# Image Processing
CSE 6367: Computer Vision

Instructor: William J. Beksi

## Introduction

- **Image processing** is the first stage in most computer vision applications, namely preprocessing and conversion of the image into a form suitable for further analysis

## Introduction

- **Image processing** is the first stage in most computer vision applications, namely preprocessing and conversion of the image into a form suitable for further analysis
- Many computer vision applications require care in designing the image processing stages in order to achieve acceptable results

## Operations on Pixels

- The simplest kinds of image processing transforms are **point operators**, where each output pixel's value depends on only the corresponding input pixel value

## Operations on Pixels

- The simplest kinds of image processing transforms are **point operators**, where each output pixel's value depends on only the corresponding input pixel value
- Examples of such operators include brightness and contrast adjustments as well as color correction and transformations

## Pixel Transforms

- A general image processing *operator* is a function that takes one or more input images and produces an output image

## Pixel Transforms

- A general image processing *operator* is a function that takes one or more input images and produces an output image
- In the continuous domain, this can be denoted as

$$g(\mathbf{x}) = h(f(\mathbf{x})) \text{ or } g(\mathbf{x}) = h(f_0(\mathbf{x}), \dots, f_n(\mathbf{x}))$$

where **x** is in the D-dimensional *domain* of the functions and the functions $f$ and $g$ operate over some *range*, which can either be a scalar or vector-valued (e.g. for color images or 2D motion)

## Pixel Transforms

- For discrete (sampled) images, the domain consists of a finite number of pixel locations $\mathbf{x} = (i, j)$

## Pixel Transforms

- For discrete (sampled) images, the domain consists of a finite number of pixel locations $\mathbf{x} = (i, j)$
- We denote the discrete operator as

$$g(i,j) = h(f(i,j))$$

## Pixel Transforms

- Two commonly used point processes are multiplication and addition with a constant

  $$g(\mathbf{x}) = af(\mathbf{x}) + b$$

  where the parameters $a > 0$ and $b$ are called the **gain** and **bias**, and control **contrast** and **brightness**, respectively

## Pixel Transforms

- Two commonly used point processes are multiplication and addition with a constant

$$g(\mathbf{x}) = af(\mathbf{x}) + b$$

where the parameters $a > 0$ and $b$ are called the **gain** and **bias**, and control **contrast** and **brightness**, respectively

- The bias and gain can also be spatially varying

$$g(\mathbf{x}) = a(\mathbf{x})f(\mathbf{x}) + b(\mathbf{x})$$

## Pixel Transforms

- Multiplicative gain (both global and spatially varying) is a linear operation since it obeys the **superposition principle**

$$h(f_0 + f_1) = h(f_0) + h(f_1)$$

## Pixel Transforms

- Multiplicative gain (both global and spatially varying) is a linear operation since it obeys the **superposition principle**

$$h(f_0 + f_1) = h(f_0) + h(f_1)$$

- Operators such as image squaring (often used to get a local estimate of the energy in a band-pass filter) are not linear

## Pixel Transforms

- Another commonly used dyadic (two-input) operator is the **linear blend** operator

$$g(\mathbf{x}) = (1 - \alpha) f_0(\mathbf{x}) + \alpha f_1(\mathbf{x})$$

## Pixel Transforms

- Another commonly used dyadic (two-input) operator is the **linear blend** operator

$$g(\mathbf{x}) = (1-\alpha)f_0(\mathbf{x}) + \alpha f_1(\mathbf{x})$$

- By varying $\alpha$ from 0 to 1, this operator can be used to perform a temporal cross-dissolve between two images or videos

## Pixel Transforms

- One highly used non-linear transform that is often applied to images before further processing is **gamma correction**, which is used to remove the non-linear mapping between input radiance and quantized pixel values

## Pixel Transforms

- One highly used non-linear transform that is often applied to images before further processing is **gamma correction**, which is used to remove the non-linear mapping between input radiance and quantized pixel values
- To invert the gamma mapping applied by the sensor, we can use

$$g(\mathbf{x}) = [f(\mathbf{x})]^{1/\gamma}$$

where a gamma value of $\gamma \approx 2.2$ is reasonable for most digital cameras

## Color Transforms

- The color channel, with range $[0, 255]$, describes the apparent **intensity** of the pixel and can affect the **hue** (dominate wavelength) and **saturation** (intensity of color)

## Color Transforms

- The color channel, with range $[0, 255]$, describes the apparent **intensity** of the pixel and can affect the **hue** (dominate wavelength) and **saturation** (intensity of color)
- Color balancing can be performed by multiplying each channel with a different scale factor or by the more complex process of mapping to an XYZ color space (e.g. CIELAB)

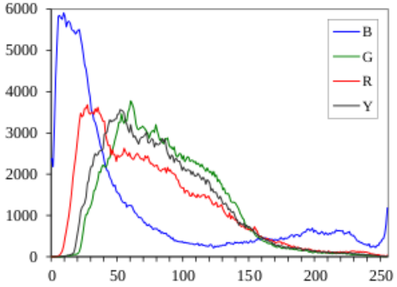
## Histogram Equalization

- How can we automatically determine brightness and gain values to improve the appearance of an image?

## Histogram Equalization

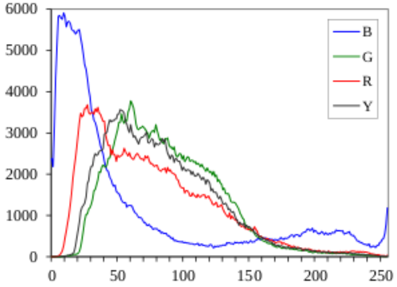
- How can we automatically determine brightness and gain values to improve the appearance of an image?
- How can we visualize the set of lightness values in an image in order to test different heuristics?

## Histogram Equalization



- We can plot the **histogram** of the individual color channels and luminance values

## Histogram Equalization



- We can plot the **histogram** of the individual color channels and luminance values
- From this distribution, we can compute relevant statistics such as the minimum, maximum, and average intensity values

## Example: Histogram Equalization using MATLAB

- Read an image into the workspace, and display the image and histogram:

```
I = imread(‘pout.tif’);
figure
subplot(1,2,1)
imshow(I)
subplot(1,2,2)
imhist(I,64)
```

## Example: Histogram Equalization using MATLAB

- Adjust the contrast using histogram equalization:

```
J = histeq(I);
subplot(1,2,1)
imshow(J)
subplot(1,2,2)
imhist(J,64)
```

## Histogram Equalization

- **Histogram equalization** finds an intensity mapping function $f(I)$ such that the resulting histogram is flat

## Histogram Equalization

- **Histogram equalization** finds an intensity mapping function $f(I)$ such that the resulting histogram is flat
- This is done by first computing the cumulative distribution function

$$c(I) = \frac{1}{N}\sum_{i=0}^{I} h(i) = c(I-1) + \frac{1}{N}h(I)$$

where $h(I)$ is the original histogram and $N$ is the number of pixels in the image

## Locally Adaptive Histogram Equalization

- Histogram equalization can result in blocking artifacts, i.e. intensity discontinuities at block boundaries

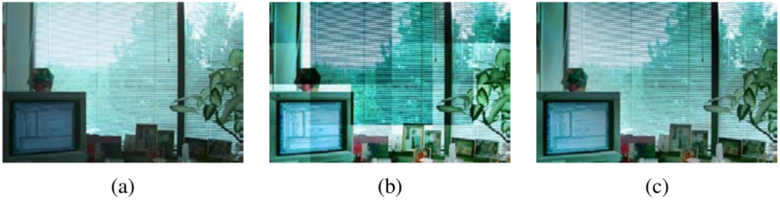
## Locally Adaptive Histogram Equalization

- Histogram equalization can result in blocking artifacts, i.e. intensity discontinuities at block boundaries
- One way to eliminate blocking artifacts is to use a **moving window**, i.e. to recompute the histogram for every $M \times M$ block centered at each pixel

## Locally Adaptive Histogram Equalization

- Histogram equalization can result in blocking artifacts, i.e. intensity discontinuities at block boundaries
- One way to eliminate blocking artifacts is to use a **moving window**, i.e. to recompute the histogram for every $M \times M$ block centered at each pixel
- A more efficient approach is to compute non-overlapped block-based equalization functions as before, but to then smoothly interpolate the transfer functions as we move between blocks (**adaptive histogram equalization**)

## Locally Adaptive Histogram Equalization



- Locally adaptive histogram equalization: (a) original image; (b) block histogram equalization; (c) full locally adaptive equalization

## Linear Filtering

- The most commonly used type of neighborhood operator is a **linear filter**, in which an output pixel's value is determined as a weighted sum of input pixel values

$$g(i,j) = \sum_{k,l} f(i+k, j+l) h(k,l) \tag{1}$$

## Linear Filtering

- The most commonly used type of neighborhood operator is a **linear filter**, in which an output pixel's value is determined as a weighted sum of input pixel values

$$g(i,j) = \sum_{k,l} f(i+k, j+l)h(k,l) \tag{1}$$

- The entries in the weight **kernel** or **mask** $h(k,l)$ are often called the **filter coefficients**

## Linear Filtering

- The most commonly used type of neighborhood operator is a **linear filter**, in which an output pixel's value is determined as a weighted sum of input pixel values

$$g(i,j) = \sum_{k,l} f(i+k, j+l)h(k,l) \tag{1}$$

- The entries in the weight **kernel** or **mask** $h(k,l)$ are often called the **filter coefficients**
- The above **correlation** operator can be more compactly written as

$$g = f \otimes h$$

## Linear Filtering

- A common variant of equation (1) is

$$
\begin{aligned}
g(i,j) &= \sum_{k,l} f(i-k, j-l)h(k,l) \\
&= \sum_{k,l} f(k,l)h(i-k, j-l)
\end{aligned}
$$

where the sign of the offsets in $f$ has been reversed

## Linear Filtering

- A common variant of equation (1) is

$$\begin{aligned} g(i,j) &= \sum_{k,l} f(i-k, j-l)h(k,l) \\ &= \sum_{k,l} f(k,l)h(i-k, j-l) \end{aligned}$$

  where the sign of the offsets in $f$ has been reversed

- This is called the **convolution** operator

$$g = f * h$$

  and $h$ is called the **impulse response function**

## Linear Filtering

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 45 | 60 | 98 | 127 | 132 | 133 | 137 | 133 |
| 46 | 65 | 98 | 123 | 126 | 128 | 131 | 133 |
| 47 | 65 | 96 | 115 | 119 | 123 | 135 | 137 |
| 47 | 63 | 91 | 107 | 113 | 122 | 138 | 134 |
| 50 | 59 | 80 | 97 | 110 | 123 | 133 | 134 |
| 49 | 53 | 68 | 83 | 97 | 113 | 128 | 133 |
| 50 | 50 | 58 | 70 | 84 | 102 | 116 | 126 |
| 50 | 50 | 52 | 58 | 69 | 86 | 101 | 120 |

$f(x,y)$

$*$

| | | |
|---|---|---|
| 0.1 | 0.1 | 0.1 |
| 0.1 | 0.2 | 0.1 |
| 0.1 | 0.1 | 0.1 |

$h(x,y)$

$=$

| | | | | | |
|---|---|---|---|---|---|
| 69 | 95 | 116 | 125 | 129 | 132 |
| 68 | 92 | 110 | 120 | 126 | 132 |
| 66 | 86 | 104 | 114 | 124 | 132 |
| 62 | 78 | 94 | 108 | 120 | 129 |
| 57 | 69 | 83 | 98 | 112 | 124 |
| 53 | 60 | 71 | 85 | 100 | 114 |

$g(x,y)$

- Neighborhood filtering (convolution): The image on the left is convolved with the filter in the middle to yield the image on the right

## Linear Filtering

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 45 | 60 | 98 | 127 | 132 | 133 | 137 | 133 |
| 46 | 65 | 98 | 123 | 126 | 128 | 131 | 133 |
| 47 | 65 | 96 | 115 | 119 | 123 | 135 | 137 |
| 47 | 63 | 91 | 107 | 113 | 122 | 138 | 134 |
| 50 | 59 | 80 | 97 | 110 | 123 | 133 | 134 |
| 49 | 53 | 68 | 83 | 97 | 113 | 128 | 133 |
| 50 | 50 | 58 | 70 | 84 | 102 | 116 | 126 |
| 50 | 50 | 52 | 58 | 69 | 86 | 101 | 120 |

$f(x,y)$

*

| | | |
|---|---|---|
| 0.1 | 0.1 | 0.1 |
| 0.1 | 0.2 | 0.1 |
| 0.1 | 0.1 | 0.1 |

$h(x,y)$

=

| | | | | | |
|---|---|---|---|---|---|
| 69 | 95 | 116 | 125 | 129 | 132 |
| 68 | 92 | 110 | 120 | 126 | 132 |
| 66 | 86 | 104 | 114 | 124 | 132 |
| 62 | 78 | 94 | 108 | 120 | 129 |
| 57 | 69 | 83 | 98 | 112 | 124 |
| 53 | 60 | 71 | 85 | 100 | 114 |

$g(x,y)$

- Neighborhood filtering (convolution): The image on the left is convolved with the filter in the middle to yield the image on the right
- The light blue pixels indicate the source neighborhood for the light green destination pixel:
  $65 \cdot 0.1 + 98 \cdot 0.1 + 123 \cdot 0.1 + 65 \cdot 0.1 + 96 \cdot 0.2 + 115 \cdot 0.1 + 63 \cdot 0.1 + 91 \cdot 0.1 + 107 \cdot 0.1 = 91.9$

## Linear Filtering

- Both correlation and convolution are **linear shift-invariant** (LSI) operators, which obey both the superposition principle

$$h \circ (f_0 + f_1) = h \circ f_0 + h \circ f_1$$

and the **shift invariance** principle

$$g(i,j) = f(i+k, j+l) \Leftrightarrow (h \circ g)(i,j) = (h \circ f)(i+k, j+l)$$

which means that shifting a signal commutes with applying the operator

## Linear Filtering

- Occasionally, a shift-variant version of correlation or convolution may be used, e.g.

$$g(i,j) = \sum_{k,l} f(i-k, j-l)h(k,l;i,j)$$

where $h(k,l;i,j)$ is the convolution kernel at pixel $(i,j)$

## Linear Filtering

- Occasionally, a shift-variant version of correlation or convolution may be used, e.g.

$$g(i,j) = \sum_{k,l} f(i-k, j-l) h(k,l;i,j)$$

where $h(k,l;i,j)$ is the convolution kernel at pixel $(i,j)$

- For example, such a spatially varying kernel can be used to model blur in an image due to variable depth-dependent defocus

## Linear Filtering

- Correlation and convolution can both be written as a matrix-vector multiply by first converting the 2D images $f(i,j)$ and $g(i,j)$ into raster-ordered vectors **f** and **g**

$$\mathbf{g} = H\mathbf{f}$$

where the (sparse) matrix $H$ contains the convolution kernels

## Linear Filtering

$$\begin{array}{|c|c|c|c|c|}\hline 72 & 88 & 62 & 52 & 37 \\ \hline\end{array} * \begin{array}{|c|c|c|}\hline {}^1\!/_4 & {}^1\!/_2 & {}^1\!/_4 \\ \hline\end{array} \Leftrightarrow \frac{1}{4}\begin{bmatrix} 2 & 1 & . & . & . \\ 1 & 2 & 1 & . & . \\ . & 1 & 2 & 1 & . \\ . & . & 1 & 2 & 1 \\ . & . & . & 1 & 2 \end{bmatrix}\begin{bmatrix} 72 \\ 88 \\ 62 \\ 52 \\ 37 \end{bmatrix}$$

- 1D signal convolution as a sparse matrix-vector multiply, $\mathbf{g} = H\mathbf{f}$

## Padding (Border Effects)

- The matrix multiply used in convolution suffers from **boundary effects**, i.e. the results of filtering the image in this form will lead to a *darkening* of the corner pixels

## Padding (Border Effects)

- The matrix multiply used in convolution suffers from **boundary effects**, i.e. the results of filtering the image in this form will lead to a *darkening* of the corner pixels
- This is due to the original image being effectively padded with 0 values wherever the convolution kernel extends beyond the original image boundaries

## Padding (Border Effects)

- To compensate for boundary effects a number of alternative **padding** or extension modes have been developed:

## Padding (Border Effects)

- To compensate for boundary effects a number of alternative **padding** or extension modes have been developed:
  - **zero**: set all pixels outside the source image to 0

## Padding (Border Effects)

- To compensate for boundary effects a number of alternative **padding** or extension modes have been developed:
    - **zero**: set all pixels outside the source image to 0
    - **constant (border color)**: set all pixels outside the source image to a specified border value

## Padding (Border Effects)

- To compensate for boundary effects a number of alternative **padding** or extension modes have been developed:
  - **zero**: set all pixels outside the source image to 0
  - **constant (border color)**: set all pixels outside the source image to a specified border value
  - **clamp (replicate or clamp to edge)**: repeat edge pixels indefinitely

## Padding (Border Effects)

- To compensate for boundary effects a number of alternative **padding** or extension modes have been developed:
  - **zero**: set all pixels outside the source image to 0
  - **constant (border color)**: set all pixels outside the source image to a specified border value
  - **clamp (replicate or clamp to edge)**: repeat edge pixels indefinitely
  - **(cyclic) wrap (repeat or tile)**: loop “around” the image in a “toroidal” configuration

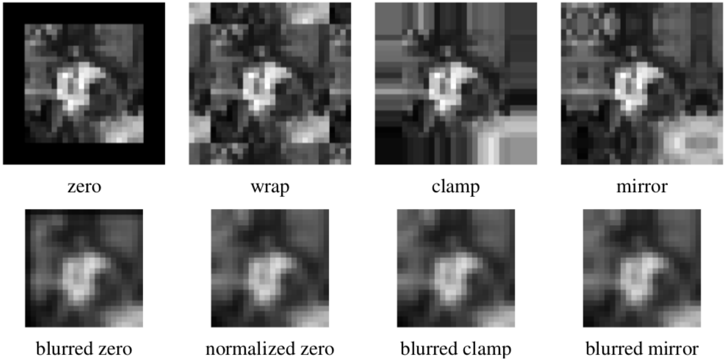
## Padding (Border Effects)

- To compensate for boundary effects a number of alternative **padding** or extension modes have been developed:
  - **zero**: set all pixels outside the source image to 0
  - **constant (border color)**: set all pixels outside the source image to a specified border value
  - **clamp (replicate or clamp to edge)**: repeat edge pixels indefinitely
  - **(cyclic) wrap (repeat or tile)**: loop “around” the image in a “toroidal” configuration
  - **mirror**: reflect pixels across the image edge

## Padding (Border Effects)

- To compensate for boundary effects a number of alternative **padding** or extension modes have been developed:
  - **zero**: set all pixels outside the source image to 0
  - **constant (border color)**: set all pixels outside the source image to a specified border value
  - **clamp (replicate or clamp to edge)**: repeat edge pixels indefinitely
  - **(cyclic) wrap (repeat or tile)**: loop "around" the image in a "toroidal" configuration
  - **mirror**: reflect pixels across the image edge
  - **extend**: extend the signal by subtracting the mirrored version of the signal from the edge pixel value

## Padding (Border Effects)



- Border padding (top row) and the results of blurring the padded image (bottom row)

## Separable Filtering

- The process of performing a convolution requires $K^2$ (multiply- add) operations per pixel, where $K$ is the size (width or height) of the convolution kernel

## Separable Filtering

- The process of performing a convolution requires $K^2$ (multiply- add) operations per pixel, where $K$ is the size (width or height) of the convolution kernel
- In many cases, this operation can be significantly sped up by first performing a 1D horizontal convolution followed by a 1D vertical convolution (which requires a total of $2K$ operations per pixel)

## Separable Filtering

- The process of performing a convolution requires $K^2$ (multiply- add) operations per pixel, where $K$ is the size (width or height) of the convolution kernel
- In many cases, this operation can be significantly sped up by first performing a 1D horizontal convolution followed by a 1D vertical convolution (which requires a total of $2K$ operations per pixel)
- A convolution kernel for which this is possible is said to be **separable**

## Separable Filtering

- The 2D kernel $K$ corresponding to successive convolution with a horizontal kernel $\mathbf{h}$ and a vertical kernel $\mathbf{v}$ is the outer product of the two kernels

$$K = \mathbf{v}\mathbf{h}^T$$

## Separable Filtering

- The 2D kernel $K$ corresponding to successive convolution with a horizontal kernel **h** and a vertical kernel **v** is the outer product of the two kernels

$$K = \mathbf{v}\mathbf{h}^T$$

- The design of convolution kernels for computer vision applications is often influenced by their separability (separation increases efficiency)

## Separable Filtering

- How can we tell if a given kernel $K$ is separable?

## Separable Filtering

- How can we tell if a given kernel $K$ is separable?
- This can be done by inspection or by looking at the analytic form of the kernel, a more direct method is to treat the 2D kernel as a 2D matrix $K$ and take its SVD

$$K = \sum_i \sigma_i \mathbf{u}_i \mathbf{v}_i^T$$

## Separable Filtering

- How can we tell if a given kernel $K$ is separable?
- This can be done by inspection or by looking at the analytic form of the kernel, a more direct method is to treat the 2D kernel as a 2D matrix $K$ and take its SVD

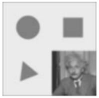
$$K = \sum_i \sigma_i \mathbf{u}_i \mathbf{v}_i^T$$

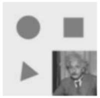
- If only the first singular value $\sigma_0$ is non-zero, then the kernel is separable and $\sqrt{\sigma_0}\mathbf{u}_0$ and $\sqrt{\sigma_0}\mathbf{v}_0^T$ provide the vertical and horizontal kernels

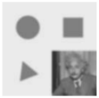
## Separable Filtering

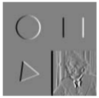
$$\frac{1}{K^2}\begin{array}{|c|c|c|c|}\hline 1 & 1 & \cdots & 1 \\ \hline 1 & 1 & \cdots & 1 \\ \hline \vdots & \vdots & 1 & \vdots \\ \hline 1 & 1 & \cdots & 1 \\ \hline \end{array} \quad \frac{1}{16}\begin{array}{|c|c|c|}\hline 1 & 2 & 1 \\ \hline 2 & 4 & 2 \\ \hline 1 & 2 & 1 \\ \hline \end{array} \quad \frac{1}{256}\begin{array}{|c|c|c|c|c|}\hline 1 & 4 & 6 & 4 & 1 \\ \hline 4 & 16 & 24 & 16 & 4 \\ \hline 6 & 24 & 36 & 24 & 6 \\ \hline 4 & 16 & 24 & 16 & 4 \\ \hline 1 & 4 & 6 & 4 & 1 \\ \hline \end{array} \quad \frac{1}{8}\begin{array}{|c|c|c|}\hline -1 & 0 & 1 \\ \hline -2 & 0 & 2 \\ \hline -1 & 0 & 1 \\ \hline \end{array} \quad \frac{1}{4}\begin{array}{|c|c|c|}\hline 1 & -2 & 1 \\ \hline -2 & 4 & -2 \\ \hline 1 & -2 & 1 \\ \hline \end{array}$$

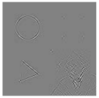
$$\frac{1}{K}\begin{array}{|c|c|c|c|}\hline 1 & 1 & \cdots & 1 \\ \hline \end{array} \quad \frac{1}{4}\begin{array}{|c|c|c|}\hline 1 & 2 & 1 \\ \hline \end{array} \quad \frac{1}{16}\begin{array}{|c|c|c|c|c|}\hline 1 & 4 & 6 & 4 & 1 \\ \hline \end{array} \quad \frac{1}{2}\begin{array}{|c|c|c|}\hline -1 & 0 & 1 \\ \hline \end{array} \quad \frac{1}{2}\begin{array}{|c|c|c|}\hline 1 & -2 & 1 \\ \hline \end{array}$$


(a) box, $K = 5$


(b) bilinear


(c) “Gaussian”


(d) Sobel


(e) corner

- Separable linear filters: top row - 2D filter kernel; middle row - the corresponding horizontal 1D kernel; bottom row - filtered image

## The Box Filter

- The simplest filter to implement is the **moving average** or **box** filter, which simply averages the pixel values in a $K \times K$ window

## The Box Filter

- The simplest filter to implement is the **moving average** or **box** filter, which simply averages the pixel values in a $K \times K$ window
- This is equivalent to convolving the image with a kernel of all ones and then scaling

## The Box Filter

- The simplest filter to implement is the **moving average** or **box** filter, which simply averages the pixel values in a $K \times K$ window
- This is equivalent to convolving the image with a kernel of all ones and then scaling
- For large kernels, a more efficient implementation is to slide a moving window across each scanline (in a separable filter) while adding the newest pixel and subtracting the oldest pixel from the running sum

## Example: Box Filtering using MATLAB

- Read an image into the workspace, perform the mean filtering using an $11 \times 11$ filter, and display the original image and the filtered image:

```
A = imread('cameraman.tif');
localMean = imboxfilt(A,11);
imshowpair(A,localMean,'montage')
```

## The Bilinear Filter

- A smoother image can be obtained by separably convolving the image with a piecewise linear “tent” function (also known as a Bartlett filter)

## The Bilinear Filter

- A smoother image can be obtained by separably convolving the image with a piecewise linear “tent” function (also known as a Bartlett filter)
- The $3 \times 3$ version of this filter is called the **bilinear** filter since it is the product of two linear (first-order) splines

## Smoothing Kernels

- The kernels we have discussed are all examples of blurring (smoothing) or **low-pass** kernels

## Smoothing Kernels

- The kernels we have discussed are all examples of blurring (smoothing) or **low-pass** kernels
- In practice, smoothing kernels are often used to reduce high-frequency noise and they can also be used to *sharpen* images using a process called **unsharp masking**

## Smoothing Kernels

- The kernels we have discussed are all examples of blurring (smoothing) or **low-pass** kernels
- In practice, smoothing kernels are often used to reduce high-frequency noise and they can also be used to *sharpen* images using a process called **unsharp masking**
- Since blurring the image reduces high frequencies, adding some of the difference between the original and blurred image makes it sharper

$$g_{\text{sharp}} = f + \gamma(f - h_{\text{blur}} * f)$$

## Example: Sharpening an Image using MATLAB

- Read an image into the workspace, then sharpen and display:

```
a = imread(‘rice.png’);
imshow(a), title(‘Original Image’);
b = imsharpen(a, ‘Radius’, 2, ‘Amount’, 1);
figure, imshow(b)
title(‘Sharpened Image’);
```

## The Gaussian Kernel Filter

- Convolving the linear tent function with itself yields the cubic approximating spline which is called the **Gaussian** kernel

## The Gaussian Kernel Filter

- Convolving the linear tent function with itself yields the cubic approximating spline which is called the **Gaussian** kernel
- Note that approximate Gaussian kernels can also be obtained by iterated convolution with a box filter

## The Gaussian Kernel Filter

- Convolving the linear tent function with itself yields the cubic approximating spline which is called the **Gaussian** kernel
- Note that approximate Gaussian kernels can also be obtained by iterated convolution with a box filter
- In applications where the filters really need to be rotationally symmetric, carefully tuned versions of sampled Gaussians should be used

## Linear Filtering for Image Preprocessing

- Linear filtering can also be used as a preprocessing stage to edge extraction and interest point detection

## Linear Filtering for Image Preprocessing

- Linear filtering can also be used as a preprocessing stage to edge extraction and interest point detection
- The Sobel operator is a $3 \times 3$ edge extractor which is a separable combination of a horizontal **central difference** (so called because the horizontal derivative is centered on the pixel) and vertical tent filter

## Linear Filtering for Image Preprocessing

- Linear filtering can also be used as a preprocessing stage to edge extraction and interest point detection
- The Sobel operator is a $3 \times 3$ edge extractor which is a separable combination of a horizontal **central difference** (so called because the horizontal derivative is centered on the pixel) and vertical tent filter
- A simple corner detector looks for simultaneous horizontal and vertical second derivatives

## Band-Pass and Steerable Filters

- The Sobel and corner operators are simple examples of band-pass and oriented filters

## Band-Pass and Steerable Filters

- The Sobel and corner operators are simple examples of band-pass and oriented filters
- More sophisticated kernels can be created by first smoothing the image with a (unit area) Gaussian filter

$$G(x, y; \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

and then taking the first or second derivatives

## Band-Pass and Steerable Filters

- The Sobel and corner operators are simple examples of band-pass and oriented filters
- More sophisticated kernels can be created by first smoothing the image with a (unit area) Gaussian filter

$$G(x, y; \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

  and then taking the first or second derivatives
- Such filters are known collectively as **band-pass** filters since they filter out both low and high frequencies

## Band-Pass and Steerable Filters

- The (undirected) second derivative of a 2D image

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 y}{\partial y^2}$$

is known as the **Laplacian** operator

## Band-Pass and Steerable Filters

- The (undirected) second derivative of a 2D image

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 y}{\partial y^2}$$

is known as the **Laplacian** operator

- Blurring an image with a Gaussian and then taking its Laplacian is equivalent to convolving directly with the **Laplacian of Gaussian** (LoG) filter

$$\nabla^2 G(x, y; \sigma) = \left( \frac{x^2 + y^2}{\sigma^4} - \frac{2}{\sigma^2} \right) G(x, y; \sigma)$$

which has desirable **scale-space properties**

## Band-Pass and Steerable Filters

- The Sobel operator is a simple approximation to a **directional** or **oriented** filter

## Band-Pass and Steerable Filters

- The Sobel operator is a simple approximation to a **directional** or **oriented** filter
- It can be obtained with a Gaussian (or some other filter) and then taking a **directional derivative** $\nabla_{\hat{\mathbf{u}}} = \frac{\partial}{\partial \hat{\mathbf{u}}}$

## Band-Pass and Steerable Filters

- The Sobel operator is a simple approximation to a **directional** or **oriented** filter
- It can be obtained with a Gaussian (or some other filter) and then taking a **directional derivative** $\nabla_{\hat{\mathbf{u}}} = \frac{\partial}{\partial \hat{\mathbf{u}}}$
- The directional derivative is computed by taking the dot product between the gradient field $\nabla$ and a unit direction $\hat{\mathbf{u}} = (\cos\theta, \sin\theta)$

$$\hat{\mathbf{u}} \cdot \nabla(G * f) = \nabla_{\hat{\mathbf{u}}}(G * f) = (\nabla_{\hat{\mathbf{u}}} G) * f$$

## Band-Pass and Steerable Filters

- The smoothed directional derivative filter

$$G_{\hat{\mathbf{u}}} = uG_x = vG_y = u\frac{\partial G}{\partial x} + v\frac{\partial G}{\partial y}$$

where $\hat{\mathbf{u}} = (u, v)$

## Band-Pass and Steerable Filters

- The smoothed directional derivative filter

$$G_{\hat{\mathbf{u}}} = uG_x = vG_y = u\frac{\partial G}{\partial x} + v\frac{\partial G}{\partial y}$$

  where $\hat{\mathbf{u}} = (u, v)$

- This is an example of a **steerable** filter since the value of an image convolved with $G_{\hat{\mathbf{u}}}$ can be computed by first convolving with the pair of filters $(G_x, G_y)$ and then **steering** the filter by multiplying this gradient field with a unit vector $\hat{\mathbf{u}}$

## Band-Pass and Steerable Filters

- How about steering a directional second derivative filter $\nabla_{\hat{\mathbf{u}}} \cdot \nabla_{\hat{\mathbf{u}}} G_{\hat{\mathbf{u}}}$, which is the result of taking a (smoothed) directional derivative and then taking the derivative again?

## Band-Pass and Steerable Filters

- How about steering a directional second derivative filter $\nabla_{\hat{\mathbf{u}}} \cdot \nabla_{\hat{\mathbf{u}}} G_{\hat{\mathbf{u}}}$, which is the result of taking a (smoothed) directional derivative and then taking the derivative again?
- It turns out that it is possible to steer *any* order derivative with a relatively small number of basis functions

## Band-Pass and Steerable Filters

- How about steering a directional second derivative filter $\nabla_{\hat{\mathbf{u}}} \cdot \nabla_{\hat{\mathbf{u}}} G_{\hat{\mathbf{u}}}$, which is the result of taking a (smoothed) directional derivative and then taking the derivative again?
- It turns out that it is possible to steer *any* order derivative with a relatively small number of basis functions
- For example, only three basis functions are required for the second-order directional derivative

$$G_{\hat{\mathbf{u}}\hat{\mathbf{u}}} = u^2 G_{xx} + 2uvG_{xy} + v^2 G_{yy}$$

## Band-Pass and Steerable Filters

- This remarkable result makes it possible to construct directional derivative filters of increasing greater *directional selectivity*, i.e. filters that only respond to edges that have strong local consistency in orientation

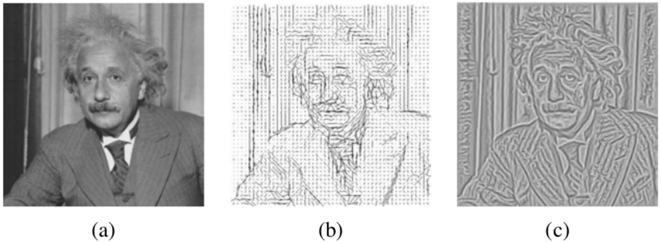
## Band-Pass and Steerable Filters

- This remarkable result makes it possible to construct directional derivative filters of increasing greater *directional selectivity*, i.e. filters that only respond to edges that have strong local consistency in orientation
- Furthermore, higher order steerable filters can respond to potentially more than a single edge orientation at a given location

## Band-Pass and Steerable Filters

- This remarkable result makes it possible to construct directional derivative filters of increasing greater *directional selectivity*, i.e. filters that only respond to edges that have strong local consistency in orientation
- Furthermore, higher order steerable filters can respond to potentially more than a single edge orientation at a given location
- Steerable filters are often used to construct both feature descriptors and edge detectors

## Band-Pass and Steerable Filters



- Second-order steerable filter: (a) original image; (b) orientation map computed from the second-order oriented energy; (c) original image with oriented structures

## Example: Smoothing an Image with a Gaussian Filter using MATLAB

- Read an image to be filtered, filter it with a Gaussian filter, and display the original and filtered image:

```
I = imread('cameraman.tif');
Iblur = imgaussfilt(I,2);
montage({I,Iblur})
title('Original Image (Left) Vs.  Gaussian
Filtered Image (Right)')
```

## Non-Linear Filtering

- The filters we have considered so far have all been *linear*, i.e. their response to a sum of two signals is the same as the sum of the individual responses

## Non-Linear Filtering

- The filters we have considered so far have all been *linear*, i.e. their response to a sum of two signals is the same as the sum of the individual responses
- This is equivalent to saying that each output pixel is a weighted summation of some number of input pixels

## Non-Linear Filtering

- The filters we have considered so far have all been *linear*, i.e. their response to a sum of two signals is the same as the sum of the individual responses
- This is equivalent to saying that each output pixel is a weighted summation of some number of input pixels
- Linear filters are easier to compose and are amenable to frequency response analysis

## Non-Linear Filtering

- In many cases, better performance can be achieved by using a **non-linear** combination of neighboring pixels

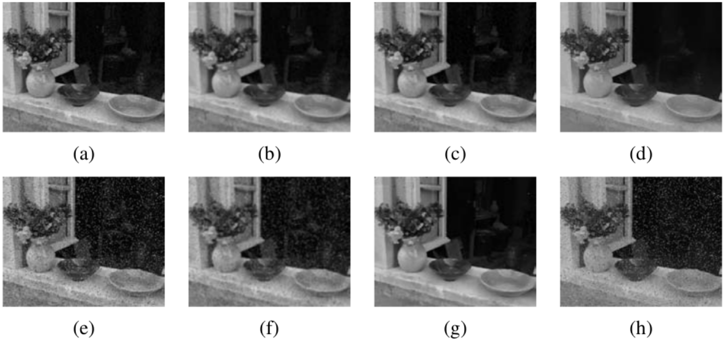
## Non-Linear Filtering

- In many cases, better performance can be achieved by using a **non-linear** combination of neighboring pixels
- For example, consider an image where the noise, rather than being Gaussian, is **shot noise**, i.e. it occasionally has very large values

## Non-Linear Filtering

- In many cases, better performance can be achieved by using a **non-linear** combination of neighboring pixels
- For example, consider an image where the noise, rather than being Gaussian, is **shot noise**, i.e. it occasionally has very large values
- In this case, regular blurring with a Gaussian filter fails to remove the noisy pixels and instead turns them into softer (but still visible) spots

## Non-Linear Filtering



- Median and bilateral filtering: (a) original image with Gaussian noise; (b) Gaussian filtered; (c) median filtered; (d) bilaterally filtered; (e) original image with shot noise; (f) Gaussian filtered; (g) median filtered; (h) bilaterally filtered

## Median Filtering

- The **median** filter selects the median value from each pixel's neighborhood

## Median Filtering

- The **median** filter selects the median value from each pixel's neighborhood
- Median values can be computed in linear time using a randomized select algorithm

## Median Filtering

- The **median** filter selects the median value from each pixel's neighborhood
- Median values can be computed in linear time using a randomized select algorithm
- Since the shot noise value usually lies well outside the true values in the neighborhood, the median filter is able to filter away bad pixels

## Median Filtering

- One downside of the median filter, in addition to its moderate computational cost, is that since it selects only one input pixel value to replace each output pixel, it is not as *efficient* at averaging away Gaussian noise

## Median Filtering

- One downside of the median filter, in addition to its moderate computational cost, is that since it selects only one input pixel value to replace each output pixel, it is not as *efficient* at averaging away Gaussian noise
- A better choice may be the $\alpha$-trimmed mean, which averages together all of the pixels except for the $\alpha$ fraction that are the smallest and the largest

## Median Filtering

- Another possibility is to compute a *weighted median*, in which each pixel is used a number of times depending on its distance from the center

## Median Filtering

- Another possibility is to compute a *weighted median*, in which each pixel is used a number of times depending on its distance from the center
- This turns out to be equivalent to minimizing the weighted objective function

$$\sum_{k,l} w(k,l)|f(i+k,j+l)-g(i,j)|^p$$

where $g(i,j)$ is the desired output value and $p = 1$ for the weighed median

## Example: Median Filtering of an Image using MATLAB

- Remove salt and pepper noise from an image:

```
I = imread('eight.tif');
figure, imshow(I)
J = imnoise(I,'salt & pepper', 0.02);
K = medfilt2(J);
imshowpair(J,K,'montage')
```

## Bilateral Filtering

- The essential idea of **bilateral** filtering is to combine a weighted filter kernel with a better version of outlier rejection

## Bilateral Filtering

- The essential idea of **bilateral** filtering is to combine a weighted filter kernel with a better version of outlier rejection
- In the bilateral filter, the output pixel value depends on a weighted combination of neighboring pixel values

$$g(i,j) = \frac{\sum_{k,l} f(k,l)w(i,j,k,l)}{\sum_{k,l} w(i,j,k,l)}$$

## Bilateral Filtering

- The weighting coefficient $w(i,j,k,l)$ depends on the product of a *domain kernel*

$$d(i,j,k,l) = \exp\left(-\frac{(i-k)^2 + (j-l)^2}{2\sigma_d^2}\right)$$

and a data-dependent *range kernel*

$$r(i,j,k,l) = \exp\left(-\frac{||f(i,j) - f(k,l)||^2}{2\sigma_r^2}\right)$$

## Bilateral Filtering

- When multiplied together, these yield the data-dependent *bilateral weight function*

$$w(i,j,k,l) = \exp\left(-\frac{(i-k)^2+(j-l)^2}{2\sigma_d^2} - \frac{||f(i,j)-f(k,l)||^2}{2\sigma_r^2}\right)$$

## Bilateral Filtering

- When multiplied together, these yield the data-dependent *bilateral weight function*

$$w(i,j,k,l) = \exp\left( -\frac{(i-k)^2 + (j-l)^2}{2\sigma_d^2} - \frac{\|f(i,j) - f(k,l)\|^2}{2\sigma_r^2} \right)$$

- Note that the range filter uses the *vector distance* between the center and the neighboring pixel

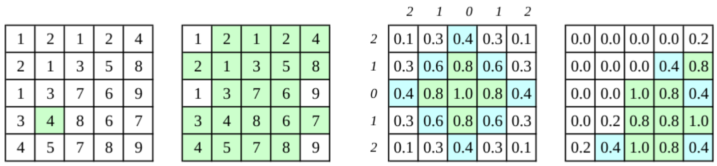
## Bilateral Filtering

- When multiplied together, these yield the data-dependent *bilateral weight function*

$$w(i,j,k,l) = \exp\left( - \frac{(i-k)^2 + (j-l)^2}{2\sigma_d^2} - \frac{||f(i,j) - f(k,l)||^2}{2\sigma_r^2} \right)$$

- Note that the range filter uses the *vector distance* between the center and the neighboring pixel
- This is important in color images since an edge in any *one* of the color bands signals a change in material and thus the need to downweight a pixel's influence

## Median and Bilateral Filtering



| | | | | |
|---|---|---|---|---|
| 1 | 2 | 1 | 2 | 4 |
| 2 | 1 | 3 | 5 | 8 |
| 1 | 3 | 7 | 6 | 9 |
| 3 | 4 | 8 | 6 | 7 |
| 4 | 5 | 7 | 8 | 9 |

(a) median = 4

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 1 | 2 | 4 |
| 2 | 1 | 3 | 5 | 8 |
| 1 | 3 | 7 | 6 | 9 |
| 3 | 4 | 8 | 6 | 7 |
| 4 | 5 | 7 | 8 | 9 |

(b) α-mean= 4.6

| | 2 | 1 | 0 | 1 | 2 |
|---|---|---|---|---|---|
| 2 | 0.1 | 0.3 | 0.4 | 0.3 | 0.1 |
| 1 | 0.3 | 0.6 | 0.8 | 0.6 | 0.3 |
| 0 | 0.4 | 0.8 | 1.0 | 0.8 | 0.4 |
| 1 | 0.3 | 0.6 | 0.8 | 0.6 | 0.3 |
| 2 | 0.1 | 0.3 | 0.4 | 0.3 | 0.1 |

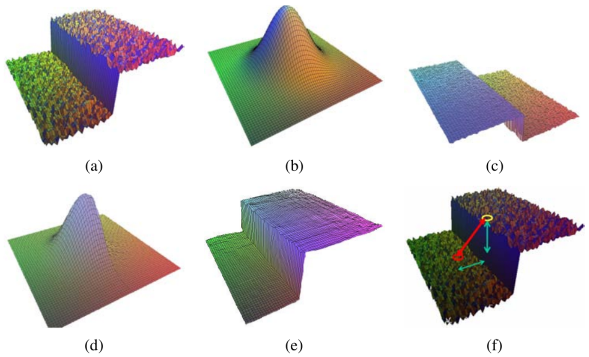
(c) domain filter

| | | | | |
|---|---|---|---|---|
| 0.0 | 0.0 | 0.0 | 0.0 | 0.2 |
| 0.0 | 0.0 | 0.0 | 0.4 | 0.8 |
| 0.0 | 0.0 | 1.0 | 0.8 | 0.4 |
| 0.0 | 0.2 | 0.8 | 0.8 | 1.0 |
| 0.2 | 0.4 | 1.0 | 0.8 | 0.4 |

(d) range filter

- Median and bilateral filtering: (a) median pixel (green); (b) selected $\alpha$-trimmed mean pixels; (c) domain filter (numbers along edge are pixel distances; (d) range filter

## Bilateral Filtering



- Bilateral filtering: (a) noisy step edge input; (b) domain filter (Gaussian); (c) range filter; (d) bilateral filter; (e) filtered step edge output; (f) 3D distance between pixels

## Morphology

- While non-linear filters are often used to enhance grayscale and color images, they are also used extensively to process binary images

## Morphology

- While non-linear filters are often used to enhance grayscale and color images, they are also used extensively to process binary images
- Such images often occur after a *thresholding* operation

$$\theta(f, t) = \begin{cases} 1 & \text{if } f \geq t, \\ 0 & \text{else} \end{cases}$$

such as converting a scanned grayscale document into a binary image for further processing (e.g. optical character recognition)

## Morphology

- The most common binary image operations are called **morphological operations** since they change the *shape* of the underlying binary objects

## Morphology

- The most common binary image operations are called **morphological operations** since they change the *shape* of the underlying binary objects
- To perform such an operation, we first convolve the binary image with a binary **structuring element**

## Morphology

- The most common binary image operations are called **morphological operations** since they change the *shape* of the underlying binary objects
- To perform such an operation, we first convolve the binary image with a binary **structuring element**
- Then, we select a binary output value depending on the thresholded result of the convolution

## Morphology

- The structuring element can be any shape, from a simple $3 \times 3$ box filter, to more complicated disc structures, it can even correspond to a particular shape that is being sought for in the image

## Morphology

- The structuring element can be any shape, from a simple $3 \times 3$ box filter, to more complicated disc structures, it can even correspond to a particular shape that is being sought for in the image
- Let $f$ be a binary image with a $3 \times 3$ structuring element $s$, then

$$c = f \otimes s$$

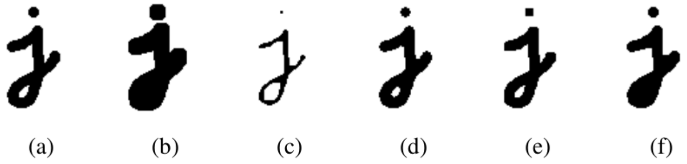
is the integer-value count of the number of 1s inside each structuring element as it is scanned over the image

## Morphology

- The standard operations used in binary morphology include:
  - **dilation**: $\text{dilate}(f, s) = \theta(c, 1)$
  - **erosion**: $\text{erode}(f, s) = \theta(c, S)$
  - **majority**: $\text{maj}(f, s) = \theta(c, S/2)$
  - **opening**: $\text{open}(f, s) = \text{dilate}(\text{erode}(f, s), s)$
  - **closing**: $\text{close}(f, s) = \text{erode}(\text{dilate}(f, s), s)$

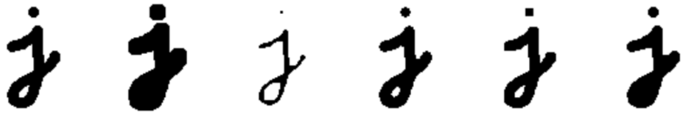
  where $S$ is the size of the structuring element (number of pixels)

## Morphology



- Binary image morphology: (a) original image; (b) dilation; (c) erosion; (d) majority; (e) opening; (f) closing

## Morphology



- Binary image morphology: (a) original image; (b) dilation; (c) erosion; (d) majority; (e) opening; (f) closing
- The structuring element for all examples is a $5 \times 5$ square

## Morphology

- The structuring element can be any shape, from a simple $3 \times 3$ box filter, to more complicated disc structures, it can even correspond to a particular shape that is being sought for in the image

## Morphology

- The structuring element can be any shape, from a simple $3 \times 3$ box filter, to more complicated disc structures, it can even correspond to a particular shape that is being sought for in the image
- Let $f$ be a binary image with a $3 \times 3$ structuring element $s$, then

$$c = f \otimes s$$

is the integer-value count of the number of 1s inside each structuring element as it is scanned over the image

## Distance Transforms

- The **distance transform** is useful in quickly precomputing the distance to a curve or set of points using a two-pass raster algorithm

## Distance Transforms

- The **distance transform** is useful in quickly precomputing the distance to a curve or set of points using a two-pass raster algorithm
- It has many applications including level sets, fast chamfer matching (binary image alignment), feathering in image stitching and blending, and nearest point alignment

## Distance Transforms

- Let $d(k, l)$ be some distance metric between pixel offsets, two commonly used metrics include the **city block** or **Manhattan** distance

$$d_1(k, l) = |k| + |l|$$

and the **Euclidean** distance

$$d_2(k, l) = \sqrt{k^2 + l^2}$$

## Distance Transforms

- Let $d(k, l)$ be some distance metric between pixel offsets, two commonly used metrics include the **city block** or **Manhattan** distance

$$d_1(k, l) = |k| + |l|$$

  and the **Euclidean** distance

$$d_2(k, l) = \sqrt{k^2 + l^2}$$

- The distance transform of a binary image $b(i, j)$ is then defined as

$$D(i, j) = \min_{k,l:b(k,l)=0} d(i - k, j - l)$$

  i.e. it is the distance to the *nearest* background pixel whose value is 0

## Distance Transforms

- The $D_1$ city block distance transform can be efficiently computed using a forward and backward pass of a simple raster-scan algorithm

## Distance Transforms

- The $D_1$ city block distance transform can be efficiently computed using a forward and backward pass of a simple raster-scan algorithm
- During the forward pass, each non-zero pixel in $b$ is replaced by the minimum of $1 +$ the distance of its north or west neighbor

## Distance Transforms

- The $D_1$ city block distance transform can be efficiently computed using a forward and backward pass of a simple raster-scan algorithm
- During the forward pass, each non-zero pixel in $b$ is replaced by the minimum of $1 +$ the distance of its north or west neighbor
- During the backward pass, the same occurs except the minimum is both over the current value $D$ and $1 +$ distance of the south and east neighbors

## Distance Transforms

- Efficiently computing the Euclidean distance transform is more complicated, just keeping the minimum scalar distance to the boundary is not sufficient

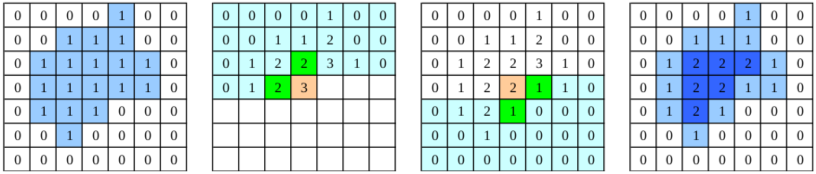
## Distance Transforms

- Efficiently computing the Euclidean distance transform is more complicated, just keeping the minimum scalar distance to the boundary is not sufficient
- Instead, a *vector-valued* distance consisting of both the $x$ and $y$ coordinates of the distance to the boundary must be kept and compared using the squared distance (hypotenuse) rule

## Distance Transforms

- Efficiently computing the Euclidean distance transform is more complicated, just keeping the minimum scalar distance to the boundary is not sufficient
- Instead, a *vector-valued* distance consisting of both the $x$ and $y$ coordinates of the distance to the boundary must be kept and compared using the squared distance (hypotenuse) rule
- Larger search regions need to be used to obtain reasonable results as well

## Distance Transforms



(a)

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

(b)

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 2 | 0 | 0 |
| 0 | 1 | 2 | 2 | 3 | 1 | 0 |
| 0 | 1 | 2 | 3 | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |

(c)

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 2 | 0 | 0 |
| 0 | 1 | 2 | 2 | 3 | 1 | 0 |
| 0 | 1 | 2 | 2 | 1 | 1 | 0 |
| 0 | 1 | 2 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

(d)

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 2 | 2 | 2 | 1 | 0 |
| 0 | 1 | 2 | 2 | 1 | 1 | 0 |
| 0 | 1 | 2 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

- City block distance transform: (a) original binary image; (b) top to bottom (forward) raster sweep: green values are used to compute the orange value; (c) bottom to top (backward) raster sweep: green values are merged with old orange values; (d) final distance transform

## Connected Components

- Another useful semi-global image operation is finding the **connected components**, i.e. regions of adjacent pixels that have the same input value (or label)

## Connected Components

- Another useful semi-global image operation is finding the **connected components**, i.e. regions of adjacent pixels that have the same input value (or label)
- Connected components can be used in a variety of applications such as finding objects in an thresholded image and computing their area statistics

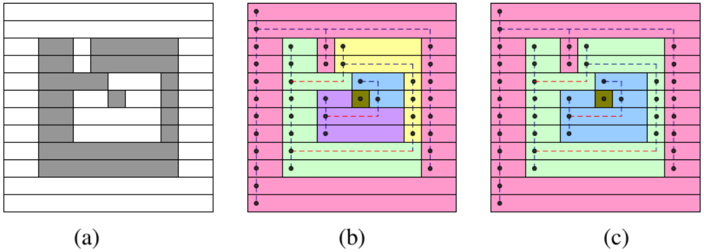
## Connected Components

- To compute the connected components of an image we first split the image into horizontal runs of adjacent pixels, and then color the runs with unique labels, reusing the labels of vertically adjacent runs whenever possible

## Connected Components

- To compute the connected components of an image we first split the image into horizontal runs of adjacent pixels, and then color the runs with unique labels, reusing the labels of vertically adjacent runs whenever possible
- In the second phase, adjacent runs of different colors are then merged

## Connected Components



- Binary image morphology: (a) original image; (b) dilation; (c) erosion;

## Connected Components

- Once a binary or multi-valued image has been segmented into its connected components, it is often useful to compute the area statistics for each individual region $\mathcal{R}$

## Connected Components

- Once a binary or multi-valued image has been segmented into its connected components, it is often useful to compute the area statistics for each individual region $\mathcal{R}$
- Such statistics include:

## Connected Components

- Once a binary or multi-valued image has been segmented into its connected components, it is often useful to compute the area statistics for each individual region $\mathcal{R}$
- Such statistics include:
  - Area (number of pixels)

## Connected Components

- Once a binary or multi-valued image has been segmented into its connected components, it is often useful to compute the area statistics for each individual region $\mathcal{R}$
- Such statistics include:
  - Area (number of pixels)
  - Perimeter (number of boundary pixels)

## Connected Components

- Once a binary or multi-valued image has been segmented into its connected components, it is often useful to compute the area statistics for each individual region $\mathcal{R}$
- Such statistics include:
  - Area (number of pixels)
  - Perimeter (number of boundary pixels)
  - Centroid (average $x$ and $y$ values)

## Connected Components

- Once a binary or multi-valued image has been segmented into its connected components, it is often useful to compute the area statistics for each individual region $\mathcal{R}$
- Such statistics include:
  - Area (number of pixels)
  - Perimeter (number of boundary pixels)
  - Centroid (average $x$ and $y$ values)
  - Second moments

$$M = \sum_{(x,y)\in\mathcal{R}} \begin{bmatrix} x - \bar{x} \\ y - \bar{y} \end{bmatrix} \begin{bmatrix} x - \bar{x} & y - \bar{y} \end{bmatrix}$$

  from which the major and minor axis orientation and lengths can be computed using eigenvalue analysis

## Changing Image Resolution

- Sometimes it's necessary to change the resolution of an image before proceeding

## Changing Image Resolution

- Sometimes it's necessary to change the resolution of an image before proceeding
- For example, we may need we may need to interpolate a small image to makes its resolution match that of the output computer screen

## Changing Image Resolution

- Sometimes it's necessary to change the resolution of an image before proceeding
- For example, we may need we may need to interpolate a small image to makes its resolution match that of the output computer screen
- Alternatively, we may want to reduce the size of an image to speed up the execution of an algorithm or to save on storage space or transmission time

## Changing Image Resolution

- It's often the case that we do not know what the appropriate resolution should be

## Changing Image Resolution

- It's often the case that we do not know what the appropriate resolution should be
- Consider the task of finding a face in a image, since we don't know the scale at which the face will appear we need to generate a whole **pyramid** of differently sized images and scan each one

## Changing Image Resolution

- Such a pyramid can also be very helpful in accelerating the search for an object by first finding a smaller instance of that object at a coarser lever of the pyramid, and then looking for the full resolution object only in the vicinity of of coarse-level detections

## Changing Image Resolution

- Such a pyramid can also be very helpful in accelerating the search for an object by first finding a smaller instance of that object at a coarser lever of the pyramid, and then looking for the full resolution object only in the vicinity of of coarse-level detections
- Image pyramids are extremely useful for performing multi-scale editing operations such as blending images while maintaining details

## Interpolation

- In order to **interpolate** (or **upscale**) an image to a higher resolution, we need to select some interpolation kernel with which to convolve the image

$$g(i,j) = \sum_{k,l} f(k,l)h(i-rk, j-rl)$$

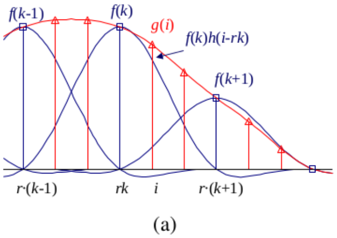
## Interpolation

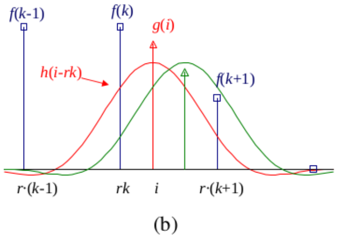
- In order to **interpolate** (or **upscale**) an image to a higher resolution, we need to select some interpolation kernel with which to convolve the image

$$g(i,j) = \sum_{k,l} f(k,l)h(i-rk, j-rl)$$

- This equation is related to the discrete convolution equation, except that we replace $k$ and $l$ in $h()$ with $rk$ and $rl$ where $r$ is the sampling rate

## Interpolation





- Signal interpolation, $g(i) = \sum_k f(k)h(i - rk)$: (a) weighted summation of input values; (b) polyphase filter interpretation

## Interpolation

- What kinds of kernels make good interpolators?

## Interpolation

- What kinds of kernels make good interpolators?
- The answer depends on the application and the computation time involved

## Interpolation

- What kinds of kernels make good interpolators?
- The answer depends on the application and the computation time involved
- **Splines** (piecewise-polynomials) have long been used for function and data value interpolation due to their ability to precisely specify derivatives at control points and efficient incremental algorithms for their evaluation

## Interpolation

- What kinds of kernels make good interpolators?
- The answer depends on the application and the computation time involved
- **Splines** (piecewise-polynomials) have long been used for function and data value interpolation due to their ability to precisely specify derivatives at control points and efficient incremental algorithms for their evaluation
- Many image processing operations can be performed by representing images as splines and manipulating them in a multi-resolution framework

## Decimation

- While interpolation is used to increase the resolution of an image, **decimation** (**downsampling**) is required to reduce the resolution

## Decimation

- While interpolation is used to increase the resolution of an image, **decimation** (**downsampling**) is required to reduce the resolution
- To perform decimation, we first (conceptually) convolve the image with a low-pass filter (to avoid aliasing) and then keep every $r$th sample

## Decimation

- In practice, we usually only evaluate the convolution at every *r*th sample

$$g(i,j) = \sum_{k,l} f(k,l)h(ri-k, rj-l)$$

## Decimation

- In practice, we usually only evaluate the convolution at every $r$th sample

$$g(i,j) = \sum_{k,l} f(k,l)h(ri-k, rj-l)$$

- Note that the smoothing kernel $h(k,l)$, in this case, is often a stretched and re-scaled version of the interpolation kernel

## Decimation

- In practice, we usually only evaluate the convolution at every *r*th sample

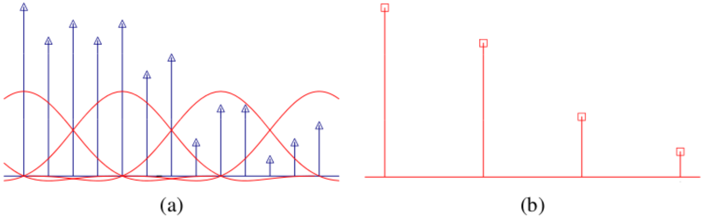
$$g(i,j) = \sum_{k,l} f(k,l)h(ri-k, rj-l)$$

- Note that the smoothing kernel $h(k,l)$, in this case, is often a stretched and re-scaled version of the interpolation kernel
- Alternatively, we can write

$$g(i,j) = \frac{1}{r}\sum_{k,l} f(k,l)h(i-k/r, j-l/r)$$

and keep the same kernel $h(k,l)$ for both interpolation and decimation

## Decimation



- Signal decimation: (a) the original samples are (b) convolved with a low-pass filter before being downsampled

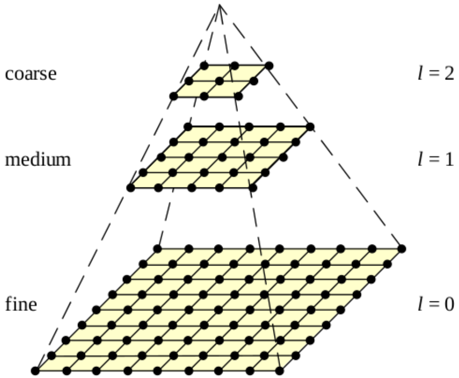
## Multi-Resolution Representations

- With interpolation and decimation algorithms we can build a complete image pyramid

## Multi-Resolution Representations

- With interpolation and decimation algorithms we can build a complete image pyramid
- Pyramids can be used to accelerate coarse-to-fine search algorithms, to look for objects or patterns at different scales, and to perform multi-resolution blending operations

## Multi-Resolution Representations



- A traditional image pyramid: each level has half the resolution (width and height), and thus a quarter of the pixels of its parent level

## Multi-Resolution Representations

- The best known pyramid in computer vision is the **Laplacian pyramid**

## Multi-Resolution Representations

- The best known pyramid in computer vision is the **Laplacian pyramid**
- To construct the pyramid, we first blur and subsample the original image by a factor of two and store this in the next level of the pyramid

## Multi-Resolution Representations

- The best known pyramid in computer vision is the **Laplacian pyramid**
- To construct the pyramid, we first blur and subsample the original image by a factor of two and store this in the next level of the pyramid
- Since adjacent levels in the pyramid are related by a sampling rate $r = 2$, this kind of pyramid is known as an **octave pyramid**

## Multi-Resolution Representations

- The Laplacian pyramid makes use of a five-tap kernel of the form

| | | | | |
|---|---|---|---|---|
| $c$ | $b$ | $a$ | $b$ | $c$ |

with $b = 1/4$ and $c = 1/4 - a/2$

## Multi-Resolution Representations

- The Laplacian pyramid makes use of a five-tap kernel of the form

| | | | | |
|---|---|---|---|---|
| $c$ | $b$ | $a$ | $b$ | $c$ |

  with $b = 1/4$ and $c = 1/4 - a/2$

- In practice, $a = 3/8$, which results in the binomial kernel

$$\frac{1}{16}\boxed{\begin{array}{c|c|c|c|c} 1 & 4 & 6 & 4 & 1 \end{array}}$$

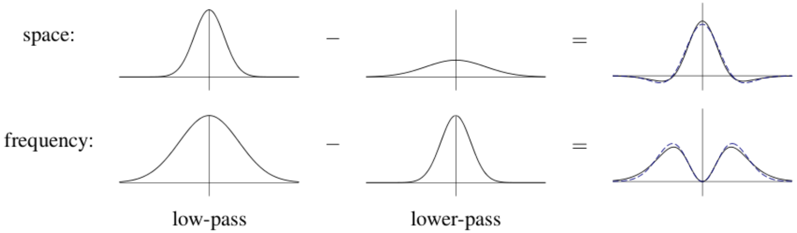
  and is particularly easy to compute using shifts and adds

## Multi-Resolution Representations

- The term Laplacian is a bit of a misnomer since their band-pass images are really differences of (approximate) Gaussians, or DoGs

$$\mathrm{DoG}\{I;\sigma_1,\sigma_2\} = G_{\sigma_1} * I - G_{\sigma_2} * I = (G_{\sigma_1} - G_{\sigma_2}) * I$$

## Multi-Resolution Representations



- DoG in both space and frequency: the difference of two low-pass filters results in a band-pass filter

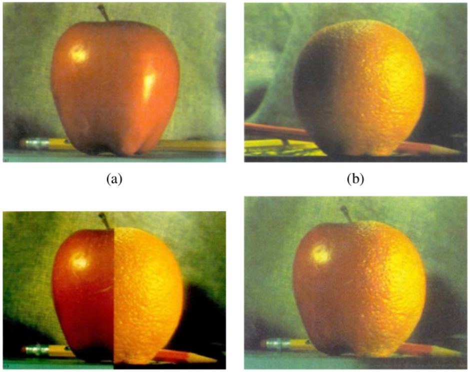
## Application: Image Blending

- An interesting and fun application of the Laplacian pyramid is the creation of blended composite images

## Application: Image Blending

- An interesting and fun application of the Laplacian pyramid is the creation of blended composite images
- For example, consider splicing together images of an apple and orange, low-frequency color variations are smoothly blended while higher-frequency textures are blended more quickly to avoid “ghosting” effects when the two textures are overlaid

## Application: Image Blending



- Laplacian pyramid blending: (a) original image of apple, (b) original image of orange, (c) regular splice, (d) pyramid blend

## Image Blending Algorithm

1. Build Laplacian pyramids $L_A$ and $L_B$ from input images $A$ and $B$

## Image Blending Algorithm

1. Build Laplacian pyramids $L_A$ and $L_B$ from input images $A$ and $B$
2. Build a Gaussian pyramid $G_R$ from the selected binary region $R$

## Image Blending Algorithm

1. Build Laplacian pyramids $L_A$ and $L_B$ from input images $A$ and $B$
2. Build a Gaussian pyramid $G_R$ from the selected binary region $R$
3. Form a combined pyramid $L_S$ from $L_A$ and $L_B$ using the nodes of $G_R$ as weights:

$$L_{S_l}(i,j) = G_{R_l}(i,j)L_{A_l}(i,j) + (1 - G_{R_l}(i,j))L_{B_l}(i,j)$$

## Image Blending Algorithm

1. Build Laplacian pyramids $L_A$ and $L_B$ from input images $A$ and $B$
2. Build a Gaussian pyramid $G_R$ from the selected binary region $R$
3. Form a combined pyramid $L_S$ from $L_A$ and $L_B$ using the nodes of $G_R$ as weights:

$$L_{S_l}(i,j) = G_{R_l}(i,j)L_{A_l}(i,j) + (1 - G_{R_l}(i,j))L_{B_l}(i,j)$$

4. Collapse the $L_S$ pyramid to get the final blended image