컴퓨터 비젼

[HW05]

학번	201203393
분반	00
이름	김헌겸
과제번호	05

제출일 : 2016년 4월 26일 화요일

1. (MATLAB) 2D 이미지에 대해 up-sampling nearest neighbor보간, bilinear 보간, bicubic 보간을 직접 구현하시오. 그리고 MATLAB function imresize()와 비교하시오.

interpolation이 구현되는 원리는 모두 같다. up-sampling과정에서 원본 이미지에서 factor 값 만큼 커진 크기의 매트릭스를 만들고, 이 매트릭스의 각 지점을 어떤 픽셀 값으로 채울지를 원본 이미지로부터 계산하여 얻어오는 것이다.

```
function scaled = up_sampling(im, factor, method)

im_size = size(im);
factored_x = ceil(im_size(1)*factor);
factored_y = ceil(im_size(2)*factor);
scaled = zeros(factored_x, factored_y);
```

우선 X와 Y에 대해서 크기가 factor 만큼 커진 이미지를 얻는 것이므로 im_size를 계산한 후, factor를 곱해 scaled된 이미지의 크기 값을 얻는다. 그리고 해당 크기만큼의 zero 매트릭스를 생성해낸다. 그리고 각 interpolation 방식마다 주어진 함수를 가지고 크기가 커진 매트릭스의 데이터를 어떻게 채울 것인지 연산하면 되는 방식이다.

```
function scaled = up_sampling(im, factor, method)

im_size = size(im);
factored_x = ceil(im_size(1)*factor);
factored_y = ceil(im_size(2)*factor);
scaled = zeros(factored_x, factored_y);

if strcmp(method,'nearest')
    for i=1 : factored_x
        for j=1 : factored_y
            x_round = round(i/2.7);
            y_round = round(j/2.7);

            if x_round == 0
                x_round = 1;
            end
            if y_round == 0
                y_round = 1;
            end
            s = im(x_round, y_round);
            scaled(i,j) = s;
        end
    end
```

(nearest neightbor 방식의 interpolation 구현)

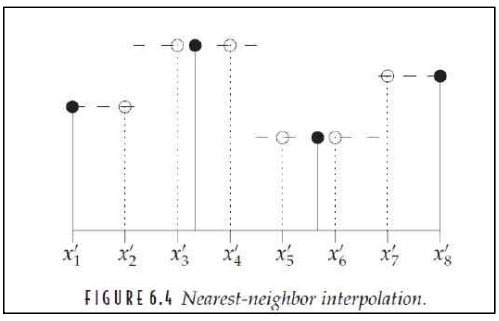
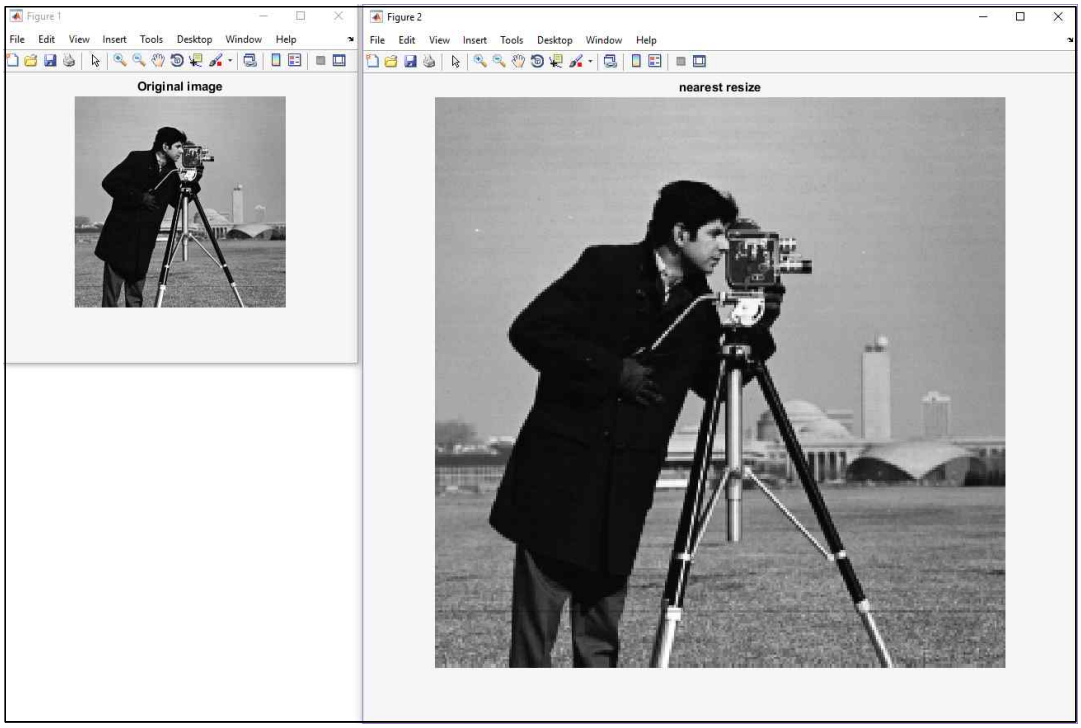


FIGURE 6.4 *Nearest-neighbor interpolation.*

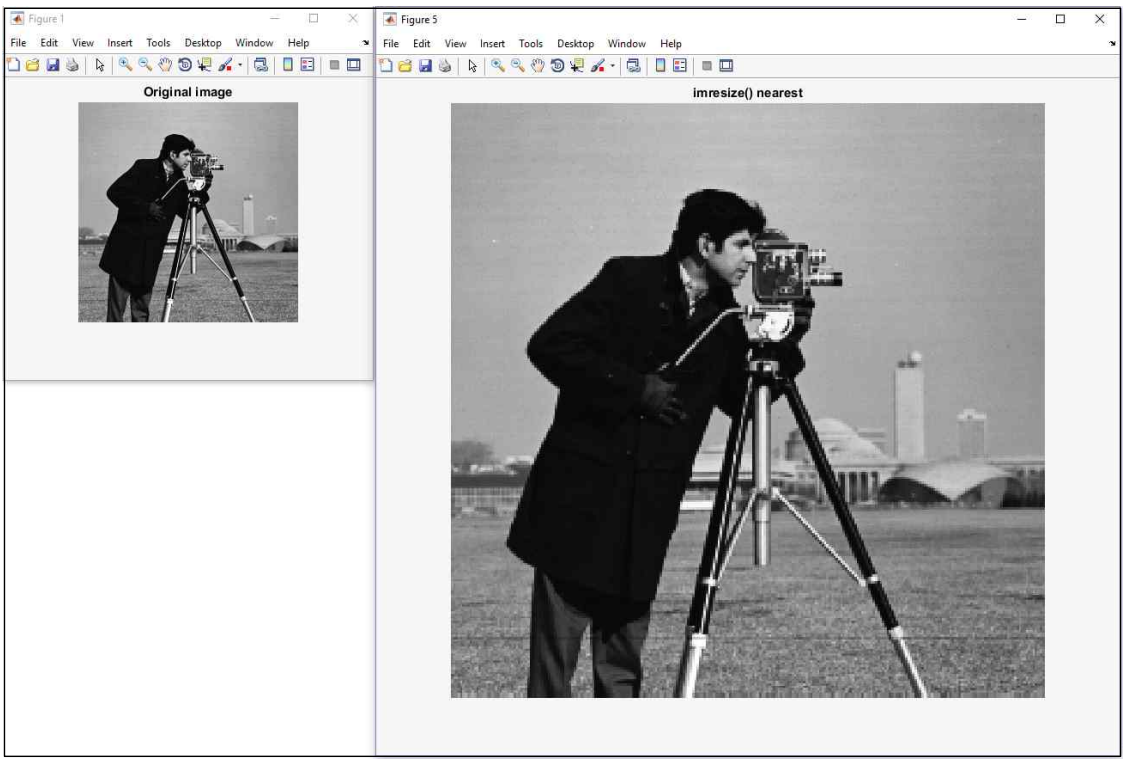
$$f(x') = R_0(-\lambda)\, f(x_1) + R_0(1-\lambda) f(x_2)$$

$$\Rightarrow\ f(x') = \begin{cases} f(x_1) & if\ \lambda < 0.5 \\ f(x_2) & otherwise \end{cases}$$

nearest neighbor 방식은 주어진 픽셀에 대해서 가장 근처에 있는 값을 가지고 계산하는 방식이다. 위에 있는 식은 1-D에 대한 식이므로 이를 2차원 X, Y 좌표에 대해서 똑같이 연산을 해주면 nearest neighbor 방식을 구현해 볼 수 있다.



(up_sampling(im, 2.7, 'nearest'))



(imresize(im, 2.7, 'nearest'))

```
elseif strcmp(method, 'bilinear')
    scale_x = factored_x./(im_size(1)-1);
    scale_y = factored_y./(im_size(2)-1);
    for i = 0 : factored_x-1
        for j = 0 : factored_y-1
            W = -(((i./scale_x)-floor(i./scale_x))-1);
            H = -(((j./scale_y)-floor(j./scale_y))-1);
            I11 = im(1+floor(i./scale_x),1+floor(j./scale_y));
            I12 = im(1+ceil(i./scale_x),1+floor(j./scale_y));
            I21 = im(1+floor(i./scale_x),1+ceil(j./scale_y));
            I22 = im(1+ceil(i./scale_x),1+ceil(j./scale_y));
            scaled(i+1,j+1) = (1-W).*(1-H).*I22 + (W).*(1-H).*I21 + (1-W).*(H).*I12 + (W).*(H).*I11;
        end
    end
```

(bilinear방식의 interpolation 구현)

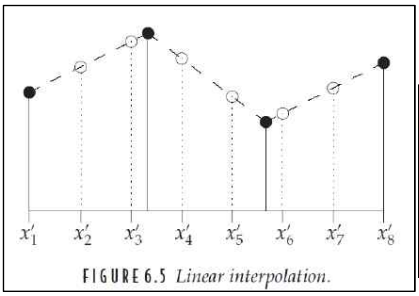
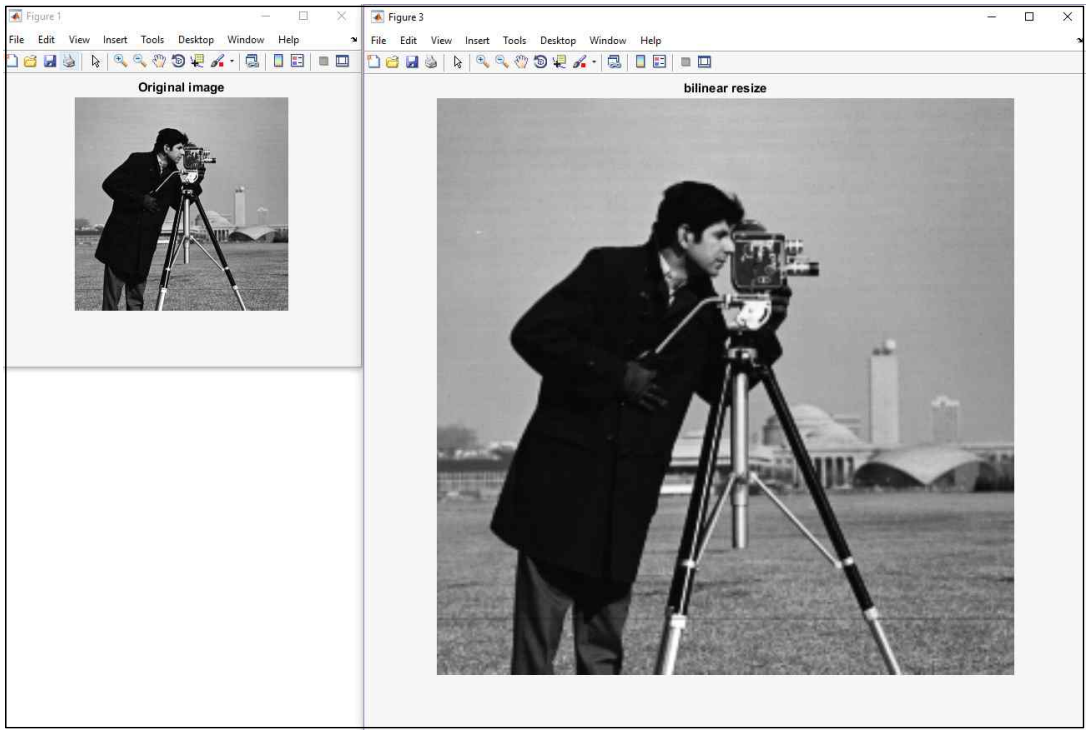


FIGURE 6.5 *Linear interpolation.*

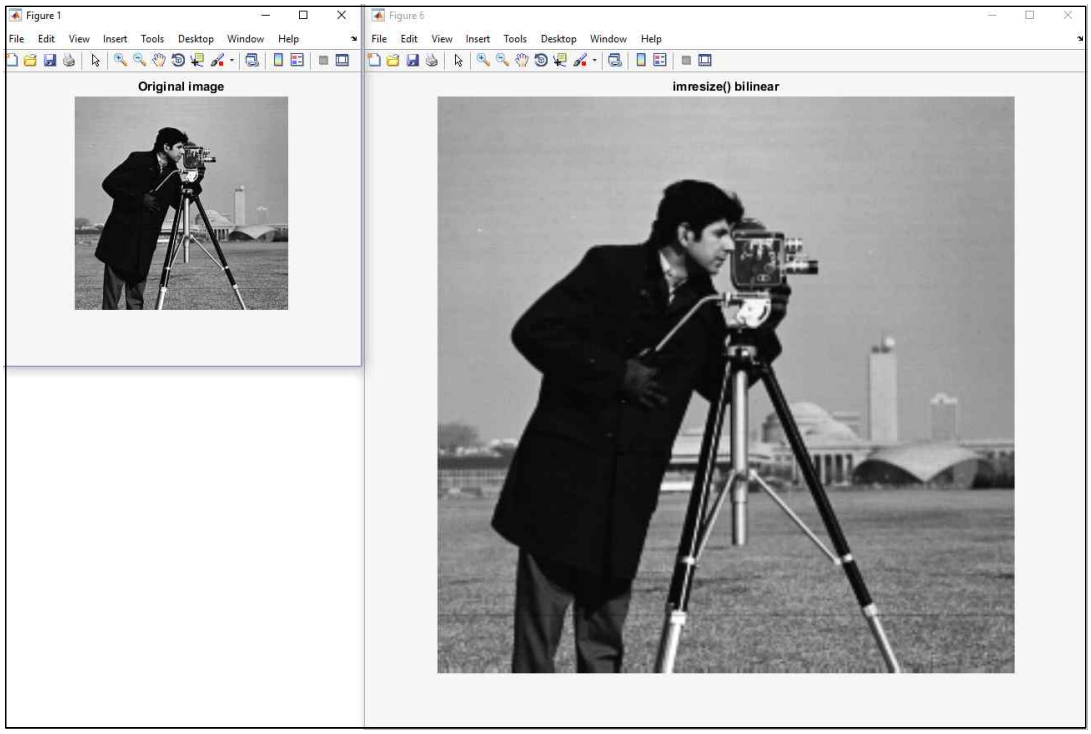
$$f(x') = R_1(-\lambda)\, f(x_1) + R_1(1-\lambda) f(x_2)$$

$$f(x') = (1-\lambda)\, f(x_1) + \lambda f(x_2)$$

bilinear의 경우, 원본이미지의 임의 두 점 사이에 선을 긋고, 그 사이에 있는 모든 점들을 이 선의 값으로 채우는 방법이다. 역시 위 그래프와 일반화식은 1-D linear 방식이므로 2-D에 대해서 연산한다. 위의 코드에서는 한 점을 둘러싸고 있는 네 개의 점을 직접 계산한 다음 마지막에 모두 더해주는 방식을 사용하고 있다.



(up_sampling(im, 2.7, 'bilinear'))



(imresize(im, 2.7, 'bilinear'))

```
elseif strcmp(method, 'bicubic')
    temp = [im(2:-1:1,:); im; im(end:-1:end-(2-1), :)];
    temp = [temp(:, 2:-1:1), temp, temp(:, end:-1:end-(2-1))];

    for i = 3 : factored_x+2
        x2 = floor(i/factor);
        x3 = ceil(i/factor);
        x1 = x2 - 1; x4 = x3 + 1;

        if x1 == 0
            x1 = 1;
        end

        W = (i/factor)-x2;

        for j = 3 : factored_y+2

            y2 = floor(j/factor);
            y3 = ceil(j/factor);

            y1 = y2 - 1; y4 = y3 + 1;

            if y1 == 0
                y1 = 1;
            end

            H = (j/factor)-y2;

            Y1 = r3(-1-H)*(r3(-1-W)*temp(x1,y1)+r3(-W)*temp(x2,y1)+r3(1-W)*temp(x3,y1)+r3(2-W)*temp(x4,y1));
            Y2 = r3(-H)*(r3(-1-W)*temp(x1,y2)+r3(-W)*temp(x2,y2)+r3(1-W)*temp(x3,y2)+r3(2-W)*temp(x4,y2));
            Y3 = r3(1-H)*(r3(-1-W)*temp(x1,y3)+r3(-W)*temp(x2,y3)+r3(1-W)*temp(x3,y3)+r3(2-W)*temp(x4,y3));
            Y4 = r3(2-H)*(r3(-1-W)*temp(x1,y4)+r3(-W)*temp(x2,y4)+r3(1-W)*temp(x3,y4)+r3(2-W)*temp(x4,y4));
            scaled(i-2,j-2) = round(Y1 + Y2 + Y3 + Y4);

        end
    end
```

(bicubic방식의 interpolation 구현)

bicubic은 좀 더 세밀하게 픽셀 값을 정하기 위해 주변의 16개의 점을 가지고 연산을 하는 방식이다. bicubic에 사용되는 식은 아래와 같다.

$$R_3(u) = \begin{cases} 1.5|u|^3 - 2.5|u|^2 + 1 & if\ |u| \le 1 \\ -0.5|u|^3 + 2.5|u|^2 - 4|u| + 2 & if\ 1 < |u| \le 2 \end{cases}$$

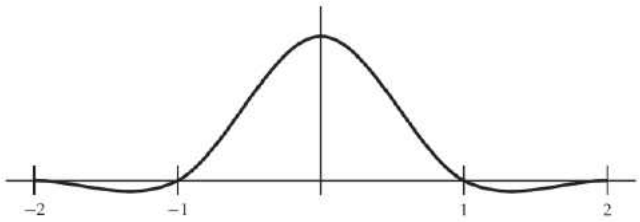


FIGURE 6.B The cubic interpolation function $R_3(u)$.

```
function y = r3(u)
abs_u = abs(u);
if abs_u <= 1
    y = (1.5 * (abs_u^3)) - (2.5 * (abs_u^2)) + 1;
elseif abs_u > 1 && abs_u <= 2
    y = (-0.5 * (abs_u^3)) + (2.5*(abs_u^2)) + (-4*abs_u) + 2;
end
```

절대 값 u에 대해서 위에 주어진 R3 식을 사용하여 y값을 얻어 낸다. 그리고

$$f(x') = R_3(-1-\lambda)f(x_1) + R_3(-\lambda)f(x_2) + R_3(1-\lambda)f(x_3) + R_4(2-\lambda)f(x_4),$$

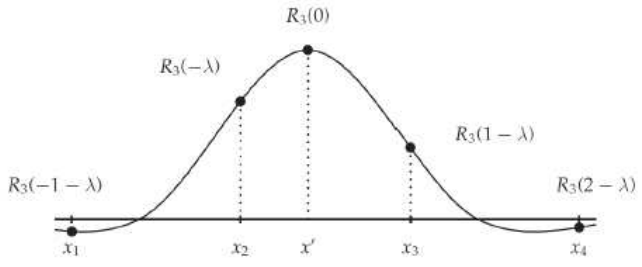
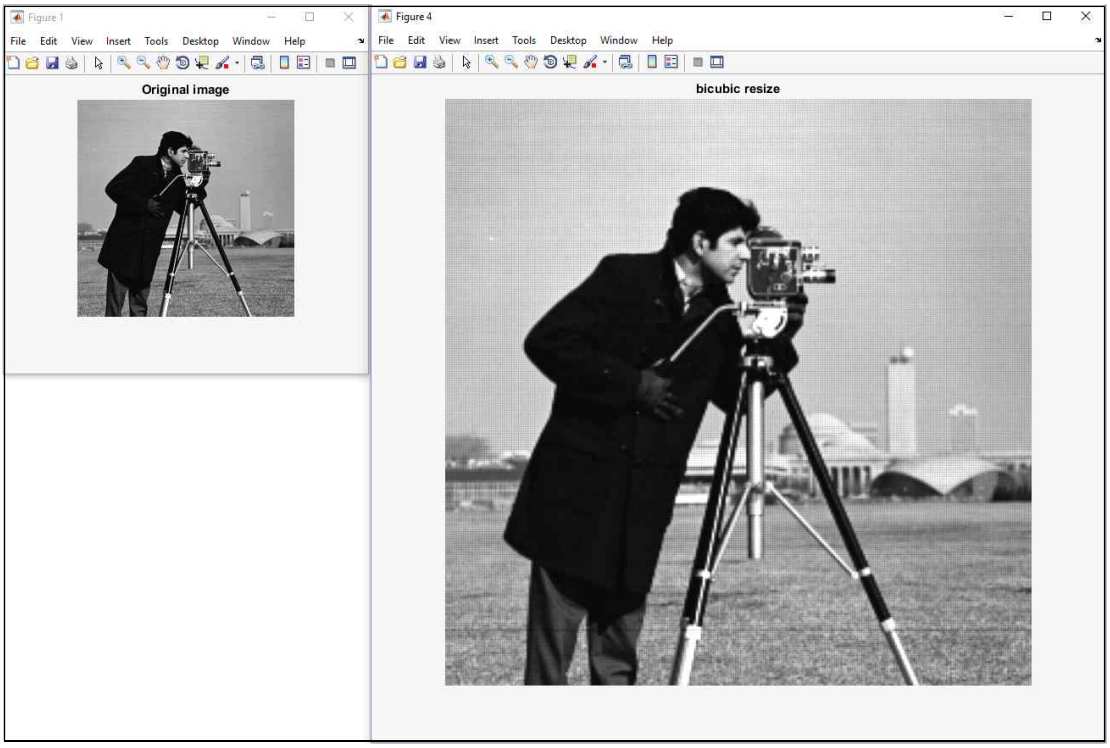
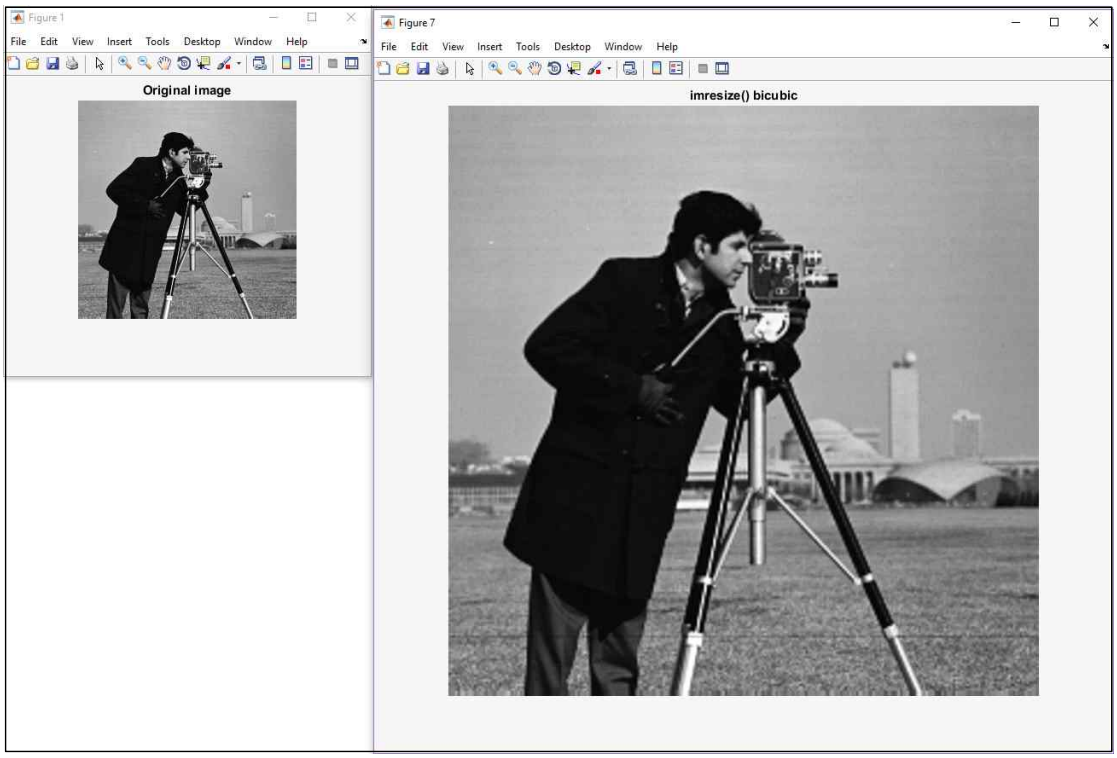


FIGURE 6.14 Using $R_3(u)$ for interpolation.

위의 그래프에 나타나있는 f(x) 값을 계산해 내면 된다. 위의 코드 구현에서는 각 x의 값에 대해서 4개의 점에 대한 interpolation을 계산하고 마지막에 이를 한번에 합치는 방법으로 계산해내었다.



(up_sampling(im, 2.7, 'bicubic'))



(imresize(im, 2.7, 'bicubic))

2. (MATLAB) 이미지 회전함수를 nearest neighbor보간, bilinear보간, bicubic보간을 사용하여 직접 구현하고, MATLAB function imrotate()와 비교하시오.

```
function rotated = imrotate_func(im, angle, method)

s = size(im);
rot_x = round(sqrt(s(1)^2 + s(2)^2));
rot_y = round(sqrt(s(1)^2 + s(2)^2));
radian = pi/180*angle;
rotated = zeros(rot_x, rot_y);

for i=1:rh
    for j=1:rw
        x = (i-rh/2)*cos(radian) + (j-rw/2)*sin(radian) + s(2)/2;
        y = -(i-rh/2)*sin(radian) + (j-rw/2)*cos(radian) + s(1)/2;
        if x <= s(1) && y<=s(2)
            if strcmp(method, 'nearest')
                ni = round(x);
                nj = round(y);
                if ni>=1 && nj>=1 && ni<=256 && nj<=256
                   rotated(i,j) = im(ni, nj);
                end
```

(nearest interpolation에 대한 rotate 함수)

```
elseif strcmp(method, 'bilinear')
        x1 = floor(x);
        y1 = floor(y);
        x2 = ceil(x);
        y2 = ceil(y);
        w = x-x1;
        h = y-y1;
    if x1>=1  && y1>=1 && x2<=256 && y2<=256
        rotated(i,j) = (1-w)*(1-h)*im(x1,y1) + w*(1-h)*im(x2,y1) + (1-w)*h*im(x1,y2) + w*h*im(x2,y2);
    end
```

(bilinear interpolation에 대한 rotate 함수)

```
elseif strcmp(method, 'bicubic')
    temp = [im(2:-1:1,:); im; im(end:-1:end-(2-1), :)];
    temp = [temp(:, 2:-1:1), temp,temp(:, end:-1:end-(2-1))];

    x2 = floor(x); y2 = floor(y);
    x3 = ceil(x);  y3 = ceil(y);
    x1 = x2 - 1;   x4 = x3 + 1;
    y1 = y2 - 1;   y4 = y3 + 1;

    w = (x)-x2;
    h = (y)-y2;

    if x1>=1  && y1>=1 && x2<=256 && y2<=256
        z1 = r3(-1-h)*(r3(-1-w)*temp(x1,y1)+r3(-w)*temp(x2,y1)+r3(1-w)*temp(x3,y1)+r3(2-w)*temp(x4,y1));
        z2 = r3(-h)*(r3(-1-w)*temp(x1,y2)+r3(-w)*temp(x2,y2)+r3(1-w)*temp(x3,y2)+r3(2-w)*temp(x4,y2));
        z3 = r3(1-h)*(r3(-1-w)*temp(x1,y3)+r3(-w)*temp(x2,y3)+r3(1-w)*temp(x3,y3)+r3(2-w)*temp(x4,y3));
        z4 = r3(2-h)*(r3(-1-w)*temp(x1,y4)+r3(-w)*temp(x2,y4)+r3(1-w)*temp(x3,y4)+r3(2-w)*temp(x4,y4));
        rotated(i,j) = z1 + z2 + z3 + z4;
    end
end
```

(bicubic interpolation에 대한 rotate 함수)

1번에서 구현했던 interpolation 방식들에 대해서 rotation을 적용한다.

1. $(x, y) \rightarrow (x', y')$: Obtain an floating output pixel (x', y') after image rotation based on an origin.

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

2. $(x', y') \rightarrow (x'_{in}, y'_{in})$: Find the nearest integer pixel (x'_{in}, y'_{in}) of th floating output pixel (x', y').

3. $(x'_{in}, y'_{in}) \rightarrow (x'', y'')$: Obtain the floating input pixel (x'', y'')

$$\begin{bmatrix} x'' \\ y'' \end{bmatrix} = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} \begin{bmatrix} x'_{in} \\ y'_{in} \end{bmatrix}$$

4. $f(x'', y'') \rightarrow f(x'_{in}, y'_{in})$ Compute the intensity value of (x'', y'') using the interpolation technique.

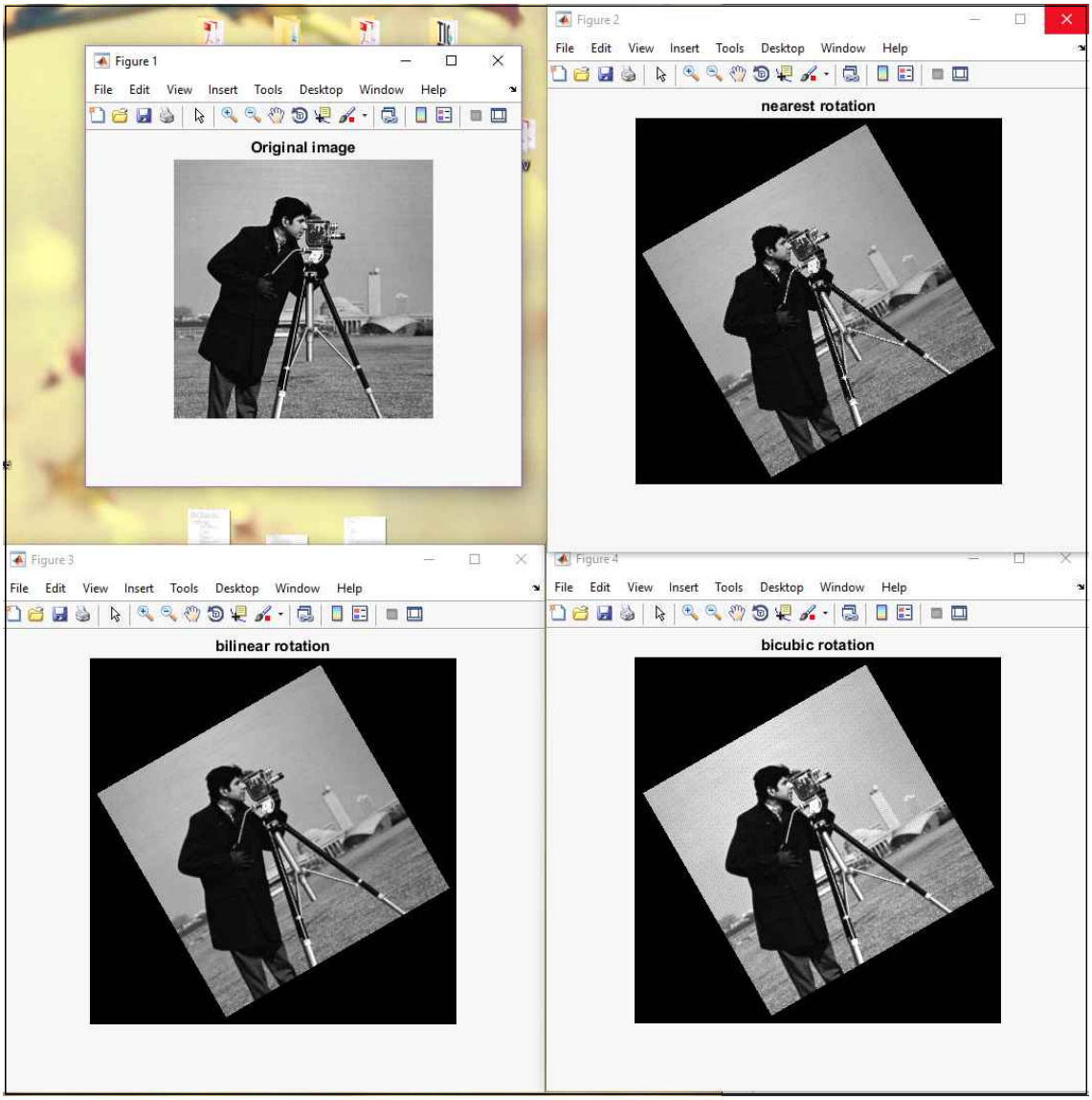
우선 회전을 먼저 한 다음, 기존에 존재하던 값이 변경된 floating point값을 얻는다. 그리고 각 값들에 대해 근처의 integer값을 구하고 이를 다시 처음의 각도로 되돌려서 실수 값들을 유지한 상태로 돌린다. 그러고 난 다음, interpolation을 적용하면 회전 시켰을 때의 이미지 pixel값들을 얻어 낼 수 있다.

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

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

* 위의 삼각함수를 보면 회전에 대한 공식에서 sin값에 붙는 음수 기호를 볼 수 있다. 그런데 ppt에 나온 함수를 보면 똑같은 값의 삼각함수를 사용하고 있어서 실제 코드를 구현할 때는 이론에 나와 있는 $\begin{pmatrix} \cos & \sin \\ -\sin & \cos \end{pmatrix}$을 활용하여 3번째 단계, 원래 회전상태로 돌리기를 구현하였다.

그리고 코드 상에서 구현 시에 이미지를 회전하게 되면 기존에 있던 픽셀 값들이 없어지면서 빈공간이 생기고 이미지의 전체 크기는 커지게 된다. 이를 위해 처음에 zero 매트릭스를 생성하고 그의 크기를 회전될 때의 크기에 맞추어서 생성한다.



각각의 결과를 보면 회전이 잘 이루어지고 있음을 알 수 있다.

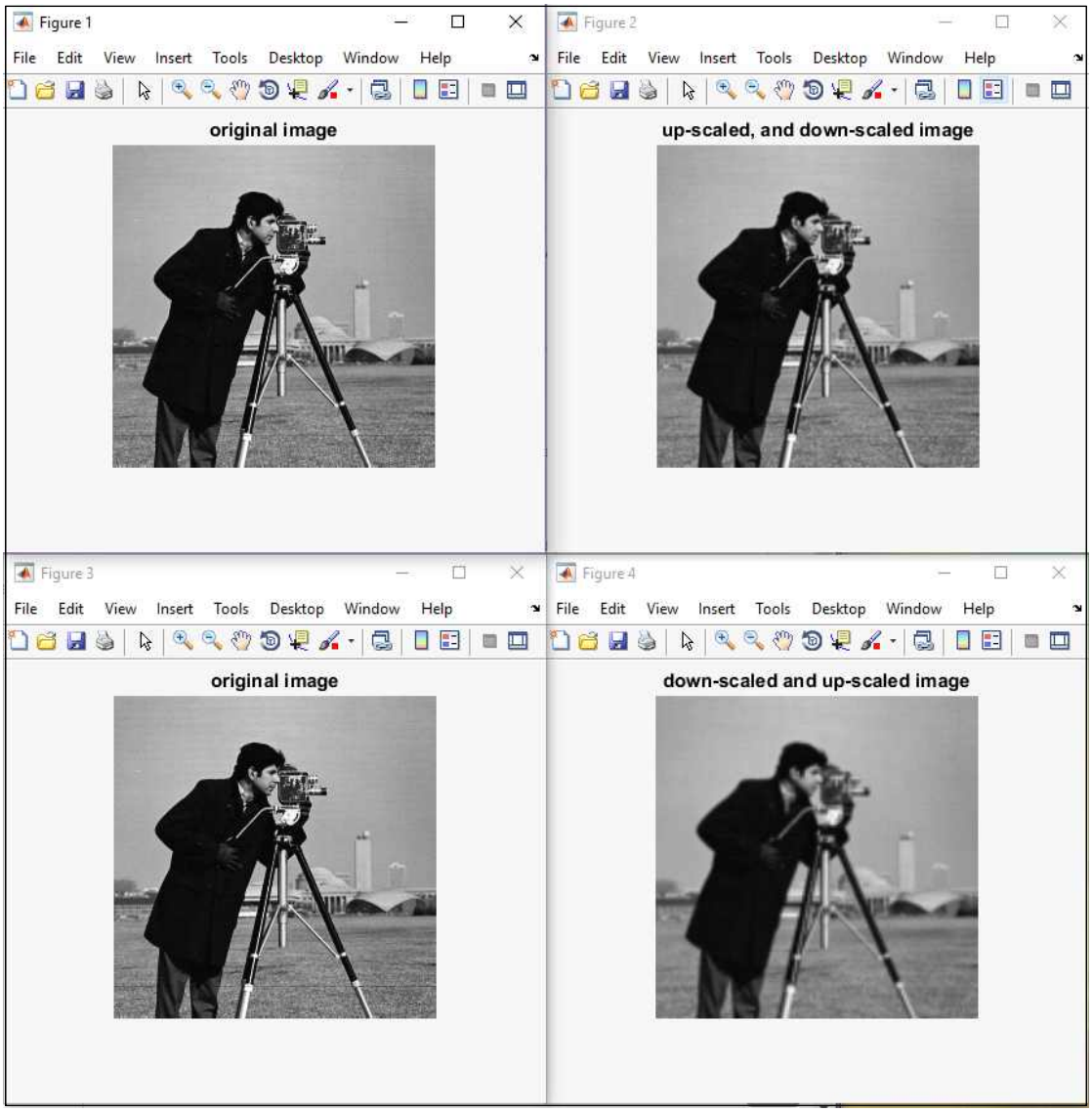
3. (MATLAB) Test exercise 5 and 6 using the codes that you implement in the homework 1. Then answer the questions together with very detailed reasons.

```
function hw_3()

im = imread('cameraman.tif');

im_first = up_sampling(im, 2.0, 'bilinear');
im_second = imresize(im_first, 0.5, 'bilinear');

im_third = imresize(im, 0.5, 'bilinear');
im_fourth = up_sampling(im_third, 2.0, 'bilinear');

figure, imshow(uint8(im)), title('original image');
figure, imshow(uint8(im_second)), title('up-scaled, and down-scaled image');

figure, imshow(uint8(im)), title('original image');
figure, imshow(uint8(im_fourth)), title('down-scaled and up-scaled image');

if im == im_second
    disp('first and second is same')
else
    disp('first and second is diff')
end

if im == im_fourth
    disp( 'third and fourth is same')
else
    disp('third and fourth is diff')
end
```

```
>> hw_3
first and second is diff
third and fourth is diff
>>
```

(함수의 실행 결과로 콘솔에 출력되는 값)

hw3는 cameraman.tif의 크기를 변경하는 과정에서 일어날 수 있는 현상을 확인해 볼 수 있는 코드이다. 위에서부터 각각 Exercies 5, 6에 대응하고 있다.



(결과 이미지)

Exercise 5. Suppose an image is enlarged by some amount k, and the result is decreased by the same amount. Should this result be exactly the same as the original? If not, why?

up_sampling() 함수를 이용해서 2배(factor값 2)로 이미지의 크기를 키운 뒤, imresize함수를 통해 이미지를 original 이미지와 같은 크기로 줄여보면 이미지가 흐려져 있긴 하지만 생각보다 픽셀 값들이 잘 유지되어 있음을 알 수 있다. 우선 이미지를 up sampling 하게 되면 기존에 존재하는 픽셀들의 값을 펼치게 되므로 이미지에 약간의 블록현상이 나타날 수 있다. 애초에 존재하는 위치의 픽셀들이 옆으로 퍼뜨려 지므로, 값을 채우는데 있어서, 값의 중복이나 손실 때문이다. 그런데 이렇게 키워놓은 이미지를 down sampling하게 되면 애초에 조금 손실 되었던 이미지의 픽셀 값들이 다시 interpolation 과정을 거치게 되기 때문에 이미 조금 퍼뜨려지고 손실되었던 픽셀 값들을 가지고

interpolation을 하게 되어 기존의 이미지보다 약간 손상된듯한 이미지가 결과로 얻어지는 것이다.

Exercise 6. What happens if the image is decreased first and the result enlarged?

이미지를 down sampling한 다음, up sampling하는 것은 좀 더 심각한 문제가 존재한다. 이미지를 down sampling하게 되면 그 과정에서 좁은 공간에 픽셀들을 우겨 넣게 되므로 버리게 되는 pixel의 수는 factor에 비례하여 증가하게 된다. Exercise 5에서는 픽셀들을 복제해서 값을 채웠기 때문에 기존의 픽셀들이 어느 정도 유지되는 면이 있었지만 Exercise 6에서는 픽셀들을 애초에 잃는 과정을 거치는 것이다. 이렇게 잃어버린 픽셀 값을 가지고 up sampling 연산을 하게 되면 이미 손실된 픽셀들을 가지고 퍼뜨리는 꼴이 되므로 많은 부분에서 픽셀의 단순 복제가 이루어지고 이로 인해 화질이 현저히 떨어지고 블록현상이 심해지는 것이다.