Convolutions and Canny Edge Detection

CSE 4310 – Computer Vision
Vassilis Athitsos
Computer Science and Engineering Department
University of Texas at Arlington

Convolution/Linear Filtering

- Linear filtering, or convolution, is an extremely common image operation.
- This operation requires an image and a kernel.
- Suppose that:
  - kernel has 2M+1 rows, 2N+1 cols.
  - input is the original image, of U rows and V cols.
  - result is the output of convolution.
- Then convolution can be coded as:

```
result = zeros(U, V);
for i = (M+1):(U-M)
    for j = (N+1):(V-N)
        result(i,j) = sum(sum(kernel .* input((i-M):(i+M), (j-N):(j+N))));
    end
end
```

Convolution/Linear Filtering

- What does .* do in the Matlab code below?

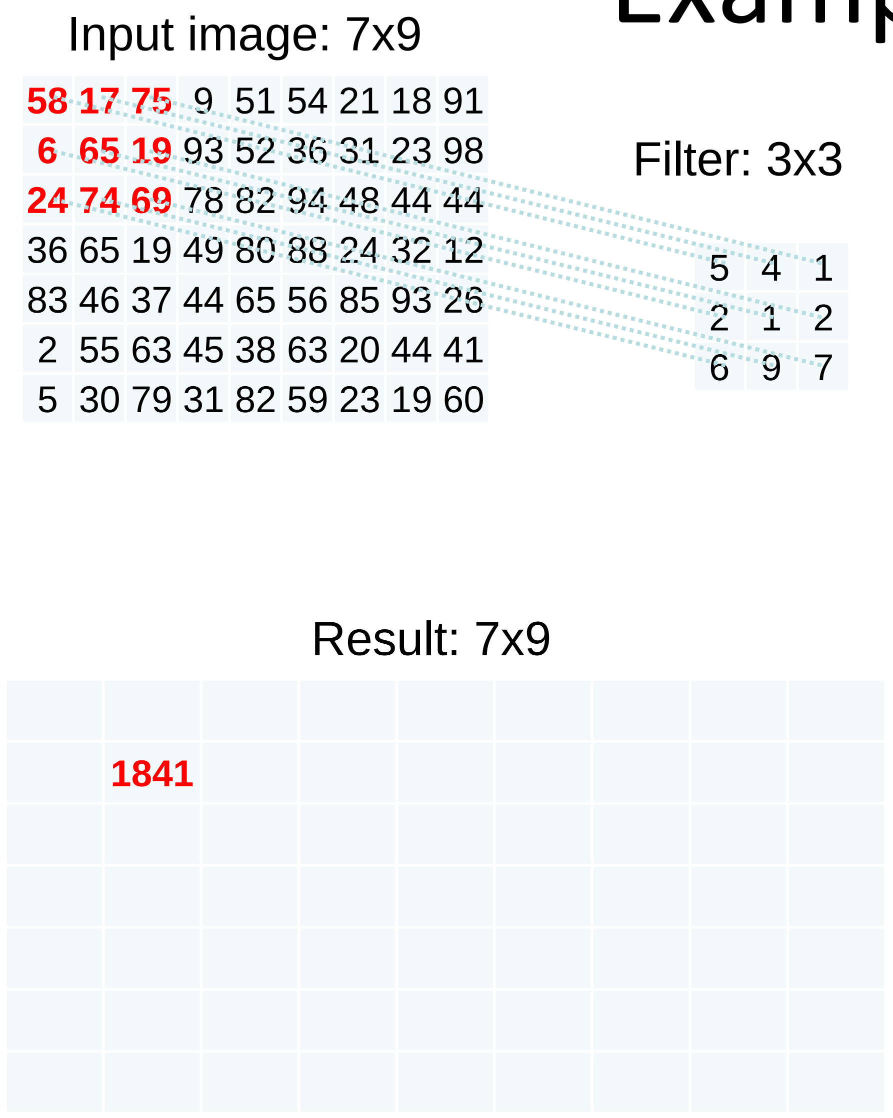
```
result = zeros(U, V);
for i = (M+1):(U-M)
    for j = (N+1):(V-N)
        result(i,j) = sum(sum(kernel .* input((i-M):(i+M), (j-N):(j+N))));
    end
end
```

Convolution/Linear Filtering

- What does .* do in the Matlab code below?
- It means “pointwise multiplication”:
  - multiply each entry of one matrix with the entry at the same location on the other matrix.
  - put the product of those two entries at the same location on the result matrix.

```
result = zeros(U, V);
for i = (M+1):(U-M)
    for j = (N+1):(V-N)
        result(i,j) = sum(sum(kernel .* input((i-M):(i+M), (j-N):(j+N))));
    end
end
```

Example With Numbers



Input image: 7x9

58	**17**	**75**	9	51	54	21	18	91
6	**65**	**19**	93	52	36	31	23	98
24	**74**	**69**	78	82	94	48	44	44
36	65	19	49	80	88	24	32	12
83	46	37	44	65	56	85	93	26
2	55	63	45	38	63	20	44	41
5	30	79	31	82	59	23	19	60

Filter: 3x3

5	4	1
2	1	2
6	9	7

Result: 7x9

	1841							

- For the time being, we postpone the discussion of how to compute values at the boundaries (top and bottom row, left and right column).
- We assume that the filter has an odd number of rows, and an odd number of columns.
  - This way, the center of a filter corresponds to a specific position on the filter (position 2,2 for a 3x3 filter).

Example With Numbers

Input image: 7x9

58	**17**	**75**	9	51	54	21	18	91
6	**65**	**19**	93	52	36	31	23	98
24	**74**	**69**	78	82	94	48	44	44
36	65	19	49	80	88	24	32	12
83	46	37	44	65	56	85	93	26
2	55	63	45	38	63	20	44	41
5	30	79	31	82	59	23	19	60

Filter: 3x3

5	4	1
2	1	2
6	9	7

- How do we compute result(2,2)?
  - Align the filter with the image, so that the center of the filter aligns with position (2,2) of the image.
- We must sum up all these values:

58*5	17*4	75*1
6*2	65*1	19*2
24*6	74*9	69*7

- result(2,2): 1841

Result: 7x9

	1841							

Example With Numbers

Input image: 7x9

58	**17**	**75**	**9**	51	54	21	18	91
6	**65**	**19**	**93**	52	36	31	23	98
24	**74**	**69**	**78**	82	94	48	44	44
36	65	19	49	80	88	24	32	12
83	46	37	44	65	56	85	93	26
2	55	63	45	38	63	20	44	41
5	30	79	31	82	59	23	19	60

Filter: 3x3

5	4	1
2	1	2
6	9	7

- How do we compute result(2,3)?

Result: 7x9

	1841	**?**						

Example With Numbers

Input image: 7x9

58	**17**	**75**	**9**	51	54	21	18	91
6	**65**	**19**	**93**	52	36	31	23	98
24	**74**	**69**	**78**	82	94	48	44	44
36	65	19	49	80	88	24	32	12
83	46	37	44	65	56	85	93	26
2	55	63	45	38	63	20	44	41
5	30	79	31	82	59	23	19	60

Filter: 3x3

5	4	1
2	1	2
6	9	7

- How do we compute result(2,3)?
  - Align the filter with the image, so that the center of the filter aligns with position (2,3) of the image.
- We must sum up all these values:

17*5	75*4	9*1
65*2	19*1	93*2
74*6	69*9	78*7

- result(2,3): 2340

Result: 7x9

	1841	**2340**						

Example With Numbers

Input image: 7x9

58	17	75	9	51	54	21	18	91
6	**65**	**19**	**93**	52	36	31	23	98
24	**74**	**69**	**78**	82	94	48	44	44
36	**65**	**19**	**49**	80	88	24	32	12
83	46	37	44	65	56	85	93	26
2	55	63	45	38	63	20	44	41
5	30	79	31	82	59	23	19	60

Filter: 3x3

5	4	1
2	1	2
6	9	7

Result: 7x9

	1841	2340						
		1771						

- How do we compute result(3,3)?
  - Align the filter with the image, so that the center of the filter aligns with position (3,3) of the image.
- We must sum up all these values:

65*5	19*4	93*1
74*2	69*1	78*2
65*6	19*9	49*7

- result(3,3): 1771

Example With Numbers

Input image: 7x9

58	17	75	9	51	54	21	18	91
6	**65**	**19**	**93**	52	36	31	23	98
24	**74**	**69**	**78**	82	94	48	44	44
36	**65**	**19**	**49**	80	88	24	32	12
83	46	37	44	65	56	85	93	26
2	55	63	45	38	63	20	44	41
5	30	79	31	82	59	23	19	60

Filter: 3x3

5	4	1
2	1	2
6	9	7

- How do we compute result(i,j)?
  - Align the filter with the image, so that the center of the filter aligns with position (i,j) of the image.
  - Multiply each value of the filter with the corresponding value in the image.
  - Sum up the results.

Result: 7x9

	1841	2340	2387	2477	2368	1825	1541	
	1503	**1771**	2014	2765	2229	1619	1089	
	1831	1888	2059	2407	2619	2722	2093	
	1693	1879	1668	1971	2067	1817	1378	
	1674	1793	2000	2127	1997	1641	1718	

Boundary Values

Input image: 7x9

58	**17**	75	9	51	54	21	18	91
6	**65**	19	93	52	36	31	23	98
24	74	69	78	82	94	48	44	44
36	65	19	49	80	88	24	32	12
83	46	37	44	65	56	85	93	26
2	55	63	45	38	63	20	44	41
5	30	79	31	82	59	23	19	60

Filter: 3x3

5	4	1
2	1	2
6	9	7

Result: 7x9

	1841	2340	2387	2477	2368	1825	1541	
	1503	1771	2014	2765	2229	1619	1089	
	1831	1888	2059	2407	2619	2722	2093	
	1693	1879	1668	1971	2067	1817	1378	
	1674	1793	2000	2127	1997	1641	1718	

- What do we do at position 1, 1?
  - According to previous instructions: "Align the filter with the image, so that the center of the filter aligns with position (1,1) of the image."
  - Problem: some part of the filter does not align with any values in the image.
- The same issue arises at the top and bottom row, and the left and right column.
  - If the filter had size 5x7, we would have this problem at the top 2 rows, bottom 2 rows, left 3 columns, right 3 columns.

Boundary Values

Input image: 7x9

58	17	75	9	51	54	21	18	91
6	65	19	93	52	36	31	23	98
24	74	69	78	82	94	48	44	44
36	65	19	49	80	88	24	32	12
83	46	37	44	65	56	85	93	26
2	55	63	45	38	63	20	44	41
5	30	79	31	82	59	23	19	60

Filter: 3x3

5	4	1
2	1	2
6	9	7

Result: 7x9

	1841	2340	2387	2477	2368	1825	1541	
	1503	1771	2014	2765	2229	1619	1089	
	1831	1888	2059	2407	2619	2722	2093	
	1693	1879	1668	1971	2067	1817	1378	
	1674	1793	2000	2127	1997	1641	1718	

- Main answer: **we do not care about boundary values**.
- A convolution is usually an intermediate step.
  - Some other module will use the convolution result to do something.
  - It is up to that other module to make sure that it ignores boundary values.
- In practice, since we do not care, we can follow various conventions, see next slides.

Boundary Values

Input image: 7x9

58	**17**	75	9	51	54	21	18	91
6	**65**	19	93	52	36	31	23	98
24	74	69	78	82	94	48	44	44
36	65	19	49	80	88	24	32	12
83	46	37	44	65	56	85	93	26
2	55	63	45	38	63	20	44	41
5	30	79	31	82	59	23	19	60

Filter: 3x3

5	4	1
2	1	2
6	9	7

- Option 1: just use image values that align with filter values.
- So, for position (1,1) of the result, we sum up:

	58*1	17*2
	6*9	65*7

Result: 7x9

601								
	1841	2340	2387	2477	2368	1825	1541	
	1503	1771	2014	2765	2229	1619	1089	
	1831	1888	2059	2407	2619	2722	2093	
	1693	1879	1668	1971	2067	1817	1378	
	1674	1793	2000	2127	1997	1641	1718	

Boundary Values

Input image: 7x9

58	**17**	**75**	9	51	54	21	18	91
6	**65**	**19**	93	52	36	31	23	98
24	74	69	78	82	94	48	44	44
36	65	19	49	80	88	24	32	12
83	46	37	44	65	56	85	93	26
2	55	63	45	38	63	20	44	41
5	30	79	31	82	59	23	19	60

Filter: 3x3

5	4	1
2	1	2
6	9	7

- Option 1: just use image values that align with filter values.
- So, for position (1,2) of the result, we sum up:

58*2	17*1	75*2
6*6	65*9	19*7

Result: 7x9

601	**1037**							
	1841	2340	2387	2477	2368	1825	1541	
	1503	1771	2014	2765	2229	1619	1089	
	1831	1888	2059	2407	2619	2722	2093	
	1693	1879	1668	1971	2067	1817	1378	
	1674	1793	2000	2127	1997	1641	1718	

Boundary Values

Input image: 7x9

58	**17**	**75**	9	51	54	21	18	91
6	**65**	**19**	93	52	36	31	23	98
24	74	69	78	82	94	48	44	44
36	65	19	49	80	88	24	32	12
83	46	37	44	65	56	85	93	26
2	55	63	45	38	63	20	44	41
5	30	79	31	82	59	23	19	60

Filter: 3x3

5	4	1
2	1	2
6	9	7

- Option 1: just use image values that align with filter values.
- So, for position (1,2) of the result, we sum up:

58*2	17*1	75*2
6*6	65*9	19*7

Result: 7x9

601	**1037**	1339	1576	1455	1051	821	1321	1147
1119	1841	2340	2387	2477	2368	1825	1541	1258
1040	1503	1771	2014	2765	2229	1619	1089	939
1405	1831	1888	2059	2407	2619	2722	2093	1264
787	1693	1879	1668	1971	2067	1817	1378	1053
745	1674	1793	2000	2127	1997	1641	1718	1352
128	491	773	886	702	731	618	502	482

- This way we can fill up all boundary values.

Boundary Values

Input image: 7x9

58	17	75	9	51	54	21	18	91
6	65	19	93	52	36	31	23	98
24	74	69	78	82	94	48	44	44
36	65	19	49	80	88	24	32	12
83	46	37	44	65	56	85	93	26
2	55	63	45	38	63	20	44	41
5	30	79	31	82	59	23	19	60

Filter: 3x3

5	4	1
2	1	2
6	9	7

- Option 2: just put zeros at the boundary.

Result: 7x9

0	0	0	0	0	0	0	0	0
0	1841	2340	2387	2477	2368	1825	1541	0
0	1503	1771	2014	2765	2229	1619	1089	0
0	1831	1888	2059	2407	2619	2722	2093	0
0	1693	1879	1668	1971	2067	1817	1378	0
0	1674	1793	2000	2127	1997	1641	1718	0
0	0	0	0	0	0	0	0	0

Boundary Values

Input image: 7x9

58	17	75	9	51	54	21	18	91
6	65	19	93	52	36	31	23	98
24	74	69	78	82	94	48	44	44
36	65	19	49	80	88	24	32	12
83	46	37	44	65	56	85	93	26
2	55	63	45	38	63	20	44	41
5	30	79	31	82	59	23	19	60

Filter: 3x3

5	4	1
2	1	2
6	9	7

Result: 5x7

1841	2340	2387	2477	2368	1825	1541
1503	1771	2014	2765	2229	1619	1089
1831	1888	2059	2407	2619	2722	2093
1693	1879	1668	1971	2067	1817	1378
1674	1793	2000	2127	1997	1641	1718

- Option 3: remove the boundary altogether.
  - Then, the result will have a smaller size than the original image.
- In this class, we will avoid this option, because it makes it more difficult to relate a value in the result with a position in the original image.
  - With the other two options, result(i,j) corresponds to the value we get from the image region centered at position (i,j).

Intuition on Convolution

```
result = zeros(U, V);
for i = (M+1):(U-M)
    for j = (N+1):(V-N)
        result(i,j) = sum(sum(kernel .* input((i-M):(i+M), (j-N):(j+N))));
    end
end
```

- result(i,j) is a weighted average of the neighborhood of input(i,j).
    - size of neighborhood defined by kernel.
    - weights for weighted average also defined by kernel.
- By using different kernels, lots of interesting operations can be defined.

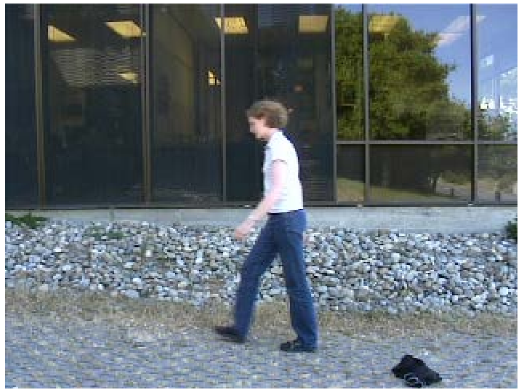
Nonlinear vs. Linear Filters

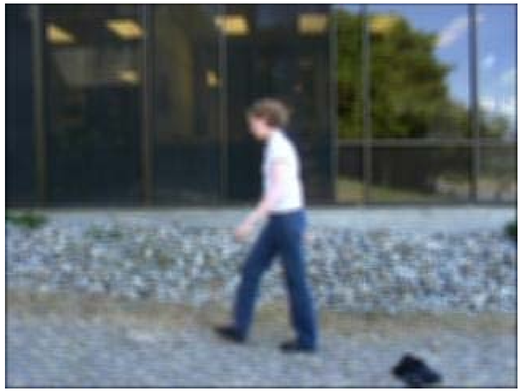
- Linear filters are convolutions with a kernel.
  - In Matlab, use imfilter, or filter2.
- A nonlinear filter is any image operation that is not a convolution.
  - The result value at a pixel is a function of the original values in a neighborhood of that pixel.
  - Each nonlinear filter must be implemented as a separate function.
- Examples of nonlinear filters:
  - Thresholding.
  - Non-maxima suppression (see lecture on edge detection).

Example of Convolution: Blurring

- Intuitively: replacing each pixel value with a weighted average of values in its neighborhood.
  - Simplest: N x N neighborhood, all weights equal.

```
original = double(imread('walkstraight/frame0062.tif'));
kernel = ones(5,5) / 25;
blurred = imfilter(original, kernel);
```


original


blurred

Blurring with a Gaussian

```
original = double(imread('walkstraight/frame0062.tif'));
kernel = fspecial('gaussian', 19, 3.0); % std = 3
blurred = imfilter(original, kernel);
```

- **`fspecial('gaussian', 19, 3.0)`** produces a kernel of size 19x19, whose values represent a symmetric 2-dimensional Gaussian with standard deviation 3.0.

A Gaussian Kernel

```
>> fspecial('gaussian', 7, 3.0)

ans =

    0.0113    0.0149    0.0176    0.0186    0.0176    0.0149    0.0113
    0.0149    0.0197    0.0233    0.0246    0.0233    0.0197    0.0149
    0.0176    0.0233    0.0275    0.0290    0.0275    0.0233    0.0176
    0.0186    0.0246    0.0290    0.0307    0.0290    0.0246    0.0186
    0.0176    0.0233    0.0275    0.0290    0.0275    0.0233    0.0176
    0.0149    0.0197    0.0233    0.0246    0.0233    0.0197    0.0149
    0.0113    0.0149    0.0176    0.0186    0.0176    0.0149    0.0113
```

- Here is a 7x7 Gaussian kernel with std = 3.0.
- Notice:
  - The highest value is at the center.
  - Values fall symmetrically as distance from center increases.

Blurring with a Gaussian

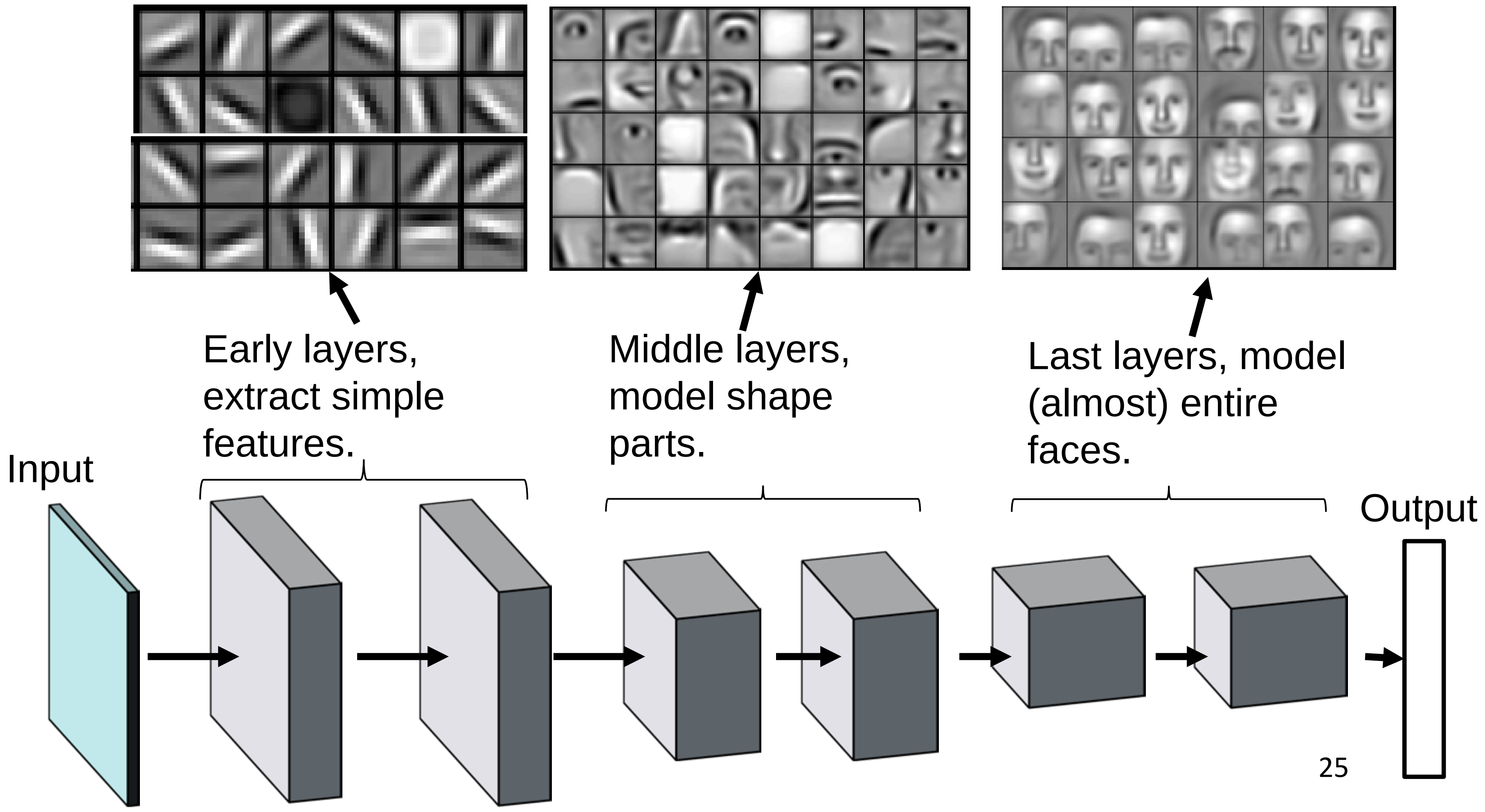
- Blurring with a Gaussian kernel gives more weight to kernel locations closer to the center.
- Blurring with a uniform kernel (like **`ones(5,5)/25`**) gives equal weight to all kernel locations.

Convolutions and Deep Learning

- This course is NOT about deep learning.
  - This slide and the next one may be the only references to deep learning this semester.
  - Nonetheless, deep learning methods are dominating these days in computer vision.
- Deep learning methods use convolutions as one of their fundamental operations.
- What is *learned* by such methods is, to a large extent, the kernels that should be used for various convolutions that are applied in parallel and in sequence.

CNN Visualization

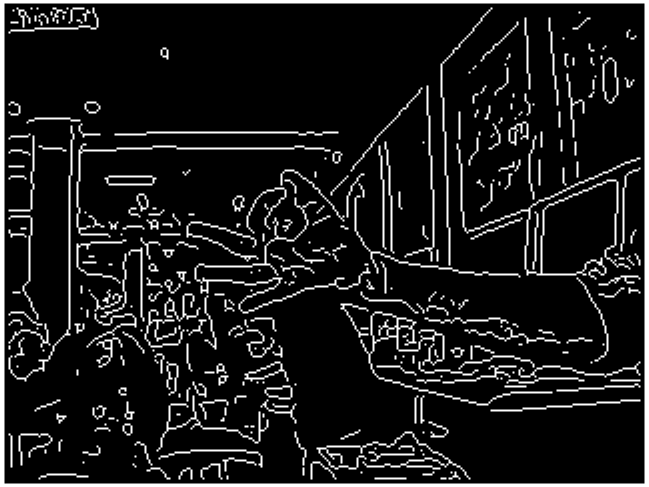
- Visualization of a deep neural network trained with face images.
  - Credit for feature visualizations: [Lee09] Honglak Lee, Roger Grosse, Rajesh Ranganath and Andrew Y. Ng., ICML 2009.

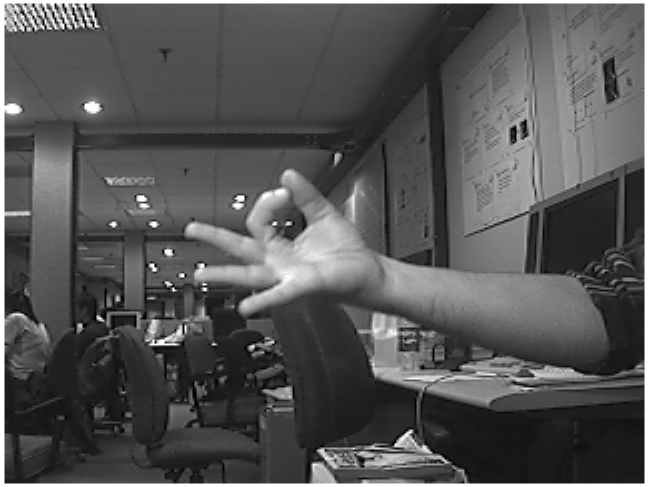


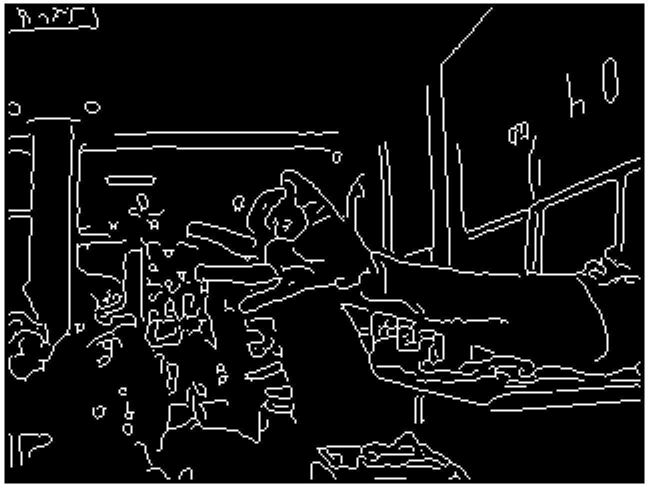
Early layers, extract simple features.
Middle layers, model shape parts.
Last layers, model (almost) entire faces.
Input
Output

What Is an Edge?

- An edge pixel is a pixel at a "boundary".

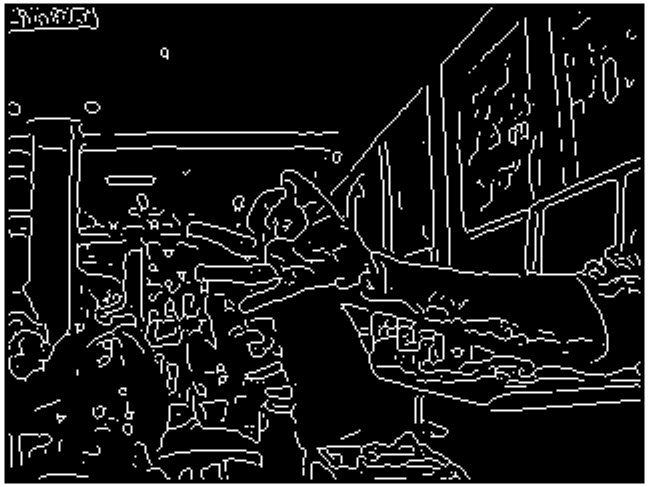

output1


input

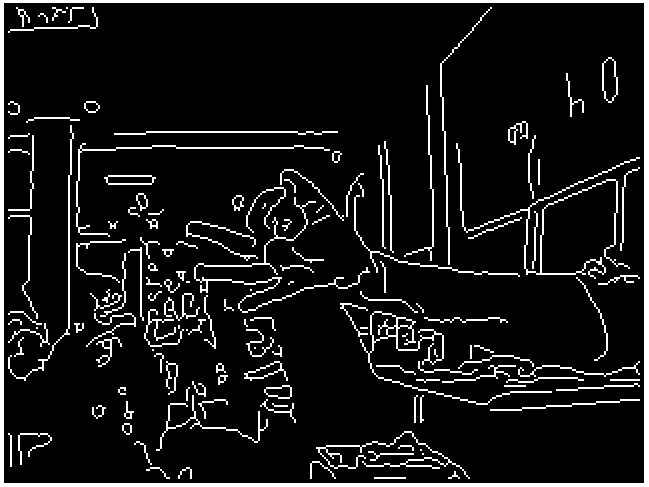

output2

What Is an Edge?

- There is no single definition for what is a “boundary”. Why?

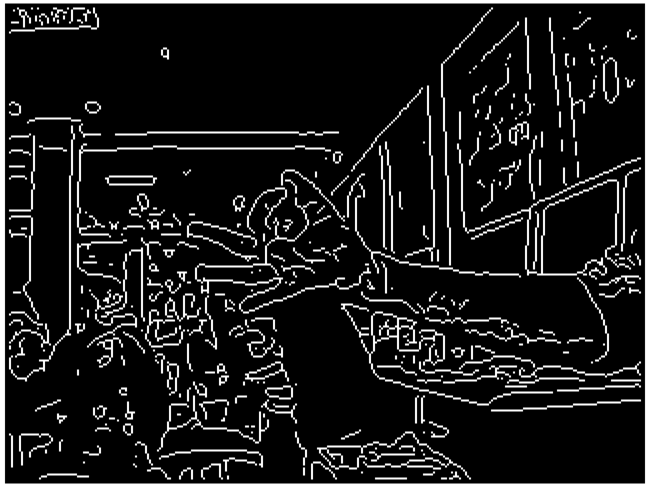

output1

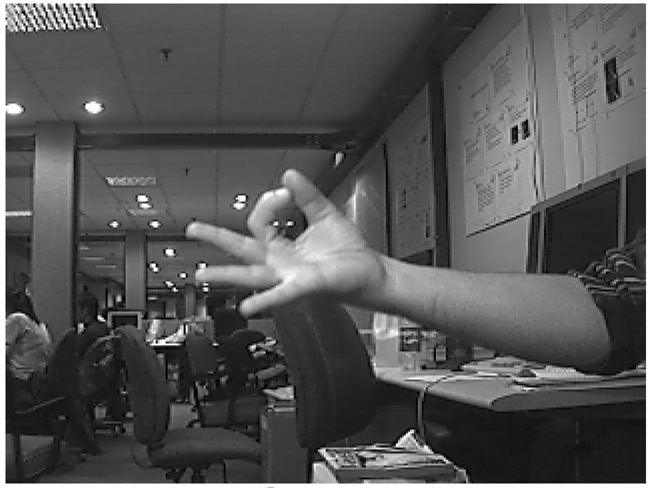
input


output2

What Is an Edge?

- There is no single definition for what is a “boundary”. Why?
  - Because breaking up an image into parts can be done in many ways.
  - For example: is “ceiling” one part, or is each tile a separate part?


output1


input

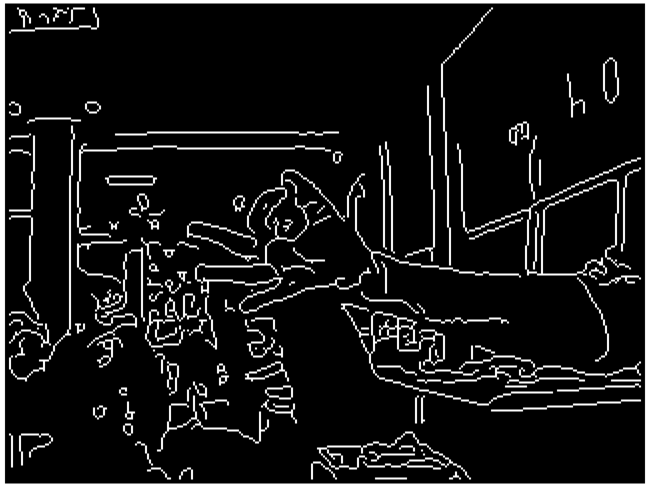

output2

Image Derivatives

- We have all learned how to calculate derivatives of functions from the real line to the real line.
  - Input: a real number
  - Output: a real number
- In taking image derivatives, we must have in mind two important differences from the "typical" derivative scenario:
  - Input: two numbers (y, x), not one number.
  - Input: discrete (integer pixel coordinates), not real numbers.
  - Output: Integer between 0 and 255.

Image Derivatives

- We have all learned how to calculate derivatives of functions from the real line to the real line.
  - Input: a real number
  - Output: a real number
- In taking image derivatives, we must have in mind two important differences from the "typical" derivative scenario:
  - Input: two numbers (y, x), not one number.
  - Input: discrete (integer pixel coordinates).
  - Output: Integer between 0 and 255.
- Why are we writing the input as (y, x) and not (x, y)?

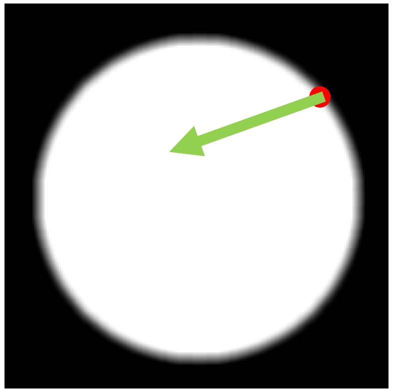
Image Derivatives

- We have all learned how to calculate derivatives of functions from the real line to the real line.
  - Input: a real number
  - Output: a real number
- In taking image derivatives, we must have in mind two important differences from the "typical" derivative scenario:
  - Input: two numbers (y, x), not one number.
  - Input: discrete (integer pixel coordinates).
  - Output: Integer between 0 and 255.
- Why are we writing the input as (y, x) and not (x, y)?
  - We will always use matrix notation: rows before columns.

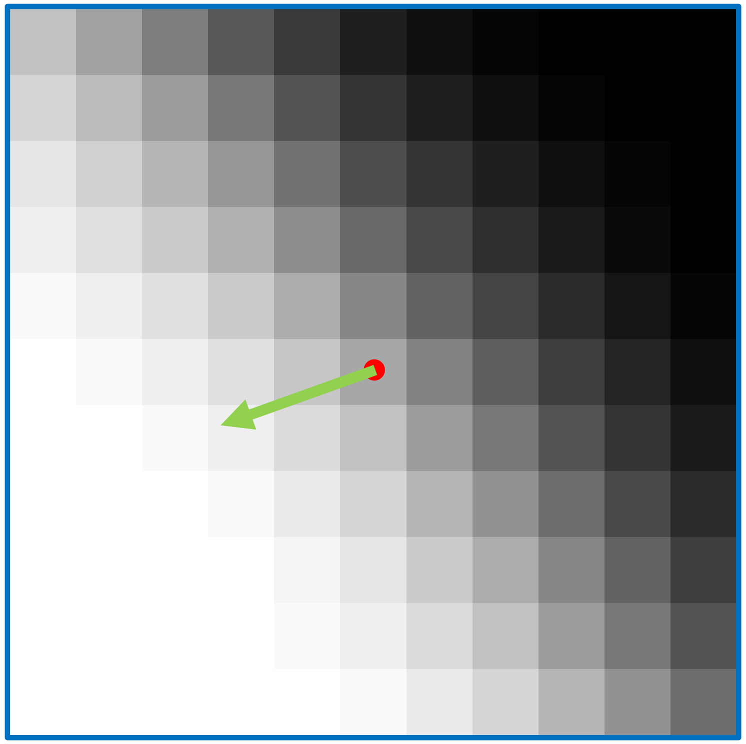
Directional Derivative

- Let $f(y, x)$ be a function mapping two *real* numbers to a real number.
- Let θ be a direction (specified as an angle from the x axis).
- Let (y_1, x_1) be a specific point on the plane.
- Define $g(x) \; = \; f(y_1 + x\sin(\theta), x_1 + \; x\cos(\theta))$.
- Then, $g(x)$ is a function from the real line to the real line.
- We define the **<u>directional derivative</u>** of f at (y_1, x_1) and direction θ to be $g'(0)$.

Directional Derivatives

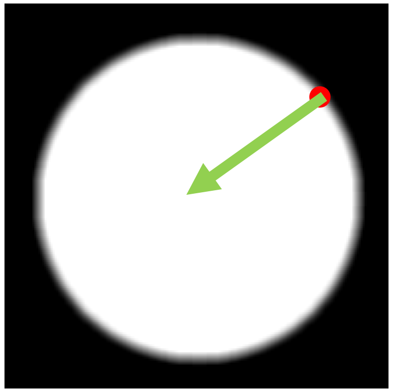


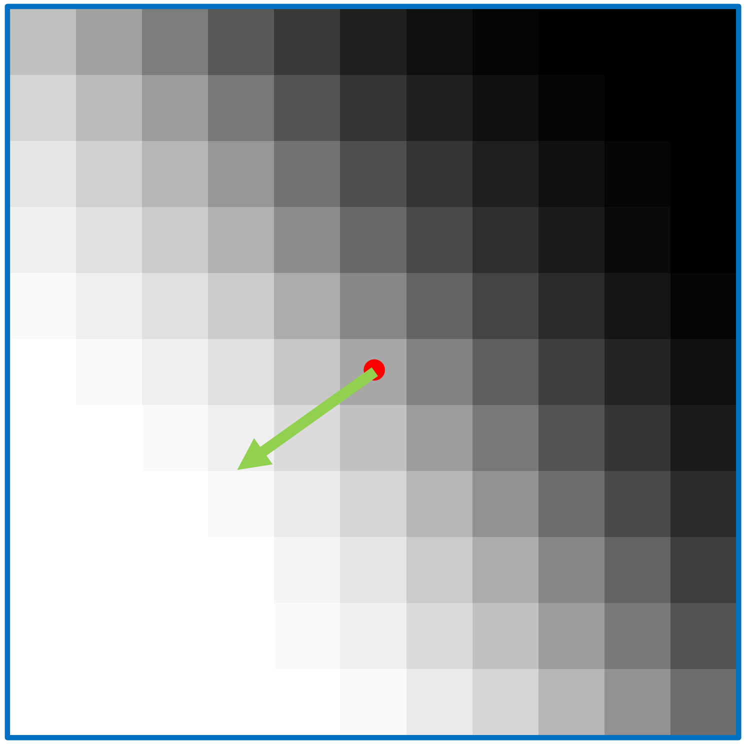
- Consider the circle shown on the top image.
- Pick a point (y_1, x_1) (shown in red).
- The bottom image shows a zoom of the region around that point.
- Pick a direction θ (shown with arrow).
- The **directional derivative** of f at (y_1, x_1) and direction θ measures how fast f increases along the chosen direction.
- Different directions give different results.



Directional Derivatives

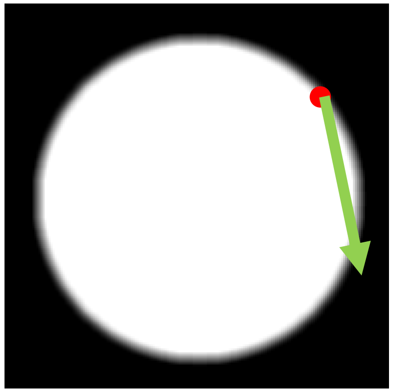
- How much does f change along the direction of the arrow?
- Quite a bit.
- **<u>IMPORTANT:</u>** we care about the difference between:
  - intensity values **towards** the direction that the arrow is pointing to.
  - Intensity values **<u>opposite</u>** of the direction that the arrow is pointing to.

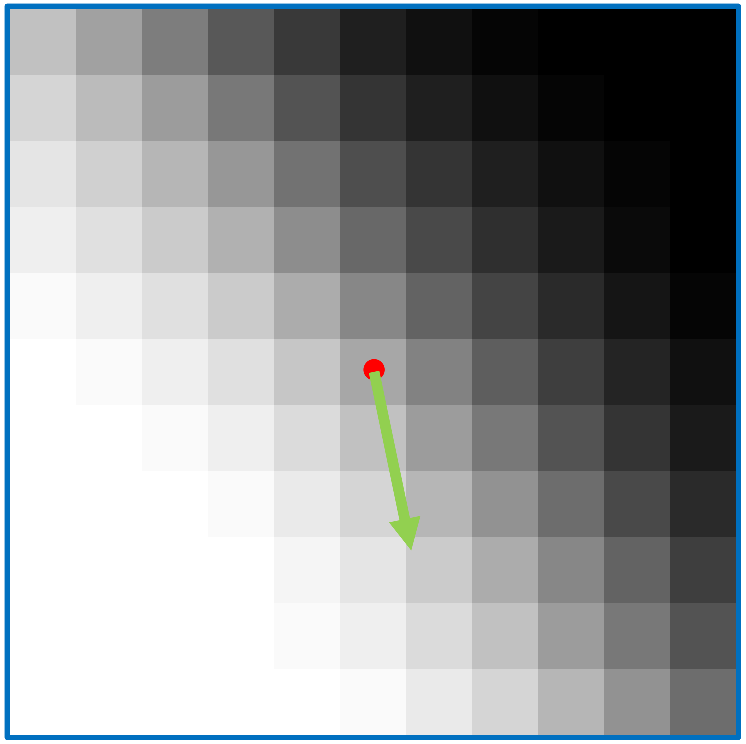




Directional Derivatives

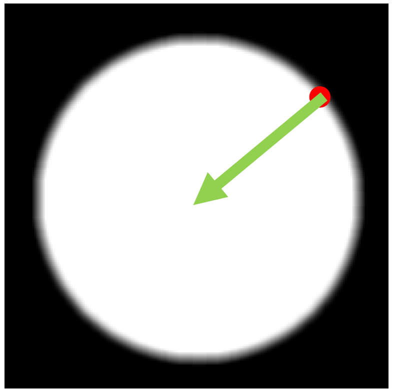
- How much does f change along the direction of the arrow?
- Noticeably less than in the previous case.

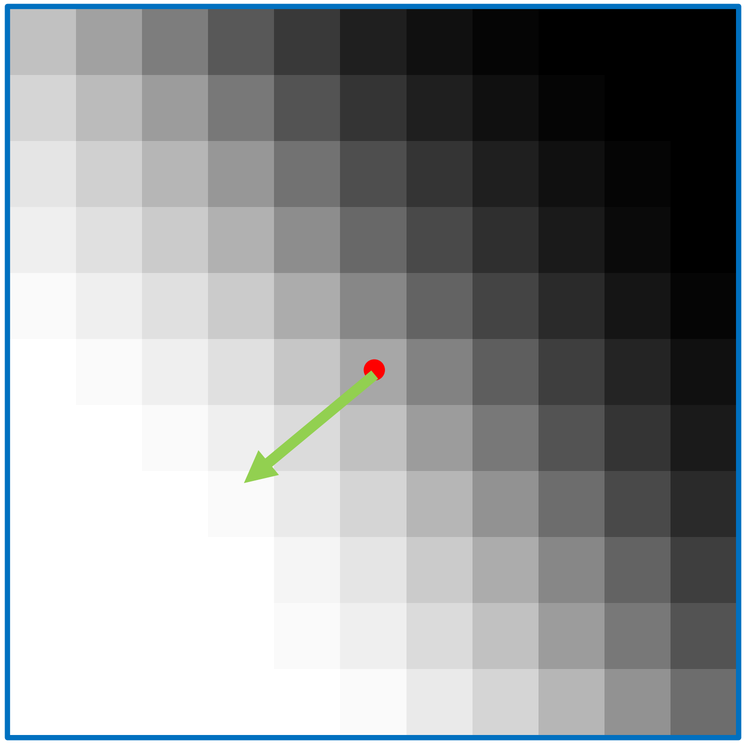




Directional Derivatives

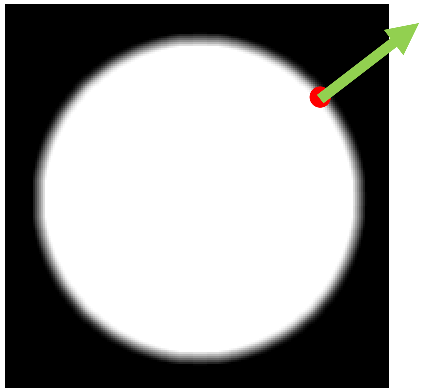
- Direction of fastest increase.

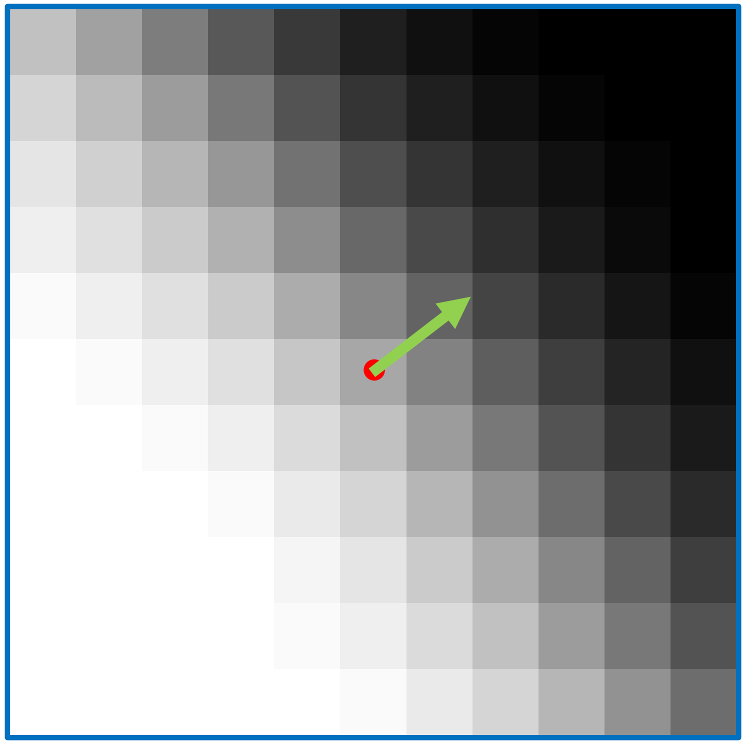




Directional Derivatives

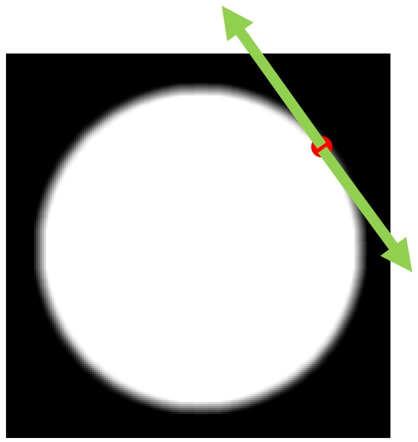
- Direction of fastest decrease.
- We will not prove it in this class, but mathematically the direction of farthest decrease is always 180 degrees opposite of the direction of fastest increase.
- Also, in the directions perpendicular to the direction of fastest increase/decrease, the directional derivative is always 0.

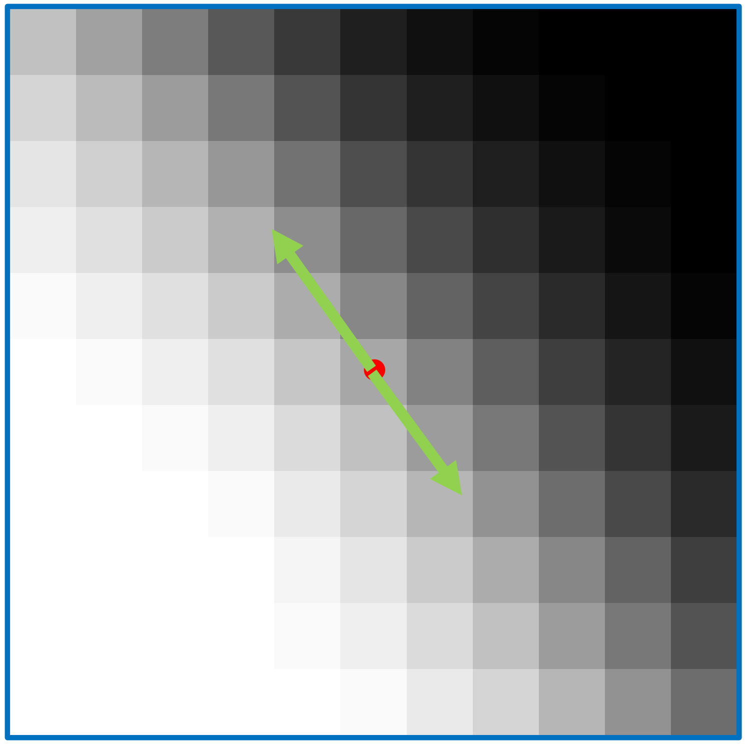




Directional Derivatives

- Directions where the directional derivative is 0.
  - Perpendicular to the directions of fastest increase/decrease.





Computing Directional Derivatives

- We have defined directional derivatives in the mathematical sense, for a function that maps a 2D **real** vector into a real number.
- However, an image is a function that maps a 2D **integer** vector to an integer.
- We still have not talked about how to compute directional derivatives on images.
- We will discuss this in the next slides.
  - First, we describe how to compute directional derivatives along the y axis and the x axis.
  - Then we generalize to any direction.

Directional Derivatives: Notation

- For the directional derivative of f along the y axis, we use notation $\frac{df}{dy}$.
- For the directional derivative of f along the x axis, we use notation $\frac{df}{dx}$.

Computing $\frac{df}{dx}$

- Mathematical definition of $\frac{df}{dx}$ for a function $f(y, x)$ whose inputs are continuous (i.e., the inputs are real numbers):

$$\frac{df(y,x)}{dx} = \lim_{\varepsilon \to 0} \frac{f(y, x+\varepsilon) - f(y, x-\varepsilon)}{2\varepsilon}$$

- Why can't we use this definition directly on an image?

Computing $\frac{df}{dx}$

- Mathematical definition of $\frac{df}{dx}$ for a function $f(y, x)$ whose inputs are continuous (i.e., the inputs are real numbers):

$$\frac{df(y,x)}{dx} = \lim_{\varepsilon \to 0} \frac{f(y, x+\varepsilon) - f(y, x-\varepsilon)}{2\varepsilon}$$

- Why can't we use this definition directly on an image?
  - On an image, the input is discrete: y and x are integers.
  - Thus, ε can be 0 or 1, but nothing in between.

Computing $\frac{df}{dx}$ via Convolution

- If $f(y, x)$ is an image, $\frac{df}{dx}$ is computed by convolving with the right kernel:

```
dx = [-0.5, 0, 0.5];
dxgray = imfilter(double(gray), dx); % gray: grayscale image
```

- Interpreting **`imfilter(gray, dx)`**:
  - For any pixel (y, x), $\frac{df(y,x)}{dx} = \frac{f(y,x+1) - f(y,x-1)}{2}$
  - Compare with continuous case: $\frac{df(y,x)}{dx} = \lim_{\varepsilon \to 0} \frac{f(y,x+\varepsilon) - f(y,x-\varepsilon)}{2\varepsilon}$
  - The discrete case is obtained from the continuous case, by using $\varepsilon = 1$.

Vertical and Horizontal Edges

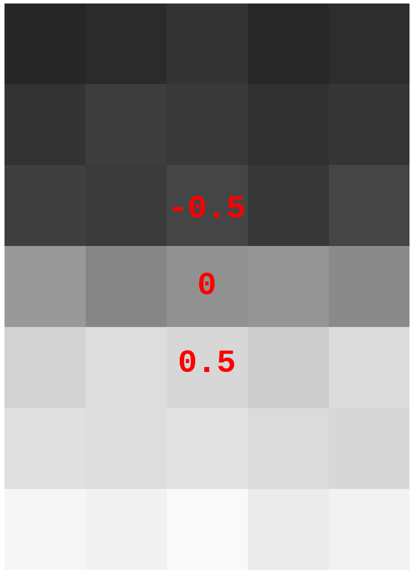
- Horizontal edges correspond to points in f with high **absolute values** of $\frac{df}{dy}$.
- Vertical edges correspond to points in f with high **absolute values** of $\frac{df}{dx}$.
- Why do we care about absolute values when we look for edges?

Vertical and Horizontal Edges

- Consider the image on the right.
- In case you are curious, here are the intensity values for that image:

```
a = [38  43  52  39  45
     51  61  57  49  53
     63  58  69  55  69
     153 133 145 149 138
     210 221 215 205 220
     224 222 226 219 215
     246 241 249 235 242];
```

- Suppose we convolve with dy.
  - Pixel (4,3) should be a horizontal edge.
  - For that pixel, does convolution with dy give a low value or a high value?



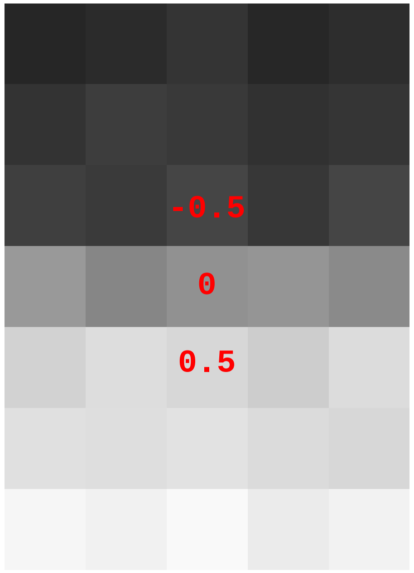
```
dy = [-0.5,
         0,
       0.5];
```

Vertical and Horizontal Edges

- Consider the image on the right.
- In case you are curious, here are the intensity values for that image:

```
a = [38  43  52  39  45
     51  61  57  49  53
     63  58  69  55  69
     153 133 145 149 138
     210 221 215 205 220
     224 222 226 219 215
     246 241 249 235 242];
```

- Suppose we convolve with dy.
  - Pixel (4,3) should be a horizontal edge.
  - For that pixel, we get a high value: $0.5 * 215 + (-0.5) * 69 = 73.$

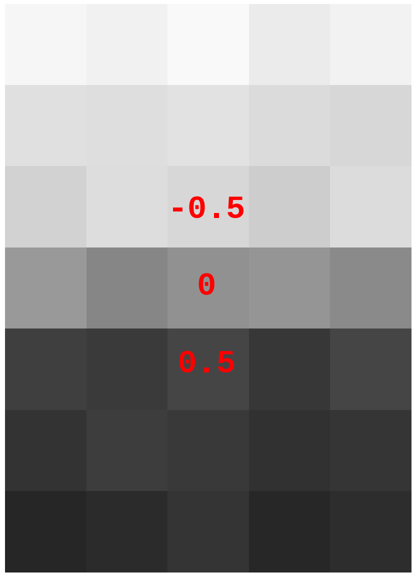


```
dy = [-0.5,
         0,
       0.5];
```

Vertical and Horizontal Edges

- Now consider this new image on the right.
- In case you are curious, here are the intensity values for that image:

```
b = [246 241 249 235 242
     224 222 226 219 215
     210 221 215 205 220
     153 133 145 149 138
     63  58  69  55  69
     51  61  57  49  53
     38  43  52  39  45];
```



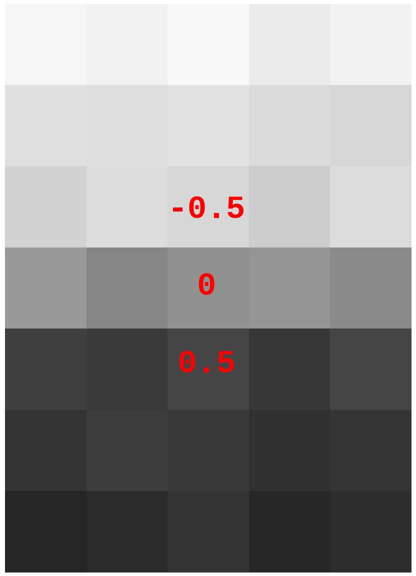
```
dy = [-0.5,
         0,
       0.5];
```

- Suppose we convolve with dy.
  - Pixel (4,3) should still be a horizontal edge.
  - For that pixel, does convolution with dy give a low value or a high value?

Vertical and Horizontal Edges

- Now consider this new image on the right.
- In case you are curious, here are the intensity values for that image:

```
b = [246 241 249 235 242
     224 222 226 219 215
     210 221 215 205 220
     153 133 145 149 138
     63  58  69  55  69
     51  61  57  49  53
     38  43  52  39  45];
```

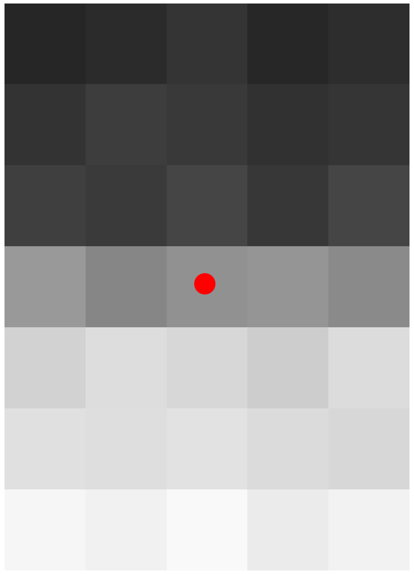


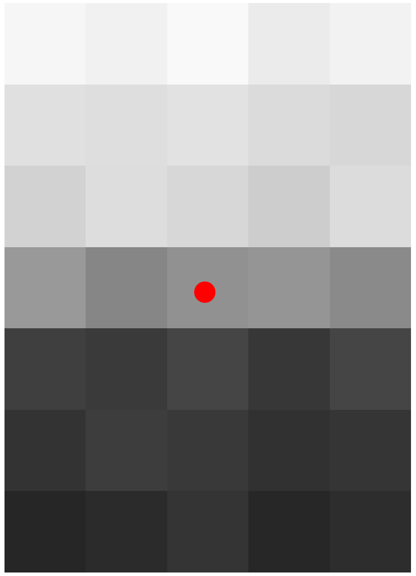
```
dy = [-0.5,
         0,
       0.5];
```

- Suppose we convolve with dy.
  - Pixel (4,3) should still be a horizontal edge.
  - For that pixel, we get a low value: $0.5 \ * 69 + \ (-0.5) * 215 = -73.$

Vertical and Horizontal Edges

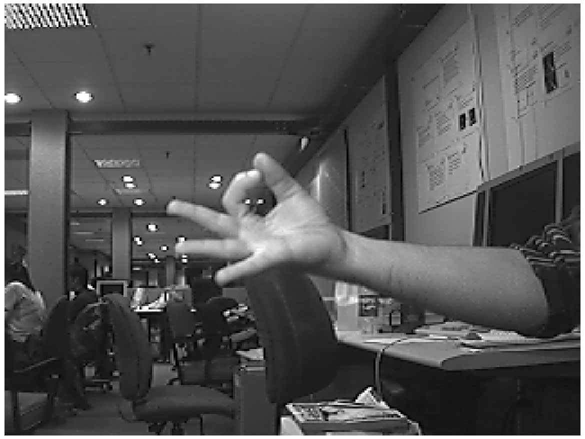
- These two images (shown in the previous slides) are flipped versions of each other.
- In both cases, pixel (4,3), and the other pixels on the fourth row, should be edge pixels.
- If we convolve with dy:
  - in the top image we get a “high” value of 73.
  - in the bottom image we get a “low” value of -73.
- In both cases, the absolute value is the same.
  - The absolute value tells us how big the rate of change is, which is what we care about when we look for edges.
  - The sign simply tells us the direction of change (which side is brighter).

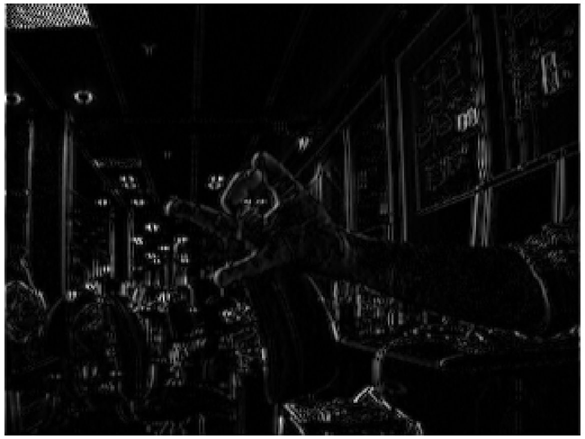


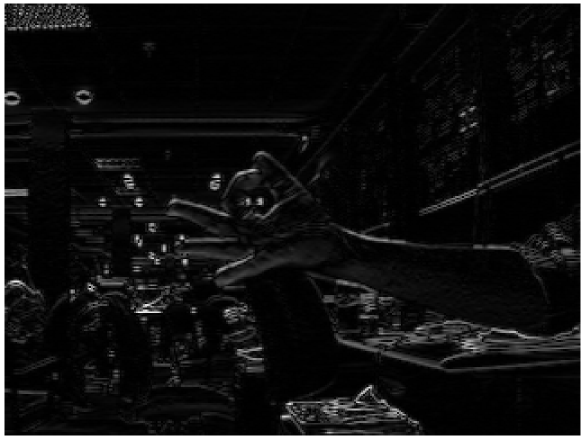


Example: Vertical/Horizontal Edges

```
gray = read_gray('data/hand20.bmp');
dx = [-0.5, 0, 0.5];
dy = dx'; % The dy kernel is just the transpose of dx
dxgray = abs(imfilter(gray, dx));
dygray = abs(imfilter(gray, dy));
```

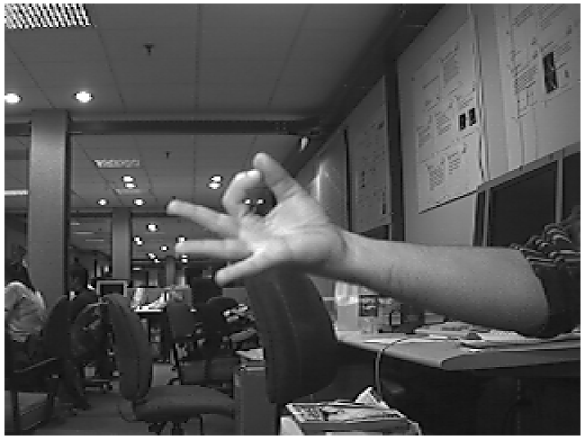

gray

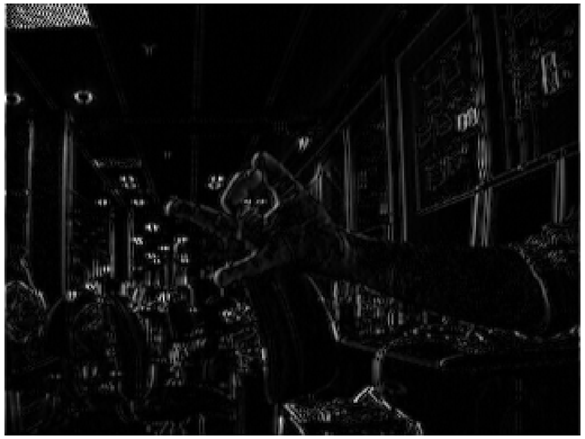

dxgray
(vertical edge scores)

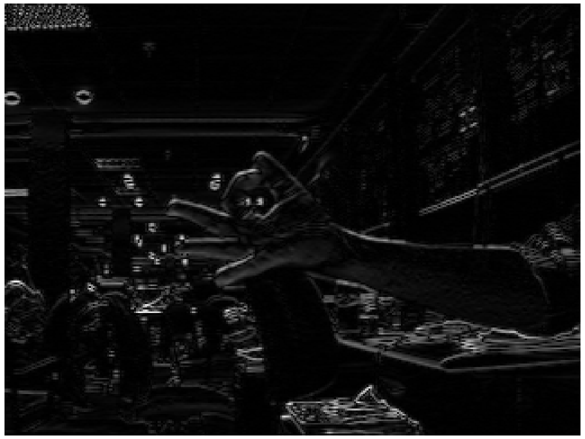

dygray
(horizontal edge scores)

Example: Vertical/Horizontal Edges

- Note: we call dxgray and dygray "edge scores", as opposed to "edge images".
  - High absolute values of dx and dy directional derivatives can be seen as preliminary "edge scores", that are associated with vertical and horizontal edges.

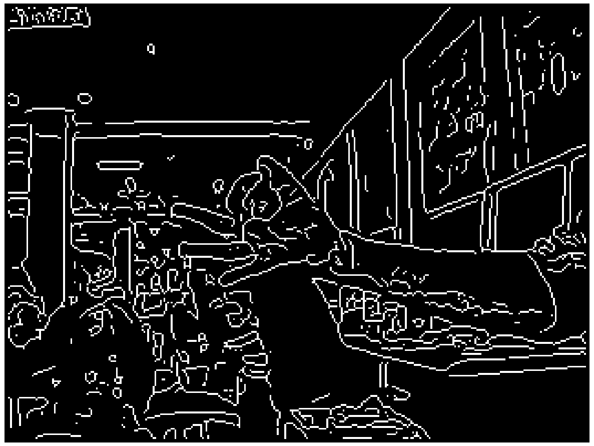

`gray`

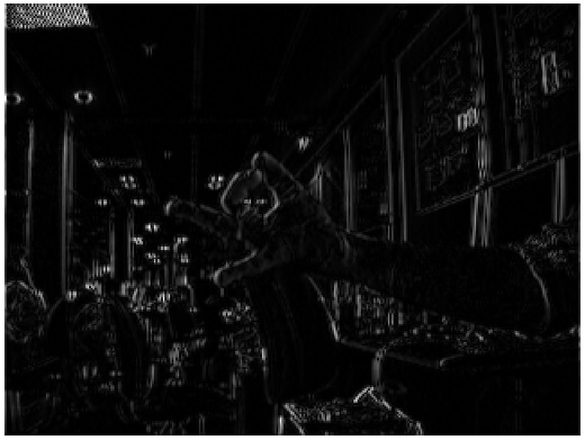

`dxgray`
(vertical edge scores)

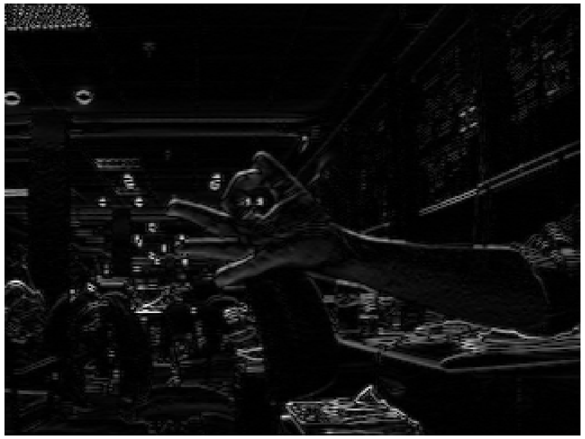

`dygray`
(horizontal edge scores)

Example: Vertical/Horizontal Edges

- There are additional steps that we will go over, in order to produce what we would call "edge images". In an edge image:
  - Edges should be localized to thin lines to the extent possible.
  - Values should be binary. An edge image indicates a hard decision on whether a pixel is an edge pixel or not. No intermediate values appear.


example edge image


`dxgray`
(vertical edge scores)


`dygray`
(horizontal edge scores)

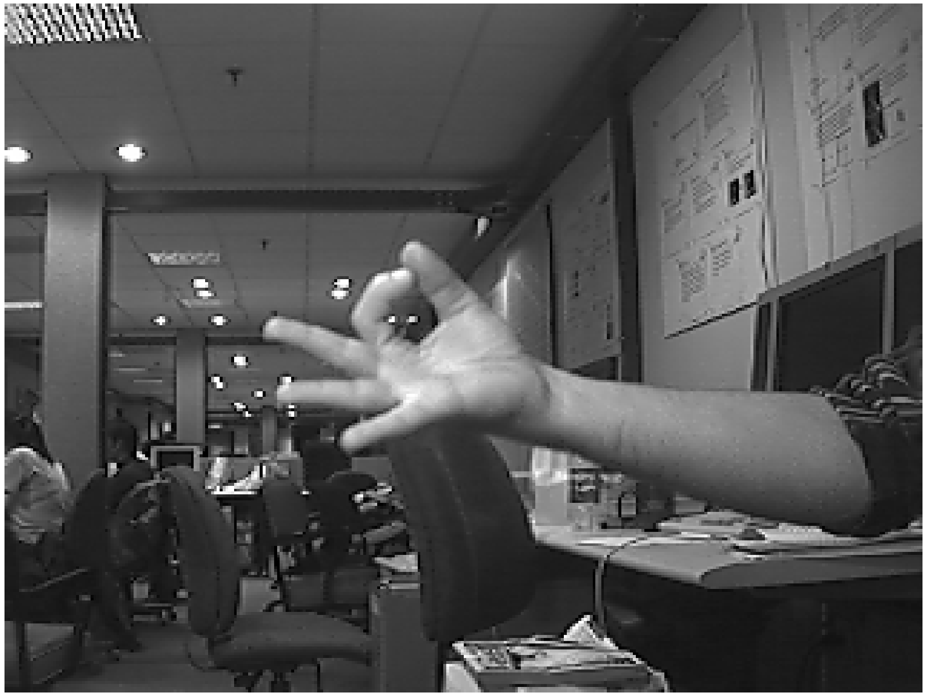
Blurring and Derivatives

- To suppress edges corresponding to small-scale objects/textures, we should first blur.

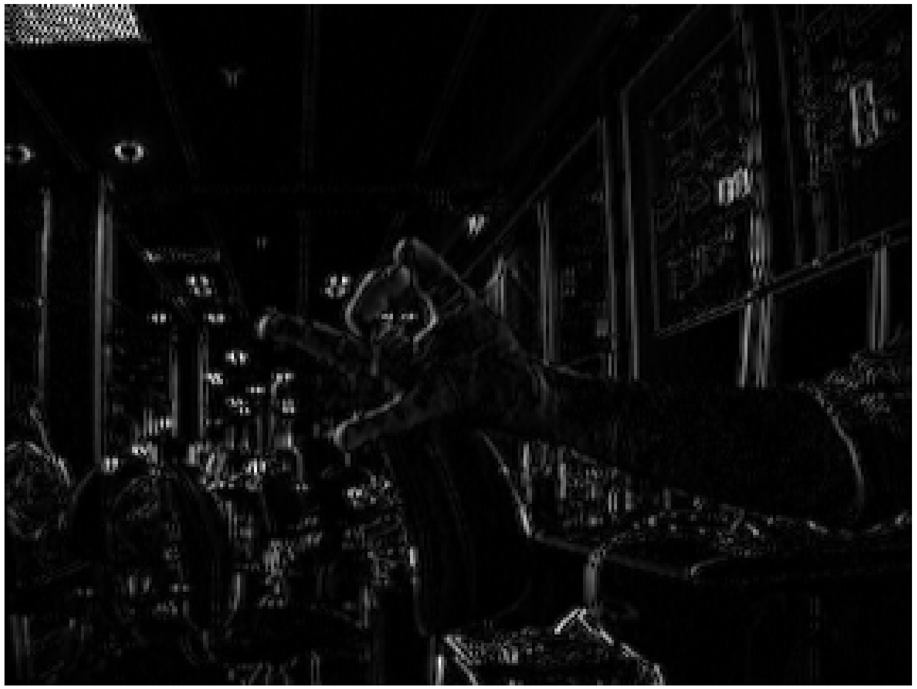
```
% generate two blurred versions of the image, see how it
% looks when we apply dx to those blurred versions.
filename = 'data/hand20.bmp';
gray = read_gray(filename);
dx = [-0.5, 0, 0.5];
dy = dx'; % The dy kernel is just the transpose of dx.

blur_window1 = fspecial('gaussian', 19, 3.0); % std = 3
blur_window2 = fspecial('gaussian', 37, 6.0); % std = 6
blurred_gray1 = imfilter(gray, blur_window1, 'symmetric');
blurred_gray2 = imfilter(gray, blur_window2 , 'symmetric');
dxgray = abs(imfilter(gray, dx, 'symmetric'));
dxb1gray = abs(imfilter(blurred_gray1, dx , 'symmetric'));
dxb2gray = abs(imfilter(blurred_gray2, dx , 'symmetric'));
```

Blurring and Derivatives: Results

The texture of the lights on the ceiling gives rise to strong vertical edge scores if we don't blur.



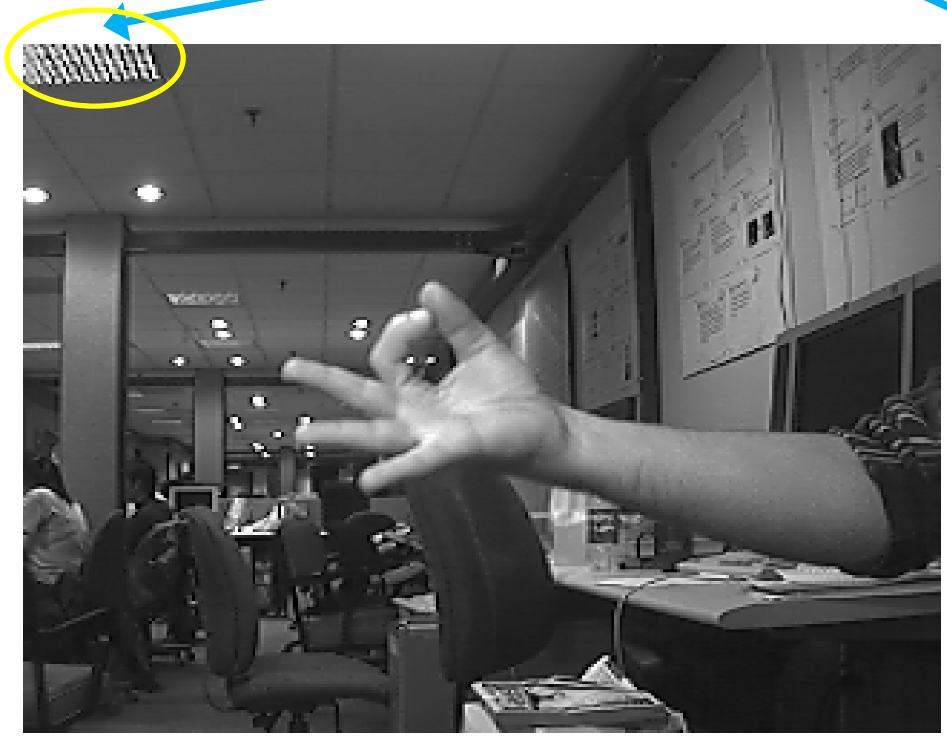
`gray`



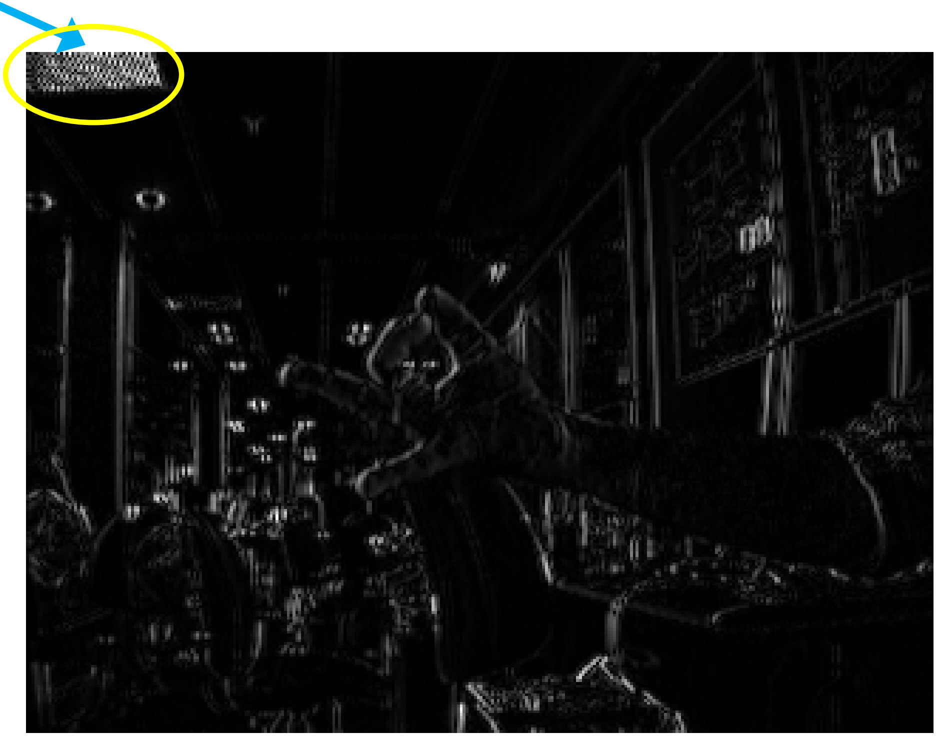
`dxgray` No blurring

Blurring and Derivatives: Results

The texture of the lights on the ceiling gives rise to strong vertical edge scores if we don't blur.



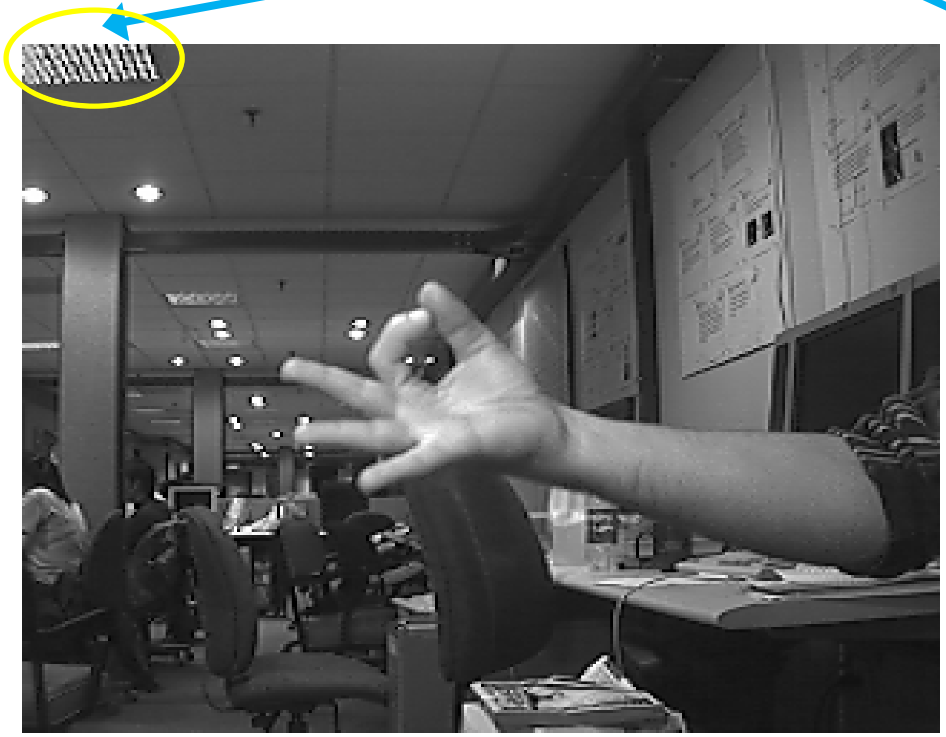
gray



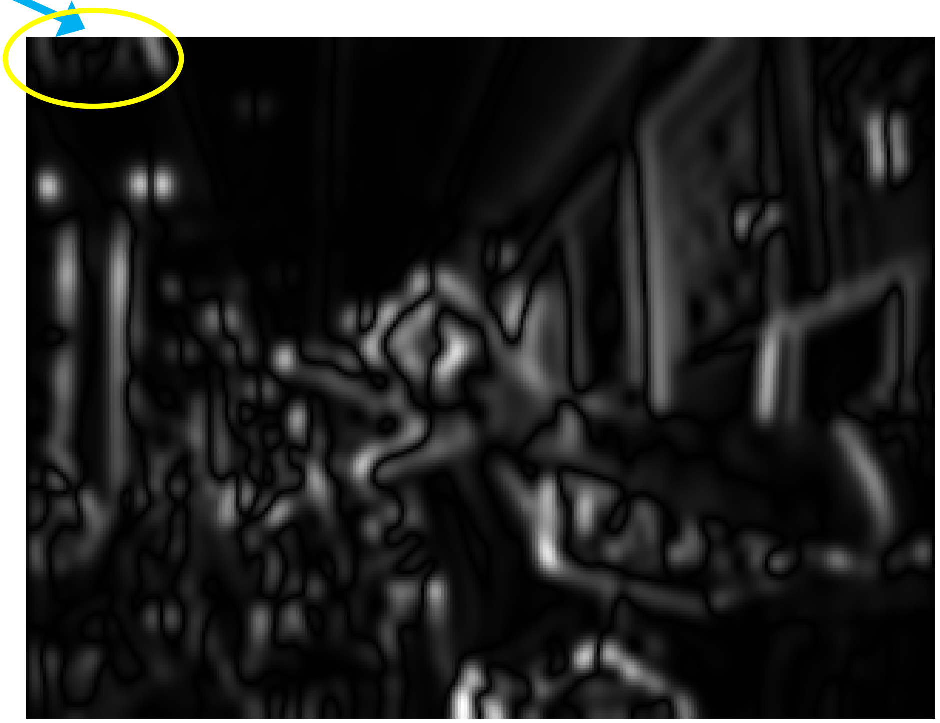
dxgray No blurring

Blurring and Derivatives: Results

If we blur sufficiently, those edges disappear, because we blur away the intensity differences that cause those high edge scores.



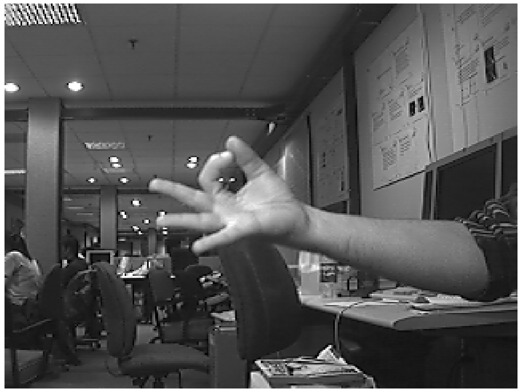
`gray`



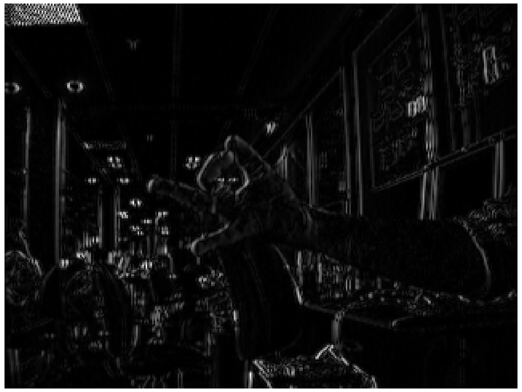
`dxb1gray` Blurring, std = 3

Blurring and Derivatives: Results

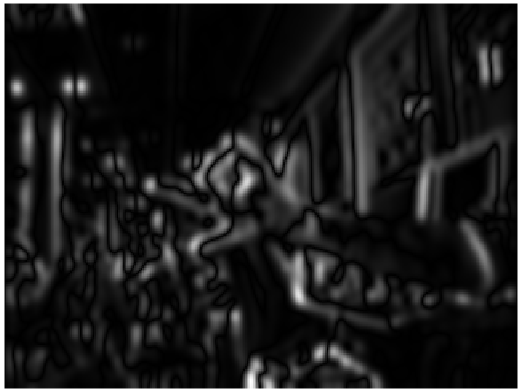
`gray`



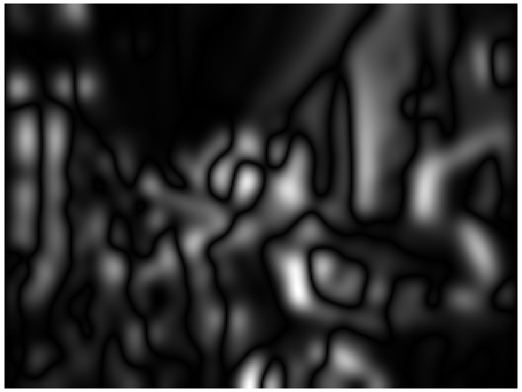
`dxgray`
No blurring



`dxb1gray`
Blurring, std = 3



`dxb2gray`
Blurring, std = 6



- If we blur, smaller details get suppressed, but edges get too thick.
  - Will be remedied in a few slides, with *non-maxima suppression.*

Combining Blurring and Derivatives

- Blurring and filtering can be done with two convolutions, as seen in the previous slides:

```
blur_kernel = fspecial('gaussian', 5, 6.0); % std = 6
blurred_gray = imfilter(gray, blur_kernel);
dx_kernel = [-0.5, 0, 0.5];
dxgray = imfilter(blurred_gray, dx_kernel);
```

- However, oftentimes we combine blurring and filtering in a single convolution, by choosing an appropriate kernel.

Combining Blurring and Derivatives

- Here is an example of how to produce a single kernel that does both blurring and convolution:

```
blur_kernel = fspecial('Gaussian', 5, 0.7); % std = 0.7
dy_kernel = [-0.5, 0, 0.5]';

% We produce a combined kernel by convolving one kernel with
% the other kernel.The Matlab flip function reverses the order
% of the rows. We use flip to make sure that top values are
% negative and bottom values are positive.
combined_blur_dy = flip(imfilter(blur_kernel, dy_kernel))

   -0.0010   -0.0211   -0.0585   -0.0211   -0.0010
   -0.0027   -0.0575   -0.1597   -0.0575   -0.0027
         0         0         0         0         0
    0.0027    0.0575    0.1597    0.0575    0.0027
    0.0010    0.0211    0.0585    0.0211    0.0010
```

Combining Blurring and Derivatives

`combined_blur_dy`:

```
   -0.0010   -0.0211   -0.0585   -0.0211   -0.0010
   -0.0027   -0.0575   -0.1597   -0.0575   -0.0027
         0         0         0         0         0
    0.0027    0.0575    0.1597    0.0575    0.0027
    0.0010    0.0211    0.0585    0.0211    0.0010
```

- We can convolve with **`combined_blur_dy`** to compute the dy derivative of a blurred version of the image.

```
dygray = imfilter(gray, combined_blur_dy);
```

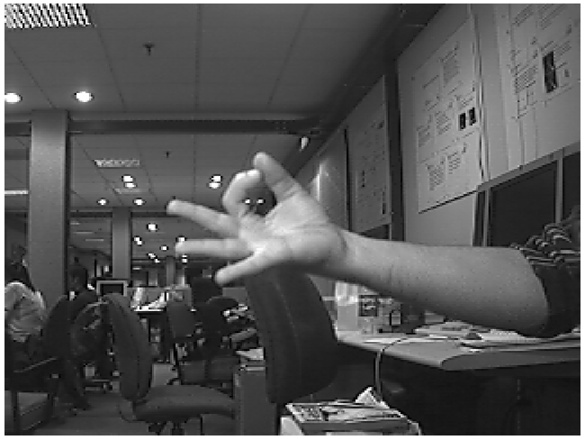
Simple Blur/Derivative Combinations

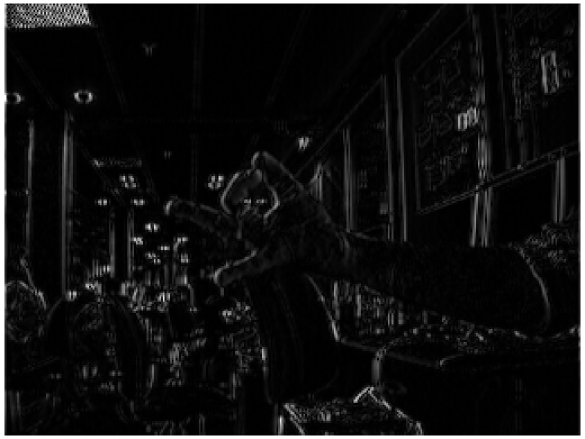
```
combined_blur_dy =[
 -1  -1  -1  -1  -1
 -1  -1  -1  -1  -1
 -1  -1  -1  -1  -1
  0   0   0   0   0
  1   1   1   1   1
  1   1   1   1   1
  1   1   1   1   1];
```

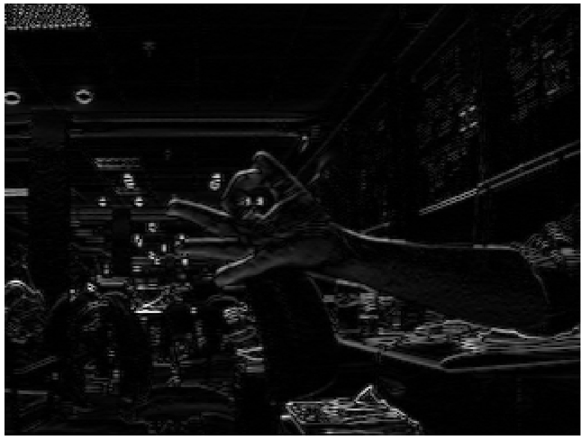
- We take the sum of a region under the center, and we subtract from it the sum of a region over the center.
- By summing regions, we effectively do blurring.
- “High” absolute values correspond to pixels where there is “high” difference between the intensities at the bottom region and the intensities at the top region.

Example

```
gray = read_gray('data/hand20.bmp');
dx = [-1 0 1;
      -2 0 2;
      -1 0 1]; % This dx kernel is another example of a blur/derivative kernel

dy = dx’; % dy is the transpose of dx
dxgray = abs(imfilter(gray, dx, 'symmetric', 'same'));
dygray = abs(imfilter(gray, dy, 'symmetric', 'same'));
```


gray


dxgray
(vertical edge scores)


dygray
(horizontal edge scores)

Edge Scores at Other Orientations

- We first need to define what we mean when we say that "an edge has orientation θ".
- In this class:
  - We will explicitly specify if we are using radians or degrees as units.
  - Orientation 0 is the orientation of the x axis.
  - Orientations increase clockwise.
    - In trigonometry textbooks, orientations typically increase counterclockwise.
    - However, in this class we follow the convention that the y axis increases downwards, and thus orientations will increase clockwise.
  - When we measure edge orientations, an orientation of θ degrees is the same as an orientation of (θ+180) degrees.

Edge Scores at Other Orientations

- Edge scores at orientation θ can be computed by taking absolute values of directional derivatives at orientation **perpendicular to θ**.
  - So, vertical edge scores (edge orientation = 90 degrees) are absolute values of the directional derivative at 0 degrees (along the x axis).
  - Horizontal edge scores (edge orientation = 0 degrees) are absolute values of the directional derivative at 90 degrees (along the y axis).

Edge Scores at Other Orientations

- Computing edge scores at an arbitrary orientations θ:
  - Use **imrotate** to rotate the image or the **dy** kernel by the right value.
    - Rotating the kernel is typically much more efficient.
  - **imrotate(my_image, degrees)** rotates **<u>clockwise</u>** the image passed as first argument by the degrees specified in its second argument.
  - To compute edge scores at an orientation of θ degrees, you must use a kernel produced by **imrotate(dy_kernel, -θ)**.

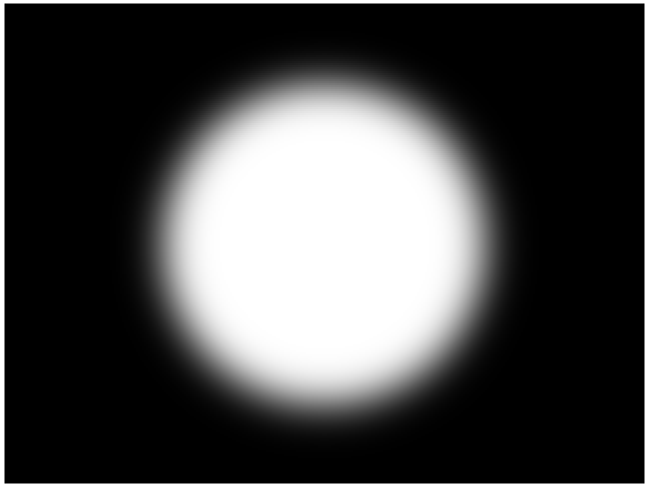
Edge Scores at Other Orientations

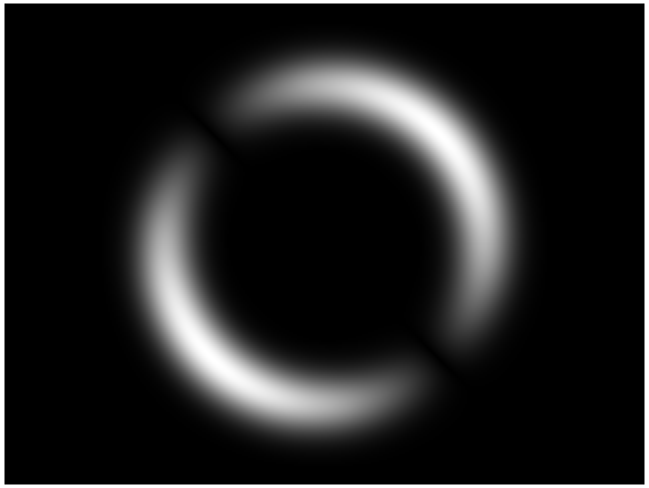
- Here is an example where we compute edge scores for orientations of 45 degrees and 135 degrees.

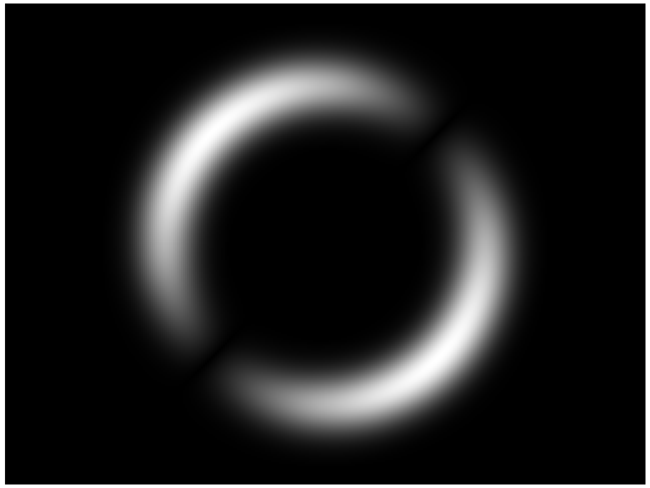
```
% Computing edge scores at 45 or 135 degrees:

fcircle = read_gray('data/blurred_fcircle.bmp');
dx = [-1 0 1; -2 0 2; -1 0 1] / 8;
dy = dx';
rot45 = imrotate(dy, -45, 'bilinear', 'loose');
rot135 = imrotate(dy, -135, 'bilinear', 'loose');

edges45 = abs(imfilter(fcircle, rot45, 'symmetric'));
edges135 = abs(imfilter(fcircle, rot135, 'symmetric'));
```

Edge Scores at Degrees 45/135

fcircle





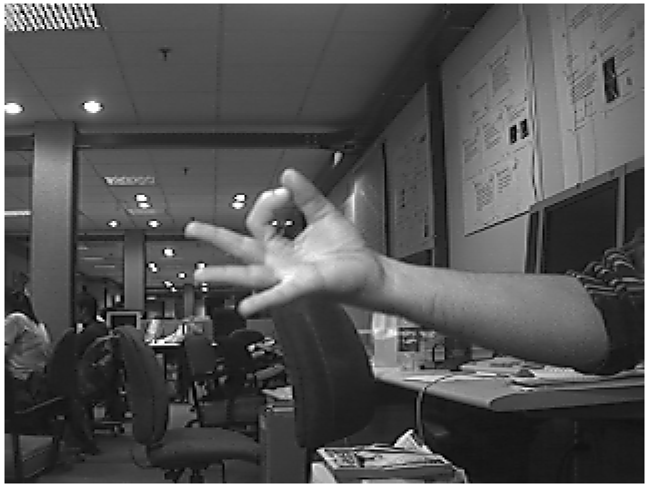


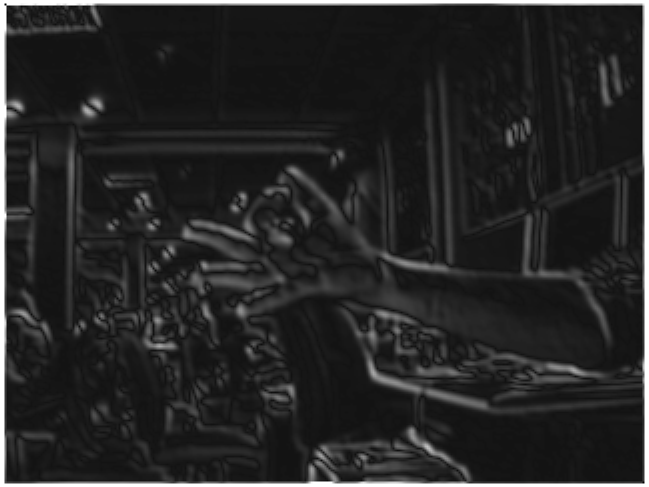
edges45

edges135

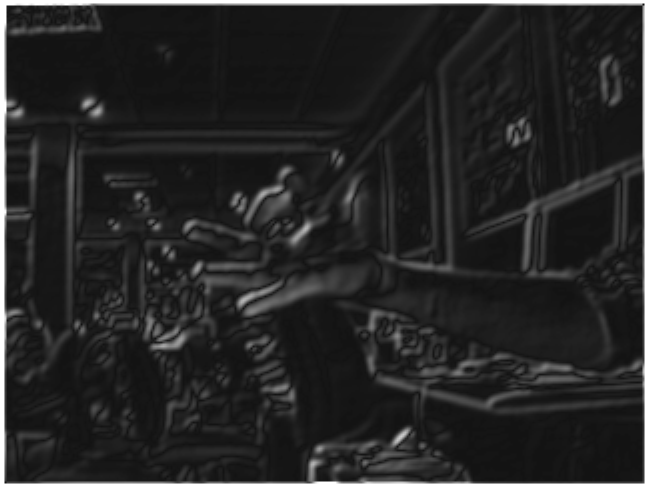
More Edges at 45/135 Degrees

original image





edge orientation: 45 degrees



edge orientation: 135 degrees

Next Steps for Edge Detection

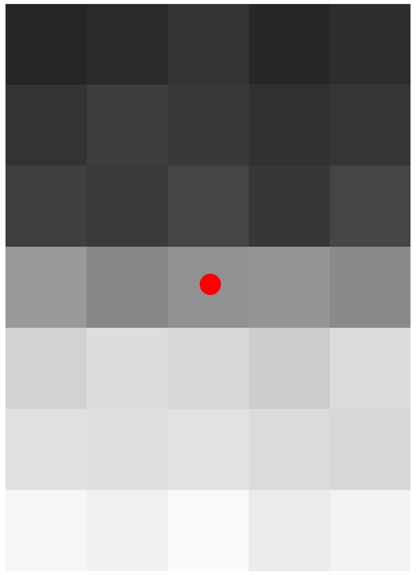
- What we have done so far:
  - We computed directional derivatives in the directions of the two axes.
  - We showed that absolute values of those directional derivatives can be seen as “edge scores” for vertical and horizontal edges.
  - For edge scores at other orientations we convolve with a rotated dy kernel.
- However, we need additional steps to complete edge detection.
  - Edges should be thin lines localized at actual boundaries.
  - Edge images are binary images. A pixel is either an edge pixel or not.
- Next:
  - We define the concept of the **gradient**.
  - The gradient of a pixel defines the **strength** and **orientation** of the edge at that pixel.
  - We will see how to produce thin edges, and how to make a hard final decision on whether a pixel should be considered an “edge pixel” or not.

Gradients

- Let $f(y, x)$ be an image, i.e., a function that maps 2D integers to integers.
- We define the gradient $\nabla f(y, x)$ to be the vector $\left(\frac{df}{dy}, \frac{df}{dx}\right)$.
- Thus, the gradient at position (y, x) is simply the vector of the two directional derivatives corresponding to the two axes.
- From calculus, we know that:
  - The norm of the gradient vector measures how fast values change at position (y, x) .
  - The gradient vector points to the direction of fastest increase at position (y, x).

Gradient Example

```
a = [38  43  52  39  45
     51  61  57  49  53
     63  58  69  55  69
     153 133 145 149 138
     210 221 215 205 220
     224 222 226 219 215
     246 241 249 235 242];
```



- What is the gradient vector at pixel (4,3)?

$$\frac{df}{dy} = 0.5 \ * \ 215 + \ (-0.5) * 69 = 73$$

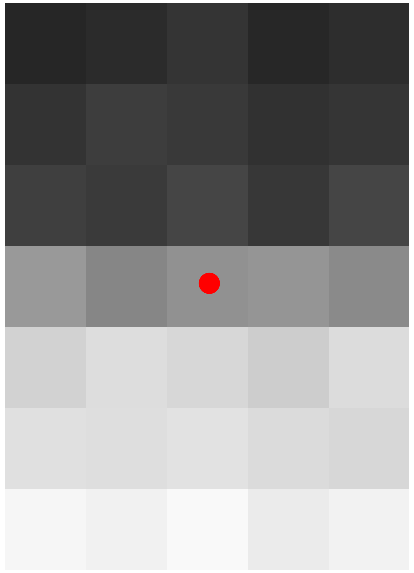
$$\frac{df}{dx} = 0.5 \ * 149 + \ (-0.5) * 133 = 8$$

```
dy = [-0.5,
         0,
       0.5];
dx = [-0.5, 0, 0.5];
```

- Thus, the gradient at $(4,3)$ is $(73, 8)$.

Gradient Norm Example

```
a = [38  43  52  39  45
     51  61  57  49  53
     63  58  69  55  69
     153 133 145 149 138
     210 221 215 205 220
     224 222 226 219 215
     246 241 249 235 242];
```

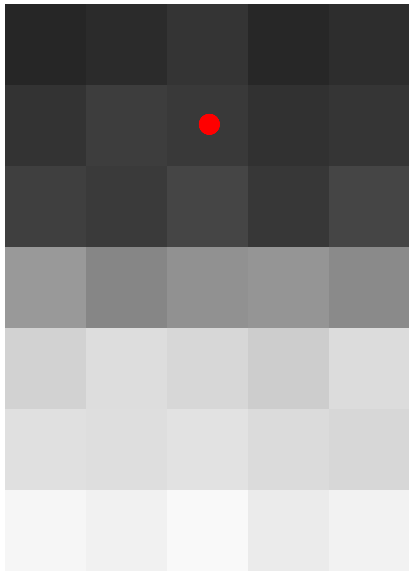


- We saw that the gradient at $(4,3)$ is $(73, 8)$.
- What is the **norm** of the gradient?
- The norm of a vector v is indicated as $||v||$, and it is the square root of the sum of the squares of the values in the vector:

$$||(73,8)|| = \sqrt{73^2 + 8^2} = 73.4$$

Gradient Example

```
a = [38  43  52  39  45
     51  61  57  49  53
     63  58  69  55  69
     153 133 145 149 138
     210 221 215 205 220
     224 222 226 219 215
     246 241 249 235 242];
```



- What is the gradient vector at pixel (2,3)?

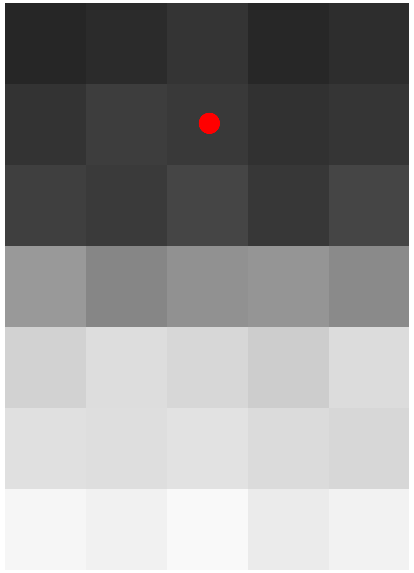
$$\frac{df}{dy} = 0.5 \ * 69 + \ (-0.5) * 52 = 8.5$$

$$\frac{df}{dx} = 0.5 \ * 49 + \ (-0.5) * 61 = -6.0$$

- Thus, the gradient at $(2,3)$ is $(8.5, -6.0)$.

Gradient Norm Example

```
a = [38  43  52  39  45
     51  61  57  49  53
     63  58  69  55  69
     153 133 145 149 138
     210 221 215 205 220
     224 222 226 219 215
     246 241 249 235 242];
```

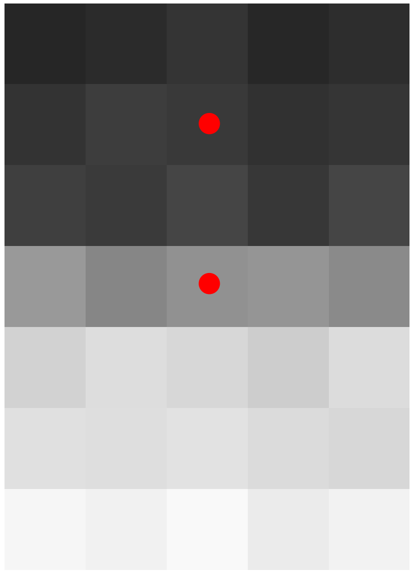


- We saw that the gradient at $(2,3)$ is $(8.5, -6.0)$.
- What is the **norm** of the gradient?

$$||(8.5{,}6.0)|| = \sqrt{8.5^2 + (-6.0)^2} = 10.4$$

Gradient Norms

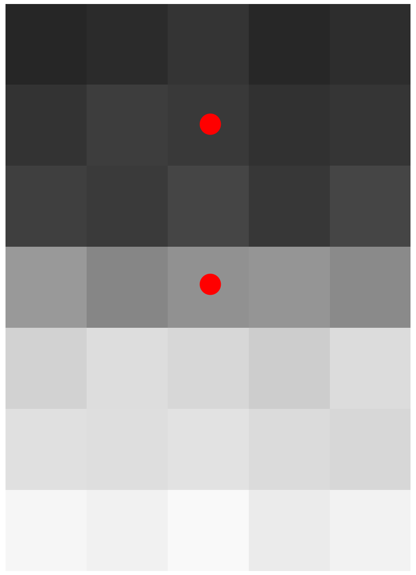
```
a = [38  43  52  39  45
     51  61  57  49  53
     63  58  69  55  69
     153 133 145 149 138
     210 221 215 205 220
     224 222 226 219 215
     246 241 249 235 242];
```



- Let’s compare the gradient norms in the two positions:
  - Gradient norm at (4,3): 73.4
  - Gradient norm at (2,3): 10.4
- What do these values tell us?

Gradient Norms

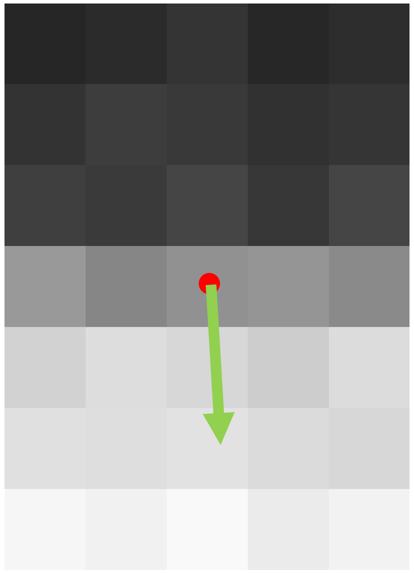
```
a = [38  43  52  39  45
     51  61  57  49  53
     63  58  69  55  69
     153 133 145 149 138
     210 221 215 205 220
     224 222 226 219 215
     246 241 249 235 242];
```



- Let’s compare the gradient norms in the two positions:
  - Gradient norm at (4,3): 73.4
  - Gradient norm at (2,3): 10.4
- What do these values tell us?
  - Pixel (4,3) is a much stronger edge pixel than pixel (2,3).
  - **The gradient norm measures the strength of an edge.**

Gradient Direction

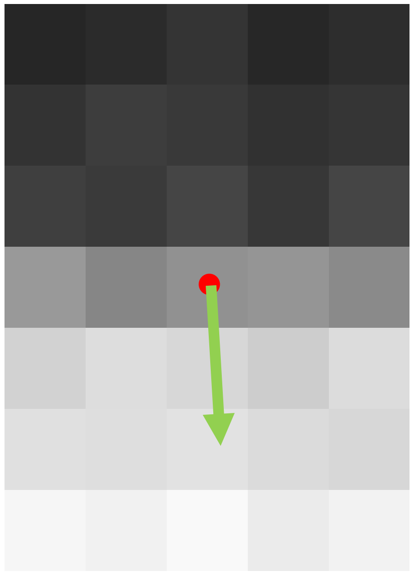
```
a = [38  43  52  39  45
     51  61  57  49  53
     63  58  69  55  69
     153 133 145 149 138
     210 221 215 205 220
     224 222 226 219 215
     246 241 249 235 242];
```



- We saw that the gradient at $(4,3)$ is $(73, 8)$.
- Let's plot an arrow pointing at the direction of the gradient:
  - It should go 73 units down for every 8 units right.
- What does the direction of the gradient tell us?

Gradient Direction

```
a = [38  43  52  39  45
     51  61  57  49  53
     63  58  69  55  69
     153 133 145 149 138
     210 221 215 205 220
     224 222 226 219 215
     246 241 249 235 242];
```



- We saw that the gradient at $(4,3)$ is $(73, 8)$.
- Let's plot an arrow pointing at the direction of the gradient:
  - It should go 73 units down for every 8 units right.
- What does the direction of the gradient tell us?
  - **The direction of the gradient is the direction in which intensity values increase the fastest.**

Computing Gradient Norms

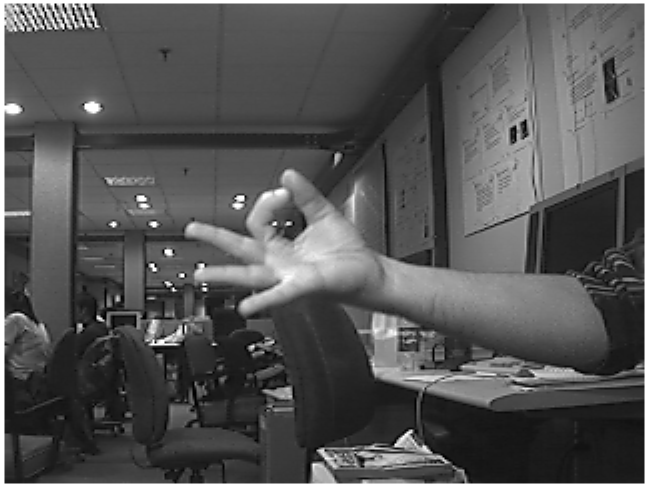
- Let:
  - dyA = imfilter(A, dy);
  - dxA = imfilter(A, dx);
- Gradient norm at pixel (y, x):
  - The norm of vector (dyA(y,x), dyA(y,x)).
  - sqrt(dyA(y,x)^2 + dxA(y,x)^2).
- The gradient norm operation identifies edge pixels at all orientations.
- Also useful for identifying smooth/rough textures.

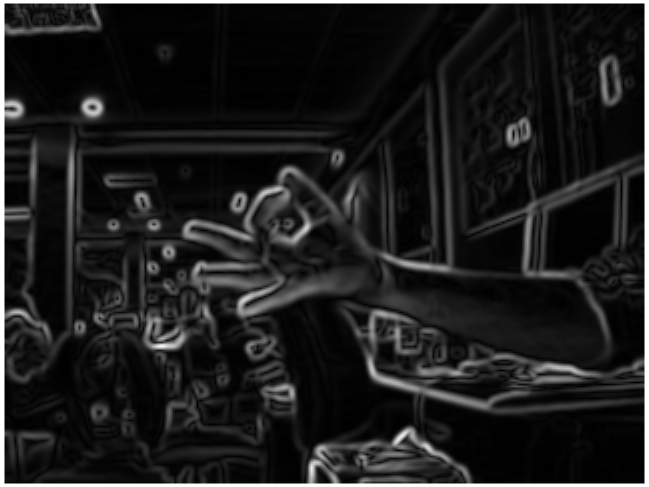
Computing Gradient Norms: Code

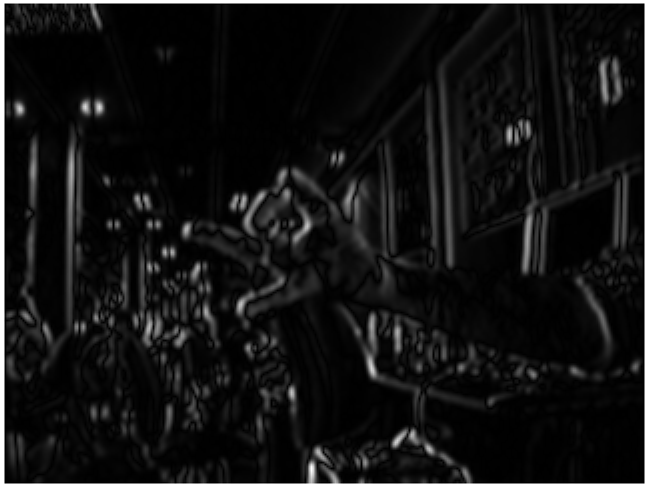
```
gray = read_gray('data/hand20.bmp');
dx = [-1 0 1; -2 0 2; -1 0 1] / 8;
dy = dx';

blurred_gray = blur_image(gray, 1.4, 1.4);
dxgray = imfilter(blurred_gray, dx, 'symmetric');
dygray = imfilter(blurred_gray, dy, 'symmetric');

% computing gradient norms
grad_norms = (dxgray.^2 + dygray.^2).^0.5;
```

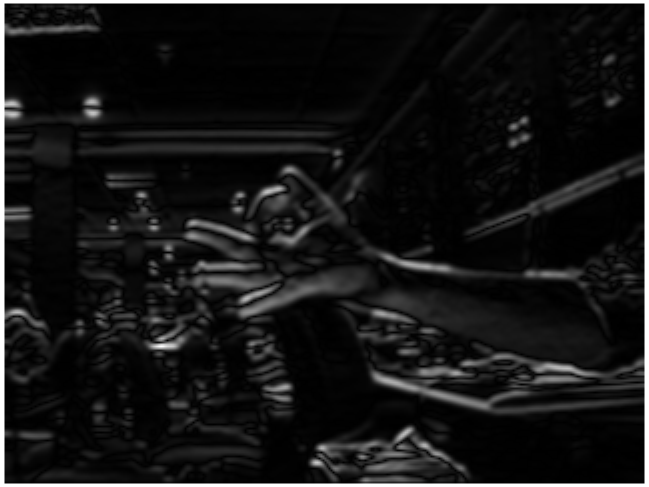
- See following functions online:
  - gradient_norms
  - blur_image

Gradient Norms: Results


gray


grad_norms


dxgray

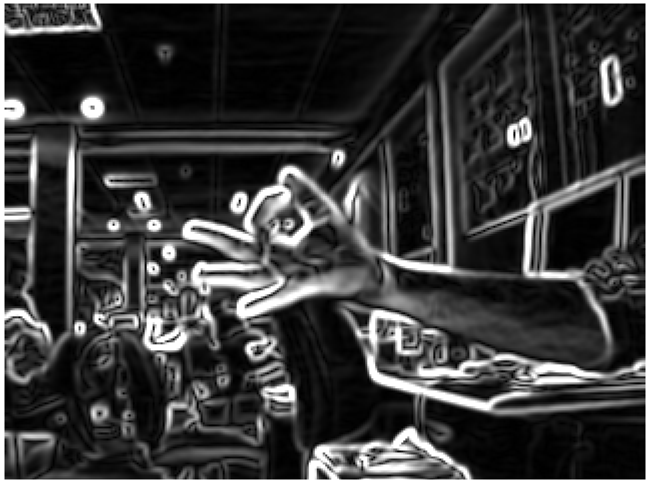

dygray

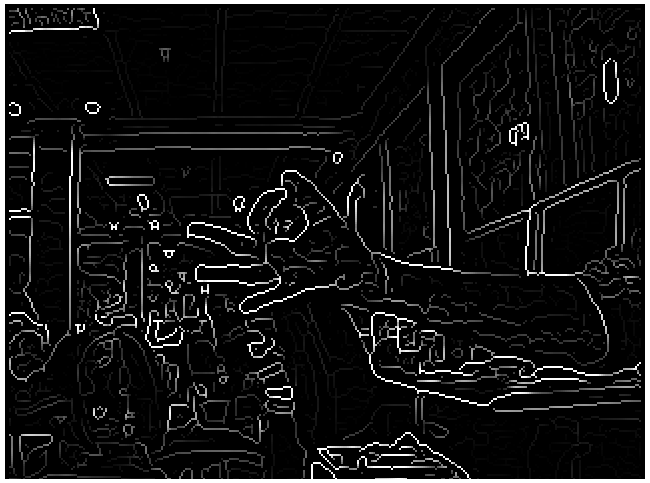
Notes on Gradient Norms

- Gradient norms detect edges at all orientations.
- So, gradient norms can be seen as "edge scores" that work for any orientation.
- However, gradient norms in themselves are not a good output for an edge detector:
  - We need thinner edges.
  - We need to decide which pixels are edge pixels, so that we can produce a binary edge image.

Non-Maxima Suppression

- Goal: produce thinner edges.
- Idea: for every pixel, decide if it is maximum along the direction of fastest change.
  - Preview of results:


gradient norms

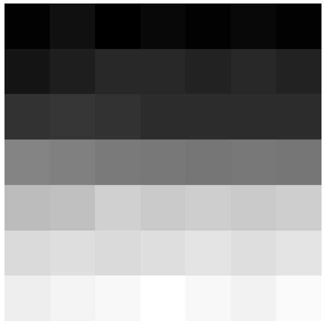

result of nonmaxima suppression

Nonmaxima Suppression Example

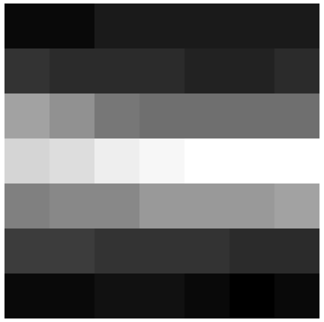
- Example:

```
 img = [112  118  111  115  112  115  112
        120  124  128  128  126  128  126
        132  134  132  130  130  130  130
        167  165  163  162  161  162  161
        190  192  199  196  198  196  198
        203  205  203  205  207  205  207
        212  214  216  219  216  213  217];

grad_norms = round(gradient_norms(img));
grad_norms = [ 5   5   7   7   7   7   7
              10   9   9   9   8   8   9
              23  21  18  17  17  17  17
              29  30  32  33  34  34  34
              19  20  20  22  22  22  23
              11  11  10  10  10   9   9
               5   5   6   6   5   4   5];
```

img



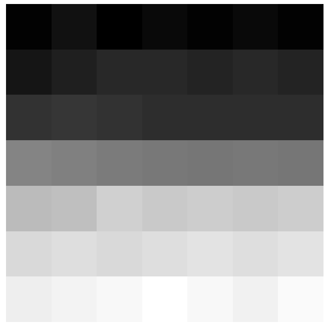
grad_norms



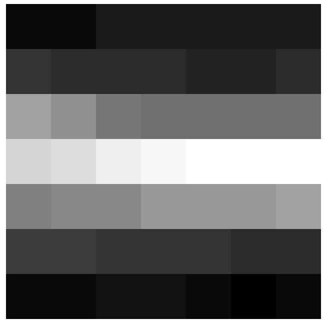
Nonmaxima Suppression Example

- Should we keep pixel (3,3)?
- result of dy filter [-0.5; 0; 0.5]
  - (img(4,3) – img(2, 3)) / 2 = 17.5.
- result of dx filter [-0.5 0 0.5]
  - (img(3,4) – img(3, 2)) / 2 = -2.
- Gradient = (17.5, -2).
- Gradient direction:
  - atan2(17.5, -2) = 1.68 rad = 96.5 deg.
  - Unit vector at gradient direction:
    - [0.9935, -0.1135]   (y direction, x direction)

img



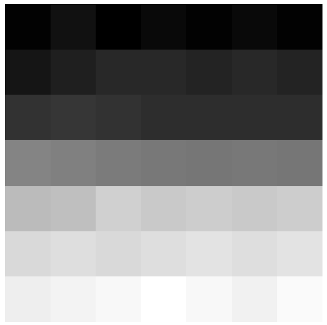
grad_norms



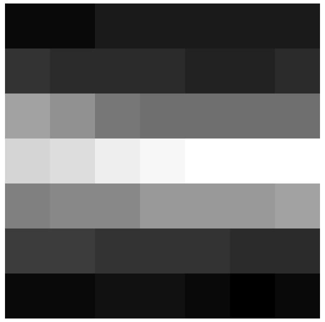
Nonmaxima Suppression Example

- Should we keep pixel (3,3)?
- Gradient direction: 96.5 degrees
  - Unit vector: disp = [0.9935, -0.1135].
  - disp defines the direction along which pixel(3,3) must be a local maximum.
  - Positions of interest:
    - [3,3] + disp, [3,3] – disp.
  - We compare grad_norms(3,3) with:
    - grad_norms(3.9935, 2.8865), and
    - grad_norms(2.0065, 3.1135)

img



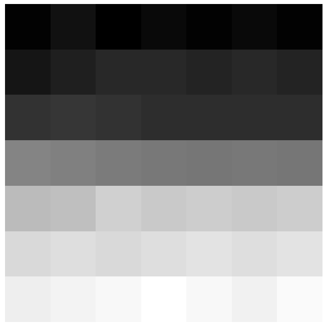
grad_norms



Nonmaxima Suppression Example

- We compare grad_norms(3,3) with:
  - grad_norms(3.9935, 2.8865), and
  - grad_norms(2.0065, 3.1135)
- grad_norms(3.9935, 2.8865) = ?
  - Use bilinear interpolation.
  - (3.9935, 2.8865) is surrounded by:
    - (3,2) at the top and left.
    - (3,3) at the top and right.
    - (4,2) at the bottom and left.
    - (4,3) at the bottom and right.

img



grad_norms

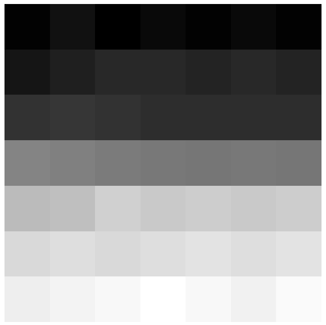
Nonmaxima Suppression Example

- grad_norms(3.9935, 2.8865) = ?
  - Weighted average of surrounding pixels.

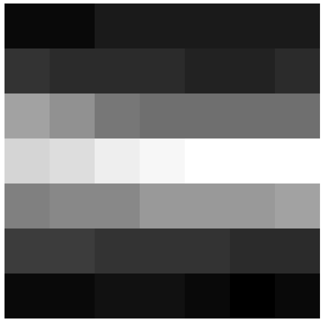
```
top_left = image(top, left);
top_right = image(top, right);
bottom_left = image(bottom, left);
bottom_right = image(bottom, right);
wy = 3.9935 – 3 = 0.9935;
wx = 2.8865 – 2 = 0.8865;
result = (1 – wx) * (1 – wy) * top_left +
         wx * (1 - wy) * top_right +
         (1 – wx) * y * bottom_left +
         x * y * bottom_right;
```

  - See function bilinear_interpolation online.

`img`



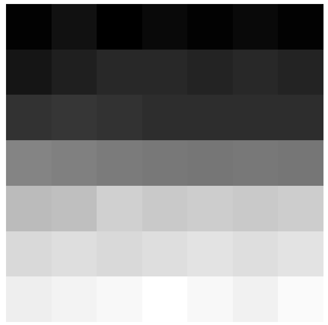
`grad_norms`



Nonmaxima Suppression Example

- grad_norms(3.9935, 2.8865) = 33.3
- grad_norms(2.0065, 3.1135) = 10.7
- grad_norms(3,3) = 18
  - Position 3,3 is not a local maximum in the direction of the gradient.
  - Position 3,3 is set to zero in the result of non-maxima suppression
  - Same test applied to all pixels.

img

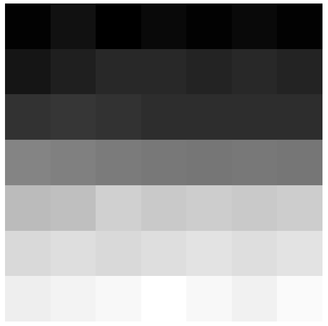


grad_norms

Nonmaxima Suppression Result

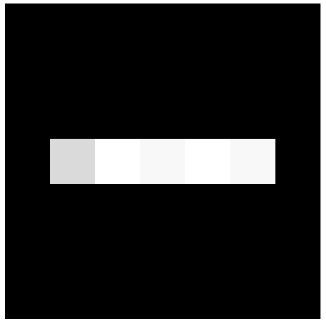
```
grad_norms =
[ 5   5   7   7   7   7   7
 10   9   9   9   8   8   9
 23  21  18  17  17  17  17
 29  30  32  33  34  34  34
 19  20  20  22  22  22  23
 11  11  10  10  10   9   9
  5   5   6   6   5   4   5];
```

img



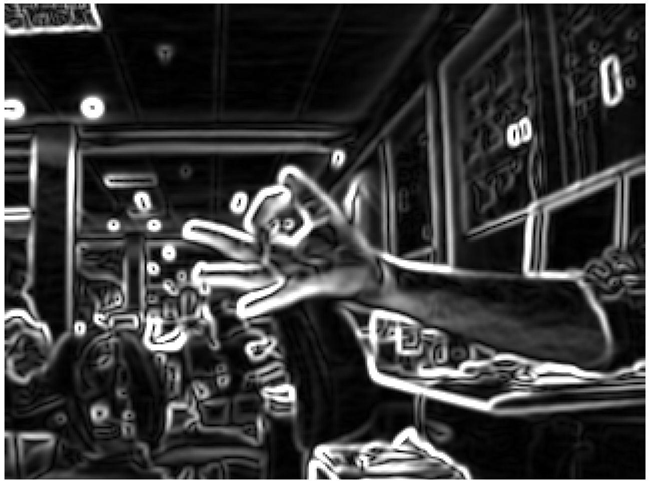
grad_norms

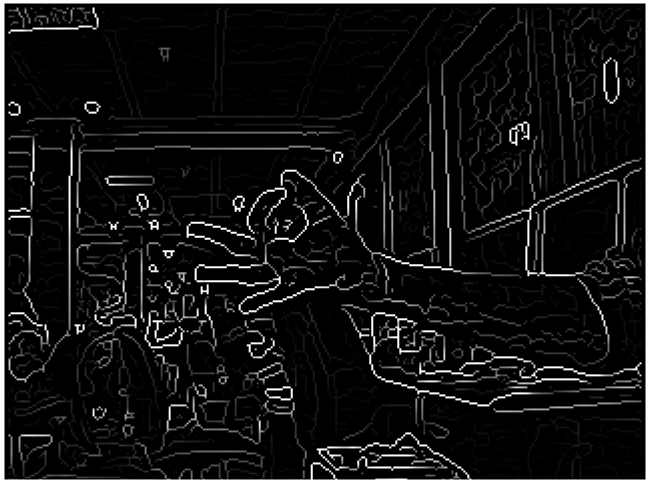
```
nonmaxima_suppression(grad_norms, thetas, 1) =
[ 0   0   0   0   0   0   0
  0   0   0   0   0   0   0
  0   0   0   0   0   0   0
  0  30  32  33  34  34   0
  0   0   0   0   0   0   0
  0   0   0   0   0   0   0
  0   0   0   0   0   0   0
```



result of non-maxima suppression

Nonmaxima Suppression Result


gradient norms


result of nonmaxima suppression

Side Note: Bilinear Interpolation

- grad_norms(3.9935, 2.8865) = ?
  - Weighted average of surrounding pixels.

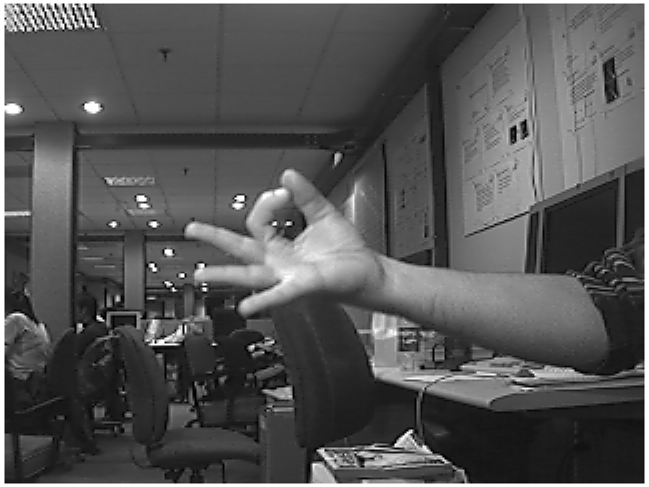
```
top_left = image(top, left);
top_right = image(top, right);
bottom_left = image(bottom, left);
bottom_right = image(bottom, right);
wy = 3.9935 – 3 = 0.9935;
wx = 2.8865 – 2 = 0.8865;
result = (1 – wx) * (1 – wy) * top_left +
         wx * (1 - wy) * top_right +
         (1 – wx) * y * bottom_left +
         wx * wy * bottom_right;
```

- Interpolation is a very common operation.
  - Images are discrete, sometimes it is convenient to treat them as continuous values.

bilinear_interpolation.m

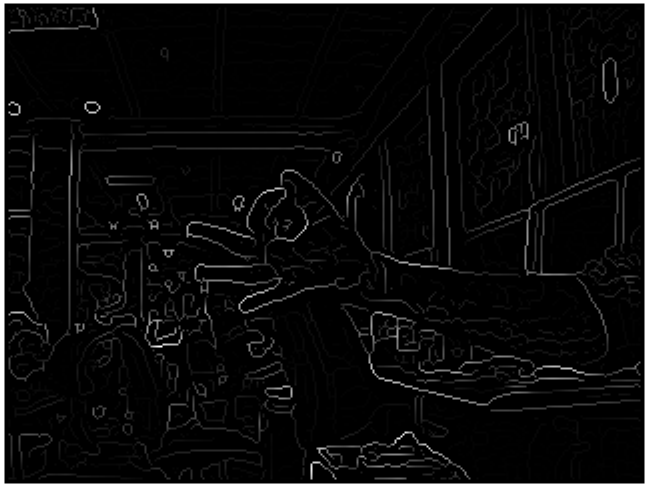
```
function result = bilinear_interpolation(image, row, col)

% row and col are non-integer coordinates, and this function
% computes the value at those coordinates using bilinear interpolation.

% Get the bounding square.
top = floor(row);
left = floor(col);
bottom = top + 1;
right = left + 1;

% Get values at the corners of the square
top_left = image(top, left);
top_right = image(top, right);
bottom_left = image(bottom, left);
bottom_right = image(bottom, right);

x = col - left;
y = row - top;

result = (1 - x) * (1 - y) * top_left;
result = result + x * (1 - y) * top_right;
result = result + x * y * bottom_right;
result = result + (1 - x) * y * bottom_left;
```

The Need for Thresholding

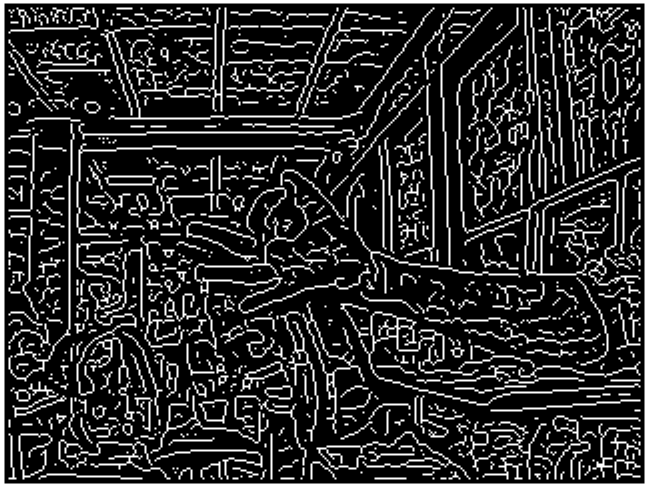


gray

- Many non-zero pixels in the result of nonmaxima suppression represent very weak edges.



nonmaxima

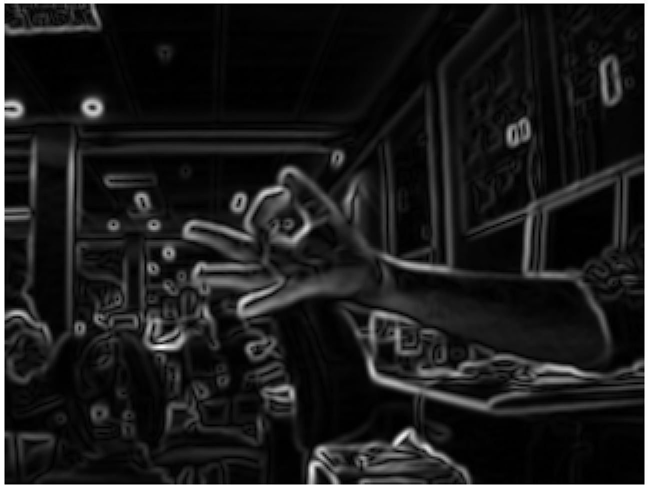


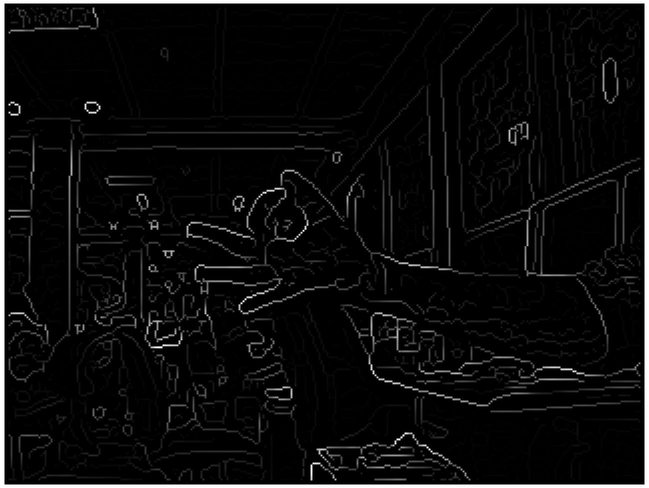
nonmaxima > 0

The Need for Thresholding

gray

- Decide which are the edge pixels:
  - Reject maxima with very small values.
  - Hysteresis thresholding.

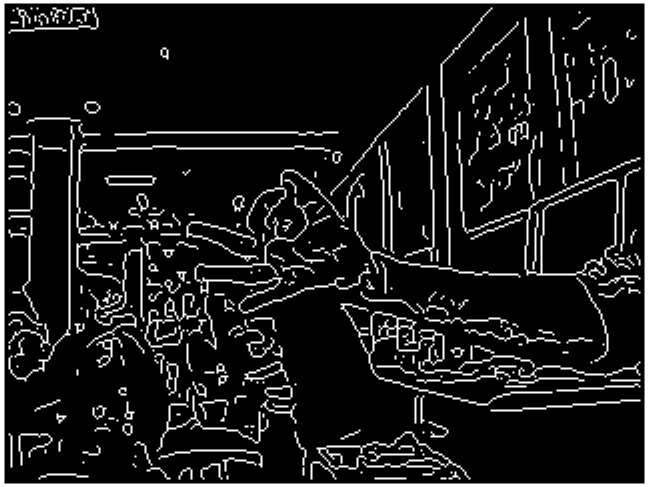

gradient norms


result of nonmaxima suppression

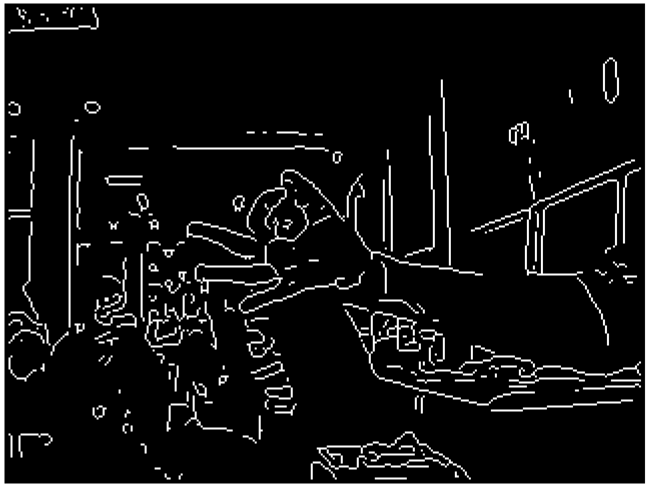
Hysteresis Thresholding

- Use two thresholds, t1 and t2.
- Pixels above t2 survive.
- Pixels below t1 do not survive.
- Pixels >= t1 and < t2 survive if:
  - They are connected to a pixel >= t2 via an 8-connected path of other pixels >= t1.

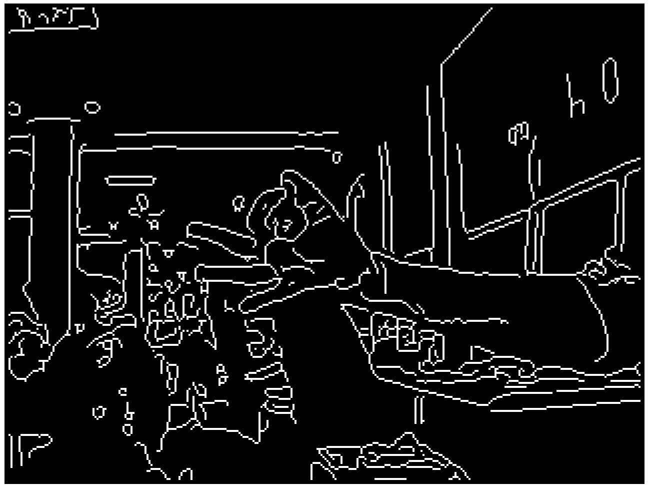
Hysteresis Thresholding Example



`A = nonmaxima >= 4`



`B = nonmaxima >= 8`



`C = hysthresh(nonmaxima, 4, 8)`

- A pixel is white in C if:
  - It is white in A, and
  - It is connected to a white pixel of B via an 8-connected path of white pixels in A.

Canny Edge Detection

- Blur input image.
- Compute dx, dy, gradient norms.
- Do non-maxima suppression on gradient norms.
- Apply hysteresis thresholding to the result of non-maxima suppression.
  - Check out these functions online:
    - blur_image
    - gradient_norms
    - gradient_orientations
    - nonmaxsup
    - hysthresh
    - canny
    - canny4

Side Note: Angles/Directions/Orientations

- To avoid confusion, you must specify:
  - Unit (degrees, or radians).
  - Do you use undirected or directed orientation?
    - Undirected: 180 degrees = 0 degrees.
    - Directed: 180 degrees != 0 degrees.
  - Which axis is direction 0? Pointing which way?
    - Class convention: direction 0 is x axis, pointing right.
  - Do angles increase clockwise or counterclockwise?
    - Class convention: clockwise.
  - Does the y axis point down? (in this class: yes)
  - What range of values do you allow/expect?
    - [-180, 180]? Any real number?

Side Note: Angles/Directions/Orientations

- Confusion and bugs stemming from different conventions are extremely common.
- When combining different code, **make sure that you account for different conventions**.

```
thetas =  atan2(dyb1gray, dxb1gray);

% atan2 convention:
% arguments: atan2(y, x)
% values: [-pi, pi]
% 0 degrees: x axis, pointing right
% 90 degrees: y axis points down
% values increase clockwise
```

Side Note: Edge Orientation

- How is the orientation of an edge pixel defined?

Side Note: Edge Orientation

- How is the orientation of an edge pixel defined?
  - It is the direction PERPENDICULAR to the gradient, i.e., the (dx, dy) vector for that pixel.
  - Typically (not always) 0 degrees = 180 degrees.
    - In other words, typically we do not care about the direction of the orientation.