Graphics Manipulation

Graphics Manipulation

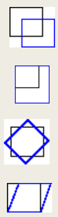
- The manipulation of graphical shape (vector images).
- We will consider scaling, translation and rotation.
- Can be applied to bitmaps.

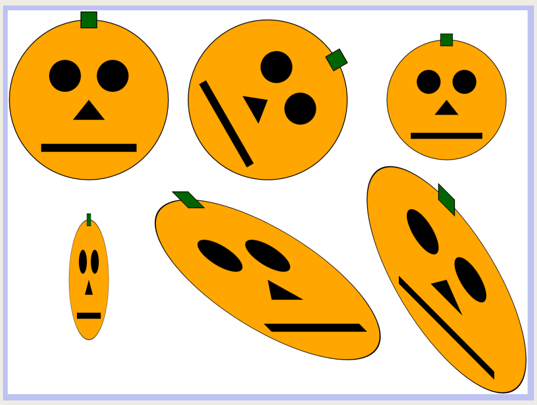
Graphic Manipulation

- Manipulation achieved as a series of transformations.
- These transformation achieved by matrices.
- Called transformation matrices.
- Complex manipulation is achieved by combination (multiplication) of the basic transformation matrices.
- We will see that the order of multiplication is important.

Manipulating a picture using affine transformation

- In bitmaps we manipulate each pixel in the image.
- Affine transformation is a linear mapping method that preserves points, straight lines, and planes. Sets of parallel lines remain parallel after an affine transformation.
- There are three basic operations:
  - *Translation*
  - *Scaling*
  - *Rotation*
  - *Shear*





The equations of manipulation

- Translation
$$\begin{aligned} x' &= x + T_x \\ y' &= y + T_y \end{aligned}$$
- Scaling.
$$\begin{aligned} x' &= x \times S_x \\ y' &= y \times S_y \end{aligned}$$
- Rotation.
$$\begin{aligned} x' &= x\cos\theta - y\sin\theta \\ y' &= x\sin\theta + y\cos\theta \end{aligned}$$

Matrix multiplication

- Multiply each element of a row of the first matrix by each element of the corresponding column of the second matrix.
- Add the products together.
- Put the result in position (row of first matrix, column of second matrix) of a new matrix.
- Repeat for all rows of the first matrix.
- Means that the number of rows of the first matrix must be the same as the number of columns of the second matrix.
- Easy for computers.

Matrix multiplication

- Example:

$$\begin{bmatrix} 5 & 8 \\ 7 & 2 \end{bmatrix} \times \begin{bmatrix} 6 & 1 \\ 4 & 3 \end{bmatrix}$$

$$= \begin{bmatrix} (5\times 6)+(8\times 4) & (5\times 1)+(8\times 3) \\ (7\times 6)+(2\times 4) & (7\times 1)+(2\times 3) \end{bmatrix}$$

$$= \begin{bmatrix} 62 & 29 \\ 50 & 13 \end{bmatrix}$$

- Try it with Matlab.

A=[5 8 ; 7 2]
B=[6 1 ; 4 3]
*A*B*
*Try B*A*
Not the same.

- *Watch this video if you don't get it now.*

https://www.youtube.com/watch?v=kuixY2bCc_0

The equations in a matrix form.

- Scaling as an example:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} S_x & 0 \\ & S_y \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix}$$

- But…

$$x' = ax + by + t_x$$
$$y' = cx + dy + t_y$$

→

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}$$

Homogeneous Coordinates

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

"Subsequent" operations are inserted here, by pre-multiplying

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

The equations in a "homogeneous" matrix form.

- Translation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- Scaling

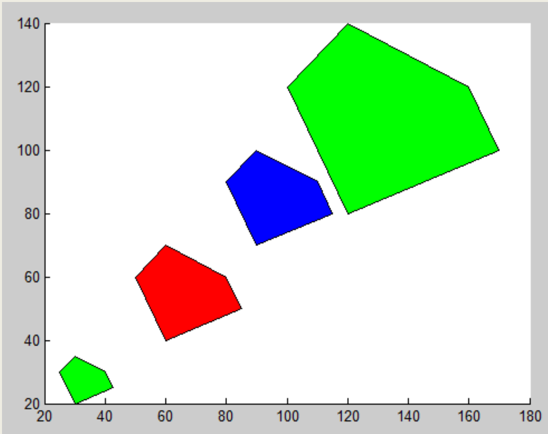
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- Rotation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

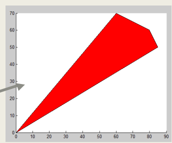
Scaling & transforming in Graphics system

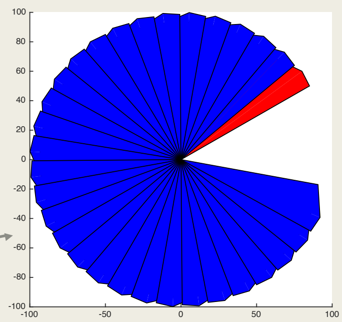
```
xy=[50 60 80 85 60;  60 70  60 50 40; 1 1 1 1 1];
x=xy(1,1:5);
y=xy(2,1:5);
patch(x,  y, 'r');
%translate  the shape
RT=[1 0 30;  0  1  30;  0  0 1];
   xyp=RT*xy;
   x=xyp(1,1:5);
   y=xyp(2,1:5);
patch(x,  y,  'b');

%Scale  up the shape
RT=[2 0 0;  0  2 0;  0  0  1];
   xyp=RT*xy;
   x=xyp(1,1:5);
   y=xyp(2,1:5);
patch(x,  y, 'g');

%Scale  down  the shape
RT=[0.5  0  0;  0 0.5  0;  0 0  1];
   xyp=RT*xy;
   x=xyp(1,1:5);
   y=xyp(2,1:5);
patch(x,  y, 'g');
```



Rotating in Graphics system

```
xy=[0 60  80  85;
      0 70 60  50;
      1 1  1  1];
x=xy(1,1:4);
y=xy(2,1:4);
patch(x,  y,  'r');

% produce rotated  object
for  th=10:10:300
   % Rotation clockwise
   RT=[cosd(th)  -sind(th) 0;
          sind(th)   cosd(th)  0;
          0  0  1];
   xyp=RT*xy;
   x=xyp(1,1:4);
   y=xyp(2,1:4);
   patch(x,  y,  'b');
end

% To  rotation anti-clockwise  use
this instead
    % RT=[cosd(th)  sind(th);
    %        -sind(th)  cosd(th)];
```





In rotation we should consider:

1. Point of rotation
2. Direction of rotation
3. Degree of rotation

Matlab rotate metrices

Command Window

```
>> rotx(30)

ans =

    1.0000         0         0
         0    0.8660   -0.5000
         0    0.5000    0.8660
```

Command Window

```
>> roty(30)

ans =

    0.8660         0    0.5000
         0    1.0000         0
   -0.5000         0    0.8660
```

Command Window

```
>> rotz(30)

ans =

    0.8660   -0.5000         0
    0.5000    0.8660         0
         0         0    1.0000
```

Exercises in translation, scaling and rotation

Now create your own transformation matrices for the following:

- Translation
  - *Move a shape 20 pixels up and 30 pixels across*
- Scaling
  - *Scale a shape to a tenth of its height and a third of its width.*
- Rotation
  - *Rotate a shape by 45º clockwise.*

Compound Transformations

By multiplying different types of transformation matrix, we can do two (or more) transformations at once. Effectively this transforms the already transformed image.

- For example:

Scaling followed by rotation.

$$scale = \begin{bmatrix} 0.5000 & 0 & 0 \\ 0 & 0.2500 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$rotate = \begin{bmatrix} 0.7070 & -0.7070 & 0 \\ 0.7070 & 0.7070 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{aligned} compound &= rotate \times scale \\ &= \begin{bmatrix} 0.7070 & -0.7070 & 0 \\ 0.7070 & 0.7070 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 0.5000 & 0 & 0 \\ 0 & 0.2500 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 0.3535 & -0.1767 & 0 \\ 0.3535 & 0.1767 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

Compound Transformations

- For example:

Scaling followed by translation.

$$scale = \begin{bmatrix} 0.5000 & 0 & 0 \\ 0 & 0.2500 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

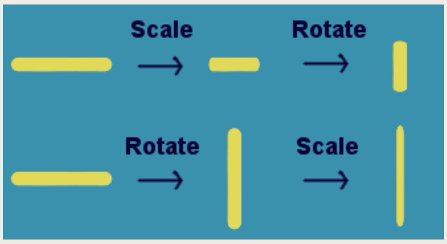
$$translate = \begin{bmatrix} 1 & 0 & 20 \\ 0 & 1 & 50 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{aligned} compound &= translate \times scale \\ &= \begin{bmatrix} 1 & 0 & 20 \\ 0 & 1 & 50 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 0.5000 & 0 & 0 \\ 0 & 0.2500 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 0.5000 & 0 & 20 \\ 0 & 0.2500 & 50 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

Order of transformation matrices

- Note that Scaling followed by translation is not the same as translation followed by scaling, because we scale the translation in the second case.

- Rotation followed by scaling distorts the transformation matrix and results in “skew”.



Order of transformation matrices

- This is true of all compound transformations, so the order is important.

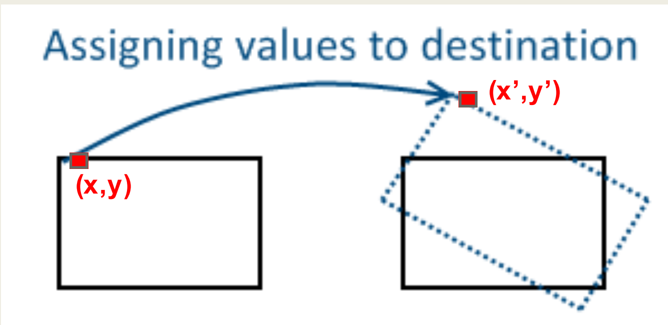
- Also, because the transformation matrix is multiplied by the object to be transformed, rather than the other way around, “later” operations are on the left of any expression. This may look like the wrong way round.

Exercises in compound transformations.

- Produce a matrix which produces a two times increase in height, halves the width, and rotates the original shape.
- Repeat the above, but move the shape 40 pixels down and 100 pixels across also.
- Now reverse the order of the transformations and observe the effect.
- Try your own.

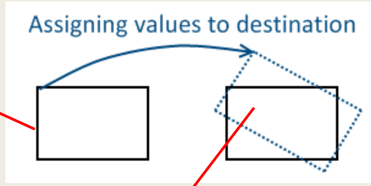
How to manipulate images using matrix transformation (rotation)

forward mapping

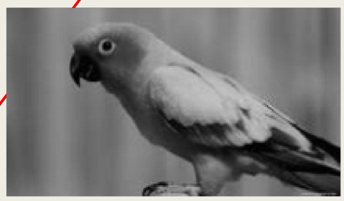


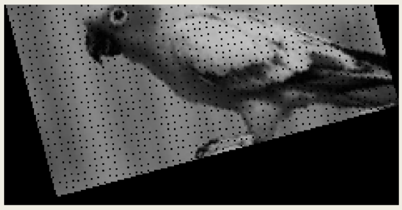
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

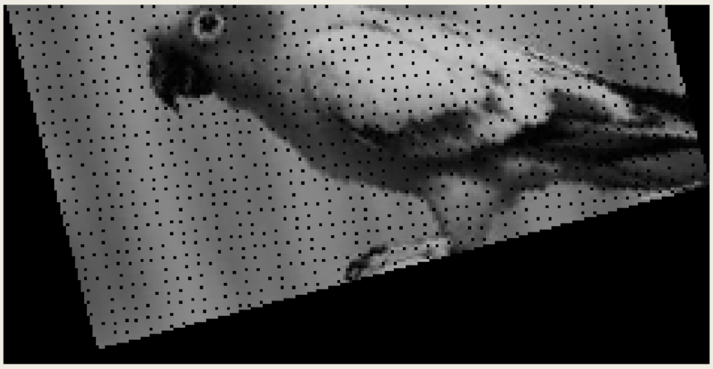
```
input_matrix  =
rgb2gray(imread( 'download1.bmp' ));
[rows,  cols] = size(input_matrix);

% rotation
degree  = 15;
radians  = (pi  * degree)  /  180;
theta  = radians;

% output  matrix
t_matrix  = uint8(zeros(rows,cols));

% transformation  matrix
T = [cos(theta)  -sin(theta)  0; sin(theta)
cos(theta)  1;  0 0  1];

% loop over  each  input_matrix  coordinate
for n = 1:numel(input_matrix)
    % current  coordinate
    [x,  y] = ind2sub([rows  cols],  n);
    v = [x;y;1];
    v = T*v;
    % only  integer  values
    a = floor(v(1));
    b = floor(v(2));
    if a > 0 && b > 0
        % replace  in t_matrix
        t_matrix(a,b)  = input_matrix(x,y);
    end
end
```



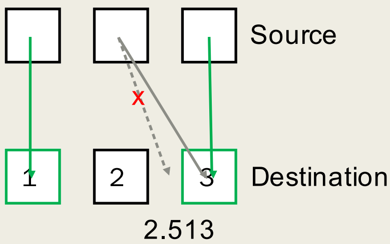
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$



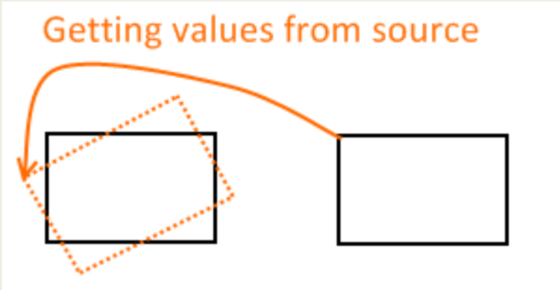




Note the speckled black pixels dotted all over. This is because some of the destination pixels (which are within the image bounds) were unassigned.
Note also that the even form patterns. This is due to the sine and cosine functions, and the regularity of pixel width and height. Sine and cosine are periodic functions. Since pixel indices are regular, therefore sine and cosine results are regular too. Hence, calculations regularly fail to assign pixel values.

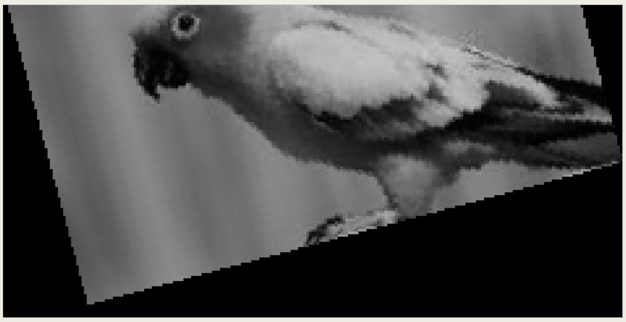


Reverse mapping

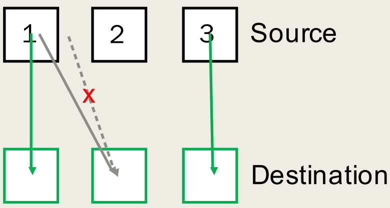


Use invert transformation matrix to back trace the source pixel.

```
...
% inverse transformation  matrix
IVT=inv(T);
v = IVT*v;
a = floor(v(1));
b = floor(v(2));
if rows>a  && a > 0 && cols>b  && b > 0
 % replace  in t_matrix
       t_matrix2(x,y)   = input_matrix(a,b);
    end
end
...
```

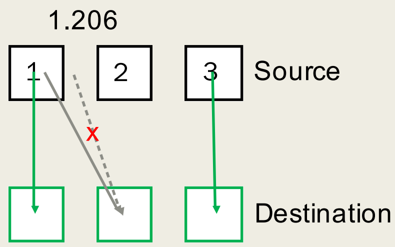


1.206

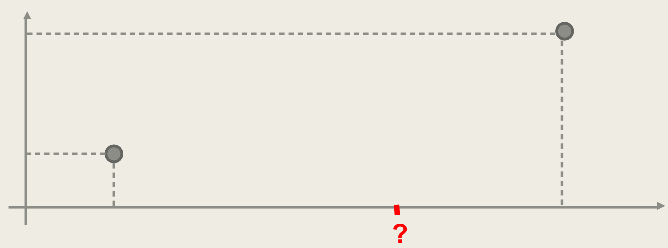


Compare the quality with the source-to-destination part. No missing pixels. It's still sort of grainy though. This is because some of the destination pixels get their values from the same source pixel, so there might be 2 side-by-side destination pixels with the same colour. This gives mini blocks of identical colour in the result, which on the whole, gives an unpolished look.

What would we find between at a position between pixel #1 and #2?

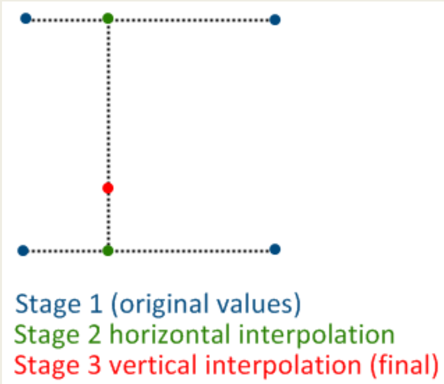


Linear and Bilinear interpolation

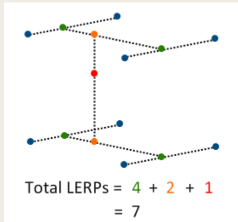


Interpolation is a method of estimating a value from a set of given values.

Linear and Bilinear interpolation

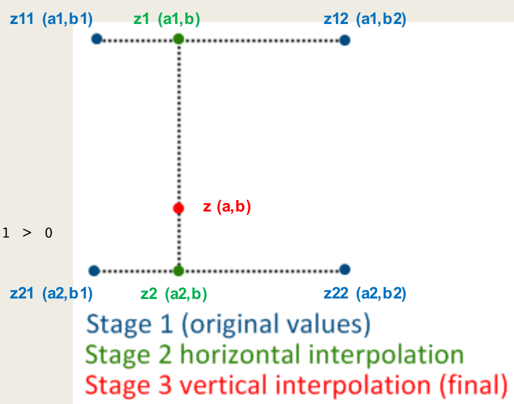


“Bilinear” means there are 2 directions to interpolate. In our case, we’re interpolating between 4 pixels. Visualise each pixel as a single point. Linearly interpolate between the top 2 pixels. Linearly interpolate between the bottom 2 pixels. Then linearly interpolate between the calculated results of the previous two. You can expand on this concept to get trilinear interpolation.

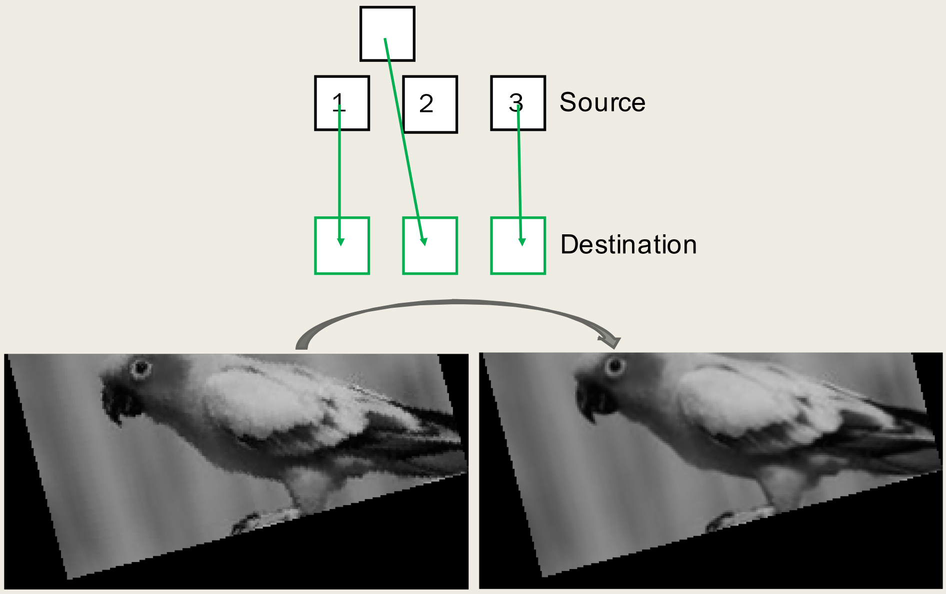


http://polymathprogrammer.com/2008/09/29/linear-and-cubic-interpolation/

```
for n = 1:numel(t_matrix)

    % current coordinate
    [x, y] = ind2sub([rows cols], n);

    v = [x;y;1];
    v = IVT*v;

    a=v(1);
    b=v(2);

    a1 = floor(a);
    a2 = ceil(a);
    b1 = floor(b);
    b2 = ceil(b);

    if rows>a2 && a1 > 0 && cols>b2 && b1 > 0

    z11=double(input_matrix(a1,b1));
    z12=double(input_matrix(a1,b2));
    z21=double(input_matrix(a2,b1));
    z22=double(input_matrix(a2,b2));

    z1=z11+(b-b1)*(z12-z11)/(b2-b1);
    z2=z21+(b-b1)*(z22-z21)/(b2-b1);
    z=z1+(z2-z1)*(a-a1)/(a2-a1);

    t_matrix3(x,y) = uint8(z);

    end
end
```



Smooth image rotation using bilinear interpolation



Example codes in NILE for exercise

```
% input image
input_matrix = rgb2gray(imread('download1.bmp'));
[rows, cols] = size(input_matrix);

% rotation
degree = 15;
radians = (pi * degree) / 180;
theta = radians;

% output matrix
t_matrix = uint8(zeros(rows,cols));
t_matrix2 = uint8(zeros(rows,cols));
t_matrix3 = uint8(zeros(rows,cols));

% transformation matrix
T = [cos(theta) -sin(theta) 0; sin(theta) cos(theta) 1; 0 0 1];

% inverse transformation matrix
IVT=inv(T);

% loop over each input_matrix coordinate
for n = 1:numel(input_matrix)
    % current coordinate
    [x, y] = ind2sub([rows cols], n);
    v = [x;y;1];

    % homogeneous coordinate
    v = T*v;

    % only integer values
    a = floor(v(1));
    b = floor(v(2));

    if a > 0 && b > 0
        % replace in t_matrix
        t_matrix(a,b) = input_matrix(x,y);
    end
end

for n = 1:numel(t_matrix)
    % current coordinate
    [x, y] = ind2sub([rows cols], n);

    % transpose
    v = [x;y;1];

    % homogeneous coordinate
    v = IVT*v;

    % only integer values
    a = floor(v(1));
    b = floor(v(2));

    if rows>a && a > 0 && cols>b && b > 0
        % replace in t_matrix
        t_matrix2(x,y) = input_matrix(a,b);
    end
end

for n = 1:numel(t_matrix)
    % current coordinate
    [x, y] = ind2sub([rows cols], n);

    % transpose
    v = [x;y;1];

    % homogeneous coordinate
    v = IVT*v;
    a=v(1);
    b=v(2);

    a1 = floor(a);
    a2 = ceil(a);
    b1 = floor(b);
    b2 = ceil(b);

    if rows>a2 && a1 > 0 && cols>b2 && b1 > 0

    z11=double(input_matrix(a1,b1));
    z12=double(input_matrix(a1,b2));
    z21=double(input_matrix(a2,b1));
    z22=double(input_matrix(a2,b2));

    z1=z11+(b-b1)*(z12-z11)/(b2-b1);
    z2=z21+(b-b1)*(z22-z21)/(b2-b1);
    z=z1+(z2-z1)*(a-a1)/(a2-a1);

    t_matrix3(x,y) = uint8(z);

    end
end

figure(1); imshow(input_matrix); % original image
figure(2); imshow(t_matrix);
figure(3); imshow(t_matrix2);
figure(4); imshow(t_matrix3);
```