

CS4495/6495

# Introduction to Computer Vision

---

2A-L2 *Filtering*

# Matlab: Row, column vs. x, y

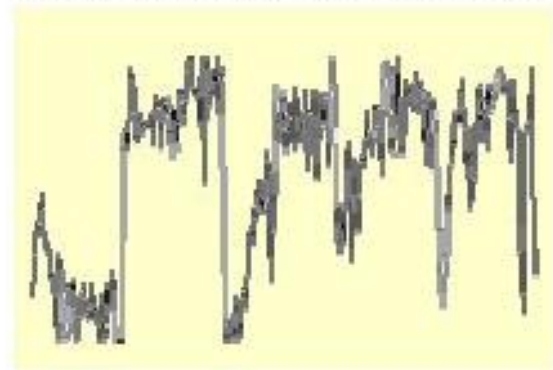
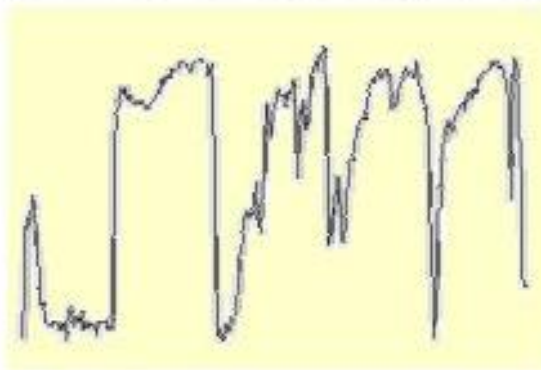
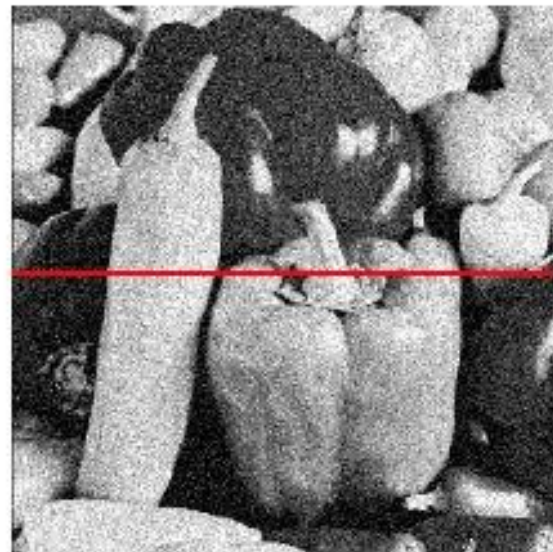
- Rows and Columns are not x,y
- When we say  $I(x,y)$ , we usually mean the x and y that you learned in middle school. X goes to the right; Y goes up. In this case  $I(0,0)$  would be at the bottom left hand corner. Why we say “usually” I’ll talk about in a minute.
- When we say  $F(i,j)$  we usually mean **row  $i$  and column  $j$** . Where rows go down and columns go over. So the origin would be at the top left of an image. So  $F(5,2)$  would be the pixel 5 over and down 2.

# Matlab: Indexing from 1!

- Finally one more confusion to remind you of: in computer science our indexing typically starts with 0. This makes indexing much simpler. And since the target audience for this class are MSCS students I won't explain that.
- But in Matlab the indexing starts at 1. So the origin of the image is  $F(1,1)$  – first row and first column. And when you display an array as an image it is at the top left of an array.
- This will no doubt inflict pain and suffering in a variety of your programs. Also, which way is “positive”  $y$  will also be confusing. In the  $I(x,y)$  notation positive  $Y$  usually means up, and in the  $F(i,j)$  – where  $i$  is the row value or the  $y$  value – positive  $Y$  goes down. Again, this will likely break your code somewhere during the course. So if your code is doing strange things relative to up and down, check your indexing.

# Gaussian noise

```
noise =  
    randn(size(im)).*sigma;  
output = im + noise;
```



$$f(x, y) = \overbrace{\hat{f}(x, y)}^{\text{Ideal Image}} + \overbrace{\eta(x, y)}^{\text{Noise process}}$$

Gaussian i.i.d. ("white") noise:  
 $\eta(x, y) \sim \mathcal{N}(\mu, \sigma)$

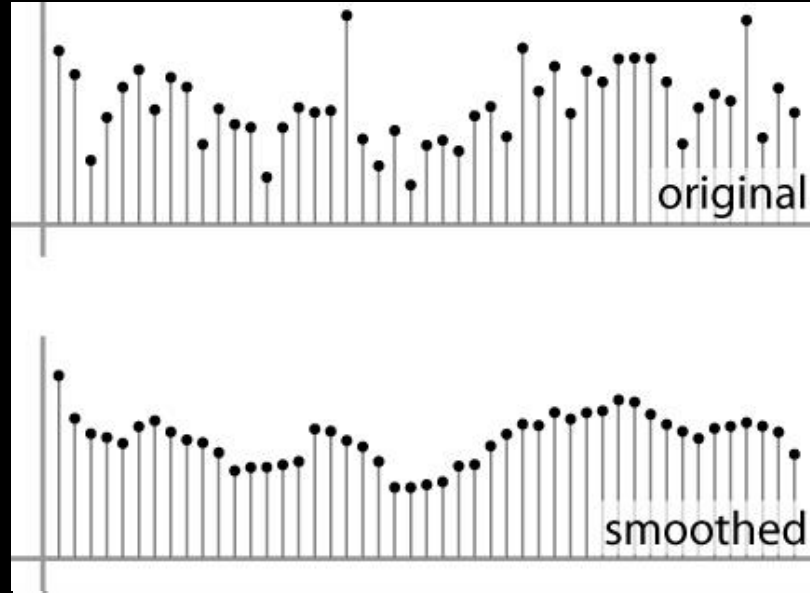
# How do we remove noise?

Now I'm sure you all have a suggestion. The typical one is "Lets replace each pixel with the average of the values around it – 'in the neighborhood'".

Illustrating that in 1D it would look something like this:

# First attempt at a solution – 1D

Replace each pixel with an average of all the values in its neighborhood – a *moving average*:



# Thoughts

- OK, seems reasonable. But the important question to ask is why was that an intuitive solution for many of you? What **assumptions** were you making about the image and the noise?
- A little thought and you'll realize that we have to make two key assumptions for the averaging method to make sense

# Averaging assumptions

1. The “true” value of pixels are similar to the true value of pixels nearby.
2. The noise added to each pixel is done independently.



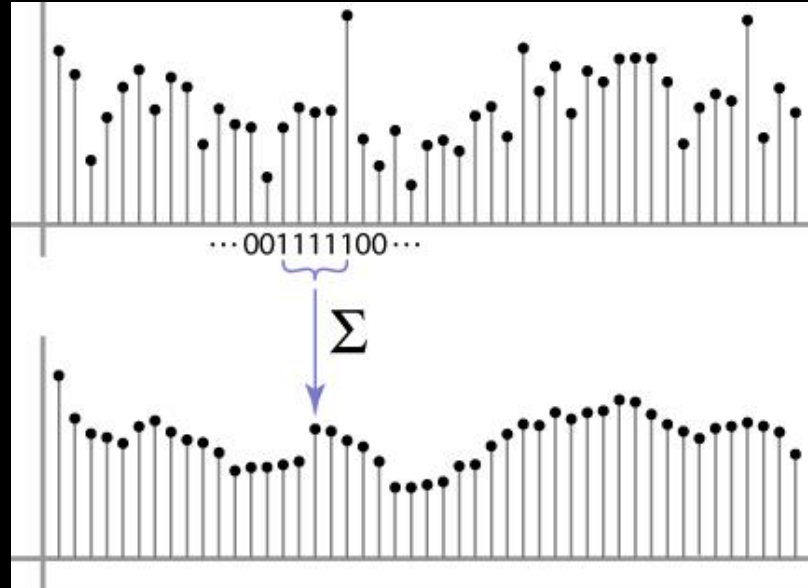
# Quiz

If noise is just a function added to an image, we could remove the noise by subtracting the noise again – that is the operation is reversible:

- a) True
- b) True but we don't know the noise function so we can't actually do the subtraction.
- c) False. Additive noise destroys information in the image and there is no way to recover it.

# Weighted Moving Average

- Can add weights to our moving average
- *Weights*  $[1, 1, 1, 1, 1] / 5$

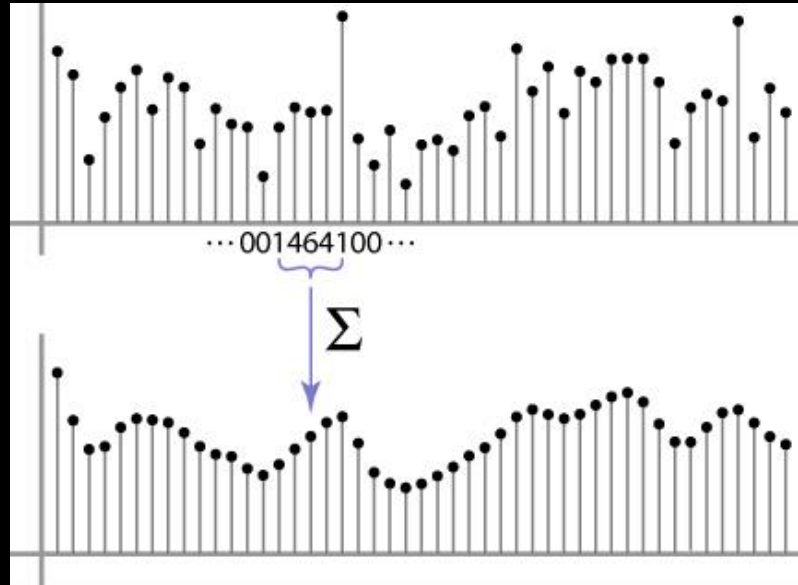


# Weighted Moving Average

- Let's examine those assumptions a bit more, especially the one about “smoothness”.
  - The basic idea is that nearby pixels have similar true underlying values.
  - Which, under pretty reasonable assumptions, implies that the closer a pixel is to some reference pixel, the more similar it would be.
  - So the more it should contribute to an average.

# Weighted Moving Average

- Non-uniform weights [1, 4, 6, 4, 1] / 16



# Quiz

To do the moving average computation the number of weights should be

- a) Odd – makes it easier to have a middle pixel
- b) Even – that way the filter can be exactly symmetric around a pixel.
- c) Either even or odd.

# Moving Average In 2D

$F(x, y)$

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	0	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	0	0	0	0	0	0	0
0	0	90	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

$G(x, y)$

	0								

# Moving Average In 2D

$$F(x, y)$$
[illegible]
$$G(x, y)$$

A 10x10 grid representing a coordinate plane. The x-axis is labeled from 0 to 9, and the y-axis is labeled from 0 to 9. A red square highlights the cell at coordinates (1, 1), which contains the number 10.

# Moving Average In 2D

$$F(x, y)$$

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	90	90	90	90	90	0
0	0	0	90	90	90	90	90	0
0	0	0	90	90	90	90	90	0
0	0	0	90	0	90	90	90	0
0	0	0	90	90	90	90	90	0
0	0	0	0	0	0	0	0	0
0	0	90	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

$$G(x, y)$$
[illegible]



# Moving Average In 2D

$$F(x, y)$$
[illegible]
$$G(x, y)$$
[illegible]

# Moving Average In 2D

$$F(x, y)$$
[illegible]
$$G(x, y)$$
[illegible]

# Moving Average In 2D

$$F(x, y)$$
[illegible]
$$G(x, y)$$
[illegible]

# Correlation filtering – uniform weights

Say the averaging window size is  $2k+1 \times 2k+1$ :

$$G[i, j] = \frac{1}{(2k + 1)^2} \sum_{u=-k}^k \sum_{v=-k}^k F[i + u, j + v]$$

*Uniform  
weight for  
each pixel*

*Loop over all pixels in  
neighborhood around  
image pixel  $F[i, j]$*

# Correlation filtering – nonuniform weights

Now generalize to allow **different weights** depending on neighboring pixel's relative position:

$$G[i, j] = \sum_{u=-k}^k \sum_{v=-k}^k \underbrace{H[u, v]}_{\text{Non-uniform weights}} F[i + u, j + v]$$

*Non-uniform weights*

This is called **cross-correlation**, denoted  $G = H \otimes F$

# Correlation filtering – nonuniform weights

Now generalize to allow **different weights** depending on neighboring pixel's relative position:

$$G[i, j] = \sum_{u=-k}^k \sum_{v=-k}^k \underbrace{H[u, v]}_{\text{Non-uniform weights}} F[i + u, j + v]$$

*Non-uniform weights*

The filter “**kernel**” or “**mask**”  $H[u, v]$  is the matrix of weights in the linear combination.

# Quiz

- The weights and shape of the filter are called the
  - a) Mask
  - b) Coefficients
  - c) Kernel
  - d) All of the above

# What makes a good kernel or filter?

- Consider the example of a uniform (averaging) filter...



# Averaging filter

$$F(x, y) \otimes H(u, v) = G(x, y)$$

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	0	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	0	0	0	0	0	0	0
0	0	90	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

1	1	1
1	1	1
1	1	1

$\frac{1}{9}$

“box filter”

?

$$G = F \otimes H$$

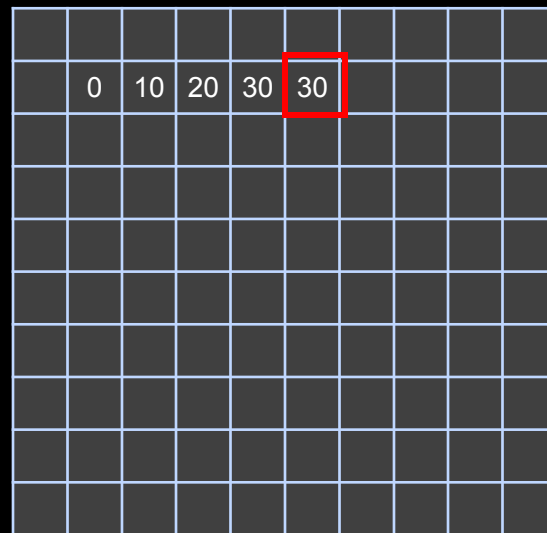
# Averaging filter

$$F(x, y) \otimes H(u, v) = G(x, y)$$


$$\frac{1}{9}$$

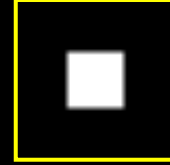
1	1	1
1	1	1
1	1	1

“box filter”

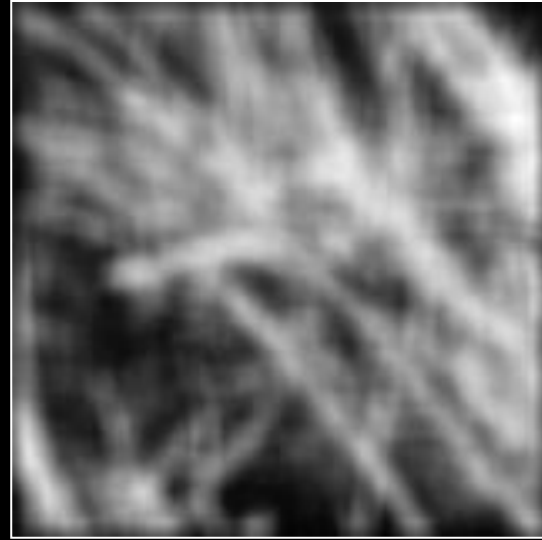


$$G = F \otimes H$$

# Smoothing by box averaging



**original**



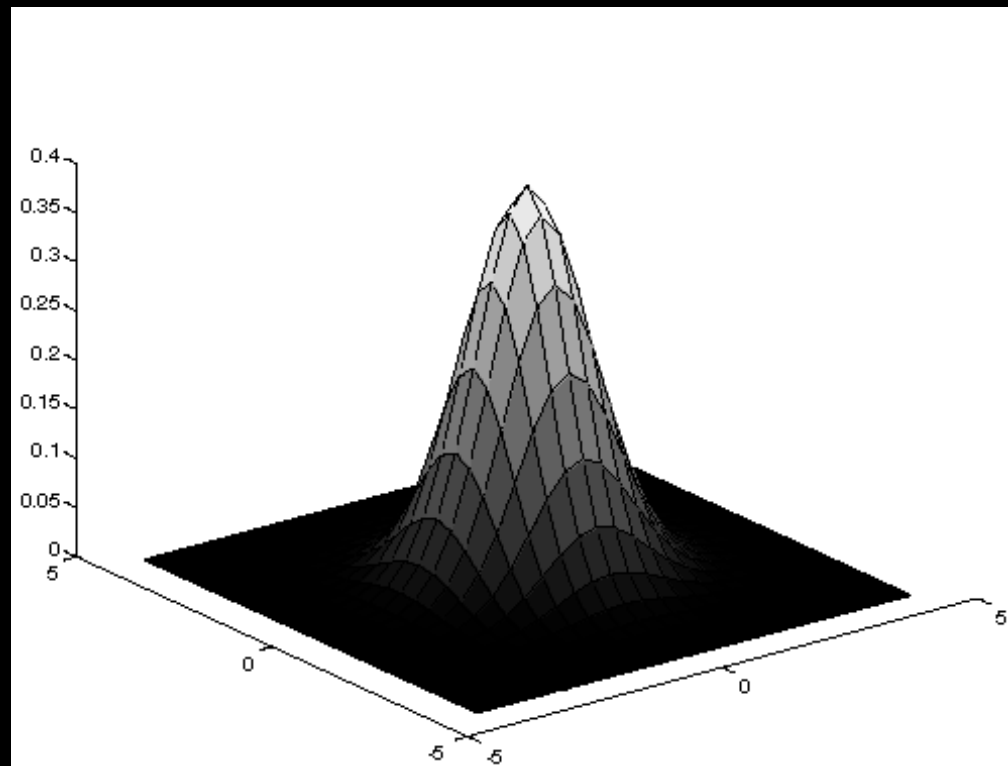
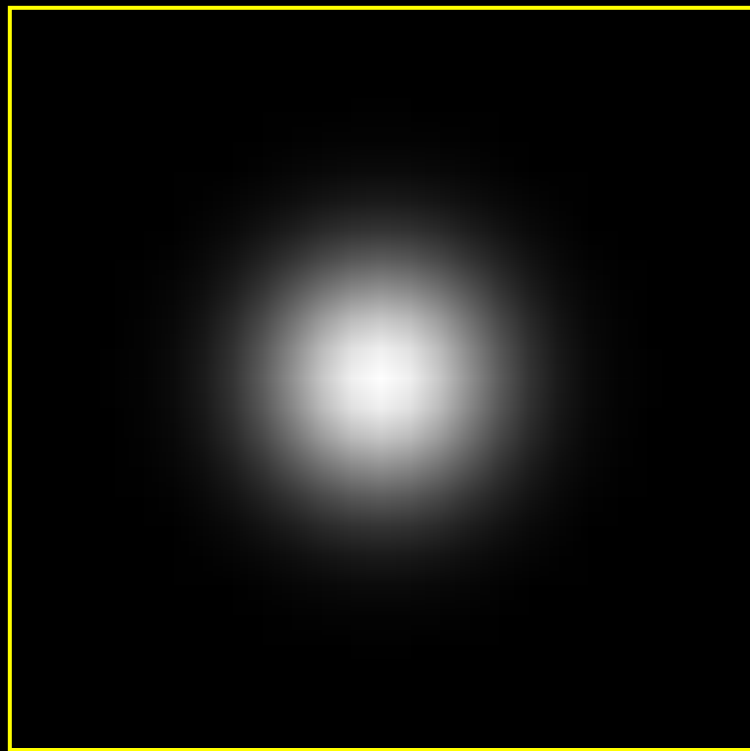
**filtered**

# Averaging filter: Drawbacks

- The problem: squares aren't smooth. And filtering an image with a filter that is not "smooth" seems wrong if we're trying to "blur" the image. – (We'll say more about what smooth really means in a few lectures.)
- So what's the problem? To get a sense of what's wrong, let's think about what a single spot of light viewed by an out of focus camera would look like. The image would look something like this:



# Blurry spot as a function



# Quiz

To blur a single pixel into a “blurry” spot, we would need to need to filter the spot with a

- a) 3x3 square of uniform weights
- b) 11x11 square of uniform weights would be better since it's bigger
- c) Something that looks like a blurry spot – higher values in the middle, falling off to the edges.

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	0	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	0	0	0	0	0	0	0
0	0	90	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

$$F(x, y)$$



0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	0	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	0	0	0	0	0	0	0
0	0	90	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

$F(x, y)$

$\frac{1}{16}$

1	2	1
2	4	2
1	2	1

$H(u, v)$

# Gaussian filter

- Nearest neighboring pixels have the most influence

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	0	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	90	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

$F(x, y)$

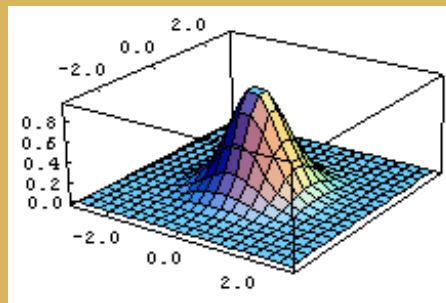
$\frac{1}{16}$

1	2	1
2	4	2
1	2	1

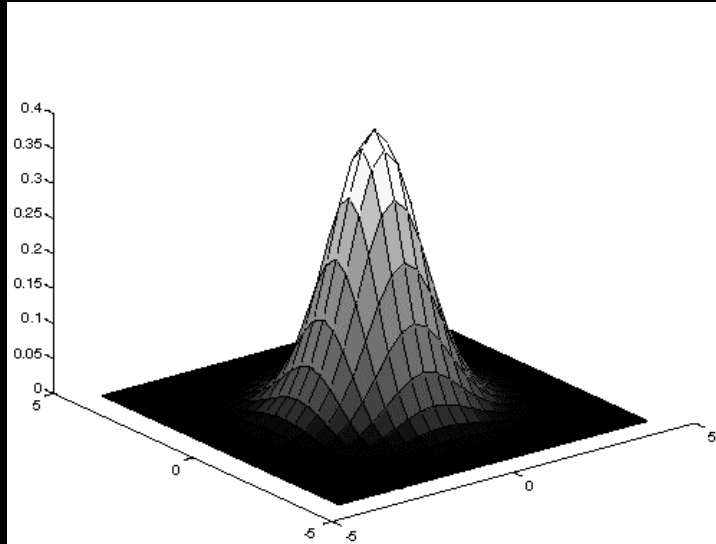
$H(u, v)$

This kernel is an approximation of a Gaussian function:

$$h(u, v) = \frac{1}{2\pi\sigma^2} e^{-\frac{u^2+v^2}{\sigma^2}}$$

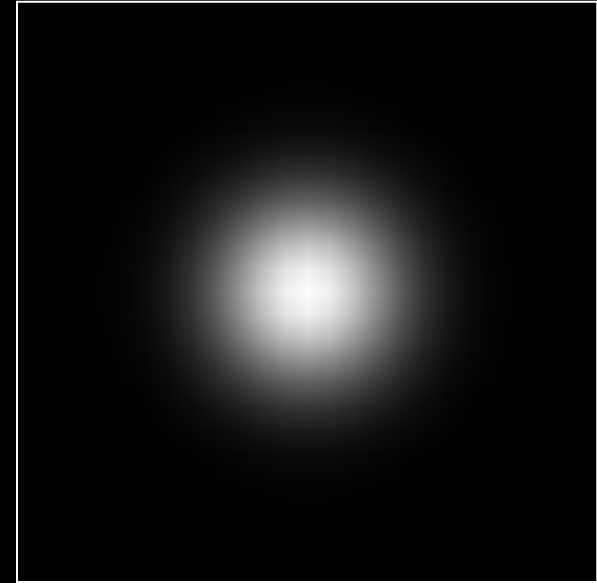


# An Isotropic Gaussian



Smoothing kernel proportional  
to:

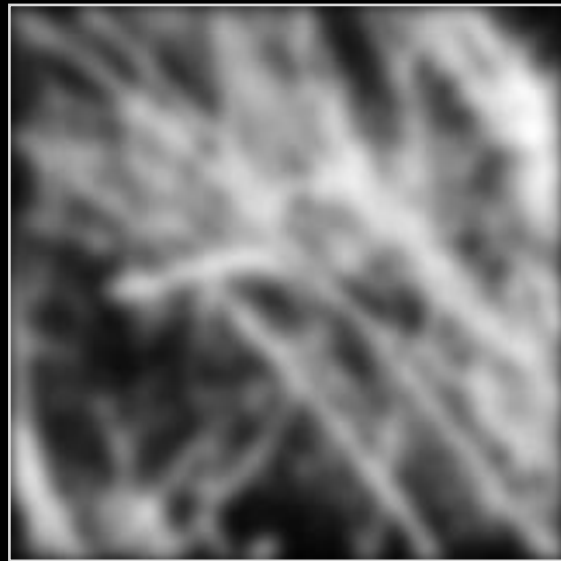
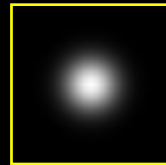
$$\exp\left(-\frac{(x^2 + y^2)}{2\sigma^2}\right)$$



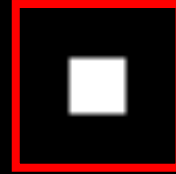
“circularly symmetric  
fuzzy blob”

D. Forsyth

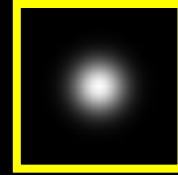
# Smoothing with a Gaussian



# Smoothing with not a Gaussian



# Smoothing with a Gaussian



# Quiz

Gaussian filters are referred to as exponentials. The complete formula is:

a) is 
$$h(u, v) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right)$$

b) Or 
$$h(u, v) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right)$$

c) Or even: 
$$h(u, v) = \frac{1}{\sqrt{2\pi\sigma}} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right)$$

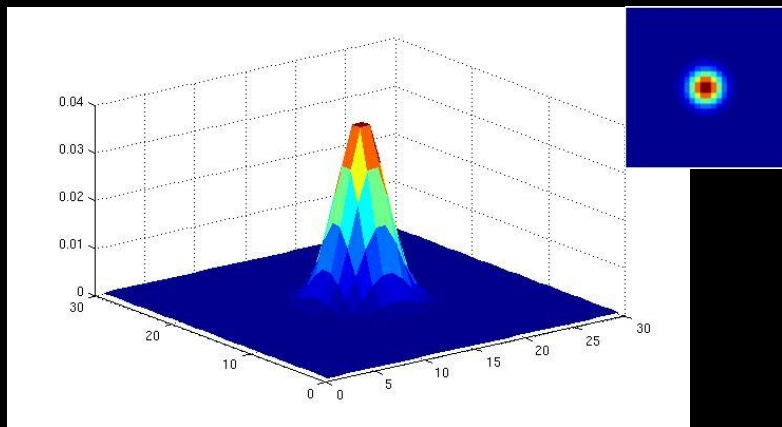
# Gaussian filter: Shape vs. size

- The Gaussian we showed was isotropic – circular – so it had only one parameter (sigma).
- But that is much easier to say when we're doing continuous math than when working on a computer.
- On a computer, we need to talk about the *size* of the filter (3x3, 5x5, 11x11) and then the *shape* of the filter values (Gaussian distribution).

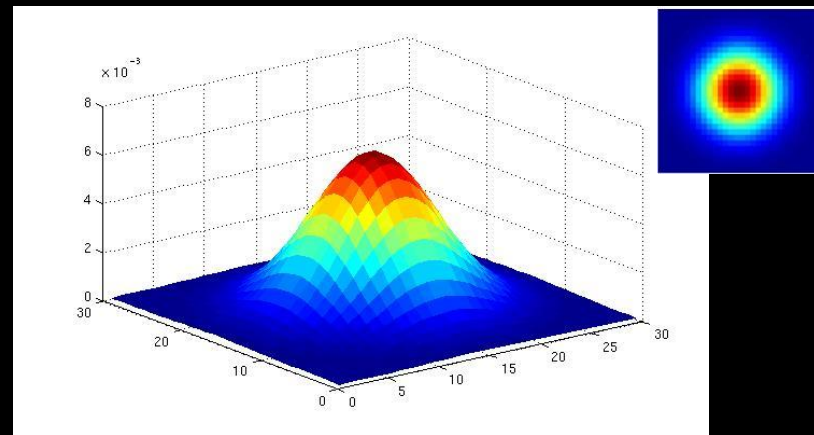


# Gaussian filters

**Variance ( $\sigma^2$ )** or **standard deviation ( $\sigma$ )** –  
determines extent of smoothing



$\sigma = 2$  with  $30 \times 30$  kernel

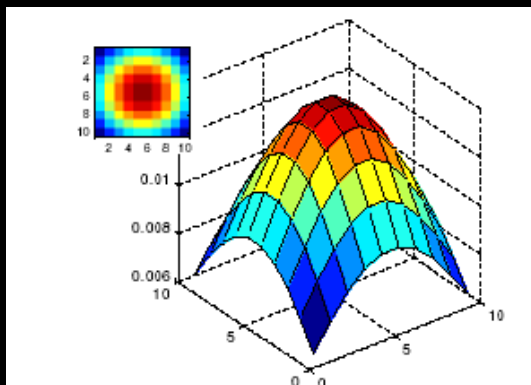


$\sigma = 5$  with  $30 \times 30$  kernel

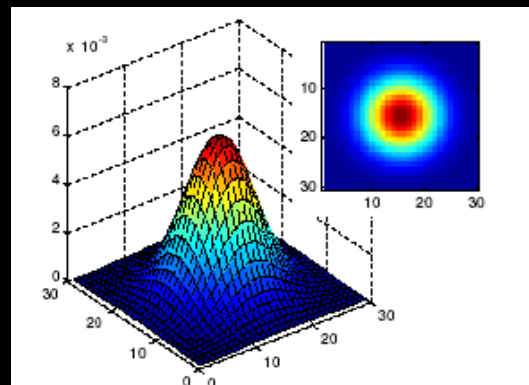
K. Grauman

# Gaussian filters

Size of kernel or mask is *not* variance



$\sigma = 5$  with  $10 \times 10$   
kernel

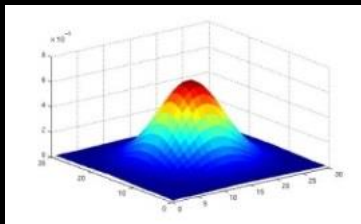


$\sigma = 5$  with  $30 \times 30$   
kernel

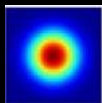
# Matlab

```
>> hsize = 31;  
>> sigma = 5;  
>> h = fspecial('gaussian', hsize, sigma);
```

```
>> surf(h);
```



```
>> imagesc(h);
```



```
>> outim = imfilter(im, h);  
>> imshow(outim);
```



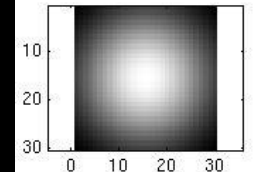
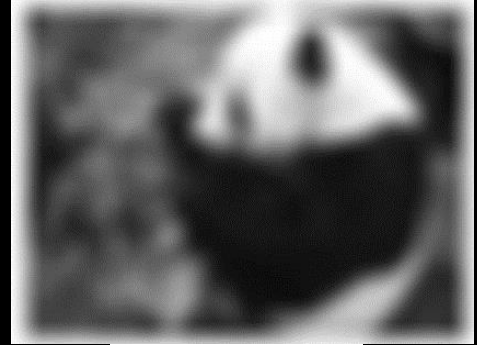
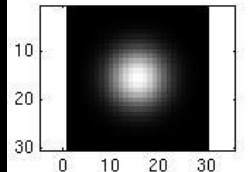
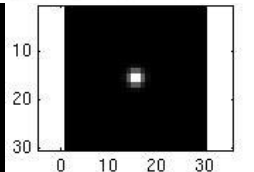
**im**



**outim**

# Smoothing with a Gaussian

```
for sigma=1:3:10
    h = fspecial('gaussian',
        fsize, sigma);
    out = imfilter(im, h);
    imshow(out);
    pause;
end
```



# Quiz

When filtering with a Gaussian, which is true:

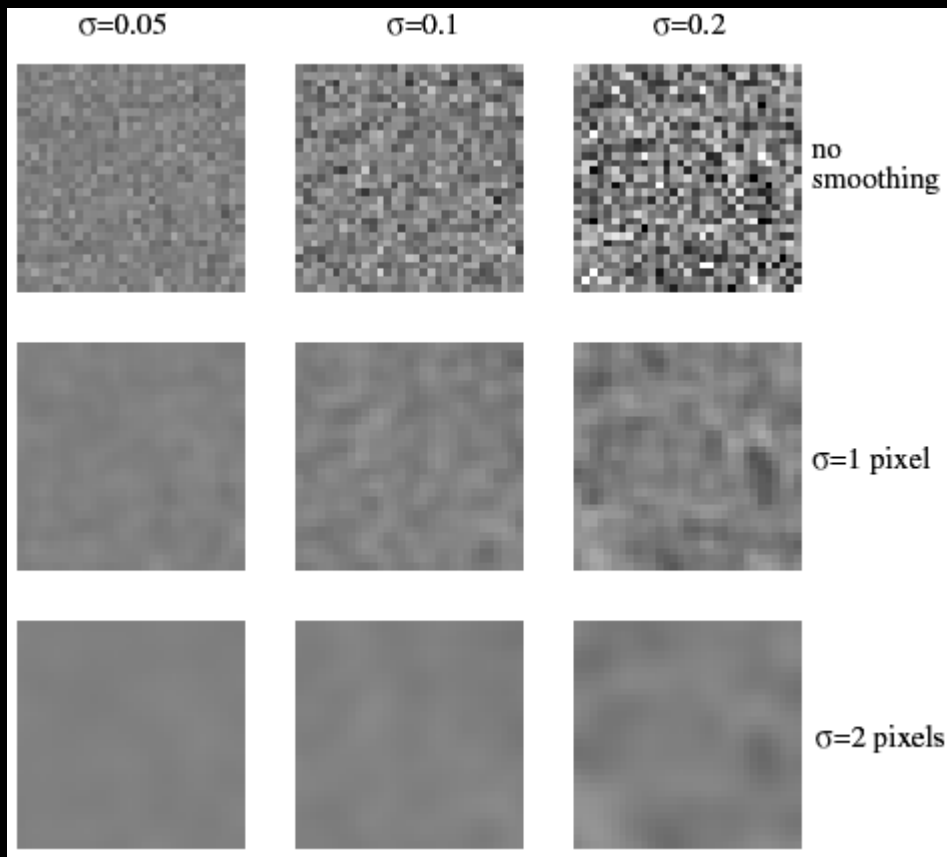
- a) The sigma is most important – it defines the blur kernel's scale with respect to the image.
- b) The kernel size is most important – because it defines the scale.
- c) Altering the normalization coefficient does not effect the blur, only the brightness.
- d) A and C.

# A tale of two Gaussians

- Finally, a quick word of warning or clarification.
- We have just talked about sigma as a width of a the Gaussian filter. Last time we talked about ***sigma*** as the ***variance*** of a noise function.
- In one case – the filter – sigma is a width in space where as with noise it's a variance in ***value***. The bigger the noise sigma was the more likely that large values of noise can be created.
- For noise it's usually sigma squared for variance; for filtering usually sigma for size.

# Keeping the two Gaussians straight...

More Gaussian noise (like earlier)  $\sigma \rightarrow$



Wider Gaussian smoothing kernel  $\sigma \rightarrow$