Imagery numérique

Theme 5

Geometric Transformations

Content of course

Semester 1

Theme 1: Introduction to image processing

Theme 2: The HVS perception and color

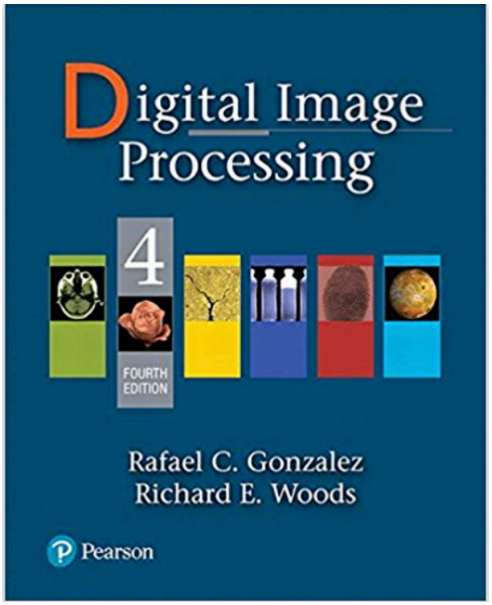
Theme 3: Image acquisition and sensing

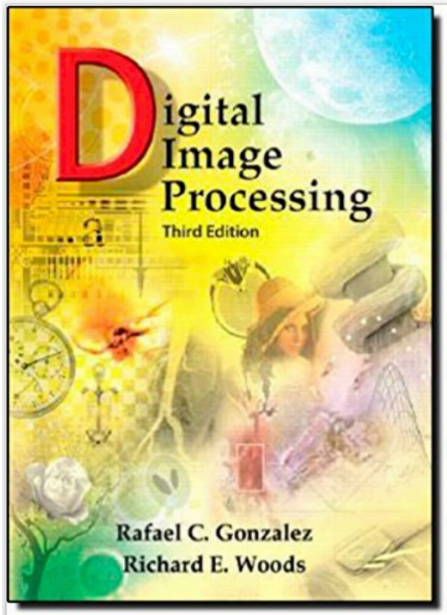
Theme 4: Histograms and point operations

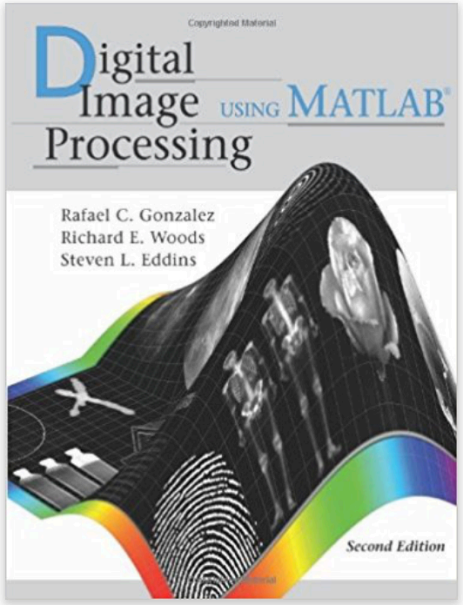
Theme 5: Geometric operations

Theme 6: Spatial filters

Recommended books





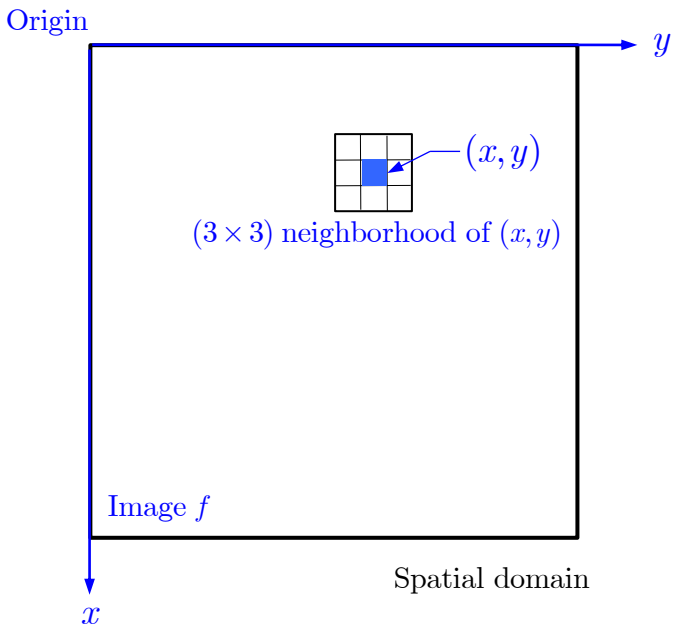


YouTube lectures

Intro to Digital Image Processing (ECSE-4540) Lectures, Spring 2015

by Prof. Rich Radke from Rensselaer Polytechnic Institute

Recall: Histograms and Point operations



$$g(x, y) = T\Big[f(x, y)\Big]$$

Procedure of **spatial filtering**

1. Consider the neighborhood of pixel (x, y)
2. Apply a transformation of pixels $T\Big[.\Big]$
3. Move to a new pixel location

Spatial filter

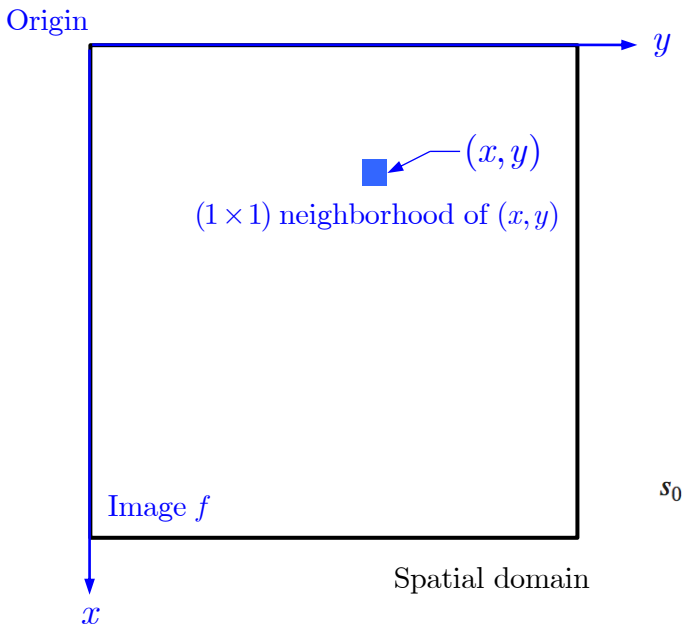
(3×3) neighborhood of (x, y)

Known as:

- *filter*
- *kernel*
- *template*
- *window*

Recall: Histograms and Point operations

Simplest form when the size of window is 1×1

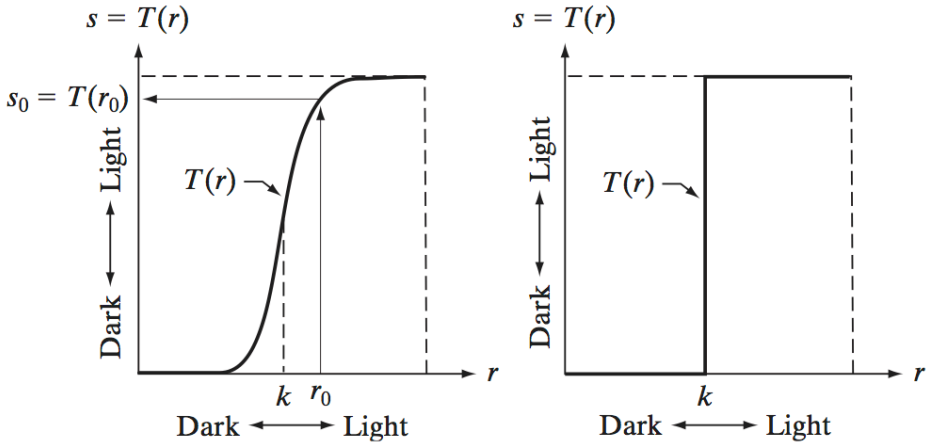


Spatial filtering → point operation

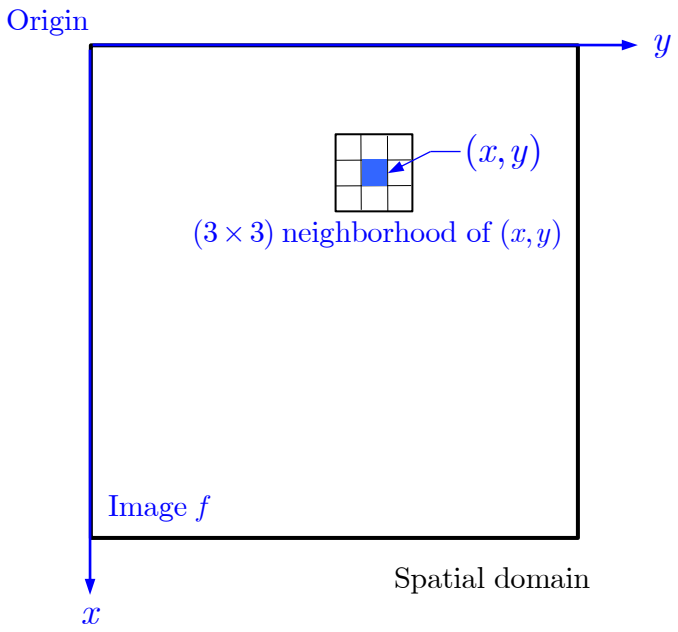
$$g(x,y) = T\big[f(x,y)\big] \qquad s = T\big[r\big]$$

output intensity — input intensity

It is a.k.a. intensity transformation



Recall: Histograms and Point operations



Types of image processing operations:

Operations on intensity of image:

- **Point-wise**: histogram based processing
  - Theme 4
- **Group or NN-wise**: filtering
  - Theme 6

Operations on image coordinates

- **Geometrical transformations**
  - Theme 5

Geometric operations

There are several types of transformations

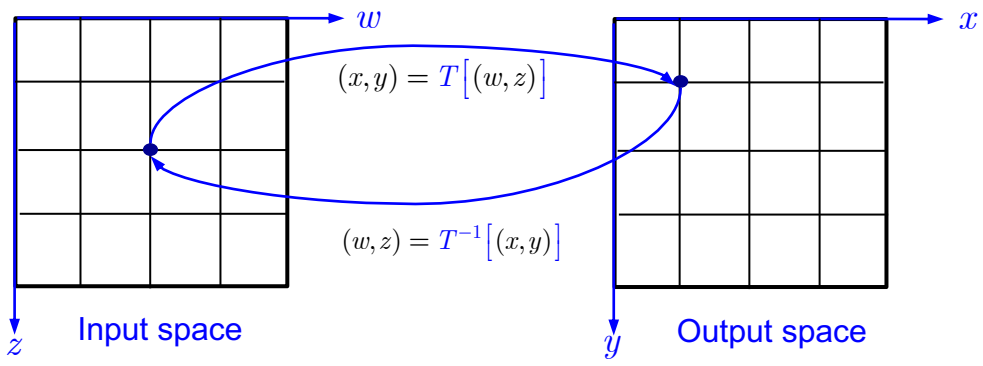
- Point or filtering operations are applied to pixel intensities $g(x,y) = T\big[f(x,y)\big]$
- Geometric transformation are applied to pixel coordinates $(x,y) = T\big[(w,z)\big]$

(w,z) - *input space*

(x,y) - *output space*

Geometric transforms:

- *forward* transform $(x,y) = T\big[(w,z)\big]$
- *inverse* transform (if exists) $(w,z) = T^{-1}\big[(x,y)\big]$



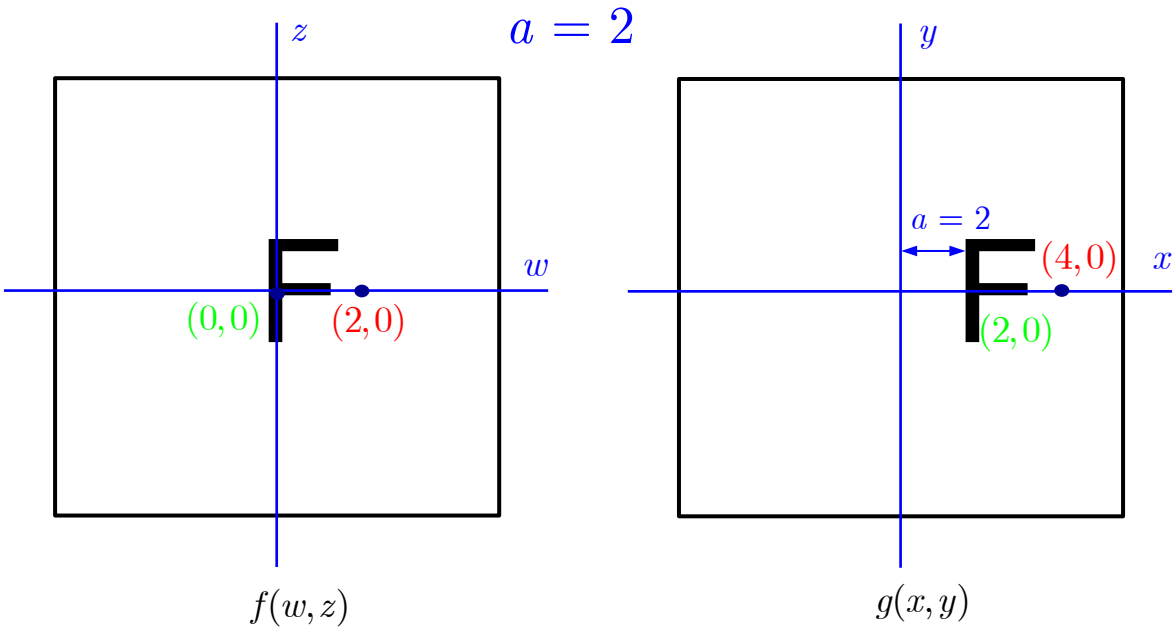
$$g(x,y) = f\left(\underbrace{T^{-1}\big[(x,y)\big]}_{(w,z)}\right)$$

Translation or shift

$$(x,y) = T\big[(w,z)\big] = (w+a,z)$$
$$(w,z) = T^{-1}\big[(x,y)\big] = (x-a,y)$$

$$\Rightarrow \quad g(x,y) = f\Big(T^{-1}\big[(x,y)\big]\Big) \quad \Rightarrow \quad g(x,y) = f(x-a,y)$$

Translation by a pixels



$$g(x,y) = f(x-a,y)$$
$$g(4,0) = f(4-2,0)$$
$$g(2,0) = f(2-2,0)$$

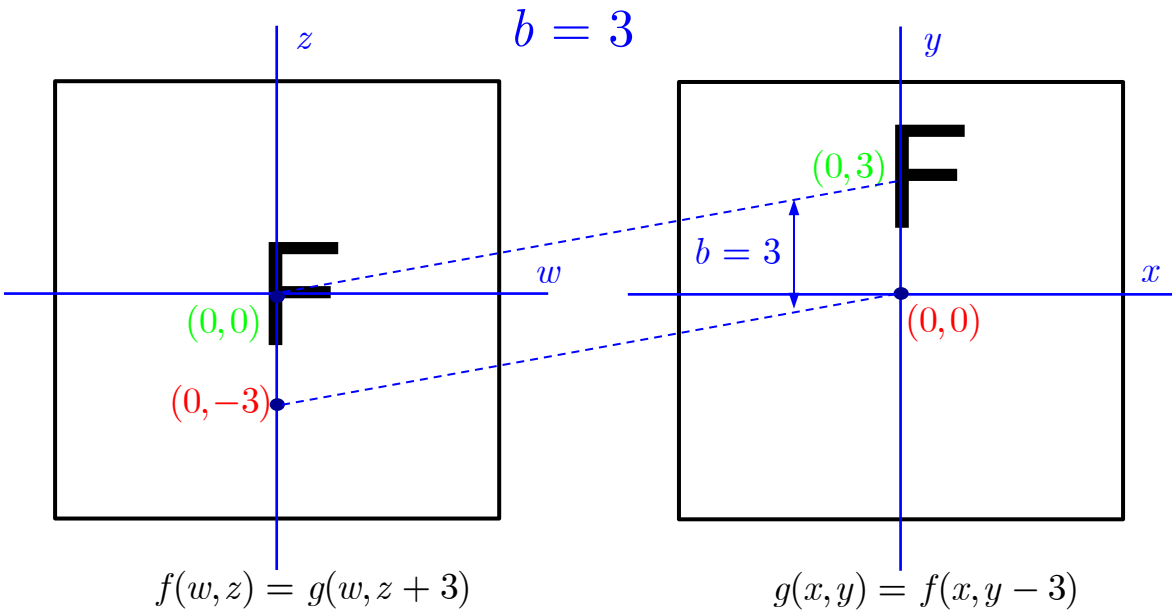
Remark: $T\big[(w,z)\big]$ should be interpreted as a transform $T\big[.\big]$ applied to a vector $T\begin{bmatrix} w \\ z \end{bmatrix}$

Translation or shift

$$(x,y) = T\big[(w,z)\big] = (w, z+b)$$
$$(w,z) = T^{-1}\big[(x,y)\big] = (x, y-b)$$

$$\Rightarrow \quad g(x,y) = f\Big(T^{-1}\big[(x,y)\big]\Big) \quad \Rightarrow \quad g(x,y) = f(x, y-b)$$

Translation up by b pixels

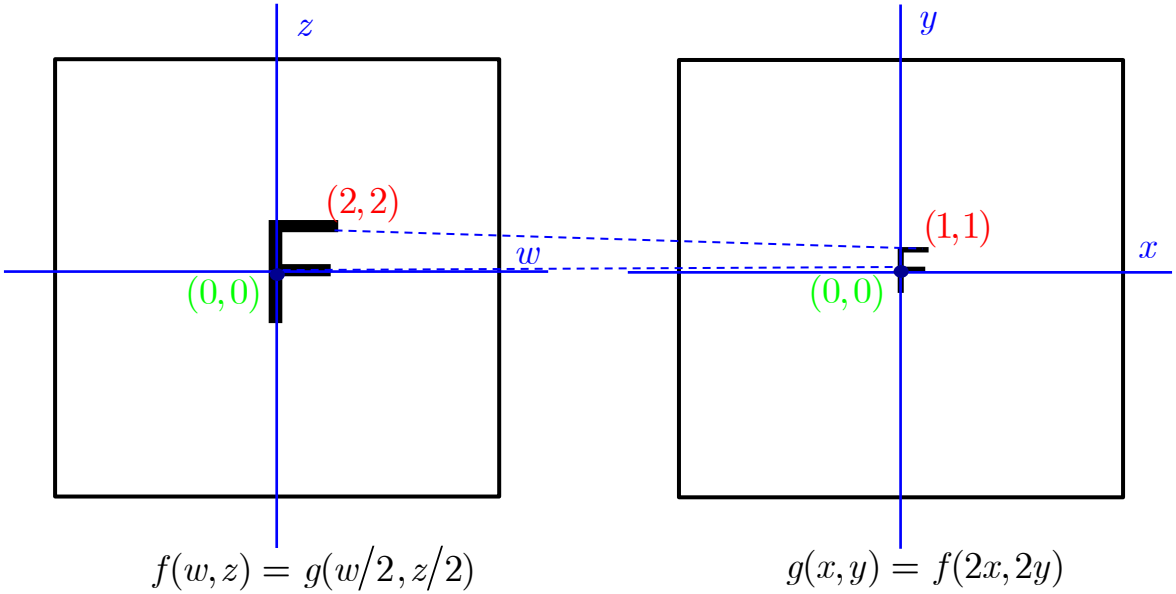


$$g(x,y) = f(x,y-b)$$
$$g(0,0) = f(0,0-3)$$
$$g(0,3) = f(0,3-3)$$

Scaling down

$$(x,y) = T\big[(w,z)\big] = (w/2, z/2)$$
$$(w,z) = T^{-1}\big[(x,y)\big] = (2x, 2y)$$

$$\Rightarrow \quad g(x,y) = f\Big(T^{-1}\big[(x,y)\big]\Big) \quad \Rightarrow \quad g(x,y) = f(2x,2y)$$

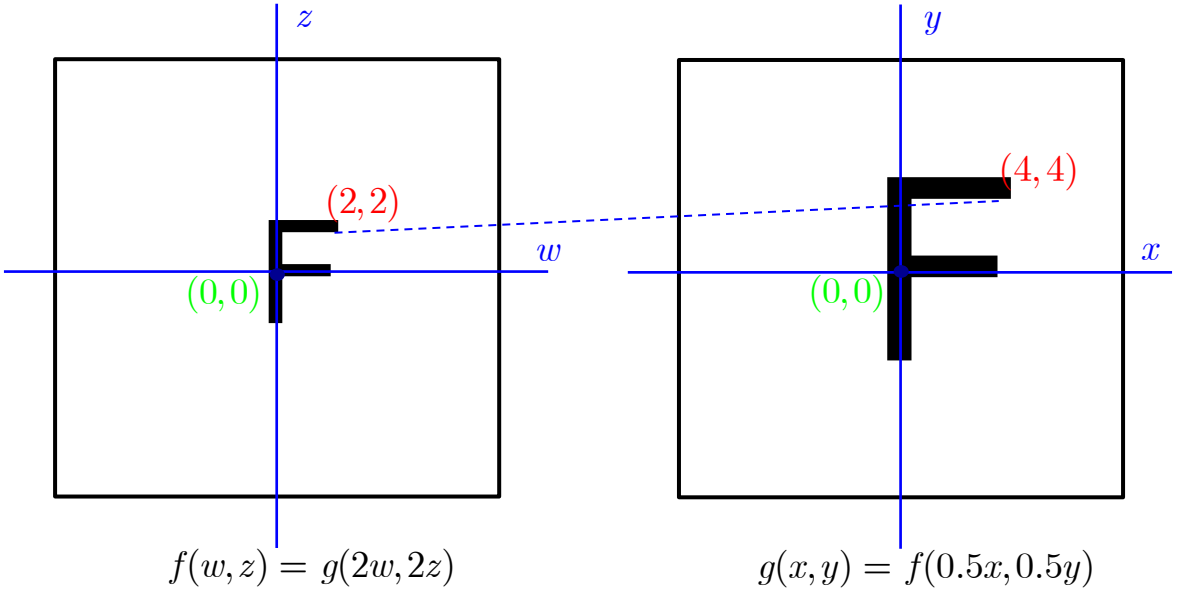


$$g(x,y) = f(2x,2y)$$
$$g(0,0) = f(0,0)$$
$$g(1,1) = f(2,2)$$

Scaling up

$$(x,y) = T\big[(w,z)\big] = (2w, 2z)$$
$$(w,z) = T^{-1}\big[(x,y)\big] = (x/2, y/2)$$

$$\Rightarrow \quad g(x,y) = f\Big(T^{-1}\big[(x,y)\big]\Big) \quad \Rightarrow \quad g(x,y) = f(x/2, y/2)$$



$f(w,z) = g(2w, 2z)$

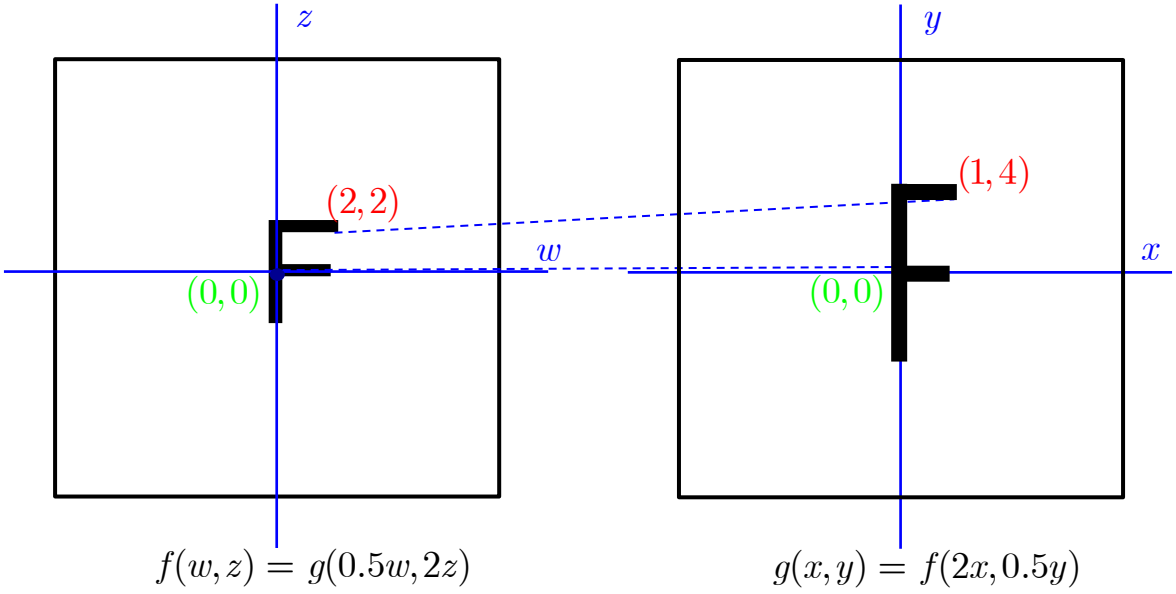
$g(x,y) = f(0.5x, 0.5y)$

$$g(x,y) = f(0.5x, 0.5y)$$
$$g(0,0) = f(0,0)$$
$$g(4,4) = f(2,2)$$

Change of aspect ratio

$$(x,y) = T\big[(w,z)\big] = (0.5w, 2z)$$
$$(w,z) = T^{-1}\big[(x,y)\big] = (2x, 0.5y)$$

$$\Rightarrow \quad g(x,y) = f\Big(T^{-1}\big[(x,y)\big]\Big) \quad \Rightarrow \quad g(x,y) = f(2x, 0.5y)$$

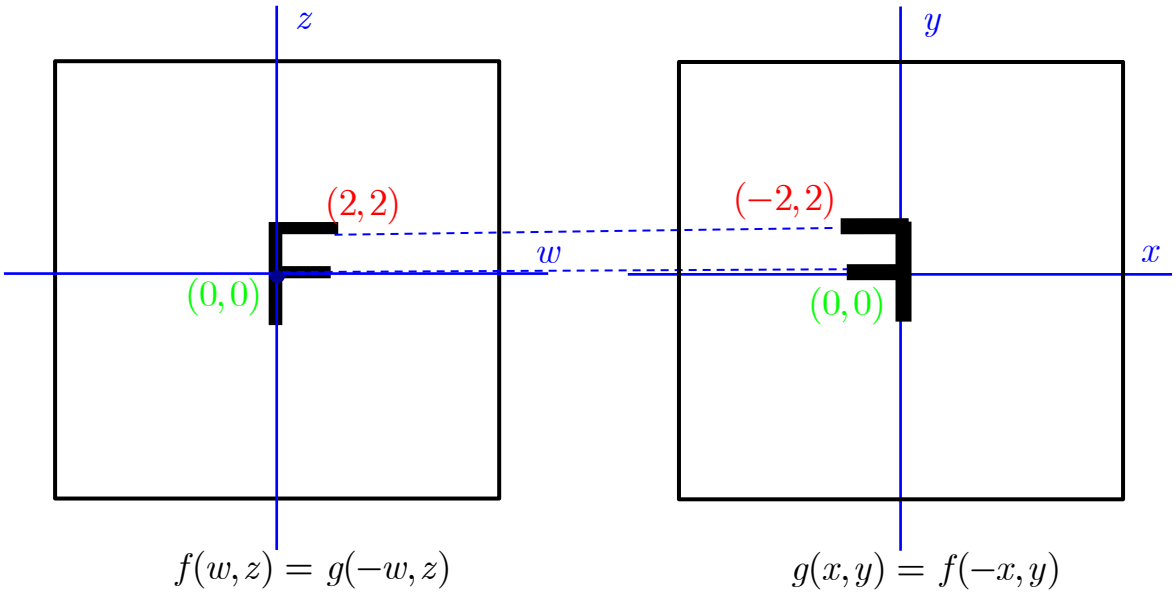


$$g(x,y) = f(2x, 0.5y)$$
$$g(0,0) = f(0,0)$$
$$g(1,4) = f(2,2)$$

Flip or reflection across an axe

$$(x,y) = T\big[(w,z)\big] = (-w,z)$$
$$(w,z) = T^{-1}\big[(x,y)\big] = (-x,y)$$

$$\Rightarrow \quad g(x,y) = f\Big(T^{-1}\big[(x,y)\big]\Big) \quad \Rightarrow \quad g(x,y) = f(-x,y)$$



$$g(x,y) = f(-x,y)$$
$$g(0,0) = f(0,0)$$
$$g(-2,2) = f(2,2)$$

Forward transform in matrix form

Affine transformation

$$\begin{bmatrix} x \\ y \end{bmatrix} = \underbrace{\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}}_{A} \begin{bmatrix} w \\ z \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

$$x = a_{11}w + a_{12}z + b_1$$
$$y = a_{21}w + a_{22}z + b_2$$

For mathematical convenience (also for Matlab), it is often defined as a single matrix multiplication – a.k.a. augmented matrix

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \left[\begin{array}{cc|c} a_{11} & a_{12} & b_1 \\ a_{21} & a_{22} & b_2 \\ 0 & 0 & 1 \end{array}\right] \begin{bmatrix} w \\ z \\ 1 \end{bmatrix}$$

$$\underbrace{\begin{bmatrix} \mathbf{x}' \\ 1 \end{bmatrix}}_{\tilde{\mathbf{x}}'} = \underbrace{\left[\begin{array}{cc|c} A & & \mathbf{b} \\ 0 & & 1 \end{array}\right]}_{\tilde{A}} \underbrace{\begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix}}_{\tilde{\mathbf{x}}} \Rightarrow \begin{array}{c} \tilde{\mathbf{x}}' = \tilde{A}\tilde{\mathbf{x}} \\ \mathbf{x}' = A\mathbf{x} + \mathbf{b} \end{array}$$

Intermediate summary

Translation

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} w \\ z \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \qquad \tilde{A} = \begin{bmatrix} 1 & 0 & b_1 \\ 0 & 1 & b_2 \\ 0 & 0 & 1 \end{bmatrix}$$

Scale

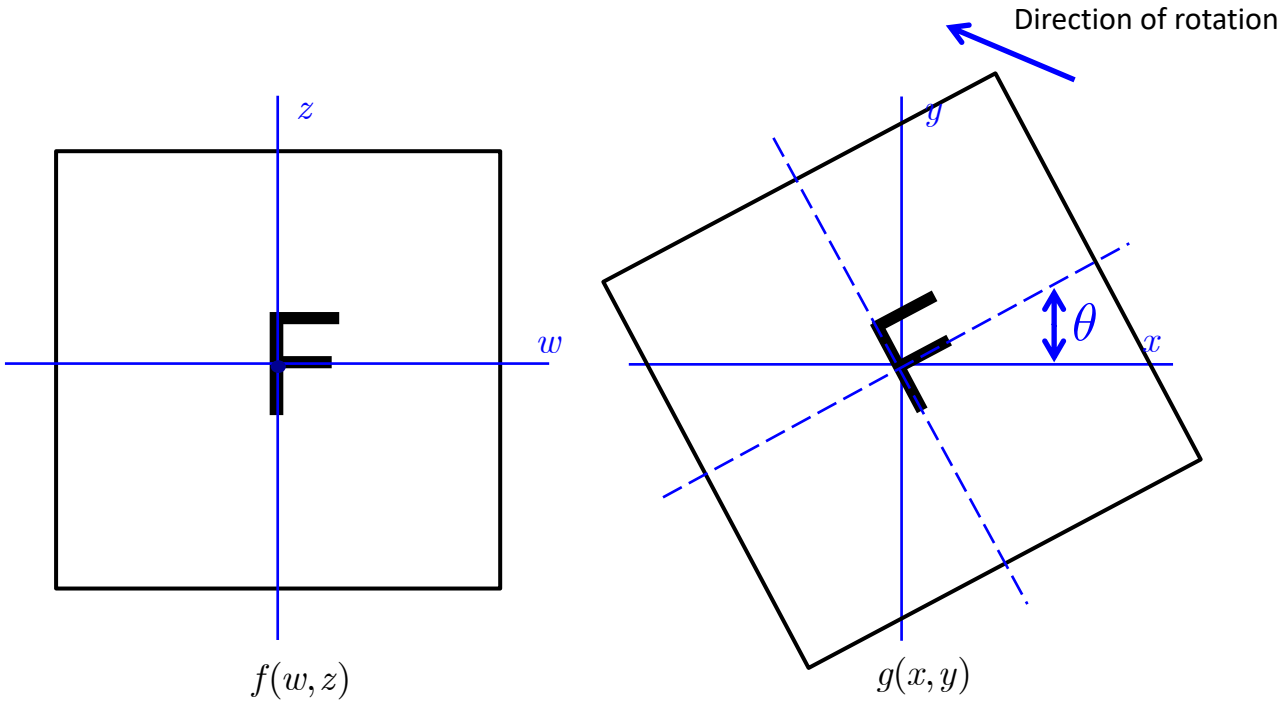
$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \alpha & 0 \\ 0 & \beta \end{bmatrix} \begin{bmatrix} w \\ z \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad \tilde{A} = \begin{bmatrix} \alpha & 0 & 0 \\ 0 & \beta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Flip

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} w \\ z \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad \tilde{A} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Rotation

Rotation by an angle θ counterclockwise (around the origin)



Rotation

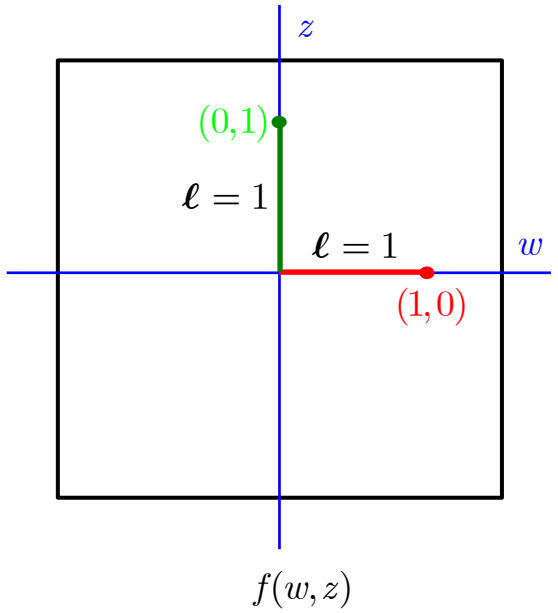
Definition

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} w \\ z \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Consider two points

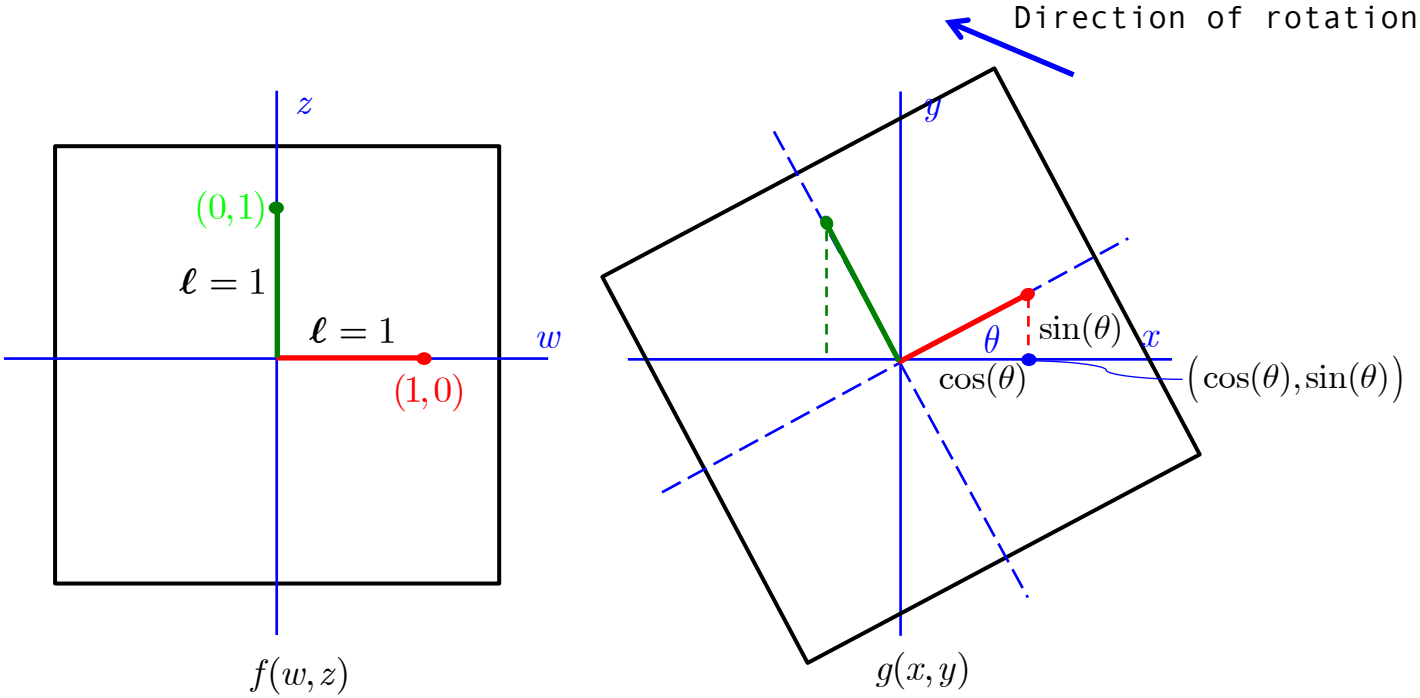
$$\begin{bmatrix} w \\ z \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \longrightarrow \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} a_{11} \\ a_{21} \end{bmatrix}$$

$$\begin{bmatrix} w \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \longrightarrow \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} a_{12} \\ a_{22} \end{bmatrix}$$



Rotation

Rotation by an angle θ counterclockwise (around the origin)

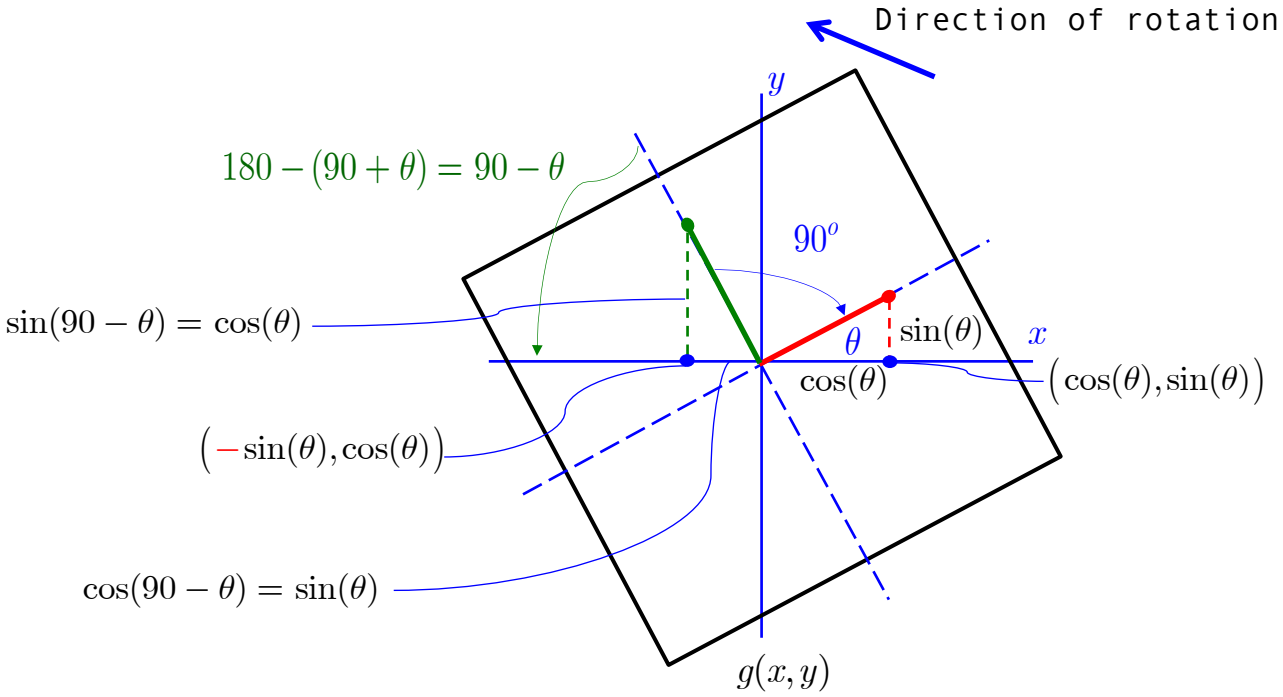


Rotation

Rotation by an angle θ counterclockwise (around the origin)

$$\sin(\theta) = \cos\left(\frac{\pi}{2} - \theta\right)$$

$$\cos(\theta) = \sin\left(\frac{\pi}{2} - \theta\right)$$



Rotation

Rotation by an angle θ counterclockwise (around the origin)

$$\begin{bmatrix} w \\ z \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \rightarrow \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix}$$

$$\begin{bmatrix} w \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} -\sin(\theta) \\ \cos(\theta) \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \underbrace{\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}}_{A} \begin{bmatrix} w \\ z \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Remark: $A^T A = I$

$A^{-1} = A^T$

Properties of considered transforms

- Any combination of *scale, shift, flip* and *rotation* is called **a similarity transform**

  It preserves parallel lines.
- If $a_{11,}a_{2,2} = \pm 1$, then this transform is also called isometric transform.

  Wiki: the Greek – isometric stands for “equal measures”, reflecting that the scale along each axis of transform is the same.

“Bending”

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} w \\ z \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Suppose we have the dependence on the axes

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 & b \\ 0 & 1 \end{bmatrix} \begin{bmatrix} w \\ z \end{bmatrix} = \begin{bmatrix} w + bz \\ z \end{bmatrix}$$

“Bending” via horizontal shearing

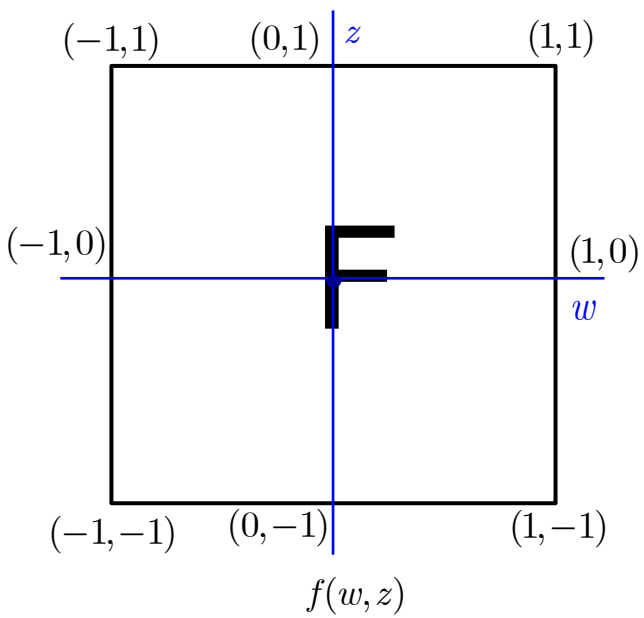
$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} w + bz \\ z \end{bmatrix}$$

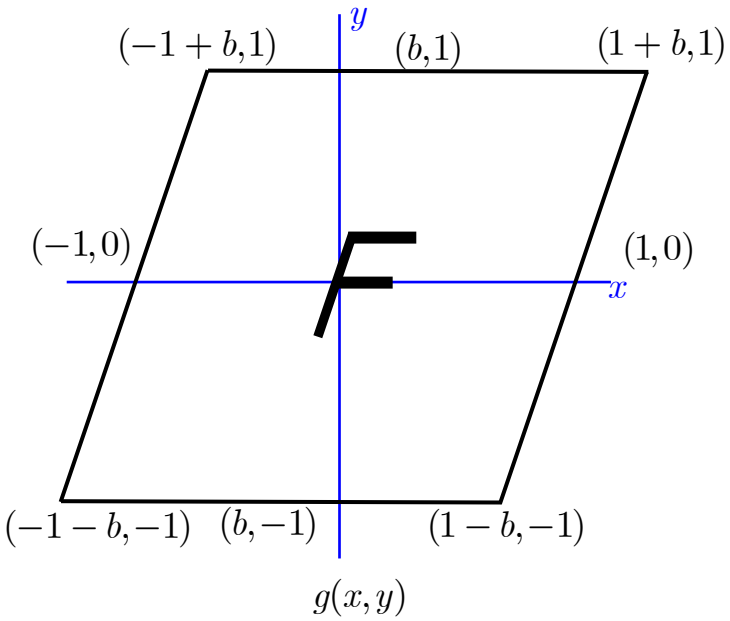
$$\begin{bmatrix} 0 \\ 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

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

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} b \\ 1 \end{bmatrix}$$

etc.

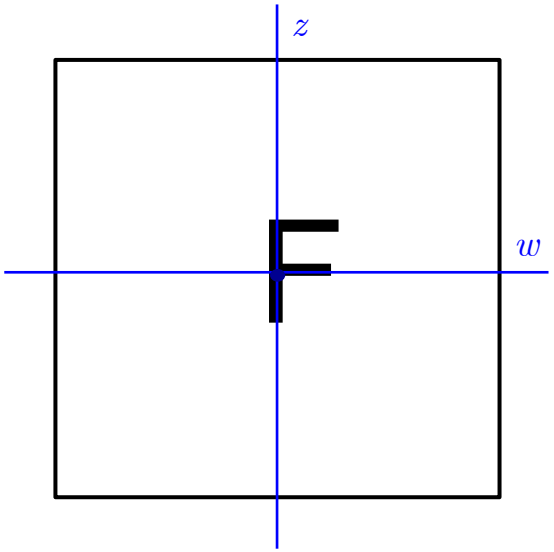




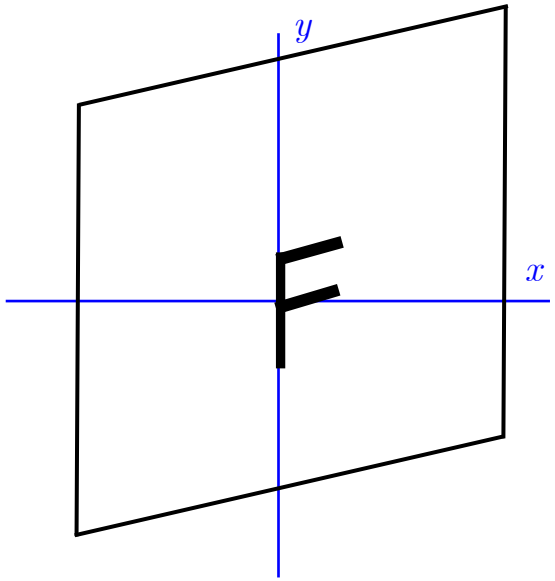
If b is negative, the shear will be in the opposite direction.

“Bending” via vertical shearing

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ c & 1 \end{bmatrix} \begin{bmatrix} w \\ z \end{bmatrix}$$



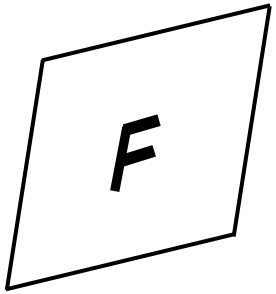
$f(w,z)$



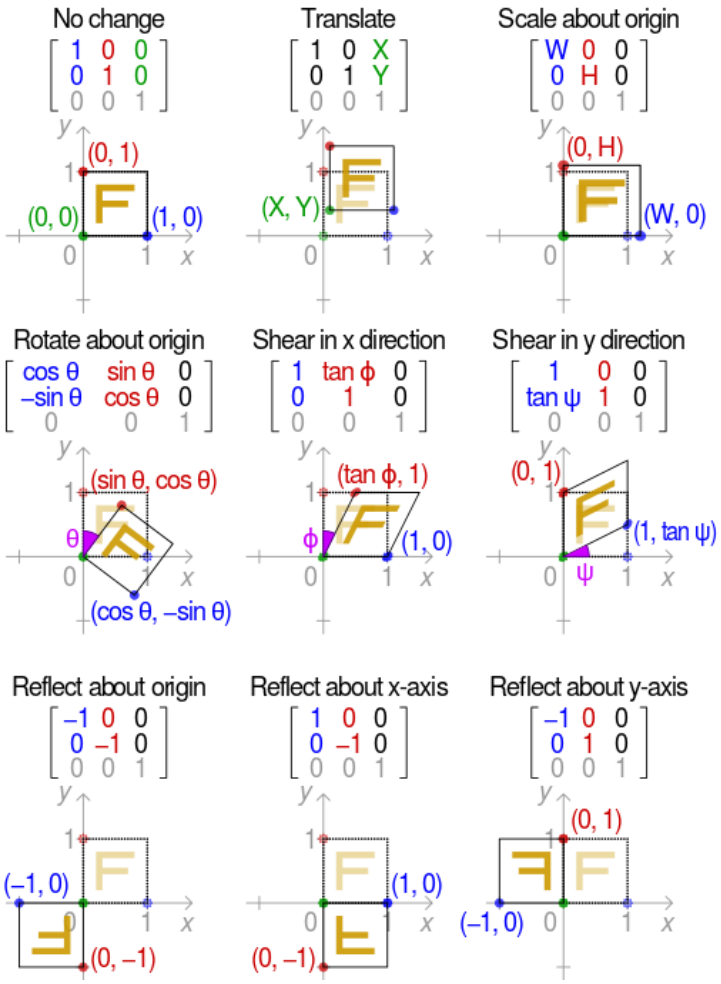
$g(x,y)$

Joint shear

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 & b \\ c & 1 \end{bmatrix} \begin{bmatrix} w \\ z \end{bmatrix}$$



Summary



$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & b_1 \\ a_{21} & a_{22} & b_2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} w \\ z \\ 1 \end{bmatrix}$$

Note: pay attention how θ and ϕ are defined; directions of rotation and shearing.

https://en.wikipedia.org/wiki/Affine_transformation#/media/File:2D_affine_transformation_matrix.svg

Matlab demo of affine transformations

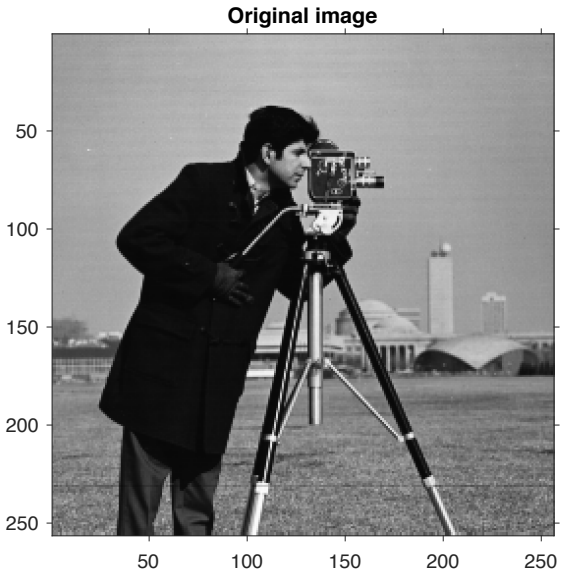
5.1. Demo in Matlab

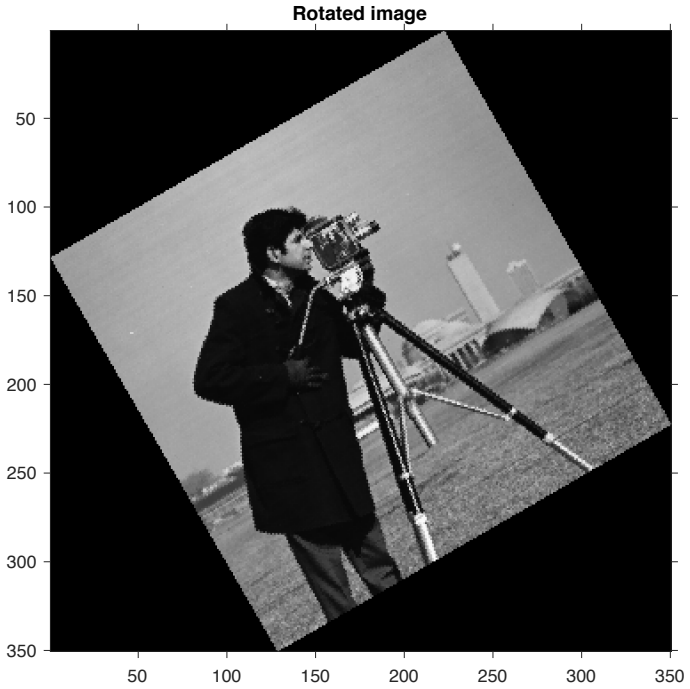
Image rotation

demo_rotation.m

```
1     % Image rotation
2     % doc imrotate
3
4     % Read image
5 -   im=imread('cameraman.tif');
6
7     % Show image
8 -   figure(1); imshow(im); title('Original image')
9
10    % Rotate image
11 -  out=imrotate(im, 30);
12
13    % Show image
14 -  figure(2); imshow(out); title('Rotated image')
15
```

5.1. Demo in Matlab

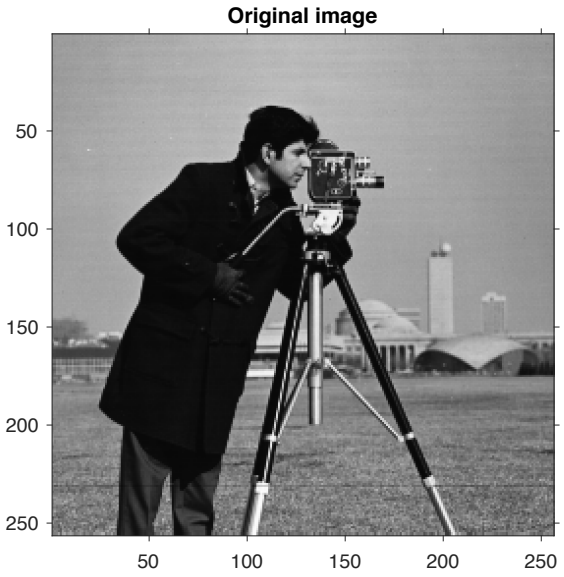


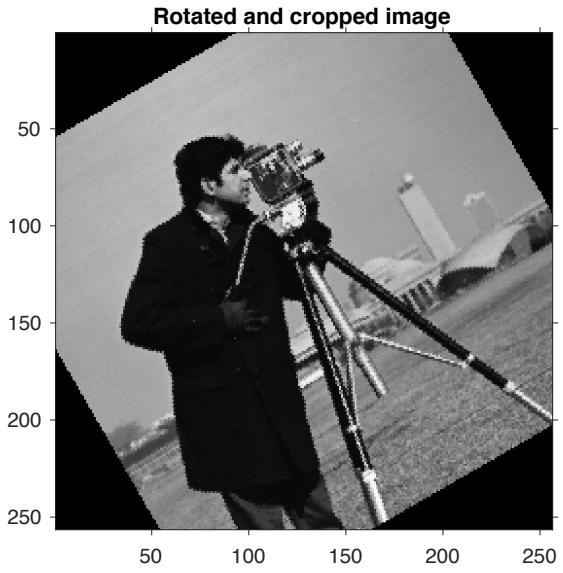


Remarks:

- Padding of rotated figure by zeros
- Quality of rotated image

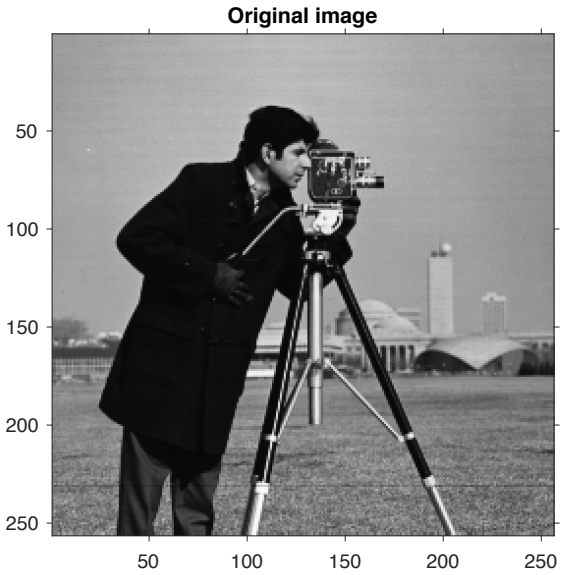
5.1. Demo in Matlab

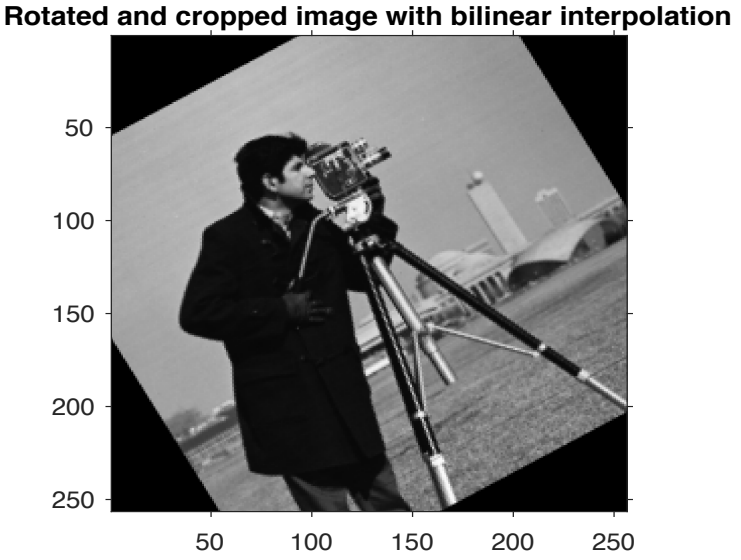





```
% Rotate image (crop)
out=imrotate(im, 30, 'crop');

% Show image
figure(3); imshow(out); title('Rotated and cropped image')
```

5.1. Demo in Matlab






```
% Rotate image (bilinear interpolation)
out=imrotate(im, 30, 'bilinear','crop');

% Show image
figure(4); imshow(out); title('Rotated and cropped image with bilinear interpolation')
```

5.1. Demo in Matlab

```
>> doc imrotate
>>
```

imrotate

Rotate image

co

Syntax

```
B = imrotate(A,angle)
B = imrotate(A,angle,method)
B = imrotate(A,angle,method,bbox)
gpuarrayB = imrotate(gpuarrayA,method)
```

Description

`B = imrotate(A,angle)` rotates image A by `angle` degrees in a counterclockwise direction around its center point. To rotate the image clockwise, specify a negative value for `angle`. `imrotate` makes the output image B large enough to contain the entire rotated image. `imrotate` uses nearest neighbor interpolation, setting the values of pixels in B that are outside the rotated image to `0` (zero).

`B = imrotate(A,angle,method)` rotates image A, using the interpolation method specified by `method`.

`B = imrotate(A,angle,method,bbox)` rotates image A, where `bbox` specifies the size of the returned image.

Demo in Matlab - Alternative way of image rotation

Note the difference in notations in Matlab:

affine2d class

2-D Affine Geometric Transformation

Description

An `affine2d` object encapsulates a 2-D affine geometric transformation.

Code Generation support: Yes.

MATLAB Function Block support: Yes.

Construction

`tform = affine2d()` creates an `affine2d` object with default property settings that correspond to the identity transformation.

`tform = affine2d(A)` creates an `affine2d` object given an input 3-by-3 matrix A that specifies a valid affine transformation.

Input Arguments

A	3-by-3 matrix that specifies a valid affine transformation of the form:
	`A = [a b 0;`
	`c d 0;`
	`e f 1];`
	Default: Identity transformation

Properties

T	3-by-3 double-precision, floating point matrix that defines the 2-D forward affine transformation
	The matrix T uses the convention:
	`[x y 1] = [u v 1] * T`
	where T has the form:
	`[a b 0;`
	`c d 0;`
	`e f 1];`

`Dimensionality`	Describes the dimensionality of the geometric transformation for both input and output points

Demo in Matlab - Alternative way of image rotation

Our notation

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} w \\ z \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

$$x = a_{11}w + a_{12}z + b_1$$
$$y = a_{21}w + a_{22}z + b_2$$

Matlab uses a transposed version

$$\begin{bmatrix} x \\ y \end{bmatrix}^T = \left(\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} w \\ z \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \right)^T = \begin{bmatrix} w \\ z \end{bmatrix}^T \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}^T + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}^T$$

$$\begin{bmatrix} x & y \end{bmatrix} = \begin{bmatrix} w & z \end{bmatrix} \begin{bmatrix} a_{11} & a_{21} \\ a_{12} & a_{22} \end{bmatrix} + \begin{bmatrix} b_1 & b_2 \end{bmatrix}$$

$$x = a_{11}w + a_{12}z + b_1$$
$$y = a_{21}w + a_{22}z + b_2$$

Demo in Matlab - Alternative way of image rotation

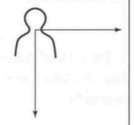
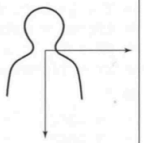
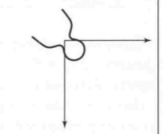
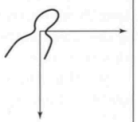
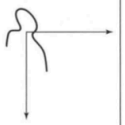
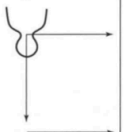
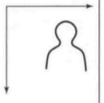
Our notation

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & b_1 \\ a_{21} & a_{22} & b_2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} w \\ z \\ 1 \end{bmatrix}$$

Matlab

$$\begin{bmatrix} x & y & 1 \end{bmatrix} = \begin{bmatrix} w & z & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{21} & 0 \\ a_{12} & a_{22} & 0 \\ b_1 & b_2 & 1 \end{bmatrix}$$

Matlab affine matrixes

Type	Affine Matrix, T	Coordinate Equations	Diagram
Identity	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x = w$ $y = z$	
Scaling	$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x = s_x w$ $y = s_y z$	
Rotation	$\begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x = w\cos\theta - z\sin\theta$ $y = w\sin\theta + z\cos\theta$	
Shear (horizontal)	$\begin{bmatrix} 1 & 0 & 0 \\ \alpha & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x = w + \alpha z$ $y = z$	
Shear (vertical)	$\begin{bmatrix} 1 & \beta & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x = w$ $y = \beta w + z$	
Vertical reflection	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x = w$ $y = -z$	
Translation	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \delta_x & \delta_y & 1 \end{bmatrix}$	$x = w + \delta_x$ $y = z + \delta_y$	

Gonzalez, p. 286

Affine2d description

Description

An `affine2d` object encapsulates a 2-D affine geometric transformation.

Code Generation support: Yes.

MATLAB Function Block support: Yes.

Construction

`tform = affine2d()` creates an `affine2d` object with default property settings that correspond to the identity transformation.

`tform = affine2d(A)` creates an `affine2d` object given an input 3-by-3 matrix A that specifies a valid affine transformation.

Input Arguments

`A`	3-by-3 matrix that specifies a valid affine transformation of the form:
	`A = [a b 0;`
	`c d 0;`
	`e f 1];`
	Default: Identity transformation

$$\begin{bmatrix} x & y & 1 \end{bmatrix} = \begin{bmatrix} w & z & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{21} & 0 \\ a_{12} & a_{22} & 0 \\ b_1 & b_2 & 1 \end{bmatrix}$$

Properties

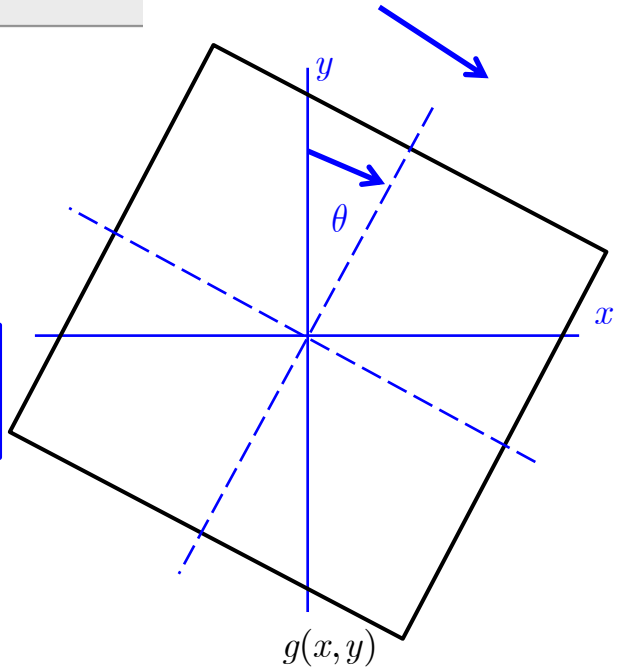
`T`	3-by-3 double-precision, floating point matrix that defines the 2-D forward affine transformation
	The matrix T uses the convention:
	`[x y 1] = [u v 1] * T`
	where T has the form:
	`[a b 0;`
	`c d 0;`
	`e f 1];`

`Dimensionality`	Describes the dimensionality of the geometric transformation for both input and output points

5.1. Demo in Matlab

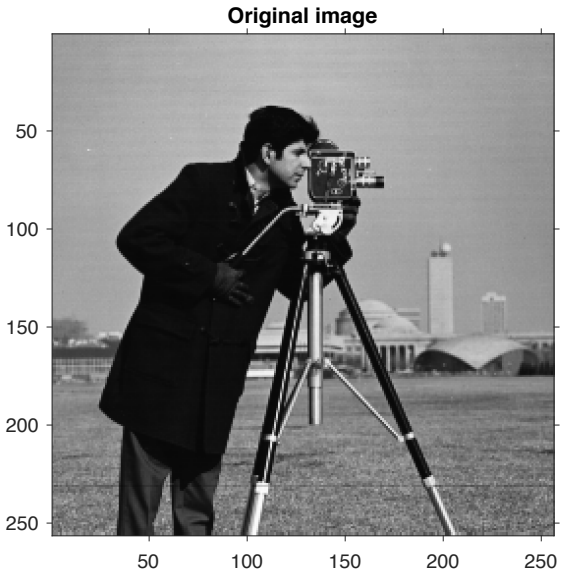
demo_affine2d.m*

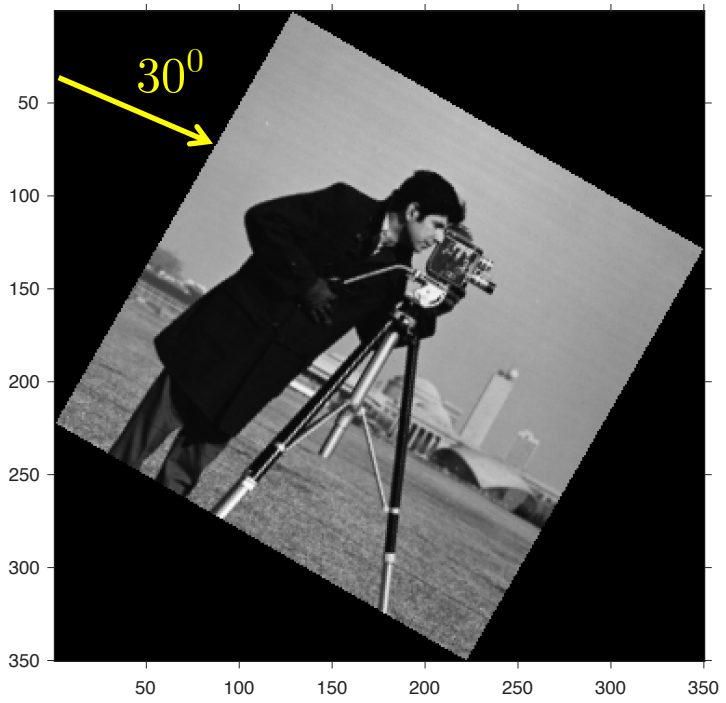
```
3
4     % Read image
5 -   im=imread('cameraman.tif');
6
7     % Show image
8 -   figure(1); imshow(im); title('Original image')
9
10 -  deg=30;
11
12    % Define the affine matrix (matrix of rotation)
13 -  T=[cosd(deg) sind(deg) 0; -sind(deg) cosd(deg) 0 ; 0 0 1];
14 -  Ta=affine2d(T);
15
16    % Rotate image (simpl
17 -  out=imwarp(im, Ta);
18
19    % Show image
20 -  figure(2); imshow(out); title('Rotated image')
21
22
```



$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} w \\ z \\ 1 \end{bmatrix} \quad \begin{bmatrix} x & y \end{bmatrix} = \begin{bmatrix} w & z \end{bmatrix} \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

5.1. Demo in Matlab





Remark: Matlab defines the angle of rotation clockwise!
and 90 degree “minus” our definition

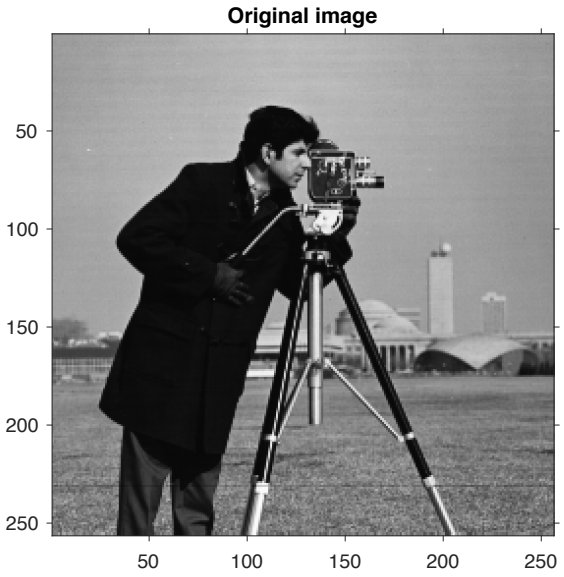
5.1. Demo in Matlab

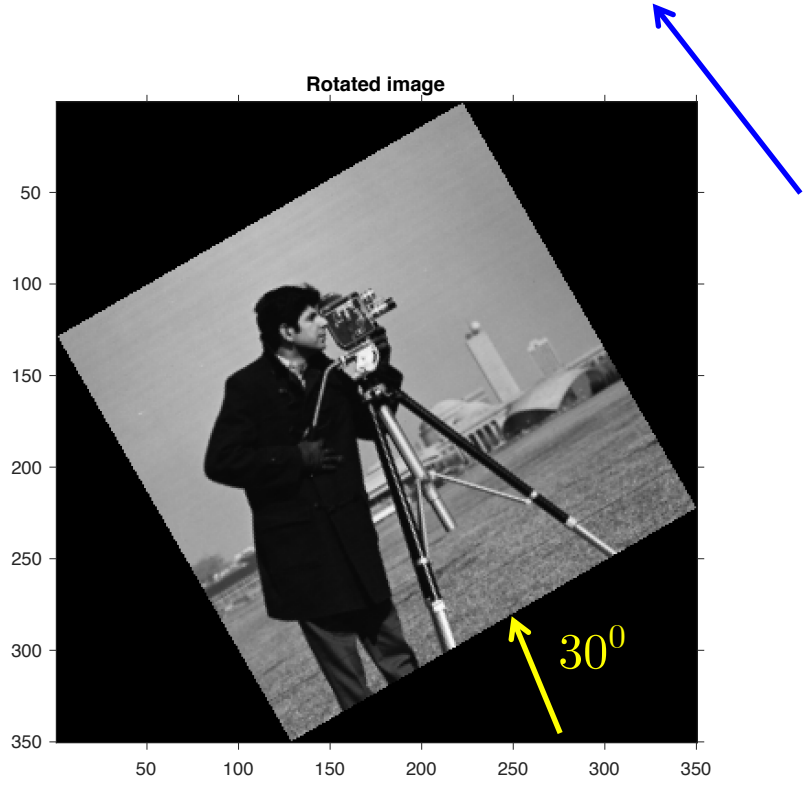
demo_affine2d.m

```
% Image rotation with affine2d
% doc affine2d

% Read image
im=imread('cameraman.tif');

% Show image
figure(1); imshow(im); title('Original image')

deg=30;

% Define the affine matrix (matrix of rotation)
T=[cosd(deg) -sind(deg) 0; sind(deg) cosd(deg) 0 ; 0 0 1];
Ta=affine2d(T);

% Rotate image (simpl
out=imwarp(im, Ta);

% Show image
figure(2); imshow(out); title('Rotated image')
```

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} w \\ z \\ 1 \end{bmatrix} \quad \begin{bmatrix} x & y \end{bmatrix} = \begin{bmatrix} w & z \end{bmatrix} \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

5.1. Demo in Matlab



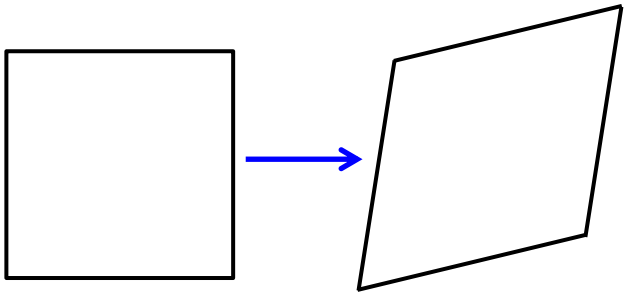


Projective transformations

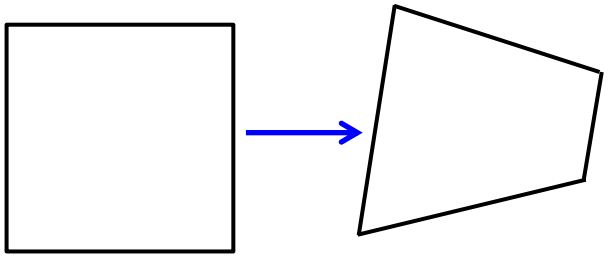
We considered affine transformations of form

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} w \\ z \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

Max degree of distortion is in a form of parallelogram (parallel lines stay parallel)

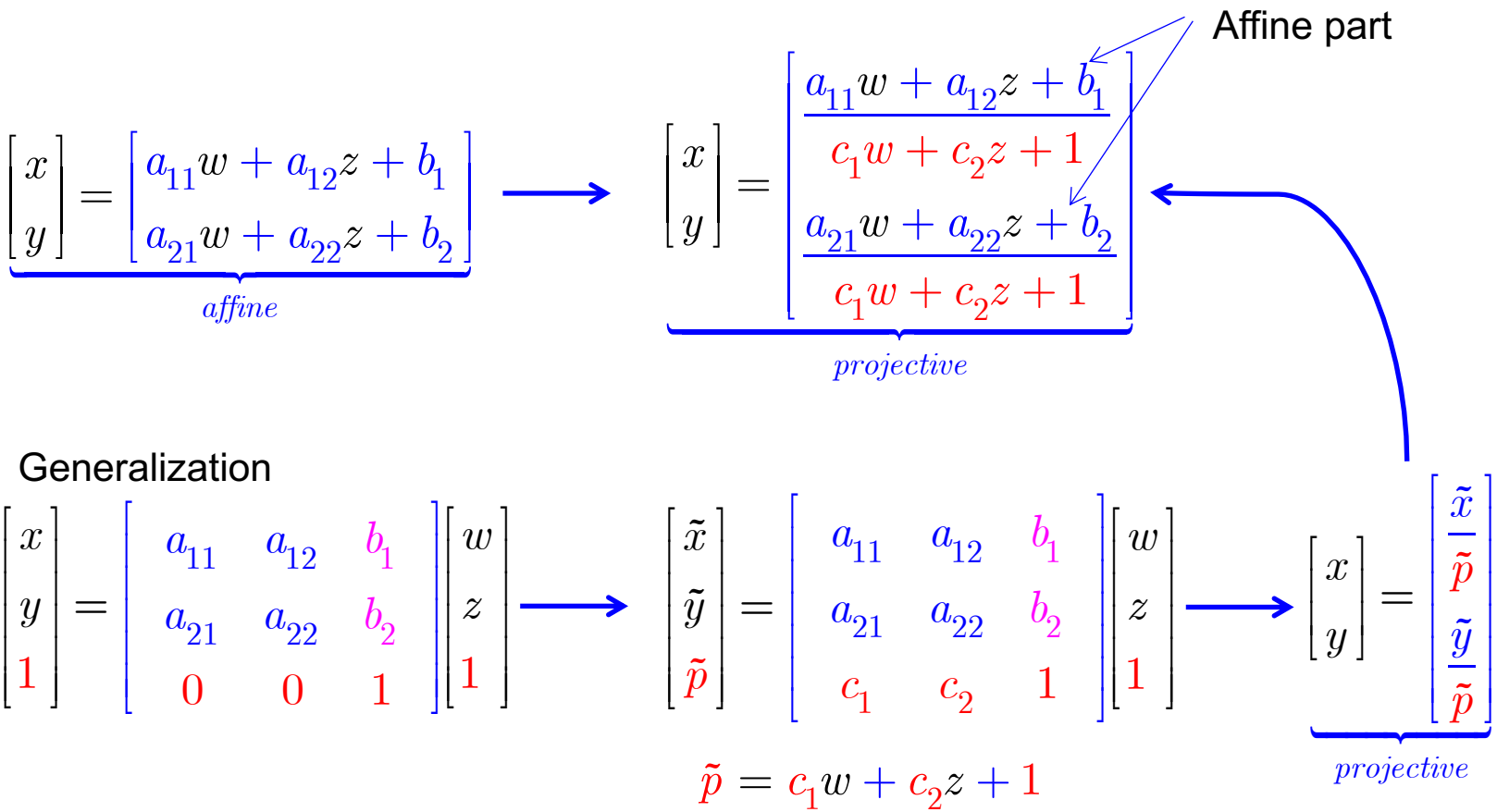


Projective transformations – more general form



Projective transformations

We considered affine transformations of form



Projective transformations

2-D Projective Geometric Transformation

Description

A `projective2d` object encapsulates a 2-D projective geometric transformation.

Code Generation support: Yes.

MATLAB Function Block support: Yes.

Construction

`tform = projective2d()` creates a `projective2d` object with default property settings that correspond to the identity transformation.

`tform = projective2d(A)` creates a `projective2d` object given an input 3-by-3 matrix A that specifies a valid projective transformation.

Input Arguments

A	3-by-3 matrix that specifies a valid projective transformation of the form:
	`A = [a b c;`
	`d e f;`
	`g h i]`
	Default: Identity transformation

$$\begin{bmatrix} \tilde{x} \\ \tilde{y} \\ p_1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & b_1 \\ a_{21} & a_{22} & b_2 \\ c_1 & c_2 & 1 \end{bmatrix} \begin{bmatrix} w \\ z \\ 1 \end{bmatrix}$$

Properties

T	3-by-3 double-precision, floating point matrix that defines the 2-D forward projective transformation.
	The matrix T uses the convention:
	`[x y 1] = [u v 1] * T`
	where T has the form:
	`[a b c;...`
	`d e f;...`
	`g h i];`

In the transposed form!

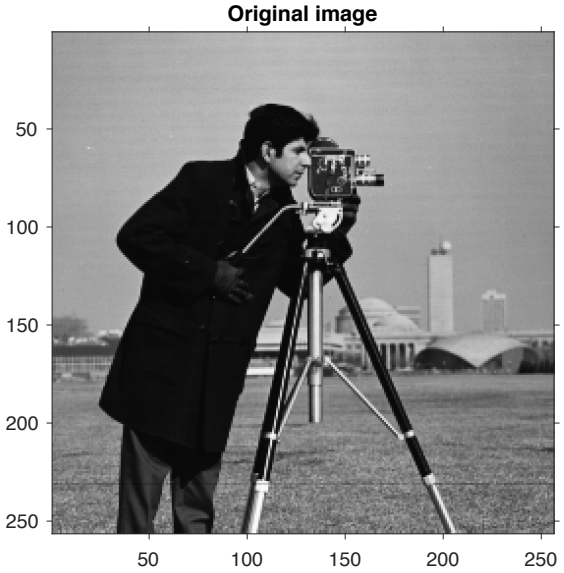
$$\begin{bmatrix} \tilde{x} & \tilde{y} & p_1 \end{bmatrix} = \begin{bmatrix} w & z & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{21} & c_1 \\ a_{12} & a_{22} & c_2 \\ b_1 & b_2 & 1 \end{bmatrix}$$

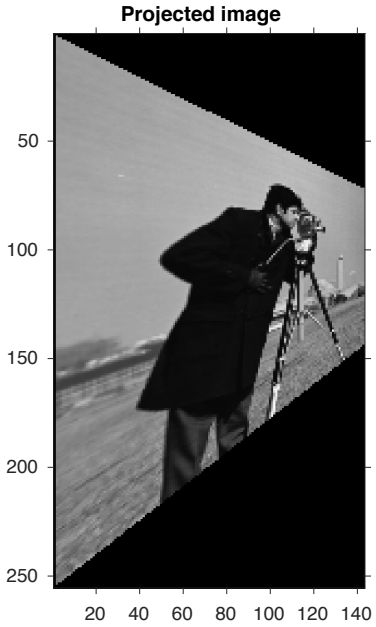
5.1. Demo in Matlab

demo_projective2d.m

```
% Image projective transformations
% doc projective2d

% Read image
im=imread('cameraman.tif');

% Show image
figure(1); imshow(im); title('Original image')

% Define the projective matrix (pay attention to the faft that Matlab uses a transposed form!)
T=[ 1 0 0.01; 0 1 0; -100 -100 1];
T=projective2d(T);

% Apply projective transform
out=imwarp(im, T);

% Show image
figure(2); imshow(out); title('Projected image')
```

5.1. Demo in Matlab



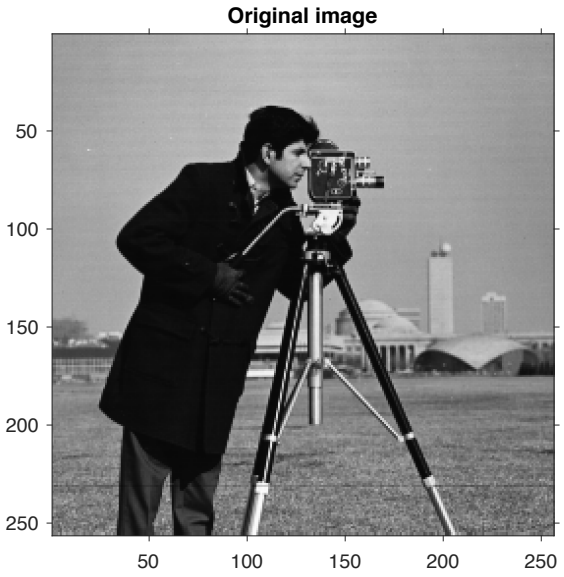


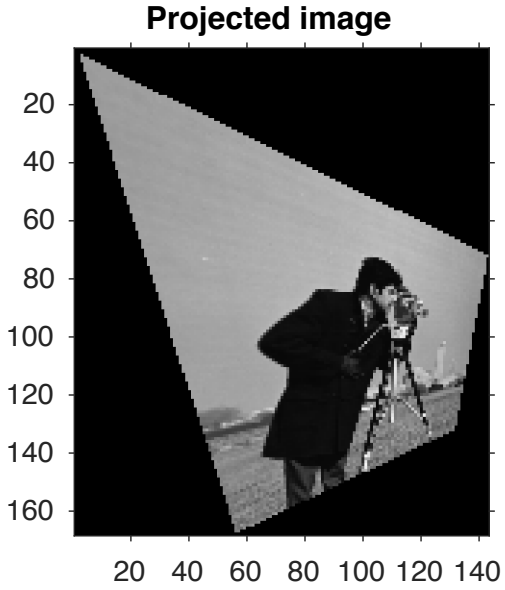
5.1. Demo in Matlab

demo_projective2d.m

```
1   % Image projective transformations
2   % doc projective2d
3
4   % Read image
5   im=imread('cameraman.tif');
6
7   % Show image
8   figure(1); imshow(im); title('Original image')
9
10  % Define the projective matrix (pay attention to the faft that Matlab uses a transposed form!)
11  T=[ 1 0 0.01; 0 1 0.01; -100 -100 1];
12  T=projective2d(T);
13
14  % Apply projective transform
15  out=imwarp(im, T);
16
17  % Show image
18  figure(2); imshow(out); title('Projected image')
19
```

5.1. Demo in Matlab



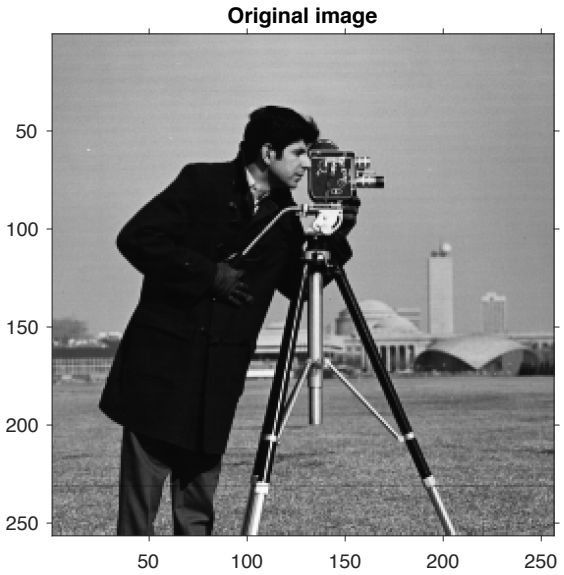


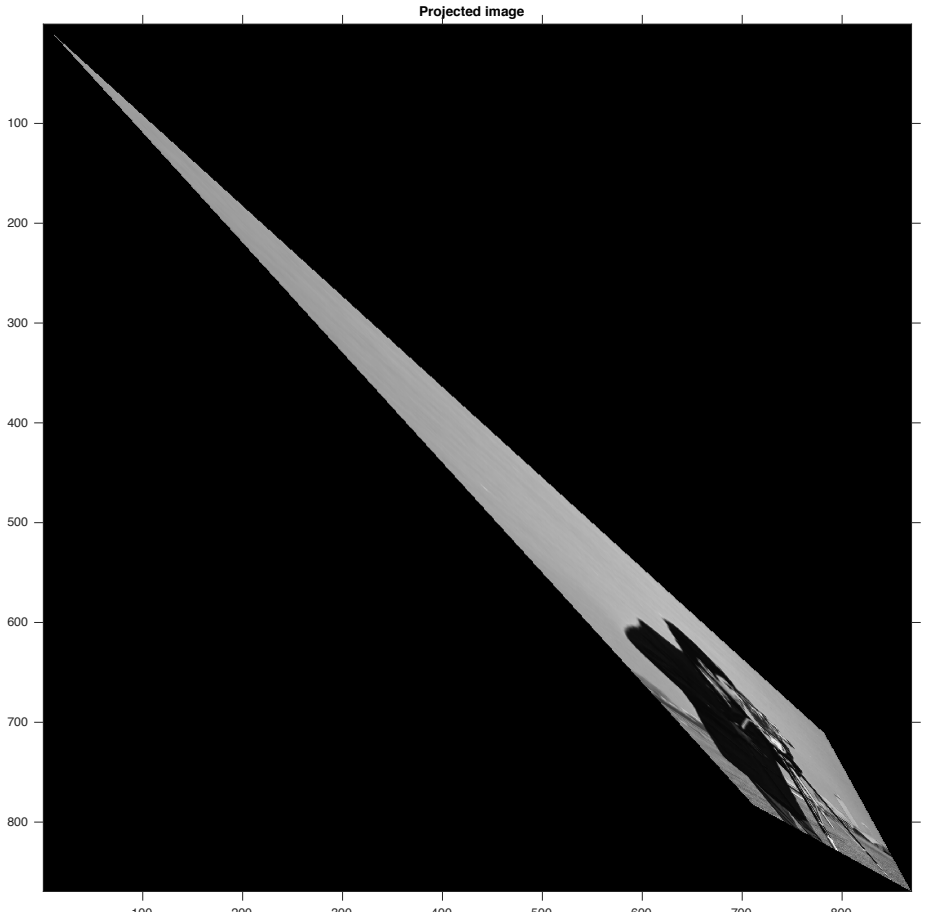
5.1. Demo in Matlab

demo_projective2d.m

```
1    % Image projective transformations
2    % doc projective2d
3
4    % Read image
5    im=imread('cameraman.tif');
6
7    % Show image
8    figure(1); imshow(im); title('Original image')
9
10   % Define the projective matrix (pay attention to the faft that Matlab uses a transposed form!)
11   T=[ 1 0 0.01; 0 1 0.01; -1000 -1000 1];
12   T=projective2d(T);
13
14   % Apply projective transform
15   out=imwarp(im, T);
16
17   % Show image
18   figure(2); imshow(out); title('Projected image')
19
```

5.1. Demo in Matlab

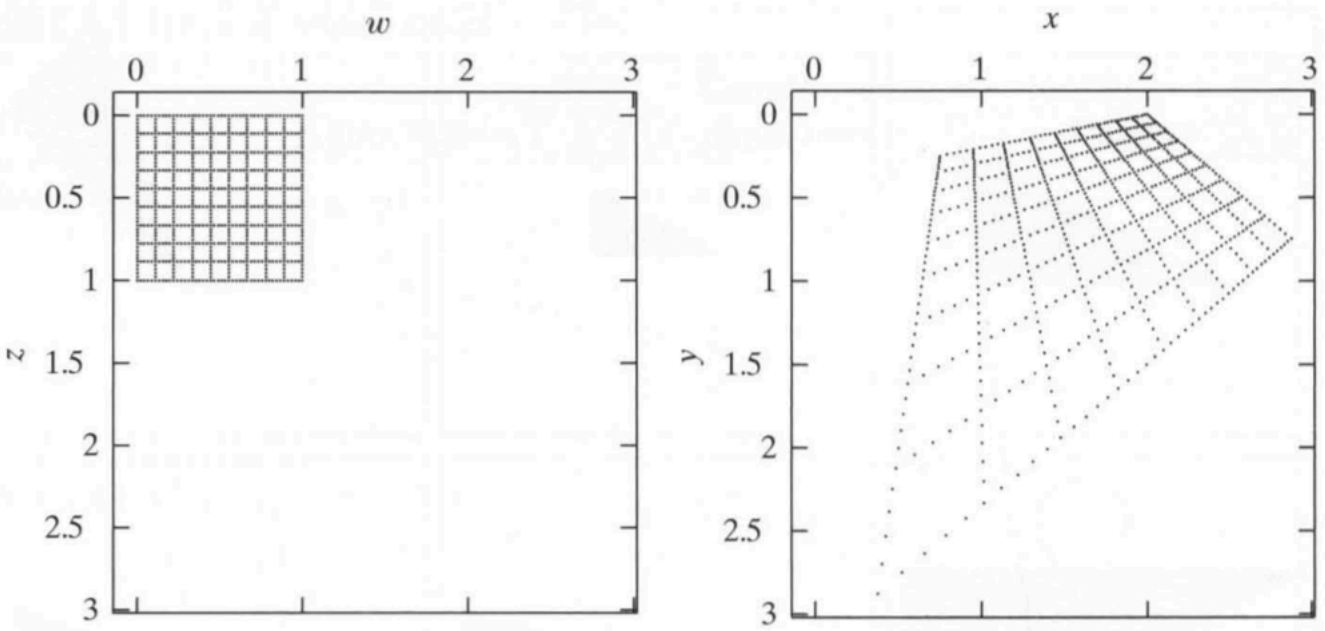




5.1. Interpretation of projective transform

a b

FIGURE 6.5
Example of a projective transformation. (a) Point grid in input space. (b) Transformed point grid in output space.



Gonzalez, p.288

5.1. Interpretation of projective transform

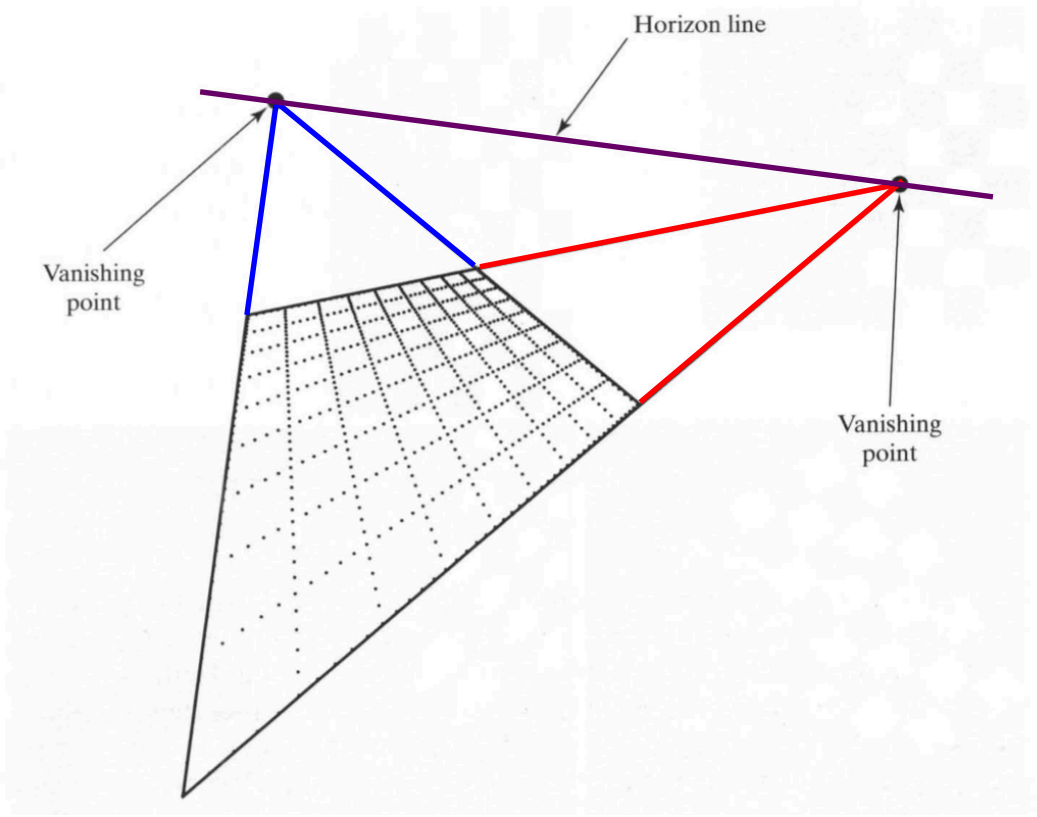


FIGURE 6.6
Vanishing points and the horizon line for a projective transformation.

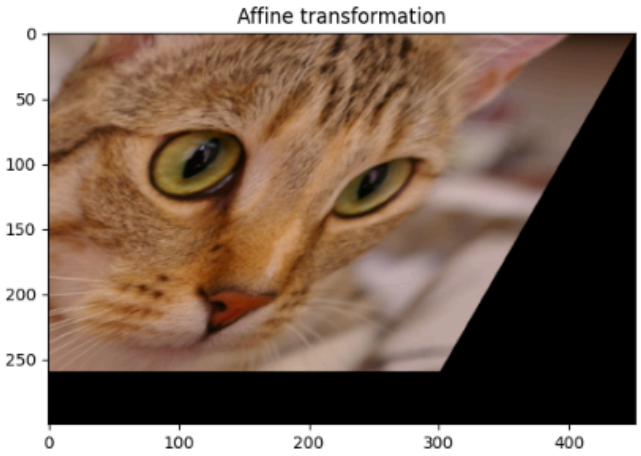
Gonzalez, p.289

Operations in Python: skimage

Affine transformation

An affine transformation preserves lines (hence the alignment of objects), as well as parallelism between lines. It can be decomposed into a similarity transform and a shear transformation.

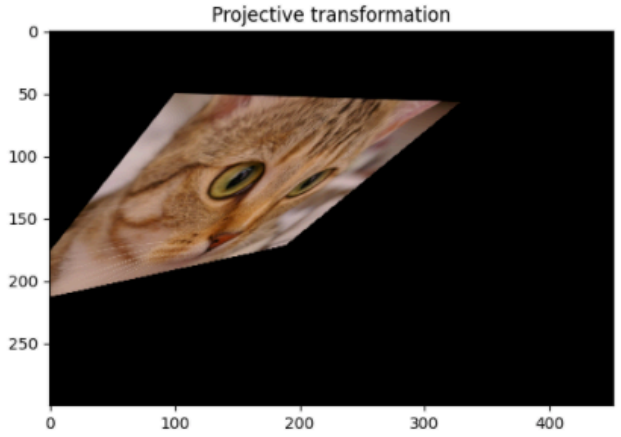
```
tform = transform.AffineTransform(
        shear=np.pi/6,
        )
print(tform.params)
tf_img = transform.warp(img, tform.inverse)
fig, ax = plt.subplots()
ax.imshow(tf_img)
_ = ax.set_title('Affine transformation')
```



Projective transformation (homographies)

A homography, also called projective transformation, preserves lines but not necessarily parallelism.

```
matrix = np.array([[1, -0.5, 100],
                   [0.1, 0.9, 50],
                   [0.0015, 0.0015, 1]])
tform = transform.ProjectiveTransform(matrix=matrix)
tf_img = transform.warp(img, tform.inverse)
fig, ax = plt.subplots()
ax.imshow(tf_img)
ax.set_title('Projective transformation')

plt.show()
```



Details about the output image

Real situation

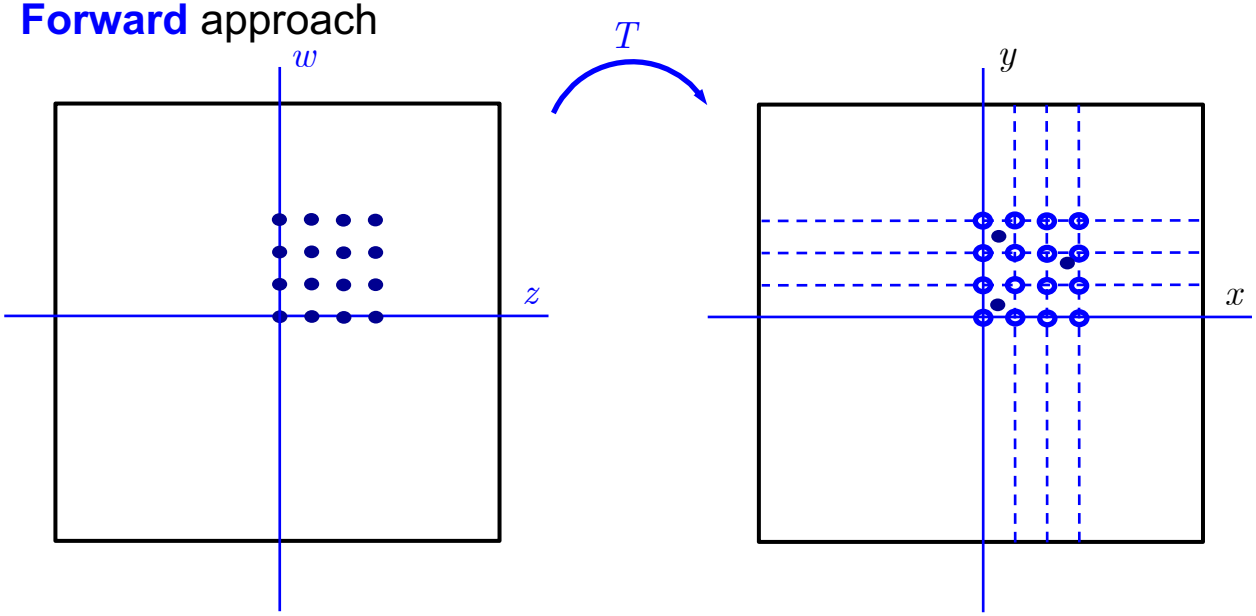
$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0.1 & 1 \\ 2 & 0.4 \end{bmatrix} \begin{bmatrix} w \\ z \end{bmatrix} + \begin{bmatrix} -1.2 \\ 5.3 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} -0.1 \\ 7.7 \end{bmatrix}$$

Problem: Integer grid goes to real grid but we have only integer grid representation.

Interpolation

Forward approach



Points are on the integer grid

New points are not on the integer grid!

How to estimate image intensity/color on the new grid?

Solution 1: find the closest point on the new (circle) and assign the intensity from the input image to it. Problem: low image quality.

Solution 2: use "**backward**" mapping assuming that the geometrical transform is *invertible*

Interpolation

"**Backward**" approach

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} w \\ z \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y \end{bmatrix} = \underbrace{A}_{2\times 2} \begin{bmatrix} w \\ z \end{bmatrix} + \underbrace{\mathbf{b}}_{2\times 1}$$

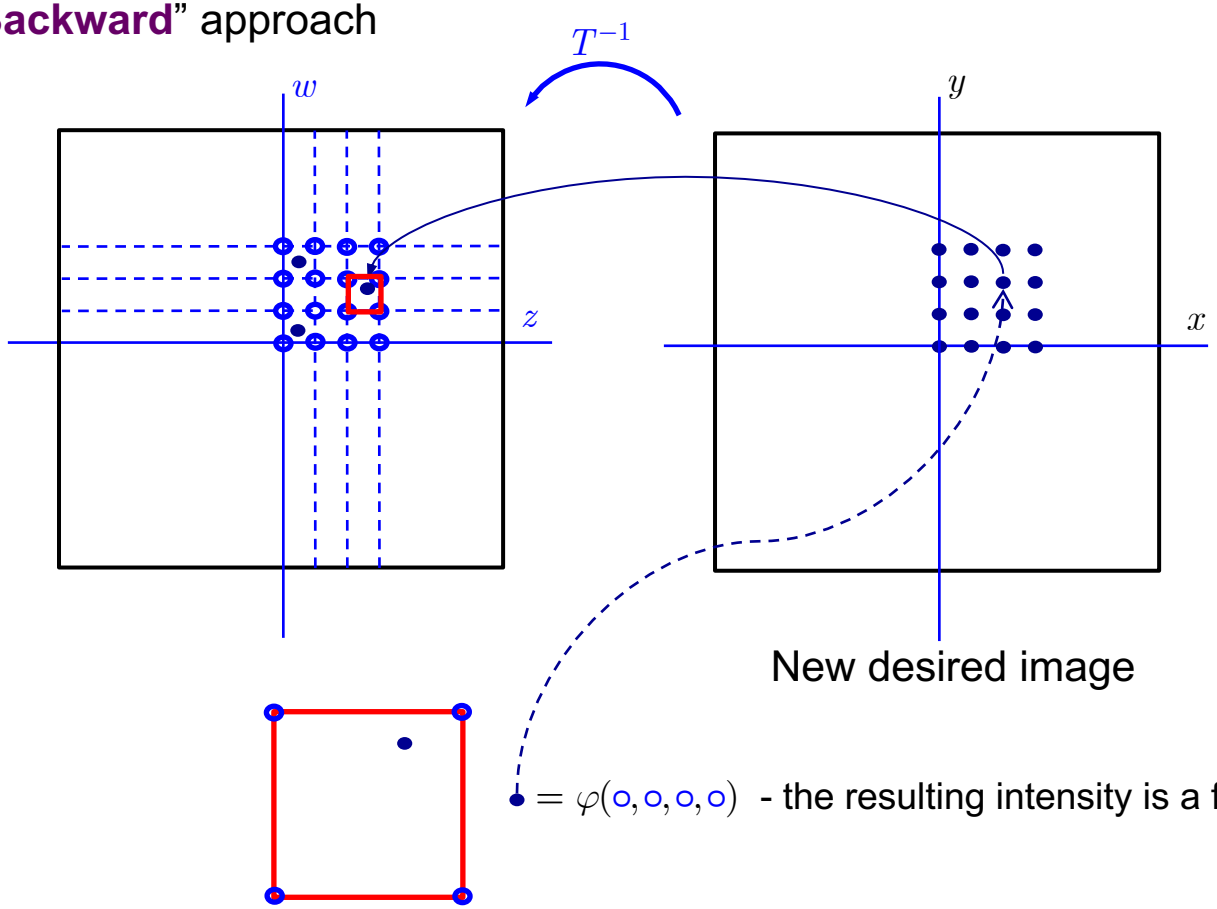
$$\begin{bmatrix} w \\ z \end{bmatrix} = A^{-1}\left(\begin{bmatrix} x \\ y \end{bmatrix} - \mathbf{b} \right)$$

$$= A^{-1} \begin{bmatrix} x \\ y \end{bmatrix} - A^{-1}\mathbf{b}$$

Note: the affine transform is invertible!

Interpolation

"**Backward**" approach



As a result we have a regular grid coverage without holes.

Methods of interpolation

```
>>
>>
>> doc imrotate
```

method — Interpolation method
`'nearest'` (default) | `'bilinear'` | `'bicubic'`

Interpolation method, specified as one of the following values:

Value	Description
`'nearest'`	Nearest-neighbor interpolation
`'bilinear'`	Bilinear interpolation
`'bicubic'`	Bicubic interpolation **Note:** Bicubic interpolation can produce pixel values outside the original range.

Example: `[ = imrotate(I,-1,'bilinear');`

Note: we have used ‘bilinear’ already in imrotate example

Recall our example on image rotation

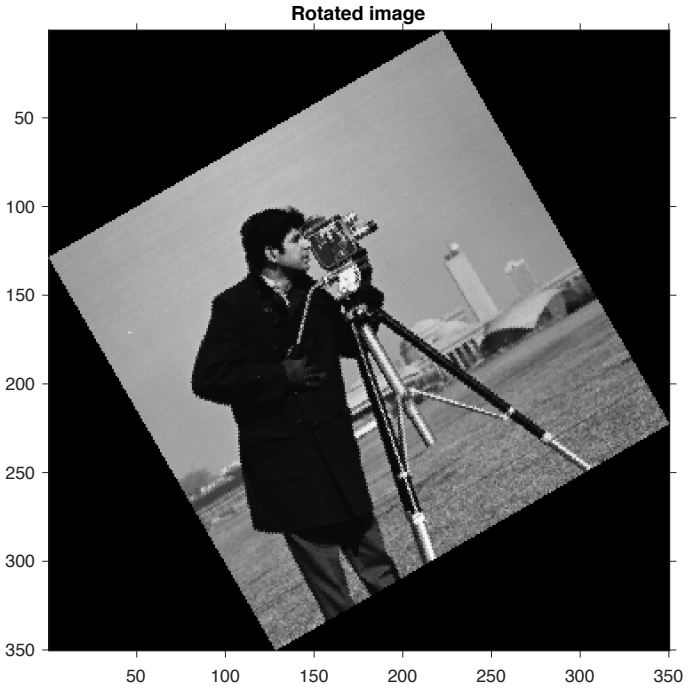
Image rotation

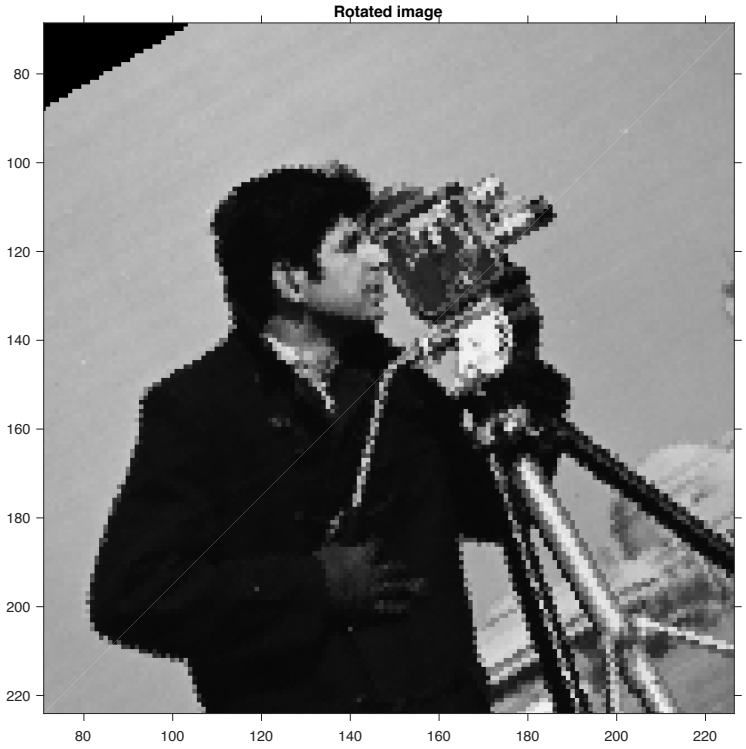
demo_rotation.m

```
% Image rotation
% doc imrotate

% Read image
im=imread('cameraman.tif');

% Show image
figure(1); imshow(im); title('Original image')

% Rotate image
out=imrotate(im, 30);

% Show image
figure(2); imshow(out); title('Rotated image')

```

Remark: ‘nearest’ is used by definition.

5.1. Demo in Matlab: nearest interpolation

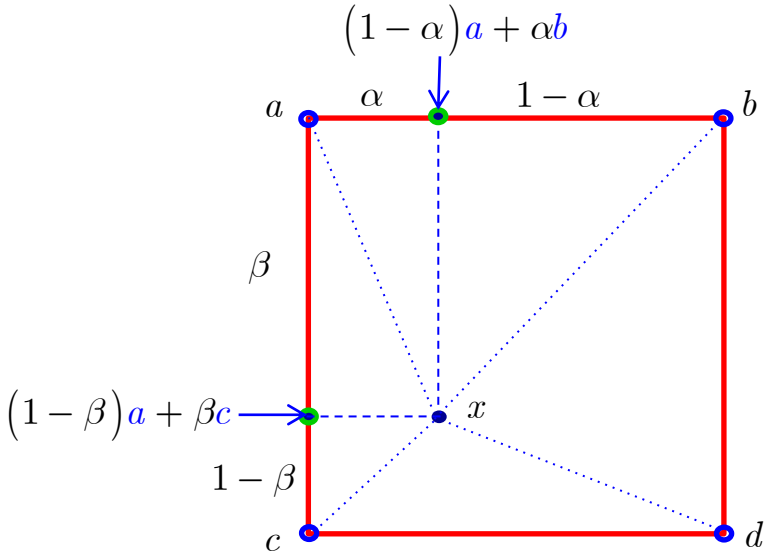




- Quality of rotated image

Reason: the pixel is replaced by the NN one in the original domain.

Bilinear interpolation



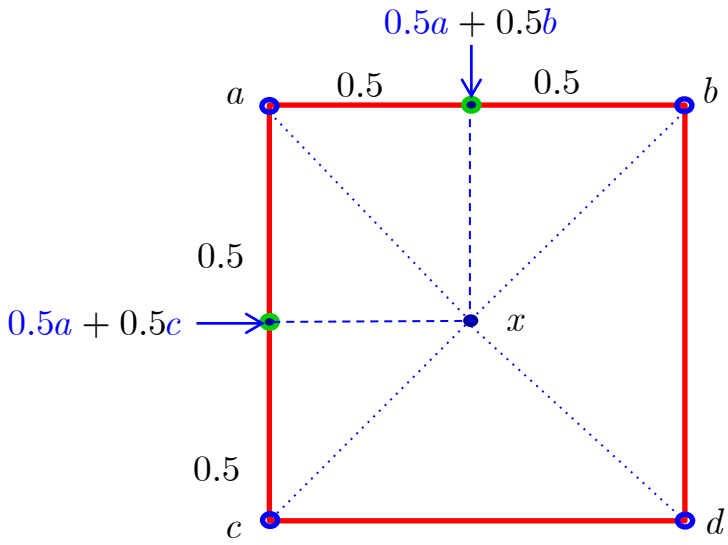
$$x = \varphi(a,b,c,d)$$

$$\begin{aligned} x = {} & (1-\alpha)(1-\beta)a \\ & +\alpha(1-\beta)b \\ & +(1-\alpha)\beta c \\ & +\alpha\beta d \end{aligned}$$

The weights are proportional to the distances to the grid pixel intensities.

Remark: in contrast to the nearest (NN) interpolation, where the intensity x depends only on one NN pixel, in the bilinear interpolation the intensity of x is determined by 4 NN pixels in proportion of distances to them.

Bilinear interpolation



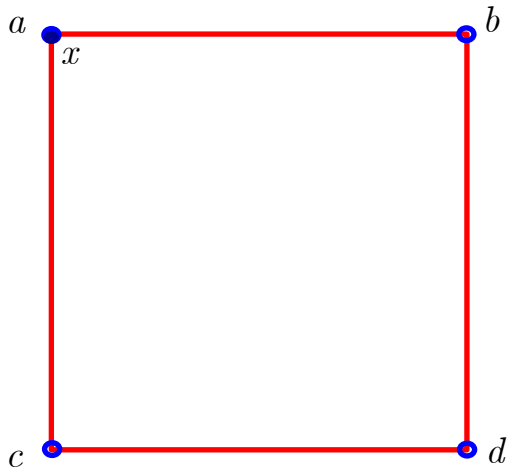
$$x = \varphi(a, b, c, d)$$

$$\begin{aligned} x = {} & 0.5 \cdot 0.5a \\ & +0.5 \cdot 0.5b \\ & +0.5 \cdot 0.5c \\ & +0.5 \cdot 0.5d \end{aligned}$$

$$= \frac{1}{4}\big(a + b + c + d\big)$$

In the center of the image the point x is just proportional to the sum of all of pixel intensities (average value).

Bilinear interpolation



$$x = \varphi(a, b, c, d)$$

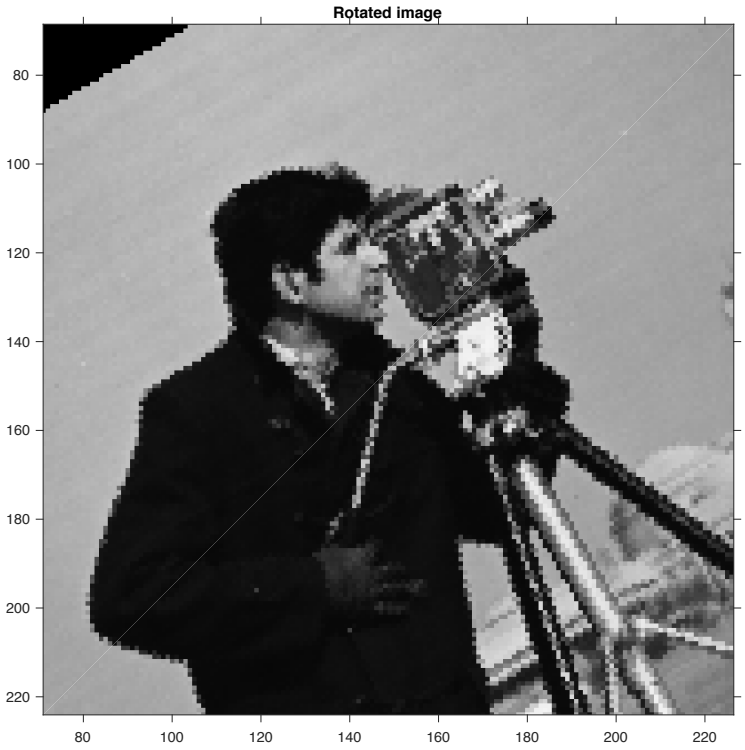
$$x = (1-\alpha)(1-\beta)a + \alpha(1-\beta)b + (1-\alpha)\beta c + \alpha\beta d$$

$$\alpha = 0$$
$$\beta = 0$$

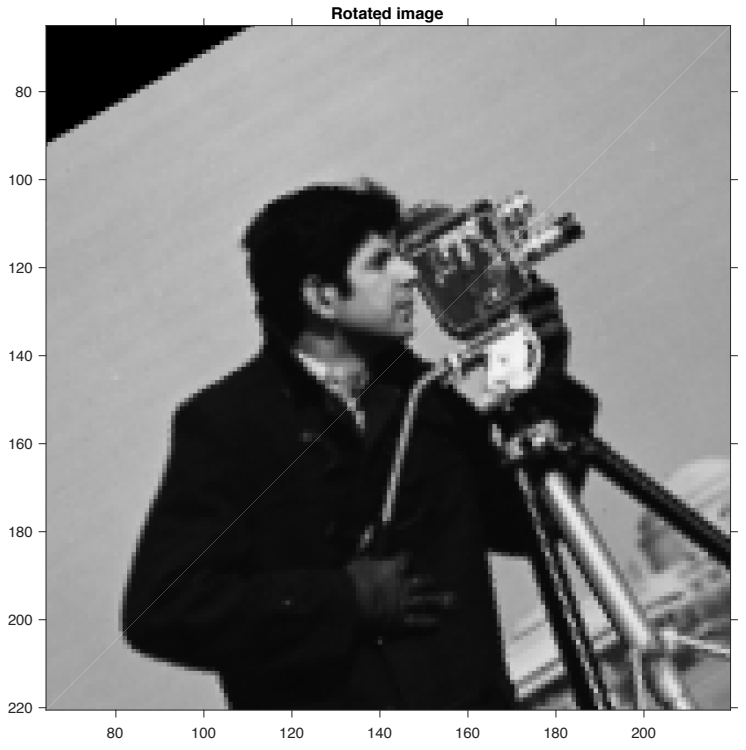
$$x = a$$

If we move closer to the corner, we will have the largest contribution of the corner point.

5.1. Demo in Matlab: nearest vs bilinear



Nearest interpolation



Bilinear interpolation

Bicubic interpolation

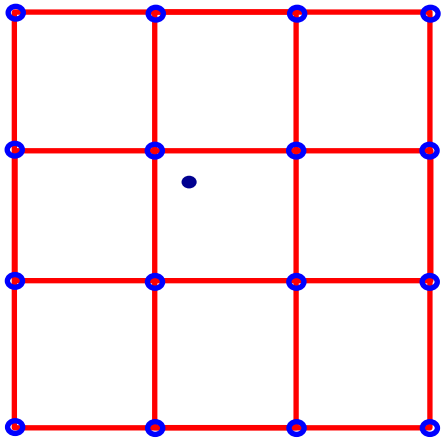
`method` — Interpolation method
`'nearest'` (default) | `'bilinear'` | `'bicubic'`

Interpolation method, specified as one of the following values:

Value	Description
`'nearest'`	Nearest-neighbor interpolation
`'bilinear'`	Bilinear interpolation
`'bicubic'`	Bicubic interpolation **Note:** Bicubic interpolation can produce pixel values outside the original range.

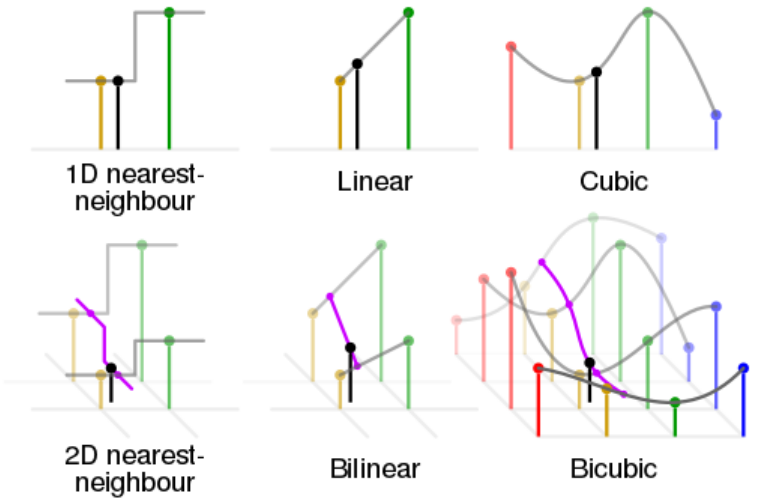
Example: `[ = imrotate(I,-1,'bilinear');`

Bicubic interpolation



Bicubic interpolation uses more points and special weights for the grid pixel intensities.

We will consider it later in more details.



wiki

Other geometric transforms