Image processing

Brian Curless
CSEP 557
Spring 2019

Reading

Jain, Kasturi, Schunck, *Machine Vision*. McGraw-Hill, 1995. Sections 4.2-4.4, 4.5(intro), 4.5.5, 4.5.6, 5.1-5.4. [online handout]

What is an image?

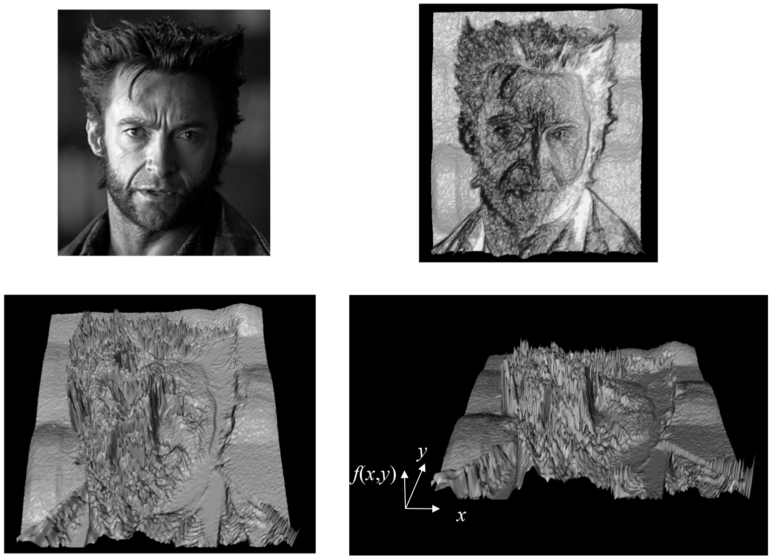
We can think of an **image** as a function, f, from R^2 to R:

- $f(x, y)$ gives the intensity of a channel at position (x, y)
- Realistically, we expect the image only to be defined over a rectangle, with a finite range:
  - $f : [a, b] \times [c, d] \rightarrow [0,1]$

A color image is just three functions pasted together. We can write this as a "vector-valued" function:

$$f(x,y) = \begin{bmatrix} r(x,y) \\ g(x,y) \\ b(x,y) \end{bmatrix}$$

Images as functions



What is a digital image?

In computer graphics, we usually operate on **digital** (**discrete**) images:

- **Sample** the space on a regular grid
- **Quantize** each sample (round to nearest integer)

If our samples are Δ apart, we can write this as:

$$f[i,j] = \text{Quantize}\{f(i\Delta, j\Delta)\}$$

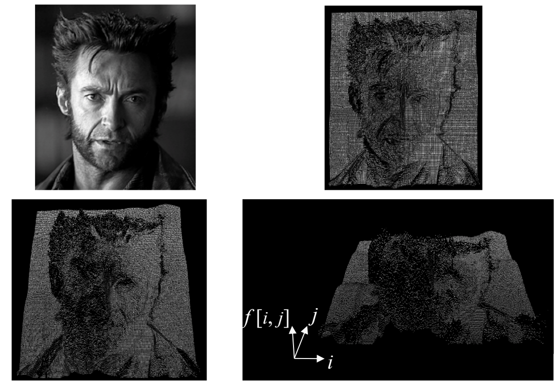


Image processing

An **image processing** operation typically defines a new image g in terms of an existing image f.

The simplest operations are those that transform each pixel in isolation. These pixel-to-pixel operations can be written:

$$g(x,y) = t(f(x,y))$$

Examples: threshold, RGB → grayscale

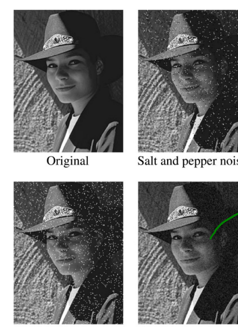
Note: a typical choice for mapping to grayscale is to apply the YIQ television matrix and keep the Y.

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.275 & -0.321 \\ 0.212 & -0.523 & 0.311 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Use Y for computing gradients in Impressionist

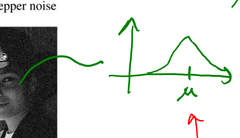
Noise

Image processing is also useful for noise reduction and edge enhancement. We will focus on these applications for the remainder of the lecture…



Original | Salt and pepper noise | Impulse noise | Gaussian noise

$N(\mu; \sigma)$



Common types of noise:

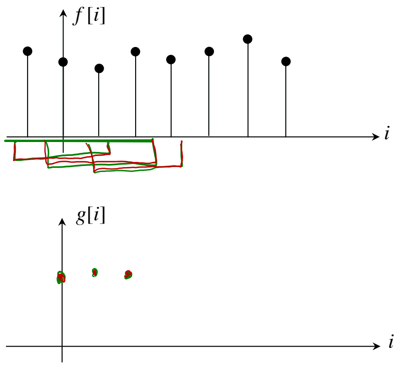
- **Salt and pepper noise**: contains random occurrences of black and white pixels
- **Impulse noise:** contains random occurrences of white pixels
- **Gaussian noise**: variations in intensity drawn from a Gaussian normal distribution

Ideal noise reduction

Ideal noise reduction

Practical noise reduction

How can we "smooth" away noise in a single image?



Is there a more abstract way to represent this sort of operation? Of course there is!

Discrete convolution

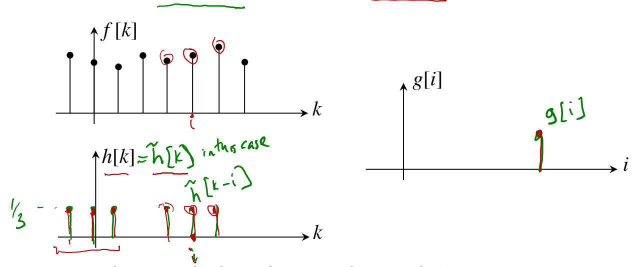
One of the most common methods for filtering an image is called discrete convolution. (We will just call this "convolution" from here on.)

$f * h$

In 1D, convolution is defined as:

$$
\begin{aligned}
g[i] &= f[i] * h[i] \\
&= \sum_k f[k]\,h[i-k] \\
&= \sum_k f[k]\,\tilde{h}[k-i]
\end{aligned}
$$

where $\tilde{h}[i] = h[-i]$.



"Flipping" the kernel (i.e., working with $h[-i]$) is mathematically important. In practice, though, you can assume kernels are pre-flipped unless I say otherwise.

Convolution in 2D

In two dimensions, convolution becomes:

$$
\begin{aligned}
g[i,j] &= f[i,j] * h[i,j] \\
&= \sum_{\ell}\sum_{k} f[k,\ell]\,h[i-k,\,j-\ell] \\
&= \sum_{\ell}\sum_{k} f[k,\ell]\,\tilde{h}[k-i,\,\ell-j]
\end{aligned}
$$

where $\tilde{h}[i,j] = h[-i,-j]$.

Again, "flipping" the kernel (i.e., working with $h[-i,-j]$) is mathematically important. In practice, though, you can assume kernels are pre-flipped unless I say otherwise.

Convolving in 2D

Since f and h are defined over finite regions, we can write them out as two-dimensional arrays:

Image $f[i,j]$

128	54	9	78	100
145	98	240	233	86
89	177	246	228	127
67	90	255	148	95
106	111	128	84	172
221	154	97	69	94

Filter $h[i,j]$ — filter kernel

0.1	0.1	0.1
0.1	0.2	0.1
0.1	0.1	0.1

3x3 footprint
support

- This is **not** matrix multiplication.
- For color images, filter each color channel separately.
- The *filter* is assumed to be zero outside its boundary.

Q: What happens at the boundary of the *image*?

Convolving in 2D

Since f and h are defined over finite regions, we can write them out as two-dimensional arrays:

Image $f[i,j]$

128	54	9	78	100
145	98	240	233	86
89	177	246	228	127
67	90	255	148	95
106	111	128	84	172
221	154	97	69	94

255
128
97

Filter $h[i,j]$

X 0.1	X 0.1	X 0.1
X 0.1	X 0.2	X 0.1
X 0.1	X 0.1	X 0.1

- This is **not** matrix multiplication.
- For color images, filter each color channel separately.
- The *filter* is assumed to be zero outside its boundary.

Q: What happens at the boundary of the *image*?

Normalization

Suppose f is a flat / constant image, with all pixel values equal to some value C.

weight, coefficients

C	C	C	C	C
C	C X h_{13}	C X h_{23}	C X h_{33}	C
C	C X h_{12}	C X h_{22}	C X h_{32}	C
C	C X h_{11}	C X h_{21}	C X h_{31}	C
C	C	C	C	C
C	C	C	C	C

$$
\begin{aligned}
g[i,j] &= C\cdot h_{3} + C\cdot h_{23} + C\cdot h_{33} + \cdots \\
&= \sum_{k,\ell} C\cdot h_{k,\ell} = C\sum h_{k,\ell}
\end{aligned}
$$

$$h[i,j] \leftarrow \frac{h[i,j]}{\sum h_{k,\ell}}$$

$$\sum_{i,j} \frac{h_{i,j}}{\sum_{k,\ell} h_{k,\ell}} = \frac{1}{\sum_{k,\ell} h_{k,\ell}} \sum_{i,j} h_{i,j} = 1$$

Q: What will be the value of each pixel after filtering?

Q: How do we avoid getting a value brighter or darker than the original image?

Mean filters

How can we represent our noise-reducing averaging as a convolution filter (know as a **mean filter**)?

$$\begin{bmatrix} 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \end{bmatrix}$$

$$\frac{1}{N^2}\underbrace{\begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{bmatrix}}_{N} \Bigg\} N \qquad \underbrace{\begin{bmatrix} 1/N^2 & \cdots & 1/N^2 \\ \vdots & \ddots & \vdots \\ 1/N^2 & \cdots & 1/N^2 \end{bmatrix}}_{N} N$$

Effect of mean filters

Gaussian noise | Salt and pepper noise

3x3

5x5

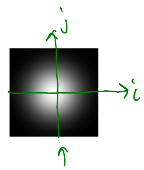
7x7

Gaussian filters

$e^{-\frac{(x^2+y^2)}{2\sigma^2}}$

Gaussian filters weigh pixels based on their distance from the center of the convolution filter. In particular:

$$h[i,j] = \frac{e^{-(i^2+j^2)/(2\sigma^2)}}{C}$$



This does a decent job of blurring noise while preserving features of the image.

What parameter controls the width of the Gaussian? σ

What happens to the image as the Gaussian filter kernel gets wider? blurrier

What is the constant C? What should we set it to?

$$C = \sum_{i,j} e^{-(i^2+j^2)/2\sigma^2}$$

Effect of Gaussian filters

Gaussian noise | Salt and pepper noise

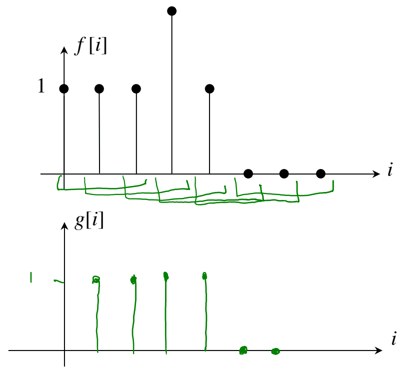
3x3

5x5

7x7

Median filters

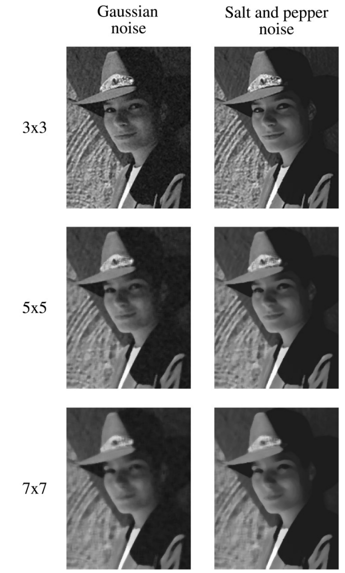
A **median filter** operates over an N x N region by selecting the median intensity in the region.



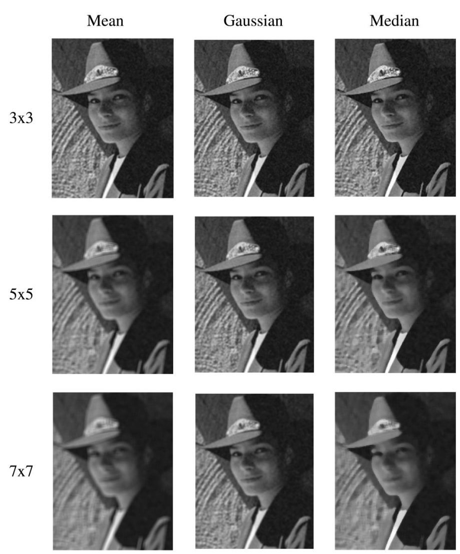
What advantage does a median filter have over a mean filter? outlier removal, edge preserve

Is a median filter a kind of convolution? No.

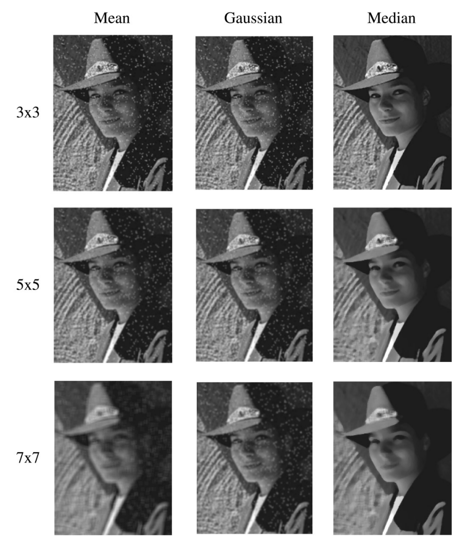
Effect of median filters



Comparison: Gaussian noise

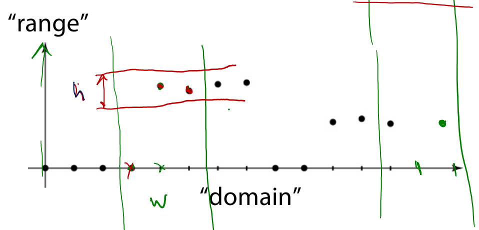


Comparison: salt and pepper noise



Mean bilateral filtering

Bilateral filtering is a method to average together nearby samples only if they are similar in value.



This is a "mean bilateral filter" where you take the average of everything that is both within the domain footprint (w x w in 2D) and range height (h). You must sum up all pixels you find in that "box" and then divide by the number of pixels.

Q: What happens as the range size becomes large?

mean filter

Q: Will bilateral filtering take care of impulse noise? No.

2D Mean bilateral filtering

Now consider filtering an image with a bilateral filter with a 3x3 domain and a total range height of 40 (i.e., range of [-20, 20] from center pixel).

205	198	190	203	210	192
191	203	191	194	206	98
210	197	204	101	98	103
205	199	104	97	94	107
190	92	106	106	100	108
110	91	101	100	96	99

$$\frac{203+210+192+194+206}{5}$$

$$\frac{104+92+106+110+91+101}{6}$$

Color bilateral filtering

Finally, for color, we simply compute range distance in R, G, B space as the length of the vector between the two color vectors. Consider colors at different pixels:

$$C_1 = \begin{bmatrix} R_1 \\ G_1 \\ B_1 \end{bmatrix} \qquad C_2 = \begin{bmatrix} R_2 \\ G_2 \\ B_2 \end{bmatrix}$$

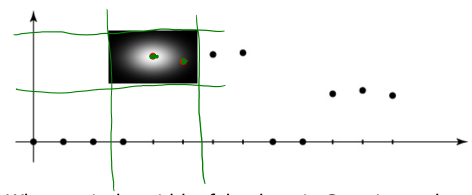
The range distance between them is then:

$$\|C_1 - C_2\|_2 = \sqrt{(R_1-R_2)^2 + (G_1-G_2)^2 + (B_1-B_2)^2}$$

After selecting pixels that are within the color distance range, you then separately average each color channel of those pixels to compute the final color.

Gaussian bilateral filtering

We can also change the filter to something "nicer" like Gaussians. Let's go back to 1D:



Where σ_d is the width of the domain Gaussian and σ_r is the width of the range Gaussian.

Note that we can write a 2D Gaussian as a product of two Gaussians (i.e., it is a separable function):

$$h[i,r] \sim e^{-(i^2+r^2)/(2\sigma^2)} = e^{-i^2/(2\sigma^2)} e^{-r^2/(2\sigma^2)}$$

where i indexes the spatial domain and r is the range difference. This would make a round Gaussian. We can make it elliptical by having different σ's for domain and range:

$$h[i,r] \sim e^{-i^2/(2\sigma_d^2)} e^{-r^2/(2\sigma_r^2)}$$
$$\sim h_d(i)\, h_r(r)$$

The math: 1D bilateral filtering

Recall that convolution looked like this:

$$g[i] = \frac{1}{C} \sum_k f[k] h_d[i-k]$$

with normalization (sum of filter values):

$$C = \sum_k h_d[i-k]$$

This was just domain filtering.

The bilateral filter is similar, but includes both domain and range filtering:

$$g[i] = \frac{1}{C} \sum_k f[k] h_d[i-k]\ h_r(f[i] - f[k])$$

with normalization (sum of filter values):

$$C = \sum_k h_d[i-k]\ h_r(f[i] - f[k])$$

Note that with regular convolution, we pre-compute C once, but for bilateral filtering, we must compute it at each pixel location where it's applied.

The math: 2D bilateral filtering

In 2D, bilateral filtering generalizes to having a 2D domain, but still a 1D range:

$$g[i,j] = \frac{1}{C} \sum_{k,\ell} f[k,\ell] h_d[i-k, j-\ell]\ h_r(f[i,j] - f[k,\ell])$$

And the normalization becomes (sum of filter values):

$$C = \sum_{k,\ell} h_d[i-k, j-\ell]\ h_r(f[i,j] - f[k,\ell])$$

For Gaussian filtering, the new form looks like this:

$$h[i,j,r] \sim e^{-(i^2+j^2)/(2\sigma_d^2)} e^{-r^2/(2\sigma_r^2)}$$
$$\sim h_d(i,j)\ h_r(r)$$

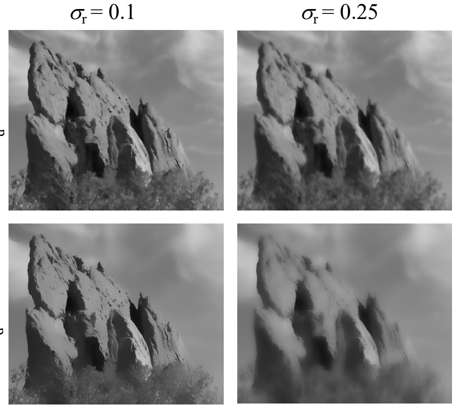
Note that Gaussian bilateral filtering is slow without some optimization, and some optimizations can be fairly complicated (if worthwhile). Fortunately, simple mean bilateral filtering is fairly fast and works well in practice.

Input

$\sigma_r = 0.1$ | $\sigma_r = 0.25$

$\sigma_d = 2$

$\sigma_d = 6$



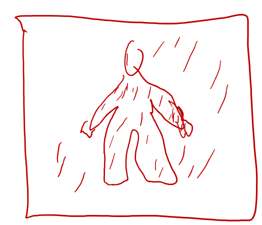
Paris, et al. SIGGRAPH course notes 2007

Edge detection

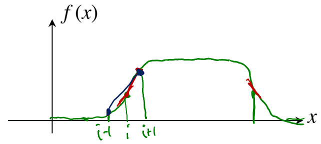
One of the most important uses of image processing is **edge detection:**

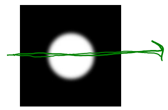
- Really easy for humans
- Really difficult for computers

- Fundamental in computer vision
- Important in many graphics applications



What is an edge?





Q: How might you detect an edge in 1D?

$\left|\frac{df}{dx}\right| > \text{thresh}$

Q: How might you approximate this edge measure with discrete samples?

finite difference

$\frac{df}{dx}[i] \approx f[i+1] - f[i]$

central difference

$\frac{df}{dx}[i] \approx \frac{f[i+1] - f[i-1]}{2}$

Q: How could you do it with discrete convolution?

$\frac{df}{dx}[i] \approx f * h$

$\frac{df}{dx}[i] \approx (-1)\cdot f[i] + (1)\cdot f[i+1]$

$[-1 \quad 1] = [0 \quad -1 \quad 1] = \tilde{h}[i]$

$h[i] = [1 \quad -1 \quad 0]$

Gradients

The **gradient** is the 2D equivalent of the derivative:

$$\nabla f(x,y) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right)$$

$\theta = \tan^{-1}\left(\frac{\partial f/\partial y}{\partial f/\partial x}\right)$

$\|f\|_2 = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2}$

$h_x = [1 \quad -1 \quad 0]$

$h_y = \begin{bmatrix} 0 \\ -1 \\ 1 \end{bmatrix}$

Properties of the gradient

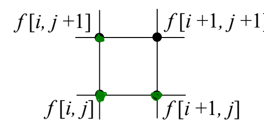
- It's a vector
- Points in the direction of maximum increase of f
- Magnitude is rate of increase

$\tilde{h}_x = [0 \quad -1 \quad 1]$

$\tilde{h}_y = \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix}$

Note: use `atan2(y,x)` to compute the angle of the gradient (or any 2D vector).

How can we approximate the gradient in a discrete image using finite differences?

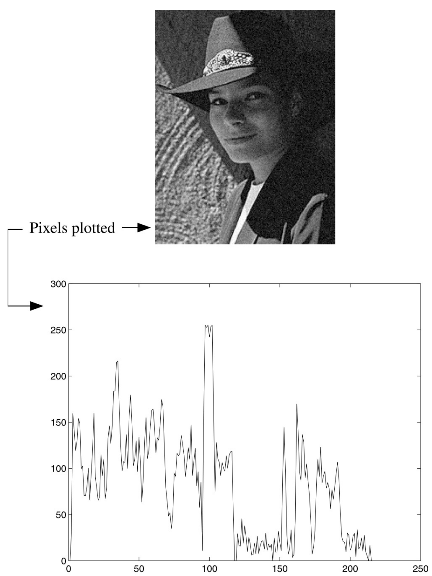


$\frac{\partial f}{\partial x} \approx f[i+1, j] - f[i, j]$

$\frac{\partial f}{\partial y} \approx f[i, j+1] - f[i, j]$

$\frac{\partial f}{\partial x} \approx f * h_x \quad \frac{\partial f}{\partial y} \approx f * h_y$

Less than ideal edges



Steps in edge detection

Edge detection algorithms typically proceed in three or four steps:

- **Filtering**: cut down on noise
- **Enhancement**: amplify the difference between edges and non-edges
- **Detection**: use a threshold operation
- **Localization** (optional): estimate geometry of edges as 1D contours that can pass between pixels

Edge enhancement

Use Sobel for gradients, applied to Y-channel (in Impressionist)

A popular gradient filter is the **Sobel operator**:

$$\tilde{s}_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

$\frac{\partial f}{\partial x}[i] \approx s_x * f$

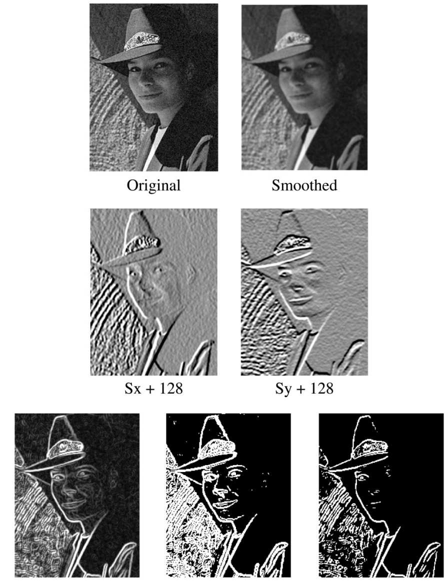
$$\tilde{s}_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

$\frac{\partial f}{\partial y}[i] \approx s_y * f$

We can then compute the magnitude of the vector $(\tilde{s}_x, \tilde{s}_y)$.

Note that these operators are conveniently "pre-flipped" for convolution, so you can directly slide these across an image without flipping first.

Results of Sobel edge detection

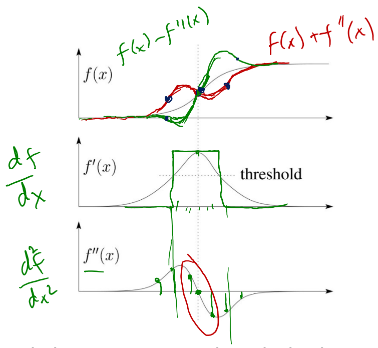


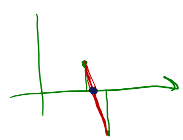
Original — Smoothed

Sx + 128 — Sy + 128

Magnitude — Threshold = 64 — Threshold = 128

Second derivative operators





The Sobel operator can produce thick edges. Ideally, we're looking for infinitely thin boundaries.

An alternative approach is to look for local extrema in the first derivative: places where the change in the gradient is highest.

Q: A peak in the first derivative corresponds to what in the second derivative?

zero

Constructing a second derivative filter

We can construct a second derivative filter from the first derivative.

First, one can show that convolution has some convenient properties. Given functions a, b, c:

$$\text{Commutative:} \quad a * b = b * a$$
$$\text{Associative:} \quad (a * b) * c = a * (b * c)$$
$$\text{Distributive:} \quad a * (b + c) = a * b + a * c$$

The "flipping" of the kernel is needed for associativity. Now let's use associativity to construct our second derivative filter…

$$\frac{d^2f}{dx^2} = \frac{d}{dx}\cdot\frac{df}{dx} \approx \frac{dg}{dx} \qquad g = \frac{df}{dx} \approx h_x * f$$

$$\frac{dg}{dx} \approx h_x * g \approx h_x * (h_x * f)$$

$$\frac{dg}{dx} = \frac{d^2f}{dx^2} \approx h_x * (h_x * f) \approx (h_x * h_x) * f$$

$h_x * h_x = h_{xx}$

$$\frac{d^2f}{dx^2} \approx h_{xx} * f$$

$$h_{xx} = h_x * h_x$$

Constructing a second derivative filter

The second derivative filter is then: $h_x * h_x$

h_x

1	-1	0

0	-1	1

$\tilde{h}_x$

$h_{xx} =$

$\begin{bmatrix} 1 & -2 & 1 \end{bmatrix}$

Localization with the Laplacian

An equivalent measure of the second derivative in 2D is the **Laplacian**:

$$\nabla^2 f(x,y) = \underline{\frac{\partial^2 f}{\partial x^2}} + \underline{\frac{\partial^2 f}{\partial y^2}} \approx h_{xx} * f + h_{yy} * f$$

$$= \underbrace{(h_{xx} + h_{yy})}_{} * f$$

Using the same arguments we used to compute the gradient filters, we can derive a Laplacian filter to be:

$h_{xx} = \begin{bmatrix} 1 & -2 & 1 \end{bmatrix}$

$$\Delta = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

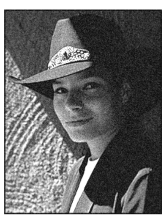
$h_{yy} = \begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix}$

$\underbrace{\begin{bmatrix} 0 & 0 & 0 \\ 1 & -2 & 1 \\ 0 & 0 & 0 \end{bmatrix}}_{h_{xx}} + \underbrace{\begin{bmatrix} 0 & 1 & 0 \\ 0 & -2 & 0 \\ 0 & 1 & 0 \end{bmatrix}}_{h_{yy}}$

(The symbol Δ is often used to refer to the *discrete* Laplacian filter.)

Zero crossings in a Laplacian filtered image can be used to localize edges.

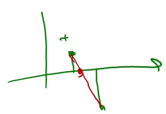
Localization with the Laplacian

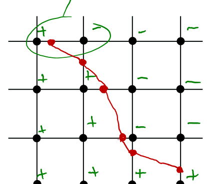


Original

Smoothed

Laplacian (+128)





marching squares

Sharpening with the Laplacian

$f - \lambda \Delta * f$

$I * f - \lambda \Delta * f$

$\underline{(I - \lambda \Delta)} * f$

$\lambda = \frac{1}{2} \Rightarrow \frac{1}{2}\begin{bmatrix} 0 & -1 & 0 \\ -1 & 6 & -1 \\ 0 & -1 & 0 \end{bmatrix}$

$\lambda = \frac{1}{4} \Rightarrow \frac{1}{4}\begin{bmatrix} 0 & -1 & 0 \\ -1 & 8 & -1 \\ 0 & -1 & 0 \end{bmatrix}$

$f - \Delta * f$

$= I * f - \Delta * f$

$= \underbrace{(I - \Delta)}_{\text{sharpen}} * f$

h_{sharp}

Original

Laplacian (+128)

$h_{sharp-better}$

$= I - \lambda \Delta$

$= [1] - \lambda \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$

$= \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} - \begin{bmatrix} 0 & \lambda & 0 \\ \lambda & -4\lambda & \lambda \\ 0 & \lambda & 0 \end{bmatrix}$

$= \begin{bmatrix} 0 & -\lambda & 0 \\ -\lambda & 1+4\lambda & -\lambda \\ 0 & -\lambda & 0 \end{bmatrix}$

$h_{sharp} = I - \Delta$

$= [1] - \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$

$= \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} - \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$

$= \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$

Original + Laplacian

Original - Laplacian

Why does the sign make a difference?

How can you write the filter that makes the sharpened image?

Summary

What you should take away from this lecture:

- The meanings of all the boldfaced terms.
- How noise reduction is done
- How discrete convolution filtering works
- The effect of mean, Gaussian, and median filters
- What an image gradient is and how it can be computed
- How edge detection is done
- What the Laplacian image is and how it is used in either edge detection or image sharpening