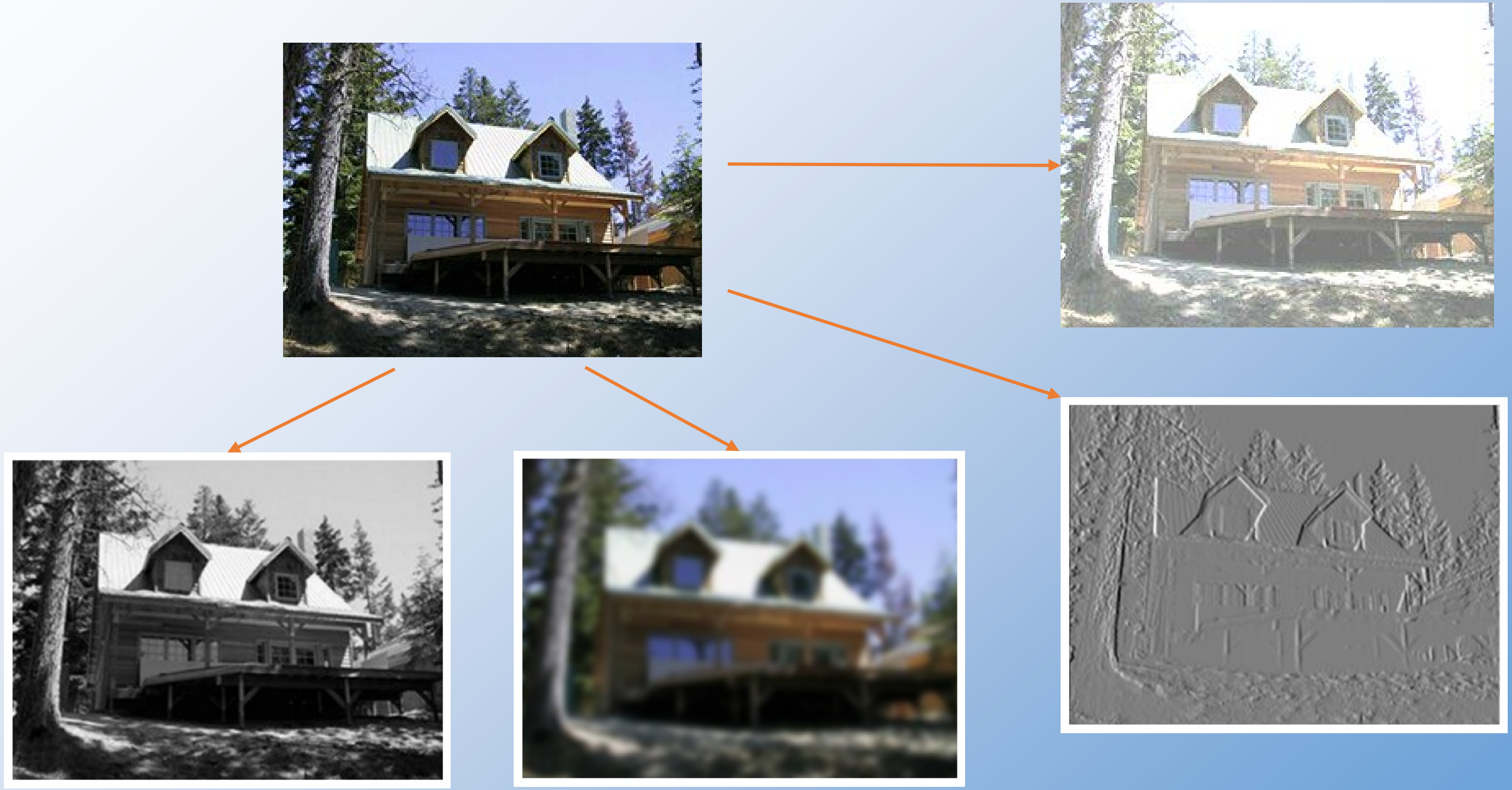
Geometric Transformations

Olac Fuentes
Computer Science Department
University of Texas at El Paso

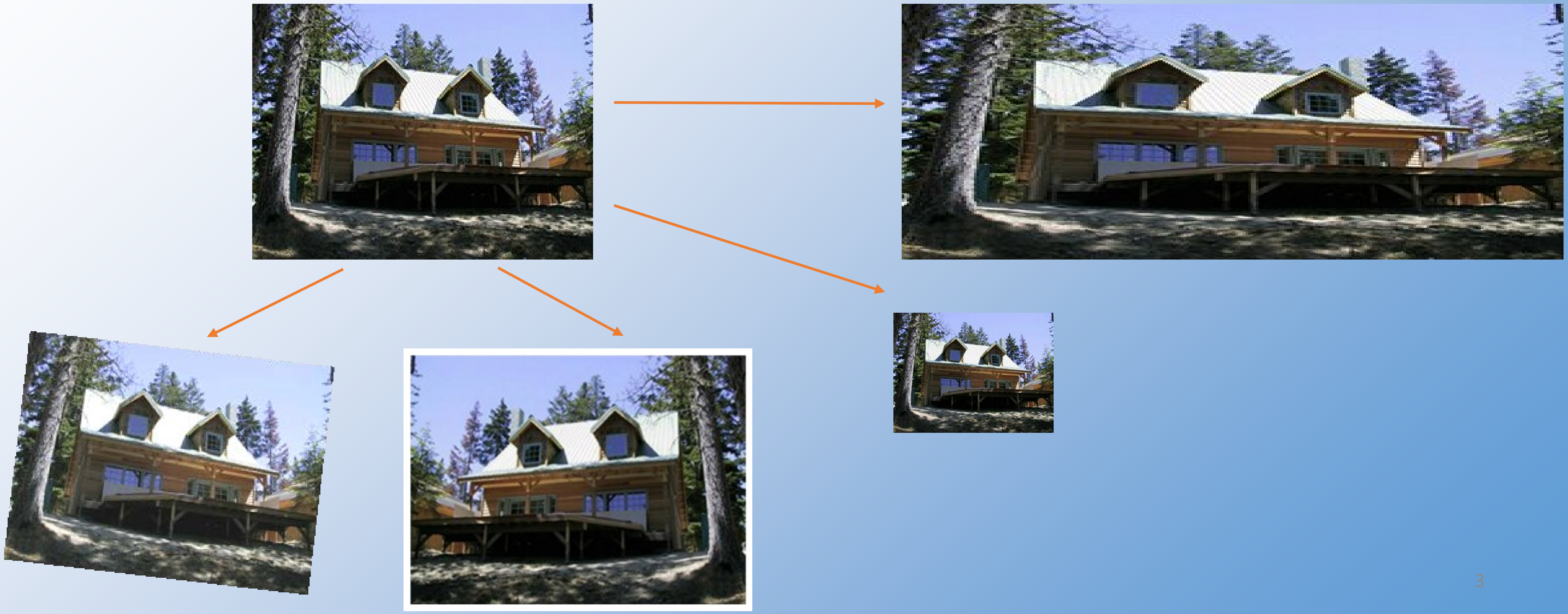
Intensity transformations

Intensity transformations deal with manipulation of pixel **values**



Geometric transformations

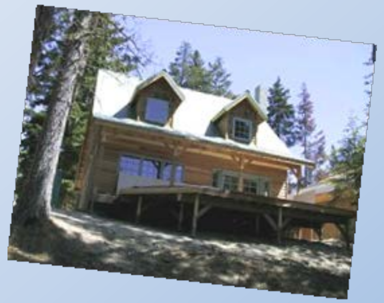
Geometric transformations deal with manipulation of pixel **coordinates**

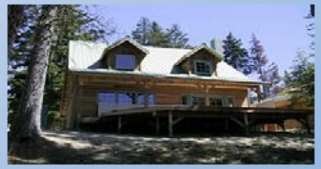


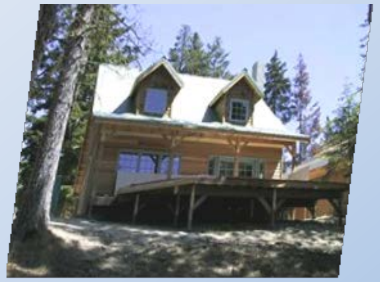
Geometric transformations

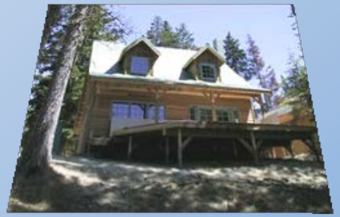
Examples:

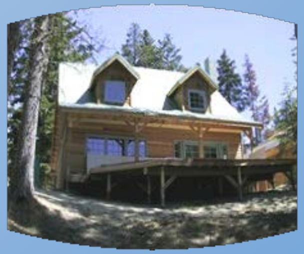
translation


rotation

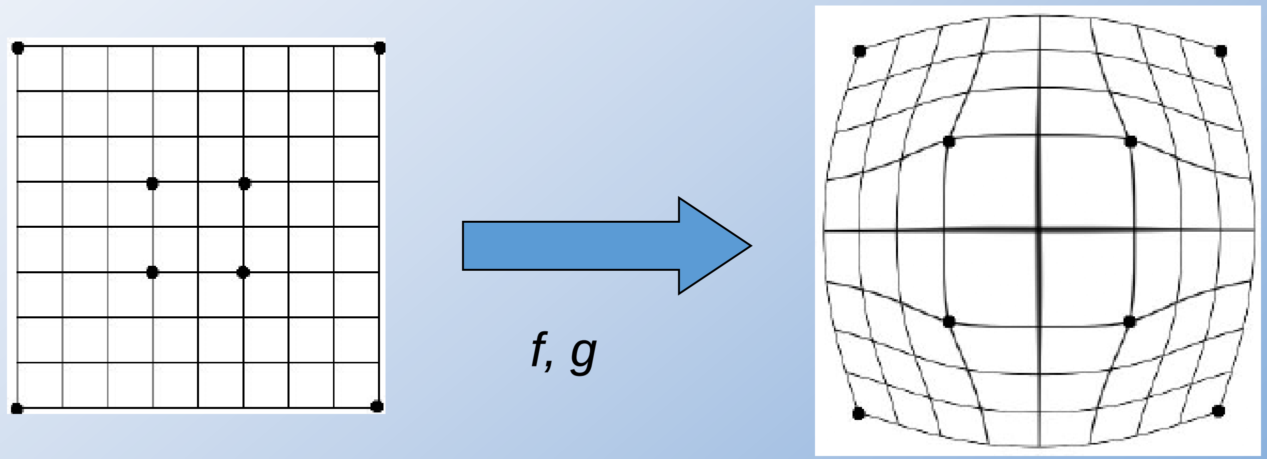

aspect


affine


perspective


cylindrical

Transformation Function



Transform the geometry of an image to a desired geometry

Definition: Image Warping

Source Image: Image to be used as the reference. The geometry of this image is no changed

Target Image: this image is obtained by transforming the reference image.

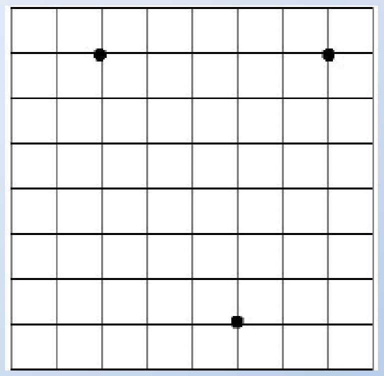
(x,y): coordinates of points in the reference image

(u,v): coordinates of points in the target image

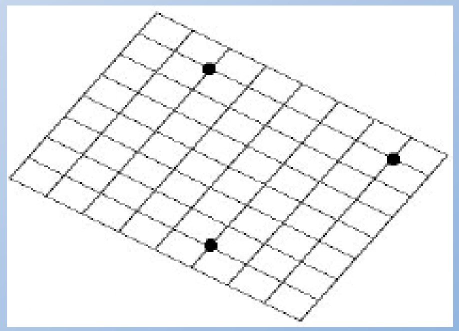
f,g or F,G: x and y components of a transformation function

Definition: Image Warping

Control points: Unique points in the reference and target images. The coordinates of corresponding control points in images are used to determine a transformation function.



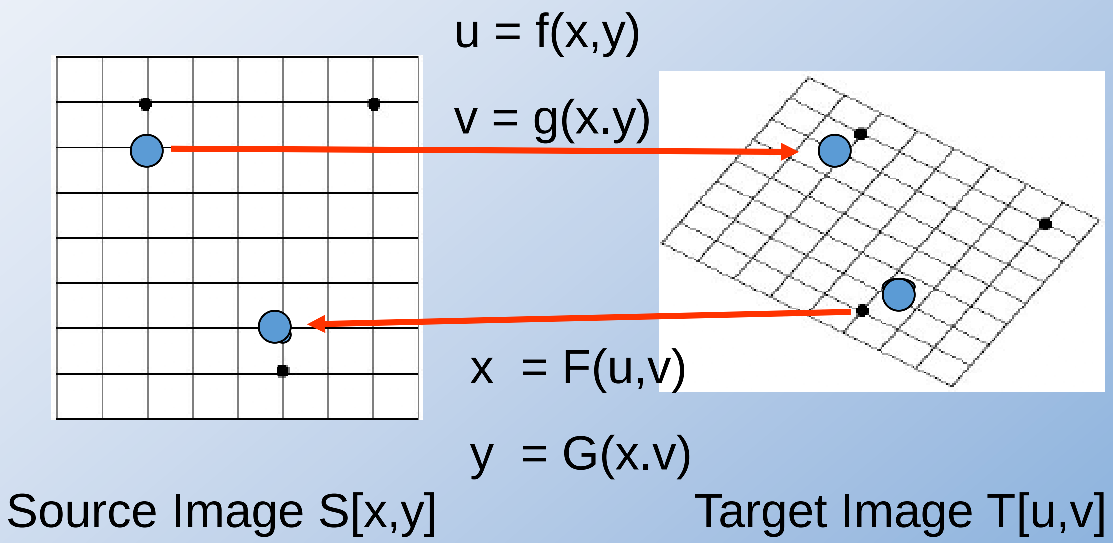
Source Image



Target Image

A Transformation Function

Used to compute the corresponding points

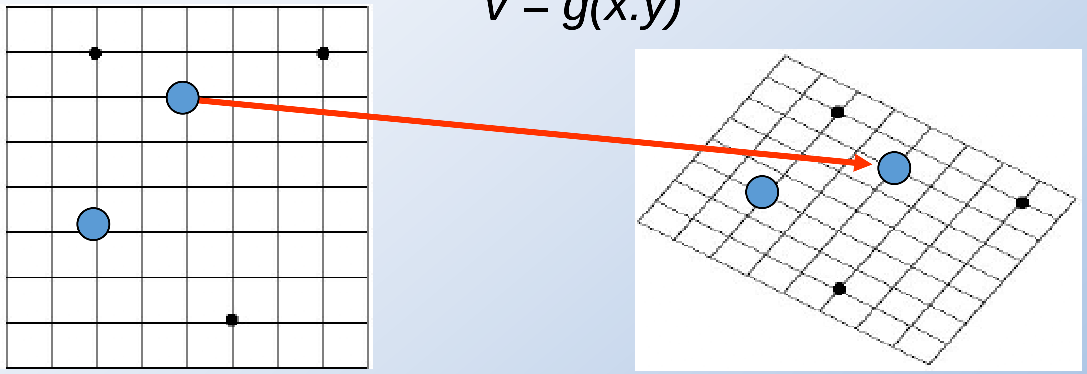


Forward Transformation

Used to compute the corresponding points

$u = f(x,y)$

$v = g(x.y)$



Source Image S[x,y]

Target Image T[u,v]

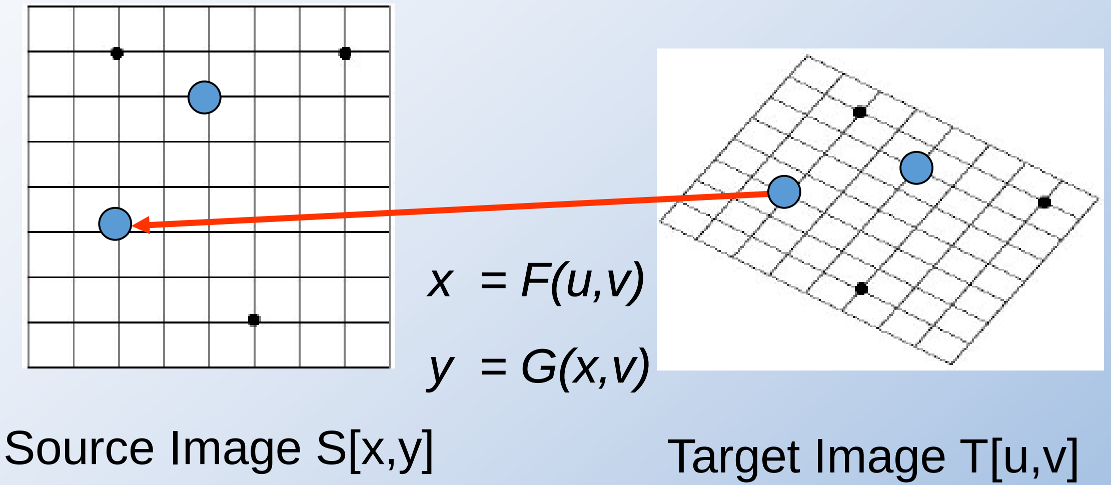
For every pixel location (x,y) in the source image S, find its corresponding location in the target image T:

```
for every x
  for every y
      T[f(x,y),g(x,y)] = S[x,y]
```

That is, find the ***destination*** in T of every pixel in S

Backward Transformation

Used to compute the corresponding points



For every pixel location (u,v) in the target image T, find its corresponding location in the source image S:

```
for every u
  for every v
      T[u,v] = S[F(u,v),G(u,v)]
```

That is, find the ***source*** in S of every pixel in T

Forward vs. Backward Transformation

Forward Transformation

For every pixel location (x,y) in the source image S, find its corresponding location in the target image T:

```
for every x
  for every y
      T[f(x,y),g(x,y)] = S[x,y]
```

That is, find the ***destination*** in T of every pixel in S

Backward Transformation

For every pixel location (u,v) in the target image T, find its corresponding location in the source image S:

```
for every u
  for every v
      T[u,v] = S[F(u,v),G(u,v)]
```

That is, find the ***source*** in S of every pixel in T

Forward vs. Backward Transformation

Forward Transformation

For every pixel location (x,y) in the source image S, find its corresponding location in the target image T:

```
for every x
  for every y
      T[f(x,y),g(x,y)] = S[x,y]
```

That is, find the ***destination*** in T of every pixel in S

Backward Transformation

For every pixel location (u,v) in the target image T, find its corresponding location in the source image S:

```
for every u
  for every v
      T[u,v] = S[F(u,v),G(u,v)]
```

That is, find the ***source*** in S of every pixel in T

Which one is better?

Forward vs. Backward Transformation

Forward Transformation

For every pixel location (x,y) in the source image S, find its corresponding location in the target image T:

```
for every x
  for every y
      T[f(x,y),g(x,y)] = S[x,y]
```

That is, find the ***destination*** in T of every pixel in S

Backward Transformation

For every pixel location (u,v) in the target image T, find its corresponding location in the source image S:

```
for every u
  for every v
      T[u,v] = S[F(u,v),G(u,v)]
```

That is, find the ***source*** in S of every pixel in T

With forward transformation, we may (usually will) have gaps on target image and multiple pixels in the source mapping to the same pixel in the destination.

Forward vs. Backward Transformation

Forward Transformation

For every pixel location (x,y) in the source image S, find its corresponding location in the target image T:

```
for every x
  for every y
      T[f(x,y),g(x,y)] = S[x,y]
```

That is, find the ***destination*** in T of every pixel in S

Backward Transformation

For every pixel location (u,v) in the target image T, find its corresponding location in the source image S:

```
for every u
  for every v
      T[u,v] = S[F(u,v),G(u,v)]
```

That is, find the ***source*** in S of every pixel in T

With forward transformation, we may (usually will) have gaps on target image and multiple pixels in the source mapping to the same pixel in the destination.

Backward Transformation is preferable

Technical Aside - Interpolation

Since S and T are images, the values of S[x,y] and T[u,v] are given only for integer values of x, y, u, and v.

However, in most cases, the transformation functions f(x,y), g(x,y), F(u,v) and G(u,v) will not return integer values.

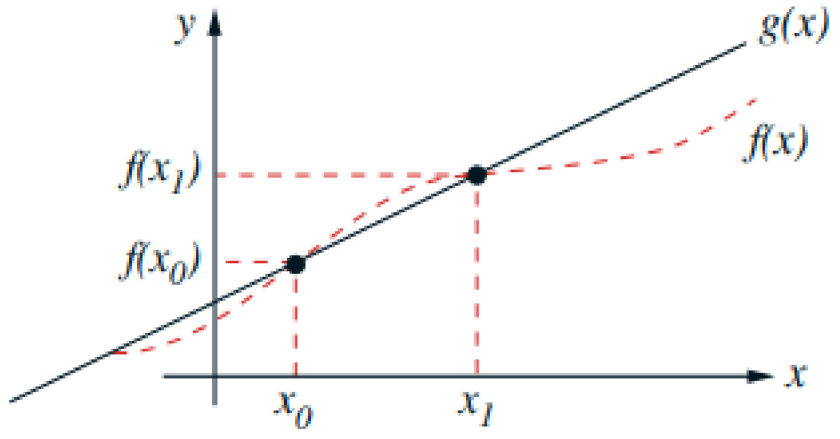
How can we compute T[f(x,y),g(x,y)] or S[F(u,v),G(u,v)] in this case?

Answer: through interpolation

Technical Aside - Interpolation

Linear Interpolation

- *Linear interpolation is obtained by passing a straight line between 2 data points*



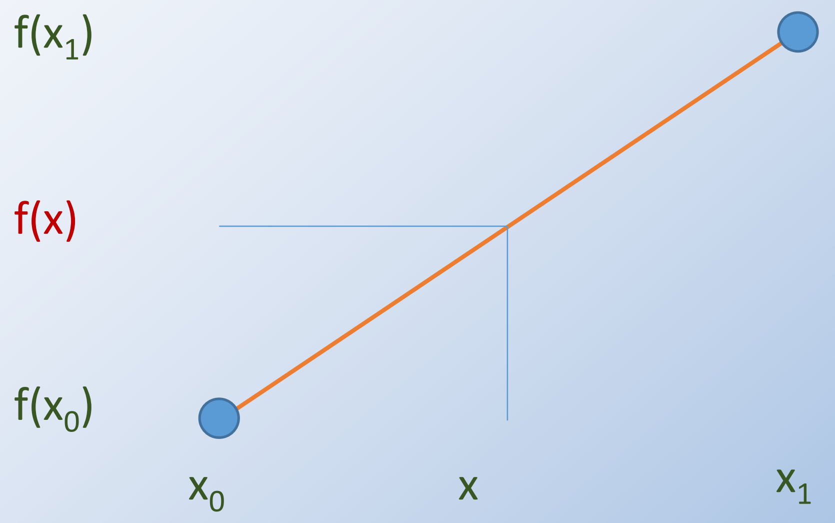
$f(x)$ = the exact function for which values are known only at a discrete set of data points

$g(x)$ = the interpolated approximation to $f(x)$

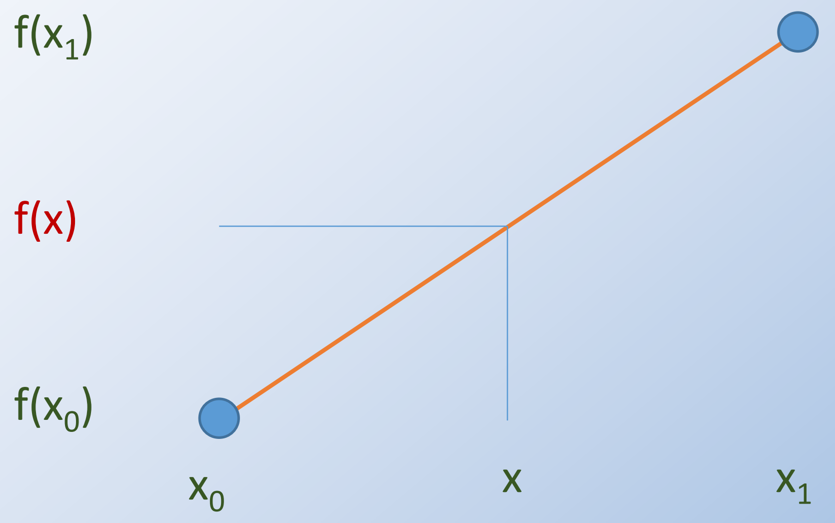
x_0 , x_1 = the data points (also referred to as interpolation points or nodes)

Linear Interpolation

Known
unknown



Linear Interpolation



Known
unknown

The tangent of the orange line and the x axis is given by:

$$\frac{f(x_1)-f(x_0)}{x_1-x_0}$$

And also by:

$$\frac{f(x)-f(x_0)}{x-x_0}$$

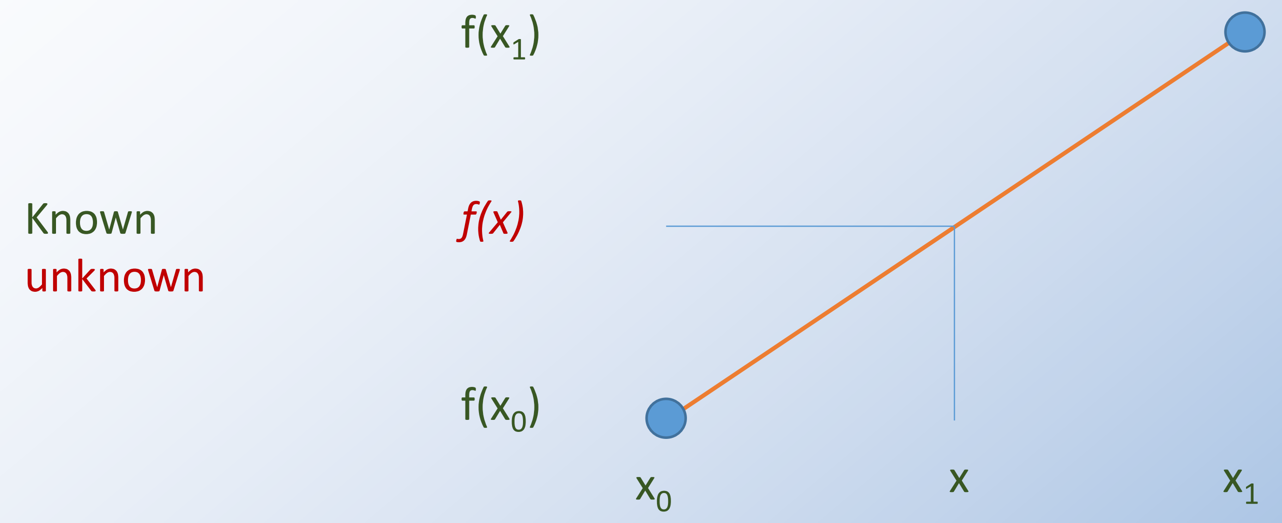
Thus:

$$\frac{f(x_1)-f(x_0)}{x_1-x_0} = \frac{f(x)-f(x_0)}{x-x_0}$$

Then:

$$f(x) = f(x_0) + \frac{(x-x_0)\,(f(x_1)-f(x_0))}{x_1-x_0}$$

Linear Interpolation



Another view
f(x) is the weighted average of *f(x₀)* and *f(x₁)* with weights proportional to the distance from *x* to *x₀* and *x₁*.

$$f(x) = w_0 f(x_0) + \ w_1 f(x_1)$$

where:

$$w_i' = \frac{1}{|x \ - xi|}$$

and

$$w_i = \frac{w_i'}{w_0' + w_1'}$$

Linear Interpolation

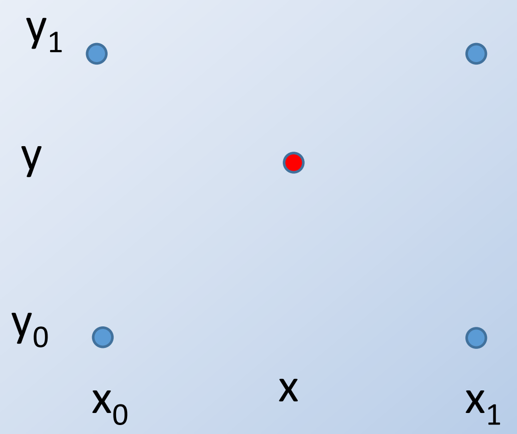
For 1D interpolation, we will use the numpy implementation:

```
xp = [1,2,3]
yp=  [3,6,9]
x = [1,1.5,2,2.5,3]
y = np.interp(x, xp,  yp)
print(y)
-> [3.0, 4.5, 6.0, 7.5, 9.0]
```

Bi-linear Interpolation

In two dimensions, we will use bilinear interpolation, which consists of applying linear interpolation first along one dimension and then along the other.

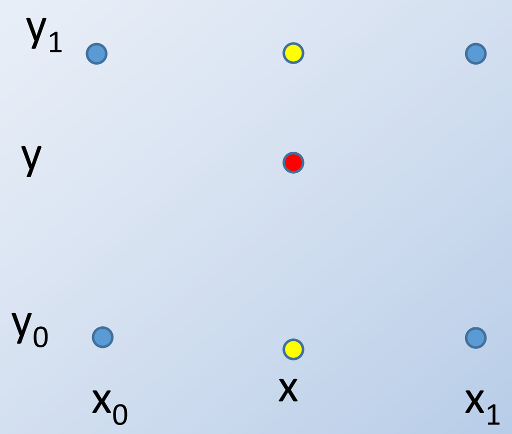
In this case, To find I[x,y] we need 4 points: I[x_0,y_0], I[x_0,y_1], I[x_1,y_0], and I[x_1,y_1]:



Bi-linear Interpolation

In two dimensions, we will use bilinear interpolation, which consists of applying linear interpolation first along one dimension and then along the other.

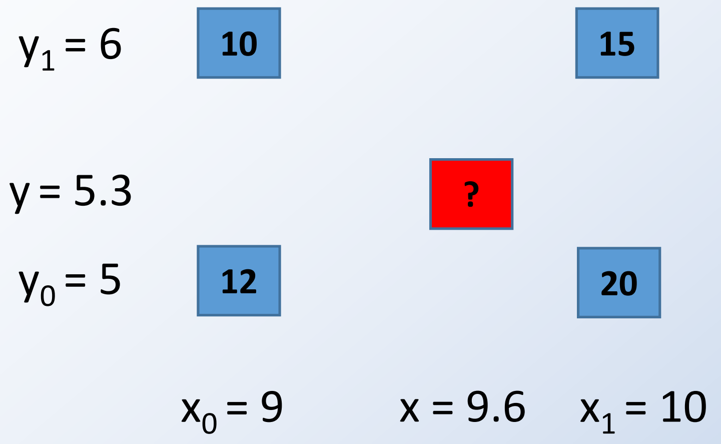
In this case, To find $I[x,y]$ we need 4 points: $I[x_0,y_0]$, $I[x_0,y_1]$, $I[x_1,y_0]$, and $I[x_1,y_1]$:



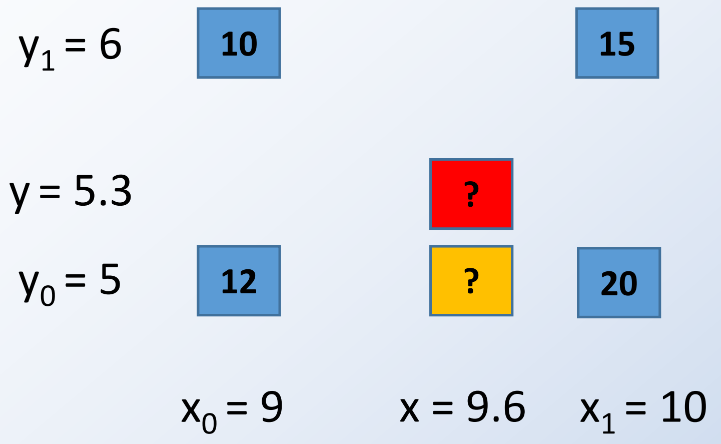
find $I[x,y_0]$ by interpolating $I[x_0,y_0]$ and $I[x_1,y_0]$
find $I[x,y_1]$ by interpolating $I[x_0,y_1]$ and $I[x_1,y_1]$
find $I[x,y]$ by interpolating $I[x,y_0]$ and $I[x,y_1]$

Bi-linear Interpolation - Example

Find the intensity of pixel I[9.6,5.3]



Bi-linear Interpolation - Example



To find the intensity of pixel I[9.6,5.3]

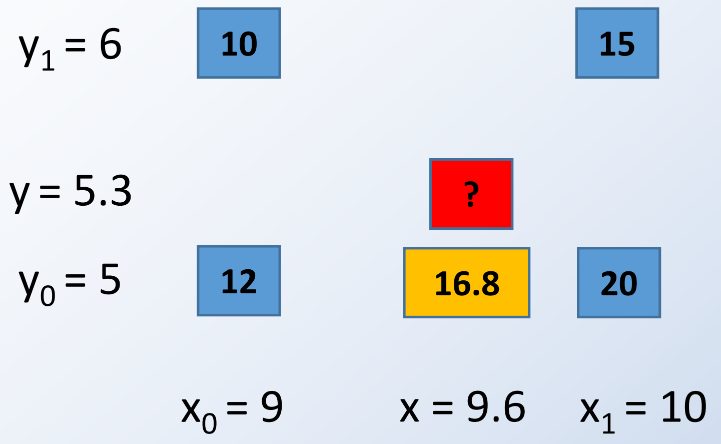
1. Find the intensity of I[9.6,5]

I[9.6,5] = I[9,5] + (9.6-9) (I[10,5]-I[9,5])/(10-9)

= 12 + (0.6)(8) = 12+4.8 = 16.8

$$f(x) = f(x_0) + \frac{(x - x_0)\,(f(x_1) - f(x_0))}{x_1 - x_0}$$

Bi-linear Interpolation - Example



To find the intensity of pixel I[9.6,5.3]

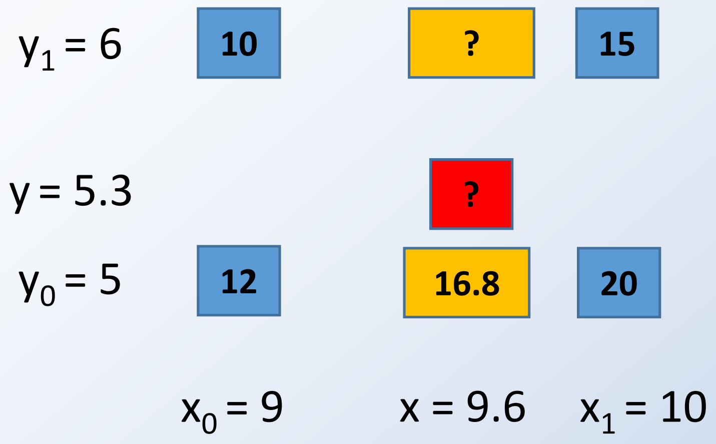
1. Find the intensity of I[9.6,5]

I[9.6,5] = I[9,5] + (9.6-9) (I[10,5]-I[9,5])/(10-9)

= 12 + (0.6)(8) = 12+4.8 = 16.8

$$f(x) = f(x_0) + \frac{(x - x_0)\,(f(x_1) - f(x_0))}{x_1 - x_0}$$

Bi-linear Interpolation - Example



To find the intensity of pixel I[9.6,5.3]

Find the intensity of I[9.6,5]

I[9.6,5] = I[9,5] + (9.6-9) (I[10,5]-I[9,5])/(10-9)

= 12 + (0.6)(20-12)/1 = 12 + (0.6)(8) = 12+4.8

= 16.8

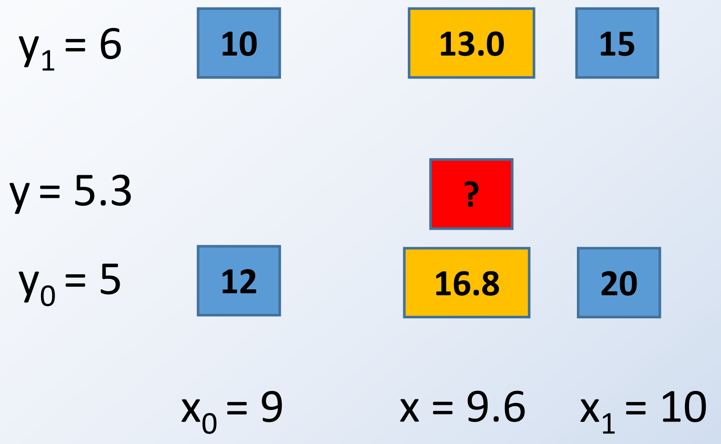
Find the intensity of I[9.6,6]

I[9.6,6] = I[9,6] + (9.6-9) (I[10,6]-I[9,6])/(10-9)

= 10 + (0.6)(15-10) = 10+(0.6)(5) = 13

$$f(x) = f(x_0) + \frac{(x - x_0)\,(f(x_1) - f(x_0))}{x_1 - x_0}$$

Bi-linear Interpolation - Example



$$f(x) = f(x_0) + \frac{(x - x_0)\,(f(x_1) - f(x_0))}{x_1 - x_0}$$

To find the intensity of pixel I[9.6,5.3]

Find the intensity of I[9.6,5]

I[9.6,5] = I[9,5] + (9.6-9) (I[10,5]-I[9,5])/(10-9)

= 12 + (0.6)(20-12)/1 = 12 + (0.6)(8) = 12+4.8

= 16.8

Find the intensity of I[9.6,6]

I[9.6,6] = I[9,6] + (9.6-9) (I[10,6]-I[9,6])/(10-9)

= 10 + (0.6)(15-10) = 10+(0.6)(5) = 13

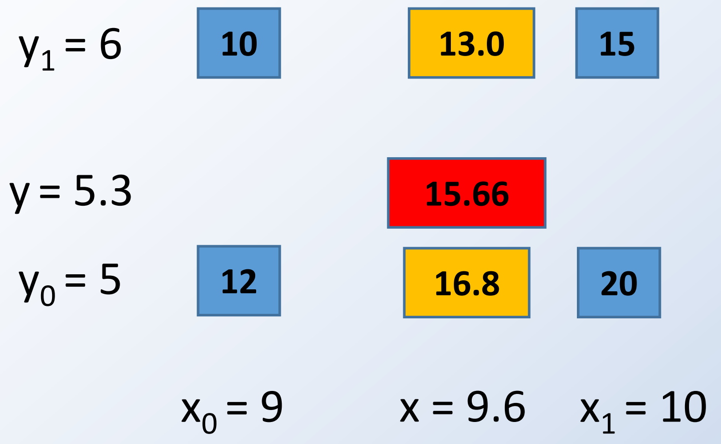
Find the intensity of I[9.6,5.3]

I[9.6,5.3] = I[9.6,5] + (5.3-5) (I[9.6,6]-I[9.6,5])/(6-5)

= 16.8 + (0.3)(13-16.8) = 16.8 – (0.3)(3.8)

= 15.436

Bi-linear Interpolation - Example



$$f(x) = f(x_0) + \frac{(x - x_0)\,(f(x_1) - f(x_0))}{x_1 - x_0}$$

To find the intensity of pixel I[9.6,5.3]

Find the intensity of I[9.6,5]

I[9.6,5] = I[9,5] + (9.6-9) (I[10,5]-I[9,5])/(10-9)

= 12 + (0.6)(20-12)/1 = 12 + (0.6)(8) = 12+4.8

= 16.8

Find the intensity of I[9.6,6]

I[9.6,6] = I[9,6] + (9.6-9) (I[10,6]-I[9,6])/(10-9)

= 10 + (0.6)(15-10) = 10+(0.6)(5) = 13

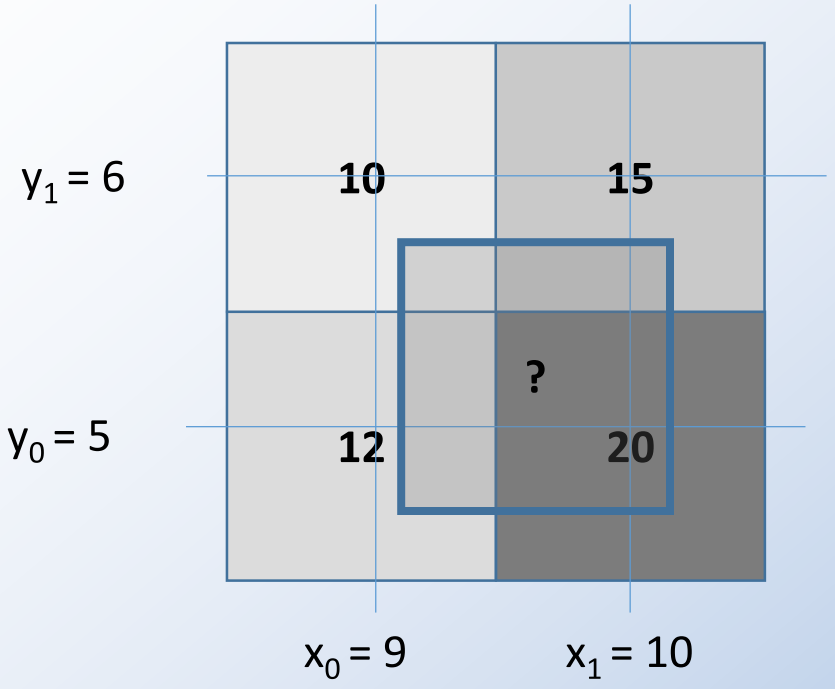
Find the intensity of I[9.6,5.3]

I[9.6,5.3] = I[9.6,5] + (5.3-5) (I[9.6,6]-I[9.6,5])/(6-5)

= 16.8 + (0.3)(13-16.8) = 16.8 – (0.3)(3.8)

= 15.66

Bi-linear Interpolation - Example



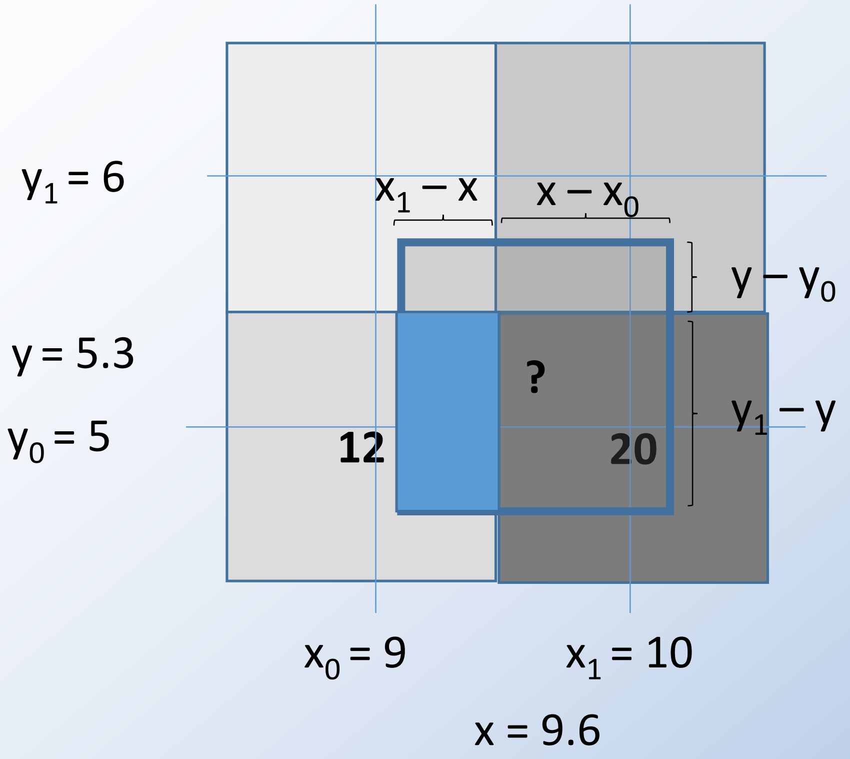
Equivalently, assuming $x_1 = x_0 + 1$ and $y_1 = y_0 + 1$

Consider pixel I[x,y] is a squared centered at [x,y]

We can compute the area of overlap of I[x,y] with each of $I[x_0,y_0]$, $I[x_0,y_1]$, $I[x_1,y_0]$, and $I[x_1,y_1]$ and use the overlap as weights

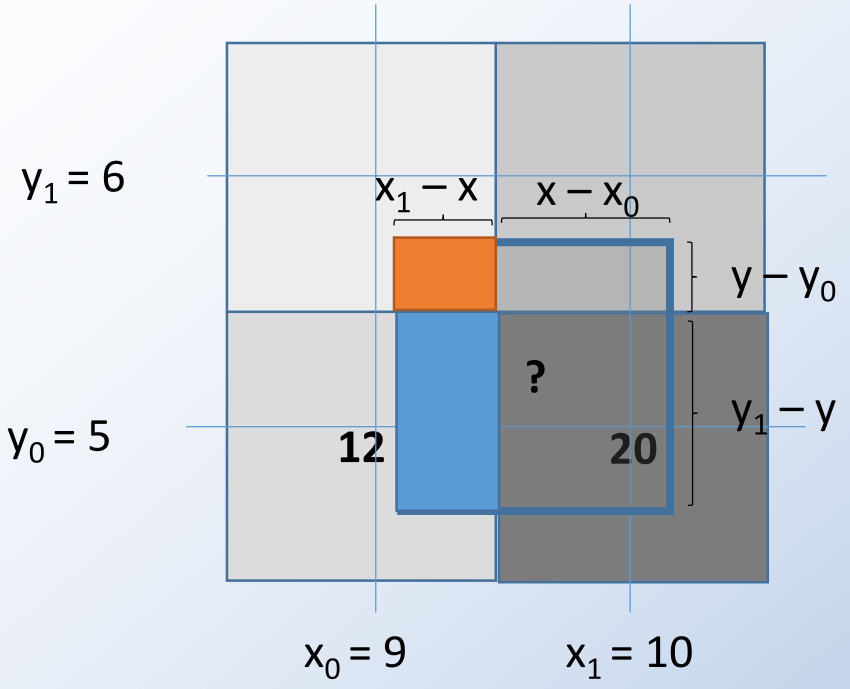
Intuitively, the larger the overlap, the more similar the intensities should be

Bi-linear Interpolation - Example



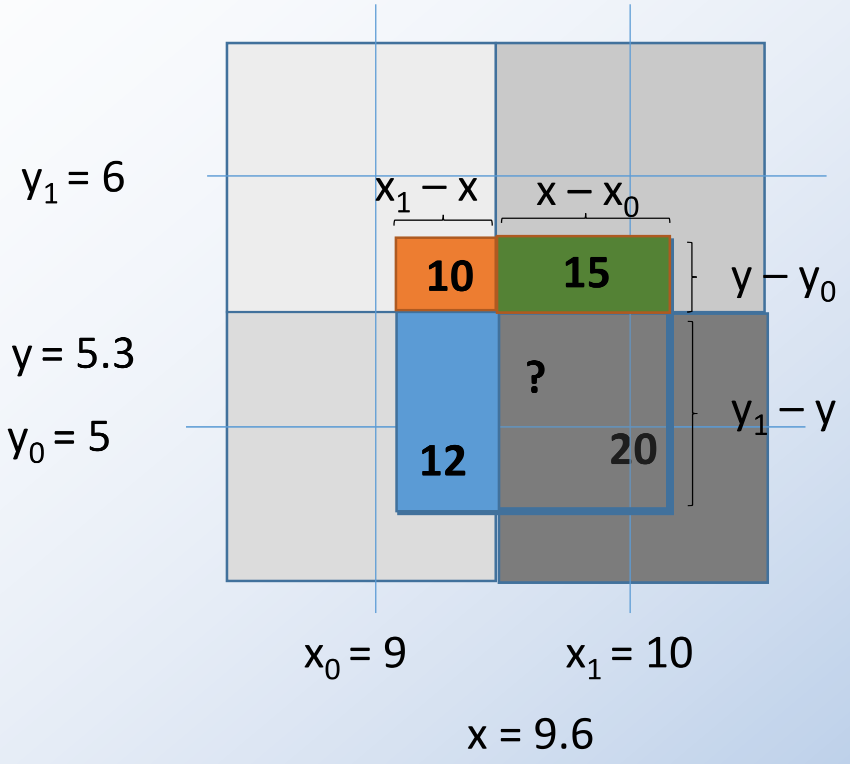
$I[x,y] = I[x_0,y_0]\,(y_1-y)(x_1-x) +$

Bi-linear Interpolation - Example



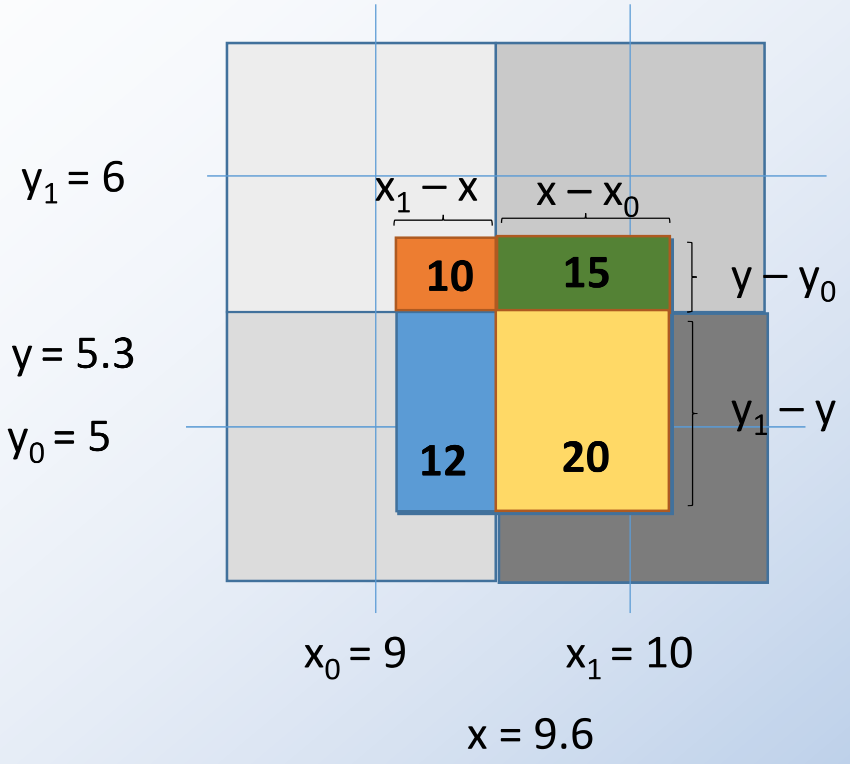
$$I[x,y] = I[x_0,y_0]\,(y_1-y)(x_1-x) +$$
$$I[x_0,y_1]\,(y - y_0)(x_1-x) +$$

Bi-linear Interpolation - Example



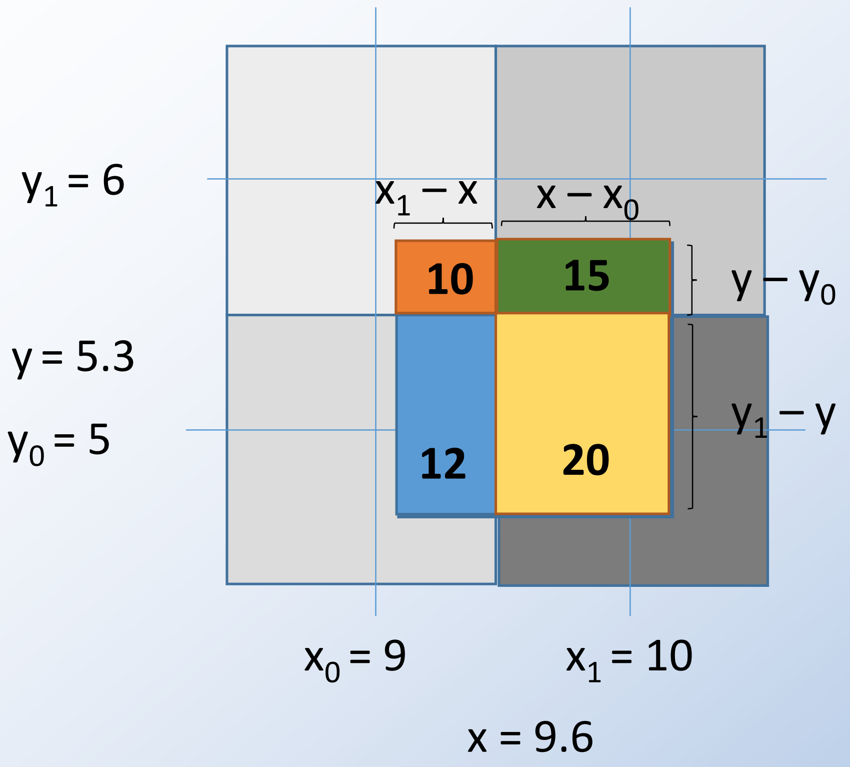
$$I[x,y] = I[x_0,y_0]\,(x_1 - x)(y_1 - y) + I[x_0,y_1]\,(x_1 - x)(y - y_0) + I[x_1,y_1]\,(x - x_0)(y - y_0) +$$

Bi-linear Interpolation - Example



$$
\begin{aligned}
I[x,y] = {} & I[x_0,y_0]\,(x_1 - x)(y_1 - y) + \\
& I[x_0,y_1]\,(x_1 - x)(y - y_0) + \\
& I[x_1,y_1]\,(x - x_0)(y - y_0) + \\
& I[x_1,y_0]\,(x - x_0)(y_1 - y) \\
= {} &
\end{aligned}
$$

Bi-linear Interpolation - Example



$$
\begin{aligned}
I[x,y] = {} & I[x_0,y_0]\,(x_1 - x)(y_1 - y) + \\
& I[x_0,y_1]\,(x_1 - x)(y - y_0) + \\
& I[x_1,y_1]\,(x - x_0)(y - y_0) + \\
& I[x_1,y_0]\,(x - x_0)(y_1 - y) \\
= {} & 12*(0.4)(0.7) + \\
& 10*(0.4)(0.3) + \\
& 15*(0.6)(0.3) + \\
& 20*(0.6)(0.7) + \\
= {} & 3.36 + 1.2 + 2.7 + 8.4 \\
= {} & 15.66
\end{aligned}
$$

Geometric Transformations in Python

Recall that in Python arrays can be used as indices

```
I = np.arange(6).reshape(2,3)*10
print(I)
[[ 0 10 20]
 [30 40 50]]
cols, rows = np.meshgrid(np.arange(I.shape[1]), np.arange(I.shape[0]))
print(rows)
[[0 0 0]
 [1 1 1]]
print(cols)
[[0 1 2]
 [0 1 2]]
print(I[rows,cols])
[[ 0 10 20]
 [30 40 50]]
```

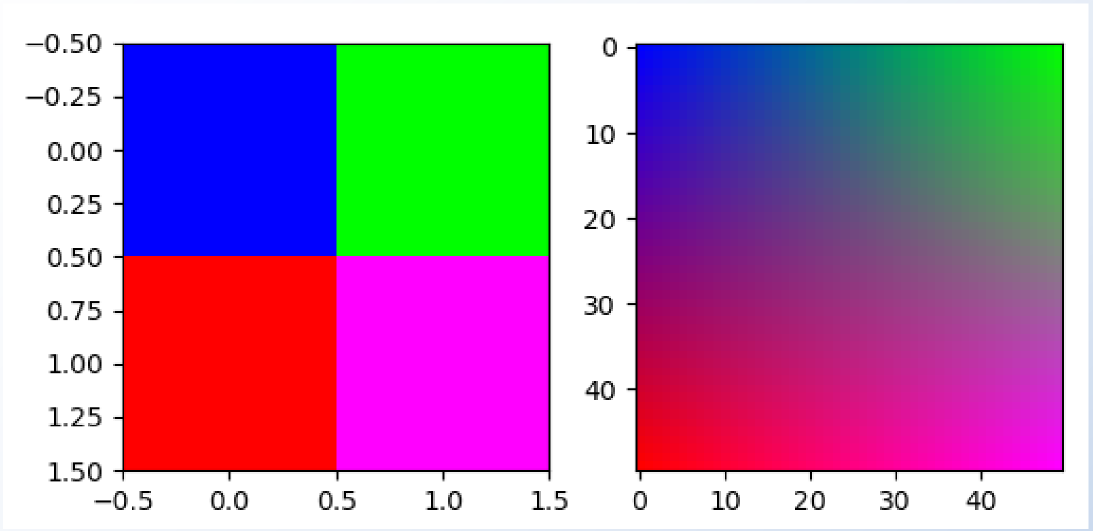
Geometric Transformations in Python

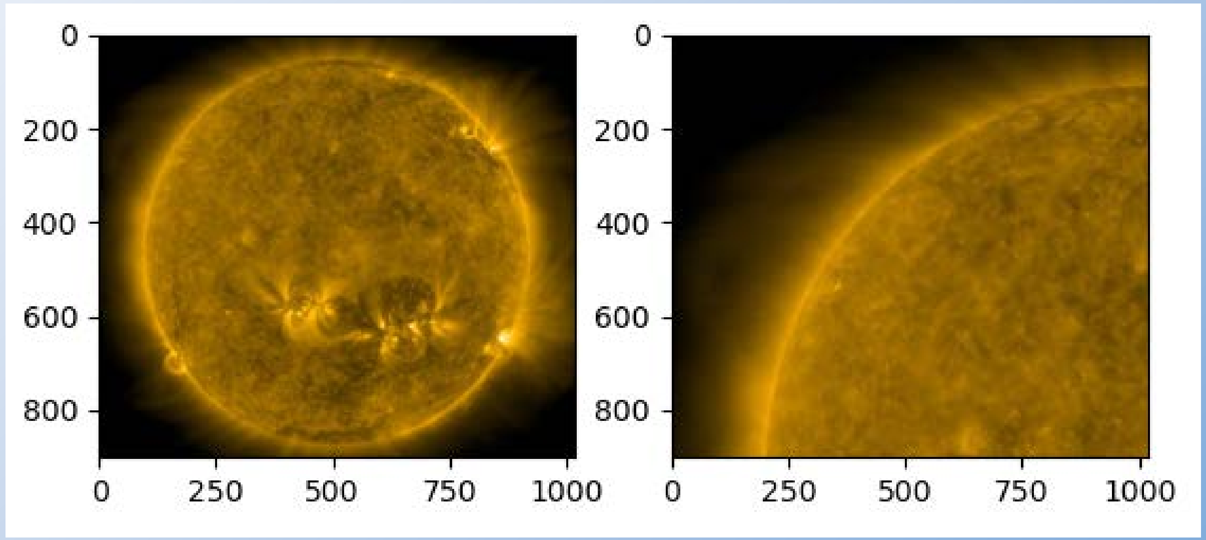
Implementation of bilinear interpolation – returns an image obtained from Source, where rows and cols are floating point arrays

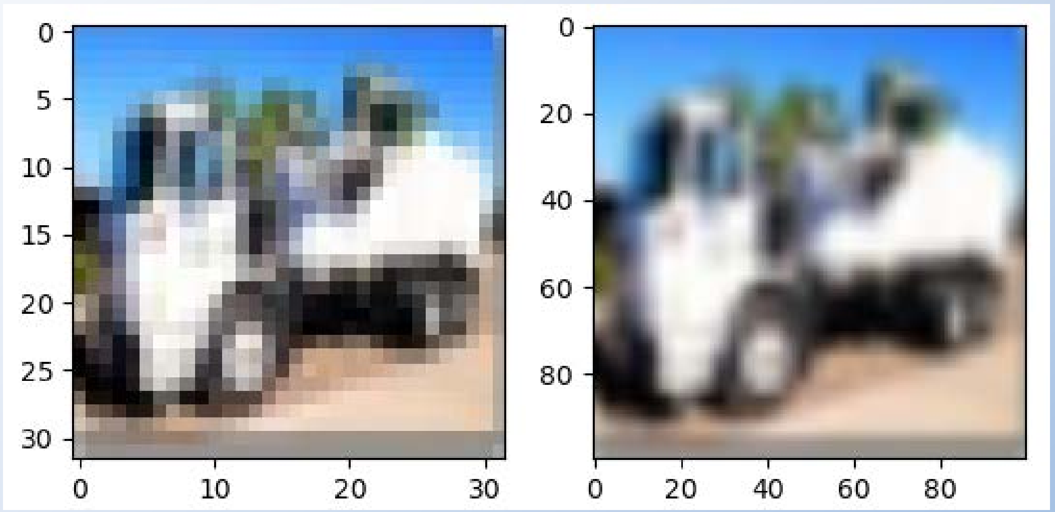
```
def real_index(Source, rows, cols):
    rows_floor = (rows).astype(int)
    rows_ceil = np.ceil(rows).astype(int)
    cols_floor = (cols).astype(int)
    cols_ceil = np.ceil(cols).astype(int)

    wr0 = 1 - rows + rows_floor
    wc0 = 1 - cols + cols_floor
    wr1 = 1 - wr0
    wc1 = 1 - wc0

    Dest = Source[rows_floor,cols_floor]*wr0*wc0 +
       Source[rows_floor,cols_ceil]*wr0*wc1 +
       Source[rows_ceil,cols_floor]*wr1*wc0 +
       Source[rows_ceil,cols_ceil]*wr1*wc1

    return Dest
```

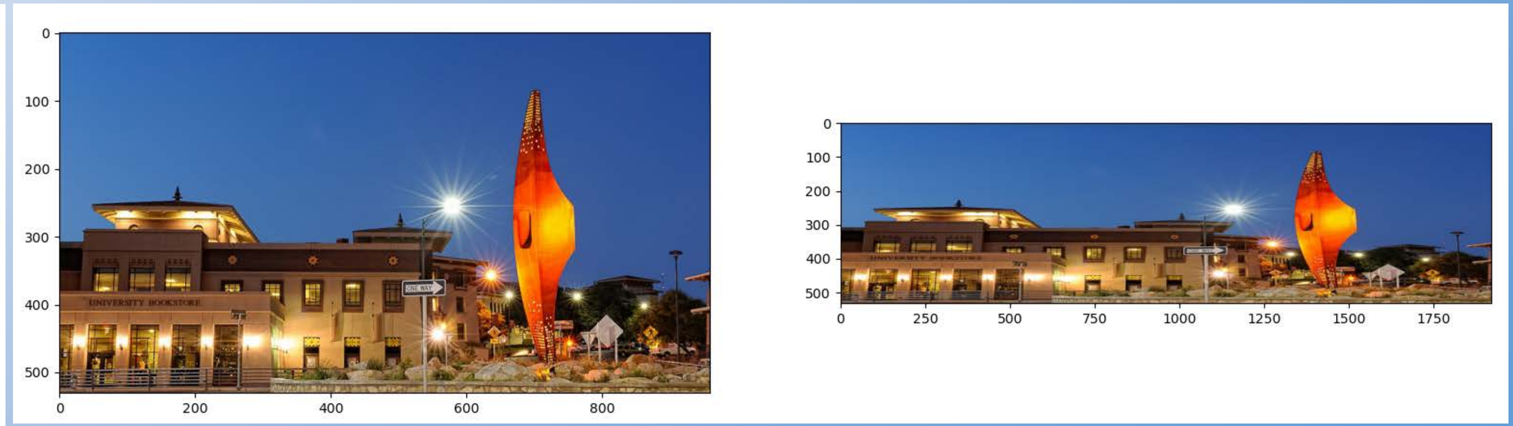
Geometric Transformations in Python

See `real_indexing_demo.py`

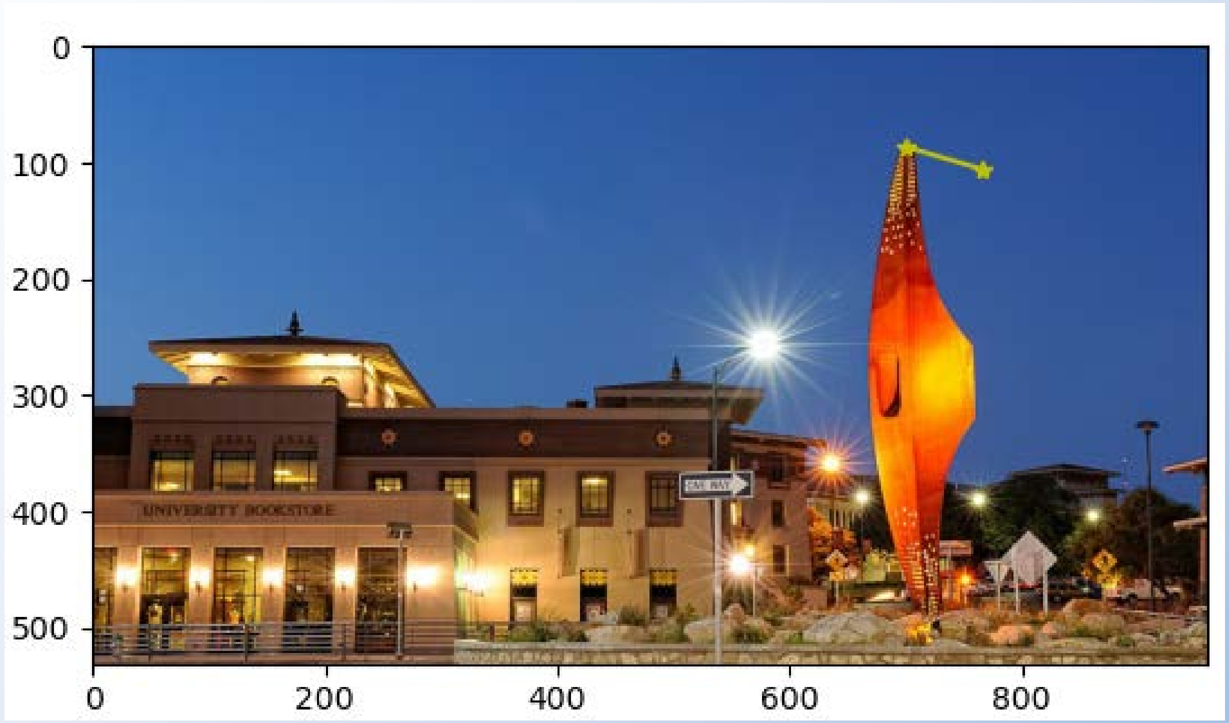


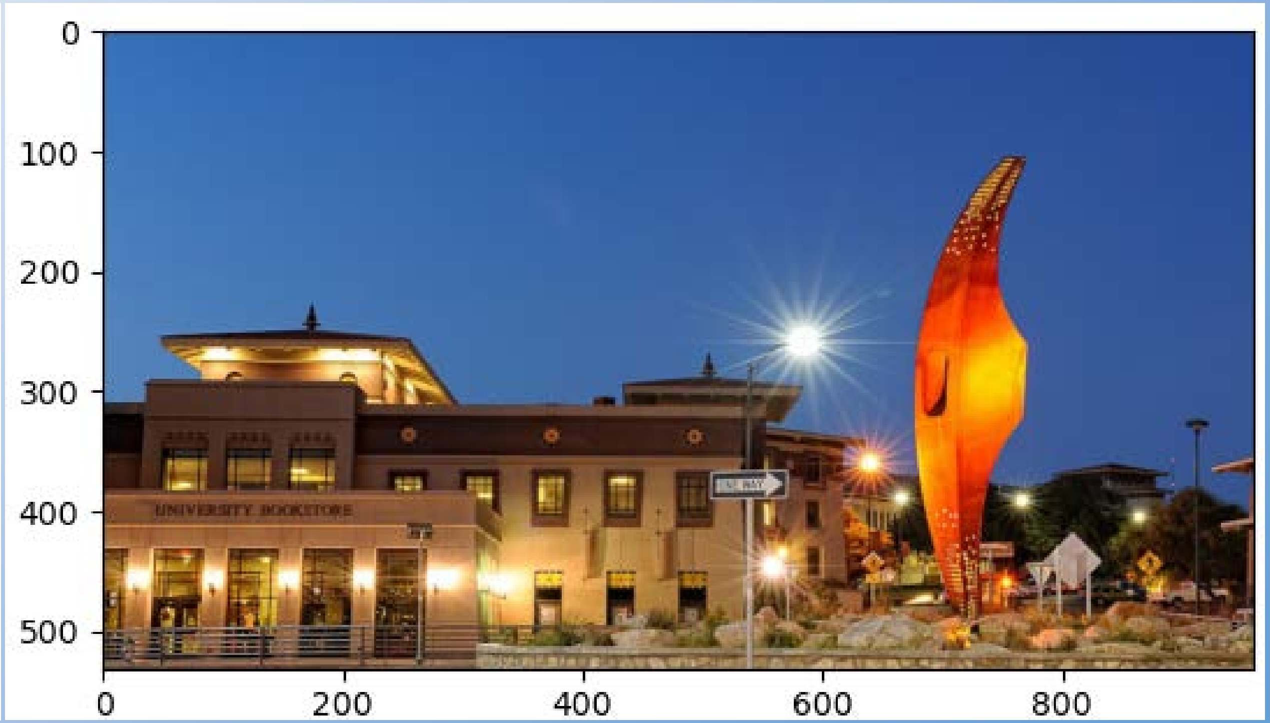






Single point warp





Single point warp

Define the source (x_0,y_0) and destination (x_1,y_1) of a single point in I

Define displacement matrices Dx and Dy of the same size as I

Compute:

$Dx[x,y] = exp(-dist((x,y),(x_1,y_1))/k)*(x_0-x_1)$

$Dy[x,y] = exp(-dist((x,y),(x_1,y_1))/k)*(y_0-y_1)$

Let *X* and *Y* be the column and row matrices

$T = I[X+Dx,Y+Dy]$

Single point warp

Exercise:

Implement single point warp as described in the previous slide

Parametric Transformations

A parametric transform has the form:

$$\begin{bmatrix} u \\ v \end{bmatrix} = H \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \qquad \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = H \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Where:

- (x,y) are the source coordinates
- (u,v) are the destination coordinates
- H is a 2-by-3 or 3-by-3 array

Technical Aside: Matrix Multiplication

The operation

$$\begin{bmatrix} u \\ v \end{bmatrix} = H \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \qquad \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = H \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Where:

- (x,y) are the source coordinates
- (u,v) are the destination coordinates
- H is a 2-by-3 or 3-by-3 array

Technical Aside: Matrix Multiplication

The operation

$$\begin{bmatrix} u \\ v \end{bmatrix} = H \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \qquad \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = H \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Where:

- (x,y) are the source coordinates
- (u,v) are the destination coordinates
- H is a 2-by-3 or 3-by-3 array

Technical Aside: Matrix Multiplication

Preliminaries:

The dot product of two vectors (or 1D arrays) x = [x0,x1,...] and y = [y0,y1,...]

is given by

$x \cdot y = x_0 y_0 + x_1 y_1 + \ldots$

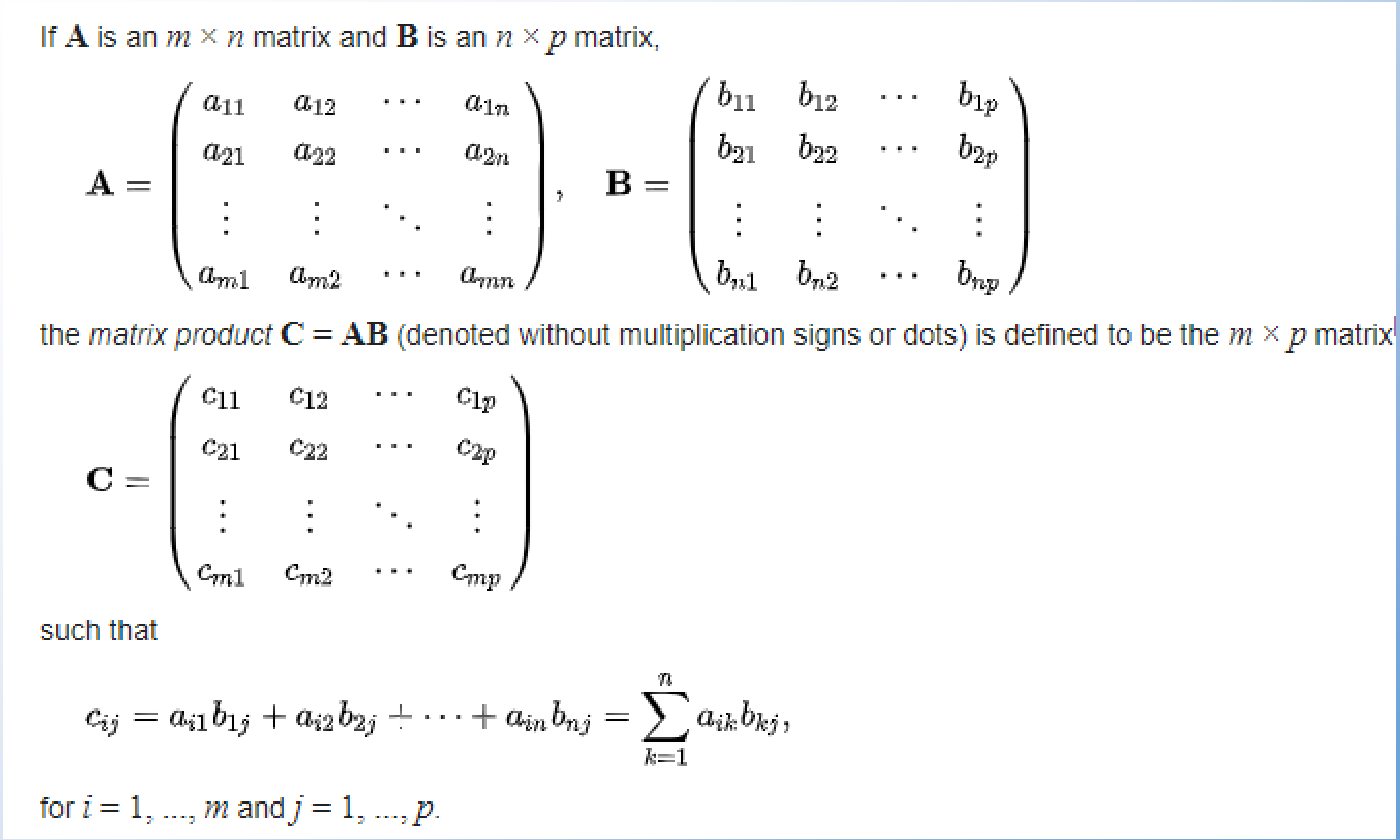
In Python, if x and y are 1D arrays of the same length:

```
dot_product = np.sum(x*y)
```

or

```
dot_product = np.dot(x,y)
```

Technical Aside: Matrix Multiplication



If $\mathbf{A}$ is an $m \times n$ matrix and $\mathbf{B}$ is an $n \times p$ matrix,

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

the *matrix product* $\mathbf{C} = \mathbf{AB}$ (denoted without multiplication signs or dots) is defined to be the $m \times p$ matrix

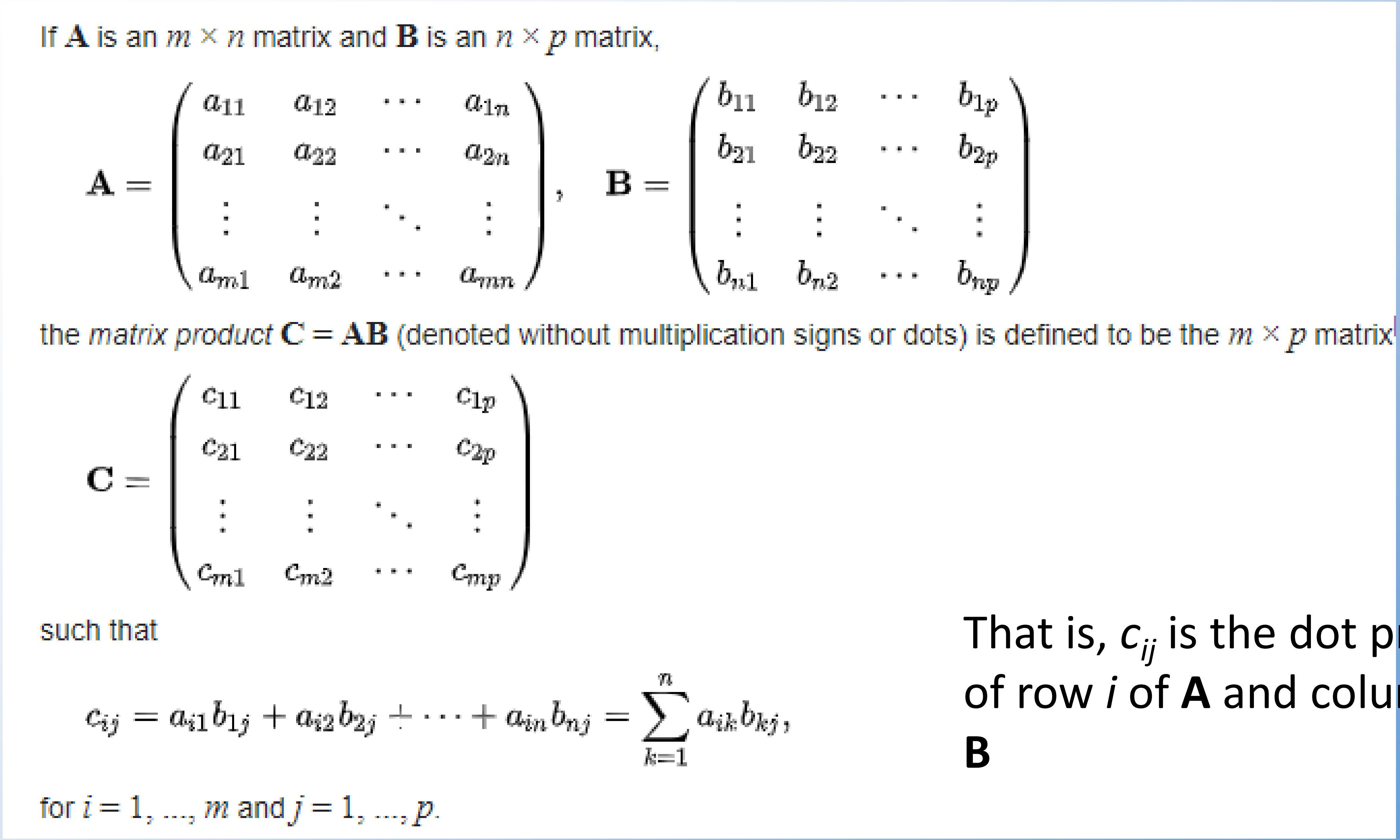
$$\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{pmatrix}$$

such that

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj},$$

for $i = 1, \ldots, m$ and $j = 1, \ldots, p$.

Technical Aside: Matrix Multiplication



If $\mathbf{A}$ is an $m \times n$ matrix and $\mathbf{B}$ is an $n \times p$ matrix,

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

the *matrix product* $\mathbf{C} = \mathbf{AB}$ (denoted without multiplication signs or dots) is defined to be the $m \times p$ matrix

$$\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{pmatrix}$$

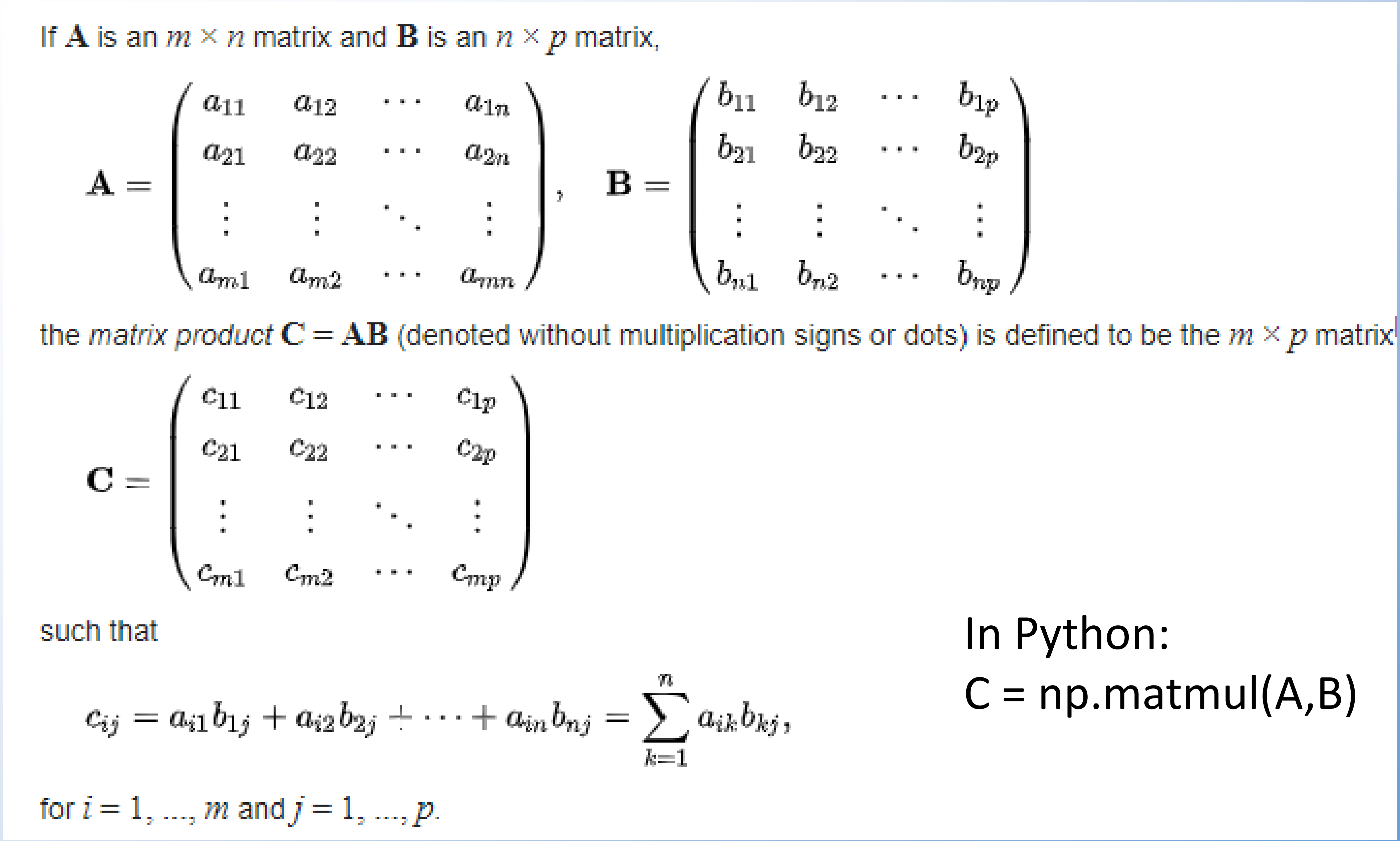
such that

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj},$$

for $i = 1, \ldots, m$ and $j = 1, \ldots, p$.

That is, c_{ij} is the dot product of row i of **A** and column j of **B**

Technical Aside: Matrix Multiplication



If $\mathbf{A}$ is an $m \times n$ matrix and $\mathbf{B}$ is an $n \times p$ matrix,

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

the *matrix product* $\mathbf{C} = \mathbf{AB}$ (denoted without multiplication signs or dots) is defined to be the $m \times p$ matrix

$$\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{pmatrix}$$

such that

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj},$$

for $i = 1, \ldots, m$ and $j = 1, \ldots, p$.

In Python:

```
C = np.matmul(A,B)
```

Parametric Transformations

$$\begin{bmatrix} u \\ v \end{bmatrix} = H \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$H = \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \end{bmatrix}$$

$$u = h_{00}x + h_{01}y + h_{02}$$
$$v = h_{10}x + h_{11}y + h_{12}$$

We can also arrange ALL source points in a 3-by-n S matrix and the result of the matrix multiplication is a 2-by-n matrix containing all the destination points

Parametric Transformations

$$\begin{bmatrix} u \\ v \end{bmatrix} = H \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

For notation convenience, we add a row with all '1's to the matrix containing the source coordinates x and y.
This allows us to represent many different transformations in the same way.
This representation is called **homogeneous coordinates**

$$H = \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \end{bmatrix}$$

$$u = h_{00}x + h_{01}y + h_{02}$$
$$v = h_{10}x + h_{11}y + h_{12}$$

Parametric Transformations

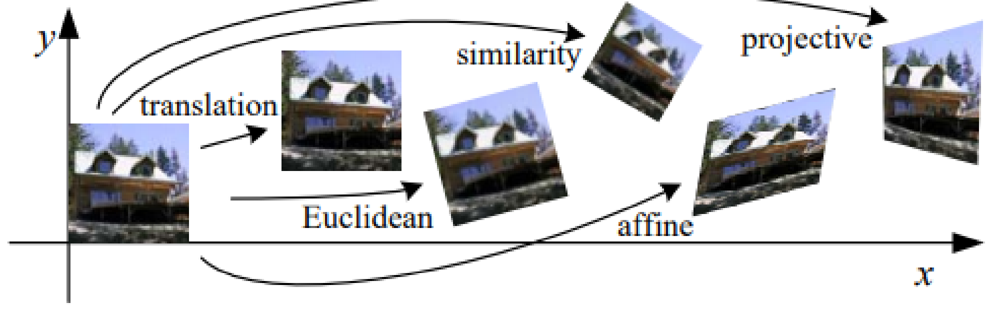


Figure 3.45 *Basic set of 2D geometric image transformations.*

Transformation	Matrix	# DoF	Preserves	Icon
translation	$\begin{bmatrix}\mathbf{I} & \mathbf{t}\end{bmatrix}_{2\times 3}$	2	orientation	□
rigid (Euclidean)	$\begin{bmatrix}\mathbf{R} & \mathbf{t}\end{bmatrix}_{2\times 3}$	3	lengths	◇
similarity	$\begin{bmatrix}s\mathbf{R} & \mathbf{t}\end{bmatrix}_{2\times 3}$	4	angles	◇
affine	$\begin{bmatrix}\mathbf{A}\end{bmatrix}_{2\times 3}$	6	parallelism	▱
projective	$\begin{bmatrix}\tilde{\mathbf{H}}\end{bmatrix}_{3\times 3}$	8	straight lines	⏢

Table 3.5 *Hierarchy of 2D coordinate transformations. Each transformation also preserves the properties listed in the rows below it, i.e., similarity preserves not only angles but also parallelism and straight lines. The* 2×3 *matrices are extended with a third* $[\mathbf{0}^T\ 1]$ *row to form a full* 3×3 *matrix for homogeneous coordinate transformations.*

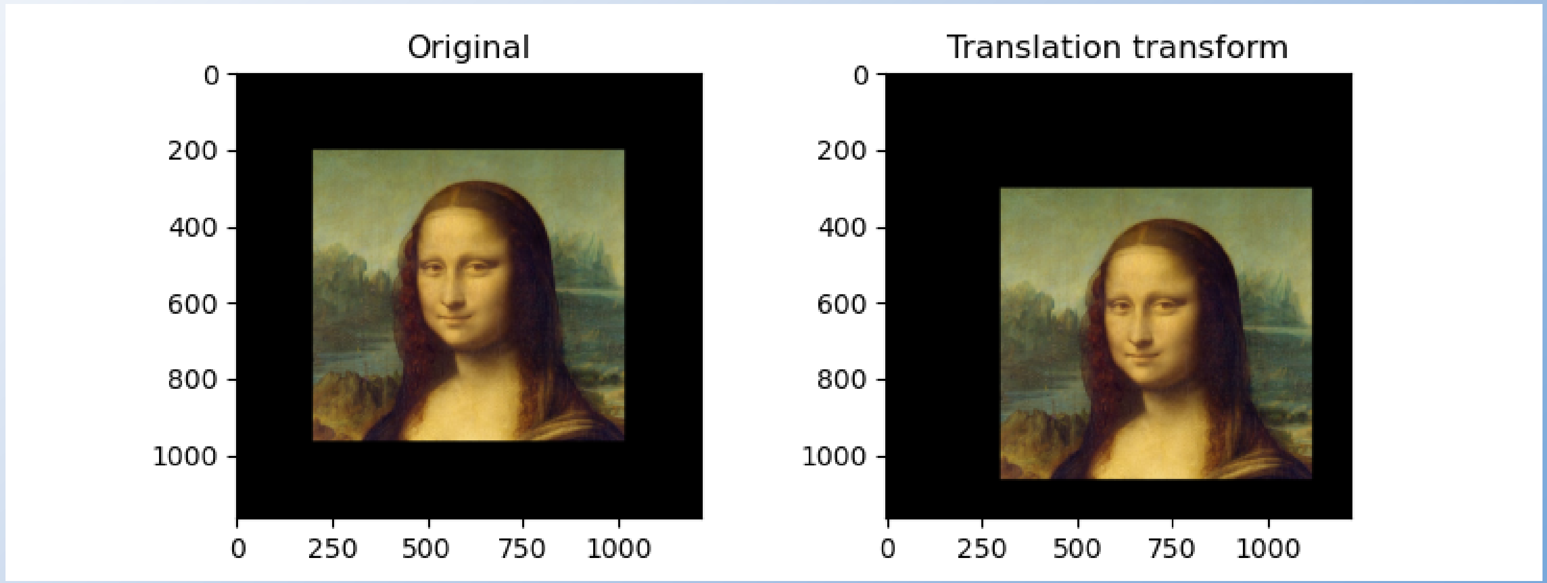
Parametric Transformations

Translation by (Δx, Δy)

$$H = \begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 0 & \Delta y \\ 0 & 0 & 1 \end{bmatrix}$$

Preserves:

- Orientation
- Lengths
- Angles
- Parallelism
- Straight lines



Parametric Transformations

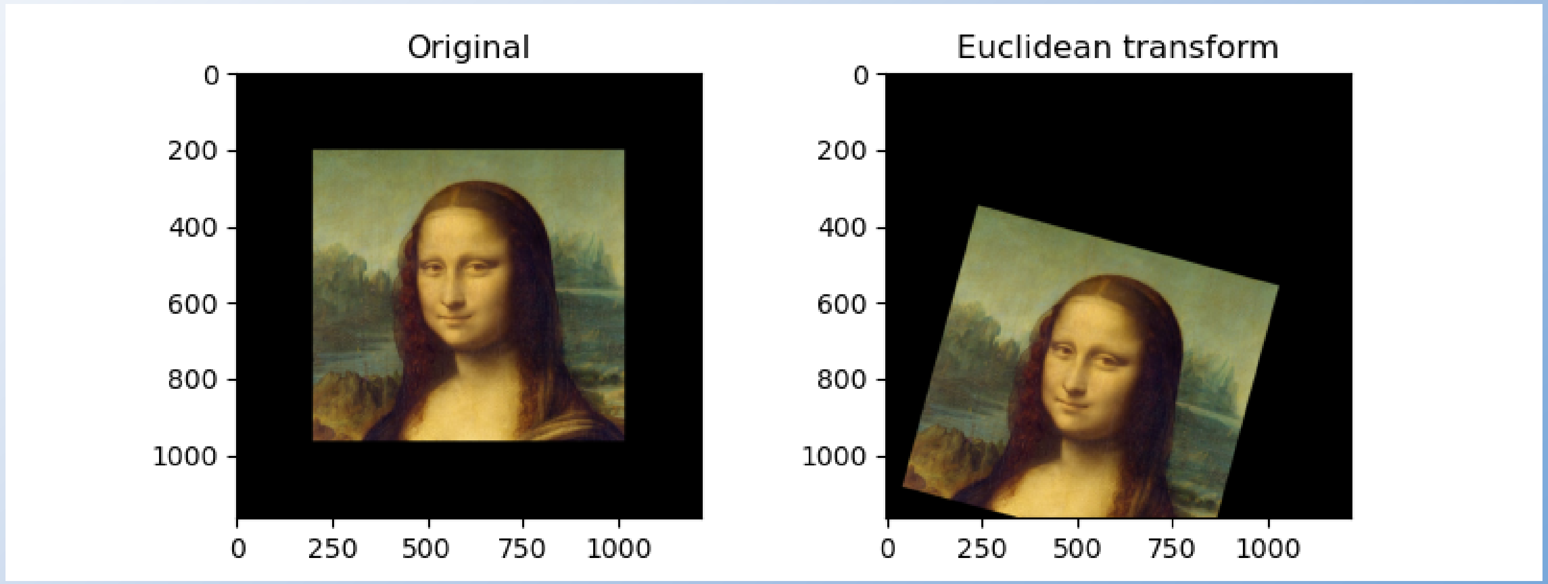
Euclidean Transformation -

Rotation by ϕ translation by (Δx, Δy)

$$H = \begin{bmatrix} \cos(\phi) & -\sin(\phi) & \Delta x \\ \sin(\phi) & \cos(\phi) & \Delta y \\ 0 & 0 & 1 \end{bmatrix}$$

Preserves:

- Lengths
- Angles
- Parallelism
- Straight lines



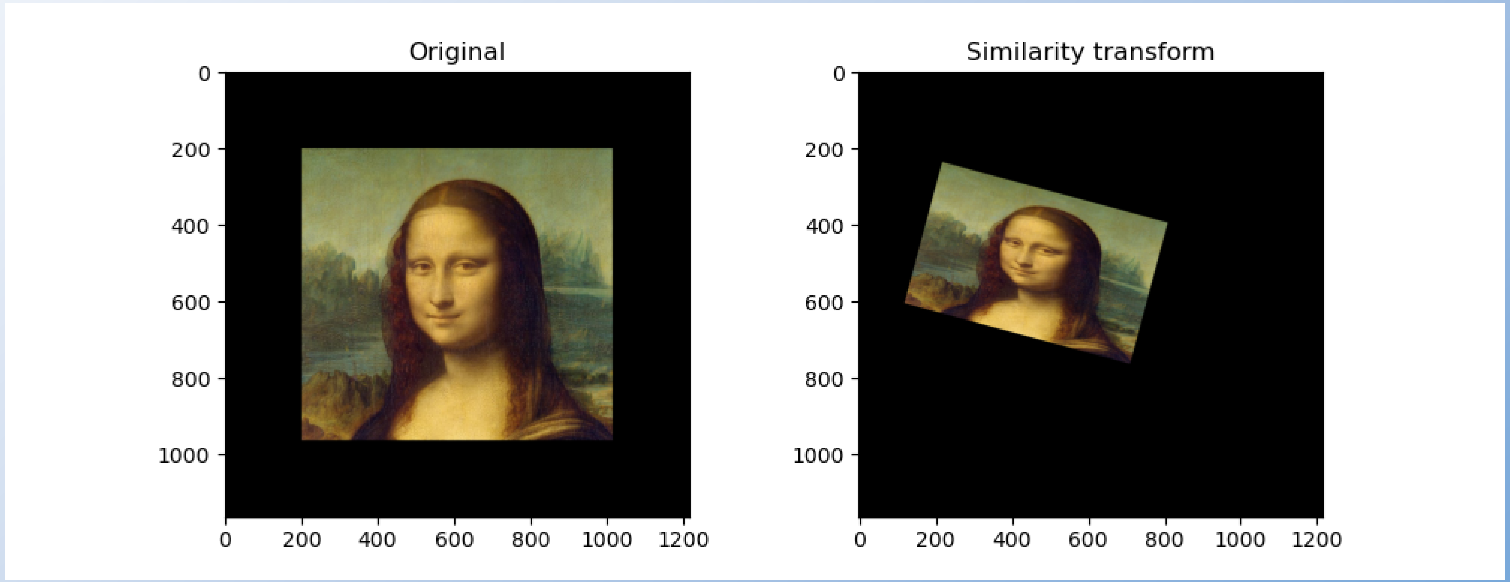
Parametric Transformations

Similarity Transformation -

Rotation by ϕ, translation by (Δx, Δy), scaling by (s_x, s_y)

$$H = \begin{bmatrix} s_x \cos(\phi) & -\,s_y \sin(\phi) & \Delta x \\ s_x \sin(\phi) & s_y \cos(\phi) & \Delta y \\ 0 & 0 & 1 \end{bmatrix}$$

Preserves:
- Angles
- Parallelism
- Straight lines



Parametric Transformations

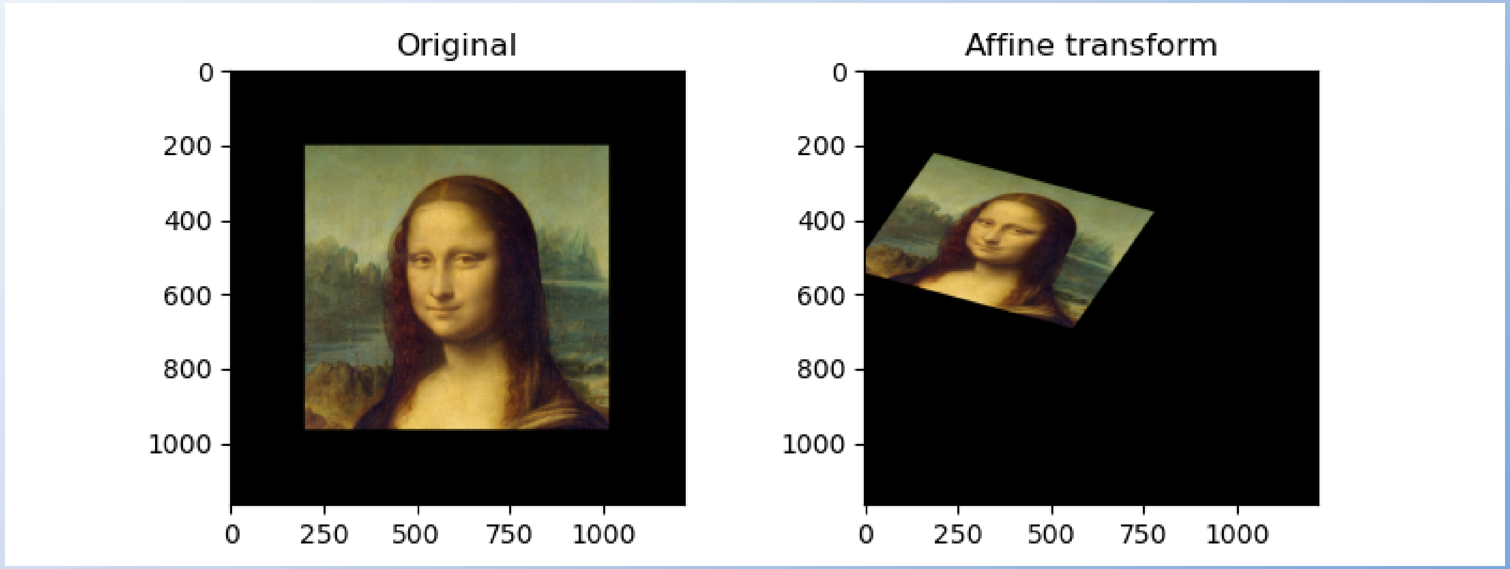
Affine Transformation -

Rotation by ϕ, translation by $(\Delta x, \Delta y)$, scaling by (s_x, s_y), shearing by α

$$H = \begin{bmatrix} h_{00} & h_{01} & \Delta x \\ h_{10} & h_{11} & \Delta y \\ 0 & 0 & 1 \end{bmatrix}$$

Preserves:

- Parallelism
- Straight lines



Parametric Transformations in Python

We will use the skimage transform library

```
from skimage import transform as tf
# Translation
H0 = tf.EuclideanTransform(translation=(100,100))
# Euclidean or rigid transformation - rotation and translation
H1 = tf.EuclideanTransform(rotation=15*np.pi/180, translation=(100,100))
# Similarity transformation - scale, rotation and translation
H2 = tf.SimilarityTransform(scale=(0.75,0.5), rotation = 15*np.pi/180,
translation=(100,100))
# Affine transformation – shear, scale, rotation and translation
H3 = tf.AffineTransform(shear=20*np.pi/180, scale=(0.75,0.5), rotation = 15*np.pi/180,
translation=(100,100))
# Projective transformation – affine + perspective
matrix = np.array([[1, -0.3, 100],
                   [0.1, 0.9,100],
                   [0.0002, -0.0001, 1]])
H4 = tf.ProjectiveTransform(matrix=matrix)
```

Parametric Transformations in Python

We will use the skimage transform library

```
from skimage import transform as tf
# Define Homography
H = tf.EuclideanTransform(translation=(100,100))
# Generate Destionation Image
warped_image = tf.warp(canvas, H)
```

Parametric Transformations in Python

All parametric geometric transformation we have described are invertible
- that is, we can always transform the destination image back into the original image

It’s easier to reason in terms of forward transformation – where will the pixels in the source image go?

We will define the homography in terms of the forward transformation

Since the destination image is generated using the backward transformation, we will warp using the inverse homography

In Python:

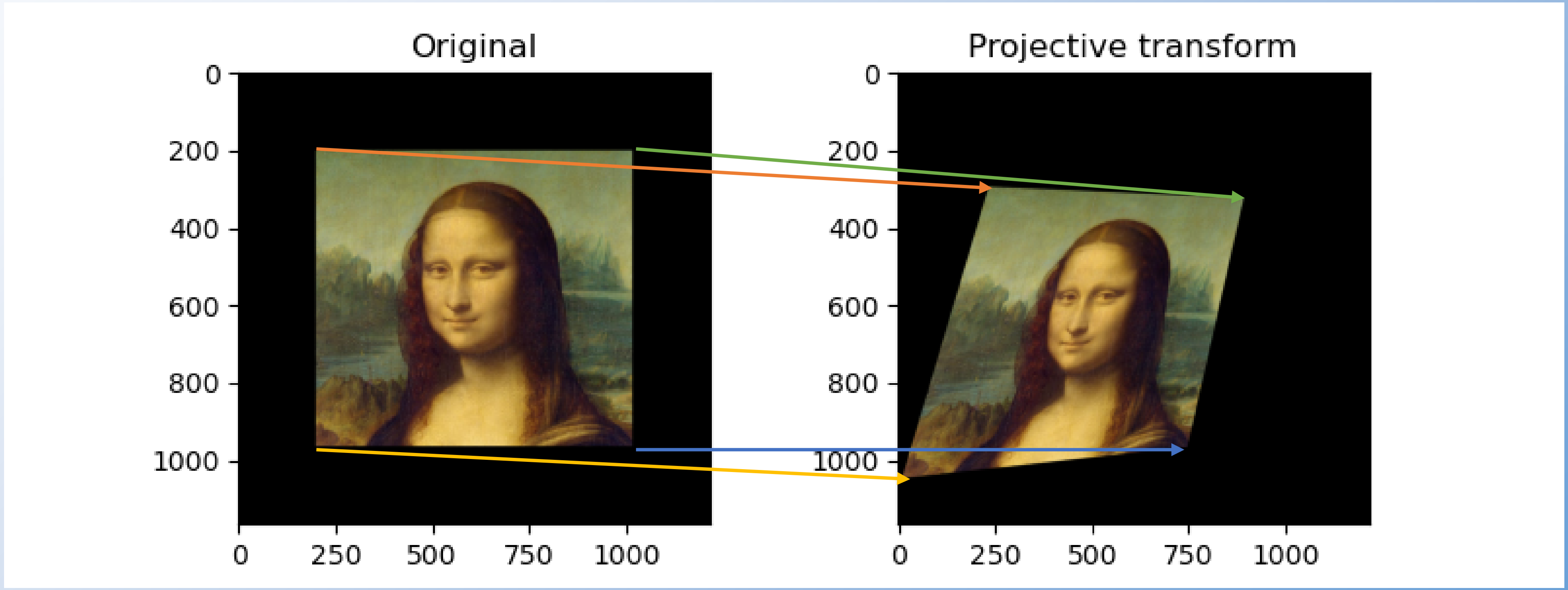
```
warped_image = tf.warp(canvas, H.inverse)
```

See homographies_demo.py

Parametric Transformations

Using a homography we can build a destination image from a source image

We can also find a homography given four corresponding points in the source and destination images

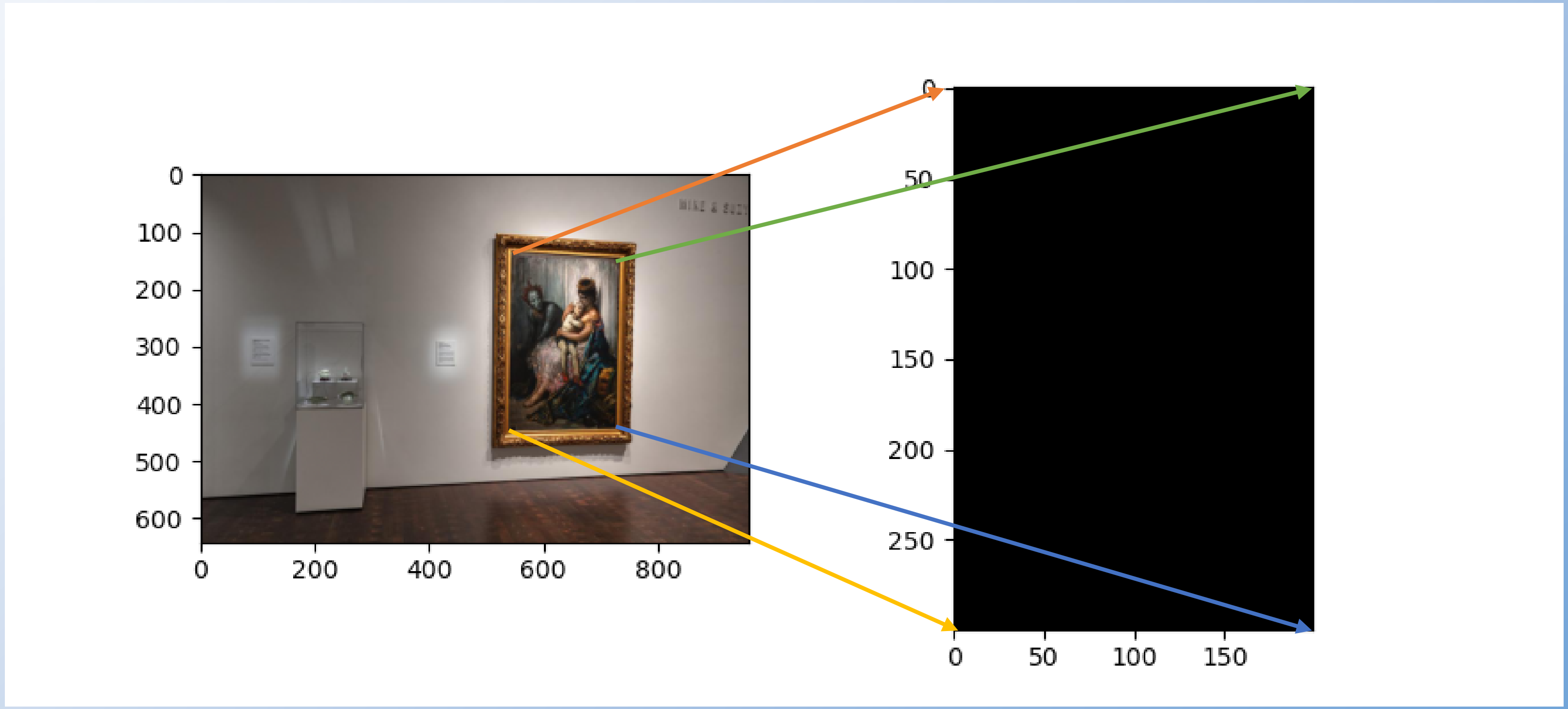


Parametric Transformations

Example:

Define point correspondences

Find homography from point correspondences



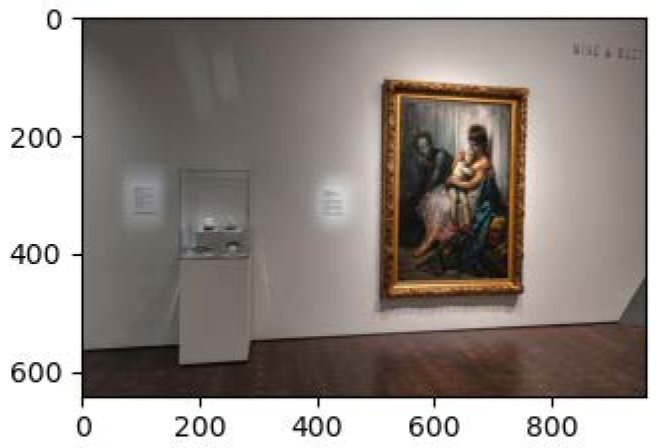
Parametric Transformations

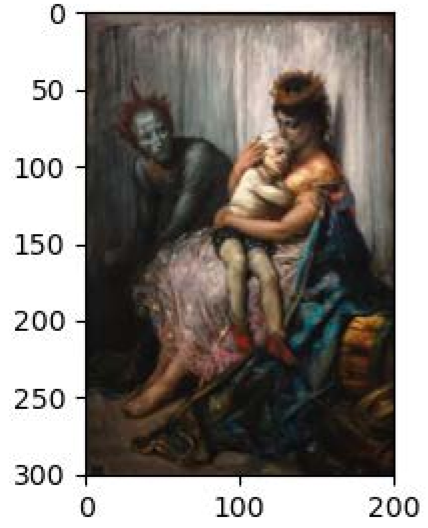
Example:

Define point correspondences

Find homography from point correspondences

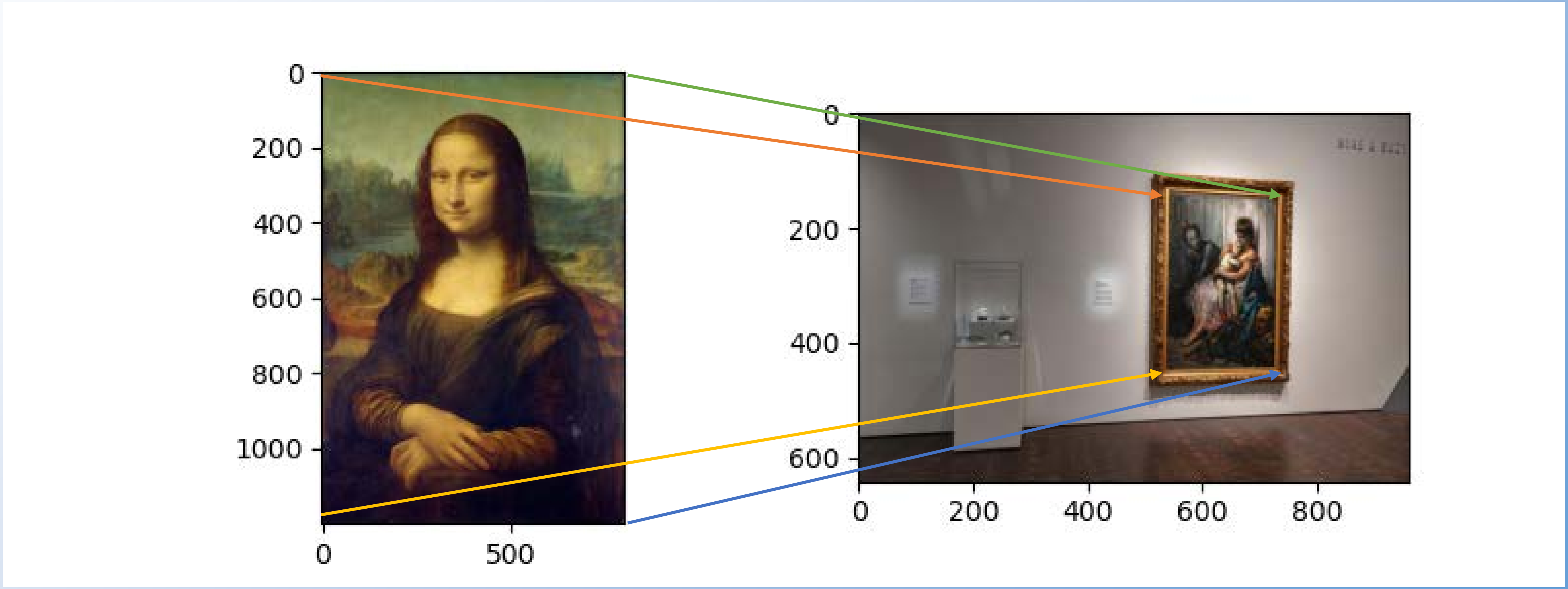
Generate destination image using source and homography





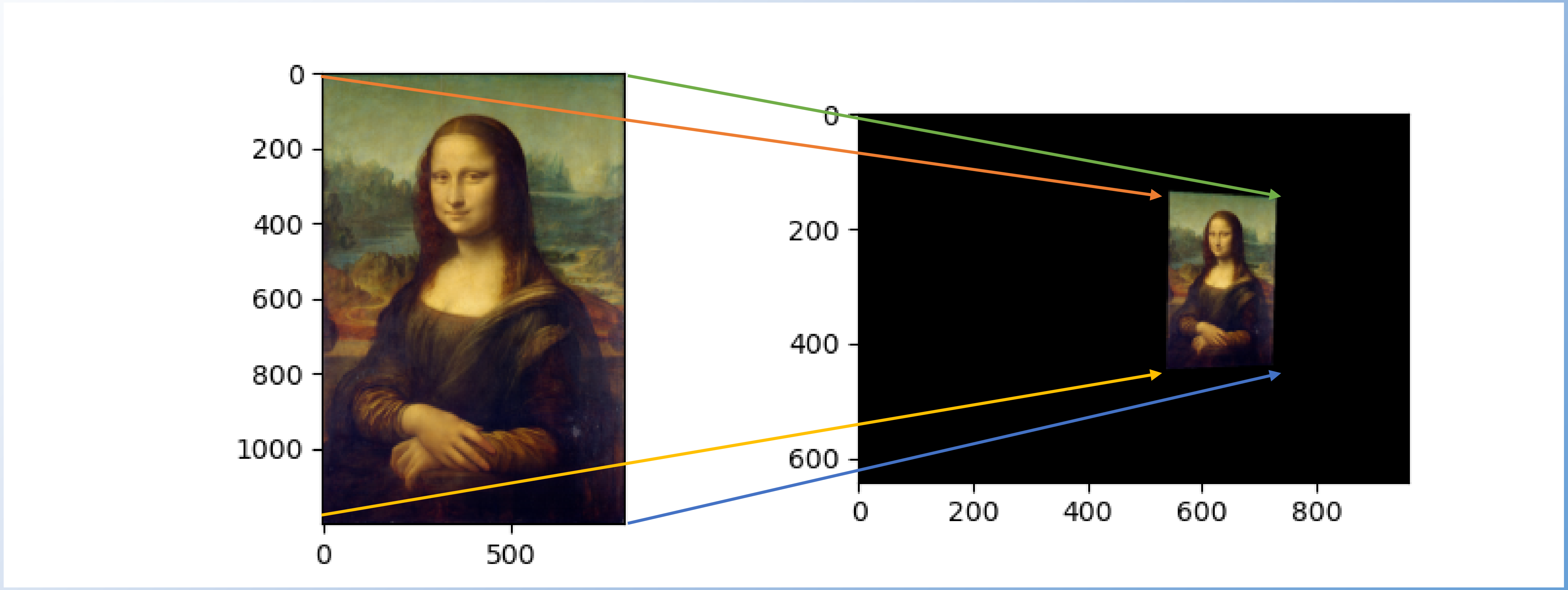
Parametric Transformations

Example 2: Replacing part of an image
Define point correspondences
Find homography from point correspondences
Generate destination image using source and homography



Parametric Transformations

Example 2: Replacing part of an image
Define point correspondences
Find homography from point correspondences
Generate destination image using source and homography



Parametric Transformations

Example 2: Replacing part of an image

Define point correspondences

Find homography from point correspondences

Generate destination image using source and homography

Replace all (0,0,0) in warped image by source image