Computer Science 1001.py

Lecture 16b[†]:Introduction to Digital Image Processing

Instructors: Benny Chor, Amir Rubinstein
Teaching Assistants: Yael Baran, Michal Kleinbort

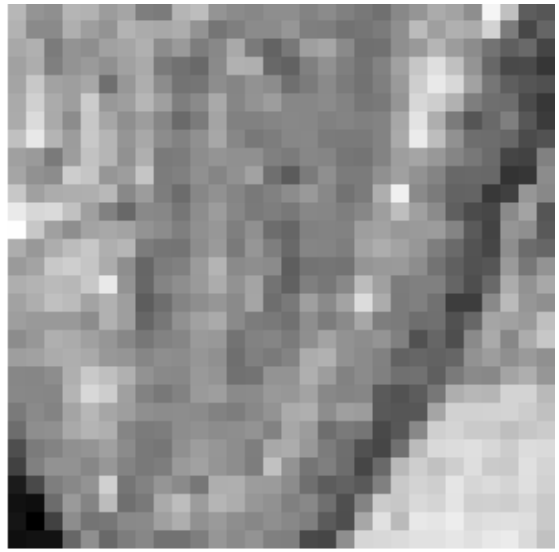
Founding Teaching Assistant (and Python Guru): Rani Hod

School of Computer Science
Tel-Aviv University, Spring Semester, 2015
http://tau-cs1001-py.wikidot.com

Plan

- Basics of Digital Image representation.
- Generating synthetic images.
- Basics of Digital Image Processing.
- Noise, and local noise reductions.

Digital Images Representation



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

```
[100 112  88 ..., 134 145 166]
[ 80 132 134 ..., 130 184 158]
[ 44  51  56 ..., 132 122   9]
...,
[ 14  17  15 ..., 206 204 184]
[ 21  11  12 ..., 203 176 185]
[ 24  13  16 ..., 200 180 182]
```

Brief "Historical" Context

At the early days of personal computers, say in the early 1980s, processors were relatively slow and quite expensive. Memory was even more expensive in relative terms.

Early e-mail (1970s to early 1980s) messages were plain ascii texts.

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

This was supposedly said when talking about IBM PC's 640KB RAM, which was a significant breakthrough over the previous 8-bit systems that were typically limited to 64KB RAM.

A Brief Context, 33 Years Later

With the proliferation of strong, inexpensive processors, larger and faster RAMs, and especially of large, non-volatile memory chips (*e.g.* flash memory, commercialized from mid 1990s) it became possible to efficiently store, process, and transmit large digital images.

Facebook stores about 350 million photos DAILY (reported September 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 December 2013). (Instagram was launched on October 2010 !!).

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

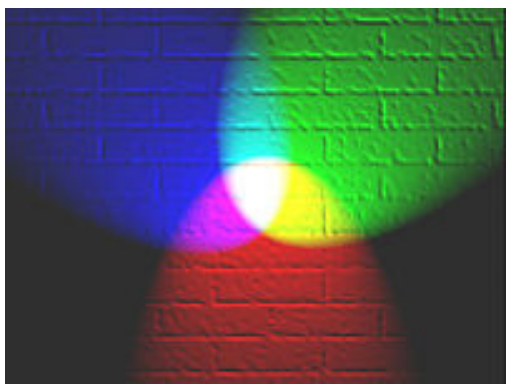
Basic Model of a Digital Image

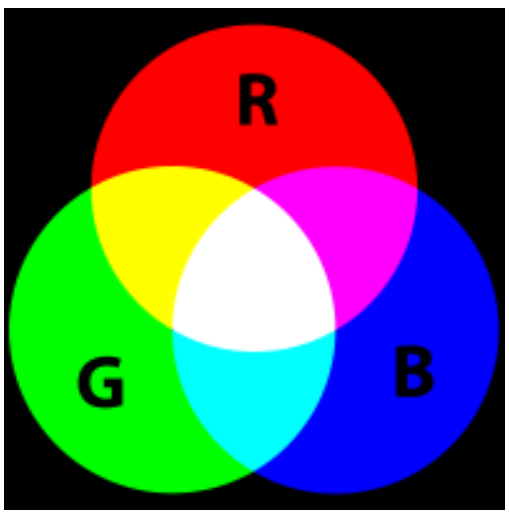
A digital image is typically encoded as a k-by-ℓ rectangle, or matrix, M, of either grey–level or color values.

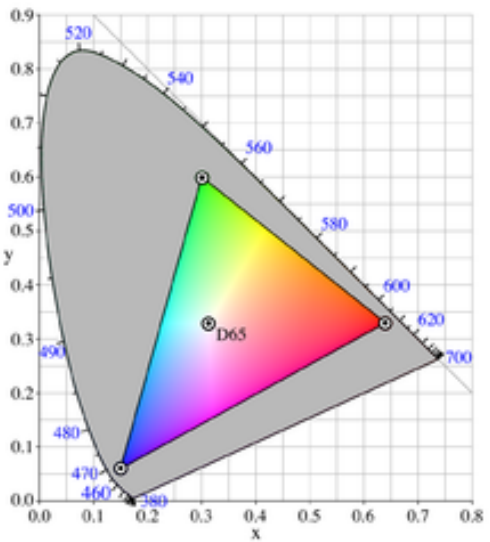
For videos (movies), there is a third dimension, "time". For each point t sampled in time, the frame at time t is nothing but a "regular" image.

Basic Model of a Digital Image, cont.

Each element $M[x, y]$ of the image is called a pixel, shorthand for picture element. For grey level images, $M[x, y]$ is a non negative real 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)

Grey Level Images

For the sake of simplicity, the remainder of this presentation will deal with grey scale images only. However, what we will do is applicable to color images as well.

Real numbers expressing grey levels have to be discretized in order to enable their representation on bounded precision digital devices.

A good quality 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 (not very intuitive, I agree :-).

For each pixel, the closer its value is to 0, the blacker it is. So 128 is a perfect grey.

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

class Matrix

```
class Matrix:
    """
    Represents a rectangular matrix with n rows and m columns.
    """

    def __init__(self, n, m, val=0):
        """
        Create an n-by-m matrix of val's.
        Inner representation: list of lists (rows)
        """
        assert n > 0 and m > 0
        #self.rows = [[val]*m]*n #why this is bad?
        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.row
```

class Matrix - Indexing

```
    # cell/sub-matrix access/assignment
    ###################################
    def __getitem__(self, ij): #ij is a tuple (i,j). Allows 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:
          # cell/sub-matrix access/assignment
    ###################################
    def __getitem__(self, ij): #ij is a tuple (i,j). Allows 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 - item access and assignment

```
>>> M = Matrix(10,10) #10x10 matrix of zeros
>>> M[4,5]
0
>>> M[4,5] = 45
>>> M[4,5]
45
```

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

```
>>> M[3:5,4:8]
<Matrix [[0, 0, 0, 0], [0, 7, 0, 0]]>
```

The .bitmap image file format

There are several common image file formats, such as jpg, bmp, tiff, png, etc. Each format represent images somewhat differently. In particular, most of them employ lossy compression to save on memory.

We will be using our own format in this course: the .bitmap format.

$$
\begin{matrix}
M_{0,0} & M_{0,1} & M_{0,2} & \dots & M_{0,m-1} \\
M_{1,0} & M_{1,1} & M_{1,2} & \dots & M_{1,m-1} \\
\dots & & & & \\
M_{n-1,0} & M_{n-1,1} & M_{n-1,2} & \dots & M_{n-1,m-1}
\end{matrix}
$$

Our `matrix.py` package has (rather simple) methods for loading and saving digital images in this format.

```
#load image into an instance of the  Matrix class
>>> m = Matrix.load("some_image.bitmap")
#save an instance of Matrix class into an image in .bitmap format
>>> m.save("my_copy.bitmap")
```

Use ./ for the current directory, and ../ to get to the parent directory:

```
>>> m = Matrix.load("./some_image.bitmap")
>>> m = Matrix.load("../../some_image.bitmap")
```

Converting to and from the .bitmap format

Our `.bitmap` format employs no compression, thus it requires large amount of storage, and consequently is not used outside this course. Images in this format are provided in the course website, so you can work directly with them.

If you are interested in converting "real" images to and from our `.bitmap` format, you can use the functions `image2bitmap` and `bitmap2image`, which are both part of the `images.py` file.

To run them, you will need to first install the PIL (Python's Image Library) package. The PILLOW version of the package is accessible and downloadable at `https://pypi.python.org/pypi/Pillow/2.8.1/` .

Converting to and from the .bitmap format, cont.

We provide these two functions to do the conversions, so we can work with "real" images:

```
def image2bitmap(path)
    ...
def bitmap2image(path)
    ...

>>> image2bitmap("./real_image.jpg")
#This creates a file with the same name in .bitmap format
>>> im = Matrix.load("./real_image.bitmap")
>>> ...
>>>     #manipulate the image
>>> bitmap2image("./real_image.bitmap")
#This created a file with the same name in .bmp format
```

You do not need to understand the fine print of these conversions (although they are rather simple).

Now you'll be able to impress your friends and family with your own homemade mini version of photoshop!

Displaying .bitmap Images

Our `matrix.py` package also has a methods for displaying the image it represents.

```
>>> m = Matrix.load("abbey_road.bitmap")
>>> m.display()
```

This opens a new, graphical window.
To run additional code in the shell, this window has to be closed first.

Again, you do not need to understand the fine print of how the `display()` method works.

Loading and Displaying Images, cont.

In the display method, we can zoom in. Note that the zoom factor has to be a positive integer.

```
>>> m=Matrix.load("abbey_road.bitmap")
>>> m.display(zoom=3)
>>> m.display(zoom=2)

>>> m.display(zoom=1.5)

>>> Exception in thread Thread-1:
Traceback (most recent call last):
  File "C:\Python32\lib\threading.py", line 740, in _bootstrap_inne
    self.run()
  File "D:\INTRO to CS\New course Lecture presentations\Lec 21-22 -
    func(root)
  File "D:\INTRO to CS\New course Lecture presentations\Lec 21-22 -
    pi = pi.zoom(zoom)
  File "C:\Python32\lib\tkinter\__init__.py", line 3249, in zoom
    self.tk.call(destImage, 'copy', self.name, '-zoom',x,y)
_tkinter.TclError: expected integer but got "1.5"
```

Grey Level Images - Another Example

```
>>> Albert=Matrix.load("albert-einstein-1951.bitmap")
>>> Albert.display(zoom=2)
>>> Albert.display()

>>> Tongue=Albert[260:300,130:160]
        # a slice of the original

>>> Tongue.display(zoom=6)
>>> Tongue.dim()
(40, 30)
```

The Tongue, in Numbers

```
>>> Tongue=Albert[260:300,130:160]
        # a slice of the original

>>> Tongue.display(zoom=6)
>>> Tongue.dim()
(40, 30)

>>> for i in range(40):
print([Tongue[i,j] for j in range(30)])
```

Tinkering with an Image

```
def copy(mat):
    ''' brand new copy of matrix '''
    n,m=mat.dim()
    new=Matrix(n,m)
    for i in range (n):
        for j in range (m):
            new[i,j]=mat[i,j]
    return new

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=copy(mat)
        for i in range (100):
            for j in range (100):
                new[i,j]=0
    return new

>>> m = Matrix.load("albert-einstein-1951.bitmap")
>>> bs = black_square(m)
>>> bs.display()
```

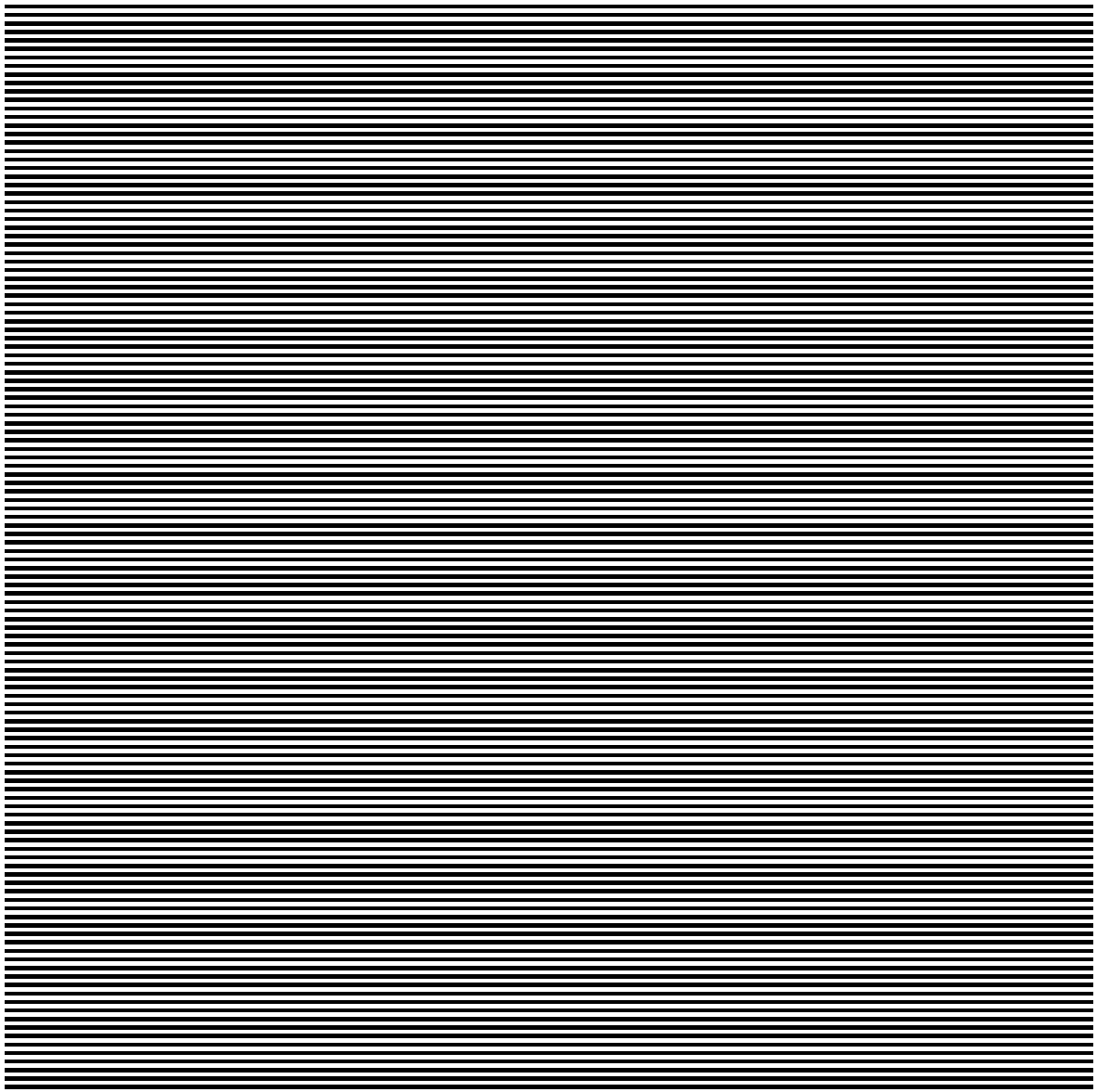
Tinkering with a Real Image, cont.

```
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<512 or m<512:
        return None
    else:
        new=copy(mat)
        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 (412,512):
            for j in range (412,512):
                new[i,j ]= 255  # white square
    return new

>>> m = Matrix.load("abbey_road.bitmap")
>>> ts = three_squares(m)
>>> ts.display()
```

Simple Synthetic Images: Lines and More

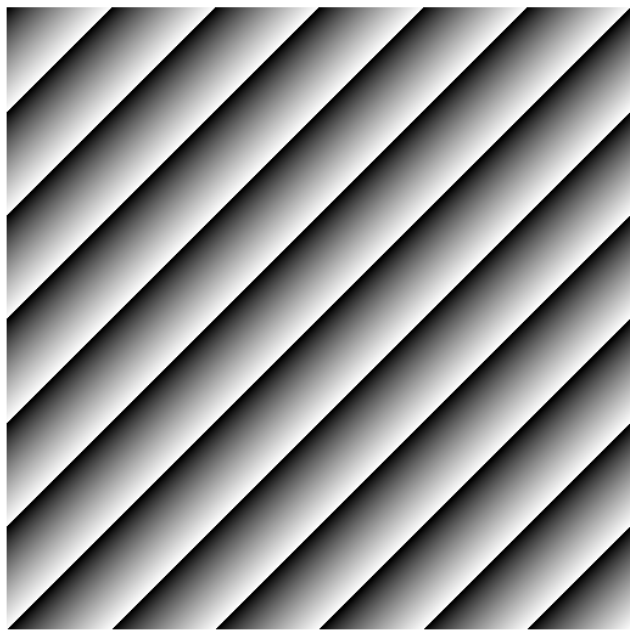
```
def horizontal():
    horizontal_lines=Matrix(512,512)
    for i in range (512):
        if i % 2 == 0:
            for j in range (512):
                horizontal_lines[i,j]=255
    return horizontal_lines

>>> A = horizontal()
>>> A.display()
>>> A.display(zoom=2)
```

Displaying Synthetic Images: Lines and More



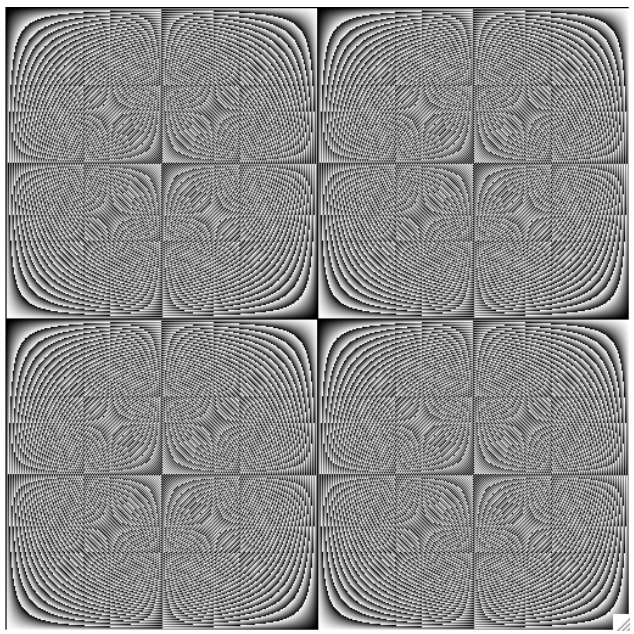
Simple Synthetic Images: Diagonal Lines

```
def diagonals(c=1):
    surprise=Matrix(512,512)
    for i in range(512):
        for j in range(512):
            surprise[i,j]= (c*(i+j)) % 256
    return surprise

>>> A = diagonals()
>>> A.display()
>>> A = diagonals(c=3)
>>> A.display()
```



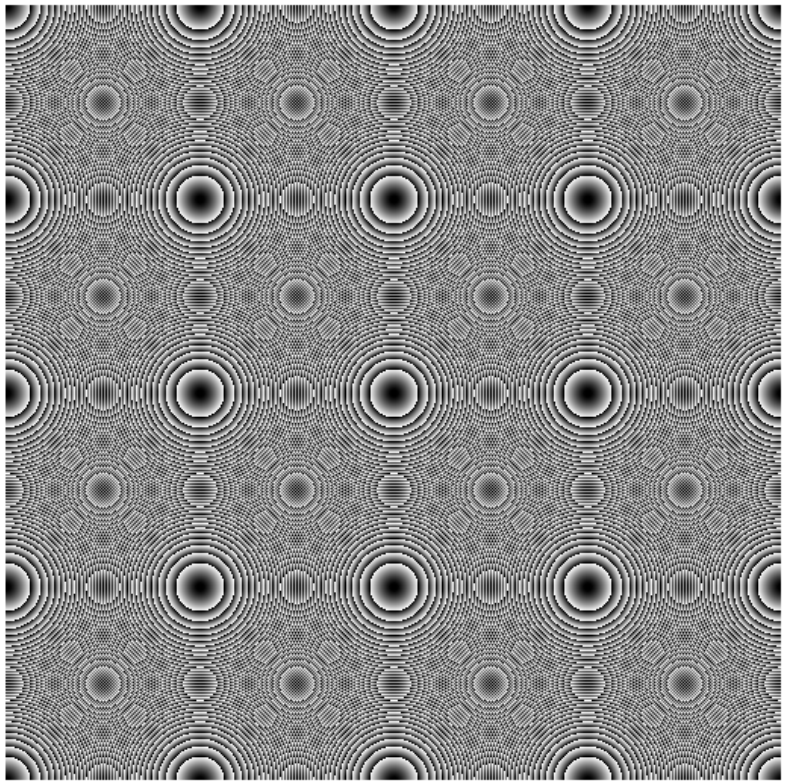
Simple Synthetic Images: Product and Circles

```
def product(c=1):
    surprise=Matrix(512,512)
    for i in range(512):
        for j in range(512):
            surprise[i,j]=(c*(i*j-512))% 256
    return surprise

>>> A = product()
>>> A.display()
>>> A = product(c=2)
>>> A.display()
```



Simple Synthetic Images: Product and Circles

```
def circles(c=1):
    surprise=Matrix(512,512)
    for i in range(512):
        for j in range(512):
            surprise[i,j]=c*((i-256)**2+(j-256)**2) % 256
    return surprise
```



Simple Synthetic Images: Miscellaneous

```
import sys, random
import math, cmath    # complex numbers

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

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

>>> B = synthetic(512, 512, lambda x,y: \
   math.sin (16*((x-256)**2 + (y-256)**2)))
>>> B.display()

>>> C = synthetic(512, 512, lambda x,y: \
   56*math.sin(32*cmath.phase(complex(x-256,y-256))))
>>> C.display()
```

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

Next: 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

Typical operations:

- Color correction.
- Image segmentation.
- Image registration (integrating different images to a unified coordinate system).
- Denoising (noise reduction).

Typical applications:

- Machine vision.
- Medical image processing.
- Face detection and recognition.
- Automatic target recognition.
- Augmented reality.

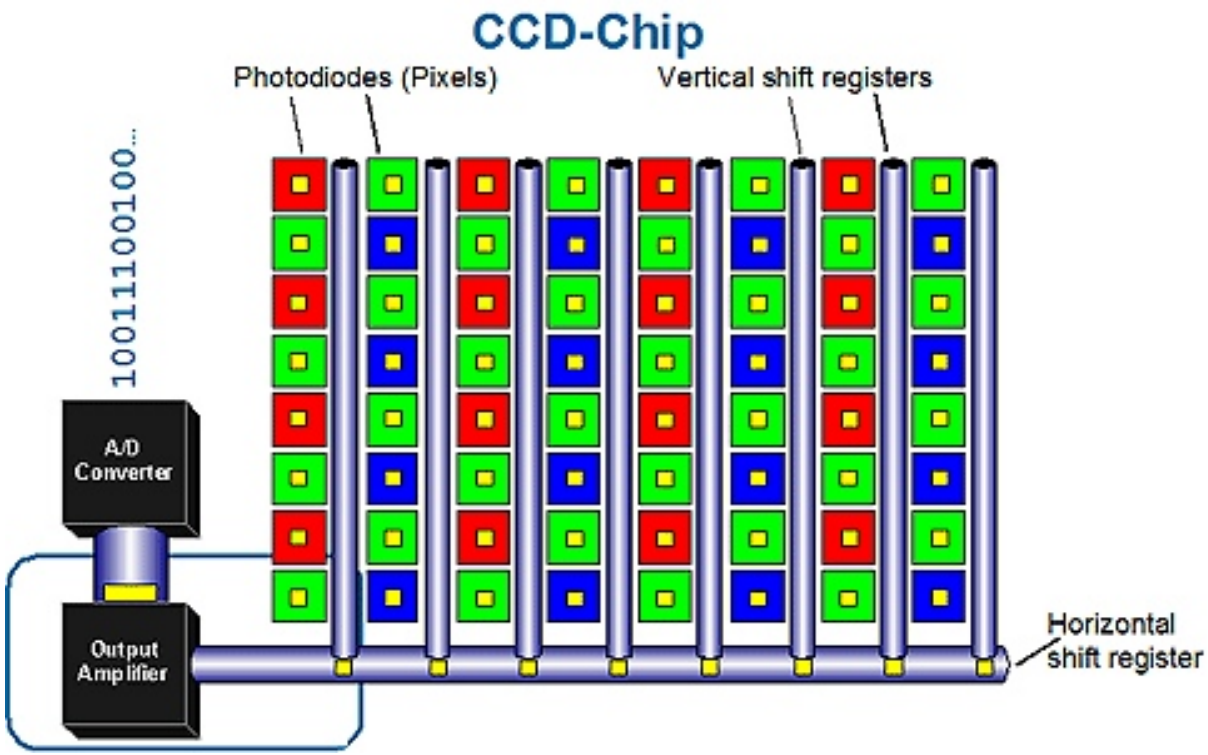
Capturing Images

Consider a specific pixel with coordinates x, y. Suppose $I(x, y)$ is 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, $S(x, y)$, is the result of the light intensity measurement, usually made by a CCD (charge coupled device, transforming light to electrical voltage) matrix, together with an optical light focusing system (lens or lenses). Each captor of the CCD is roughly a square area, in which the number of incoming photons is being counted for a fixed period corresponding to the obturation time.

CCD and DSP

Digital signal processing (DSP) takes the raw data from the sensor and assembles it in correct color space or bitmap structure. In doing so, it handles white balance, brightness, sharpness ,contrast and noise levels.



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

Blur

The two major effects hampering image accuracy are termed blur and noise. Blur is an intrinsic phenomenon to digital image acquisition, resulting from limits on sampling rates. Blur has the effect of reducing the image's high-frequency components. 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).

An original image (left) and a blurred version thereof (right). Taken from Wikipedia (which ran out of the "g'day mate" version).

Noise and Denoising

The observed value at pixel x, y, $S(x, y)$, equals the sum of the true value $I(x, y)$ plus noise $N(x, y)$.

$$S(x, y) = I(x, y) + N(x, y) \ .$$

The goal of denoising algorithms is, given the observed image $S(x, y)$ $(0 \leq x < k,\ 0 \leq y < \ell)$ to produce a new image, $\hat{I}(x, y)$, which should be close to the original image $I(x, y)$ $(0 \leq x < k,\ 0 \leq y < \ell)$.

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

Gaussian Noise Models

Image model: 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 $I(x_1, y_1)$ and $I(x_2, y_2)$ attain close enough values.

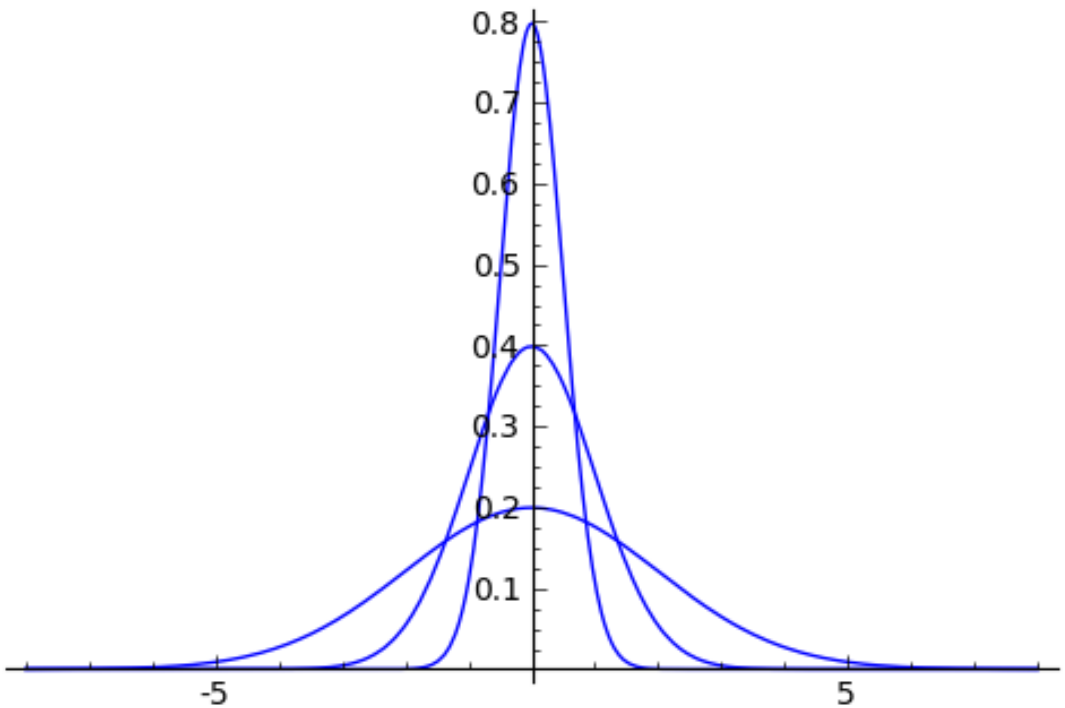
Gaussian noise model: The noise at pixel x, y, $N(x, y)$, is a random variable. It is usually assumed that $N(x, y)$ is "white noise", distributed independently of the noise at other pixels.

The noise level depends monotonically on the signal level. Higher intensity pixels produce higher measured noise.

Specifically, if a pixel intensity is σ^2 photons, then the additive noise is distributed according to $G_\sigma(x)$, a Gaussian with standard deviation σ.

Gaussians

The probability density function $G_\sigma(x) = e^{-(x/\sigma)^2/2}/\sqrt{2\pi\sigma^2}$ 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 $\sigma = 0.5, 1, 2$ ($\sigma = 0.5$ is the narrowest).

Gaussian Noise: Python Code

The function `random.gauss(mu, sigma)` returns a floating point number, distributed according to a Gaussian distribution with expected value μ and standard deviation 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.

```
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'''
  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
```

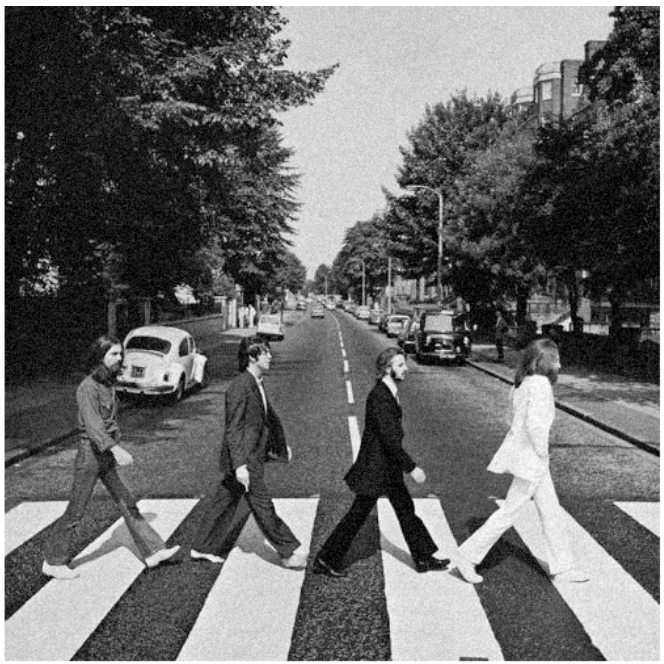
A Useful Utility: Joining Images, Side by Side

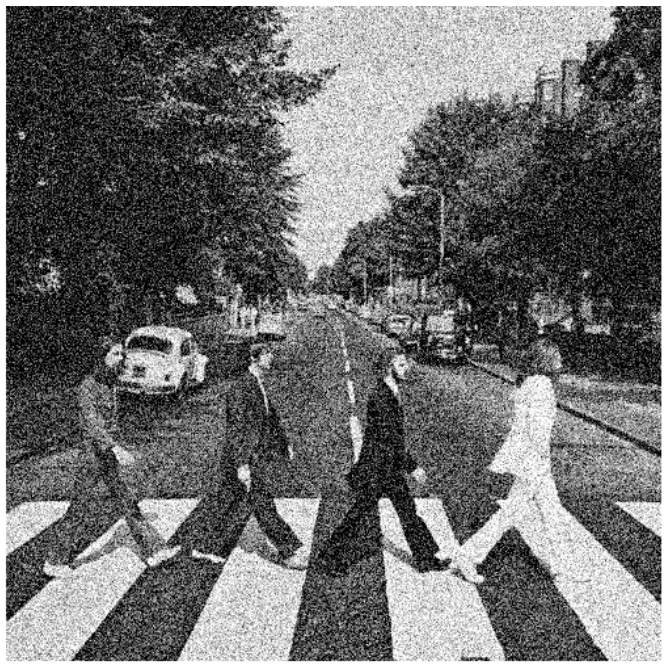
```
def join_h(mat1, mat2):
    """ joins two matrices, side by side with some separation """
    n1,m1 = mat1.dim()
    n2,m2 = mat2.dim()
    m = m1+m2+10
    n = max(n1,n2)
    new = Matrix(n, m, val=255)# fill new matrix with white pixels

    new[:n1,:m1] = mat1
    new[:n2,m1+10:m] = mat2
    '''
    #without slicing:
    for i in range(n1):
        for j in range(m1):
            new[i,j] = mat1[i,j]

    for i in range(n2):
        for j in range(m2):
            new[i,j+m1+10] = mat2[i,j]
    '''
    return new
```

Adding Gaussian Noise to an Image: Examples

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

```
>>> A = Matrix.load("abbey_road.bitmap")
>>> new10 = add_gauss(A)
>>> new20 = add_gauss(A,sigma=20)
>>> new50 = add_gauss(A,sigma=50)
>>> N1 = join_h(A,new10)
>>> N2 = join_h(new10,new20)
>>> N3=join_h(new10,new50)
>>> N1.display()
>>> N2.display()
>>> N3.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

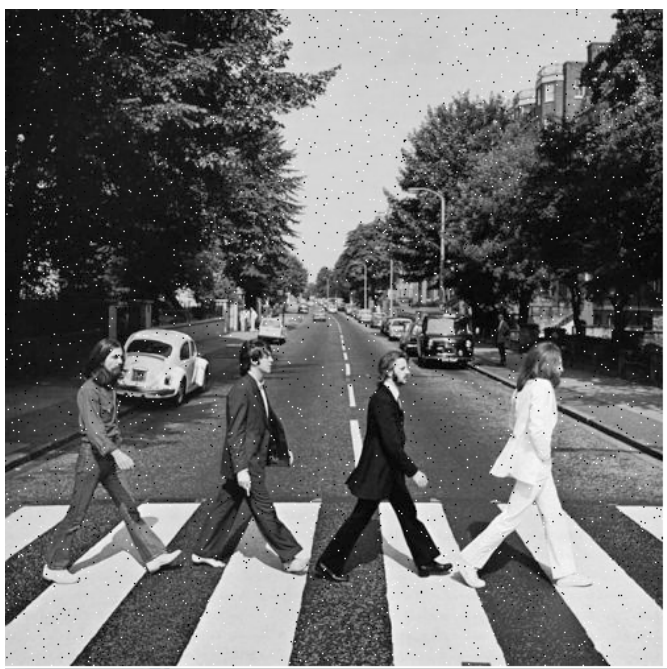
Salt and pepper noise is not as fundamental in statistics as Gaussian noise is, so we'll write the code for it ourselves. The default parameter `p=0.01` is the probability of a pixel being "hit" by the SP noise.

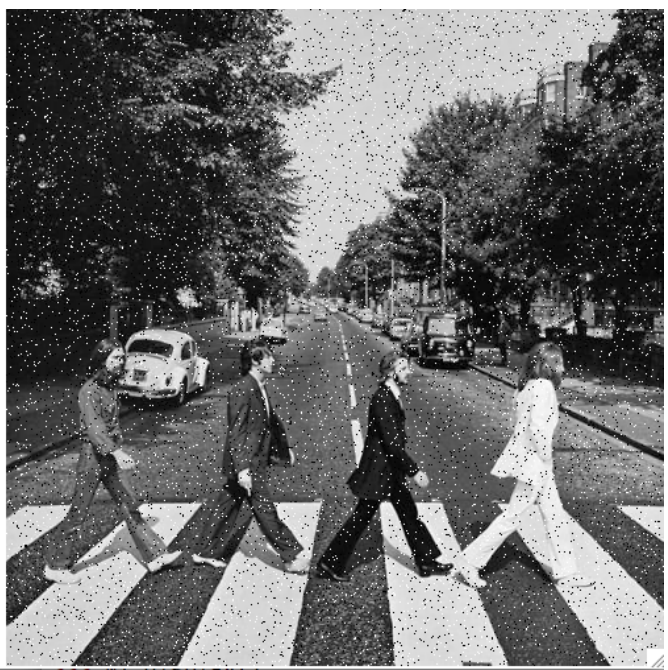
```
def add_SP(mat, p=0.01):
    ''' Generates salt and pepper noise: Each pixel is "hit" indep.
      with prob. p. If hit, it has fifty fifty chance of becoming
      white or black. '''
    n,m = mat.dim()
    new = copy(mat)
    for i in range(n):
        for j in range (m):
            rand = random.random() #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 Salt and Pepper Noise to an Image: Examples

We add Salt and Pepper noise at different levels.

```
>>> A=Matrix.load("abbey_road.bitmap")
>>> SP1 = add_SP(A)
>>> SP2 = add_SP(A,p=0.02)
>>> SP5 = add_SP(A,p=0.05)
>>> N1 = join_h(A,SP1)
>>> N2 = join_h(A,SP2)
>>> N5 = join_h(SP1,SP5)
>>> N1.display()
>>> N2.display()
>>> N5.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 Means

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.

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

Local Means: Auxiliary Code

```
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

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

`items(mat)` returns a list whose elements are the matrix' elements.
Average is rounded to nearest integer.

Local Means: Code

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

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

The code operates only on pixels in the center of a $2k+1$-by-$2k+1$ window all of which fits within the matrix. Other, "boundary" pixels, are left intact.

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]
```

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

Denoising by Local Means

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

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

Denoising by Local Means: Motivation

If the pixel x, y pixel 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, averaging $(2k+1)^2$ independent random variables with standard deviation σ, the standard deviation of the average decreases to $\sigma/\sqrt{(2k+1)^2}$ ($= \sigma/3$ for the $N_{3\times 3}(x, y)$ neighborhood).

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

Local Means: Weighted Variants

Uniform averaging based on the whole neighborhood, as discussed before, can be expressed as the matrix dot product

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

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

- 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.
- 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 $N_{3\times 3}(x, y)$ neighborhood, one pixel, *e.g.* $x-1, y-1$, is observed as very dark, *e.g.* intensity level around 20, due to noise.
- Indeed, $\hat{I}(x-1, y-1)$ 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), $S(x, y)$, by the median of the observed values in a neighborhood of (x, y) (for example, the 3-by-3 neighborhood, $N_{3\times 3}(x, y)$, 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.
- However, the median tends to eliminate small, fine features in the image, such as thin contours.

Local Medians: Code

```
def median(lst):
    sort_lst = sorted(lst)
    l = 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+l//2]) + int(sort_lst[l//2])) // 2

def local_operator(A, op, k=1):
  n,m = A.dim()
  res = copy(A)  # brand new copy of A
  for i in range(k,n-k):
    for j in range(k,m-k):
      res[i,j] = op(items(A[i-k:i+k+1,j-k:j+k+1]))
  return res

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

Comment: Median is computed by first sorting the values in the local window, and taking, ahhm, their median.

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_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]
```

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 $(n-2k)(m-2k)$. Assuming k is significantly smaller than m, n, this is $O(n \cdot m)$. 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 saw it in the recitation, named `select(lst,k)`). This will then be $O(k^2)$ steps too.

So we got $O(k^2)$ steps per window, for a total of $O(n \cdot m)$ windows. All by all, $O(k^2 nm)$ steps (recall that the input size is nm !) for both algorithms. (The hidden constant for the local means will be smaller than the hidden constant for the local medians.)

Putting Local Means to the Test

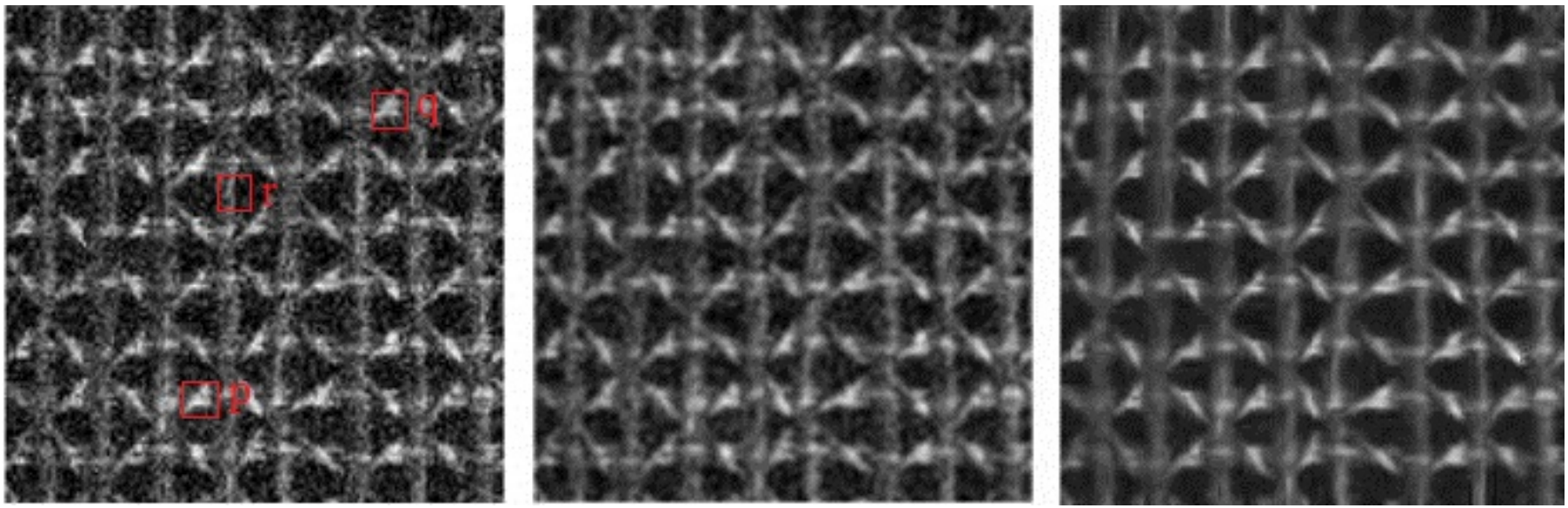
We will explore different local denoising methods on-line, and display results back to back with original or each other.

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 Local means Non-local means

Window centered at p is similar to the one centered at q, 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.

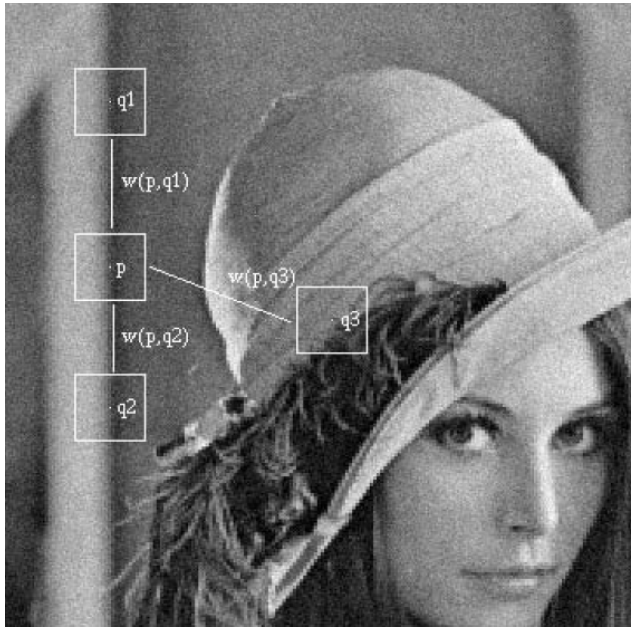
In other words, we look for all (x', y') such that the "distance" between the windows $N(x, y)$ and $N(x', y')$, is below some fixed threshold h).

We compute the weighted average value of all center pixels (x', y') (including (x, y) itself), with higher weights assigned to windows that are more similar. The corrected value, $\hat{I}(x, y)$, equals this average.

The method is called non-local since the windows that effect the corrected value $\hat{I}(x, y)$ 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.

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 will do our best to avoid objectification of women in the course or the course material.