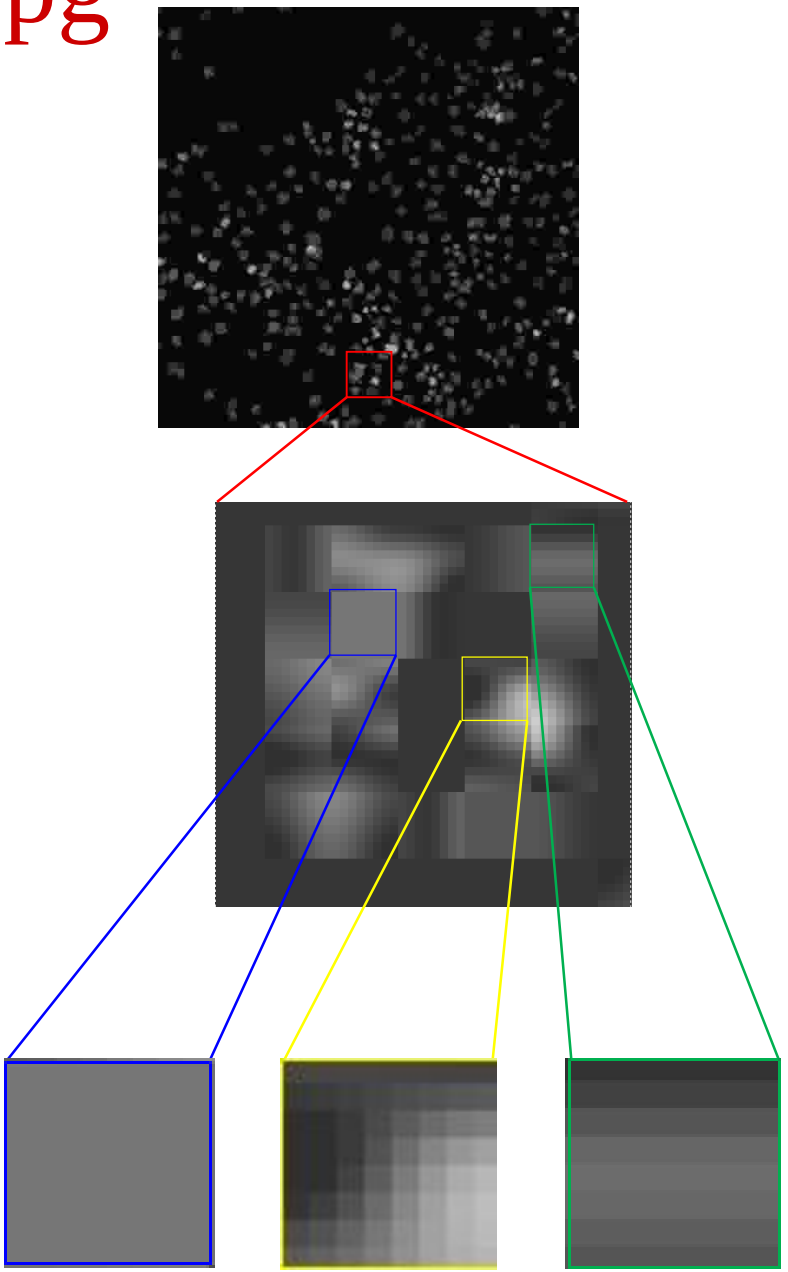
highly compressed version



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.

Numpy (intro to DS course)

The fundamental package for scientific computing with Python

GET STARTED

NumPy v1.19.0 First Python 3 only release - Cython interface to numpy.random complete

POWERFUL N-DIMENSIONAL ARRAYS

Fast and versatile, the NumPy vectorization, indexing, and broadcasting concepts are the de-facto standards of array computing today.

NUMERICAL COMPUTING TOOLS

NumPy offers comprehensive mathematical functions, random number generators, linear algebra routines, Fourier transforms, and more.

INTEROPERABLE

NumPy supports a wide range of hardware and computing platforms, and plays well with distributed, GPU, and sparse array libraries.

PERFORMANT

The core of NumPy is well-optimized C code. Enjoy the flexibility of Python with the speed of

EASY TO USE

NumPy's high level syntax makes it accessible and productive for programmers from any

OPEN SOURCE

Distributed under a liberal BSD license, NumPy is developed and maintained publicly on GitHub

https://numpy.org/

Scikit-image

Download Gallery Documentation Community Guidelines Source Search documentation ...

Stable (release notes)
0.18.0rc0 - November 2020

Download

Development
pre-0.19

Download

GitHub *source & bug reports*
Contribute *get involved*
Mailing List *dev. discussion*

Image processing in Python

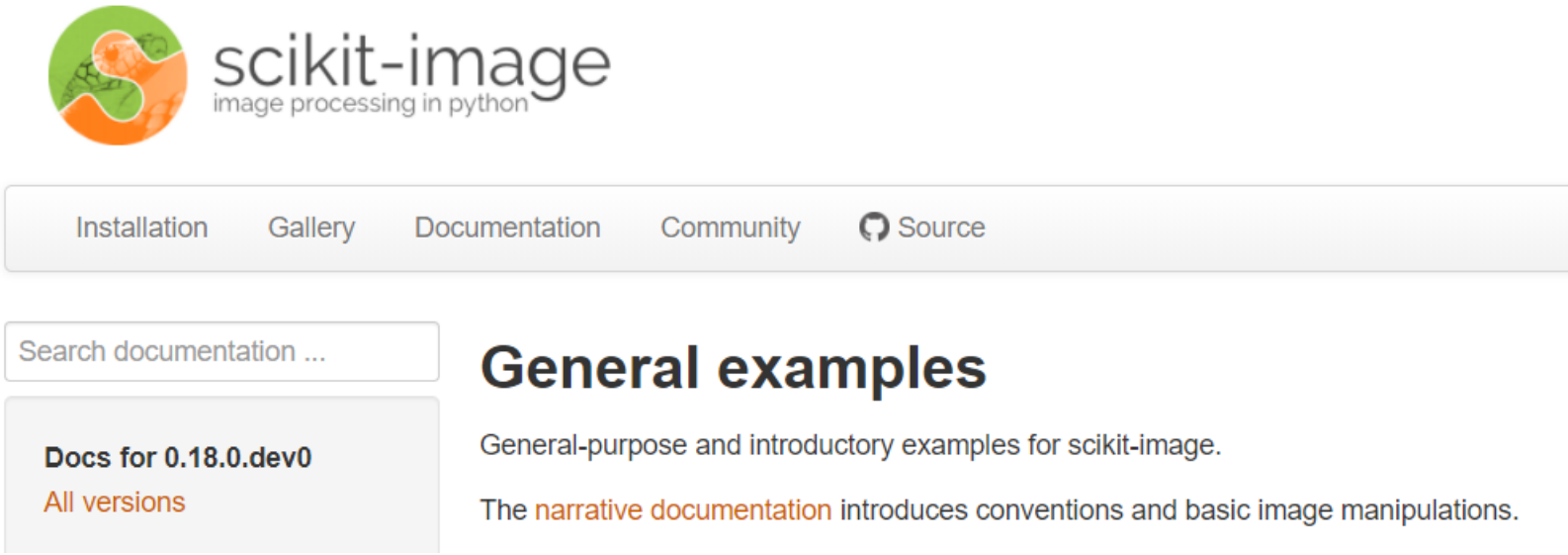
scikit-image is a collection of algorithms for image processing. It is available free of charge and free of restriction. We pride ourselves on high-quality, peer-reviewed code, written by an active community of volunteers.

If you find this project useful, please cite: [BiBTeX]

Stéfan van der Walt, Johannes L. Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D. Warner, Neil Yager, Emmanuelle Gouillart, Tony Yu and the scikit-image contributors. **scikit-image: Image processing in Python**. PeerJ 2:e453 (2014) https://doi.org/10.7717/peerj.453

https://scikit-image.org/

Example applications with scikit-image



https://scikit-image.org/docs/dev/auto_examples/