Computer Science 1001.py
Lecture 18: A Crash Intro to Digital Images[†], Noise, and Noise Reduction

Instructor: Benny Chor

Teaching Assistant (and Python Guru): Rani Hod

School of Computer Science
Tel-Aviv University
Fall Semester, 2011/12
http://tau-cs1001-py.wikidot.com

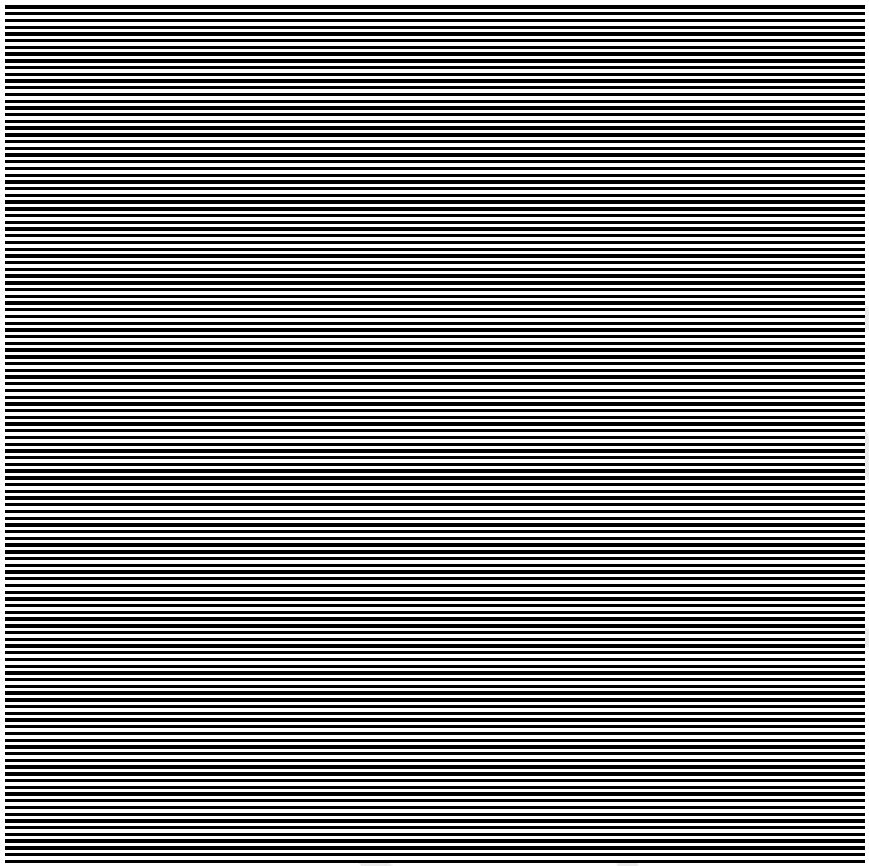
Thanks to Matan Gavish (Statistics, Stanford), Micha Lindenbaum (CS, Technion) and Nir Sochen (Math., TAU) for helpful suggestions and discussions.

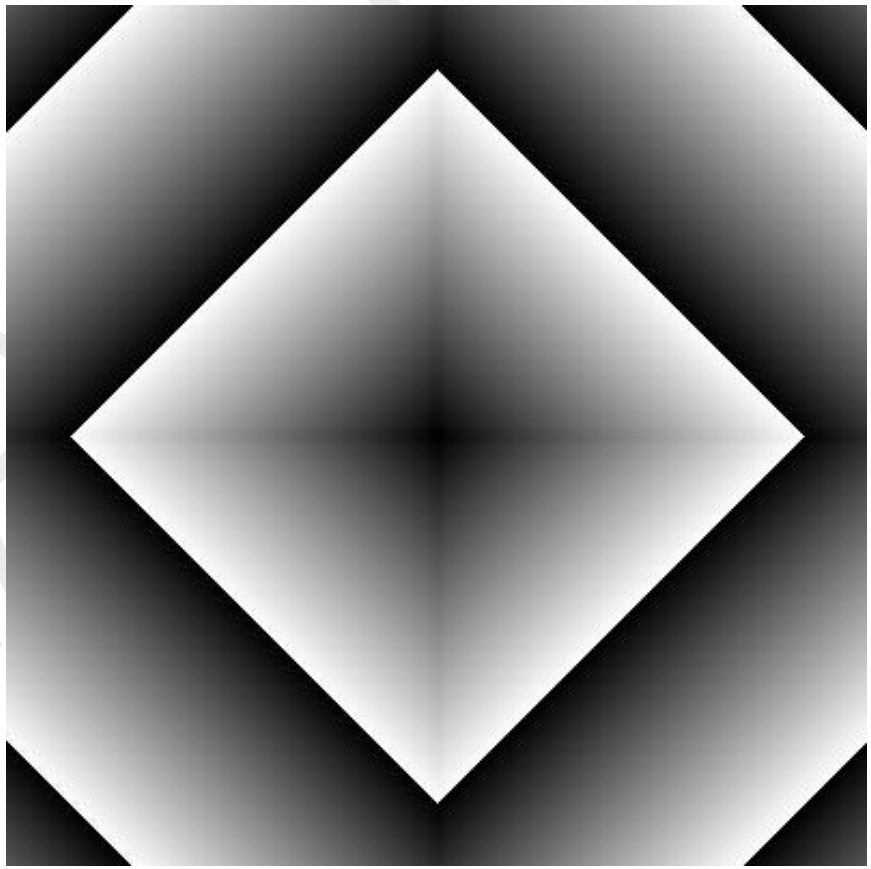
Simple Synthetic Images: Horizontal Lines and More

```
import numpy
from PIL import Image

horizontal_lines=numpy.zeros((256,256),dtype='uint8')
for x in range(256):
    if x % 2 == 0:
        for y in range(256):
            horizontal_lines[x,y]=255
Lines256=Image.fromarray(horizontal_lines)
Lines256.save("horizontal_Lines256","jpeg")
Lines256.show()
del horizontal_lines
s
abs_cont=numpy.zeros((400,400),dtype='uint8')
for x in range(400):
        for y in range(400):
            abs_cont[x,y]=1.5*(abs(x-200)+abs(y-200))
Abs400=Image.fromarray(abs_cont)
Abs400.save("Abs_Cont400","jpeg")
Abs400.show()
del abs_cont
```

Displaying Synthetic Images: Horizontal Lines and More





Simple Synthetic Images: Scaling Lines

```
import numpy
from PIL import Image
A= numpy.zeros((512,512),dtype='uint8')
B= numpy.zeros((512,512),dtype='uint8')
C= numpy.zeros((512,512),dtype='uint8')

for i in range (512):
    for j in range (512):
        A[i,j]=(i+j -512)
Line = Image.fromarray(numpy.uint8(A))

for i in range (512):
    for j in range (512):
        B[i,j]=2*(i+j -512)
Line2 = Image.fromarray(numpy.uint8(B))

for i in range (512):
    for j in range (512):
        C[i,j]=4*(i+j -512)
Line4 = Image.fromarray(numpy.uint8(C))

Line4.show()
Line2.show()
Line.show()
```

Simple Synthetic Images: Circles and More

```
import numpy
from PIL import Image

A= numpy.zeros((512,512),dtype='uint8')
for i in range (512):
    for j in range (512):
        A[i,j]=((i -256)**2+(j -256)**2)//256
Circ256 = Image.fromarray(A)

for i in range (512):
    for j in range (512):
        A[i,j]=((i -256)**2+(j -256)**2)//16
Circ16 = Image.fromarray(A)

for i in range (512):
    for j in range (512):
        A[i,j]=((i -256)**2+(j -256)**2)//4
Circ4 = Image.fromarray(A)

Circ4.show()
Circ16.show()
Circ256.show()
```

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

Simple Synthetic Images: Miscellaneous

```
import sys
import numpy
from PIL import Image
import math
import cmath    # complex numbers

A= numpy.zeros((512,512),dtype='uint8')
B= numpy.zeros((512,512),dtype='uint8')
C= numpy.zeros((512,512),dtype='uint8')

for i in range (512):
    for j in range (512):
        A[i,j]=(math.sin ((i-256)**2 + (j-256)**2 )* 16)
Trig = Image.fromarray(numpy.uint8(A))

for i in range (512):
    for j in range (512):
        B[i,j]=256*math.sin(25*cmath.phase(complex(i-256,j-256)))
Angle = Image.fromarray(numpy.uint8(B))

Angle.show()
Trig.show()
```

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

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 "gidday mate" version).

Edges

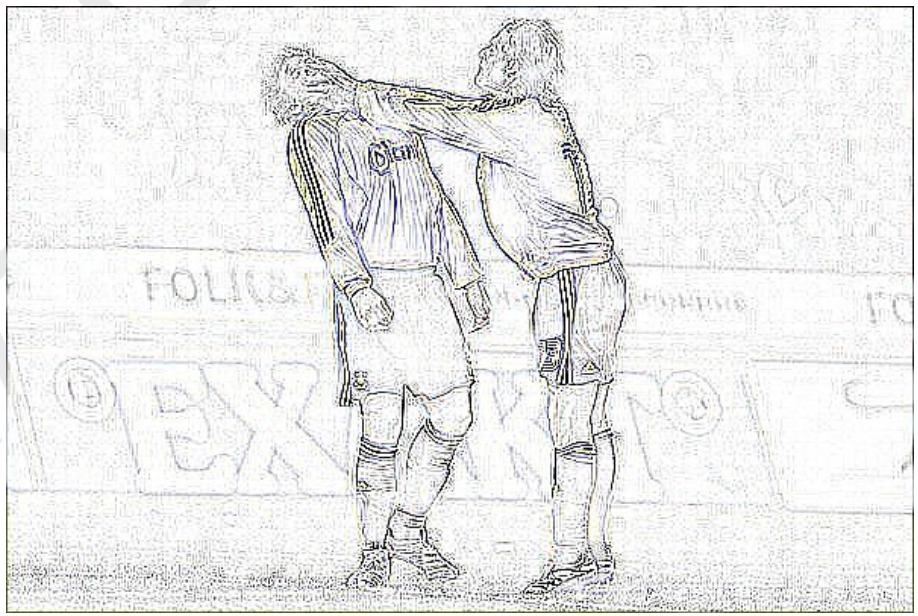
Images of interest are usually not completely smooth. While most areas are smooth, parts of images exhibit sharp changes in intensity from one pixel to the next. These boundaries are termed edges, and often capture much of the meaningful information in an image.

The problem of edge detection is a central problem in image processing, and many algorithms attempt to solve it.

An original image (of the world famous Lena) (center), and the results of two different edge detection algorithms (Sobel, left and Laplacian, right). Images taken from http://www.pages.drexel.edu/∼weg22/edge.html.

Built in Filters in PIL

The PIL package has a number of built in filters. For example, it has a contours finding filter, whose result is shown here.



(see http://www.riisen.dk/dop/pil.html)

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.

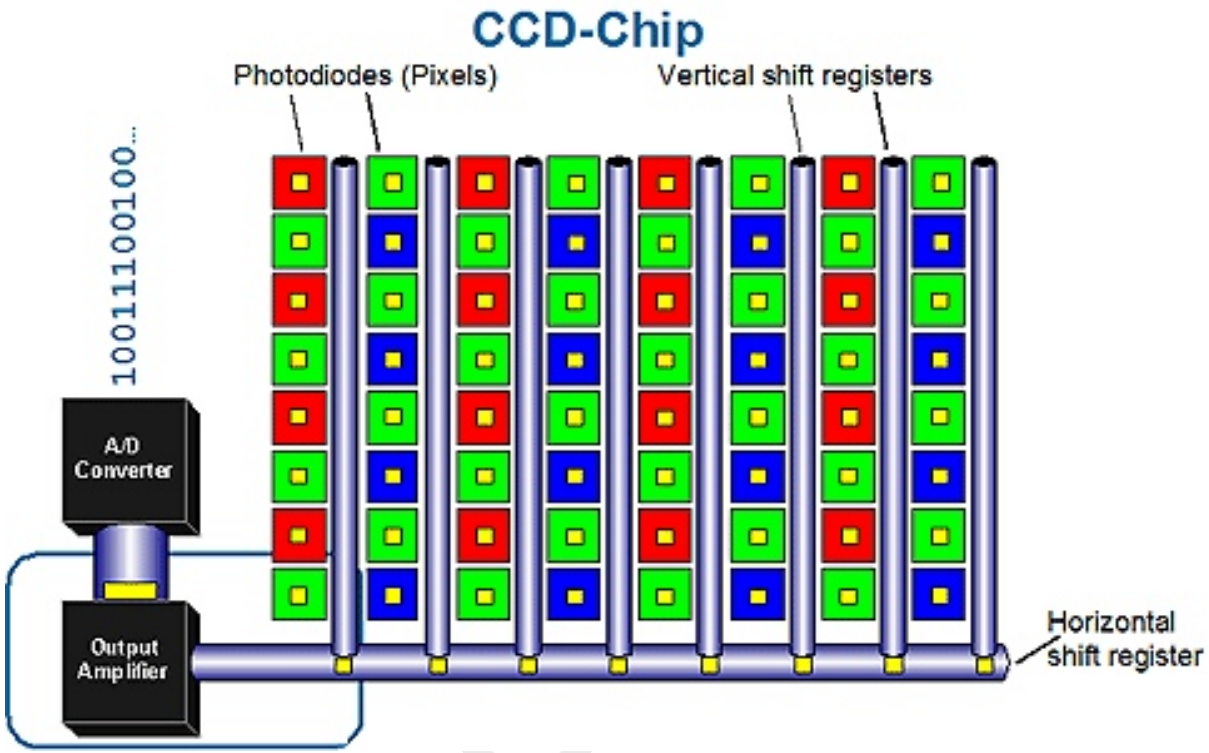
Capturing Images

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

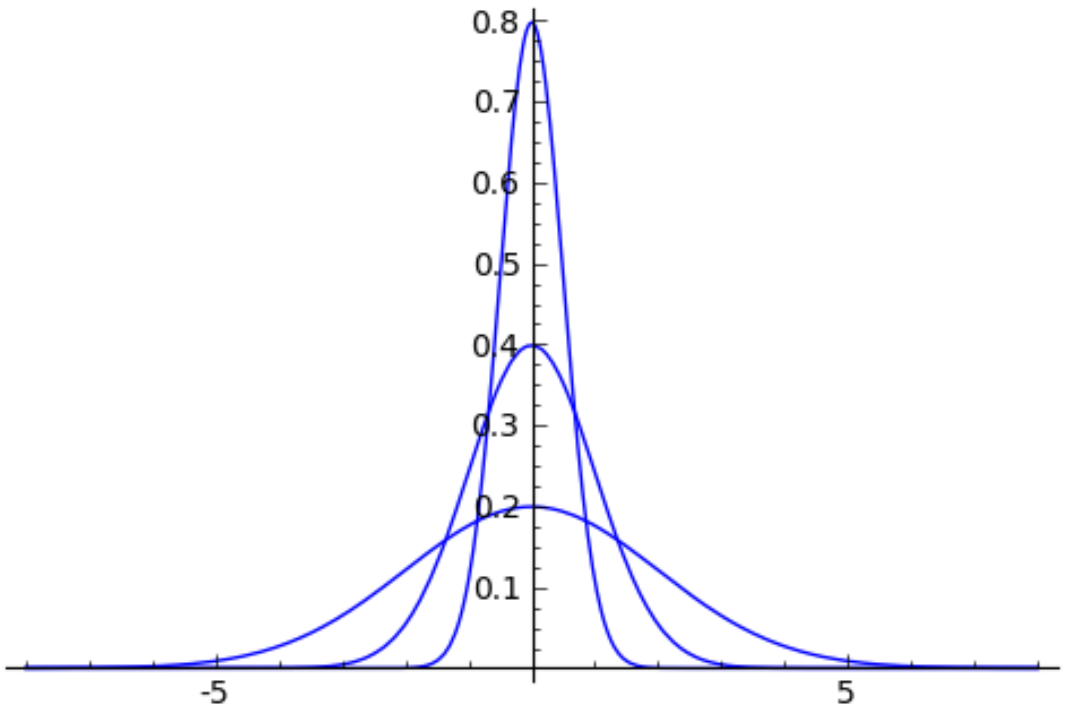
Digital signal processing (DSP) takes the raw data from the sensor and assembles it in correct color space or bitmap structure. In doing so, it handles white balance, brightness, sharpness ,contrast and noise levels.



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

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).

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 similar 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.

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.

An original image (of the world famous Lena) (center), and the results of adding two different types of noise (Gaussian, left and salt and pepper, right). Images taken from https://ece.uwaterloo.ca/∼z70wang/research/quality_index/demo_lena.html.

Denoising Algorithms

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

- Local means.
- Local Medians.
- 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 k-by-k square matrix of coordinates centered at x, y, where k is a small odd integer – typically 3 or 5.

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

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, $N_{3\times 3}(x, y)$).

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 k independent random variables with variance σ, the variance of the average reduces to $\sigma/\sqrt{k}$ ($\sigma/3$ for the $N_{3\times 3}(x, y)$ neighborhood).

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

Local Means: 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 is smaller here.

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 undesireable.

Local Means: Code

```
from numpy import mean,median,array

def local_means (a,k=1):
    """ applies local means to array (NOT image)
        in windows of size 2k+1 by 2k+1 """
    n,m=a.shape
    result=a.copy()
    for i in range (k,n-k):
        for j in range (k,m-k):
            result[i,j] = round(mean(a[i-k:i+k+1,j-k:j+k +1]))
    return result
```

Comments: `result=a.copy()` creates a copy, as opposed to `result=a`, which would point to an identical object. Arrays being mutable, catastrophic consequences would follow.

The code operates only on pixels in the center of a (2k+1)-by-(2k+1) window (default parameter is `k=1`) all of which fits within the matrix. Other, "boundary" pixels, are left intact.

The average is rounded to the nearest integer (`n.5` is rounded down to `n`).

We have used `numpy` function `mean`, expecting it to be more efficient.

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

```
from numpy import mean,median,array

def local_medians (a,k=1):
    """ applies local means to array (NOT image)
        in windows of size 2k+1 by 2k+1 """
    n,m=a.shape
    result=a.copy()
    for i in range (k,n-k):
        for j in range (k,m-k):
            result[i,j] = median(a[i-k:i+k+1,j-k:j+k +1])
    return result
```

Comments: `result=a.copy()` creates a copy, as opposed to `result=a`, which would point to an identical object. Arrays being mutable, catastrophic consequences would follow.

The code operates only on pixels in the center of a (2k+1)-by-(2k+1) window (default parameter is `k=1`) all of which fits within the matrix. Other, "boundary" pixels, are left intact.
Average is rounded to nearest integer (`n.5` is rounded down to `n`).

We have used `numpy` function `median`, expecting it to be more efficient.

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 a small constant, 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 mean. For median, one way to compute it is to sort first. This takes $O(k^2 \log k^2) = O(k^2 \log k)$ steps. But faster median finding, running in time which is linear in the number of items, is known (you probably saw it in the recitation as well). 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: Noisy Lena

```
import numpy
from PIL import Image
from local_denoise import local_means,local_medians

im0 = Image.open("lena_original.jpg")
Lena=im0.convert("L")   # converts to 8 bits black and white

im1 = Image.open("lena_Gaussian.jpg")
Lena_G=im0.convert("L")   # converts to 8 bits black and white

im2 = Image.open("lena_salt_pepper.jpg")
Lena_SP=im0.convert("L")   # converts to 8 bits black and white

print ( Lena.size, Lena.mode)

A= numpy.array(Lena_G)
B= numpy.array(Lena_SP)
A.setflags(write=1)  # makes A writeable
B.setflags(write=1)  # makes B writeable
```

Putting Local Means to the Test: Noisy Lena, cont.

```
A1=local_means(A)
A2=local_means(A,k=2)   # 5-by-5 windows
B1=local_means(B)
B2=local_means(B,k=2)   # 5-by-5 windows


LenaG1 = Image.fromarray(numpy.uint8(A1))
LenaG2 = Image.fromarray(numpy.uint8(A2))
LenaG1.save("LenaG1","jpeg")
LenaG2.save("LenaG2","jpeg")

LenaSP1 = Image.fromarray(numpy.uint8(B1))
LenaSP2 = Image.fromarray(numpy.uint8(B2))
LenaSP1.save("LenaSP1","jpeg")
LenaSP2.save("LenaSP2","jpeg")

im0.show()
im1.show()
im2.show()
LenaG1.show()
LenaG2.show()
LenaSP1.show()
LenaSP2.show()
```

Putting Local Medians to the Test: Noisy Lena

```
import numpy
from PIL import Image
from local_denoise import local_means,local_medians

im0 = Image.open("lena_original.jpg")
Lena=im0.convert("L")   # converts to 8 bits black and white

im1 = Image.open("lena_Gaussian.jpg")
Lena_G=im0.convert("L")   # converts to 8 bits black and white

im2 = Image.open("lena_salt_pepper.jpg")
Lena_SP=im0.convert("L")   # converts to 8 bits black and white

print ( Lena.size, Lena.mode)

A= numpy.array(Lena_G)
B= numpy.array(Lena_SP)
A.setflags(write=1)  # makes A writeable
B.setflags(write=1)  # makes B writeable

A1=local_medians(A)
A2=local_medians(A,k=2)  # 5-by-5 windows
B1=local_medians(B)
B2=local_medians(B,k=2)  # 5-by-5 windows
```

Putting Local Medians to the Test: Noisy Lena, cont.

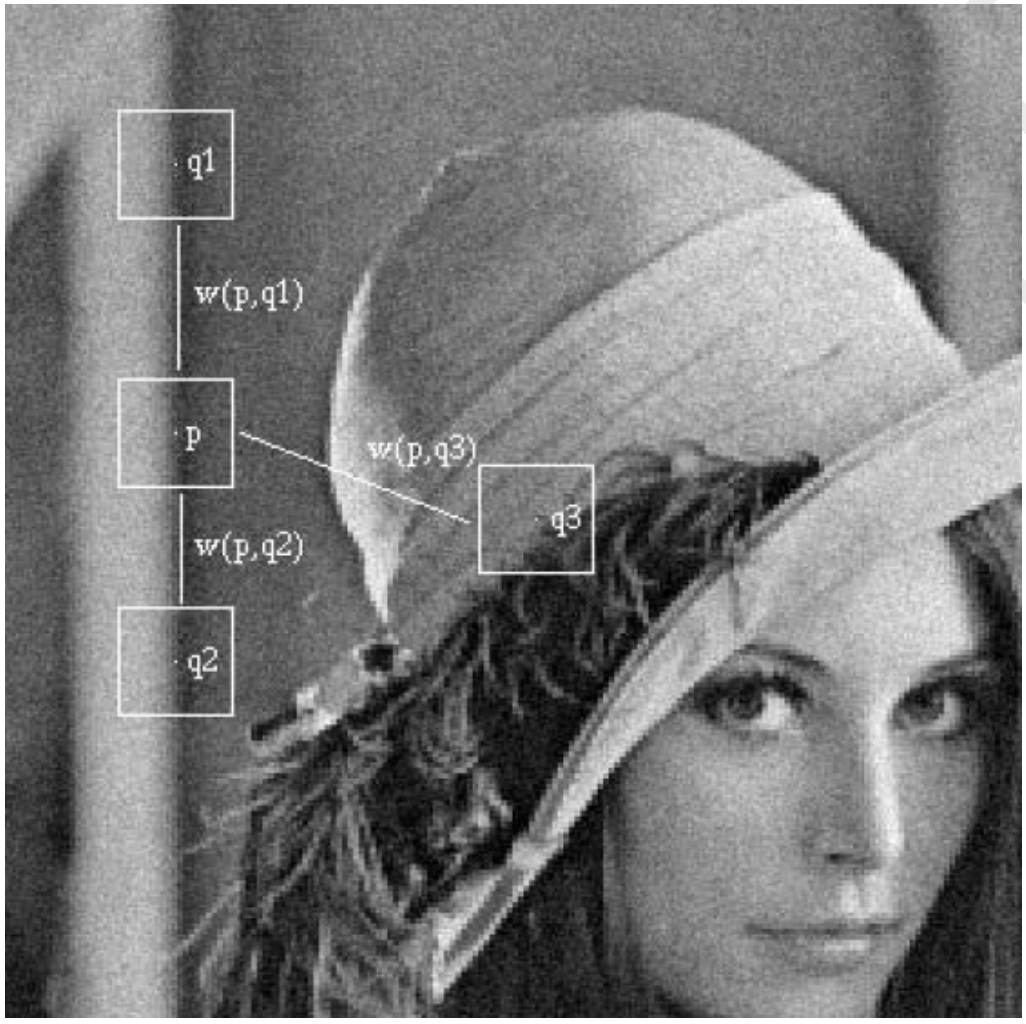
```
LenaG1 = Image.fromarray(numpy.uint8(A1))
LenaG2 = Image.fromarray(numpy.uint8(A2))
LenaG1.save("LenaG1_medians","jpeg")
LenaG2.save("LenaG2_medians","jpeg")

LenaSP1 = Image.fromarray(numpy.uint8(B1))
LenaSP2 = Image.fromarray(numpy.uint8(B2))
LenaSP1.save("LenaSP1_medians","jpeg")
LenaSP2.save("LenaSP2_medians","jpeg")

im0.show()
im1.show()
im2.show()
LenaG1.show()
LenaG2.show()
LenaSP1.show()
LenaSP2.show()
```

Towards Non-Local Means: Regularity in Natural Images

Most 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.



Window centered at p is similar to those centered at q_1 and q_2, but not to the one centered at q_3. Image taken from the NL means paper by A. Buades, B. Coll, and J. M. Morel.

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.