ELEC 421
Digital Signal and Image Processing

Siamak Najarian, Ph.D., P.Eng.,

Professor of Biomedical Engineering (retired),

Electrical and Computer Engineering Department,

University of British Columbia

Course Roadmap for DIP

Lecture	Title
Lecture 1	Digital Image Modalities and Processing
Lecture 2	The Human Visual System, Perception, and Color
Lecture 3	Image Acquisition and Sensing
Lecture 4	Histograms and Point Operations
Lecture 5	Geometric Operations
Lecture 6	Spatial Filters

Lecture 5:
Geometric Operations

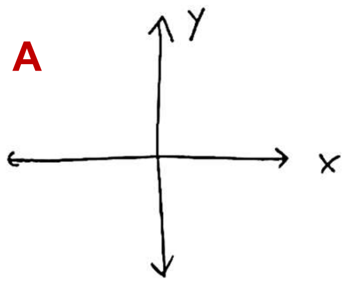
Table of Contents

Geometric operations

GEOMETRIC OPERATIONS.

TODAY WE STAY IN EUCLIDEAN COORDINATES

A



(1) $$J(x,y) = I(T(x,y)) \leftarrow \text{GEOMETRIC OPERATION}$$

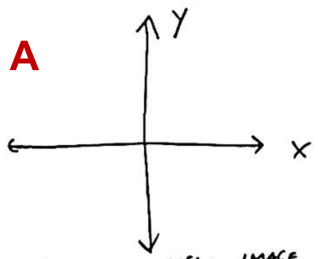
- In the last lecture, we talked about histogram and point operations. That would be a very general way of talking about changing the colors/intensities of an image. The positions of the pixels in the image did not change, and we were just changing how the image looked like.
- In this lecture, we want to focus on **geometric operations**, which are actually going to ***change the size and the shape***. So, in this lecture, we are going to stay in the world of **Euclidean coordinates**. The Euclidean coordinates, A, means we have an **x**-axis and a **y**-axis, just like what we are used to. Here, we mostly are going to talk about the situations where we have a transformation that looks like (1). That is, we have a new image, **J(x,y)**, and we have an old image, **I(x,y)**, and then we have a transformation, **T(x,y)**, so that **J(x,y) = I(T(x,y))**, as shown in (1). This is a geometric operation, why is that? This is asking where pixel **(x,y)** of this new image, **J(x,y)**, comes from. It comes from some transformed version of **(x,y)** in the original image, i.e., **T(x,y)**. This is like saying the pixel of **J(x,y)** will take its color from some pixel of **I(x,y)**. The transformation function **T(x,y)** takes a pixel location **(x,y)** in the original image **I(x,y)** and transforms it to a new location in the output image **J(x,y)**. Here, the colors have not changed, but the location has.

Geometric operations

GEOMETRIC OPERATIONS.

TODAY WE STAY IN EUCLIDEAN COORDINATES

A



COORDS OF NEW IMAGE — COORDS OF OLD IMAGE.

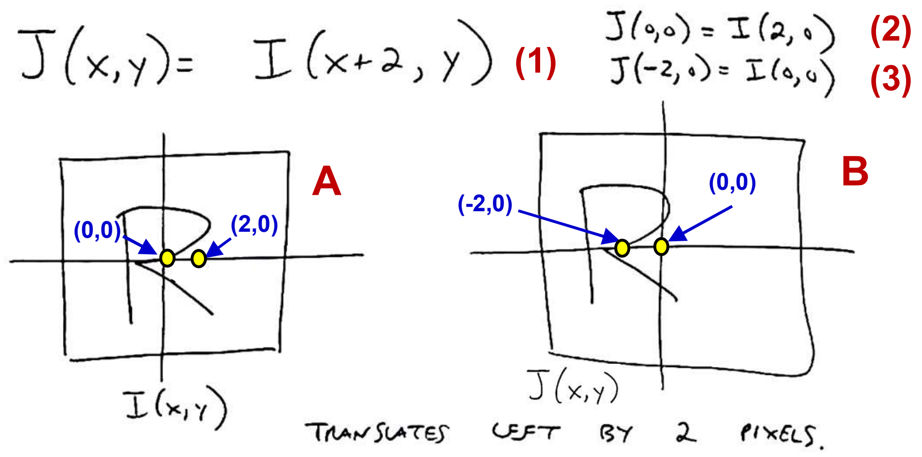
(1) $J(x,y) = I(T(x,y))$ ← GEOMETRIC OPERATION

VS.

(2) $J(x,y) = T(I(x,y))$ ← POINT OPERATION

- Let us contrast that with what we talked about last time. Last time, we talked about saying this **J(x,y)** was the transformation of the color in **(x,y)** or **J(x,y) = T(I(x,y))**, as represented in **(2)**. This is like saying take the color at **(x,y)** and turn it to some other color, but keep it in the same place. Equation **(1)** is the other way around, saying keep the same color, but get that color from a different place. So, these are the two kinds of dual operations of each other.
- **Conclusion:** In **(1)**, **J(x,y)** is like the coordinate of the new image, and **T(x,y)** are like coordinates of the old image.

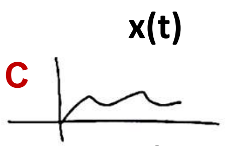
Translation

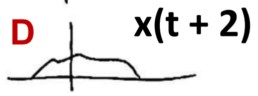


- Let us make this concrete with a bunch of examples. This is actually very similar to the lecture that we did in class on DSP, where we talked about this notion of scaling a signal, shifting a signal, and clipping a signal. We could do all the same things with images. In some sense, in images, it is a little bit more intuitive.
- **Image Translation:** For example, we may have something like $J(x,y) = I(x+2,y)$, **(1)**. **What is this process going to do to the image?** Let us say our old image is $I(x,y)$, **A**. In this lecture, we are going to usually assume, for the most part, that $(0,0)$ is in the middle of the image. We want to draw what happens to our new image. Let us also make an image in **A**, shown by "**R**". What equation **(1)** is saying is that, for example, $J(0,0) = I(2,0)$, **(2)**. That is, J at location $(0,0)$ equals I at location $(2,0)$. This means if we want to know where $(0,0)$ comes from in **B**, it comes from pixel $(2,0)$ in **A**. So, all that means is that the image is going to be translated to the left by **2** pixels. The same process applies to **(3)**, where $J(-2,0) = I(0,0)$.

Translation

(4) $y(t) = x(t+2)$

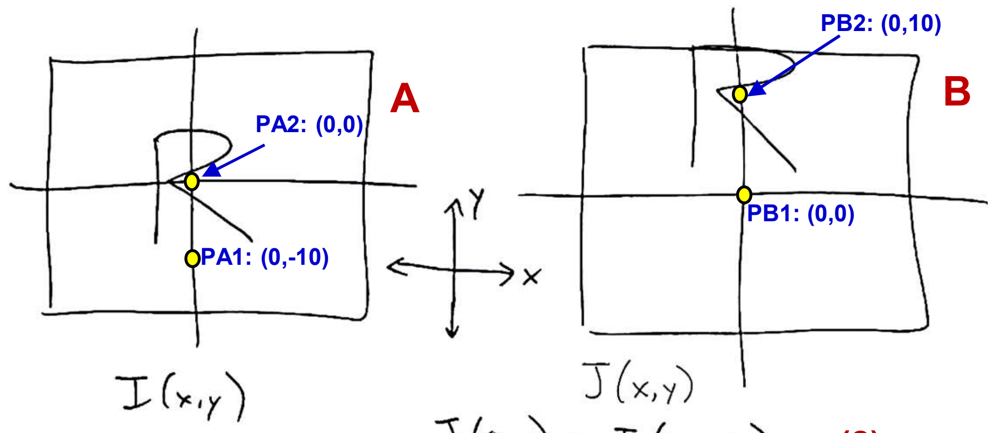
C


D


- This is actually very similar to what we discussed in DSP. The original signal, **x(t)**, is shown in **C**. This is like saying we a one-dimensional image. Let us say we have **y(t) = x(t + 2)**, **(4)**. Equation **(4)** means if we have our old signal as in **C**, then our new signal is going to shift left. This is like a ***negative delay*** by **2** units, shown in **D**.
- If we are not quite sure about which direction the shifting is happening, it is always easiest is to plug in some values of the new image and see where the pixels come from in the old image. This is the easiest way to tell whether we are shifting to the right or to the left.

Translation

$$J(x,y) = I(x, y-10) \quad (1)$$

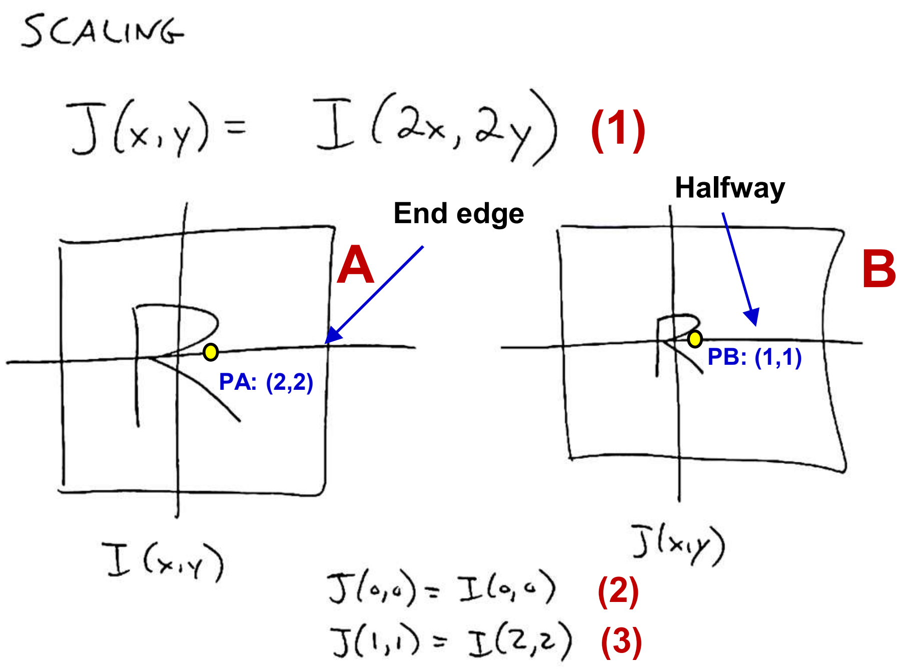


$$J(0,0) = I(0,-10) \quad (2)$$

$$J(0,10) = I(0,0) \quad (3)$$

- In the same way, we could also have **J(x,y) = I(x,y-10)**, **(1)**. Here, **A** is our old image and **B** is our new image. By plugging in some numbers, we can see that this operation is going to ***shift things up*** in the **y**-direction.
- For example, equation (**2**) is saying that **J(0,0)** is coming from **I(0,-10)**. That means point **PB1: (0,0)** is ***inheriting*** its value from point **PA1: (0,-10)**. Also, using **(3)**, we can see that point **PB2: (0,10)** is inheriting its value from point **PA2: (0,0)**.
- **Conclusion:** What we just described is basically the notion of translating the image and all that means is we are taking the original image, picking it up, and moving it over. Of course, we can have translation in both directions if we wanted to.

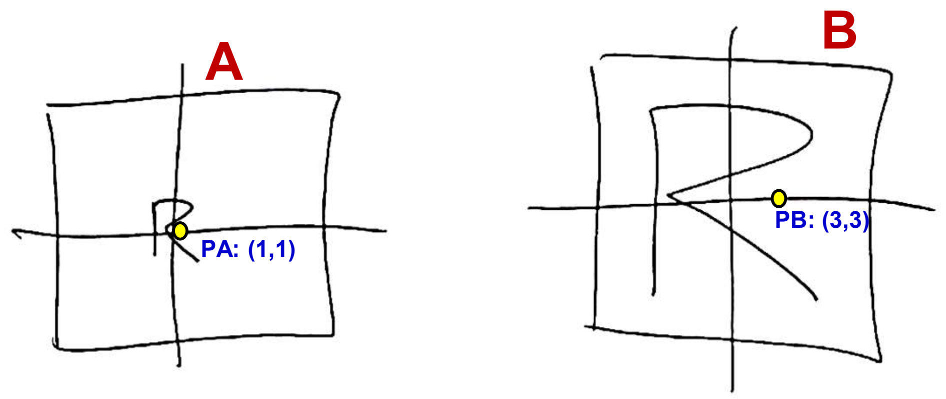
Scaling



- **Image Scaling:** We can also do image scaling. Suppose we have a **10** megapixel image, and we want to scale it down to our web page to make it, say, a **720** pixel wide image. In **(1)**, scaling is like saying our new image is our old image with the factor of **2** that is multiplied by both **x** and **y**. Let us say **A** is our original image and **B** is our new image. Equation **(2)** is saying $\mathbf{J(0,0) = I(0,0)}$, and **(3)** is saying $\mathbf{J(1,1) = I(2,2)}$. Here, $\mathbf{J(1,1)}$ is what used to be $\mathbf{I(2,2)}$. That means that pixel **PB: (1,1)** in **B** is getting its intensity from pixel **PA: (2,2)** in **A**. The overall effect is that the image is going to ***shrink down***. So, we are going to get a smaller image because by the time we get to **halfway** in **B**, we are already at the **end edge** of image **A**.
- This may be a little bit counterintuitive in the sense that we see the **2** multiplying things, and we think this is magnifying things by **2**. But just like in the digital signal processing systems world, if we had the **2** inside the parentheses, ***things would get shorter***. This is similar to playing an audio signal twice as fast, meaning it lasts twice as short or it is half in short.

Scaling

$$J(x,y) = I\left(\tfrac{1}{3}x, \tfrac{1}{3}y\right) \quad (1)$$

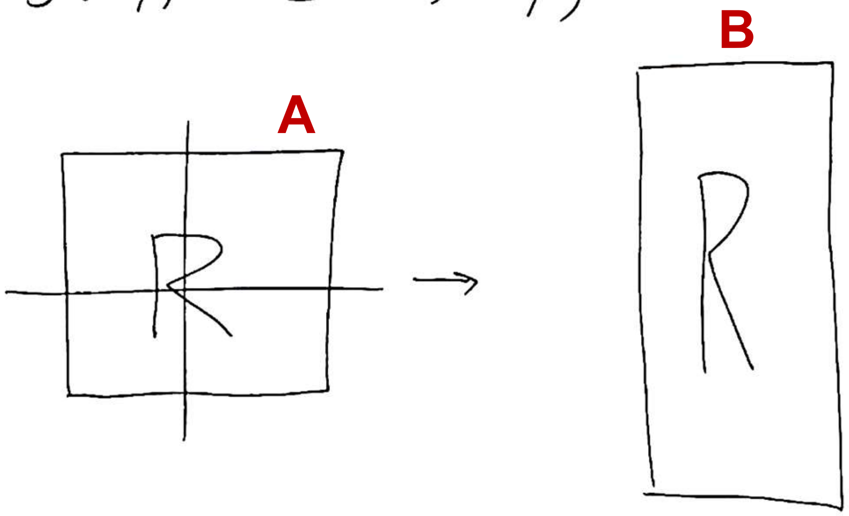


$$J(3,3) = I(1,1) \quad (2)$$

- In the same way, let us say we had a different factor, for example, **1/3** or **I(1/3x, 1/3y)**. That would be like saying that if **A** is our original image, then **B** is going to be our new image. Based on **(1)** and **(2)**, pixel **PB: (3,3)** in **B** gets its value from pixel **PA: (1,1)** in **A**. So, it means that things are going to be ***magnified*** and we are going to have an image that is three times as big.

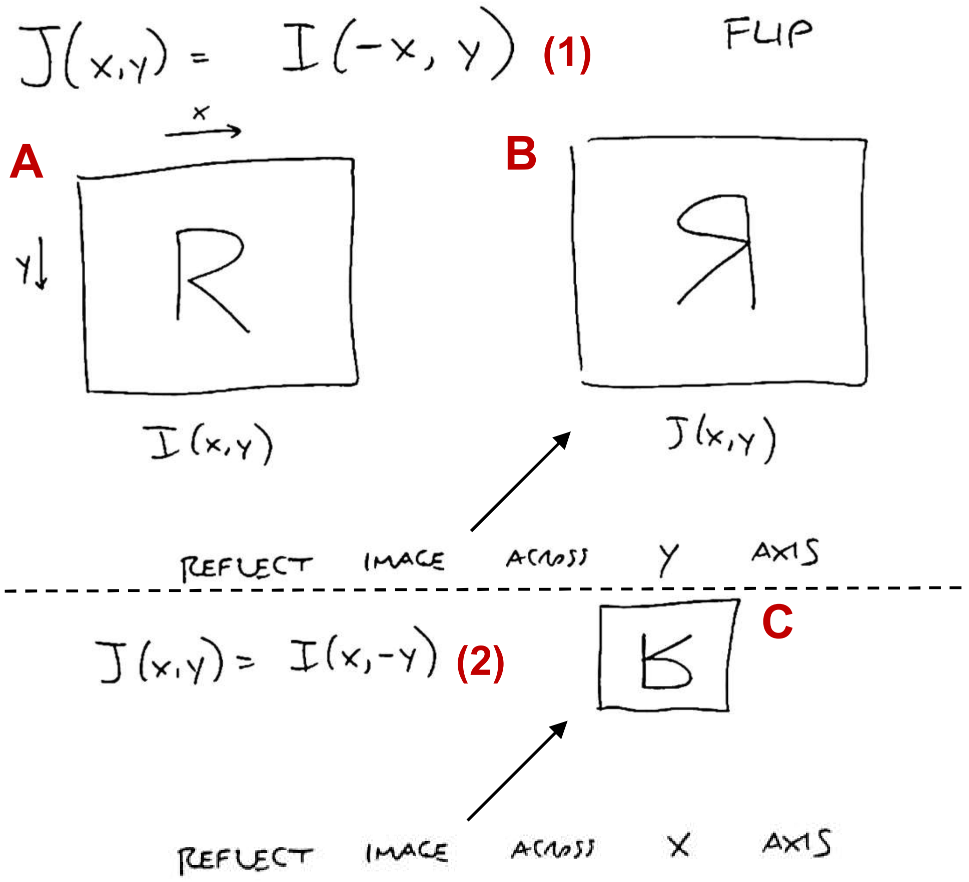
Scaling

$$J(x,y) = I\left(2x, \tfrac{1}{2}y\right) \quad (1)$$



- We can also combine these two operations in different directions. For example, as shown by (1), **J(x,y) = I(2x,1/2y)**. We could see that an image like this is going to be smaller in the **x**-direction and bigger in the **y**-direction, compare A to B.
- So, we have an image that starts up like A, with "**R**" in it, and then it turns into a skinnier version of "**R**" in the **x**-direction but taller in the **y**-direction, as shown in B.
- **Conclusion:** We can do ***non-uniform changes*** in image shape. In Photoshop or similar software packages, these operations are all really automated for us. We just tell the program how much we want to change the image and it gives us back the image without us having to do any real calculations. But, the above is what is going on under the hood.

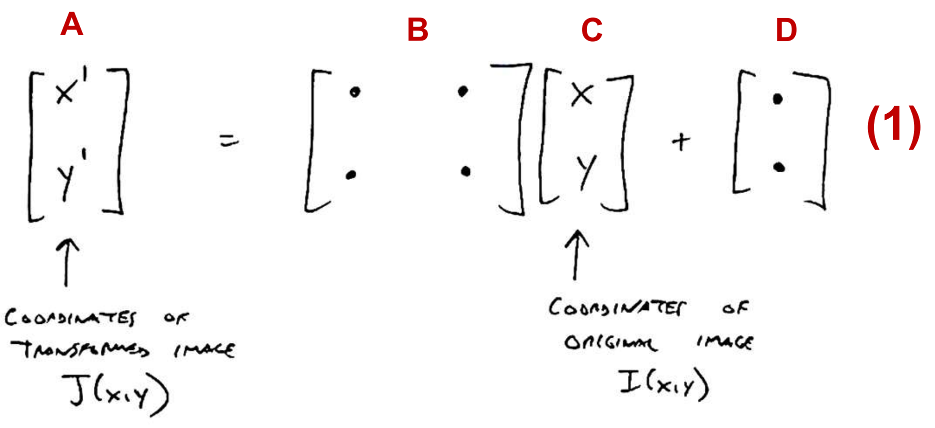
Flipping



- **Image Flipping:** Sometimes, we want to do **mirror image** or **flipping** the image across one of its axis. For example, we have got an interviewer talking with somebody on their right, and we want to flip it around so that it is like they are talking with someone on their left. Flipping of a signal is simply putting a negative sign inside the parenthesis, as shown in **(1)**, $\mathbf{J(x,y) = I(-x,y)}$. That is like saying we flip the **x**-values, but we keep the **y**-values the same. If **A** is our original image, that is like saying we are going to reflect the image across the **y**-axis, which leads to our new image **B**. In summary, equation **(1)** flips the image horizontally (also called **mirroring**). In fact, we are flipping over the **y**-axis (**horizontal flip**).
- In the same way, using **(2)**, $\mathbf{J(x,y) = I(x,-y)}$, we are going to flip over the **x**-axis (**vertical flip**). That is, this equation flips the image **vertically** (also called **upside down**), which leads to **C**.

Linear transformations

IT'S COMMON FOR SCALE + SHIFT + FLIP TO BE COMBINED INTO A 2D LINEAR TRANSFORMATION.


A
B
C
D
x'
y'
=
x
y
+
(1)
COORDINATES OF TRANSFORMED IMAGE
J(x,y)
COORDINATES OF ORIGINAL IMAGE
I(x,y)

WHERE DOES (x,y) IN THE OLD IMAGE GO TO?
(FORWARD MAPPING)

- **2D Linear Transformation:** We can take all these ideas, the scale, the shift, and the flip, and put them into one nice package. It is very common for **scale + shift + flip** to be combined into what is called a **2D linear transformation**, **(1)**. In this transformation, **C** is the coordinates of the original image, **I(x,y)**, and **J(x,y)**, shown by **A**, is the coordinates of the transformed image. Here, we have six numbers, in **B** and **D**, that specify this transformation. So, this is basically like a ***recipe*** that says where **(x,y)** in the old image goes to. This is also known as the ***forward mapping***.
- Forward mapping refers to the process of using the transformation matrix to determine the new location of a pixel in the transformed image based on its original location in the source image.

Linear transformations

TRANSLATION:
$$\begin{array}{ccccccc} \text{a1} & & \text{a2} & \text{a3} & & \text{a4} & \\ \begin{bmatrix}x'\\y'\end{bmatrix} & = & \begin{bmatrix}1&0\\0&1\end{bmatrix} & \begin{bmatrix}x\\y\end{bmatrix} & + & \begin{bmatrix}t_1\\t_2\end{bmatrix} & \text{A}\end{array}$$

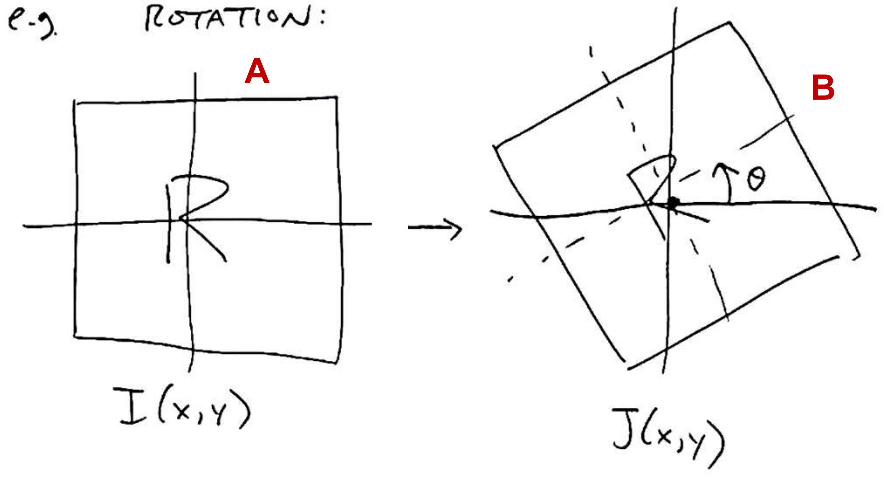
SCALE:
$$\begin{array}{ccccc} \text{b1} & & \text{b2} & \text{b3} & \\ \begin{bmatrix}x'\\y'\end{bmatrix} & = & \begin{bmatrix}\alpha&0\\0&\beta\end{bmatrix} & \begin{bmatrix}x\\y\end{bmatrix} & \text{B}\end{array}$$

FLIP:
$$\begin{array}{ccccc} \text{c1} & & \text{c2} & \text{c3} & \\ \begin{bmatrix}x'\\y'\end{bmatrix} & = & \begin{bmatrix}-1&0\\0&1\end{bmatrix} & \begin{bmatrix}x\\y\end{bmatrix} & \text{C}\end{array}$$

- **Translation:** Using the **matrix form**, we can represent various operations more easily. For example, **translation** is easily represented as **A**, which gives us the new image coordinate. The new image, **a1**, is equal to the identity matrix, **a2** (that does not change anything), times the old image coordinate, **a3**, plus some translation, **a4**.
- **Scaling:** Here, the new image, **b1**, is some factor, **b2**, times the old image coordinate, **b3**.
- **Flipping:** Horizontal flip would be **c1**, which is equal to matrix, **c2**, that says flip the **x**-value and keep the **y**-value the same, and then multiply by **c3**. We can see that **C** is basically a special case of scaling, shown in **B**, where one of the alphas or the betas could be **+1** or **-1**.
- Usually, we write these transformations more compactly. For example in **A**, we specify the matrix, **a2**, and the vector, **a4**.

Rotation

WE CAN DO THINGS TO IMAGES THAT WE CAN'T DO TO 1-D SIGNALS, e.g. ROTATION:



ROTATION BY θ° COUNTERCLOCKWISE. (AROUND ORIGIN)

C

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cdot & \cdot \\ \cdot & \cdot \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \cdot \\ \cdot \end{bmatrix}$$

- **Rotation:** Using the matrix form just described, we can see that there are some things that we can do to images here that we could not really do to one-dimensional signals. One very common scenario is **rotation**. Oftentimes, we take an image and it looks a little bit off and we have to rotate it back to make it look right. For example, we want to take the horizon line and make it horizontal.
- **How does rotation look in this world?** In A, what we want to do is to take our original image and we want to rotate it by some number of degrees around the origin, shown in B. So, we want to know how to rotate image A by **θ** degrees, in this case, counterclockwise. **What is the corresponding matrix, C, that accomplishes the rotation operation?** The easiest thing to do this is to fill in the blanks in matrix C, i.e., fill in the dots for the missing numbers.

Rotation

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \underbrace{\begin{bmatrix} a & b \\ c & d \end{bmatrix}}_{\text{C}} \overbrace{\begin{bmatrix} x \\ y \end{bmatrix}}^{(1)} \quad \text{A}$$

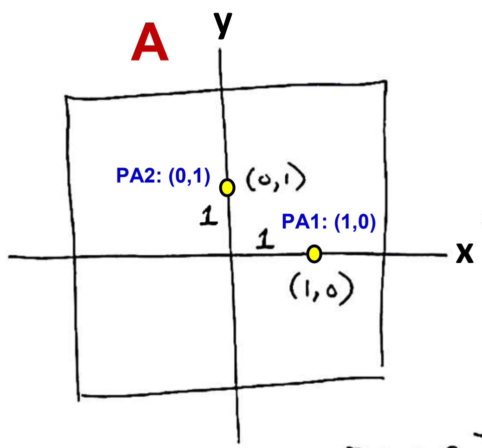
$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}\begin{bmatrix} 1 \\ 0 \end{bmatrix} = \overset{(2)}{\begin{bmatrix} a \\ c \end{bmatrix}}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad \begin{bmatrix} x' \\ y' \end{bmatrix} = \overset{(3)}{\begin{bmatrix} b \\ d \end{bmatrix}}$$

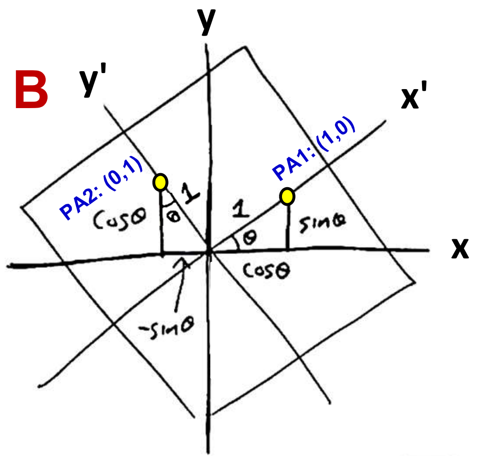
- In **A**, let us suppose we plug in **1** and **0** for **x** and **y** in **(1)**. Here, the output coordinate is going to be **(2)**, i.e., just the first column. The same way, if we plug in **0** and **1** for **x** and **y**, then our output is going to be **(3)**, i.e., just the second column. So, where these two important points go to can help us fill on the entries of matrix **C**.

Rotation

A



B



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

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

CCW

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \underbrace{\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}}_{(3)} \begin{bmatrix} x \\ y \end{bmatrix}$$

C

CW

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \underbrace{\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}}_{(4)} \begin{bmatrix} x \\ y \end{bmatrix}$$

D

- Let us find the **rotation transformation matrix**. Here, **A** is our original image, with the coordinates **PA1: (1,0)** and **PA2: (0,1)** shown on **x**- and **y**-axis. Using trigonometry, we can find the new coordinates in **B**. The right-angle triangle properties lead to the mappings shown in **(1)** and **(2)**. That means what used to be **(1,0)** moves over to **cos** θ and **sin** θ, and what used to be **(0,1)** moves over to **-sin** θ and **cos** θ.
- As shown in **C**, if we put the whole rotation together, for **CCW** rotation, the new coordinate is the special matrix shown as **(3)** that should be multiplied by the old coordinates.
- If we want to rotate the image, going the other way, i.e., for **CW** rotation or clockwise direction, we should use the transpose of **(3)**, shown by **(4)**. The new coordinates can be obtained by **D**.

Similarity transformations

ANY COMBINATION OF SCALE, SHIFT, ROTATE IS CALLED A SIMILARITY TRANSFORMATION

- PRESERVES PARALLEL LINES

IF $\alpha, \beta = \pm 1$, ISOMETRIC TRANSFORMATION (RIGID MOTION)

- PRESERVES SHAPES, ANGLES

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \alpha & 0 \\ 0 & \beta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \quad (1)$$

- **Similarity Transformation:** So far, we have got all the main transformations covered: shifting, scaling, flipping, and rotation. Any combination of these is called a similarity transformation. It does things like preserving parallel lines. Shape and orientation are preserved. Size can change due to scaling (objects may be enlarged or reduced). Examples: Scaling an image, rotating it, translating (shifting), or flipping it while allowing for resizing.
- **Isometric Transformation (Rigid Motion):** In (1), if the $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ in the scaling function are equal to $\pm\mathbf{1}$, then it is a type of isometric transformation. Shape, size, and orientation are preserved. No scaling is allowed (the object's size remains exactly the same). Examples: Rotation, translation (shift), and reflection (flip) without changing the object's size.
- **Conclusion:** A similarity transformation can include scaling, which affects size. An isometric transformation does not include scaling and keeps the object's size unchanged. Thus, an isometric transformation is a special case of a similarity transformation where the scaling factor is exactly ± 1 (no resizing). So, the key difference between a similarity transformation and an isometric transformation lies in whether the size of the object can change.

Shears

WE CAN ALSO "BEND" THE IMAGE . . .

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} \quad (1)$$

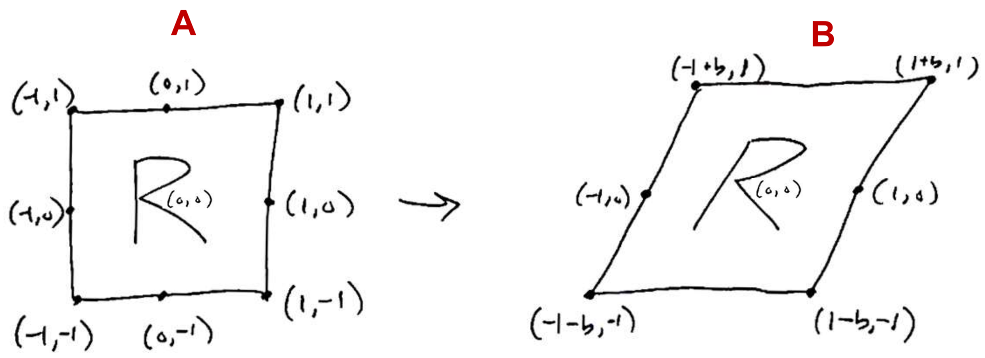
WHAT IF :

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & b \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \quad (2)$$

$$= \begin{bmatrix} x + by \\ y \end{bmatrix} \quad (3)$$

- **Bending the Image (shearing):** Sometimes we want to ***bend the image***. Let us start from the transformation we had before, **(1)**. There are some degrees of freedom in this representation that we still have not used up yet. For example, what if we had transformation like **(2)**. This is not really one of the transformations we have talked about so far. It is not a rotation, it is not an exact scaling, it is not a translation! So, what is happening here? In **(3)**, this is like saying that the new **x**-coordinate, **x + by**, is the old one plus some multiple of **y**, and the **y**-coordinate stays the same. What does that do to our image? Here, **y** is unchanged, and **x** is changing, and changing differently depending on the **y**-value in the image.

Shears



$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x + by \\ y \end{bmatrix} \quad \textbf{(1)}$$

SHEAR

$(0,0) \rightarrow (0,0)$

$(1,0) \rightarrow (1,0)$

$(-1,0) \rightarrow (-1,0)$

$(-1,-1) \rightarrow (-1-b,-1)$

$(0,1) \rightarrow (b,1)$

$(1,1) \rightarrow (1+b,1)$

$(-1,1) \rightarrow (-1+b,1)$

$(0,-1) \rightarrow (-b,-1)$

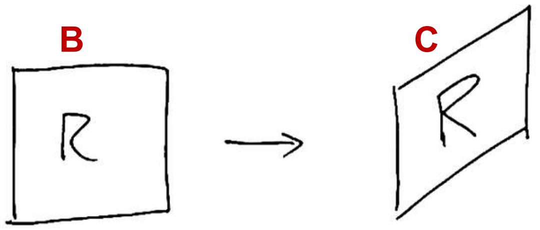
$(1,-1) \rightarrow (1-b,-1)$

- **Horizontal Shear:** Let us apply **(1)** to the coordinates in image **A**, such as (**0,0**), (**1,0**), (**-1,0**), (**0,1**), etc. Some points are mapped to the same coordinates and some are shifted over. The transformed image in shown in **B**. The **y**-values are not going to change and that means the height of the output is going to be the same as the height of the input. But, the **x**-values are going to change according to **x + by**. As evident from **B**, all the points are going to get shifted over by some value **b** (unless their **y**-coordinate is **0**). Let us suppose that **b** is positive. In that case, for example, (**1,1**) goes to (**1+b,1**), shifted to the right, and (**-1,-1**) goes to **(-1-b,-1**), shifted to the left. In all cases, the **y**-value stays the same. The resulting image is called a **horizontal shear**. It is like we are pushing the image over horizontally. Note that everything on the **x**-axis will remain the same, i.e., on the horizonal line passing through (**-1,0**), (**0,0**), and (**1,0**).
- The bigger the **b** is, the more **squeezed** or the more **sheared** that image is going to be. If we were to choose a negative **b**, then the image would be generally pushed over to the left (on the top) instead of generally pushed over to the right.

Shears

A

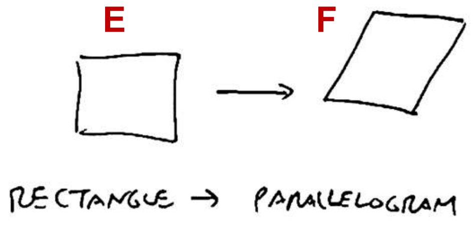
$$\begin{bmatrix}x'\\y'\end{bmatrix}=\begin{bmatrix}1&0\\c&1\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}\qquad(1)$$



Vertical Shear

D

$$\begin{bmatrix}1&b\\c&1\end{bmatrix}$$



Combined Shear

Matrix	Transformation	Description
[1 b; c 1] (both b & c ≠ 0)	Combined Shear	Tilts both vertical and horizontal sides (not aligned with axes).
[1 b; c 1] (b ≠ 0, c = 0)	Horizontal Shear	Tilts vertical sides, horizontal sides remain parallel to **x**-axis.
[1 0; c 1] (b = 0, c ≠ 0)	Vertical Shear	Tilts horizontal sides, vertical sides remain parallel to **y**-axis.

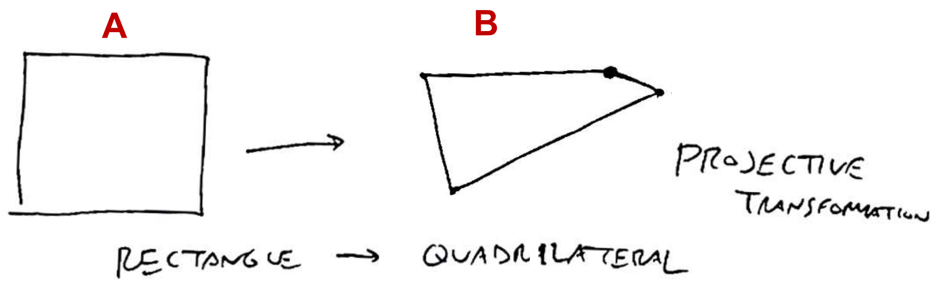
- **Vertical Shear:** The same way we could have a matrix like **A** in **(1)**, where we put an off-diagonal element, **c** in **A**. This geometric transformation will turn **B** into **C**. This is called **vertical shear**.
- **Combined Shear:** If we have both off-diagonal elements **b**, and **c** in matrix **D**, then what used to be a square, **E**, will turn it into a parallelogram, **F**, where the axes are not aligned with the **xy**-axes. It looks as if we have pushed it over in one direction vertically and push it a little bit more in another direction horizontally. That is definitely going to change the image more substantially.
- See the table shown below for a summary of various shear transformations.

Affine transformations

A TRANSFORMATION OF THE FORM

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix} \quad (1)$$

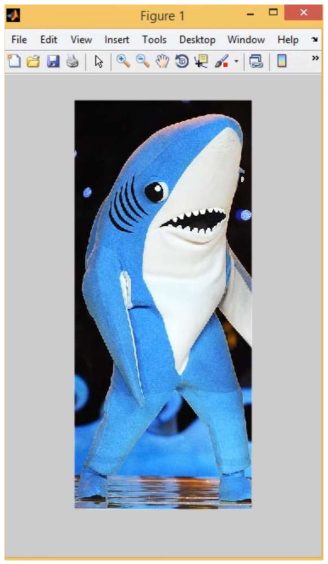
IS CALLED AN AFFINE TRANSFORMATION.



- **Affine Transformation:** If we put everything we discussed so far together, we arrive at **affine transformation**, **(1)**. We could do even more general transformations than affine transformation.
- **Projective Transformation:** We are going to be basically dealing with not much more than the affine transformations, but there are some cases where we need to do what is called a **perspective transformation** or a **projective transformation**. Here, we have to distort the image even more, as shown in **A** and **B**. In **A**, with an affine transformation, we cannot do anything more than turn the rectangle into a parallelogram. But, if we want to turn the rectangle into some other weird generic quadrilateral, i.e., some general four-sided object, we can use a projective transformation.
- **Conclusion:** Affine transformations are linear mappings that preserve lines and parallelism but not necessarily lengths or angles. Projective transformations are more general than affine transformations. They map points, lines, and even planes to other points, lines, and planes. They can distort shapes in ways that affine transformations cannot, such as introducing vanishing points or making parallel lines appear to converge.

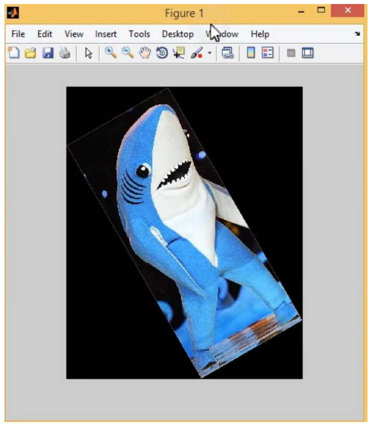
MATLAB examples

```
>> im = imread('leftshark.png');
>> imshow(im)
```



A

```
>> out = imrotate(im, 30);
>> imshow(out)
```
(1)



B

- **How to do image rotating in MATLAB?** Let us take a look at how we do these things in MATLAB. We are going to use **'leftshark.png'** image, **A**. MATLAB has a bunch of image processing tools to deal with ***geometric operations***. Let us start by rotating the image. The rotating command is shown in **(1)**, which is **imrotate (im, 30)** or another angle instead of **30** degrees. The output image, **B**, is going to be our input image rotated by **30** degrees counterclockwise.
- As expected, we can see that MATLAB has chosen to fill in the rest of the pixels with black. This is because when we rotate the image, it is becoming bigger, i.e., instead of having the original height and width, it is getting taller and wider. And, we have to fill in those pixels with something. So, MATLAB has filled them in black.

MATLAB examples

The imwarp function in MATLAB:
- The **imwarp** function in MATLAB is a versatile tool used for applying geometric transformations to spatial referenced images. Here is a breakdown of its functionalities:

Purpose:
- imwarp takes a spatially referenced image (represented by image data and a corresponding spatial referencing object) and applies a specified geometric transformation to it.
- The output is another spatially referenced image, where the pixel values have been warped according to the transformation.

Transformations Supported:
- imwarp supports various geometric transformations, including:
    - ➢ **Affine transformations:** These include scaling, rotation, shearing, and translation.
    - ➢ **Projective transformations:** In specific cases, with additional information, projective transformations can be achieved.

Key Inputs:
- **Image Data (A):** This is the actual image data you want to transform, typically a 2D array representing pixel intensities.
- **Spatial Referencing Object (RA):** This object defines the spatial information associated with the image data. It specifies how the pixel coordinates relate to real-world units (e.g., meters, degrees).
- **Transformation (tform):** This represents the geometric transformation you want to apply. It can be defined in various ways, such as using an affine transformation matrix or a more complex function.

MATLAB examples

Output:

- imwarp returns a new spatially referenced image (**B** and **RB**) where the image data (**B**) has been warped according to the specified transformation. The corresponding spatial referencing object (**RB**) reflects the transformed coordinate system.

Applications:

- imwarp is used in various image processing tasks, including:
  - **Image registration:** Aligning two or more images from different viewpoints.
  - **Image correction:** Correcting perspective distortions or image warping caused by camera lens imperfections.
  - **Object tracking:** Tracking the movement of objects in an image sequence by applying appropriate transformations to follow their positions.
  - **Image manipulation:** Creating artistic effects by applying creative geometric distortions.

Additional Notes:

- imwarp works seamlessly with the Image Processing Toolbox in MATLAB.
- It offers flexibility in defining and applying various geometric transformations to your images.
- Understanding spatial referencing objects is crucial for ensuring accurate interpretation of the transformed image data.

By effectively using imwarp, we can achieve a wide range of image manipulations and geometric corrections within our MATLAB workflows.

MATLAB examples

- The complete command for imwarp in MATLAB will depend on the specific transformation we want to apply and the desired output. However, here is a general template that outlines the basic structure:

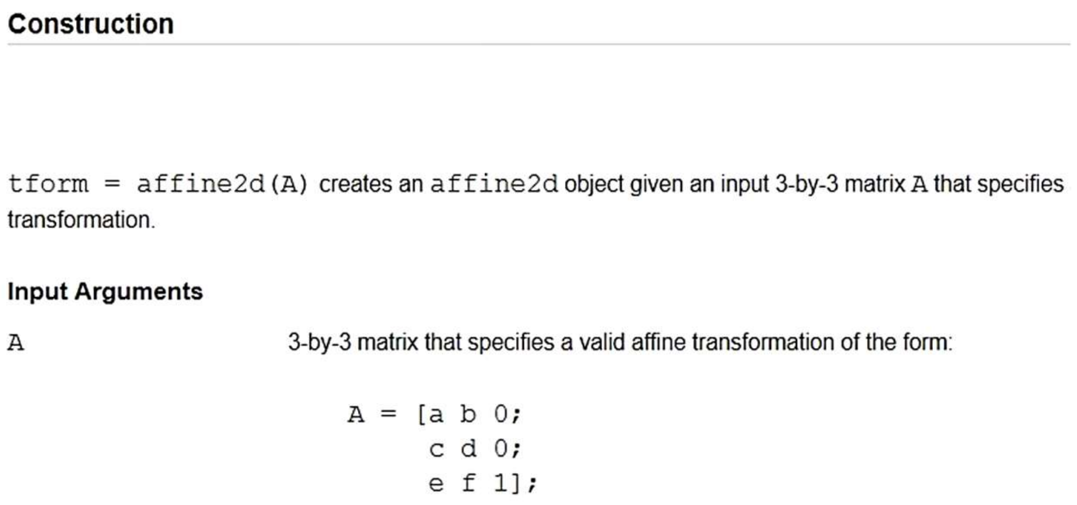
B = imwarp(A, RA, tform);

Explanation of Arguments:

- **B:** This variable will store the output image data after applying the transformation.
- **A:** This is the original image data you want to transform (typically a 2D array).
- **RA:** This is the spatial referencing object associated with the image **A**. It defines the real-world units corresponding to pixel coordinates.
- **tform:** This represents the geometric transformation you want to apply. It can be defined in various ways:
    - **Affine transformation matrix:** A **3x3** matrix representing scaling, rotation, shear, and translation.
    - **Geometric transformation objects:** MATLAB provides built-in objects like **affine2d**, **projective2d** for defining transformations.
    - **Functions:** In specific cases, we can use custom functions to define the transformation logic.

MATLAB examples

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix} \quad (1)$$



Construction

`tform = affine2d(A)` creates an `affine2d` object given an input 3-by-3 matrix `A` that specifies 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];
```

$$\longrightarrow \quad A = \begin{pmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{pmatrix}$$

- **How to do affine transformation in MATLAB?** If we want to make an affine transformation, **(1)**, there is a command called **affine2d**. Here, we are basically specifying the **a, b, c, d, e, f** that we want to use. The **a, b, c, d** is the part that encapsulates the rotation and scaling, and the **e, f** is what encapsulates the translation.

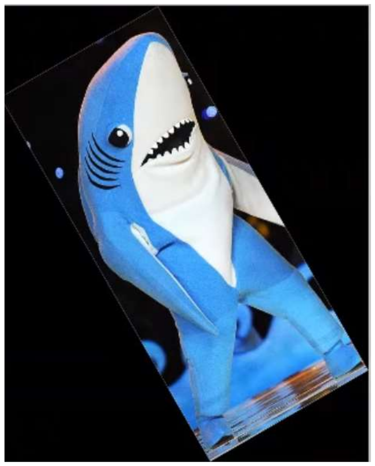
MATLAB examples

```
>> cos(30)
                    (1)
ans =

      0.1543

>> cosd(30)
                    (2)
ans =

      0.8660
```

```
>> T = [cosd(30) -sind(30) 0; sind(30) cosd(30) 0; 0 0 1]
                                                     (3)
T =

    0.8660   -0.5000         0
    0.5000    0.8660         0                (4)
         0         0    1.0000
```

$$T = A = \begin{pmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{pmatrix}$$

```
>> Tf = affine2d(T)    (5)

Tf =

  affine2d with properties:

                 T: [3x3 double]
    Dimensionality: 2

>> out = imwarp(im, Tf); (6)
>> imshow(out)
```


A

- **How to get the tilted image, this time, using affine2d and imwarp command?** Before we do this part, note that **cos (30)** in MATLAB will give us **0.1543**, **(1)**, which is in radians. But, if we want it in degrees, we should use **cosd (30)**, **(2)**, which will give us **0.8660**.
- Let us first define our rotation transformation matrix **T** as **(3)**. Matrix **T** is shown by **(4)**. In **(5)**, we create an **affine object** using matrix **T** and store it in **Tf**. Next, we apply our affine transformation, **Tf**, to the image by the imwarp command, **(6)**. As expected, we get the same image, **A**, that we had before using **imrotate (im, 30)**.

MATLAB examples

Matrix	Transformation	Description
[1 b; c 1] (both b & c ≠ 0)	Combined Shear	Tilts both vertical and horizontal sides (not aligned with axes).
[1 b; c 1] (b ≠ 0, c = 0)	Horizontal Shear	Tilts vertical sides, horizontal sides remain parallel.
[1 0; c 1] (b = 0, c ≠ 0)	Vertical Shear	Tilts horizontal sides, vertical sides remain parallel.

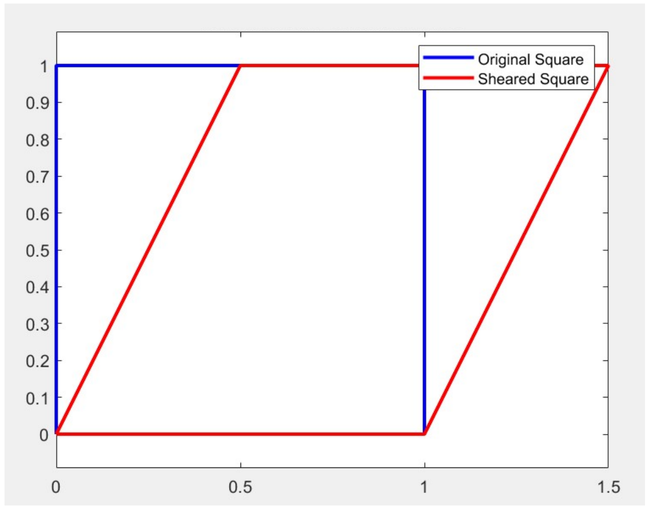
- Let us apply various affine transformations to a square image in MATLAB.

$$A = \begin{pmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{pmatrix}$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \longrightarrow A = \begin{pmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

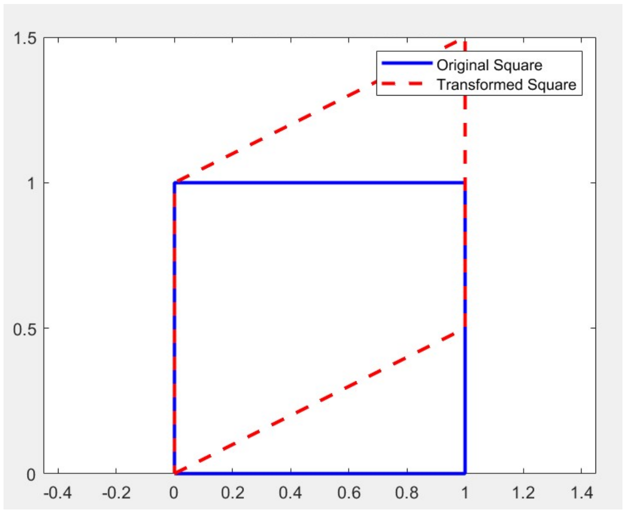
```
>> square = [0 0; 1 0; 1 1; 0 1; 0 0];
shear_factor = 0.5;
T = [1 shear_factor 0; 0 1 0; 0 0 1];
tform = affine2d(T);
```

Horizontal Shear



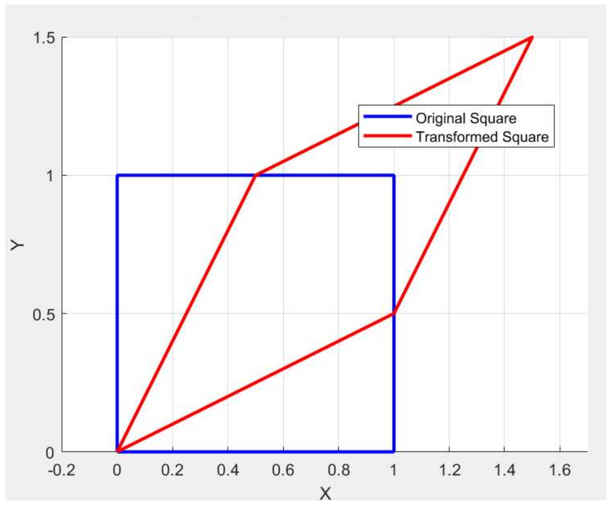
```
>> square = [0 0; 1 0; 1 1; 0 1; 0 0];
shear_factor = 0.5;
T = [1 0 0; shear_factor 1 0; 0 0 1];
tform = affine2d(T);
```

Vertical Shear

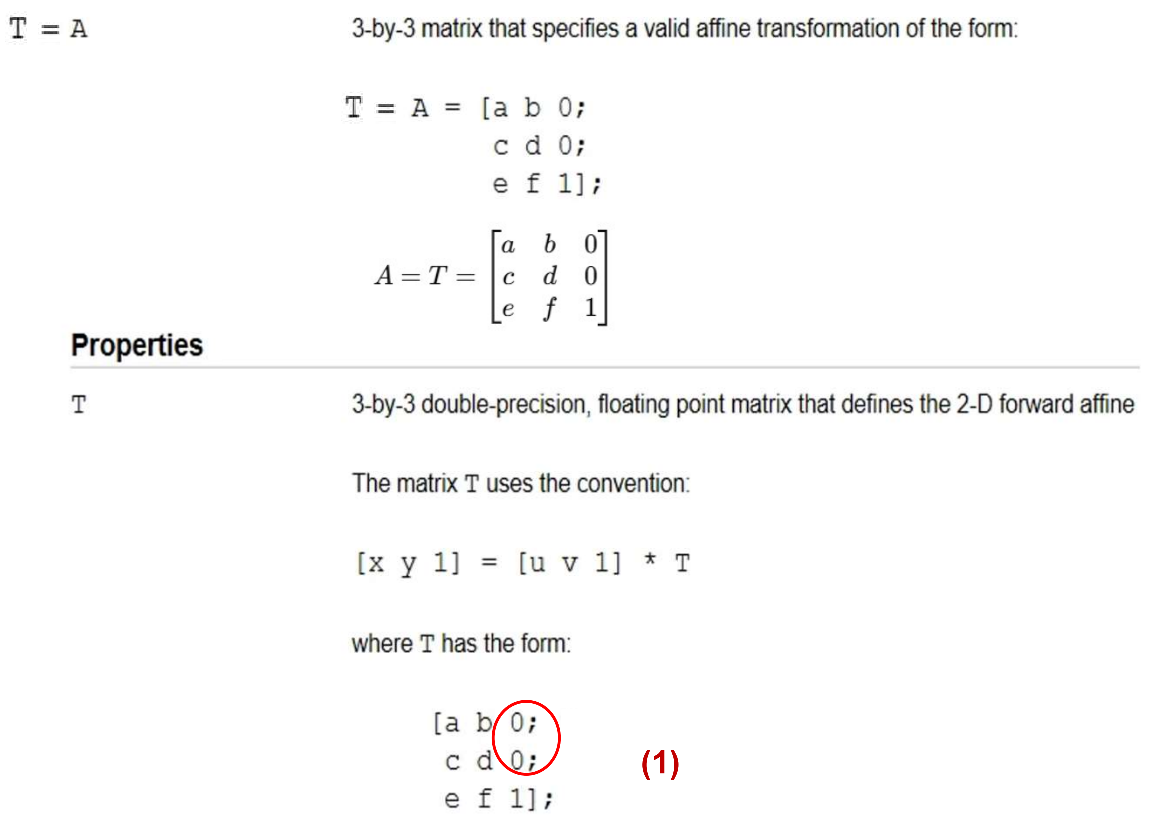


```
>> square = [0 0; 1 0; 1 1; 0 1; 0 0];
shear_factor = 0.5;
T = [1 0.5 0; 0.5 1 0; 0 0 1];
tform = affine2d(T);
```

Combined shear: parallelogram



Projective transformations



```
T = A
```
3-by-3 matrix that specifies a valid affine transformation of the form:

```
T = A = [a b 0;
         c d 0;
         e f 1];
```

$$A = T = \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}$$

Properties

`T` 3-by-3 double-precision, floating point matrix that defines the 2-D forward affine

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];
```

(1)

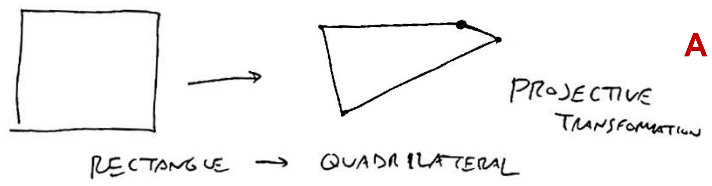
- In the affine transformation in **(1)**, we see two **0**'s in the matrix. They are related to what is called the **projective transformation**.
- The equation **[x y 1] = [u v 1]*T** represents the core principle of applying an affine transformation defined by the matrix **T** (or **A**) to a point with coordinates **(u, v)**.
- **[x y 1]:** This represents a 3D vector where **x** and **y** are the coordinates of the point after applying the transformation.
- **[u v 1]:** This represents a 3D vector where **u** and **v** are the original coordinates of the point before the transformation.
- **T:** This is the **3x3** transformation matrix that encodes the specific affine operation to be applied.

Projective transformations

A TRANSFORMATION OF THE FORM

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix}$$

IS CALLED AN AFFINE TRANSFORMATION.



A

PROJECTIVE TRANSFORMATION

(2) $$\tilde{z}' = c_1 x + c_2 y + 1$$

(1) $$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \dfrac{a_{11}x + a_{12}y + b_1}{c_1 x + c_2 y + 1} \\ \dfrac{a_{21}x + a_{22}y + b_2}{c_1 x + c_2 y + 1} \end{bmatrix}$$

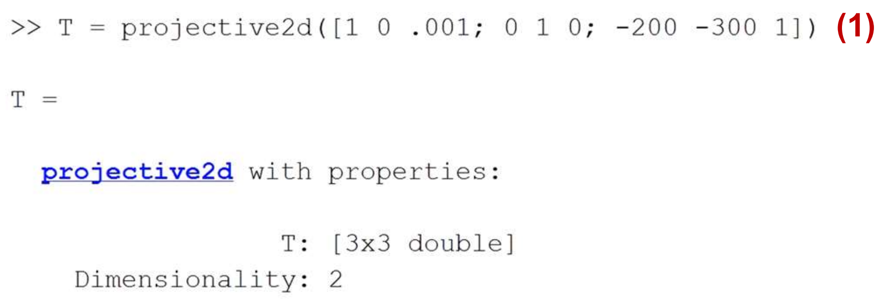
(3) $$x' = \frac{\tilde{x}'}{\tilde{z}'}$$

(4) $$y' = \frac{\tilde{y}'}{\tilde{z}'}$$

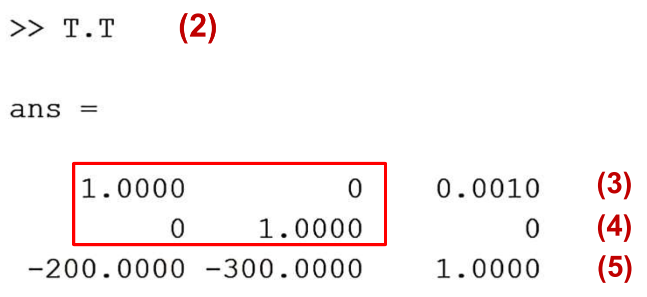
(5) $$\begin{bmatrix} \tilde{x}' \\ \tilde{y}' \\ \tilde{z}' \end{bmatrix} = \underbrace{\begin{bmatrix} a_{11} & a_{12} & b_1 \\ a_{21} & a_{22} & b_2 \\ c_1 & c_2 & 1 \end{bmatrix}}_{B} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- **How could we do a projective transformation, A?** **Projective transformation** is more general than affine transformation and will look a little bit more complicated. In **(1)**, the numerator looks still kind of affine and it is the denominator that makes things more complicated. Equation **(1)** can be simply shown by **(3)** and **(4)**, where the denominator is just **$c_1x + c_2y + 1$**, i.e., **(2)**.
- Equation **(1)** can be collected into **8** numbers, as shown in **(5)**, not including "**1**" in matrix **B**, since it is already known. Equation **(5)** shows how projective transformation works. MATLAB basically asks for matrix **B** when performing perspective transformation. It turns out that **c_1** and **c_2** in **B** have some effect on image translation.
- There is a command called **projective2d** or a better new version of it **projtform2d** in MATLAB for projective or perspective 2D.

Projective transformations



```
>> T = projective2d([1 0 .001; 0 1 0; -200 -300 1]) (1)

T =

  projective2d with properties:

                 T: [3x3 double]
    Dimensionality: 2
```



```
>> T.T     (2)

ans =

    1.0000         0    0.0010     (3)
         0    1.0000         0     (4)
 -200.0000 -300.0000    1.0000     (5)
```

$$\begin{bmatrix} a_{11} & a_{12} & b_1 \\ a_{21} & a_{22} & b_2 \\ c_1 & c_2 & 1 \end{bmatrix} \quad \mathbf{A}$$

- Line **(1)** shows a projective2d transformation. To see matrix **T** better, we can use **T.T**, **(2)**. Now, line **(5)** is telling us we have translation, i.e., we are moving around the image a little bit. The upper **2** by **2** part, shown in **red** box, is telling us we do not really have any rotation. But, **0.0010** in line **(3)**, ($\mathbf{b_1}$ **= 0.0010**, shown in **A**) can ***skew the image***. The value of $\mathbf{b_1}$ affects the shear along the **x**-axis. When $\mathbf{b_1}$ **= 0**, there is no shear along the **x**-axis (no distortion). If $\mathbf{b_1}$ is positive, it introduces shear to the right (stretching horizontally). If $\mathbf{b_1}$ is negative, it introduces shear to the left (compressing horizontally). This element introduces ***projective distortion***, which is also sometimes referred to as ***keystoning***. Here, the image will appear as if it is viewed from an angle, creating a perspective effect similar to what you might see when looking at a distant object.
- The value $\mathbf{c_1}$ **= -200** introduces a significant translation along the **x**-axis (leftward shift). The value $\mathbf{c_2}$ **= -300** introduces a significant translation along the **y**-axis (downward shift).

Creating the output image

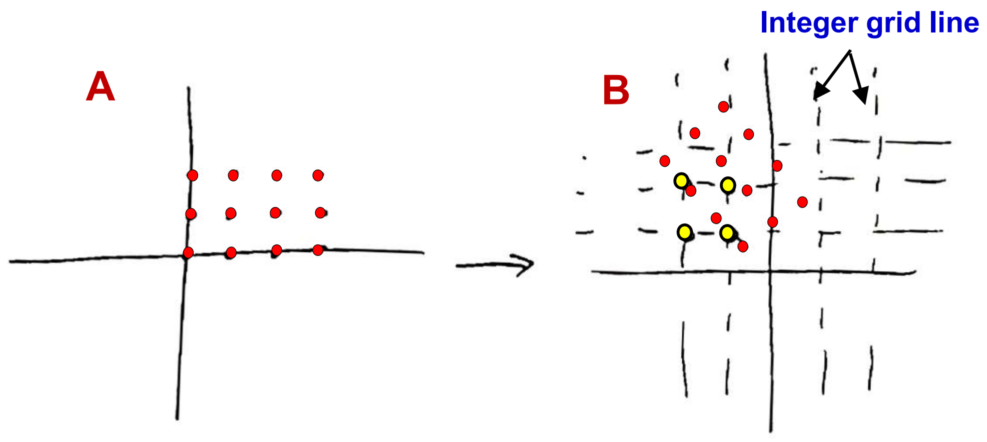
HOW TO ACTUALLY CREATE THE OUTPUT IMAGE?

A
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1.2 & 1 \\ 1 & 0.8 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 3.2 \\ -1.6 \end{bmatrix}$$

B
$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} \longrightarrow \begin{bmatrix} 5.4 \\ 0.2 \end{bmatrix}$$
C

- When we take an image and we change its shape, bending it, making it bigger in some directions, and so on, we clearly have to figure out new pixel colors that were not in the original image. It is not like we can just sample colors from the original image. We have to do some hallucination of new colors. So, how does that process work? **How to actually create the output image?**
- As an example, suppose we have the affine transformation, **A**. In real life, the numbers that we have in these matrices are not just integers. Let us say we want to know where does the point **(1,1)** go, i.e., **B**? We plug in our numbers in **A** and we get **C**. So, that is a problem because we have all integer coordinates in our original image before and now we are transforming it into all these non-integer coordinates.

Creating the output image



HOW TO GET IMAGE COLORS / INTENSITIES ON THE NEW GRID?

IT'S MORE CONVENTIONAL TO USE BACKWARDS MAPPING...

- **Backwards Mapping:** What we used to have was a nice grid in **A** where our pixel values fell on nice integer locations. **B** is after transformation. Let us suppose that the dashed lines are the **integer grid lines**. Our new image may happen at red points. That is, they go to a different bunch of weird places. The integer points are at the intersection of dashed lines, shown by yellow points
- How are we going to get the red points values? Or, **how to get image colors or intensities on the new grid?** We could argue that one thing we can do is just find the nearest yellow dot and take its color. This would be a crude solution that could kind of work. But, on the other hand, if the red dots are really far away from the yellow dots, then we are going to have a big error in what that image looks like.
- What we could do is to go the other way. So, it is more conventional to use ***backwards mapping*** instead of ***forwards mapping***.

Creating the output image

More on The Issue with Forwards Mapping:

- In **forwards mapping**, we take each point from the input image and map it directly to its new location in the output image. Here is why this can cause problems:

1. Gaps (Missing Pixels):

- ➢ When transforming the image, the new locations of the input points often do not line up exactly with the integer grid points (yellow points) of the output image.
- ➢ This means some integer grid points in the output image will be left **empty** (no pixel from the input image lands there). These gaps appear because we are "pushing" points from the input image to the output image, but the target locations might not align perfectly with the pixel grid.

2. Overlaps (Duplicate Pixels):

- ➢ Multiple input pixels might map to the **same** location in the output image, causing overlap.
- ➢ When this happens, it is unclear which input pixel value should be used, leading to data loss or blending issues.

Summary:

- **Forwards Mapping**: Pushes input points to the output, leading to gaps (missing data) and overlaps (duplicate data).
- **Backwards Mapping**: Pulls values from the input for each output point, ensuring that every output pixel is filled correctly without gaps or overlaps.
  - ➢ In practice, backwards mapping might involve more computation (because we need to trace back for each output pixel), but it provides a more accurate and complete result without missing or duplicated pixels. This is why it is the preferred approach in image transformations, despite being conceptually more complex.

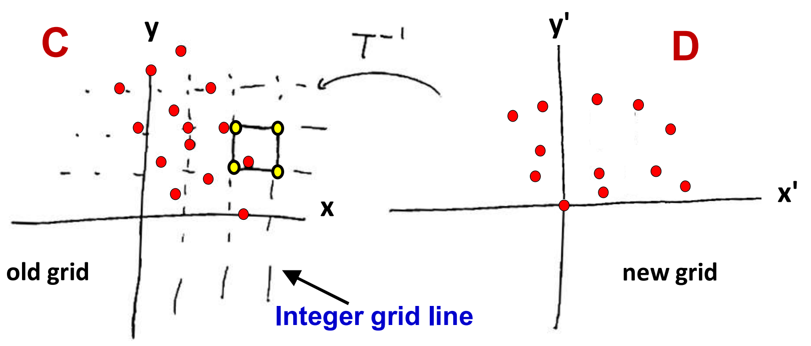
Creating the output image

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \underset{2\times 2}{A} \begin{bmatrix} x \\ y \end{bmatrix} + \underset{2\times 1}{b} \quad (1)$$

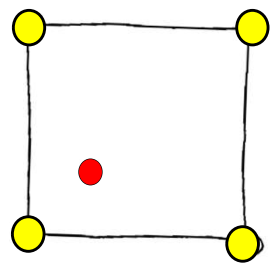
$$\begin{bmatrix} x \\ y \end{bmatrix} = A^{-1}\left(\begin{bmatrix} x' \\ y' \end{bmatrix} - b \right) \quad (2)$$

$$= A^{-1} \begin{bmatrix} x' \\ y' \end{bmatrix} - A^{-1} b \quad (3)$$

INVERSE TRANSFORMATION



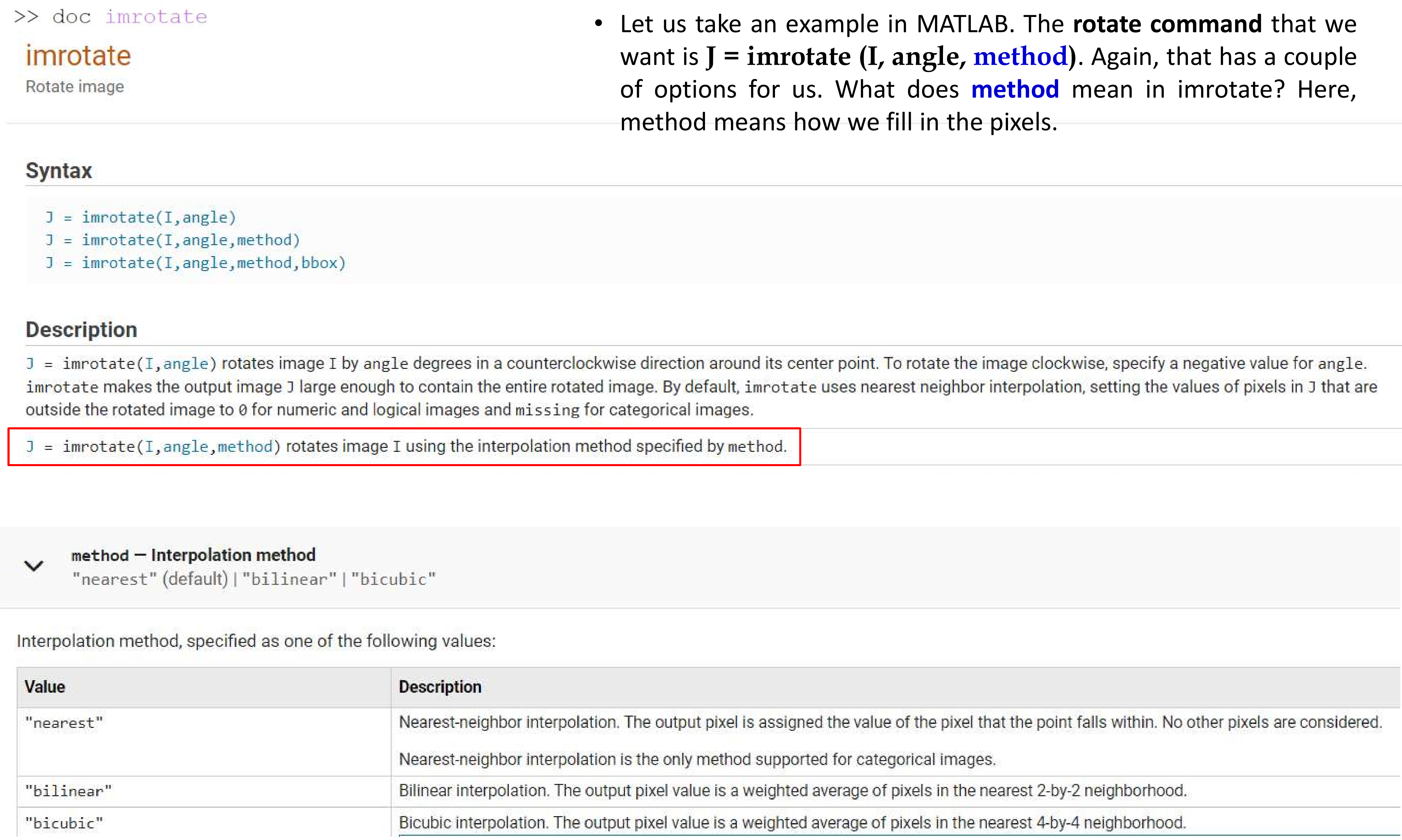
E



- **The Math Behind Backwards or Inverse Transformation:** If we have an affine transformation, we should be able to invert it. Equation **(1)** shows our original forwards mapping, pushing pixels from one image to another. Matrix **A** is a **2** by **2** matrix and matrix **b** is a **2** by **1** vector. We should be able to undo this process. If we wanted to get **[x y]** by itself, **(2)**, what we could do is multiply **(1)** by the inverse of **A**. This also turns into some new affine transformation, **(3)**. This is called the **inverse transformation**. This shows us how to do backwards transformation mathematically.
- **How does the above process help?** Here, we go from **[x' y']**, the new grid, to **[x y]**, the old grid, and we use a transformation represented by **T**$^{-1}$. That is, we go from **D** to **C**. We already know the color or intensity of the yellow dots/points in **C**. What we want is the color or intensity of the red dots/points in **C**. If we blow up the square in **C**, our goal is to analyze what happens inside this square.
- In **E**, the red dot is the point that we want to figure out the color of. It is surrounded by a bunch of yellow points, where we know the color based on the original image.

Creating the output image

- Let us take an example in MATLAB. The **rotate command** that we want is **J = imrotate (I, angle, method)**. Again, that has a couple of options for us. What does **method** mean in imrotate? Here, method means how we fill in the pixels.



```
>> doc imrotate
```

imrotate

Rotate image

Syntax

```
J = imrotate(I,angle)
J = imrotate(I,angle,method)
J = imrotate(I,angle,method,bbox)
```

Description

`J = imrotate(I,angle)` rotates image `I` 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 `J` large enough to contain the entire rotated image. By default, `imrotate` uses nearest neighbor interpolation, setting the values of pixels in `J` that are outside the rotated image to `0` for numeric and logical images and `missing` for categorical images.

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

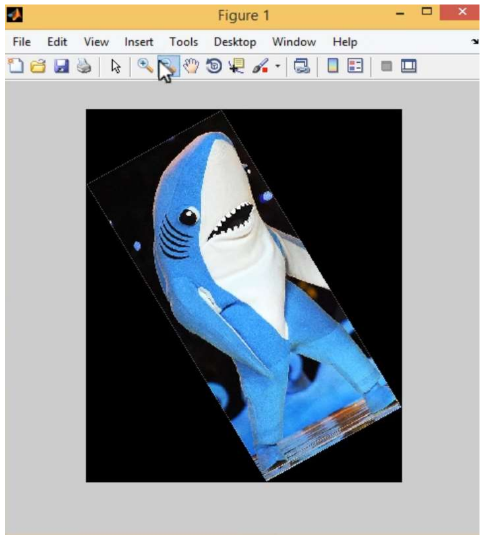
`method` — Interpolation method
`"nearest"` (default) | `"bilinear"` | `"bicubic"`

Interpolation method, specified as one of the following values:

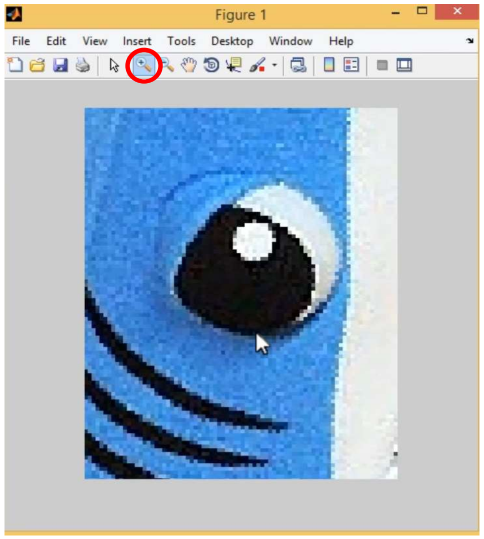
Value	Description
`"nearest"`	Nearest-neighbor interpolation. The output pixel is assigned the value of the pixel that the point falls within. No other pixels are considered. Nearest-neighbor interpolation is the only method supported for categorical images.
`"bilinear"`	Bilinear interpolation. The output pixel value is a weighted average of pixels in the nearest 2-by-2 neighborhood.
`"bicubic"`	Bicubic interpolation. The output pixel value is a weighted average of pixels in the nearest 4-by-4 neighborhood.

Creating the output image

```
>> doc imrotate
>> out = imrotate(im, 30);
>> imshow(out)
```



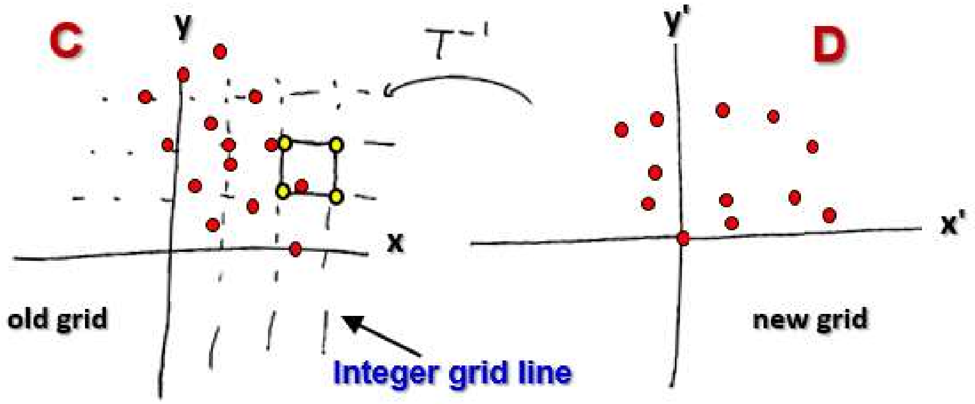
A

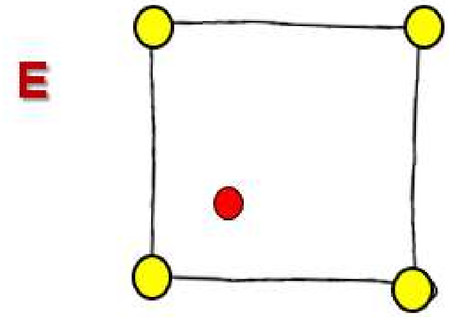


B

- It turns out that in **imrotate** the default method is exactly the **nearest neighbor interpolation**. If we were to look closely at the shark in **A** and zoom in on the shark, **B**, we can see that the pixels are a little blocky or a little bit jaggedy. We can do ***better*** than that with a method which is called **bilinear interpolation**.

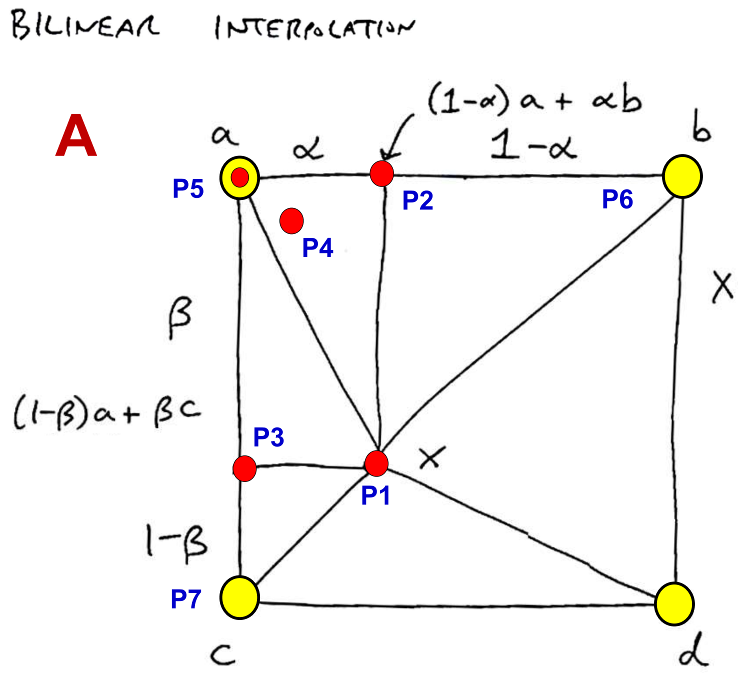
Bilinear interpolation





- **Bilinear Interpolation:** This is a method used in image processing to estimate the value of a pixel based on the values of its four nearest neighbors. In order to see how this method operates, we are going to redraw our previous diagram in more detail. Here, we will focus our attention on the block shown in **E**.

Bilinear interpolation



(2)

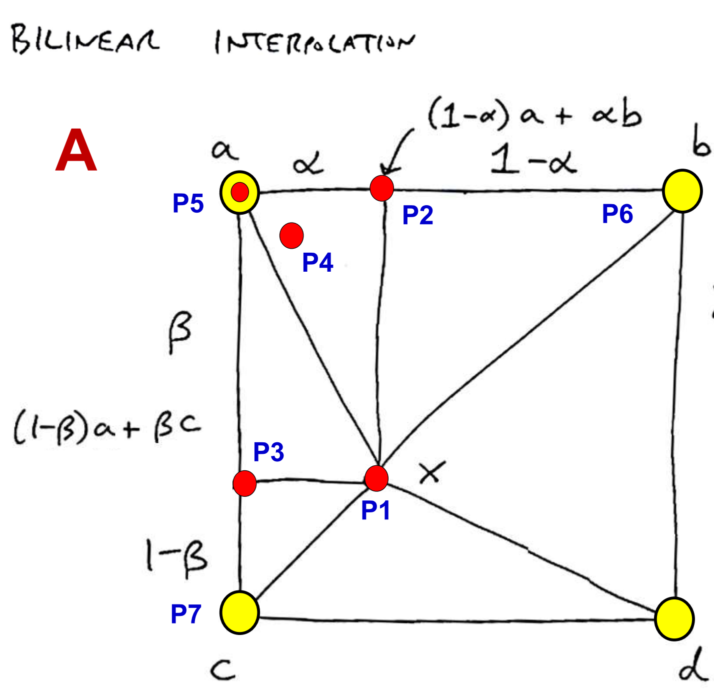
$$X = (1-\alpha)(1-\beta)\,a + (1-\beta)\alpha\,b + (1-\alpha)\beta\,c + \alpha\beta\,d$$

(1)

$$X = \left[(1-\beta)a + \beta c\right](1-\alpha) + \left[(1-\beta)b + \beta d\right]\alpha$$

- Let us suppose that we are blowing up this one square, shown in A, between four ***known pixels***. We are going to call these known pixels, **a**, **b**, **c**, and **d**. So, if our **red** dot lands on **a**, **P5**, then we should just take **a**. If the red dot lands between **a** and **b**, **P2**, we should take a combination of **a** and **b**. For example, if we are right halfway between **a** and **b**, then we should take half and half. If we are closer to **a**, we should take more of **a**. And, if we are closer to **b**, we should take more of **b**.
- The distance between the two ***integer*** coordinates of **a** and **b** is **1**. This means that if we assume that the distance between **P5** and **P2** is $\boldsymbol{\alpha}$, then the distance between **P2** and **P6** is $\mathbf{1} - \boldsymbol{\alpha}$.
- If we are fully in the middle of the block, then we should be basically combining the pixel values from all four of the corner pixels, ***in proportion to*** how close we are to each of them. In general, pixel **x** at **P1** should have basically a combination of the pixels at the four corners, as shown by **(1)**.

Bilinear interpolation



(2)

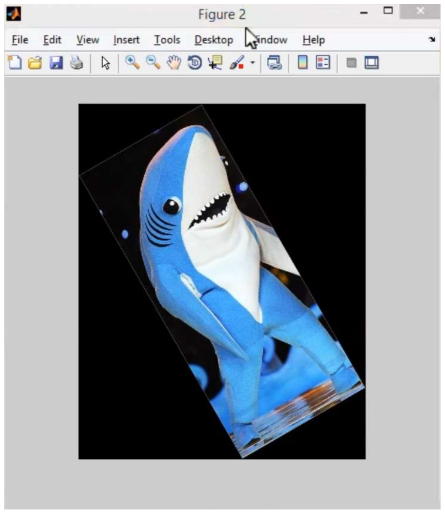
$$
\begin{aligned}
X = &\ (1-\alpha)(1-\beta)\ a \quad \textbf{(2.1)}\\
&+ (1-\beta)\alpha\ b\\
&+ (1-\alpha)\beta\ c\\
&+ \alpha\beta\ d
\end{aligned}
$$

(1) $X = \left[(1-\beta)a + \beta c\right](1-\alpha) + \left[(1-\beta)b + \beta d\right]\alpha$

- As shown by **(2)**, which is basically the expanded form of **(1)**, it should stand to reason that if $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ are both small, that means we are at somewhere like **P4**. This means we should be taking most of our image color from **a**, and that the term **(2.1)** will be really big. Note that all these weights add up to **1**. So, if we are right in the middle, then the weight from each of the corner pixels is going to be important. This is called **bilinear interpolation**.

Bilinear interpolation

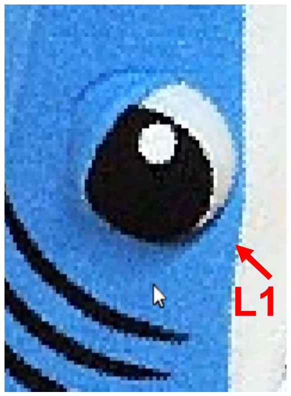
```
>> out2 = imrotate(im, 30, 'bilinear');    (1)

>> imshow(out2)
```



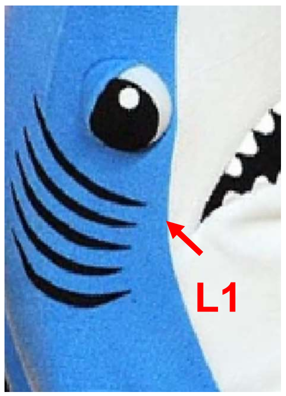
A

- Now, we can see why this is better in MATLAB. If we were to make a new image, using **imrotate** command, but instead, we want it to be bilinear interpolation, **(1)**, we get **A**. Let us zoom in on the eye region of the shark in both the **nearest neighbor interpolation**, **NNI**, and the **bilinear interpolation**, **BLI**, shown in **B** and **C**. There is a huge difference in the quality of the images. For example, the line between the blue and the white, **L1**, looks much smoother in **C** than it does in **B**. Image **B** is a noisier looking image because we are not doing any sort of smoothing between noisy pixels in the original image. Whereas in **C**, we are getting a lot of nicer curves. If we zoom in on the curve below the shark's eye, we can see that it looks kind of jagged in **D**, **L2**, but in **E**, **L2** looks smoother. Perhaps MATLAB does not do bilinear interpolation by default because of computation time.

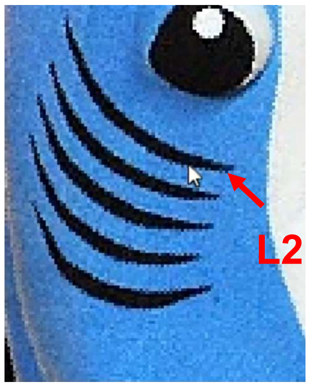
B NNI



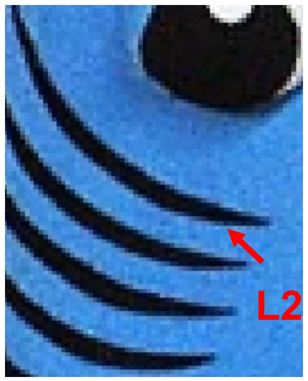
C BLI



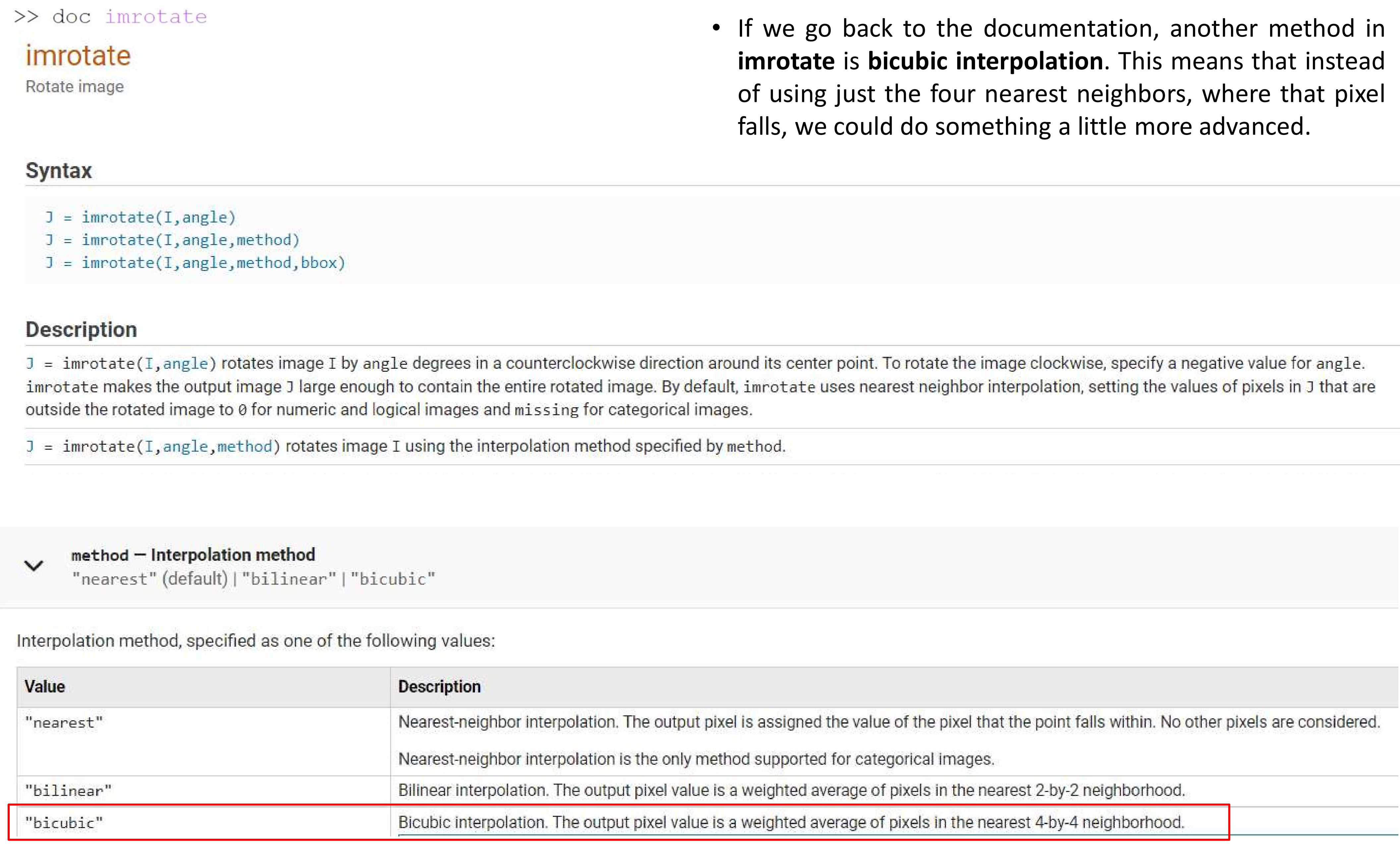
D NNI



E BLI



Bilinear interpolation



```
>> doc imrotate
```

imrotate

Rotate image

Syntax

```
J = imrotate(I,angle)
J = imrotate(I,angle,method)
J = imrotate(I,angle,method,bbox)
```

Description

`J = imrotate(I,angle)` rotates image `I` 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 `J` large enough to contain the entire rotated image. By default, `imrotate` uses nearest neighbor interpolation, setting the values of pixels in `J` that are outside the rotated image to `0` for numeric and logical images and `missing` for categorical images.

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

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

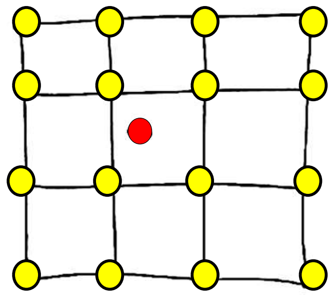
Interpolation method, specified as one of the following values:

Value	Description
`"nearest"`	Nearest-neighbor interpolation. The output pixel is assigned the value of the pixel that the point falls within. No other pixels are considered. Nearest-neighbor interpolation is the only method supported for categorical images.
`"bilinear"`	Bilinear interpolation. The output pixel value is a weighted average of pixels in the nearest 2-by-2 neighborhood.
`"bicubic"`	Bicubic interpolation. The output pixel value is a weighted average of pixels in the nearest 4-by-4 neighborhood.

- If we go back to the documentation, another method in **imrotate** is **bicubic interpolation**. This means that instead of using just the four nearest neighbors, where that pixel falls, we could do something a little more advanced.

Extensions

A

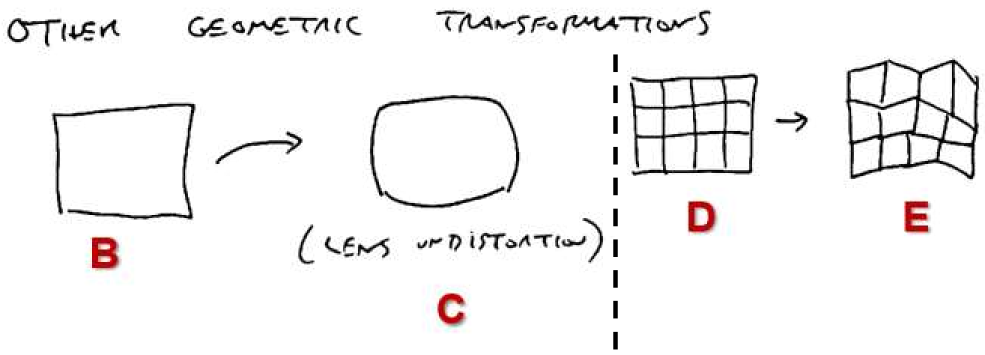


BICUBIC INTERPOLATION –

USES MORE POINTS;
LOOK SMOOTHER

- **Bicubic Interpolation:** Let us say **A** is our original grid of image pixels in which the **red** pixel falls into. Here, instead of just using the **4** neighbors of the red pixel to estimate the pixel intensity, we actually bring in all the pixels that we can use, i.e., all **16** neighbors. We can imagine that we could fit a nice bicubic spline surface to this grid. Bicubic interpolation uses more points and the image is likely to look smoother.

Extensions



$$x' = f(x,y) \quad (1)$$
$$y' = g(x,y) \quad (2)$$

- There are lots of geometric transformations that do not fit into affine or projective. That is, many other geometric transformations also exist, such as, **lens undistortion**. Here, the edges of image **B** blow out a little bit and turn into **C**.
- We could also do something even more complicated, where we are going to take image **D**, and we are going to grit it up, and then move each of the control points separately. Now, we could work every rectangle of the image on its own to make them the weird looking distorted rectangles. Then, we could do bilinear interpolation inside each of these deformed quadrangles. This would also be a geometric transformation. So, all these nonlinear transformations are also possible.
- **General Geometric Transformation:** There is all sorts of weird local distortion we could do and all that means is that instead of having a nice function that is an affine transformation, we can have some sort of complicated functions shown by **(1)** and **(2)**. That is, the output is some complicated function of the input pixel. This is called a **general geometric transformation**.

End of Lecture 5