Computer Graphics

Prof. Feng Liu

Fall 2021

http://www.cs.pdx.edu/~fliu/courses/cs447/

10/11/2021

Last Time

- Image file formats
- Color quantization

Today

- Dithering
- Signal Processing

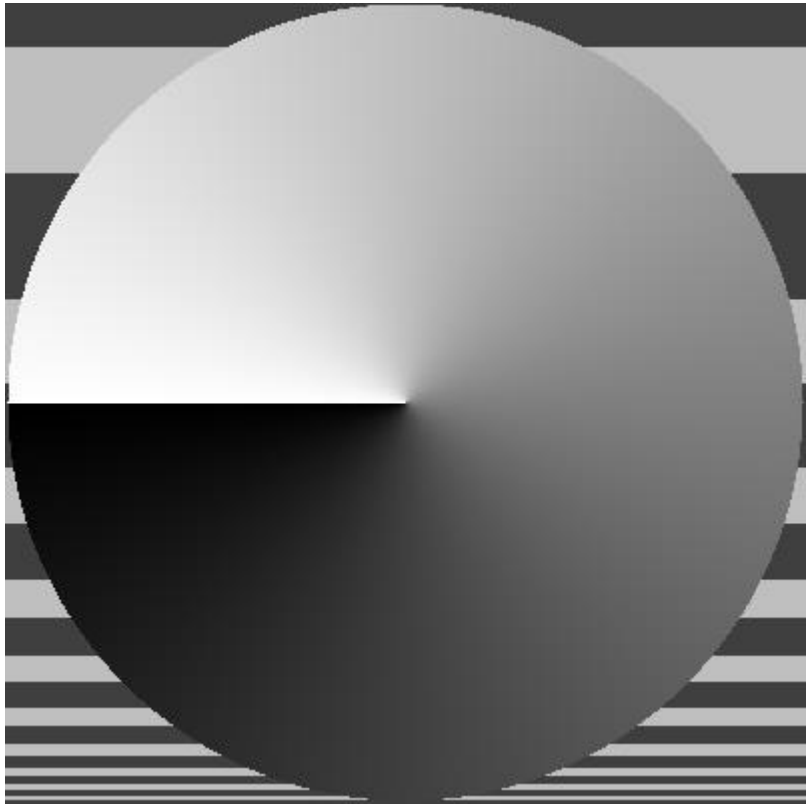
Dithering to Black-and-White

- Black-and-white is still the preferred way of displaying images in many areas
  - Black ink is cheaper than color
  - Printing with black ink is simpler and hence cheaper
  - Paper for black inks is not special

Dithering to Black-and-White

- To get color to black and white, first turn into grayscale: I=0.299R+0.587G+0.114B
  - This formula reflects the fact that green is more representative of perceived brightness than blue is
  - NOTE that it is **not** the equation implied by the RGB->XYZ color space conversion matrix
- **For all dithering we will assume that the image is gray and that intensities are represented as a value in [0, 1.0)**
  - Define new array of floating point numbers
  - `new_image[i] = old_image[i] / (float)256;`
  - To get back:
    `output[i]=(unsigned char)floor(new_image[i]*256)`

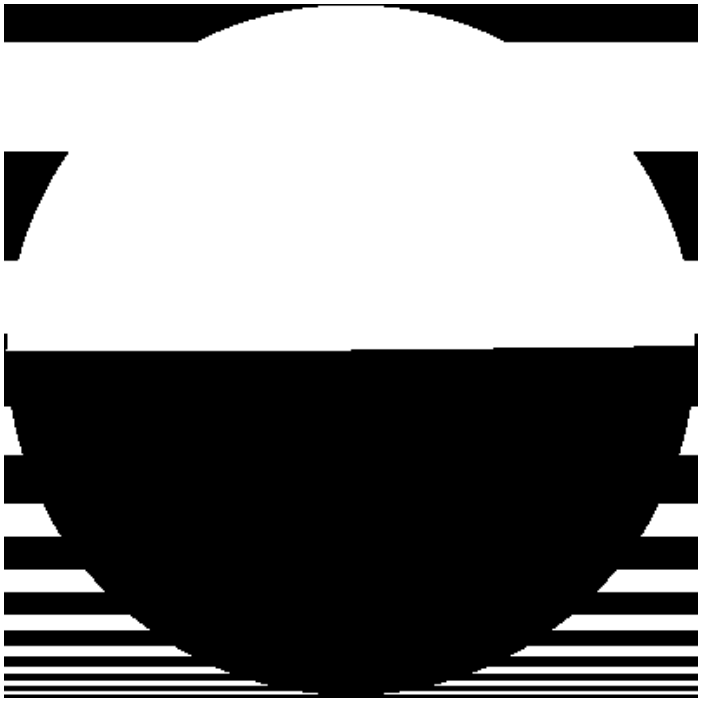
Sample Images



Threshold Dithering

- For every pixel: If the intensity < 0.5, replace with black, else replace with white
  - 0.5 is the threshold
  - This is the naïve version of the algorithm

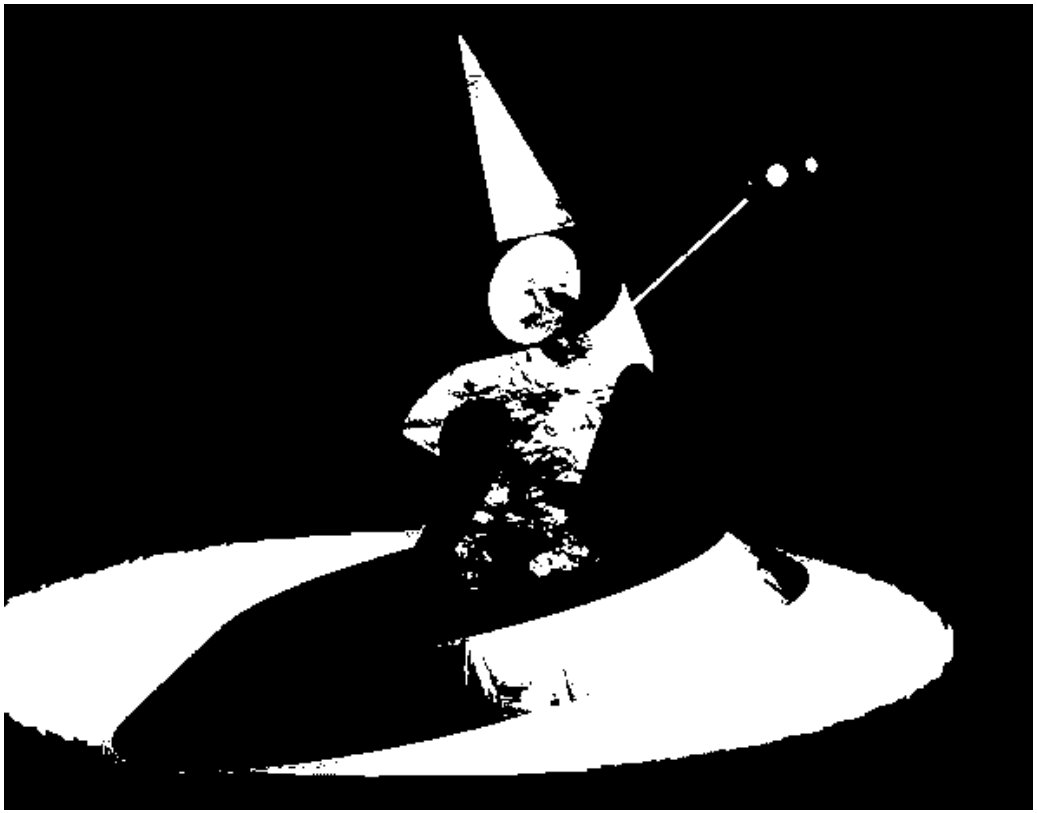
Naïve Threshold Algorithm

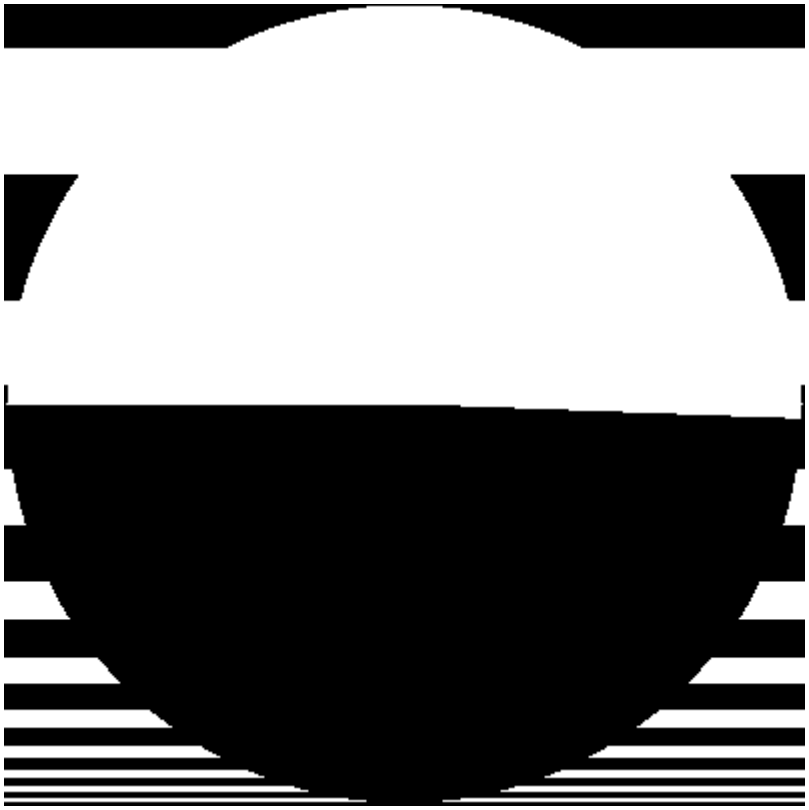


Threshold Dithering

- For every pixel: If the intensity < 0.5, replace with black, else replace with white
  - 0.5 is the threshold
  - This is the naïve version of the algorithm
- To keep the overall image brightness the same, you should:
  - Compute the average intensity over the image
  - Use a threshold that gives that average
  - For example, if the average intensity is 0.6, use a threshold that is higher than 40% of the pixels, and lower than the remaining 60%

Brightness Preserving Algorithm

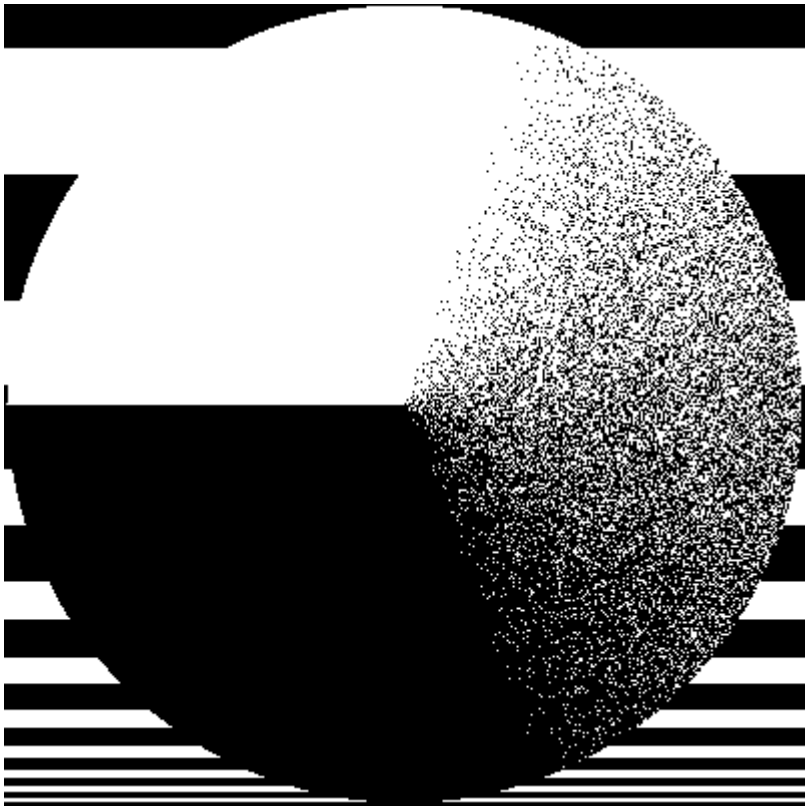




Random Modulation

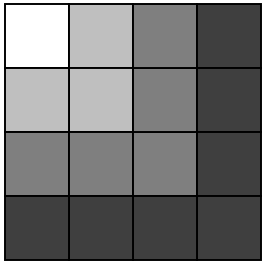
- Add a random amount to each pixel *before* thresholding
  - Typically add *uniformly* random amount from [-a,a]
- Pure addition of noise to the image
  - For better results, add better quality noise
  - For instance, use Gaussian noise (random values sampled from a normal distribution)
- Should use same procedure as before for choosing threshold
- Not good for black and white, but OK for more colors
  - Add a small random color to each pixel before finding the closest color in the table

Random Modulation



Ordered Dithering

- Break the image into small blocks
- Define a *threshold matrix*
  - Use a different threshold for each pixel of the block
  - Compare each pixel to its own threshold
- The thresholds can be clustered, which looks like newsprint
- The thresholds can be “random” which looks better

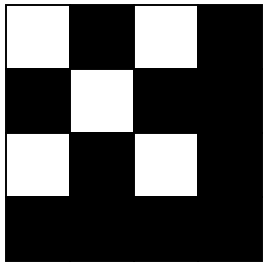


$$\begin{bmatrix} 1 & 0.75 & 0.5 & 0.25 \\ 0.75 & 0.75 & 0.5 & 0.25 \\ 0.5 & 0.5 & 0.5 & 0.25 \\ 0.25 & 0.25 & 0.25 & 0.25 \end{bmatrix}$$

Threshold matrix

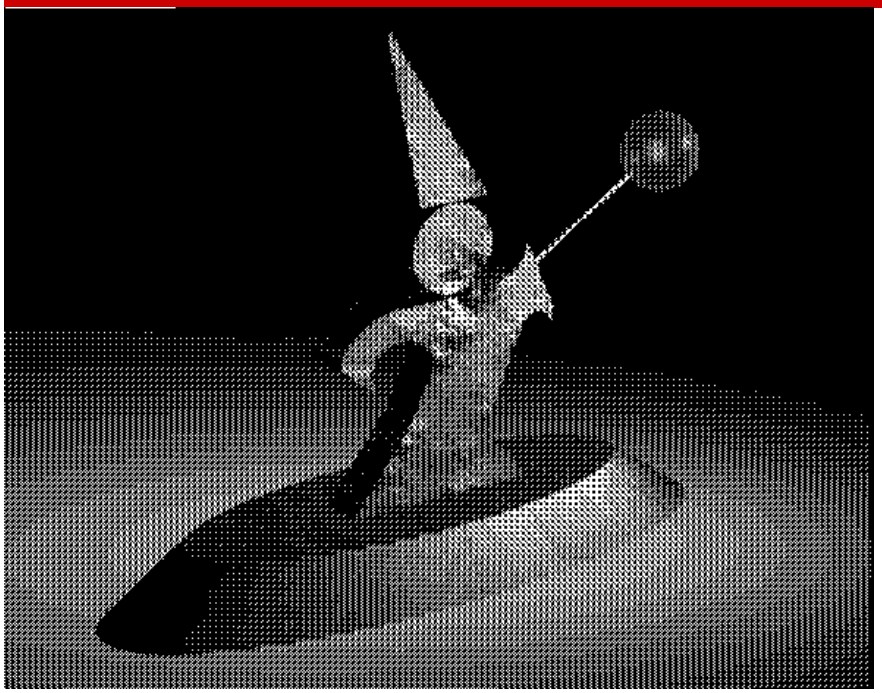
$$\frac{1}{16}\begin{bmatrix} 2 & 16 & 3 & 13 \\ 10 & 6 & 11 & 7 \\ 4 & 14 & 1 & 15 \\ 12 & 8 & 9 & 5 \end{bmatrix}$$

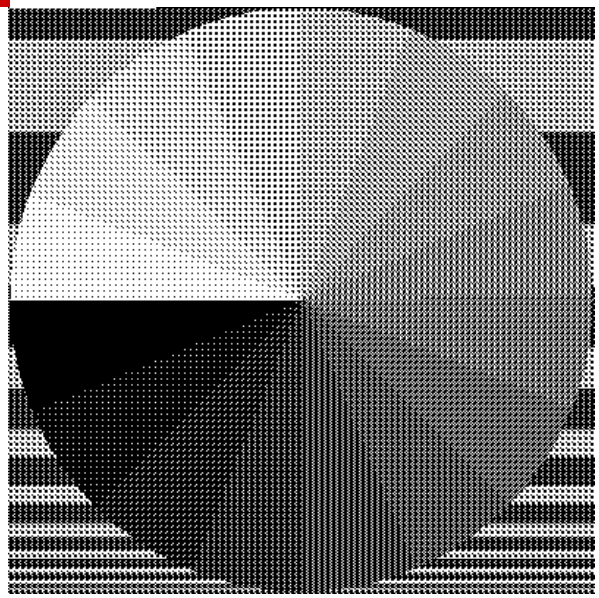
Result



$$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

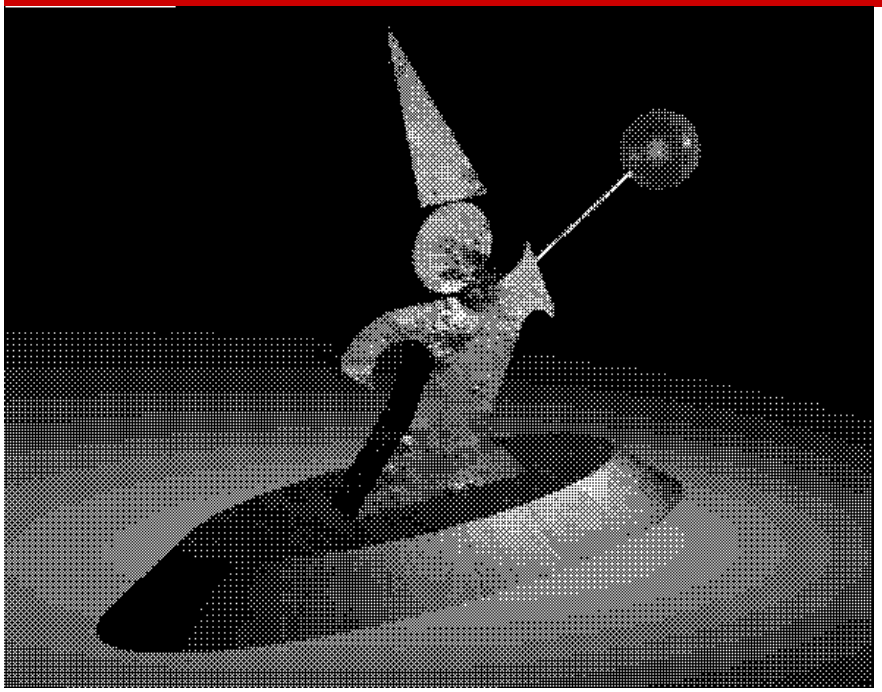
Clustered Dithering

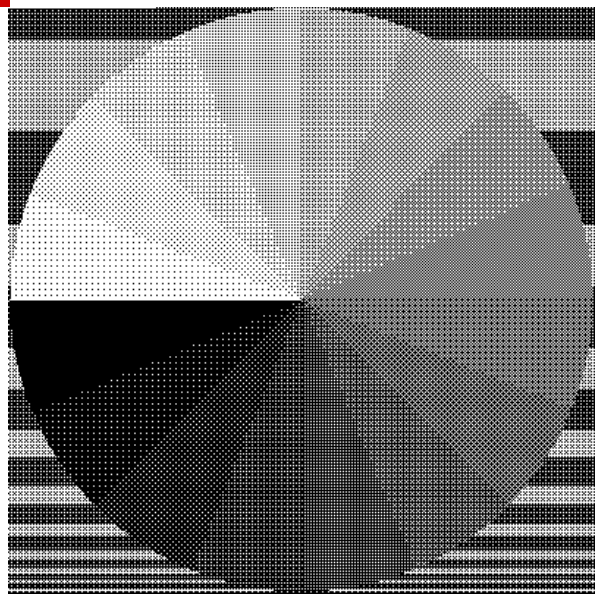




.75	.375	.625	.25
.0625	1	.875	.4375
.5	.8125	.9375	.125
.1875	.5625	.3125	.6875

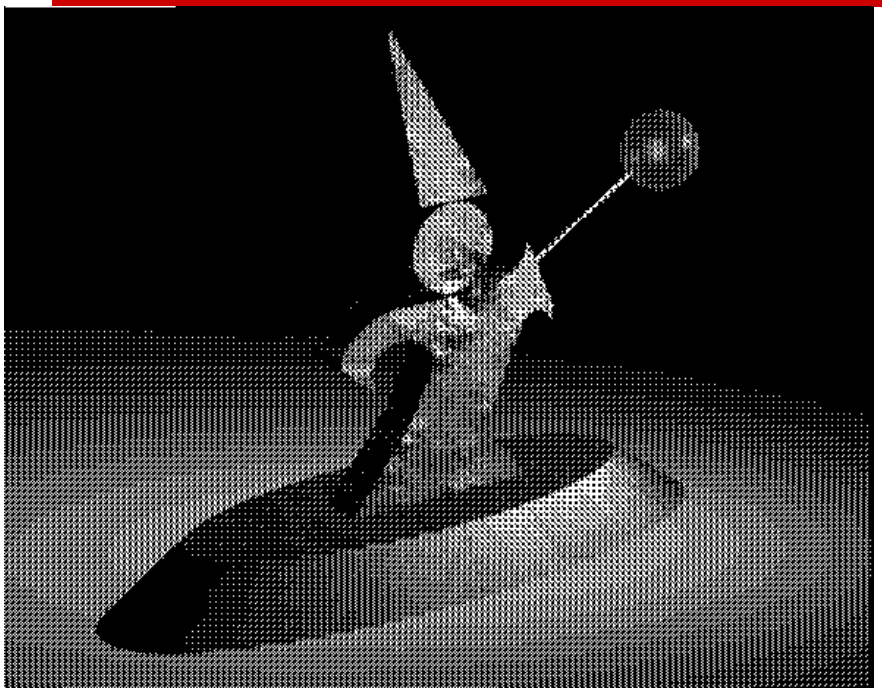
Dot Dispersion



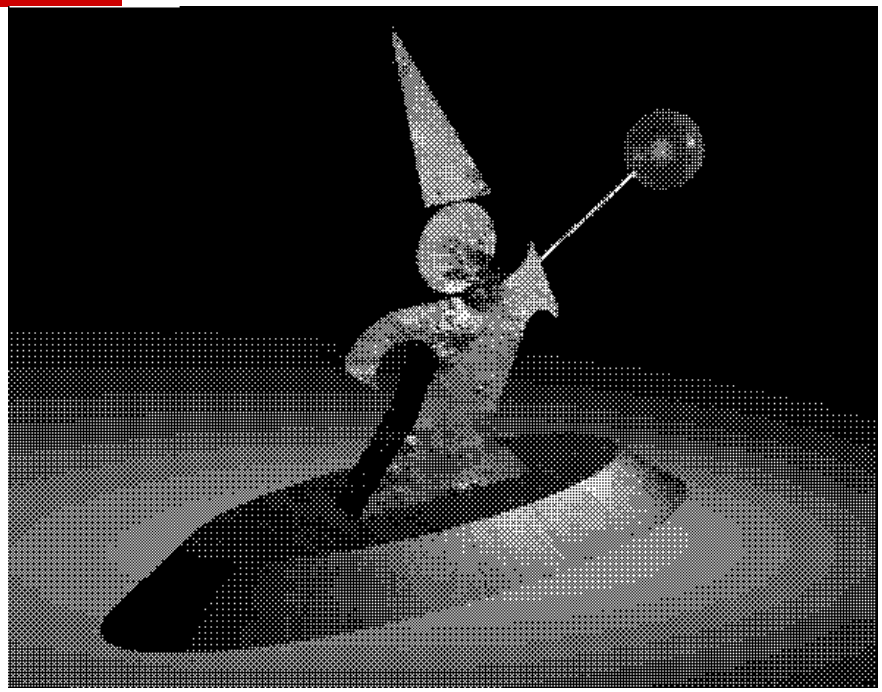


$$\begin{bmatrix} .125 & 1 & .1875 & .8125 \\ .625 & .375 & .6875 & .4375 \\ .25 & .875 & .0625 & .9375 \\ .75 & .5 & .5625 & .3125 \end{bmatrix}$$

Comparison



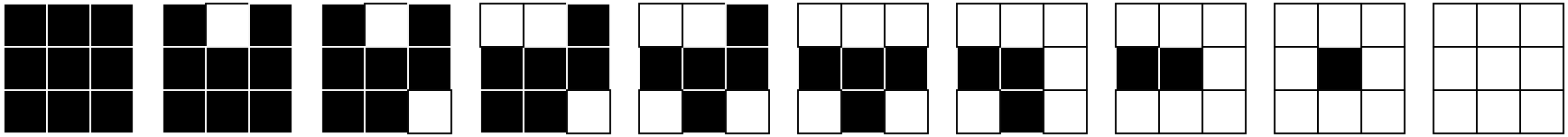
Clustered



Dot Dispersion

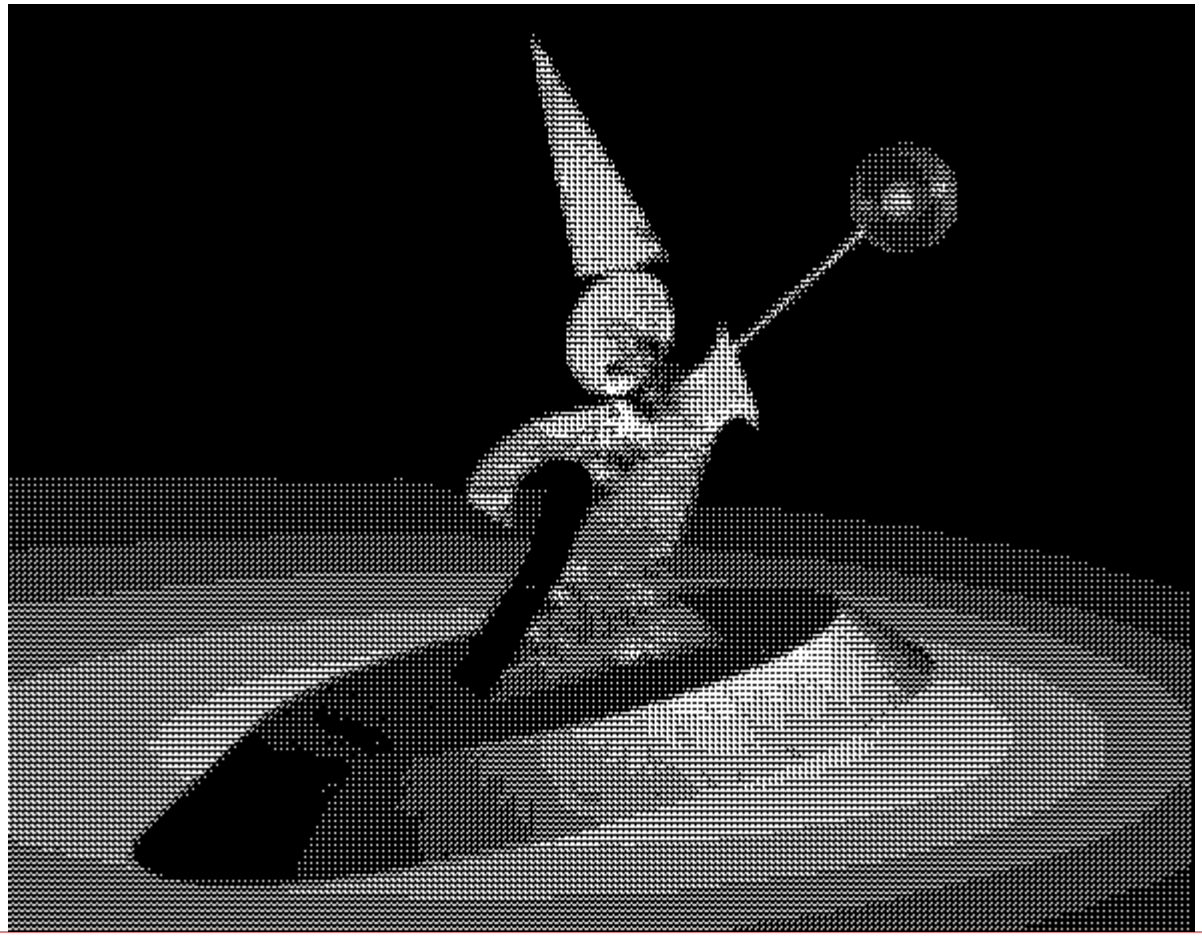
Pattern Dithering

- Compute the intensity of each sub-block and index a pattern

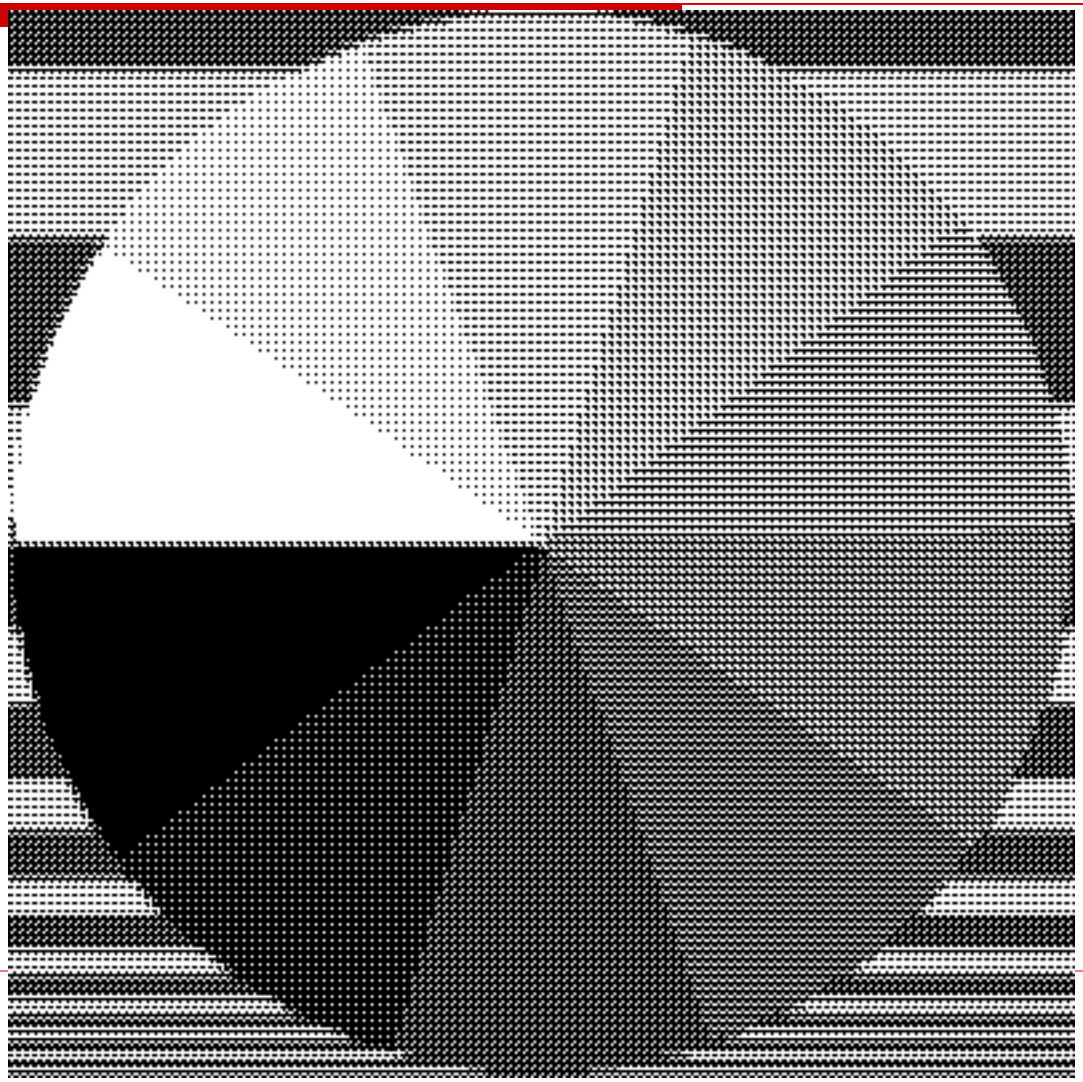


- NOT the same as before
  - Here, each sub-block has one of a fixed number of patterns – pixel is determined only by average intensity of sub-block
  - In ordered dithering, each pixel is checked against the dithering matrix before being turned on
- Used when display resolution is higher than image resolution – not uncommon with printers
  - Use 3x3 output for each input pixel

Pattern Dither (1)

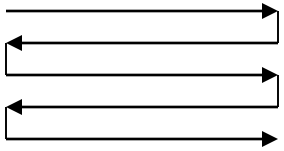


Pattern Dither (2)

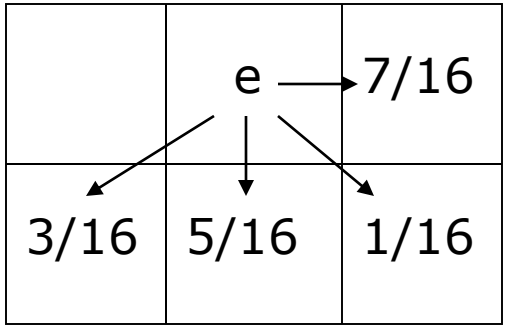


Floyd-Steinberg Dithering

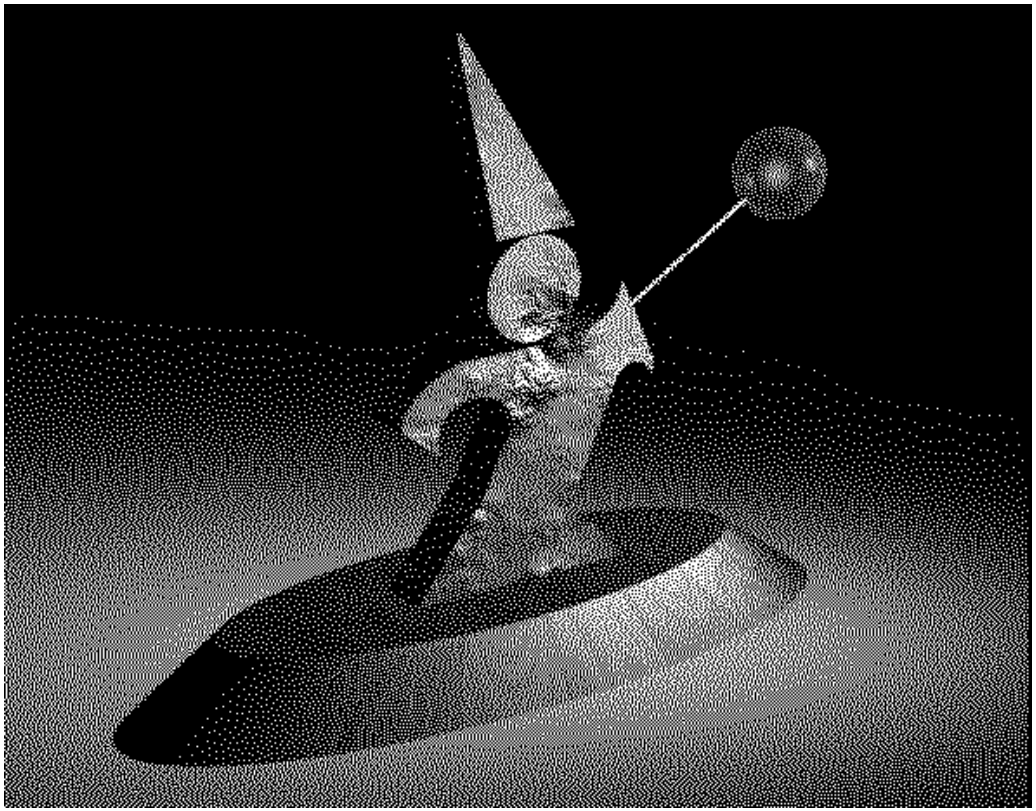
- Start at one corner and work through image pixel by pixel
  - Usually scan top to bottom in a zig-zag

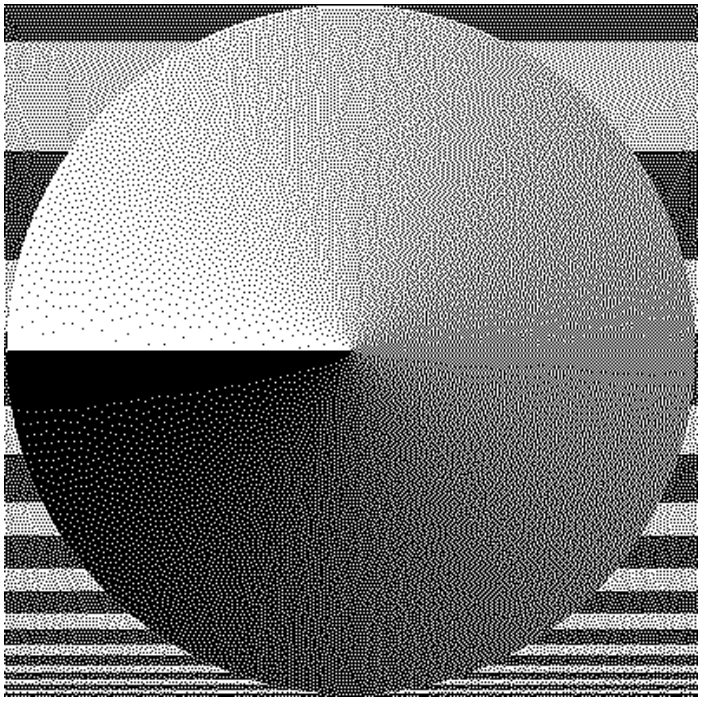


- Threshold each pixel
- Compute the error at that pixel: The difference between what should be there and what you did put there
  - If you made the pixel 0, *e* = original; if you made it 1, *e* = original-1
- Propagate error to neighbors by adding some proportion of the error to each unprocessed neighbor
  - A *mask* tells you how to distribute the error
- Easiest to work with floating point image
  - Convert all pixels to 0-1 floating point
- More detail in class reading materials



Floyd-Steinberg Dithering





Color Dithering

- All the same techniques can be applied, with some modification
- Example is Floyd-Steinberg:
  - Uniform color table
  - Error is difference from nearest color in the color table
  - Error propagation same as that for greyscale
    - Each color channel treated independently

Color Dithering

Comparison to Uniform Quant.

Same color table!

Today

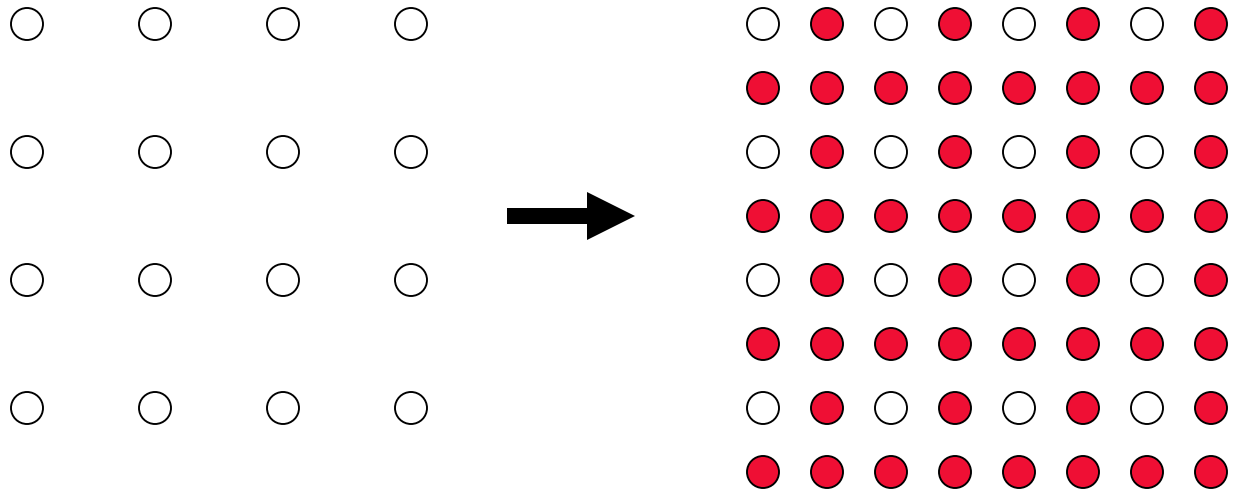
- Dithering
- Signal Processing

Image Manipulation

- We have now looked at basic image formats and color, including transformations of color
- Next, operations involving image *resampling*
  - Scaling, rotating, morphing, …
- But first, we need some signal processing
  - Also important for *anti-aliasing*, later in class

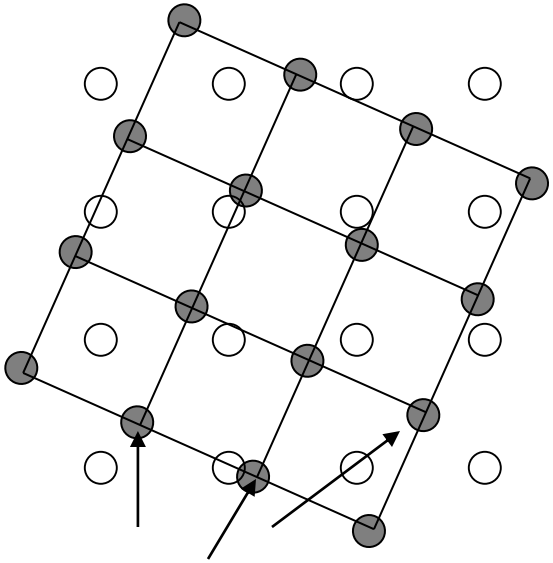
Enlarging an Image

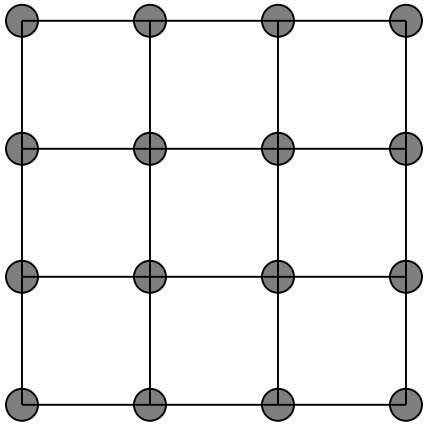
- To enlarge an image, you have to add pixels between the old pixels
- What values do you choose for those pixels?



Rotating an Image

- Pixels in the new image come from their rotated positions in the original image
- These rotated locations might not be in “nice” places



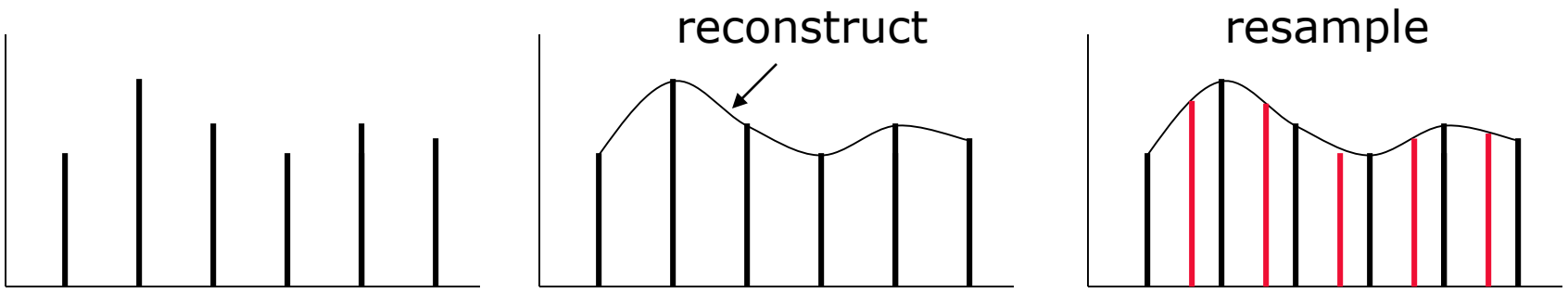


New image

The positions of the new pixels in the original image

Images as Samples of Functions

- We can view an image as a set of *samples* from an ideal function
- If we knew what the function was, we could enlarge the image by *resampling* the function
- Failing that, we can *reconstruct* the function and then resample it



Why Signal Processing?

- Signal processing provides the tools for understanding sampling and reconstruction

Function representations

- A function can be represented as a sum of sin's and cos's of (possibly) all frequencies:

$$f(x) = \frac{1}{2\pi}\int_{-\infty}^{\infty} F(\omega)e^{i\omega x}d\omega$$

$$e^{i\omega x} = \cos\omega x + i\sin\omega x$$

- $F(\omega)$ is the *spectrum* of the function $f(x)$
  - The spectrum is how much of each frequency is present in the function
  - We're talking about functions, not colors, but the idea is the same

Fourier Transform

- *F(ω)* is computed from *f(x)* by the *Fourier Transform:*

$$F(\omega) = \int_{-\infty}^{\infty} f(x) e^{-i\omega x} dx$$

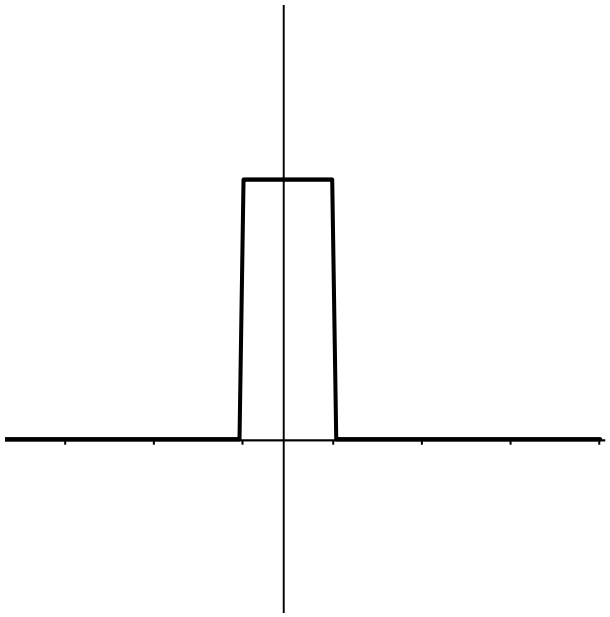
Example: Box Function

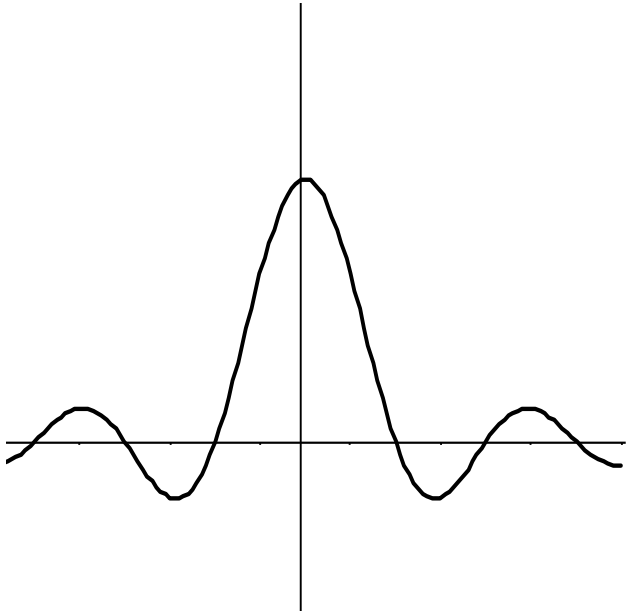
$$f(x) = \begin{cases} 1 & |x| \le 1/2 \\ 0 & |x| > 1/2 \end{cases}$$

$$F(\omega) = \frac{\sin \pi f}{\pi f} \qquad f = \frac{\omega}{2\pi}$$

$$= \operatorname{sinc} f$$

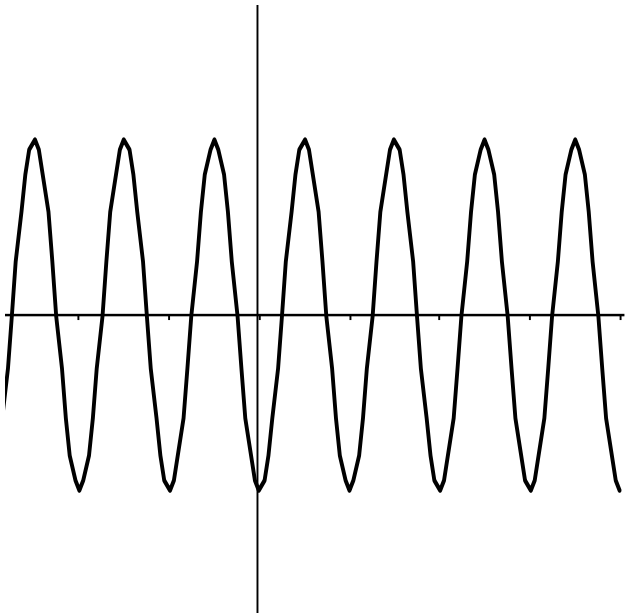
Box Function and Its Transform

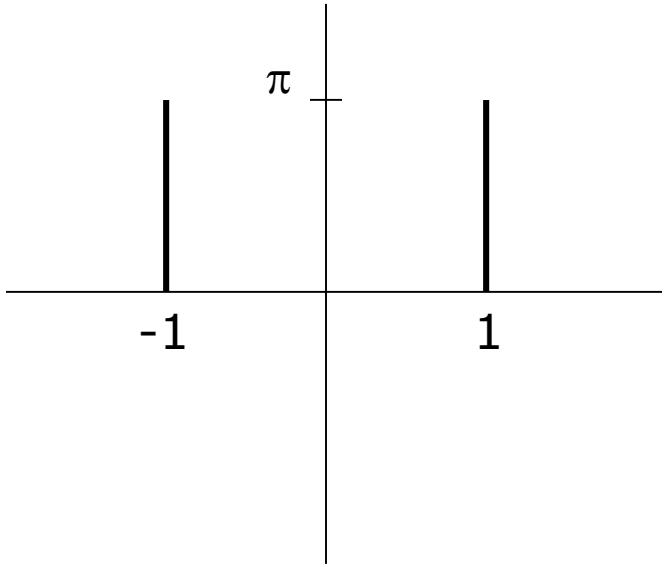




- Two different *representations* of the *same* function
  - $f(x)$ spatial domain
  - $F(\omega)$ frequency domain

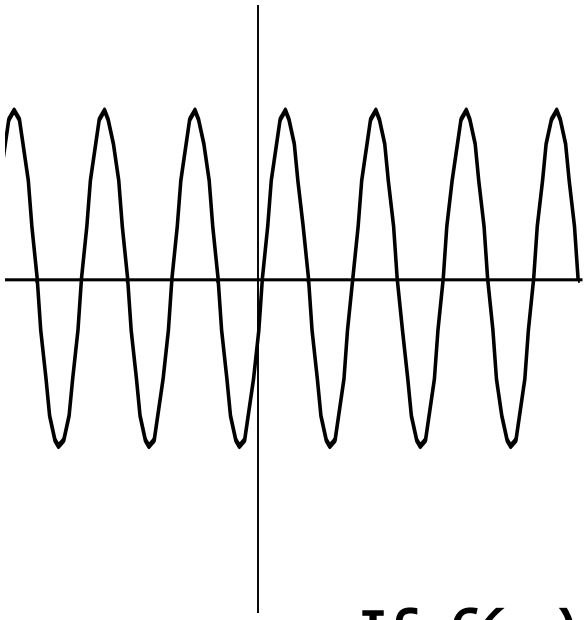
Cosine and Its Transform

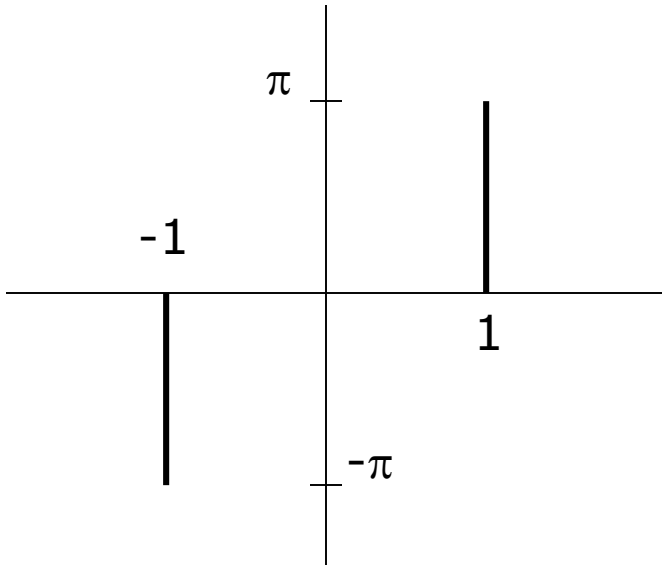




If *f(x)* is even, so is *F(ω)*

Sine and Its Transform



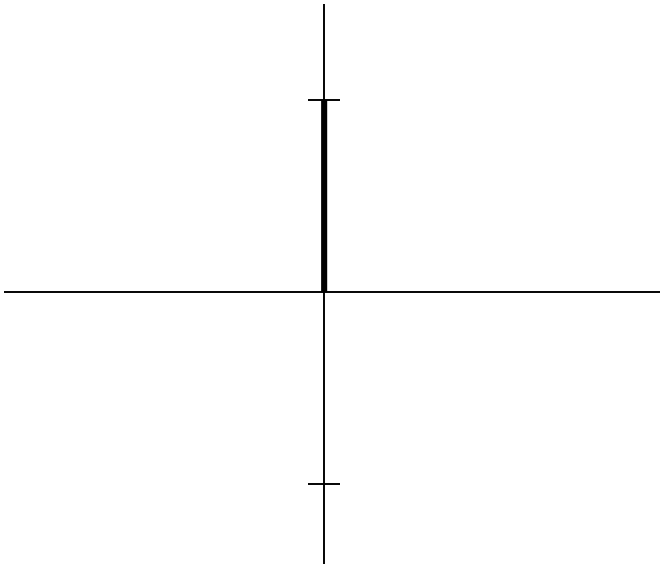


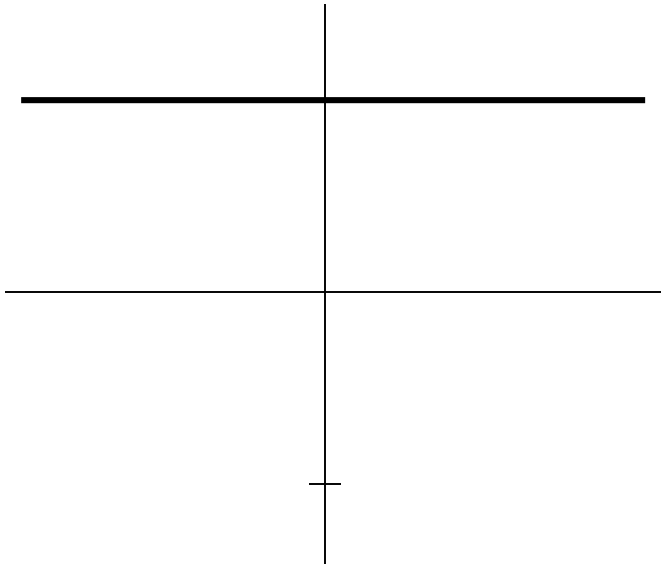
If *f(x)* is odd, so is *F(ω)*

Constant Function and Its Transform

The constant function only contains the 0^{th} frequency – it has no wiggles

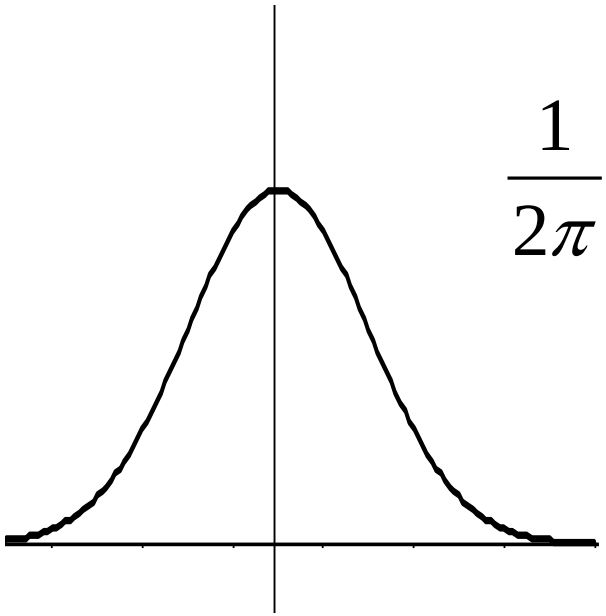
Delta Function and Its Transform

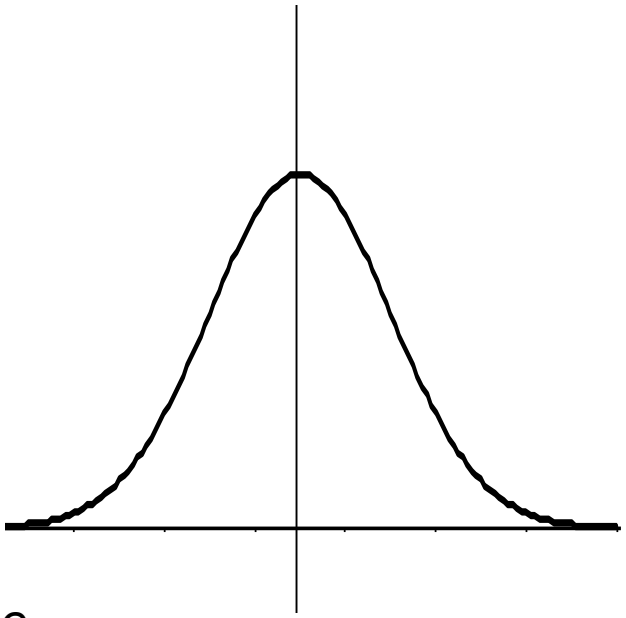




Gaussian and Its Transform

$$\frac{1}{2\pi} e^{-\frac{x^2}{2}}$$





They are the same

Qualitative Properties

- The spectrum of a function tells us the relative amounts of high and low frequencies
  - **Sharp edges give high frequencies**
  - **Smooth variations give low frequencies**
- A function is *bandlimited* if its spectrum has no frequencies above a maximum limit
  - sin, cos are band limited
  - Box, Gaussian, etc are not

Functions to Images

- Images are 2D, discrete functions
- 2D Fourier transform uses product of sin's and cos's
- Fourier transform of a discrete, quantized function will only contain discrete frequencies in quantized amounts
  - In particular, we can store the Fourier transform of a discrete image in the same amount of space as we can store the image
- Numerical algorithm: Fast Fourier Transform (FFT) computes discrete Fourier transforms

Next Time

- Filtering
- Resampling
- Aliasing
- Compositing