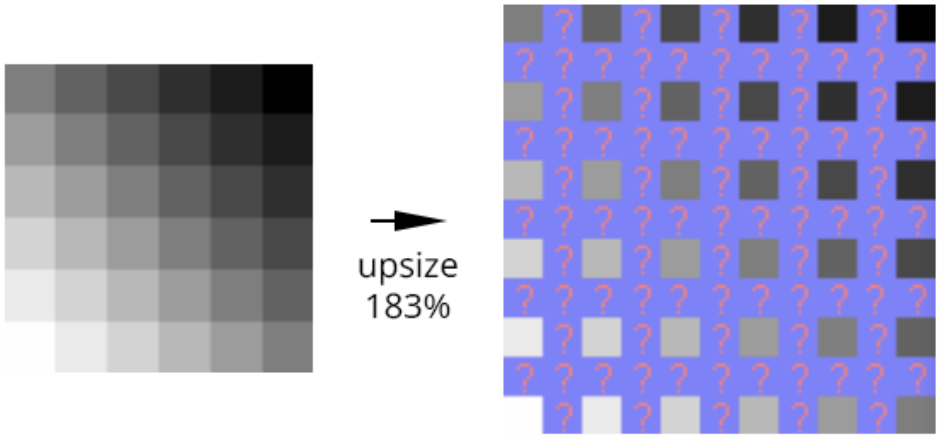
Image Interpolation

Interpolation is a method of constructing (finding) new data points based on the range of a discrete set of known data points. In image processing we are filling in the gaps in our pixel data when resize the image. Either adding pixels when the image is upsized or removing them when it is downsized. Information is always lost when a resizing occurs, interpolation tries to make the best guess to fill in the gaps from the lost data.

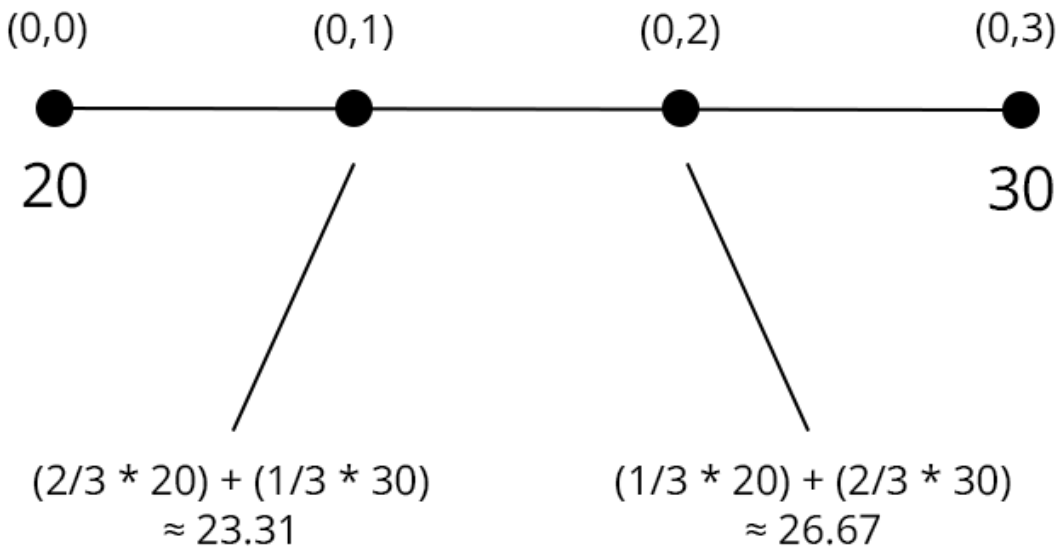


How do we fill in the missing values? Let's start with finding points along a line and build from there.

Linear Interpolation

We have already used interpolation with the HSV color space to set the values to the correct range. Remember: `np.interp()`

Simply, **Linear Interpolation** is a weighting of the known data points by their distance to the unknown data points.



In [ ]:
```python
# 1D Weighted Linear Interpolation
def weighted_linear_interpolation(dist, x0, x1):
    # dist is the distance from x1. x0 and x1 are known values.
    return round((1 - dist) * x0 + (dist * x1))

print(weighted_linear_interpolation(2/3, 20, 30))
```

```
27
```

More precisely, **Linear interpolation** is a method of using linear polynomials to construct new data points within the range of a discrete set of known data points.

Formula:

$y = y_1 + (((x - x_1) * (y_2 - y_1)) / (x_2 - x_1))$

So to find a point on the line. Given,

(x1,y1) = (1,4)

(x2,y2) = (6,9)

x = 5

From Linear Interpolation formula,

$y = y_1 + (((x - x_1) * (y_2 - y_1)) / (x_2 - x_1))$

y = 4+(((5-1) x (9-4))/(6-1))

= 4+((4×5)/5)

= 4+(20/5)

= 4+4

= 8

So new data point (x, y) is (5,8)

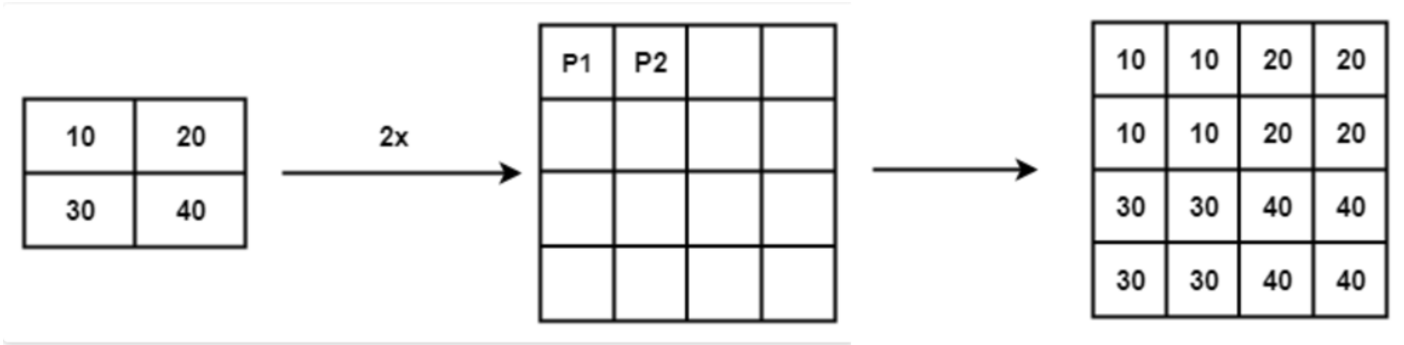
In [ ]:
```python
# # Linear Interpolation
# def Linear_Interpolation(x, x0, x1, y0, y1):
#     # x0, x1, y0, y1 are the known points
#     return round(y0 + (x - x0) * (y1 - y0) / (x1 - x0))

# print(Linear_Interpolation(5,1,6,4,9))

# print(round(np.interp(5,[1,6],[4,9])))
```

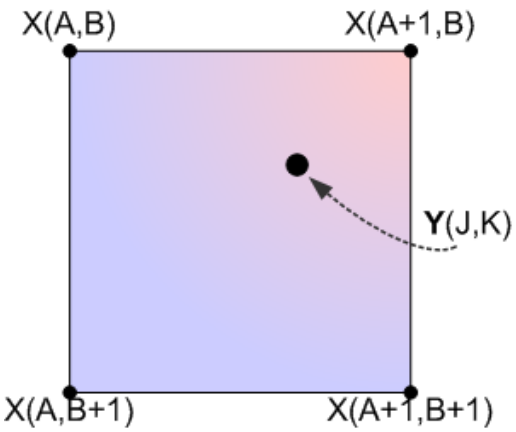
Let's go through three algorithms for resizing images. Each has it's advantages and disadvantages.

Nearest Neighbor

The nearest neighbor algorithm selects the value of the nearest point and does not consider the values of neighboring points at all. Simply, project the resized image back onto the original.

A very simple algorithm with a fast execution time widely used in real-time rendering. Nearest Neighbor produces sharp edges and is ideal for images with hard lines.





If (K-B < B+1-K)

    Pixel is one of the top two.

Else

    Pixel is one of the bottom two.

ENDIF

IF (J-A < A+1-J)

    Pixel is one of the left two.

Else

    Pixel is one of the right two.

ENDIF

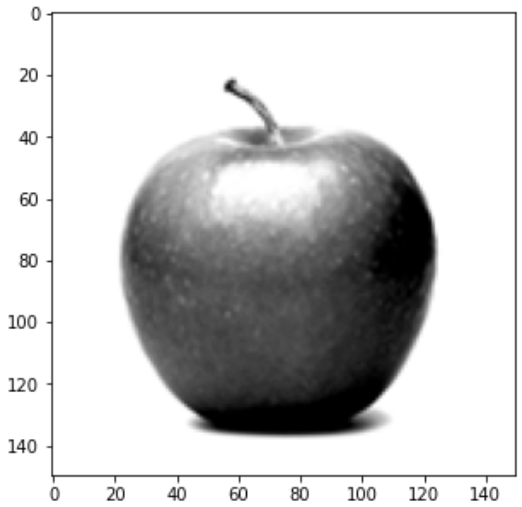
In [ ]:

```
# CODE FOR NN ALGO

import numpy as np
import cv2
import matplotlib.pyplot as plt

# NEarest Neighbor Algorithm
def NN_interpolation(img, scale_factor):
    # Get the dimensions of the image
    height = img.shape[0]
    width = img.shape[1]
    print(scale_factor)
    # Calculate the new dimensions of the image
    new_height = int(height * scale_factor)
    new_width = int(width * scale_factor)
    # Create a new image of the new dimensions
    new_img = np.zeros((new_height, new_width), dtype=np.uint8)
    # Calculate the scaling factor to find original pixel values.
    scale_x = width / new_width
```

```
    scale_y = height / new_height
    print(scale_x)
    # Loop through the new image and find the nearest neighbor in the original image
    for y in range(0,new_height):
        for x in range(0,new_width):
            # Calculate the coordinates of the nearest neighbor in the original image
            x0 = int(x * scale_x)
            y0 = int(y * scale_y)
            # Assign the pixel value to the new image
            new_img[y, x] = img[y0, x0]
    return new_img

#Saving the image in a variable
img = cv2.imread('Graphics/apple.png')

#OpenCV by defualt uses BGR instead of RGB.  Old digital camera standard.
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
print(img.shape)

# Set the size of the plot.
fig = plt.figure(figsize = (5,5))

#Use matplotlib while using a python notebook. Draw the image.
plt.imshow(img, cmap='gray')
```

```
(150, 150)
```

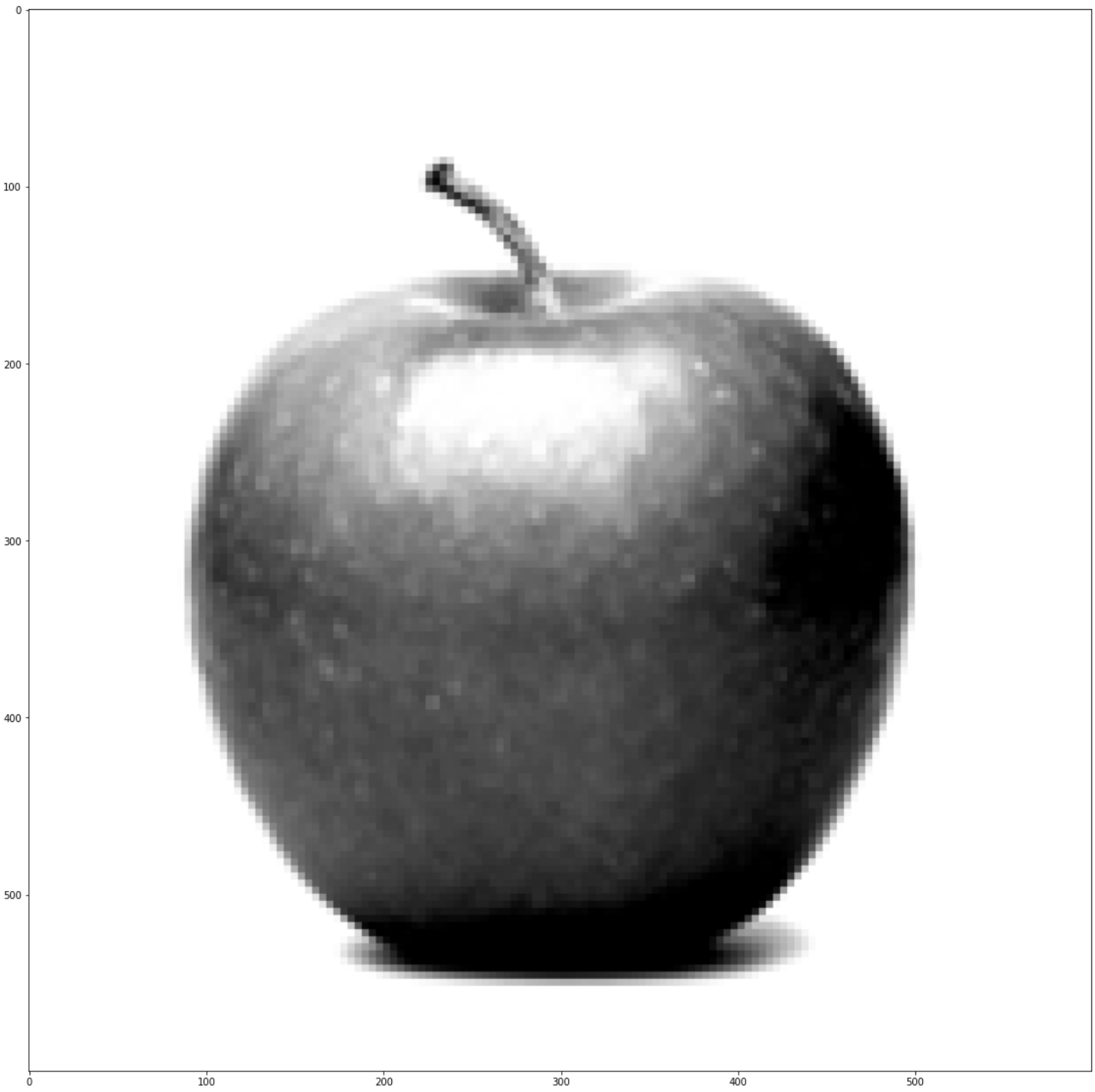
Out[ ]:
```
<matplotlib.image.AxesImage at 0x7fc479295220>
```



In [ ]:
```
img_NN = NN_interpolation(img, 4)
print(img_NN.shape)

# Set the size of the plot.
fig = plt.figure(figsize = (20,20))
#Use matplotlib while using a python notebook. Draw the image.
plt.imshow(img_NN, cmap='gray')
```

```
4
0.25
(600, 600)
```

Out[ ]:
```
<matplotlib.image.AxesImage at 0x7fc479306580>
```



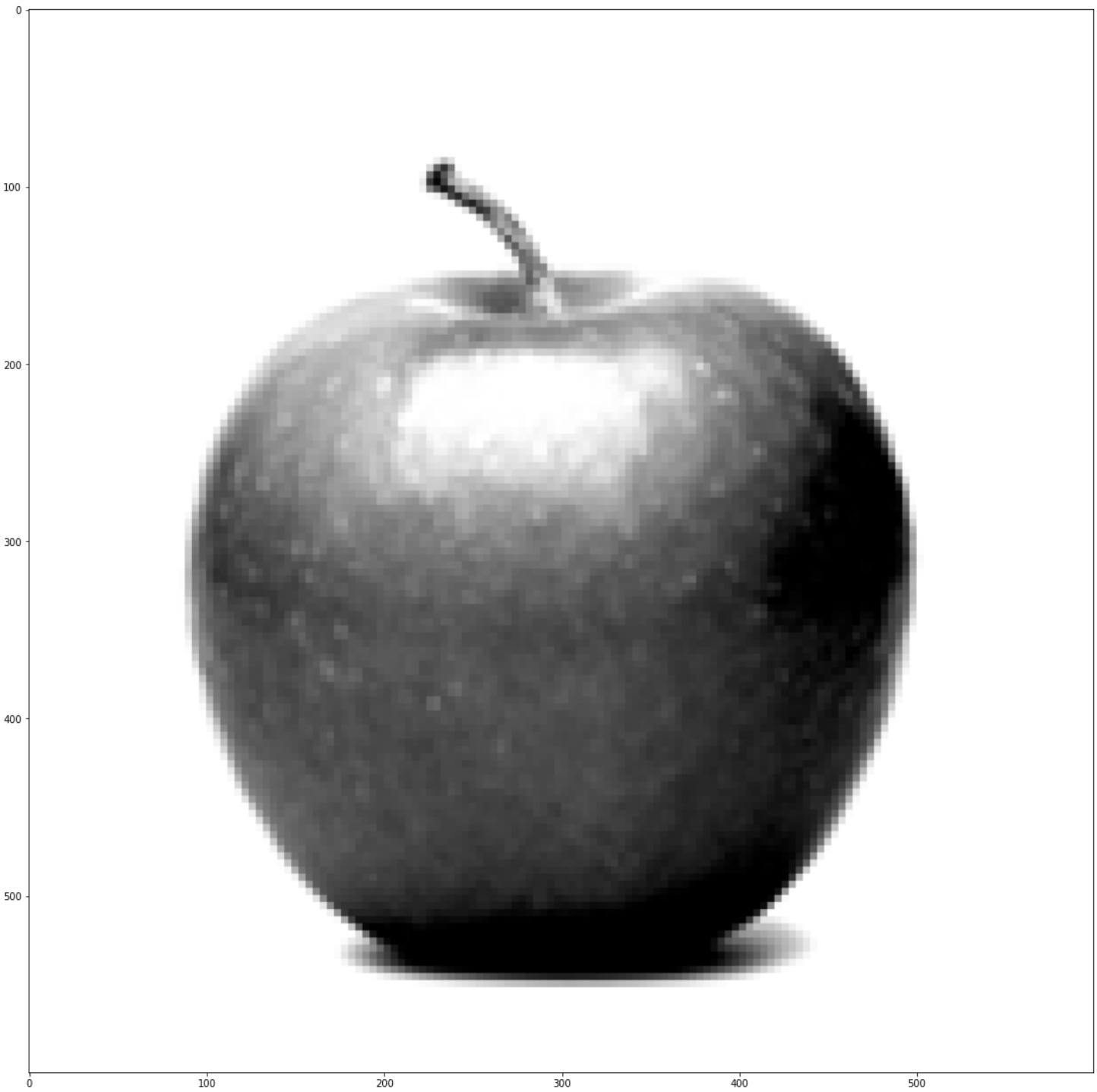
In [ ]:

```
oCv_img_NN = cv2.resize(img, None, fx = 4, fy = 4, interpolation=cv2.INTER_NEAREST)
print(oCv_img_NN.shape)

# Set the size of the plot.
fig = plt.figure(figsize = (20,20))
#Use matplotlib while using a python notebook. Draw the image.
plt.imshow(oCv_img_NN, cmap='gray')
```

```
(600, 600)
```

Out[ ]:

```
<matplotlib.image.AxesImage at 0x7fc47643feb0>
```



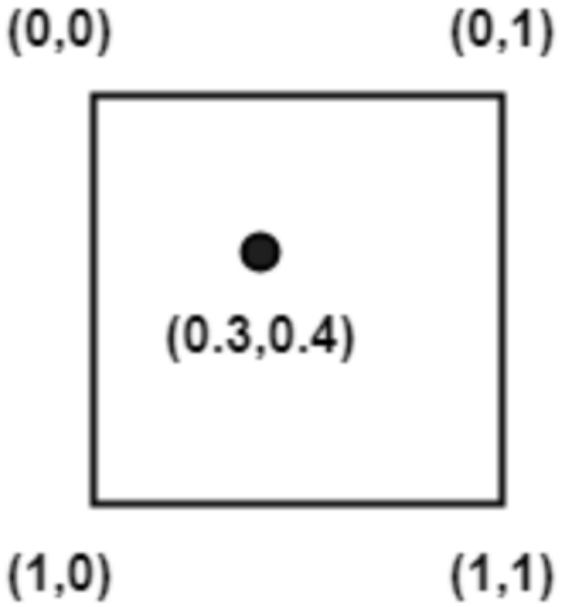
Bi-Linear Interpolation

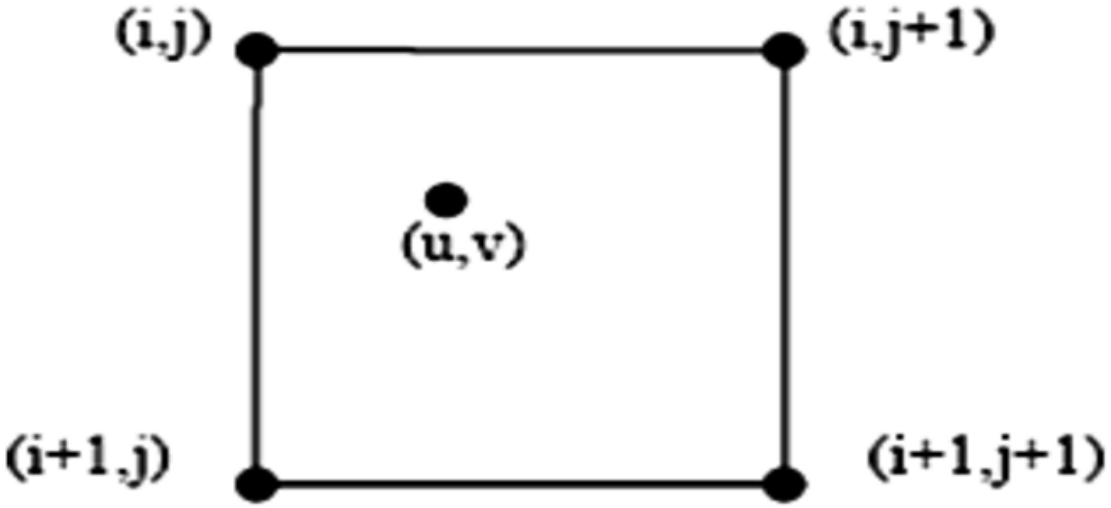
We already went over to to Linear Interpolation along a 1-D line but how do we accomplish the same interpolation for our 2-D image.

Instead of a set of two known data points we now have a set of four known data points.

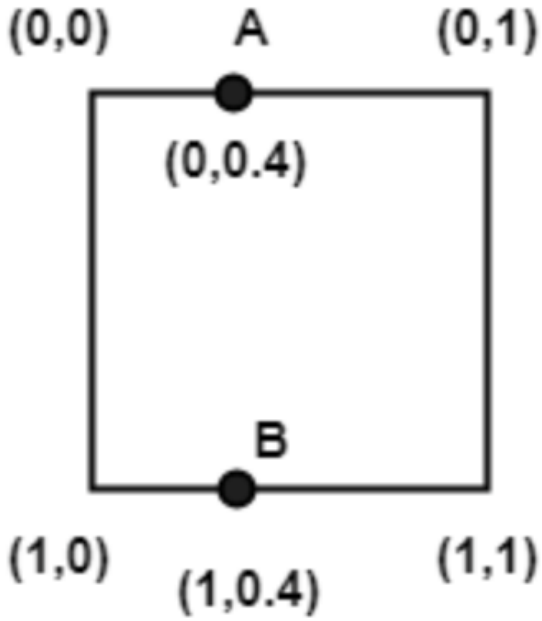
Given, 4 pixels (0,0), (1,0), (0,1), and (1,1).

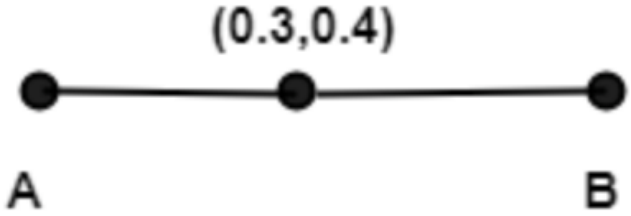
What is the value at (0.3,0.4)



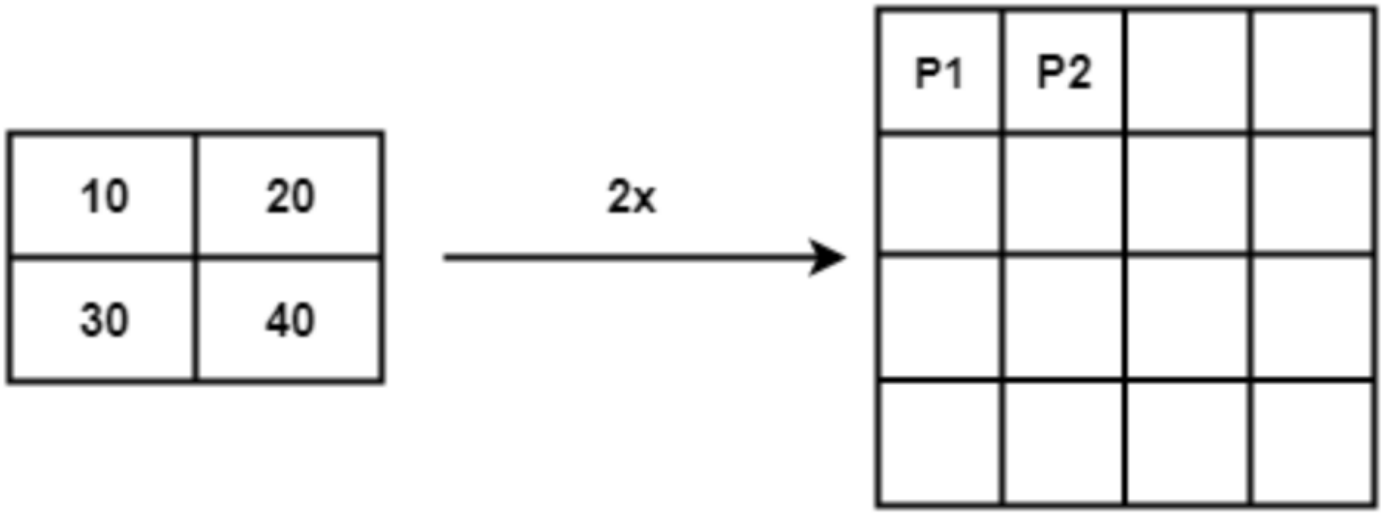


1. First, find the value along rows i.e at position A:(0,0.4) and B:(1,0.4) by linear interpolation.After getting the values at A and B, apply linear interpolation for point (0.3,0.4) between A and B and this is the final result.





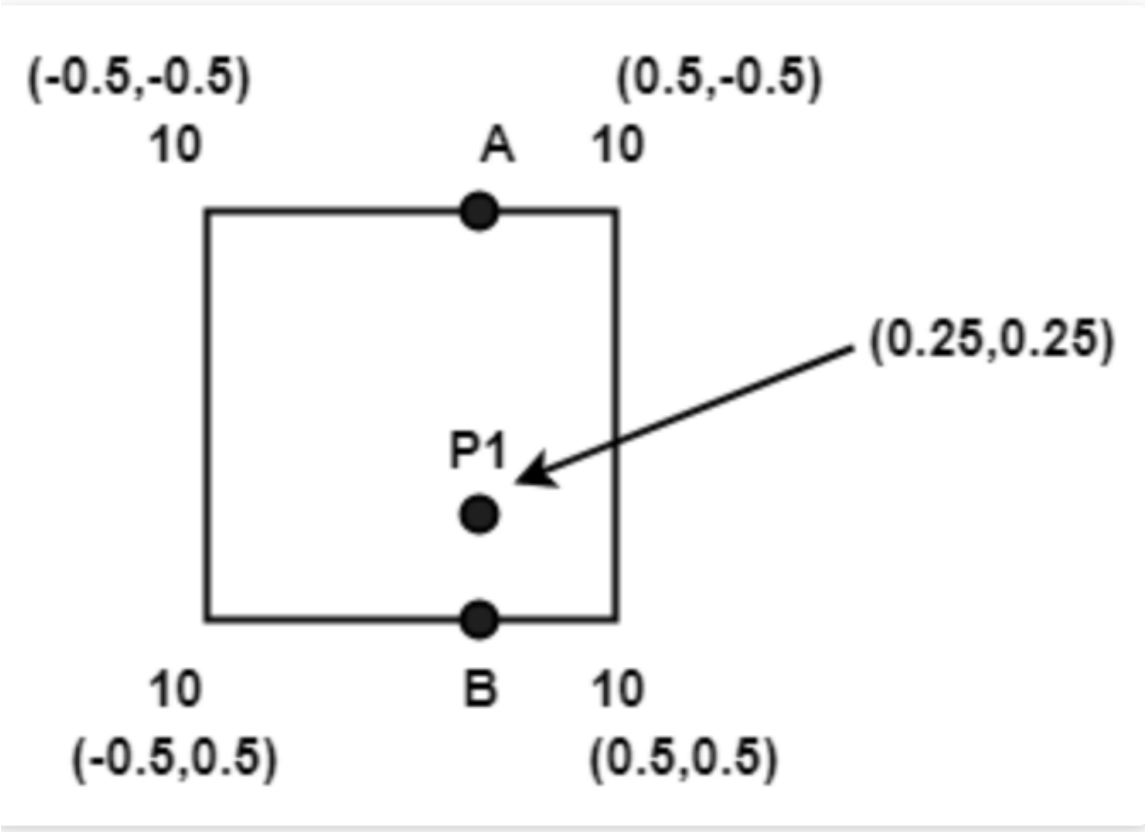
Let's use this process for a 2x2 image scaled to a 4x4 image.



3. Let’s take ‘P1’. First, we find the position of P1 in the input image. By projecting the 4×4 image on the input 2×2 image we get the coordinates of P1 as (0.25,0.25).
4. Since P1 is the border pixel and has no values to its left, so OpenCV replicates the border pixel. This means the row or column at the very edge of the original is replicated to the extra border(padding).
5. So, now our input image (after border replication) looks like this. Note the values in red shows the input image.

10	10	20	20
10	10	20	20
30	30	40	40
30	30	40	40

To find the value of P1, let's first visualize where P1 is in the input image (previous step image). Below figure shows the upper left 2×2 input image region and the location of P1 in that.



No we need to weights to calculate the value of P1. OpenCv uses the following formula to find the weights.

```
fx = (float)((dx+0.5)*scale_x - 0.5)
sx = cvFloor(fx)
fx -= sx
```

Where dx is the column index of the unknown pixel and fx is the weight that is assigned to the right pixel, 1-fx is given to the left pixel. Scale_x is the ratio of input width by output width. Similarly, for y, dy is the row index and scale_y is the ratio of heights now.

6. For P1, both row and column index i.e dx, and dy =0 so, fx = 0.75 and fy =0.75.

7. We apply linear interpolation with weights fx for both A and B(See Image-1) as 0.75*10(right) + 0.25*10 = 10 (Explained in the Algorithm above)
8. Now, for P1 apply linear interpolation between A and B with the weights fy as 0.75*10(B) +0.25*10(A) = 10
9. So, we get P1 = 10. Similarly, repeat for other pixels.

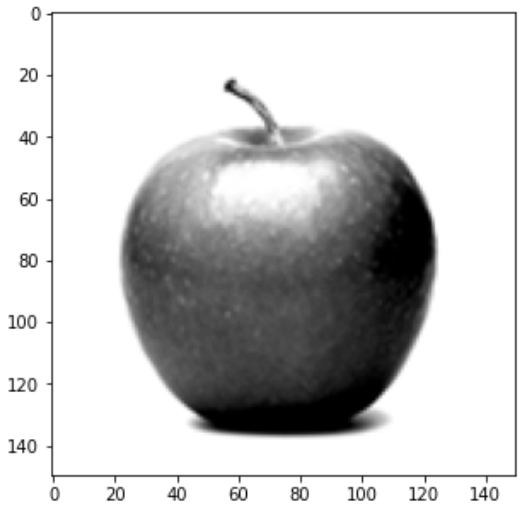
In [ ]:

```
# Bi-Linear Interpolation Algorithm
def bilinear_interpolation(img, scale_factor):
    # Get the dimensions of the image
    height, width = img.shape[:2]
    # Calculate the new dimensions of the image
    new_height = int(height * scale_factor)
    new_width = int(width * scale_factor)
    # Create a new image of the new dimensions
    new_img = np.zeros((new_height, new_width), dtype=np.uint8)
    # Calculate the scaling factor
    scale_x = width / new_width
    scale_y = height / new_height
    # Loop through the new image and find the nearest neighbor in the original image
    for y in range(new_height):
        for x in range(new_width):
            # Calculate the coordinates of the nearest neighbor in the original image
            x0 = int(x * scale_x)
            y0 = int(y * scale_y)
            # Calculate the coordinates of the next nearest neighbor in the original image.  Make sure it's
            x1 = min(x0 + 1, width - 1)
            y1 = min(y0 + 1, height - 1)
            # Calculate the distances to the nearest neighbors
            dx = (x * scale_x) - x0
            dy = (y * scale_y) - y0
            # Calculate the pixel values of the nearest neighbors
            q11 = img[y0, x0]
            q12 = img[y1, x0]
            q21 = img[y0, x1]
            q22 = img[y1, x1]
            # Calculate the weighted interpolated pixel value
            p11 = (1 - dx) * (1 - dy) * q11
            p12 = (1 - dx) * dy * q12
            p21 = dx * (1 - dy) * q21
            p22 = dx * dy * q22

            # Add the weighted values together and assign the pixel value to the new image
            new_img[y, x] = int(p11 + p12 + p21 + p22)
            # new_img[y, x] = int((1 - dx) * (1 - dy) * q11 + (1 - dx) * dy * q12 + dx * (1 - dy) * q21 + dx
    return new_img

# Linear Interpolation
def Linear_Interpolation(x, x0, x1, y0, y1):
    # x0, x1, y0, y1 are the known points
    return y0 + (x - x0) * (y1 - y0) / (x1 - x0)

#Saving the image in a variable
img = cv2.imread('Graphics/apple.png')

#OpenCV by defualt uses BGR instead of RGB.  Old digital camera standard.
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
print(img.shape)

# Set the size of the plot.
fig = plt.figure(figsize = (5,5))

# Draw the image.
plt.imshow(img, cmap='gray')
```

```
(150, 150)
```

Out[ ]:

```
<matplotlib.image.AxesImage at 0x7fc477616d90>
```



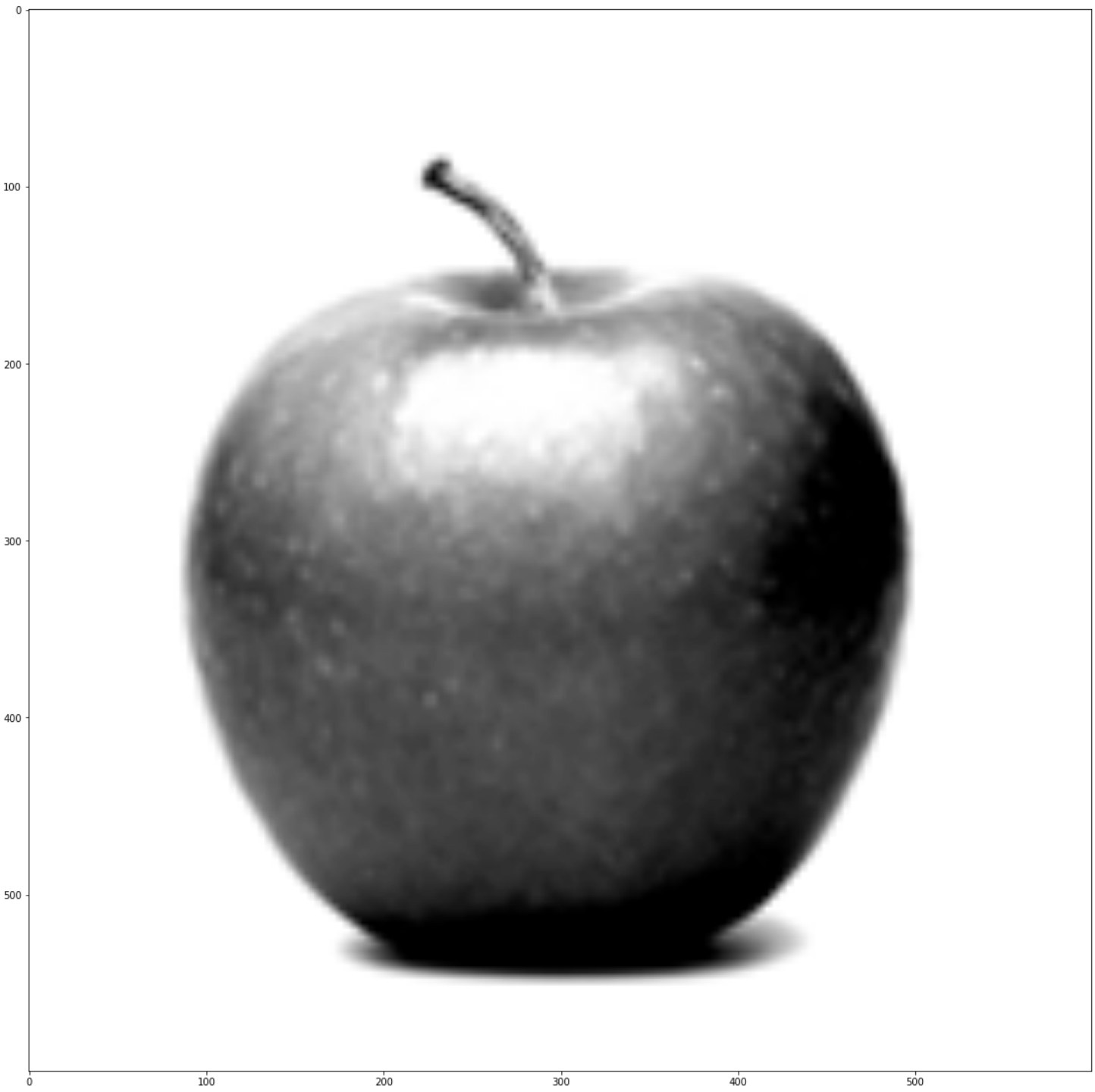
In [ ]:

```
img_BL = bilinear_interpolation(img, 4)
print(img_BL.shape)

# Set the size of the plot.
fig = plt.figure(figsize = (20,20))

plt.imshow(img_BL, cmap='gray')
```

```
(600, 600)
```

Out[ ]:

```
<matplotlib.image.AxesImage at 0x7fc45f1cb370>
```



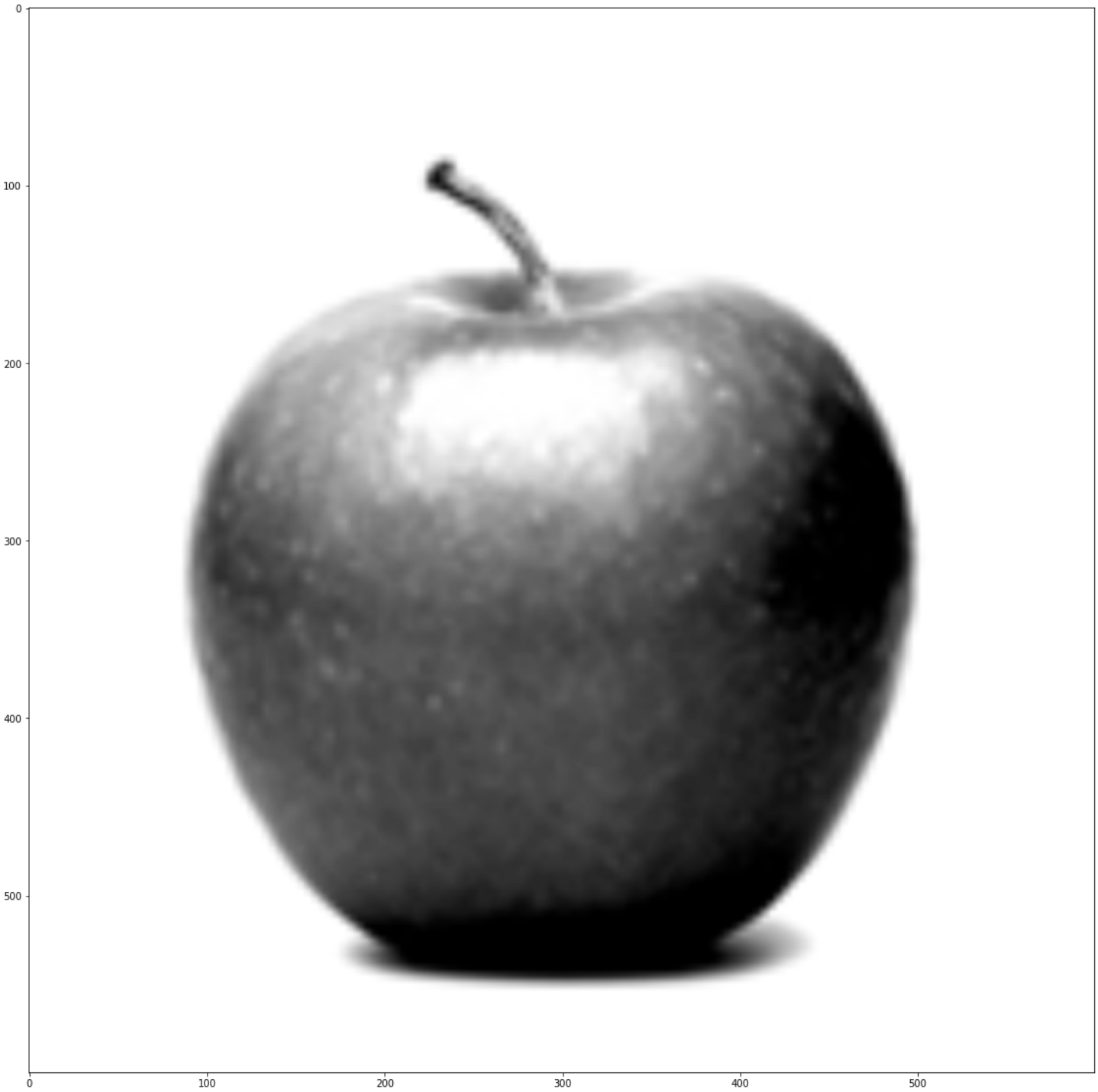
In [ ]:

```
oCv_img_BL = cv2.resize(img, None, fx = 4, fy = 4, interpolation=cv2.INTER_LINEAR)
print(oCv_img_BL.shape)

# Set the size of the plot.
fig = plt.figure(figsize = (20,20))
#Use matplotlib while using a python notebook. Draw the image.
plt.imshow(oCv_img_BL, cmap='gray')
```

```
(600, 600)
```

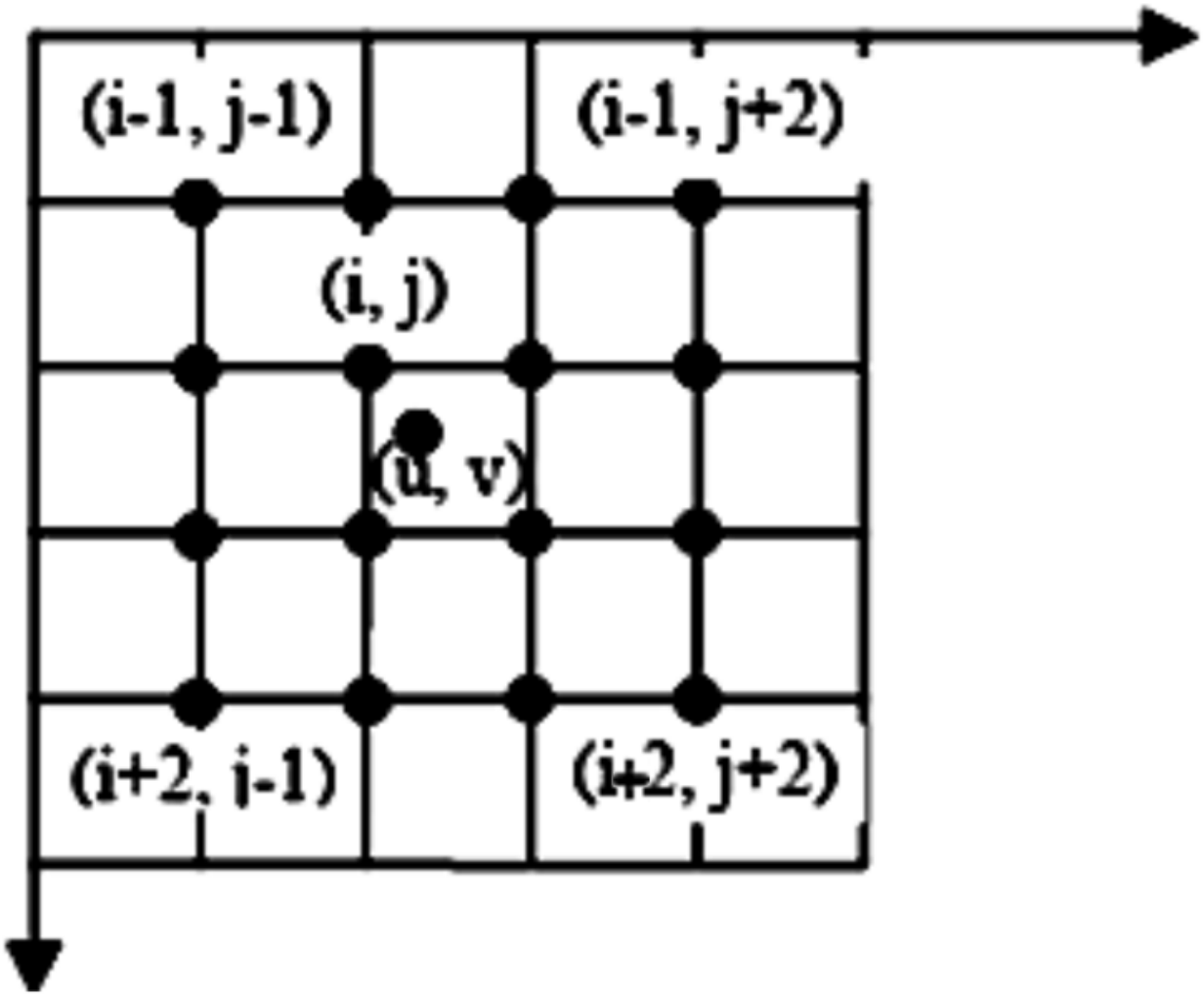
Out[ ]:

```
<matplotlib.image.AxesImage at 0x7fc477b887c0>
```



BiCubic

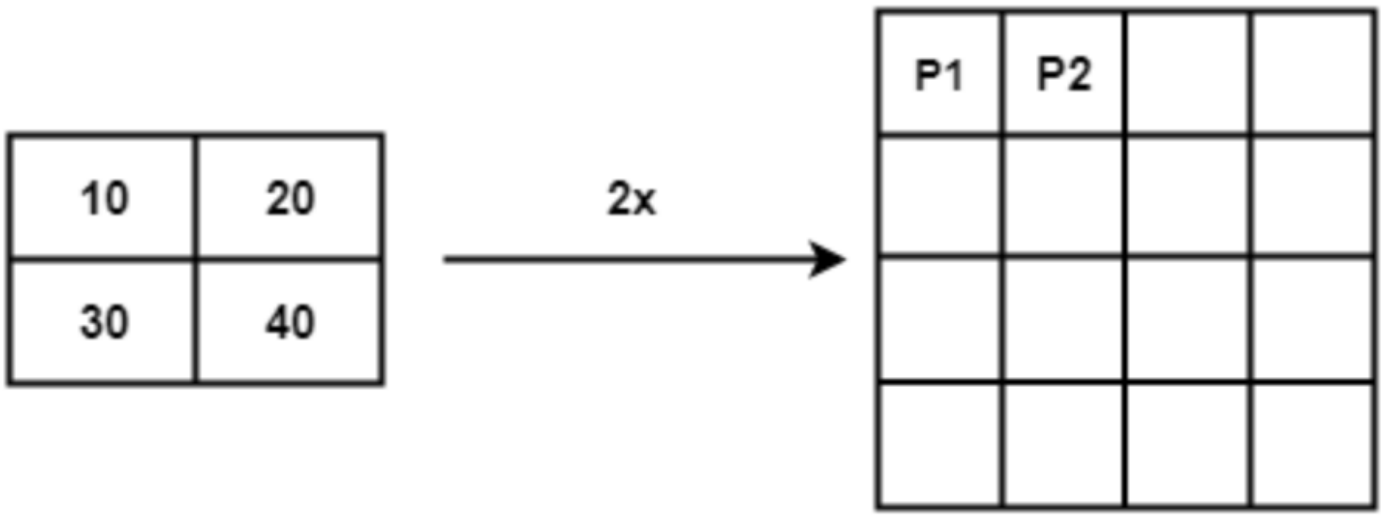
BiLinear Interpolation uses the 4 (2x2 frame) nearest neighbors to find the new pixel value. Bicubic interpolation uses the 16 nearest neighbors (4x4 frame).



x used in the above code is calculated from below code where x = fx

```
fx = (float)((dx+0.5)*scale_x - 0.5);
sx = cvFloor(fx);
fx -= sx;
```

Let's use the previous example:

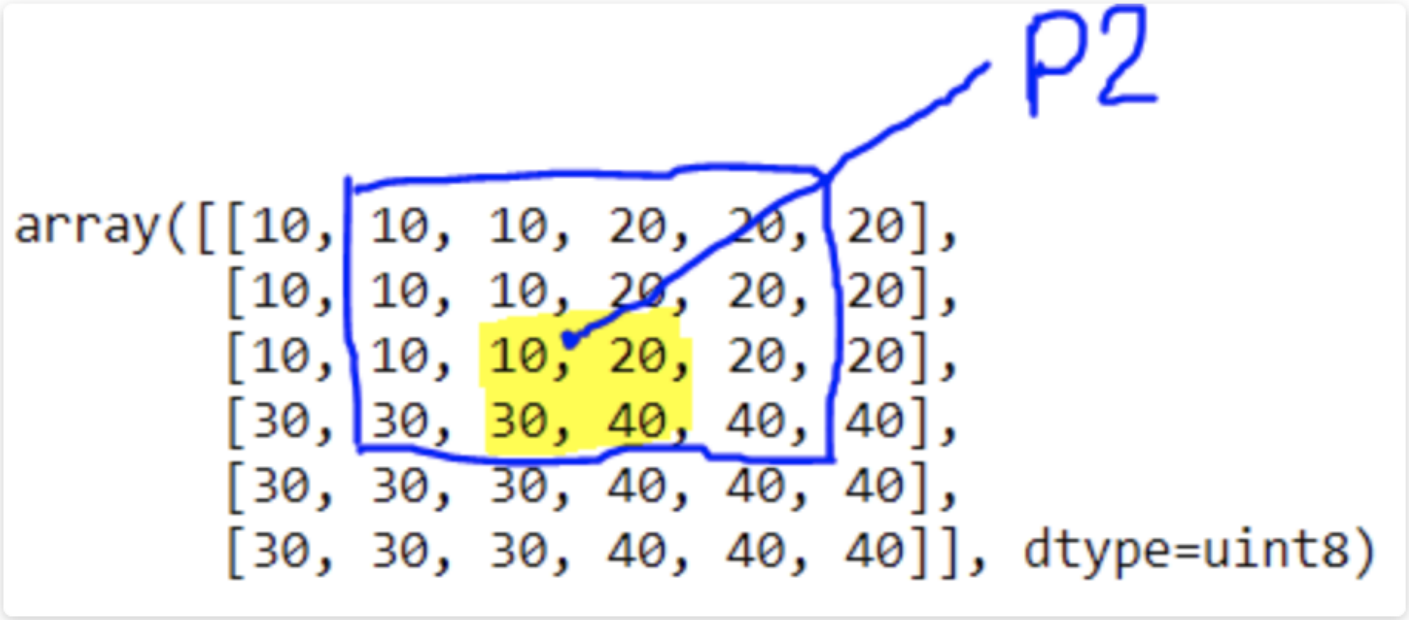


1. We previously calculated P1. This time let's take 'P2'. First, we find the position of P2 in the input image as we did before. So, we find P2 coordinate as (0.75,0.25) with dx = 1 and dy=0.
2. Because cubic needs 4 pixels (2 on left and 2 on right) so, we pad the input image. OpenCV has different methods to add borders which you can check here. Here, I used cv2.BORDER_REPLICATE method.

After padding the input image looks like this:

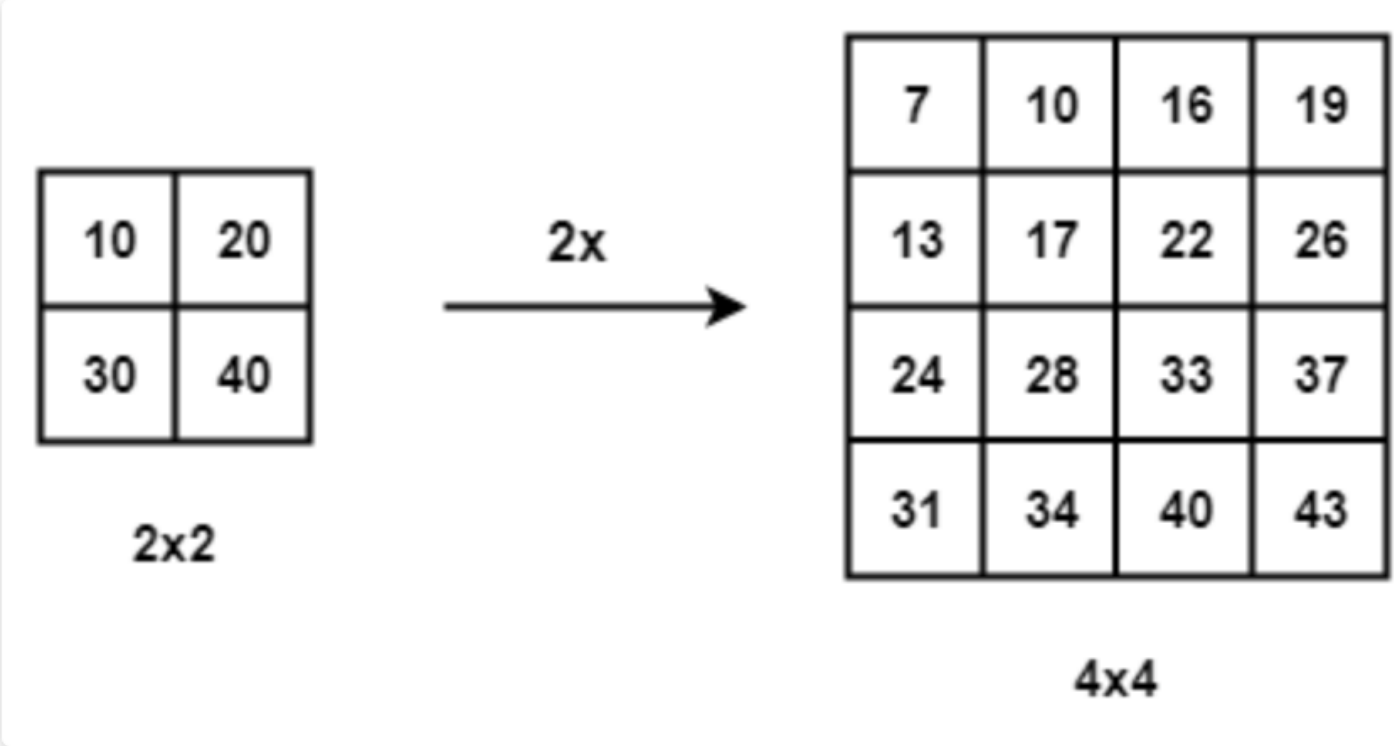
```
array([[10, 10, 10, 20, 20, 20],
       [10, 10, 10, 20, 20, 20],
       [10, 10, 10, 20, 20, 20],
       [30, 30, 30, 40, 40, 40],
       [30, 30, 30, 40, 40, 40],
       [30, 30, 30, 40, 40, 40]], dtype=uint8)
```

3. To find the value of P2, let's first visualize where P2 is in the image. Yellow is the input image before padding. We take the blue 4×4 neighborhood as shown below



4. For P2, using dx and dy we calculate fx and fy from code above. We get, fx=0.25 and fy=0.75
5. Now, we substitute fx and fy in the above code to calculate the four coefficients. Thus we get coefficients = [-0.0351, 0.2617,0.8789, -0.1055] for fy =0.75 and for fx=0.25 we get coefficients = [-0.1055 , 0.8789, 0.2617, -0.0351]
6. First, we will perform cubic interpolation along rows(as shown in the above figure inside blue box) with the above calculated weights for fx as -0.1055 *10 + 0.8789*10 + 0.2617*20 -0.0351*20 = 12.265625
   -0.1055 *10 + 0.8789*10 + 0.2617*20 -0.0351*20 = 12.265625
   -0.1055 *10 + 0.8789*10 + 0.2617*20 -0.0351*20 = 12.265625
   -0.1055 *30 + 0.8789*30 + 0.2617*40 -0.0351*40 = 32.265625
7. Now, using above calculated 4 values, we will interpolate along columns using calculated weights for fy as -0.0351*12.265 + 0.2617*12.265 + 0.8789*12.265 -0.1055*32.625 = 10.11702
8. Similarly, repeat for other pixels.

The final result:

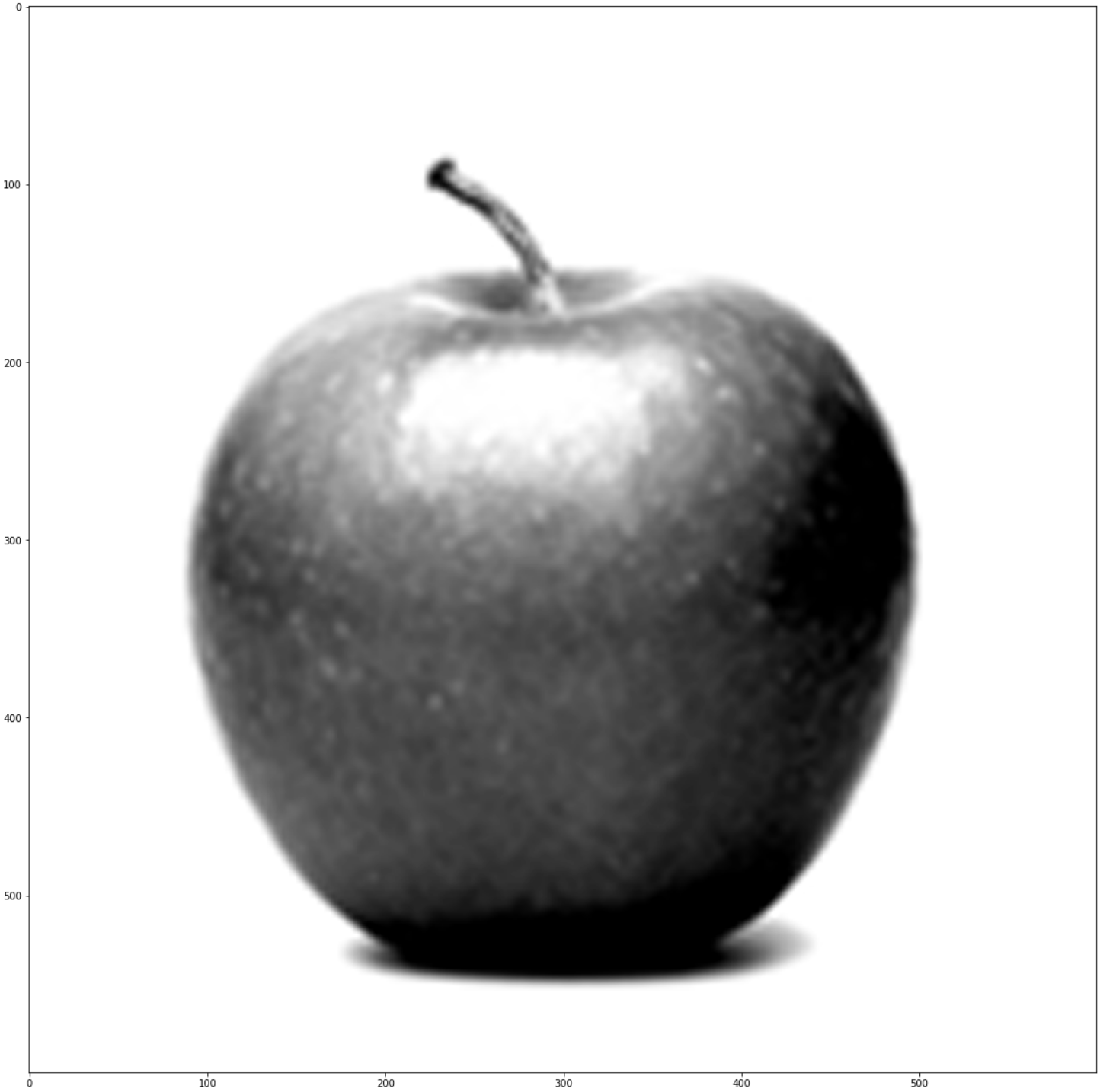


This produces noticeably sharper images than the previous two methods and balances processing time and output quality. That's why it is used widely

In [ ]:
```
oCv_img_NN = cv2.resize(img, None, fx = 4, fy = 4, interpolation=cv2.INTER_CUBIC)
print(oCv_img_NN.shape)

# Set the size of the plot.
fig = plt.figure(figsize = (20,20))
#Use matplotlib while using a python notebook. Draw the image.
plt.imshow(oCv_img_NN, cmap='gray')
```

```
(600, 600)
```

Out[ ]:
```
<matplotlib.image.AxesImage at 0x7fc45f21dbb0>
```


0
100
200
300
400
500
0
100
200
300
400
500