

Image Transforms and Interpolations

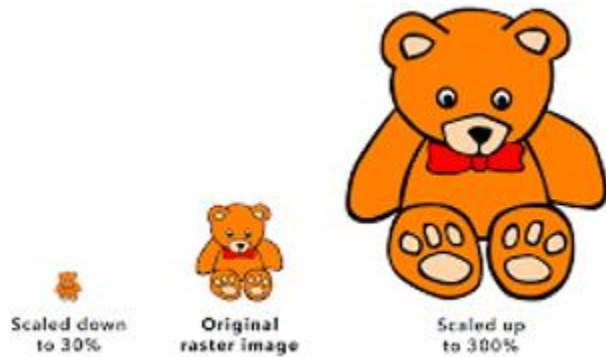
Recap

- Elementwise Versus Matrix Operations
- Operations on Images
 - Arithmetic Operations
 - Set and Logical Operations
- Spatial Operations
 - Single-pixel Operations
 - Neighborhood Operations
 - Geometric Transformations

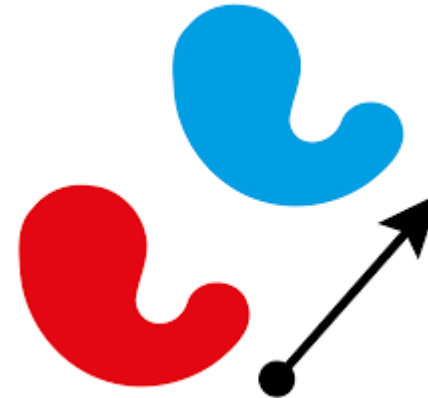
Lecture Objectives

- Spatial Operations
 - Geometric spatial transformations
- Image Interpolation
- Image Registration
- Image Domain Transforms

Image Transformation



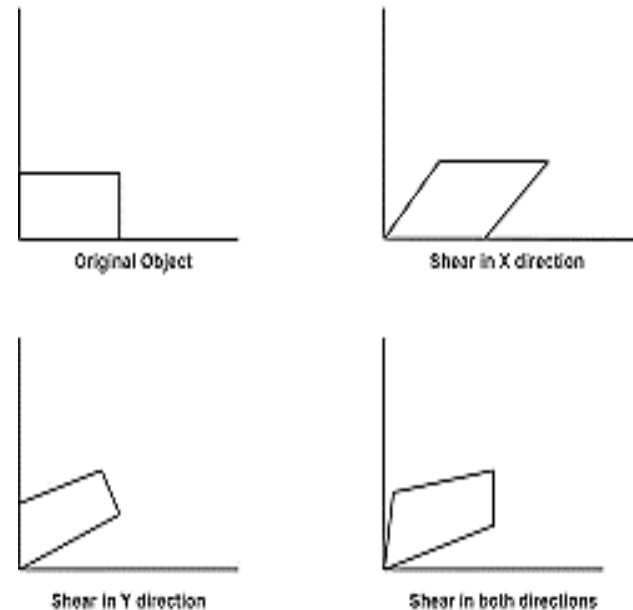
Scaling



Translation



Rotation



Shearing

Geometric Spatial Transformations

Geometric Spatial Transformations

- We use geometric transformations to *modify the spatial arrangement of pixels* in an image.
- These transformations are called ***rubber-sheet transformations*** because they may be viewed as analogous to “printing” an image on a rubber sheet, then stretching or shrinking the sheet according to a predefined set of rules.
- Geometric transformations of digital images consist of two basic operations:
 - 1) Spatial transformation of coordinates.
 - 2) Intensity interpolation that assigns intensity values to the spatially transformed pixels.

Spatial transformation of coordinates

- The spatial transformation of coordinates may be expressed as:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \mathbf{T} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} t_{11} & t_{12} \\ t_{21} & t_{22} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

where (x, y) are *pixel coordinates* in the original image, \mathbf{T} is the transformation matrix, and (x', y') are the *corresponding pixel coordinates* of the transformed image.

- Example:** The transformation $(x', y') = (x/2, y/2)$ shrinks the original image to half its size in both spatial directions.

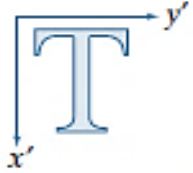
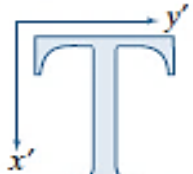
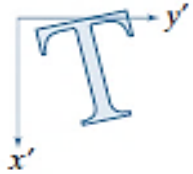
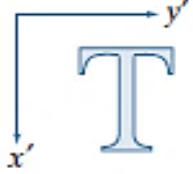
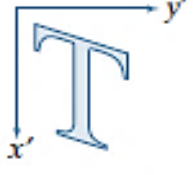

Affine transformations

- **The key characteristic of an affine transformation in 2-D image is that it preserves points, straight lines, and planes after the transformations.**
- **Includes:** scaling, translation, rotation, and shearing.
- It is possible to use homogeneous coordinates to express all four affine transformations using a single 3 × 3 matrix in the following general form:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \mathbf{A} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$





For an image, the working space is bi-dimensional and will be extended to 3 dimensions by the use of a **homogeneous coordinate**.

Affine transformations

Transformation Name	Affine Matrix, A	Coordinate Equations	Example
Identity	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{aligned} x' &= x \\ y' &= y \end{aligned}$	
Scaling/Reflection (For reflection, set one scaling factor to -1 and the other to 0)	$\begin{bmatrix} c_x & 0 & 0 \\ 0 & c_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{aligned} x' &= c_x x \\ y' &= c_y y \end{aligned}$	
Rotation (about the origin)	$\begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{aligned} x' &= x \cos \theta - y \sin \theta \\ y' &= x \sin \theta + y \cos \theta \end{aligned}$	
Translation	$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{aligned} x' &= x + t_x \\ y' &= y + t_y \end{aligned}$	
Shear (vertical)	$\begin{bmatrix} 1 & s_v & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{aligned} x' &= x + s_v y \\ y' &= y \end{aligned}$	
Shear (horizontal)	$\begin{bmatrix} 1 & 0 & 0 \\ s_h & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{aligned} x' &= x \\ y' &= s_h x + y \end{aligned}$	

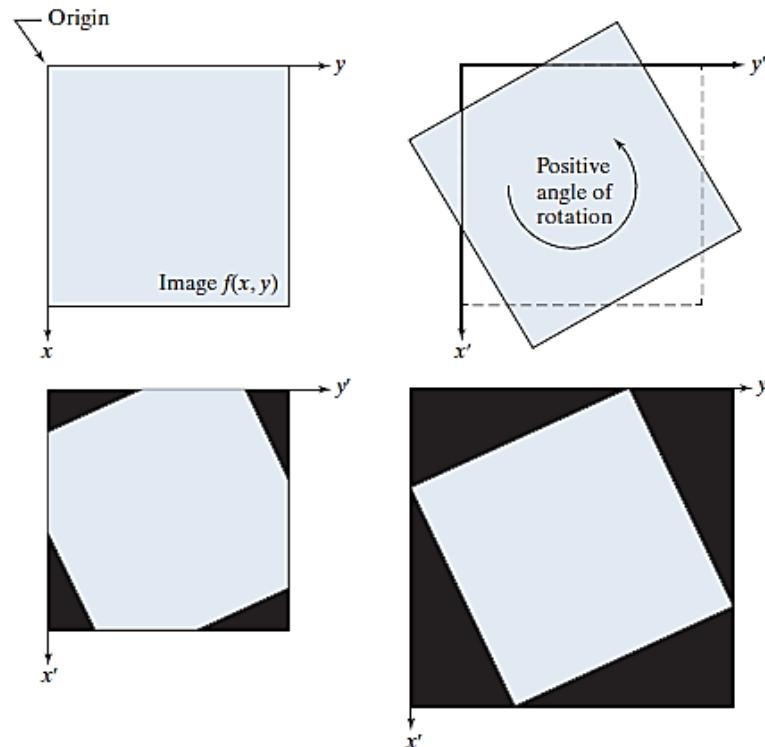
Angles of clockwise rotation are **negative** and vice versa

Affine transformations

Affine Transform	Example	Transformation Matrix	
Translation		$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix}$	t_x specifies the displacement along the x axis t_y specifies the displacement along the y axis.
Scale		$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$	s_x specifies the scale factor along the x axis s_y specifies the scale factor along the y axis.
Shear		$\begin{bmatrix} 1 & sh_y & 0 \\ sh_x & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	sh_x specifies the shear factor along the x axis sh_y specifies the shear factor along the y axis.
Rotation		$\begin{bmatrix} \cos(q) & \sin(q) & 0 \\ -\sin(q) & \cos(q) & 0 \\ 0 & 0 & 1 \end{bmatrix}$	q specifies the angle of rotation.

Affine transformations-rotation issue

- The *size* of the spatial rectangle needed to contain a rotated image is **larger** than the rectangle of the original image. We have two options for dealing with this:
 - 1) we can crop the rotated image so that its size is equal to the size of the original image, or
 - 2) we can keep the larger image containing the full rotated original.



Affine transformations

- We can concatenate a sequence of transformation operations as a single affine matrix **A**. All we have to do is to form a 3×3 matrix equal to the **product of the required transformation matrices** from the previous slide table.

Example: perform both Scaling and Translation on an image together

c_x	0	0
0	c_y	0
0	0	1

Scaling

 \times

1	0	t_x
0	1	t_y
0	0	1

Translation

 $=$

c_x	0	$c_x t_x$
0	c_y	$c_y t_y$
0	0	1

Scaling & Translation

- Pixels are relocated to new locations. **What's next?**
 - We have to assign intensity values to new locations.**

Image Interpolation

Refer to below links for more details:

- <https://tech-algorithm.com/index.php?category=3>
- <https://www.cambridgeincolour.com/tutorials/image-interpolation.htm>
- <https://slidetodoc.com/geometric-operations-and-morphing-1-geometric-transformation-operations/>

Image Interpolation

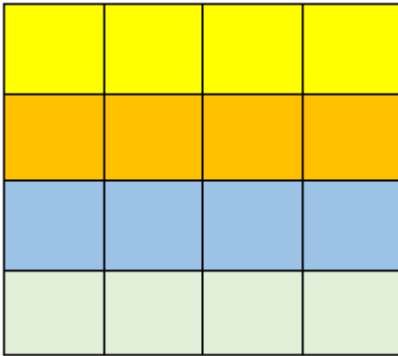
- Interpolation is used in tasks such as **zooming**, **shrinking**, **rotating**, and **geometrically correcting** digital images.
- An image represented as a function $f(x,y)$ tells us the **intensity values** at the coordinates (x,y) when x and y are both **integers**.
- **Image interpolation** refers to the “guess” of intensity values at missing locations using intensity values at known locations. The two steps in image interpolation are:
 - 1) **First**, map the locations of source and target image pixels using **forward/inverse** mapping functions.
 - 2) **Second**, assign the intensities of source pixels to mapped target pixels.



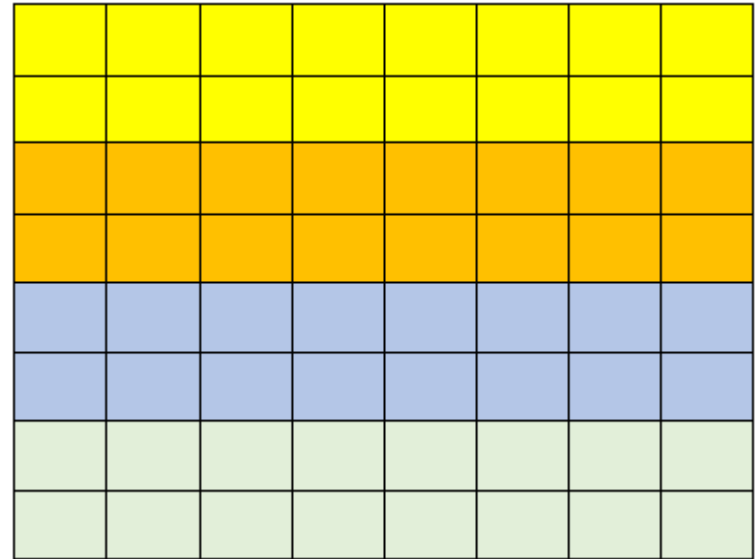
Image Interpolation

- **ZOOMING** an image to its **2x** times.

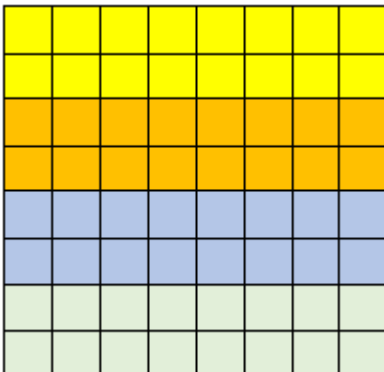
1. Original image



2. A 2x output grid having same pixel spacing as the original image



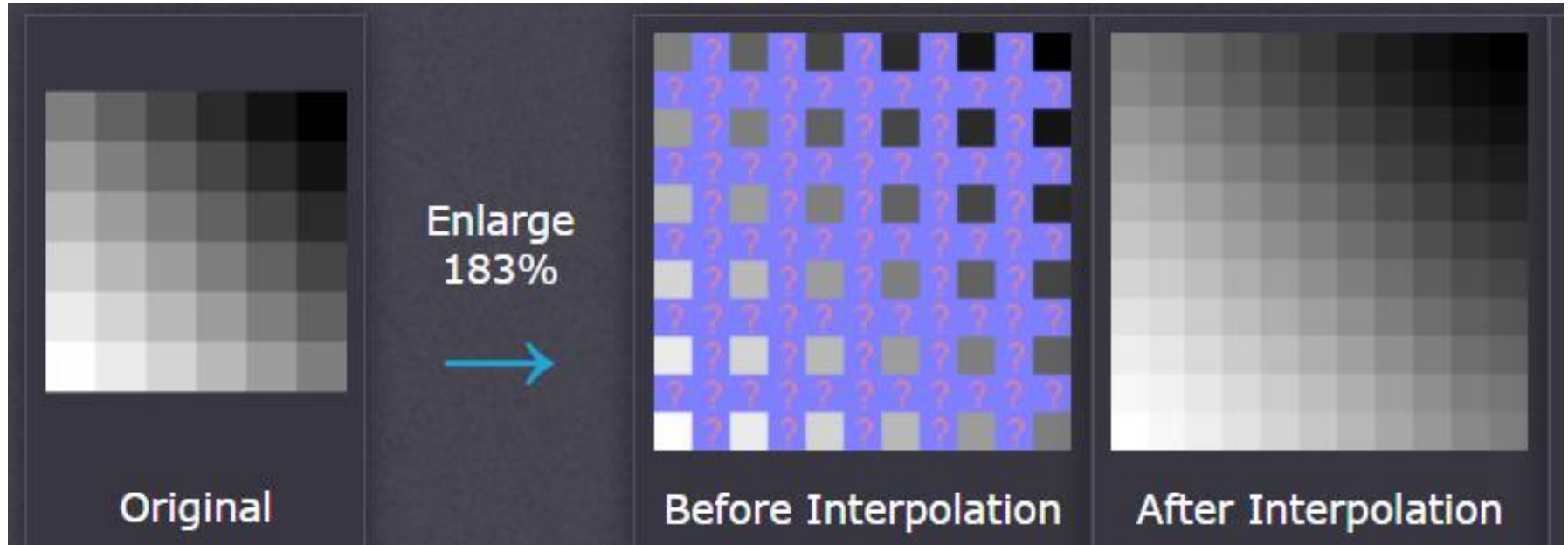
3. Shrink output grid to original image size and perform pixel location mapping



4. Assign intensities to all the point locations in the output grid (overlay) and expand it back to the 2x size to obtain the resized image

Image Interpolation

- **ZOOMING** an image.



The **transformations** relocate pixels on an image to new locations. To complete the process, we have to assign intensity values to those locations.

Image Interpolation - Forward mapping

Step-1: Scan the pixels of the input image $f(x,y)$.

Step-2: At each location (x, y) , compute (**mapping**) the spatial location (x', y') of the corresponding pixel in the output image $f'(x', y')$ and copy the intensity value:

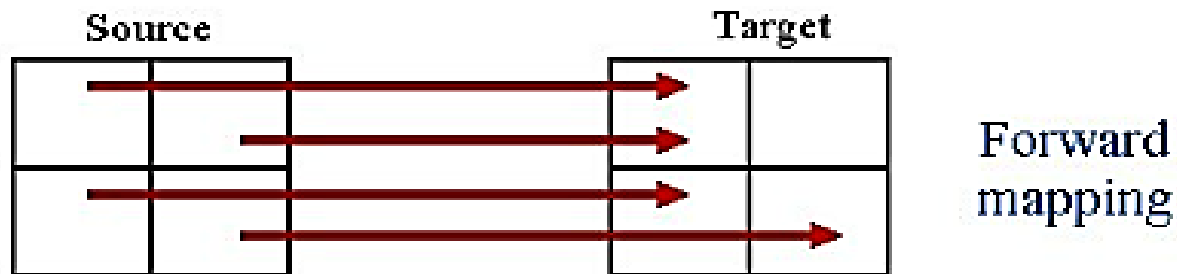
$$(x', y') = T(x, y)$$

– where, T is a **mapping function** from $f(x,y)$ to $f'(x', y')$

- Depending on the specific spatial transformation function, there are two problems with forward mapping:
 - **Overlaps:** Two or more pixels in the input image can be transformed to the same location in the output image, raising the question of how to combine multiple output values into a single output pixel value.
 - **Holes:** In addition, it is possible that some output locations may not be assigned a pixel at all.

Image Interpolation - Forward mapping

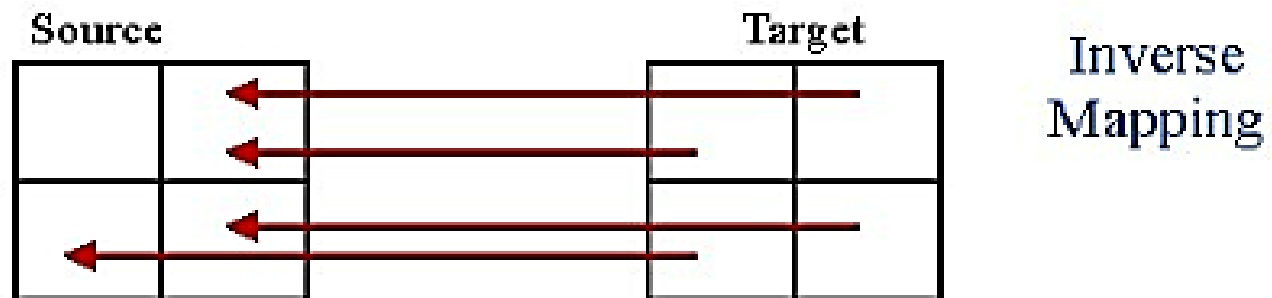
- Forward mapping:
$$x \rightarrow f_x(x, y) = x'$$
$$y \rightarrow f_y(x, y) = y'$$



- Problems with forward mapping due to sampling:
 - Holes (some target pixels are not populated)
 - Overlaps (some target pixels assigned few colors)

Image Interpolation - Inverse mapping

- Inverse mapping:
 $x' \rightarrow f_x^{-1}(x', y') = x$
 $y' \rightarrow f_y^{-1}(x', y') = y$



- Each target pixel assigned a single color.
- Color Interpolation is required.

Image Interpolation - Inverse mapping

Step-1: Scan the pixels of the output image $f'(x', y')$.

Step-2: Compute (**inverse mapping**) the corresponding location (x, y) in the input image $f(x, y)$ and copy the intensity value:

$$(x, y) = T^{-1}(x', y')$$

– where, **T** is a **inverse mapping function** from $f(x, y)$ to $f'(x', y')$

- **Interpolate** among the nearest input pixels (using *nearest neighbor*, *bilinear*, *bicubic* etc.) to determine the intensity of the output pixel value.
- Inverse mappings are more efficient to implement than forward mappings, and are used in numerous commercial implementations of spatial transformations (for example, MATLAB uses this approach).

Image Interpolation - Inverse mapping

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ 0 & 0 & 1 \end{bmatrix}^{-1} \cdot \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

For scaling, rotation

$$\text{Scaling: } \begin{bmatrix} 1/S_x & 0 & 0 \\ 0 & 1/S_y & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \text{Rotation: } \begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The inverse transformation is found by matrix inversion of the transformation matrix

Image Interpolation - Example

- Example: Translation

$$x' = f_x(x, y) = x + 3$$

$$y' = f_y(x, y) = y - 1$$

$$I'(x + 3, y - 1) = I(x, y)$$

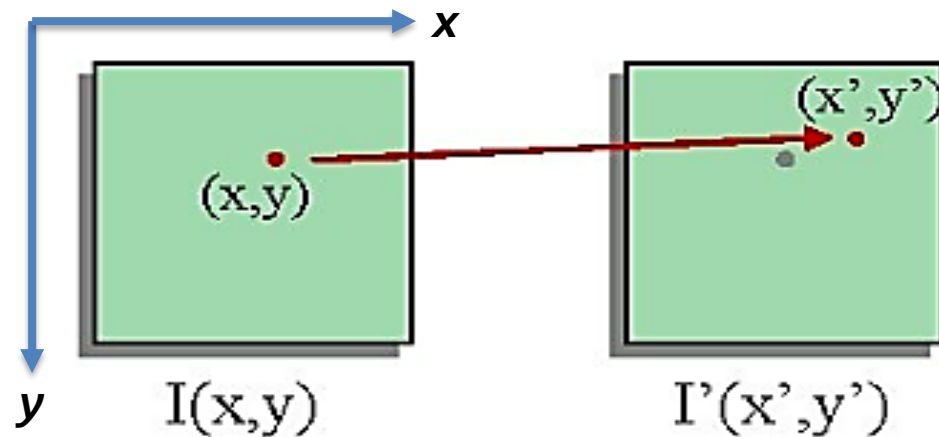
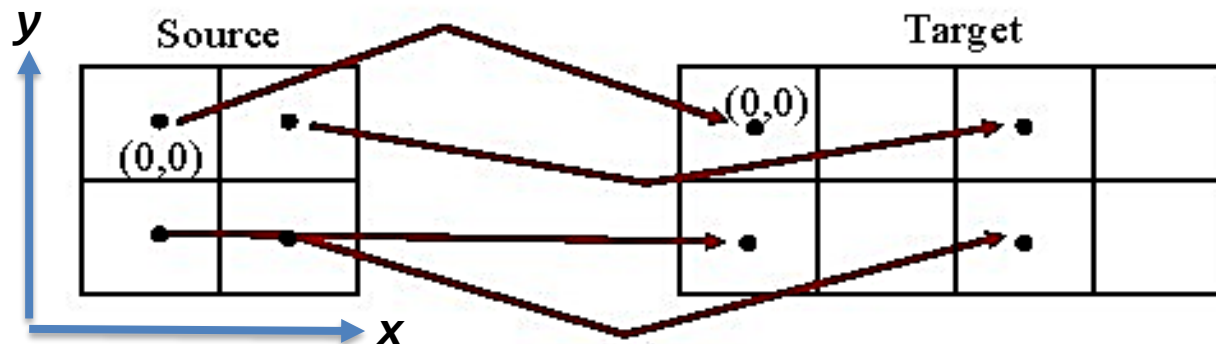


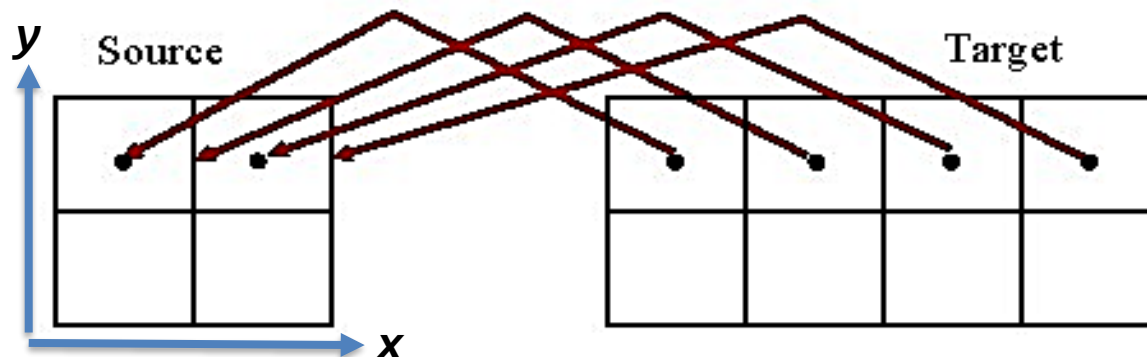
Image Interpolation - Example

- Example: Scaling along X

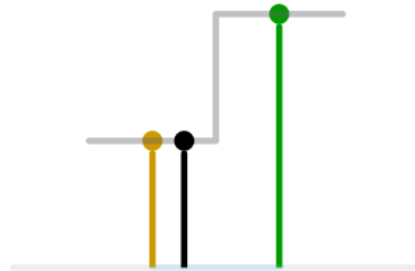
– Forward mapping: $x' = 2x$; $y' = y$



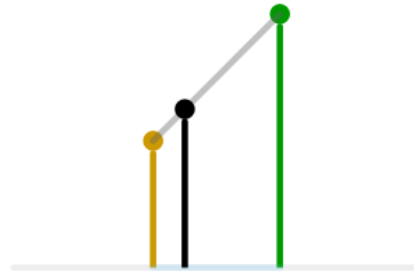
– Inverse mapping: $x = x' / 2$; $y = y'$



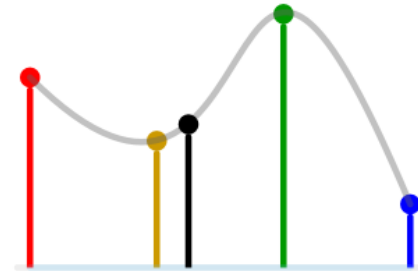
Different Interpolation Functions



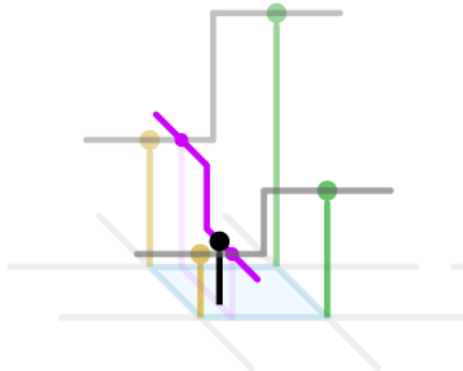
1D nearest-neighbour



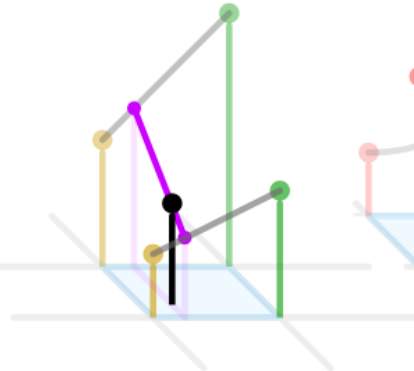
Linear



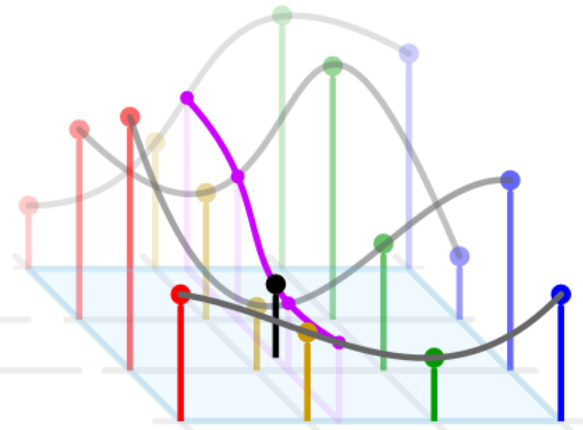
Cubic



2D nearest-neighbour



Bilinear



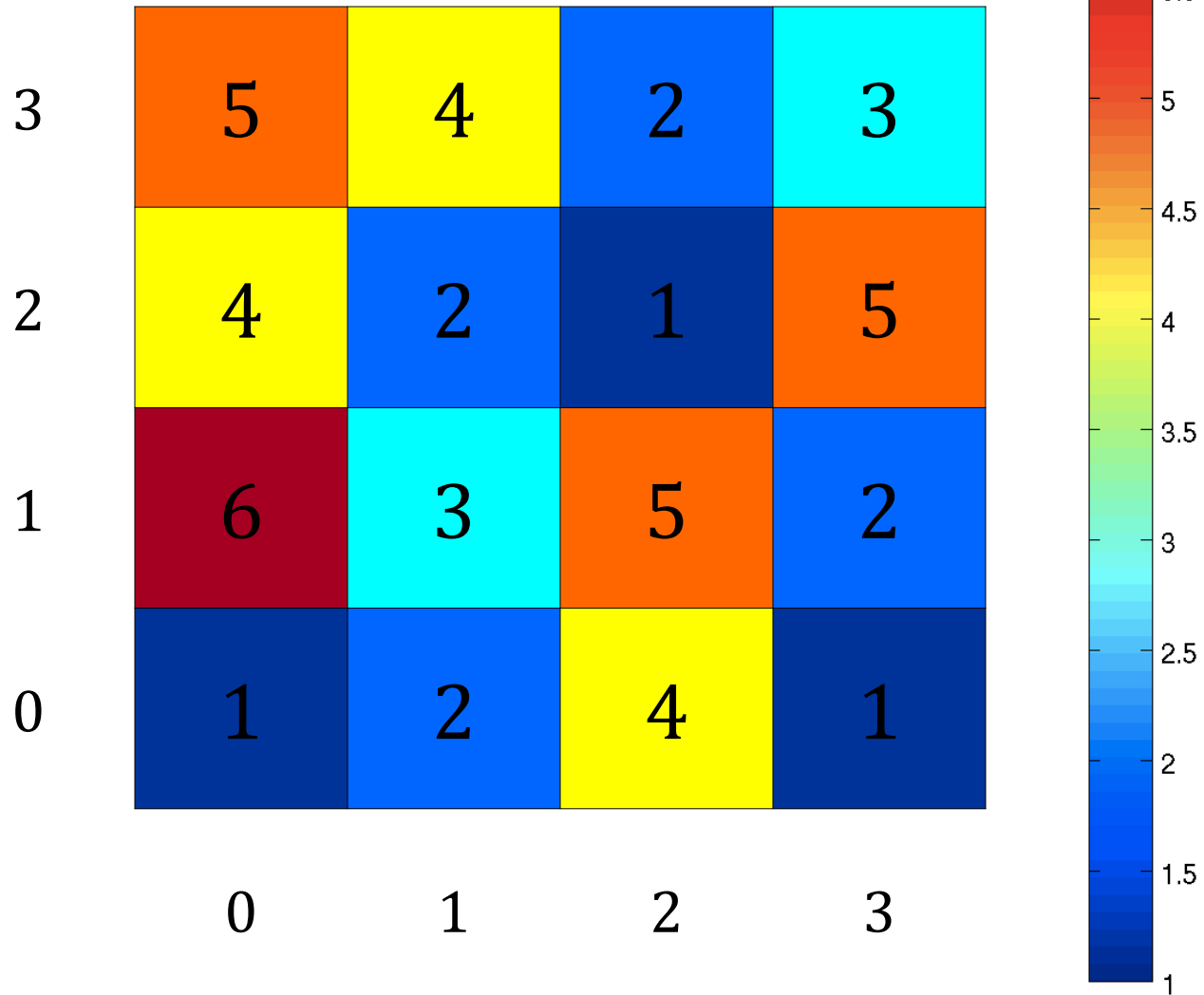
Bicubic

Image Interpolation Methods

- We study the following:
 1. Nearest neighbor
 2. Bilinear
 3. Bicubic

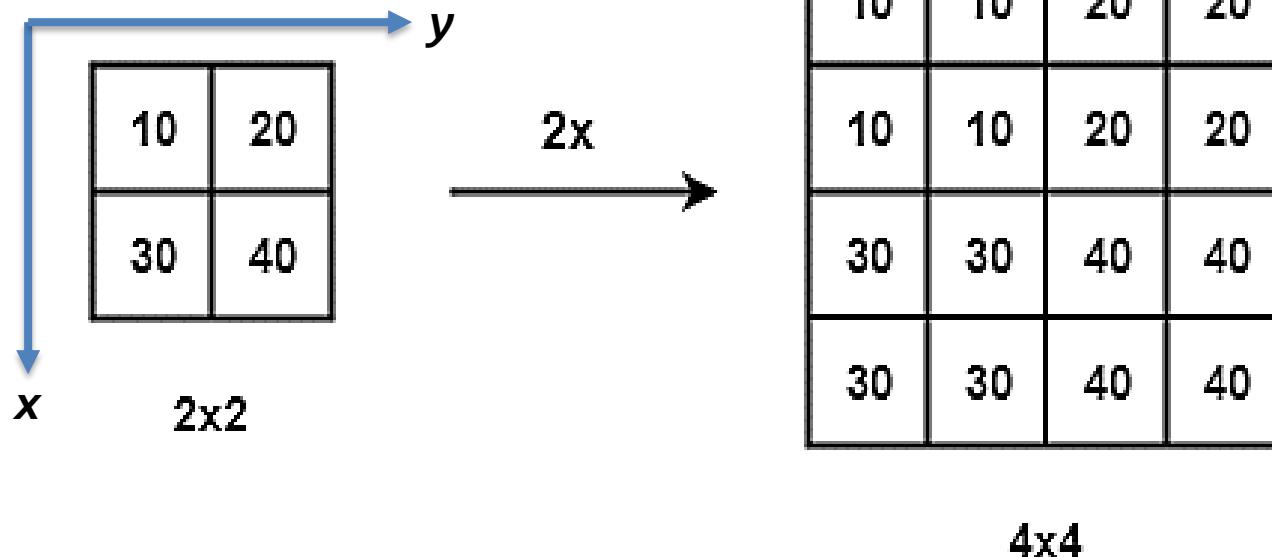
Example

$f(x, y) =$



Nearest Neighbor Interpolation

- *Nearest neighbor algorithm* selects the value of the nearest point (**closest integer coordinate neighbor to a point (x,y)**) and does not consider the values of all the neighboring points, yielding a piecewise-constant interpolant.



Nearest Neighbor Interpolation – mapping

- Assume:
 - Image I has size $M1 \times N1$
 - Image I' has size $M2 \times N2$
- Then $\mathbf{x}' = \mathbf{c}_x \mathbf{x}$ and $\mathbf{y}' = \mathbf{c}_y \mathbf{y}$ where \mathbf{c}_x , \mathbf{c}_y are the scaling factors:

$$c_x = M2/M1$$

$$c_y = N2/N1$$

- We actually want the inverse function:

$$x = x' / c_x$$

$$y = y' / c_y$$

$$I'(x', y') = I(\text{round}\{f_x^{-1}(x', y')\}, \text{round}\{f_y^{-1}(x', y')\})$$

Image $I(x, y)$



$M1 \times N1$

Image $I'(x', y')$



$M2 \times N2$

Nearest Neighbor Interpolation – mapping example

- Assume:
 - Image I1 has size 2×2
 - Image I2 has size 4×4
- Then $\mathbf{x}' = \mathbf{c}_x \mathbf{x}$ and $\mathbf{y}' = \mathbf{c}_y \mathbf{y}$ where \mathbf{c}_x , \mathbf{c}_y are the scaling factors:

$$c_x = 4/2 = 2$$

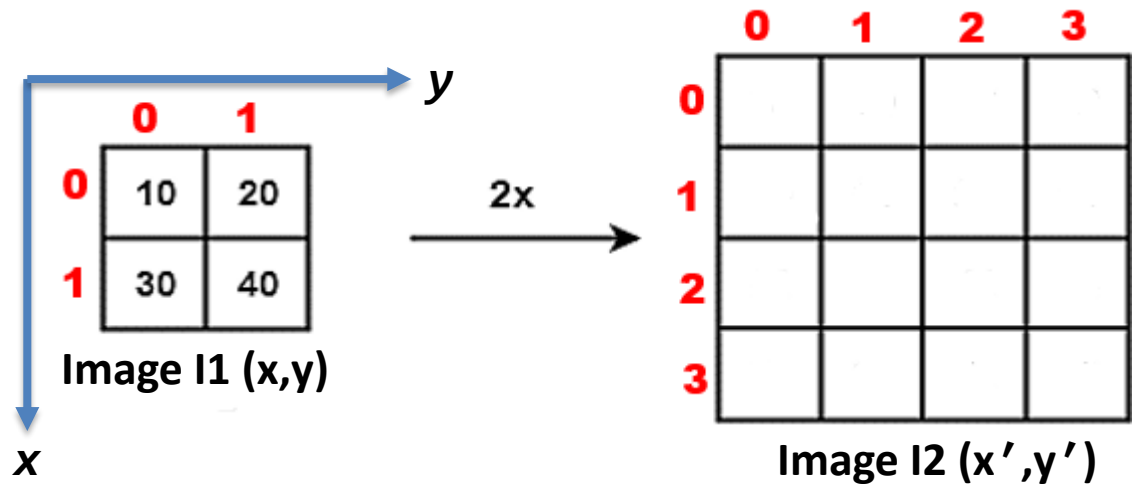
$$c_y = 4/2 = 2$$

$$I'(x', y') = I(\text{round} \{f_x^{-1}(x', y')\}, \text{round} \{f_y^{-1}(x', y')\})$$

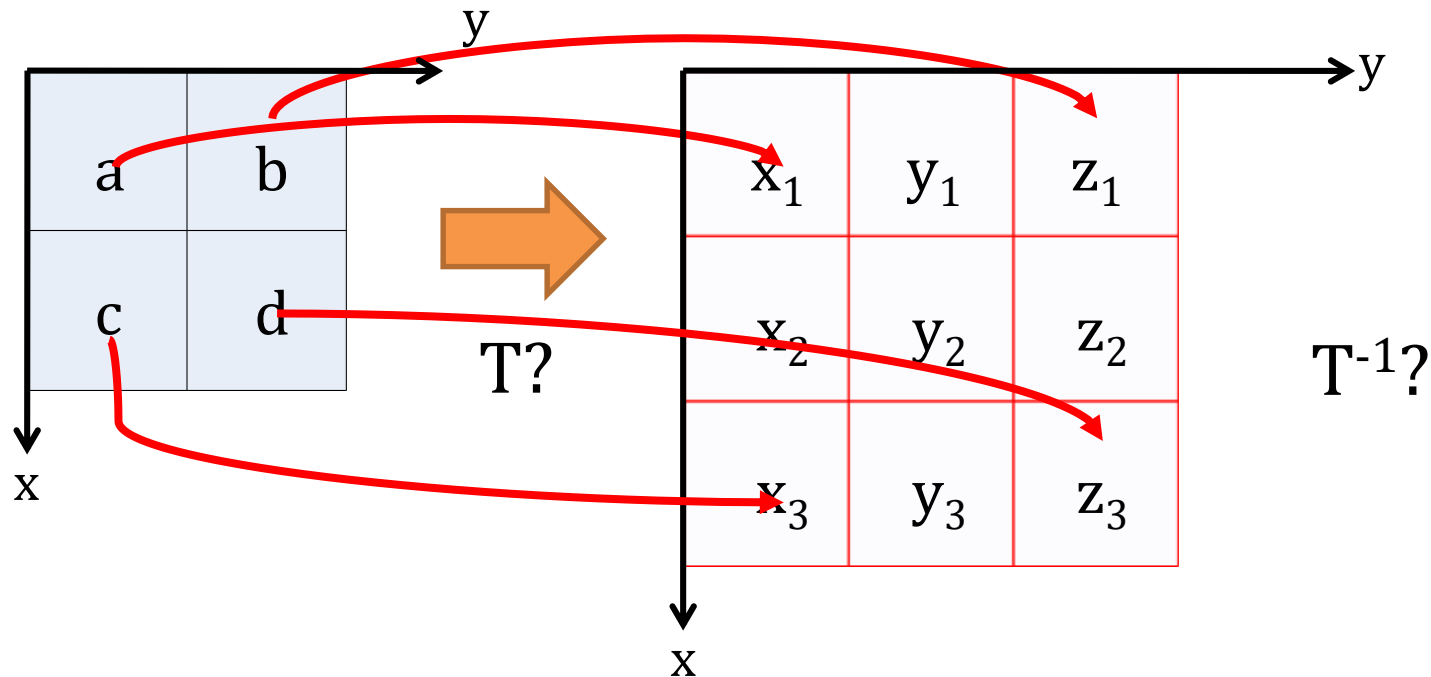
- The inverse function is:

$$x = x' / c_x$$

$$y = y' / c_y$$



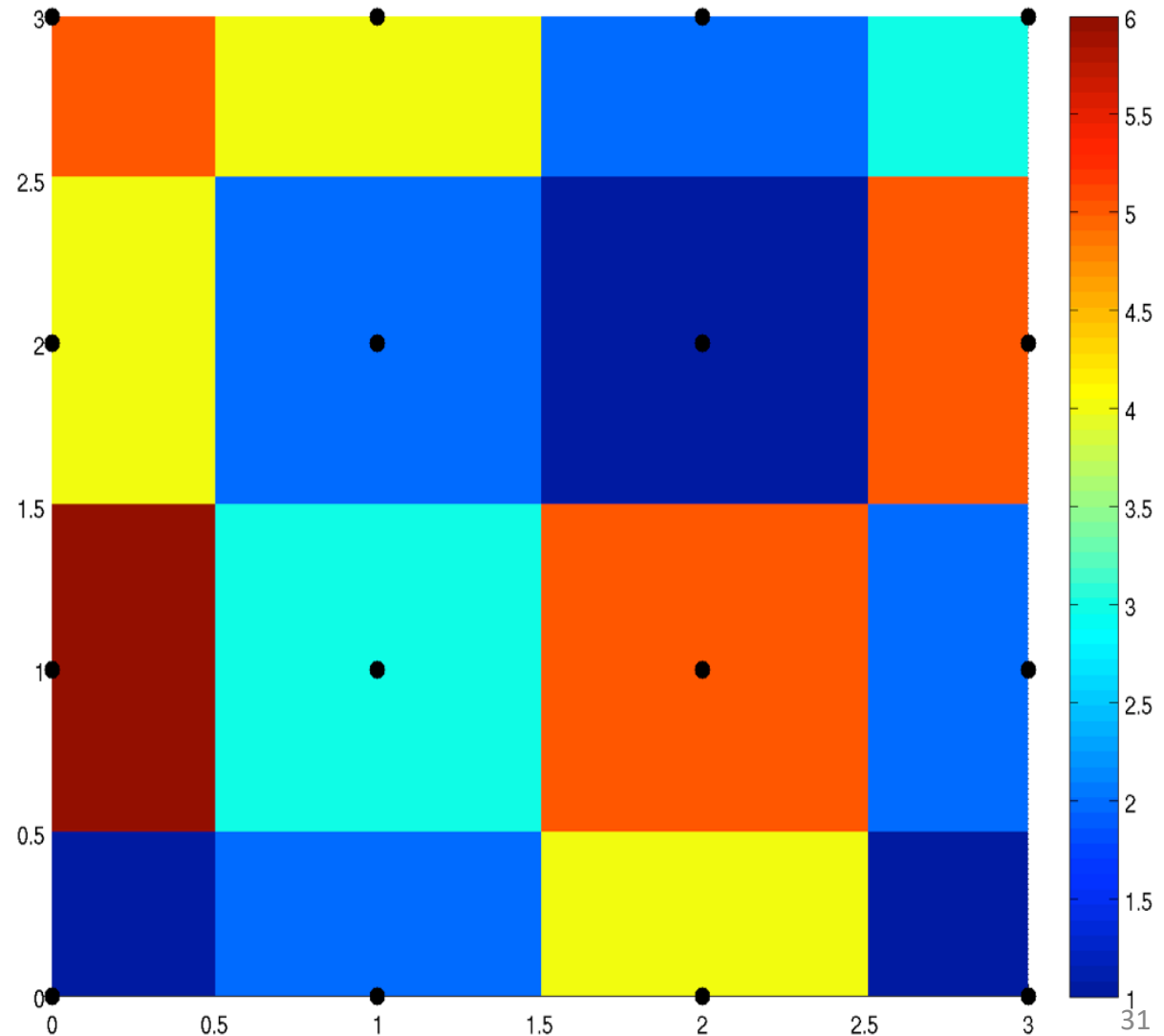
Nearest Neighbor Interpolation – mapping example



$x_1=a, y_1=a, z_1=b,$
 $x_2=a, y_2=a, z_2=b,$
 $x_3=c, y_3=c, z_3=d,$

Nearest Neighbor Interpolation – example

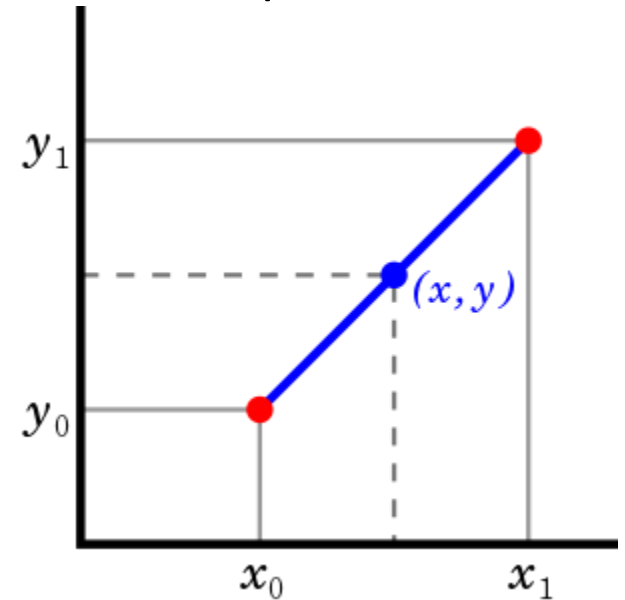
3	5	4	2	3
2	4	2	1	5
1	6	3	5	2
0	1	2	4	1
	0	1	2	3



Linear Interpolation

- If the two known points are given by the coordinates (x_0, y_0) and (x_1, y_1) , the linear interpolant is on the **straight line** between these points.
- For a value x in the interval (x_0, x_1) , the value y along the straight line is given from the **equation of slope**:

$$\frac{y - y_0}{x - x_0} = \frac{y_1 - y_0}{x_1 - x_0}$$

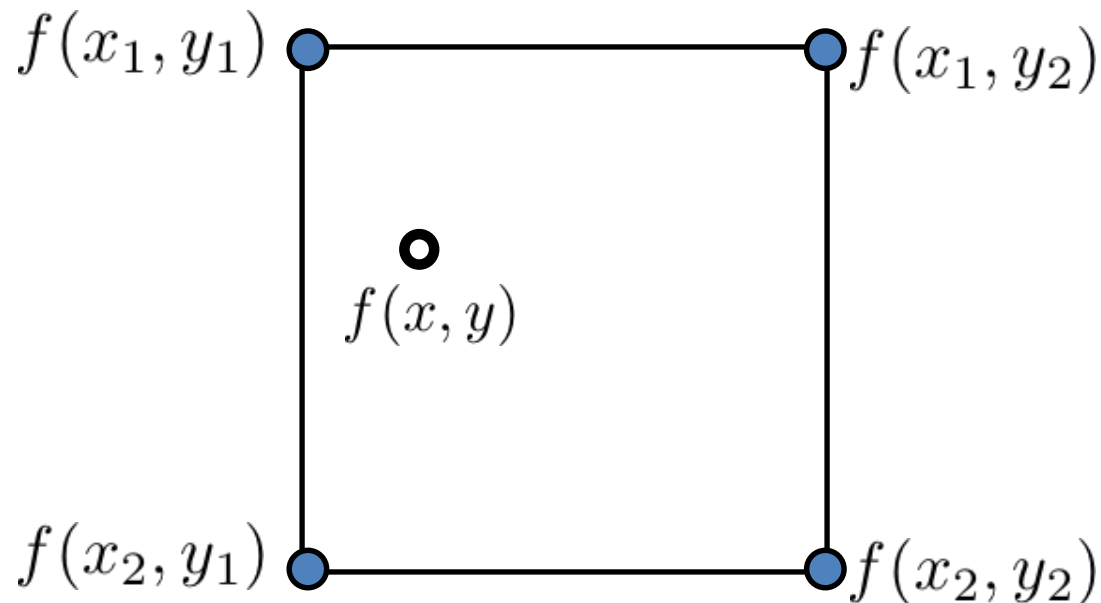


- Solving this equation for y , which is the unknown value at x , gives:

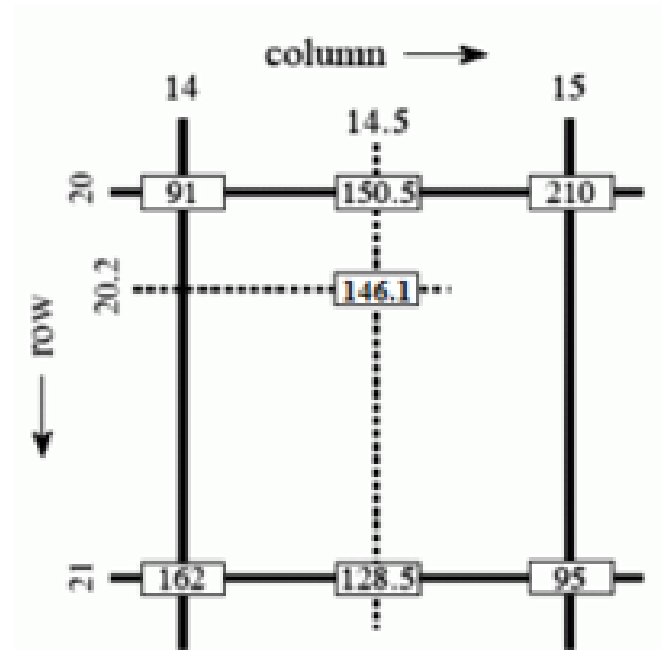
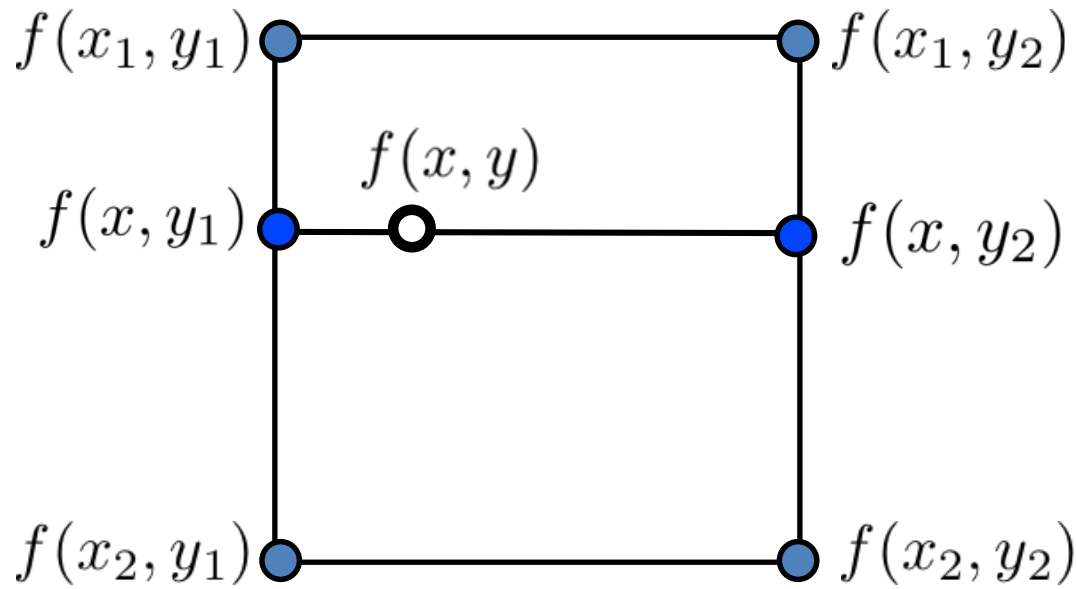
$$y = y_0 + (x - x_0) \frac{y_1 - y_0}{x_1 - x_0} = \frac{y_0(x_1 - x) + y_1(x - x_0)}{x_1 - x_0};$$

Bilinear Interpolation

- **Bilinear interpolation** is performed using linear interpolation first in one direction, and then again in the other direction.
- Bilinear interpolation is a **separable process**:
 - Can perform two 1D linear interpolations and compute the result.



Bilinear Interpolation (method-1)



Step 1: $\alpha = \frac{x - x_1}{x_2 - x_1} \quad \beta = \frac{y - y_1}{y_2 - y_1}$

Step 2: $f(x, y_1) = (1 - \alpha)f(x_1, y_1) + \alpha f(x_2, y_1)$

$$f(x, y_2) = (1 - \alpha)f(x_1, y_2) + \alpha f(x_2, y_2)$$

Step 3: $f(x, y) = (1 - \beta)f(x, y_1) + \beta f(x, y_2)$

Bilinear Interpolation (method-2)

- Let (x, y) denote the **coordinates** of the location to which we want to assign an intensity value, and let $f(x, y)$ denote that **intensity value**.
- For bilinear interpolation, the assigned value is obtained using the equation:

$$f(x, y) = ax + by + cxy + d$$

- where the **four coefficients** (a,b,c,d) are determined from the four equations in four unknowns that can be written using the *four* nearest neighbors of point (x, y) .

Bilinear Interpolation (method-2)

$$f(x, y) = ax + by + cxy + d$$

	10	11	12
20		N1	N2
21		x	N3
22		N4	

$$20a + 11b + 220c + 1 = 220$$

$$20a + 12b + 240c + 1 = 240$$

$$21a + 12b + 252c + 1 = 200$$

$$22a + 11b + 242c + 1 = 180$$

4 Unknown Calculator

20	w+	11	x+	220	y+	1	z=	220
20	w+	12	x+	240	y+	1	z=	240
21	w+	12	x+	252	y+	1	z=	200
22	w+	11	x+	242	y+	1	z=	180

Calculate

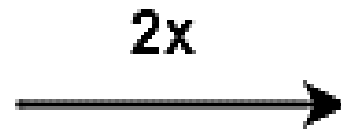
Reset

w	x	y	z
200	420	-20	-4000

Bilinear Interpolation

10	20
30	40

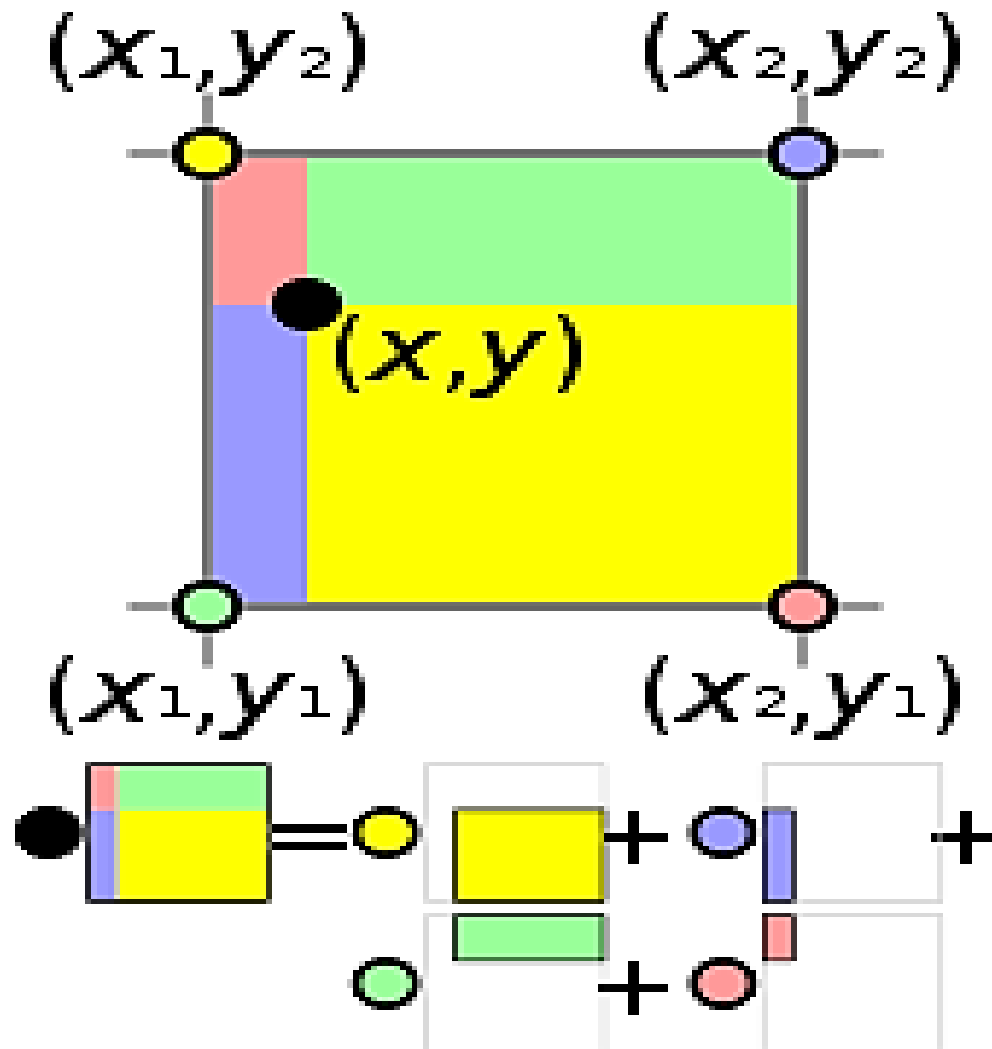
2x2



10	12	17	20
15	17	22	25
25	27	32	35
30	32	37	40

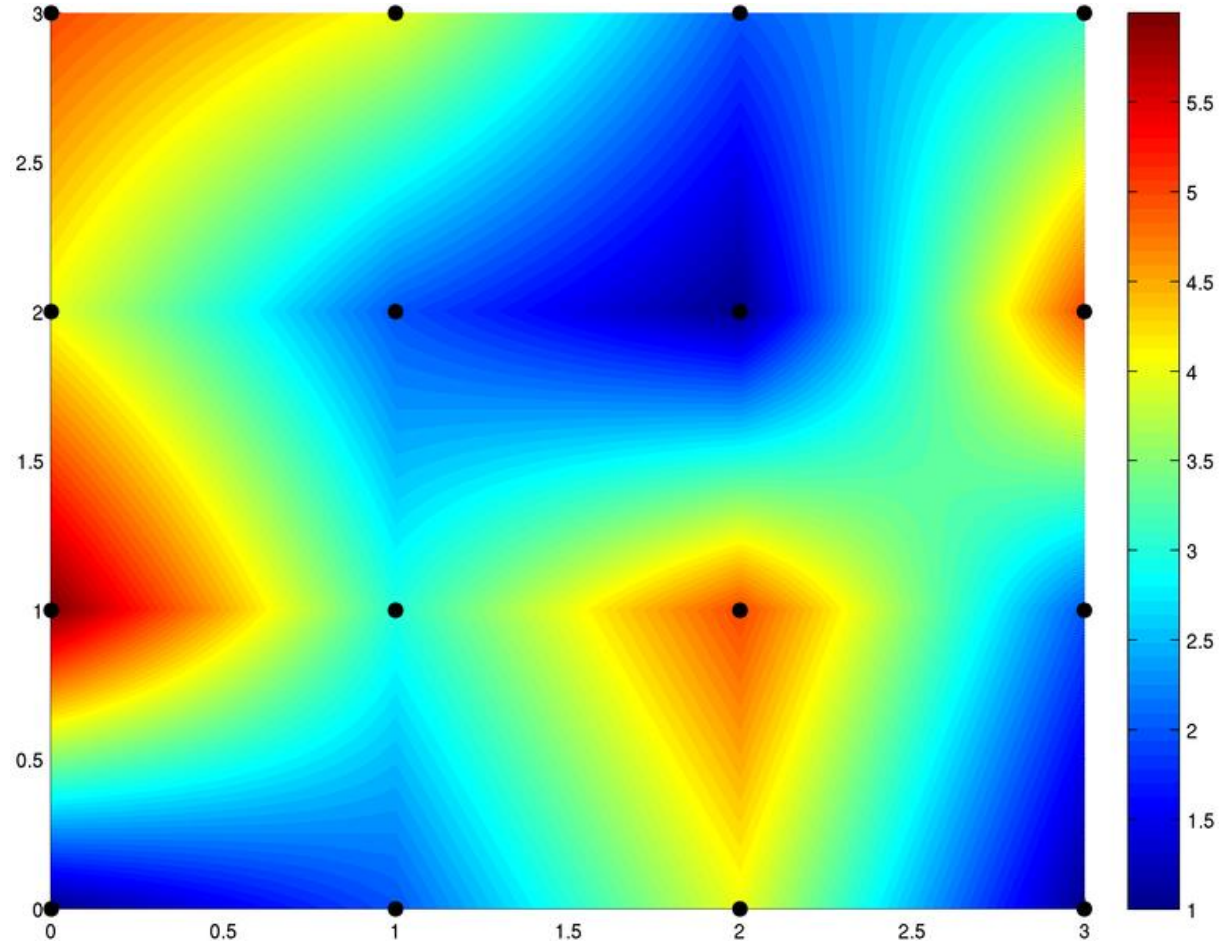
4x4

Bilinear Interpolation



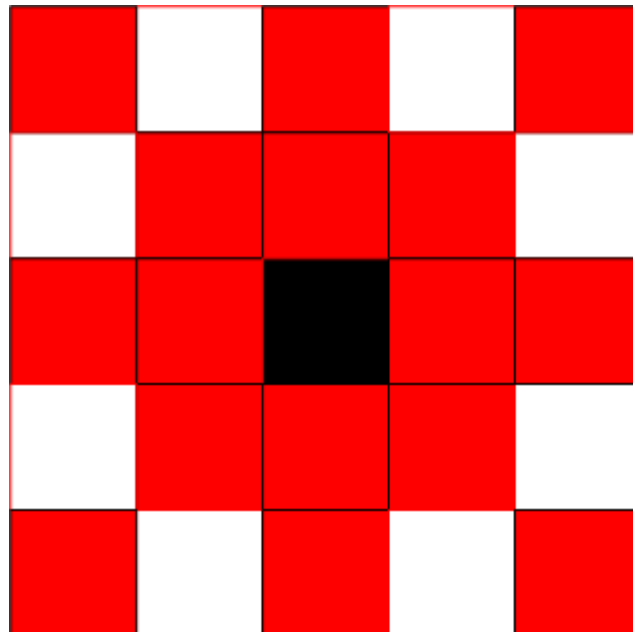
Bilinear Interpolation

3	5	4	2	3
2	4	2	1	5
1	6	3	5	2
0	1	2	4	1
	0	1	2	3



Bicubic Interpolation

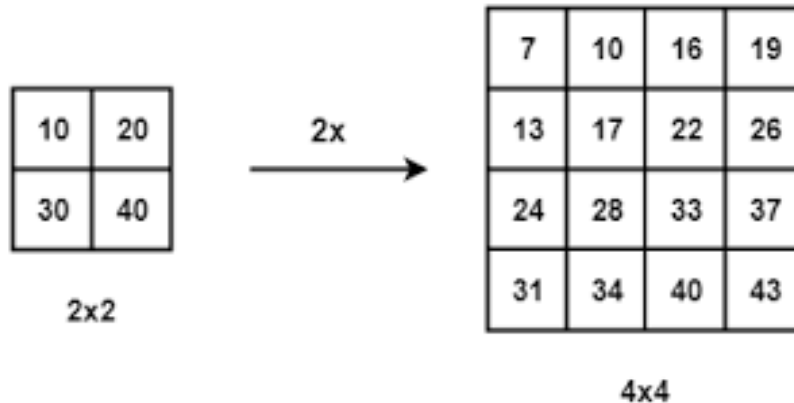
Bicubic interpolation goes one step beyond bilinear by considering the closest **4x4 neighborhood** of known pixels — for a total of **16** pixels.



Bicubic Interpolation

$$v(x, y) = \sum_{i=0}^3 \sum_{j=0}^3 a_{ij} x^i y^j$$

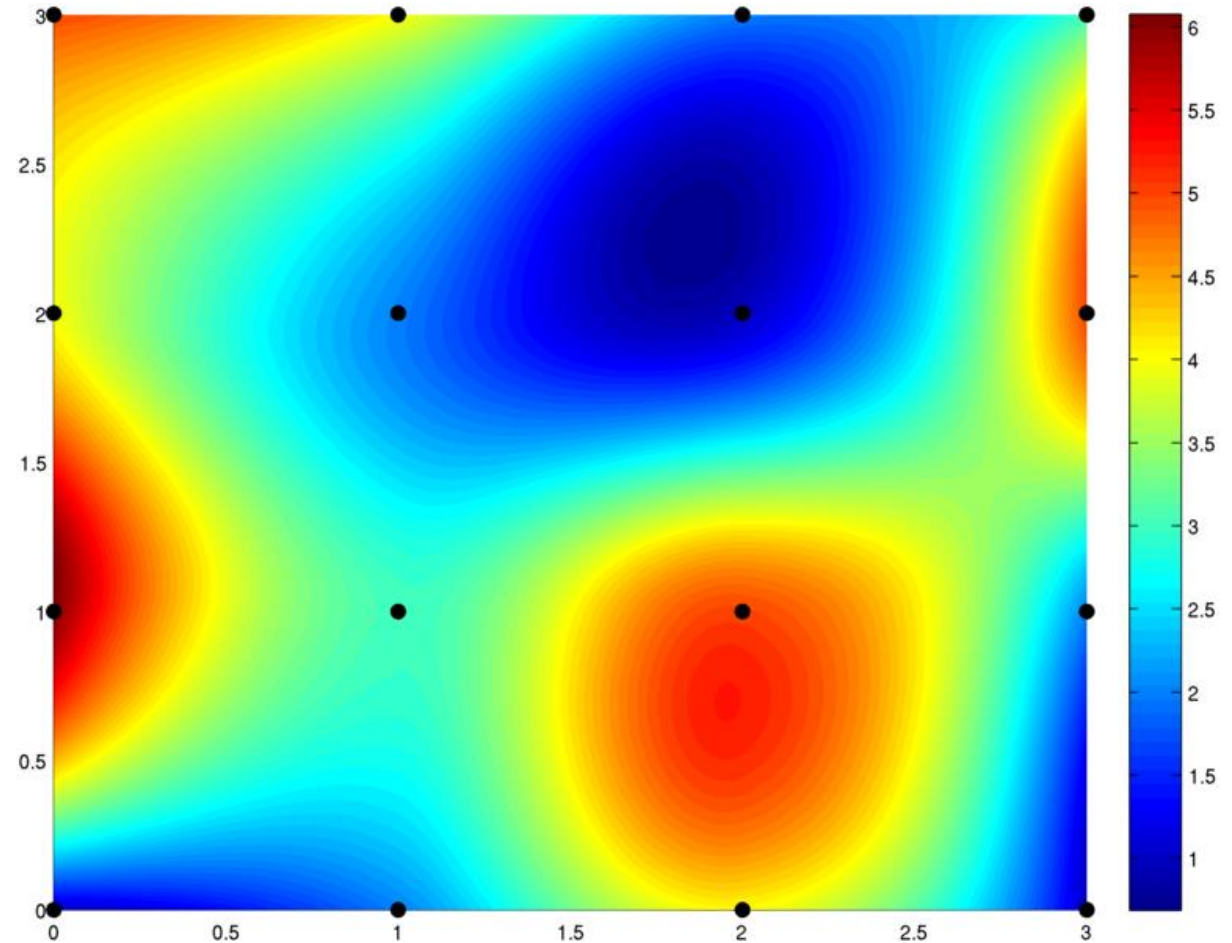
Where the **sixteen coefficients** are determined from the sixteen equations with sixteen unknowns that can be written using the sixteen nearest neighbors of point (x, y) .



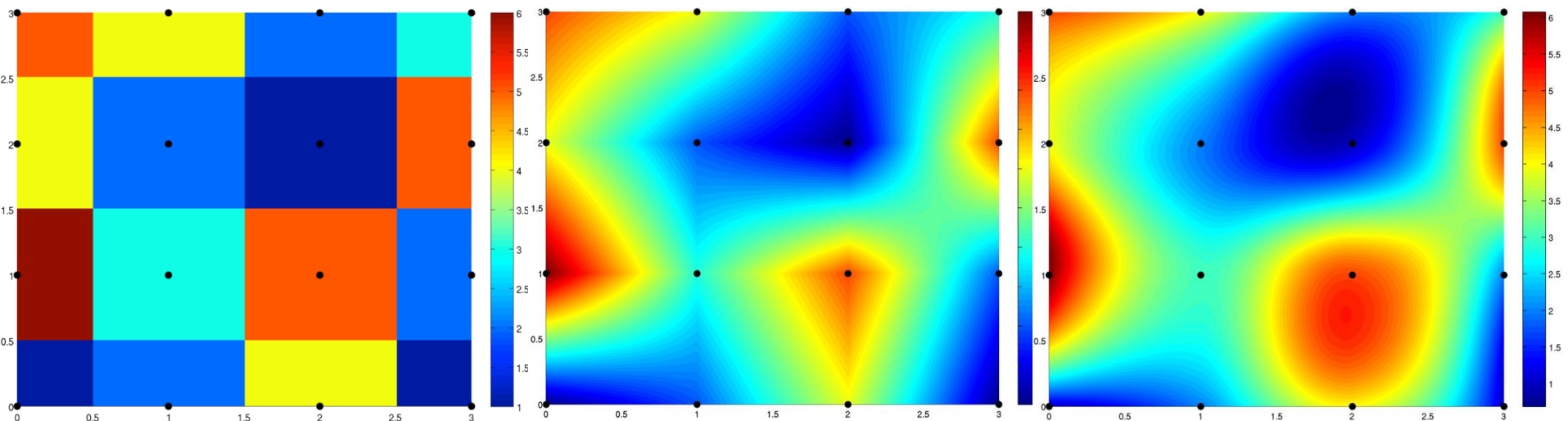
- https://en.wikipedia.org/wiki/Bicubic_interpolation
- <https://theailearner.com/2018/12/29/image-processing-bicubic-interpolation/>

Bicubic Interpolation

3	5	4	2	3
2	4	2	1	5
1	6	3	5	2
0	1	2	4	1
	0	1	2	3



Interpolations - comparison



Nearest neighbor

Bilinear

Bicubic

Image Interpolation



Nearest Neighbor



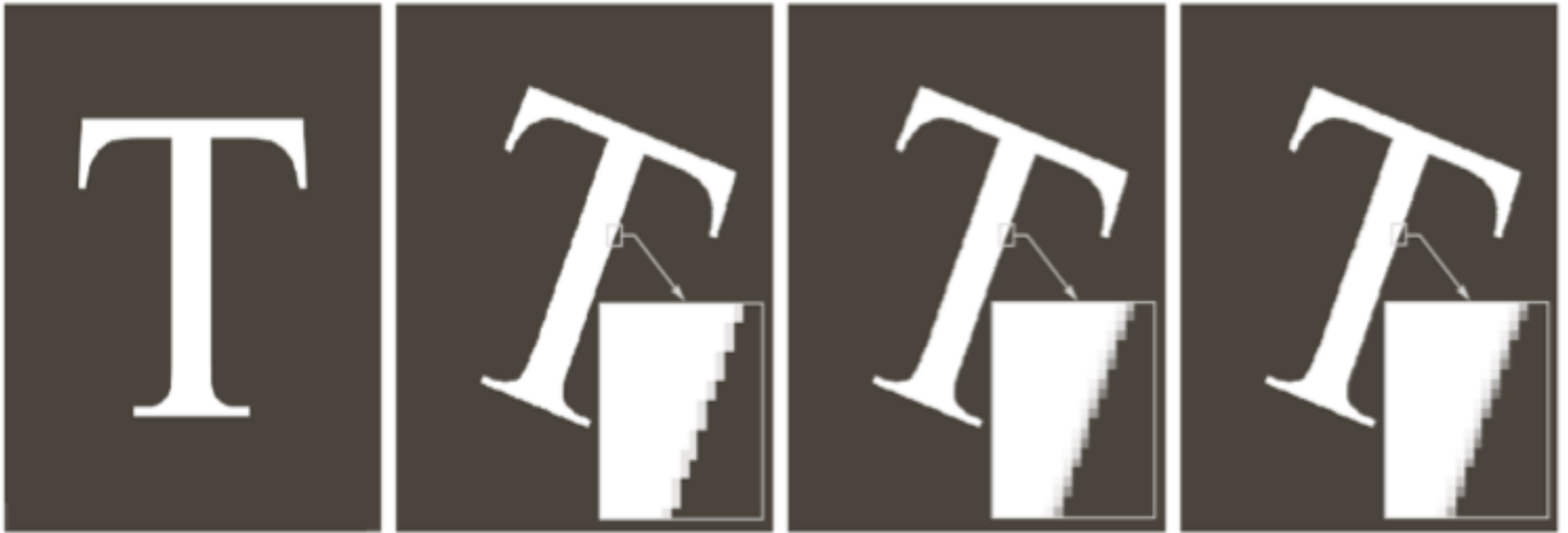
Bilinear



Bicubic

Image reduced to **213 × 162** and zoomed back to its original size: **3692 × 2812**

Image Interpolation



Nearest Neighbor

Bilinear

Bicubic

Rotated image with -21° angle

Image Registration

Image Registration

- Image registration in digital image processing is used to align two or more images of the same scene.
- In image registration, we have an *input image* and a *reference image*.
- **The objective is to transform the input image geometrically to produce an output image that is aligned (registered) with the reference image.**
- The transformation functions are not known and the geometric transformation needed to register the image must be estimated.

Image Registration

Example of image registration:

- Aligning two or more images taken at approximately the same time, but using different imaging systems, such as an **MRI** (magnetic resonance imaging) scanner and a **PET** (positron emission tomography) scanner.
- The images taken at different times using the same instruments, such as **satellite images** of a given location taken several days, months, or even years apart.

Image Registration

Selecting reference points for the alignment:

- The principal approach for selecting a reference point for the alignment is to use **tie points** (also called **control points**).
- These are corresponding points whose ***locations are known precisely*** in the input and reference images.
- Some imaging systems have ***physical artifacts*** which produce a set of known points (called ***reseau marks*** or ***fiducial marks***) directly on all images captured by the system. These known points can then be used as guides for establishing tie points.
- Approaches for selecting tie points range from selecting them ***interactively*** to ***using algorithms*** that detect these points automatically.

Image Registration

Estimating the transformation function:

- Suppose that we have a **set of four tie points** (8 tie points in total) each in an input and a reference image.
- Now, we can write eight equations and use them to solve for the eight unknown coefficients, **c1** through **c8**.
- A simple model based on a **bilinear approximation** for estimating the transformation function needed for image registration is given by:

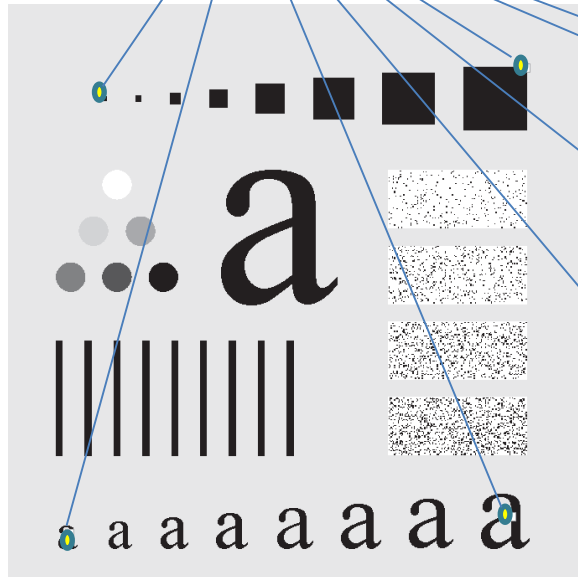
$$x = c_1v + c_2w + c_3vw + c_4$$

$$y = c_5v + c_6w + c_7vw + c_8$$

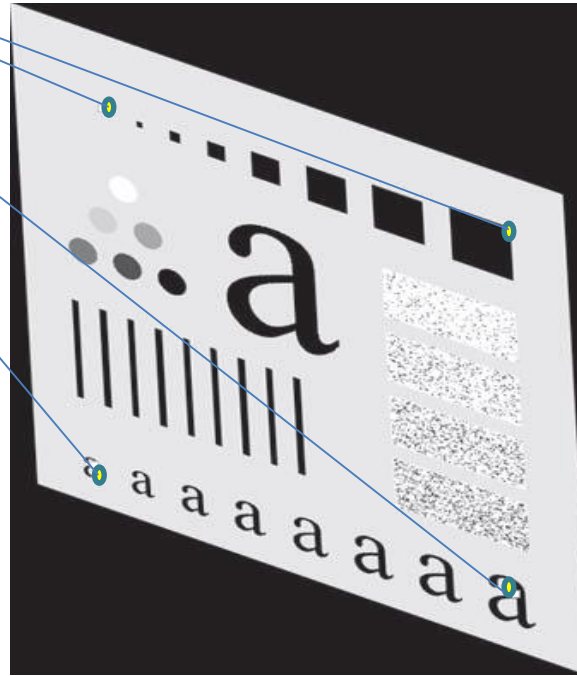
- After the coefficients have been computed, we let **(v,w)** denote the coordinates of each pixel in the **input image**, and **(x, y)** become the corresponding coordinates of the **output image**.
 - The same set of coefficients, c1 through c8 , are used in computing all coordinates (x, y); we just step through all (v,w) in the input image to generate the corresponding (x, y) in the output, registered image.

Image Registration

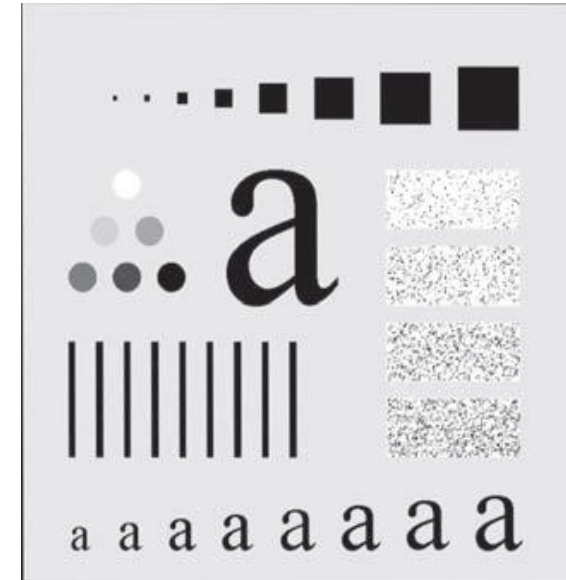
Tie points



Reference image (a)



Input image (b)



Registered image (c)



Difference b/w (a) & (c)

- Observe that *registration was not perfect*, as is evident by the black edges in the registered image. The difference image shows more clearly the slight lack of registration between the reference and corrected images.
- **The reason for the discrepancies is error in the manual selection of the tie points.**

Image Domain Transforms

Image Domain Transforms

- All the image processing approaches discussed thus far operate directly on the pixels of an input image.
 - that is, they work directly in the **spatial domain**.
- In some cases, image processing tasks are best formulated by transforming the input images to a ***non-spatial domain***, carrying the specified task in ***non-spatial domain (transform domain)***, and applying the ***inverse transform*** to return to the ***spatial domain***.

Image Domain Transforms

- A particularly important class of **2-D linear transforms**, denoted as $T(u, v)$, can be expressed in the general form:

$$T(u, v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y) r(x, y, u, v)$$

- where $f(x, y)$ is an input image, $r(x, y, u, v)$ is called a **forward transformation kernel**, and the above equation is evaluated for $u = 0, 1, 2, \dots, M - 1$ and $v = 0, 1, 2, \dots, N - 1$.
- x and y are **spatial variables**, while M and N are the **row** and **column** dimensions of input image $f(x, y)$.
- Variables u and v are called the **transform variables**.
- $T(u, v)$ is called the **forward transform** of $f(x, y)$.

Image Domain Transforms

- Given the *forward transform* $T(u, v)$:

$$T(u, v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y) r(x, y, u, v) \quad \text{..... (1)}$$

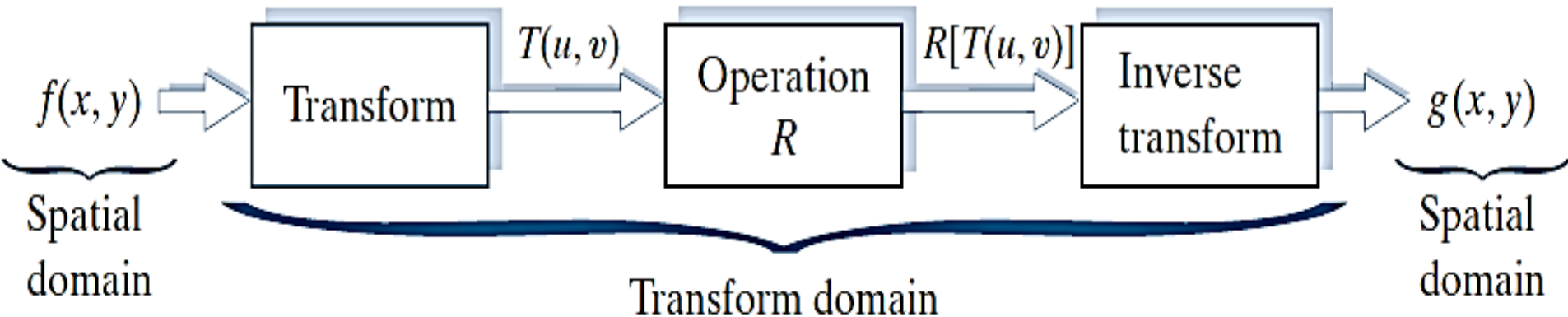
we can recover $f(x, y)$ using the *inverse transform* of $T(u, v)$ as:

$$f(x, y) = \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} T(u, v) s(x, y, u, v) \quad \text{..... (2)}$$

for $x = 0, 1, 2, \dots, M - 1$ and $y = 0, 1, 2, \dots, N - 1$, where $s(x, y, u, v)$ is called an *inverse transformation kernel*.

- Together, Eqs. (1) and (2) are called a ***transform pair***.

Image Domain Transforms



**Basic steps for performing image processing in the
Linear Transform Domain**

Image Domain Transforms

- **Separable** forward transformation kernel:
 - $r(x, y, u, v) = r_1(x, u) r_2(y, v)$
- **Symmetric** forward transformation kernel:
 - $r(x, y, u, v) = r_1(x, u) r_1(y, v)$
- The nature of a transform is determined by its kernel.
- *Identical comments apply to the inverse kernel.*

Image Domain Transforms

- A transform of particular importance in digital image processing is the ***Fourier transform***, which has the following **forward** and **inverse** kernels:

Forward transform kernel

$$r(x, y, u, v) = e^{-j2\pi(ux/M + vy/N)}$$

Inverse transform kernel

$$s(x, y, u, v) = \frac{1}{MN} e^{j2\pi(ux/M + vy/N)}$$

- where $j = \sqrt{-1}$, so these kernels are complex functions

Image Domain Transforms

- Substituting the preceding **Fourier Transform kernels** into the general transform formulations in Eqs. (1) and (2) of slide number 56 gives us the **discrete Fourier transform pair**:

$$T(u, v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y) e^{-j2\pi(ux/M + vy/N)}$$

$$f(x, y) = \frac{1}{MN} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} T(u, v) e^{j2\pi(ux/M + vy/N)}$$

- Fourier kernels** are both **separable** and **symmetric**.

Image Domain Transforms

- Substituting the preceding **Fourier Transform kernels** into the general transform formulations in Eqs. (1) and (2) of slide number 45 gives us the **discrete Fourier transform pair**:

$$T(u, v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y) r(x, y, u, v) \quad \text{..... (1)}$$

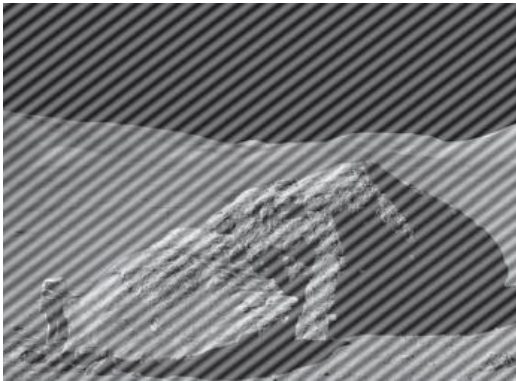
$$T(u, v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y) e^{-j2\pi(ux/M + vy/N)}$$

$$f(x, y) = \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} T(u, v) s(x, y, u, v) \quad \text{..... (2)}$$

$$f(x, y) = \frac{1}{MN} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} T(u, v) e^{j2\pi(ux/M + vy/N)}$$

Image Domain Transforms

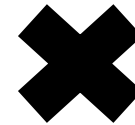
Spatial Domain Input



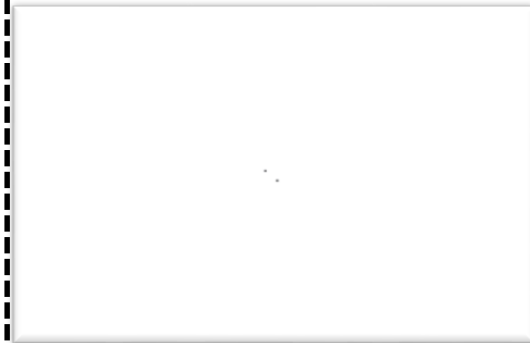
$$r(x, y, u, v) = e^{-j2\pi(ux/M + vy/N)}$$



Fourier
Domain
Image



$$s(x, y, u, v) = \frac{1}{MN} e^{j2\pi(ux/M + vy/N)}$$



Fourier
Domain
Binary
Mask

Back to Spatial Domain

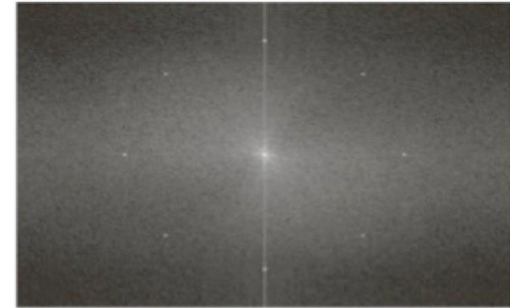


Image Domain Transforms

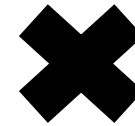
Spatial Domain Input



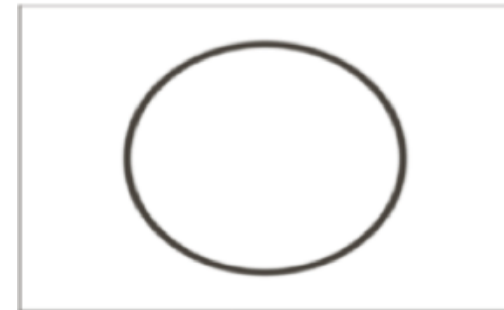
$$r(x, y, u, v) = e^{-j2\pi(ux/M + vy/N)}$$



Fourier Domain Image



$$s(x, y, u, v) = \frac{1}{MN} e^{j2\pi(ux/M + vy/N)}$$



Fourier Domain Binary Mask



Back to Spatial Domain

Removal of periodic noise from images using Fourier Transform

Next Lecture

- Spatial Domain Transformations - preview
- The Transformation Operator
- Intensity Transformations – examples
 - Contrast stretching
 - Image thresholding
- Basic Intensity Transformation Functions
 - Basic transforms
 - Piecewise-linear transformations