Digital Halftoning

Prof. George Wolberg
Dept. of Computer Science
City College of New York

Objectives

- In this lecture we review digital halftoning techniques to convert grayscale images to bitmaps:
  - Unordered (random) dithering
  - Ordered dithering
  - Patterning
  - Error diffusion

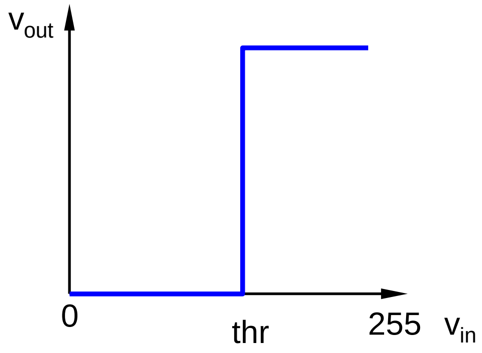
Background

- An 8-bit grayscale image allows 256 distinct gray levels.
- Such images can be displayed on a computer monitor if the hardware supports the required number of intensity levels.
- However, some output devices print or display images with much fewer gray levels.
- In these cases, the grayscale images must be converted to binary images, where pixels are only black (0) or white (255).
- Thresholding is a poor choice due to objectionable artifacts.
- Strategy: sprinkle black-and-white dots to simulate gray.
- Exploit spatial integration (averaging) performed by eye.

Thresholding

- The simplest way to convert from grayscale to binary.

8 bpp (256 levels)



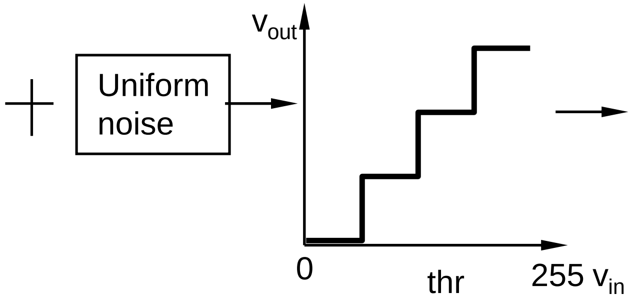
1 bpp (two-level)

Loss of information is unacceptable.

Unordered Dither (1)

- Reduce quantization error by adding uniformly distributed white noise (dither signal) to the input image prior to quantization.
- Dither hides objectional artifacts.
- To each pixel of the image, add a random number in the range [-*m*, *m*], where *m* is MXGRAY/quantization-levels.

8 bpp (256 levels)



3 bpp (8 levels)

Unordered Dither (2)

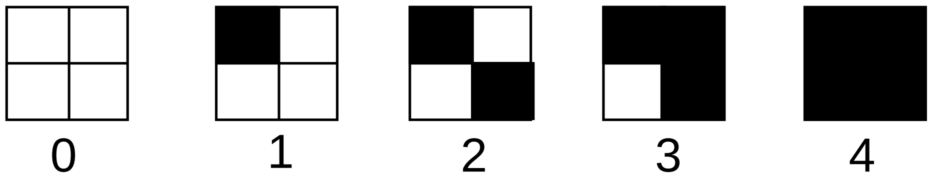
1 bpp | 2 bpp | 3 bpp | 4 bpp

Quantization

Dither/
Quantization

Ordered Dithering

- Objective: expand the range of available intensities.
- Simulates n bpp images with m bpp, where n>m (usually m = 1).
- Exploit eye's spatial integration.
  - Gray is due to average of black/white dot patterns.
  - Each dot is a circle of black ink whose area is proportional to (1 – intensity).
  - Graphics output devices approximate the variable circles of halftone reproductions.



- 2 x 2 pixel area of a bilevel display produces 5 intensity levels.
- *n x n* group of bilevel pixels produces n^2+1 intensity levels.
- Tradeoff: spatial vs. intensity resolution.

Dither Matrix (1)

- Consider the following 2x2 and 3x3 dither matrices:

$$D^{(2)} = \begin{bmatrix} 0 & 2 \\ 3 & 1 \end{bmatrix} \qquad D^{(3)} = \begin{bmatrix} 6 & 8 & 4 \\ 1 & 0 & 3 \\ 5 & 2 & 7 \end{bmatrix}$$

- To display a pixel of intensity *I*, we turn on all pixels whose associated dither matrix values are less than *I.*
- The recurrence relation given below generates larger dither matrices of dimension *n* x *n*, where *n* is a power of 2.

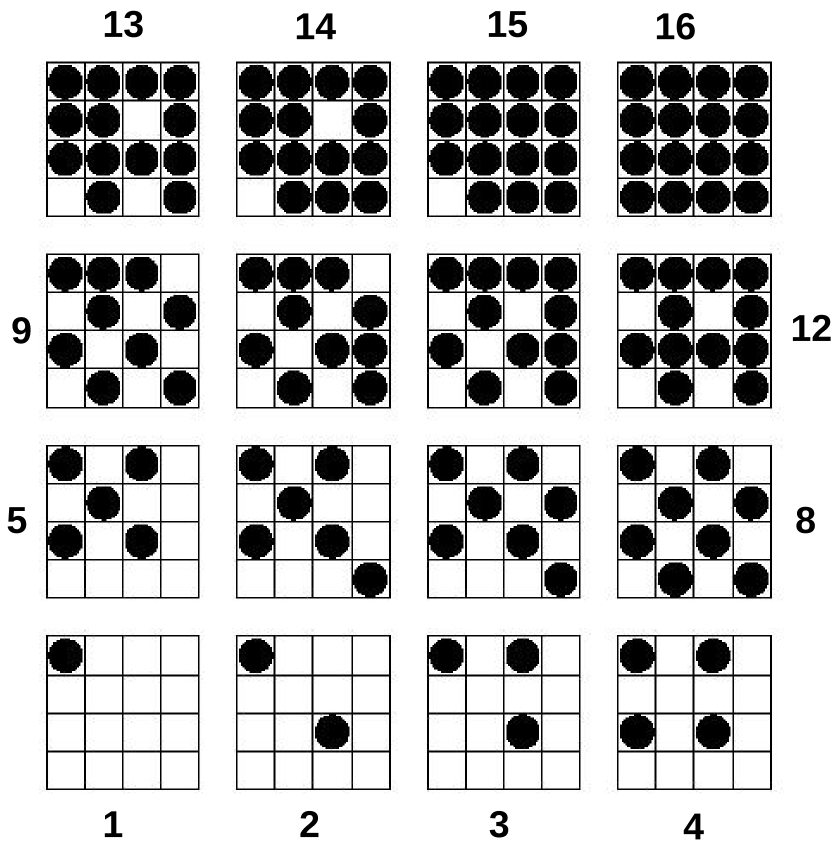
$$D^{(n)} = \begin{bmatrix} 4D^{(n/2)} + D_{00}^{(2)}U^{(n/2)} & 4D^{(n/2)} + D_{01}^{(2)}U^{(n/2)} \\ 4D^{(n/2)} + D_{10}^{(2)}U^{(n/2)} & 4D^{(n/2)} + D_{11}^{(2)}U^{(n/2)} \end{bmatrix}$$

where $U^{(n)}$ is an *n* x *n* matrix of 1's.

Dither Matrix (2)

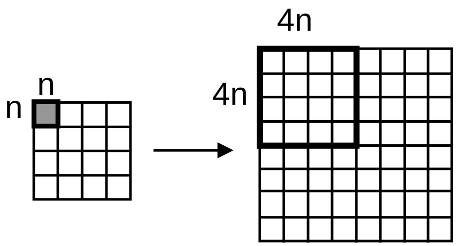
- Example: a 4x4 dither matrix can be derived from the 2x2 matrix.

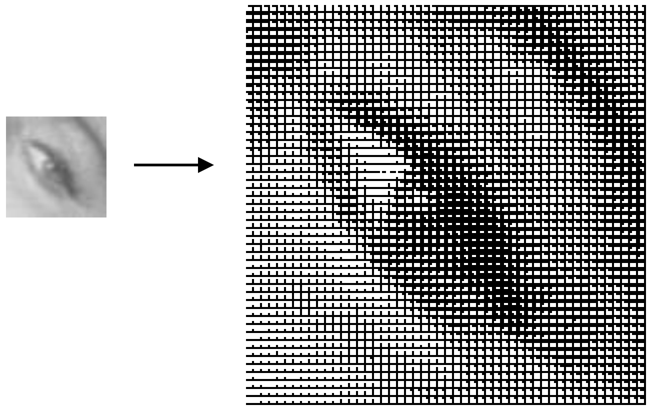
$$D^{(4)} = \begin{bmatrix} 0 & 8 & 2 & 10 \\ 12 & 4 & 14 & 6 \\ 3 & 11 & 1 & 9 \\ 15 & 7 & 13 & 5 \end{bmatrix}$$



Patterning

- Let the output image be larger than the input image.
- Quantize the input image to [0…n^2] gray levels.
- Threshold each pixel against all entries in the dither matrix.
  - Each pixel forms a 4x4 block of black-and-white dots for a $D^{(4)}$ matrix.
  - An $n \times n$ input image becomes a $4n \times 4n$ output image.
- Multiple display pixels per input pixel.
- The dither matrix $D_{ij}^{(n)}$ is used as a spatially-varying threshold.
- Large input areas of constant value are displayed exactly as before.





Implementation

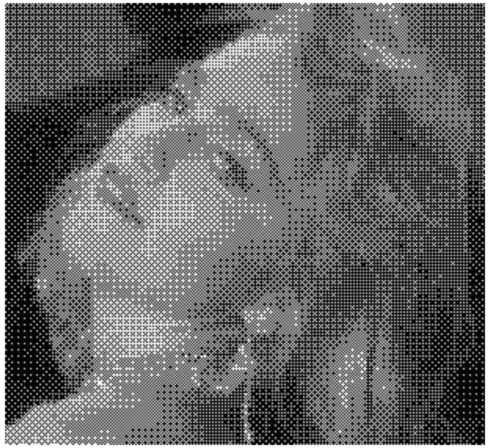
- Let the input and output images share the same size.
- First quantize the input image to [0…n^2] gray levels.
- Compare the dither matrix with the input image.

```
for(y=0; y<h; y++)          // visit all input rows
 for(x=0; x<w; x++){        // visit all input cols
      i = x % n;            // dither matrix index
      j = y % n;            // dither matrix index

      // threshold pixel using dither value Dij(n)
       out[y*w+x] = (in[y*w+x] > Dij(n))? 255 : 0;
 }
```

Examples

8 bpp (256 levels)

1 bpp (D^3)



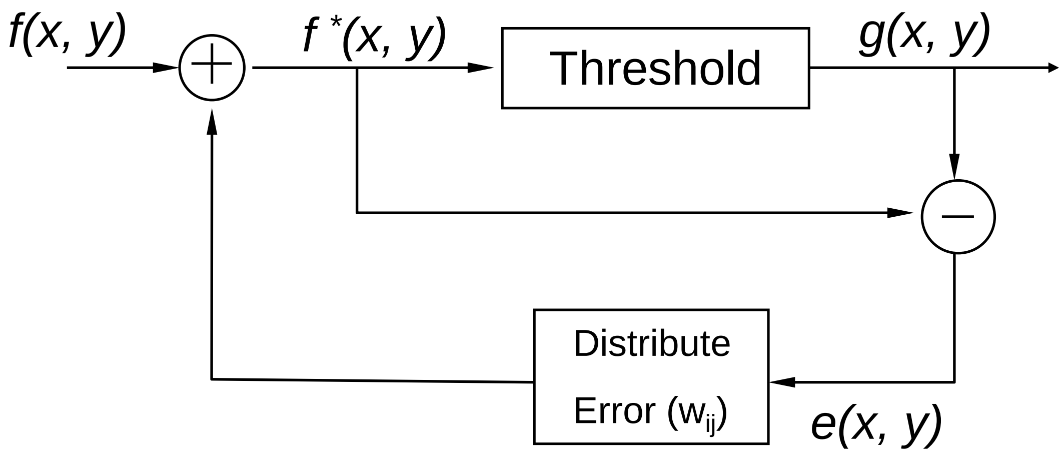
1 bpp (D^4)

1 bpp (D^8)

Error Diffusion

- An error is made every time a grayvalue is assigned to be black or white at the output.
- Spread that error to its neighbors to compensate for over/undershoots in the output assignments
  - If input pixel 130 is mapped to white (255) then its excessive brightness (255-130) must be subtracted from neighbors to enforce a bias towards darker values to compensate for the excessive brightness.
- Like ordered dithering, error diffusion permits the output image to share the same dimension as the input image.

Floyd-Steinberg Algorithm



$$f^*(x,y) = f(x,y) + \sum_i \sum_j w_{ij} e(x-i, y-j) = \text{"corrected intensity value"}$$

$$g(x,y) = \begin{cases} 255 & \text{if } f^*(x,y) > MXGRAY/2 \\ 0 & \text{otherwise} \end{cases}$$

$$e(x,y) = f^*(x,y) - g(x,y)$$

$$\sum_i \sum_j w_{ij} = 1$$

Error Diffusion Weights

- Note that visual improvements are possible if left-to-right scanning among rows is replaced by serpentine scanning (zig-zag). That is, scan odd rows from left-to right, and scan even rows from right-to-left.
- Further improvements can be made by using larger neighborhoods.
- The sum of the weights should equal 1 to avoid emphasizing or suppressing the spread of errors.

$$\begin{array}{ccc} \multicolumn{3}{c}{\xrightarrow{\hspace{6em}}} \\ \rightarrow & x & 7/16 \\ 3/16 & 5/16 & 1/16 \end{array}$$

Floyd-Steinberg

$$\begin{array}{ccccc} \multicolumn{5}{c}{\xrightarrow{\hspace{12em}}} \\ \multicolumn{2}{c}{\xrightarrow{\hspace{5em}}} & x & 7/48 & 5/48 \\ 3/48 & 5/48 & 7/48 & 5/48 & 3/48 \\ 1/48 & 3/48 & 5/48 & 3/48 & 1/48 \end{array}$$

Jarvis-Judice-Ninke

$$\begin{array}{ccccc} \multicolumn{5}{c}{\xrightarrow{\hspace{12em}}} \\ \multicolumn{2}{c}{\xrightarrow{\hspace{5em}}} & x & 8/42 & 4/42 \\ 2/42 & 4/42 & 8/42 & 4/42 & 2/42 \\ 1/42 & 2/42 & 4/42 & 2/42 & 1/42 \end{array}$$

Stucki

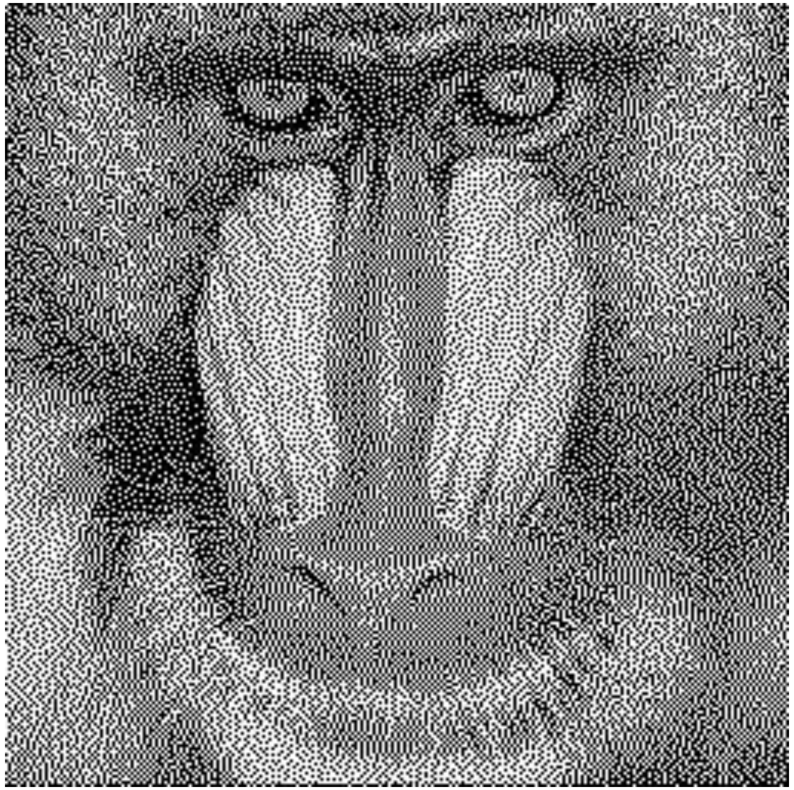
Examples (1)

Floyd-Steinberg

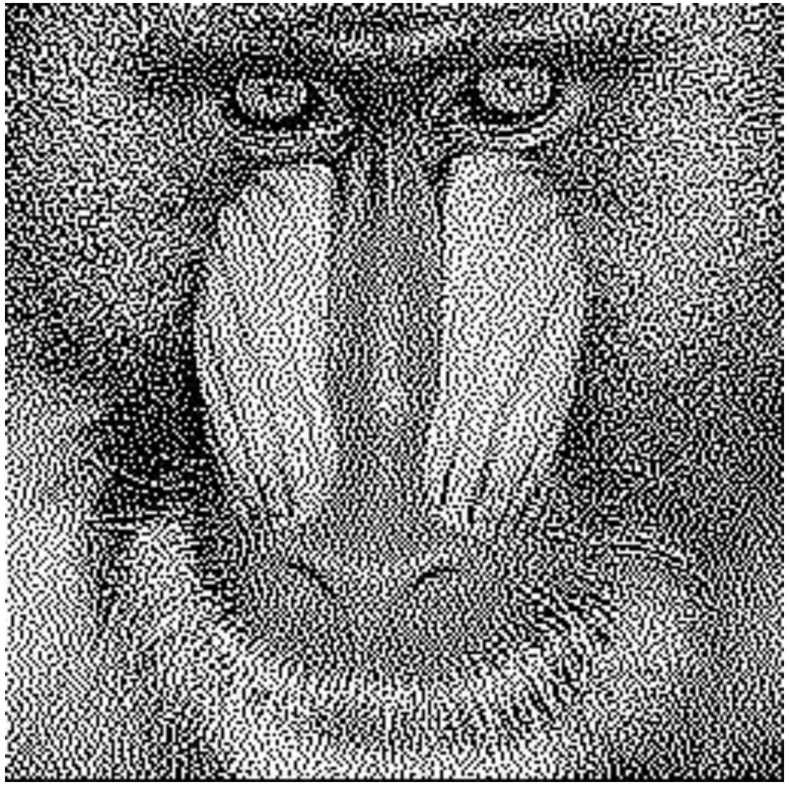
Jarvis-Judice-Ninke

Examples (2)

Floyd-Steinberg

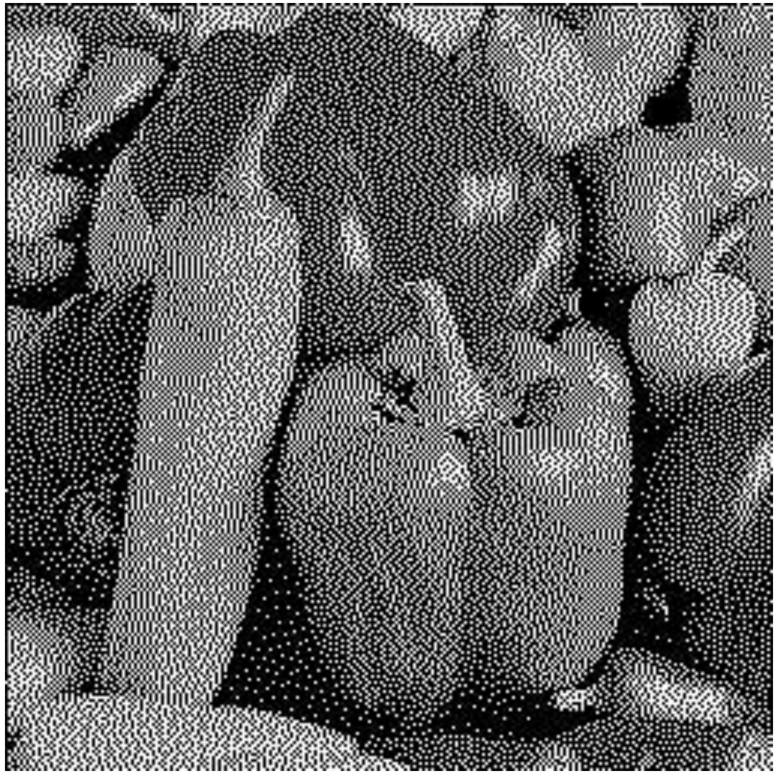


Jarvis-Judice-Ninke

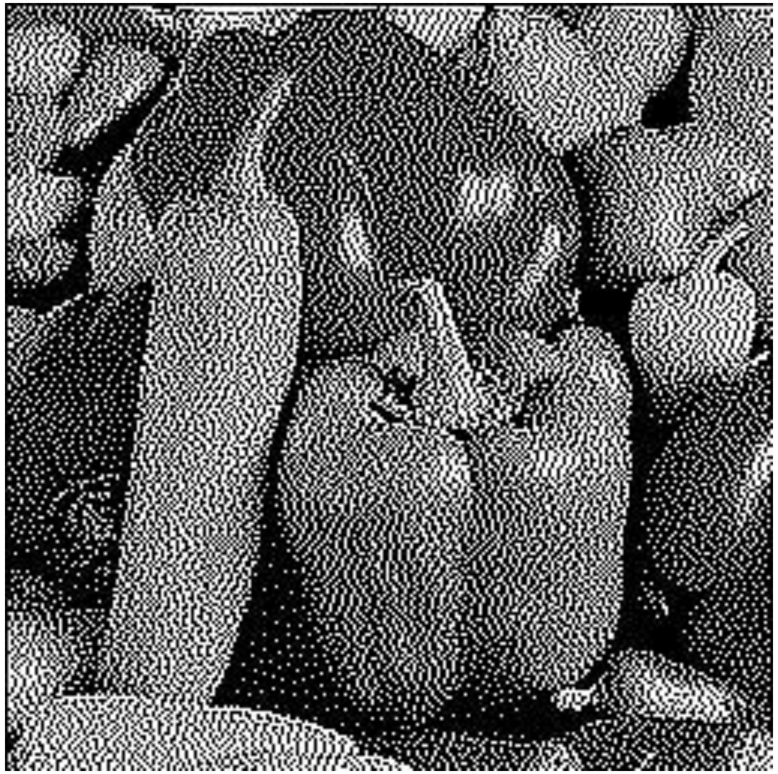


Examples (3)

Floyd-Steinberg



Jarvis-Judice-Ninke



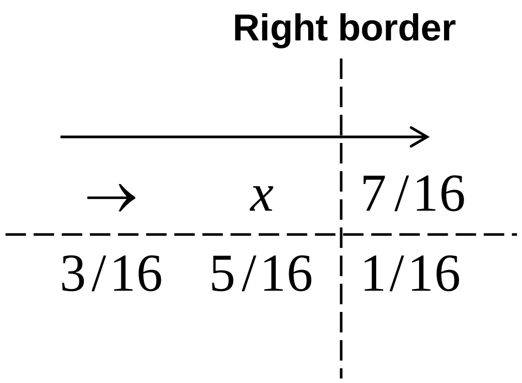
Implementation

```
thr = MXGRAY /2;                   // init threshold value
for(y=0; y<h; y++){                // visit all input rows
 for(x=0; x<w; x++) {              // visit all input cols
     *out = (*in < thr)?           // threshold
            BLACK : WHITE;         // note: use LUT!

     e = *in - *out;               // eval error
     in[ 1 ] +=(e*7/16.);          // add error to E  nbr
     in[w-1] +=(e*3/16.);          // add error to SW nbr
     in[ w ] +=(e*5/16.);          // add error to S  nbr
     in[w+1] +=(e*1/16.);          // add error to SE nbr

     in++;                         // advance input  ptr
     out++;                        // advance output ptr
 }
}
```

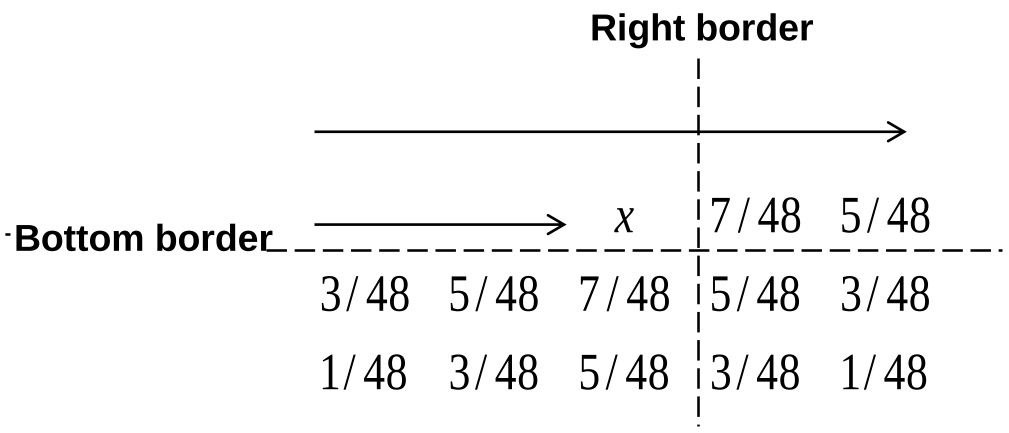
Comments

- Two potential problems complicate implementation:
  - errors can be deposited beyond image border
  - errors may force pixel grayvalues outside the [0,255] range

True for all neighborhood ops



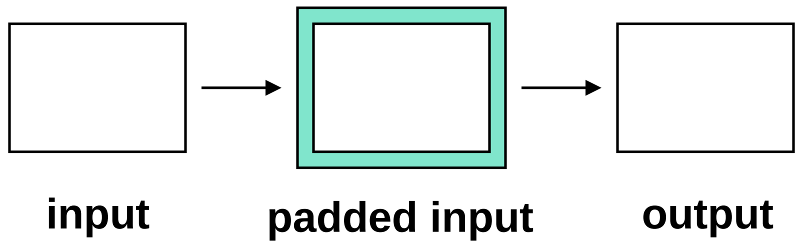
Floyd-Steinberg



Jarvis-Judice-Ninke

Solutions to Border Problem (1)

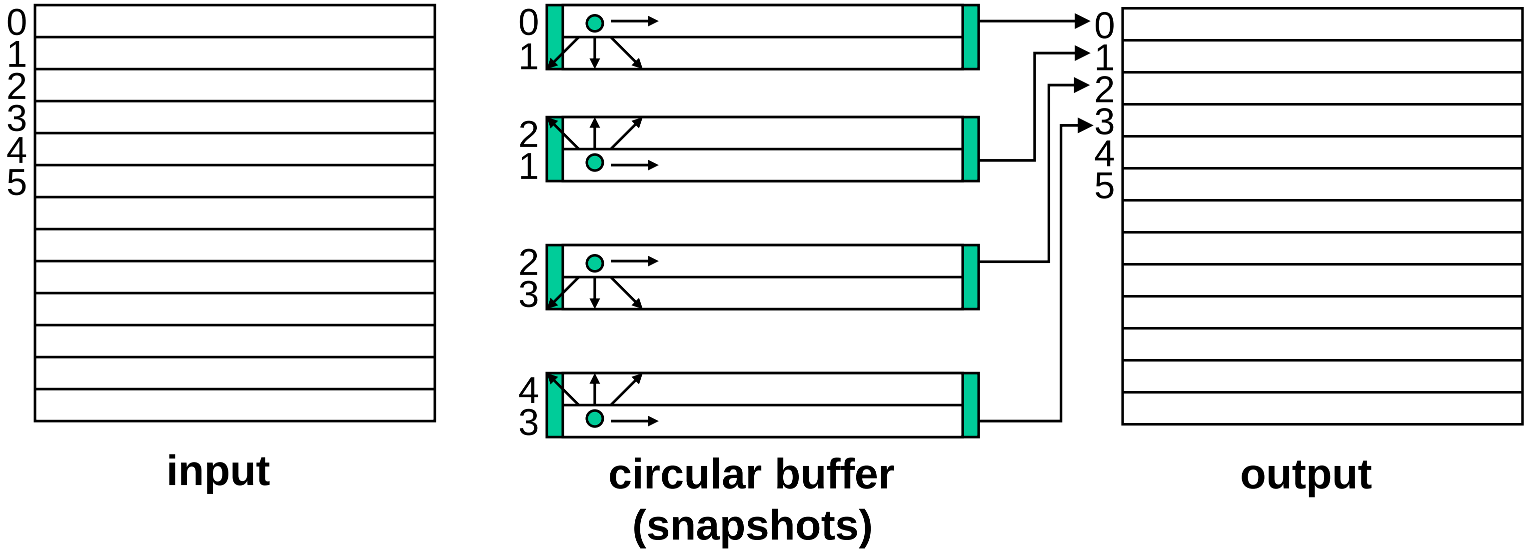
- Perform `if` statement prior to every error deposit
  - Drawback: inefficient / slow
- Limit excursions of sliding weights to lie no closer than 1 pixel from image boundary (2 pixels for J-J-N weights).
  - Drawback: output will be smaller than input
- Pad image with extra rows and columns so that limited excursions will yield smaller image that conforms with original input dimensions. Padding serves as placeholder.
  - Drawback: excessive memory needs for intermediate image



Solutions to Border Problem (2)

- Use of padding is further undermined by fact that 16-bit precision (`short`) is needed to accommodate pixel values outside [0, 255] range.
- A better solution is suggested by fact that only two rows are active while processing a single scanline in the Floyd-Steinberg algorithm (3 for JJN).
- Therefore, use a 2-row (or 3-row) circular buffer to handle the two (or three) current rows.
- The circular buffer will have the necessary padding and 16-bit precision.
- This significantly reduces memory requirements.

Circular Buffer



New Implementation

```
thr = MXGRAY /2;                  // init threshold value
copyRowToCircBuffer(0);           // copy row 0 to circular buffer
for(y=0; y<h; y++){               // visit all input rows
  copyRowToCircBuffer(y+1);       // copy next row to circ buffer
  in1 = buf[  y   %2] + 1;        // circ buffer ptr; skip over pad
  in2 = buf[(y+1)%2] + 1;         // circ buffer ptr; skip over pad
  for(x=0; x<w; x++) {            // visit all input cols
        *out = (*in1 < thr)? BLACK : WHITE;   // threshold

        e = *in1 - *out;          // eval error
        in1[ 1] +=(e*7/16.);      // add error to E  nbr
        in2[-1] +=(e*3/16.);      // add error to SW nbr
        in2[ 0] +=(e*5/16.);      // add error to S  nbr
        in2[ 1] +=(e*1/16.);      // add error to SE nbr

        in1++; in2++              // advance circ buffer ptrs
        out++;                    // advance output ptr
  }
}
```