In this report I will discuss my implementation of the **bicubic interpolation** algorithm. I implemented it in MATLAB by developing several functions. I chose this style of coding as opposed to a single procedural file, because this modulatiry has allowed me to focus my desgin and debugging on different parts of the algorithm.

The general description of the algortihm is as follows:

1. Starting from the top left pixel of the image, a 4x4 matrix is filled with this pixel and its 15 nearest neighbours. This filling up is done in a specific way that ensures each pixel is in its correct spot in the matrix.
2. This 4x4 matrix is then passed into a function which is responsible for deriving the 16 coeffecients. This is done by employing the linear equation provided from lecture *(Bibcubic Interpolation)*.
3. These coeffecients are then passed into a function responsible for interpolating the 4 pixels. Three of these pixels will be new. One of them them however will simply be the pixel that our 4x4 window was formed around.
4. This function also takes in a pair of *x* and *y* values. These values are (0,0), (0,0.5), (0.5,0) and (0.5,0.5). These values are the arguments for resolving the interpolated pixels at specific points. This function then puts the interpolated pixels into a 2x2 matrix. Sidenote: the pixels are interpolated by having the function further pass the parameters into yet another function which is responsible for getting the actual interpolated pixel value. It does this by simply solving the double sum describing the bicubic interpolation.
5. Once this 2x2 matrix is filled, it is simply “stitched” with the preceding 2x2 matrix. This process is done in a loop, which repeats until a whole column of the new interpolated image is done.
6. Once a column is done, it is “stitched” with the preceding column. This process is also done in a loop (which is the outer loop of the loop mentioned in the step above).
7. This whole process repeats until the whole image has been interpolated.

The number of operations (multiplications, additions, and divisions) performed to interpolate a pixel is as follows:

1. The coeffecients relevant to the pixel are determined. This is a matrix multiplication done between three matrices. Each of these matrices are 4x4. Therefore there will be a total of 16x4 = 64 multiplications and 3x4x4 = 48 additions. Therefore total operations is 64+48 = **112**.
   NOTE: although this process is necessary in order to interpolate a pixel, it is *not needed* to repeat this process for every single pixel we want to interpolate. This process is done once for every 4 pixels.
2. Next, the bicubc double sum is solved to interpolate a pixel value. This solution includes 14 additions, and 40 multiplications. This is a total of 14+40 = **54**.

Therefore, the total number of operations to interpolate a pixel (assuming the coeffecients for that specific grouping hasn't been already found) is 112+54 = **166**.

However, if the coeffectients have already been found, the total number of operations if just **54**. This means that to find the hashed white dot in the diagram given, it takes the same amount of operations as the other two white dot. This is simply because they share the same coeffecients.

The complexity:

The algorithm comlexity is O(n). This is because the operations performed are scaled in a completely linear manner. For example, if image R has twice the amount of pixels as image Q, we expect image R to take twice as long to interpolate.

The following MATLAB test case was ran to confirm the complexity as O(n):

```
clear all;
clc;
s = magic(30);
s = uint8(s);
imwrite(s,'test.bmp')
im = imread('test.bmp');
[height width a] = size(im);
number_of_pixels = height*width
tic;
interpolateImage('test.bmp','test_output_1.bmp');
toc
im = imread('test_output_1.bmp');
[height width a] = size(im);
number_of_pixels = height*width
tic;
interpolateImage('test_output_1.bmp','test_output_2.bmp');
toc
im = imread('test_output_2.bmp');
[height width a] = size(im);
number_of_pixels = height*width
tic;
interpolateImage('test_output_2.bmp','test_output_3.bmp');
toc
im = imread('test_output_3.bmp');
[height width a] = size(im);
number_of_pixels = height*width
tic;
interpolateImage('test_output_3.bmp','test_output_4.bmp');
toc
im = imread('test_output_4.bmp');
[height width a] = size(im);
number_of_pixels = height*width
tic;
interpolateImage('test_output_4.bmp','test_output_5.bmp');
toc
```

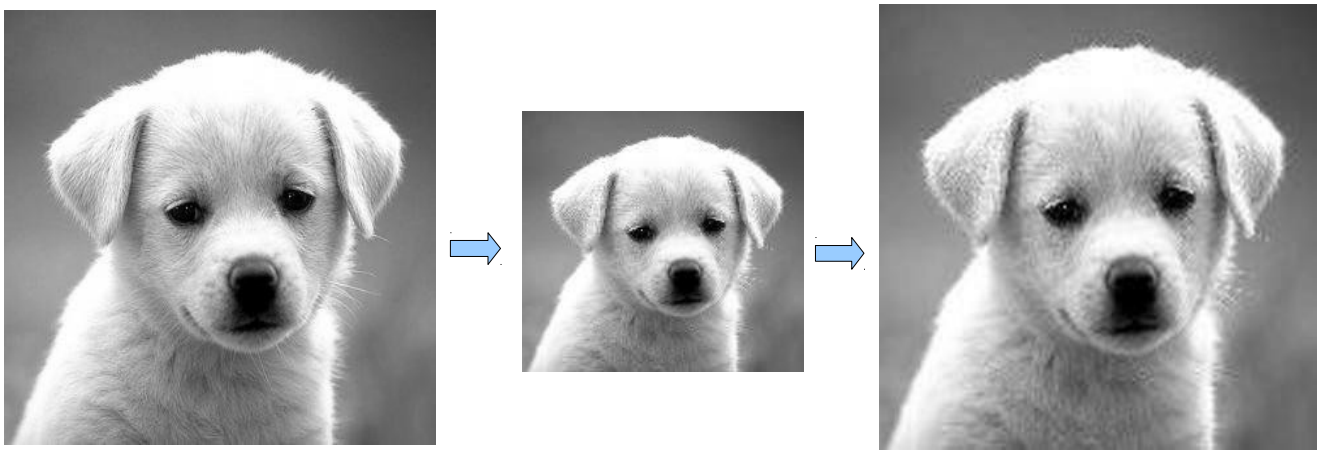
```
number_of_pixels =

   900

Elapsed time is 0.069757 seconds.

number_of_pixels =

        3600

Elapsed time is 0.083965 seconds.

number_of_pixels =

       14400

Elapsed time is 0.298142 seconds.

number_of_pixels =

       57600

Elapsed time is 1.191620 seconds.

number_of_pixels =

      230400

Elapsed time is 4.987507 seconds.
```

The test displays the size and amount of time taken for the image to be interpolated. For O(n) to be true, we expect each succesive interoplation to take approximately 4 times as long (because there will be 4 times as many pixels in the image). By dividing the time by the previous interpolation's time, we can get a ratio.

4.987507 / 1.191620 = **4.185**
1.191620 / 0.298142 = **3.997**
0.298142 / 0.083965 = **3.551**

The data clearly shows a trend of taking approximately 4 times as long for each successive interpolation. Therefore the complexity is indeed **O(n)**.

The following image was downsampled by a factor of 2 (dropping every other row and column). Then it was interpolated back to its original size (so by a factor of 2):

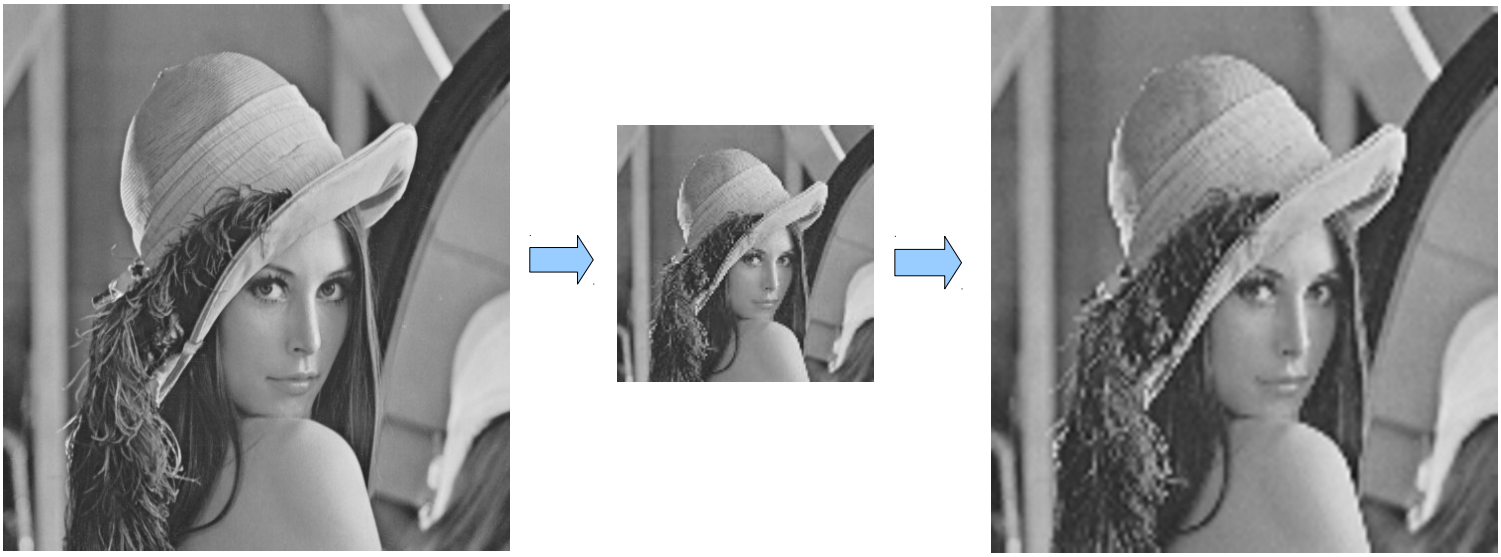


The **mean-squared error** of the two images is **0**.

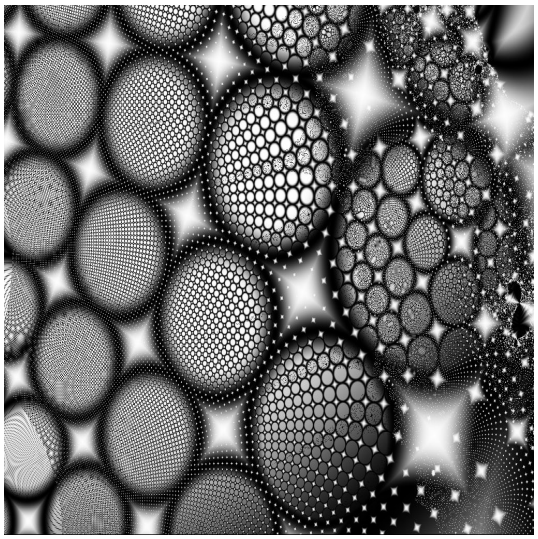
From looking at the image on the right, it can be seen that there is definitely some loss of data. This is manifested as graininess around the eyes of the puppy, and the detail loss can be especially noticed around the edges, looking at the fine hairs such as the whiskers. This can be explained by the fact that when we downsampled the image, we removed pixels, which permanently removes data. In other words, downsampling the image was a lossy operation, and interpolating it would simply be an attempt at predicting what those missing pixels weer. Of course, it is not possible to get those pixels back perfectly.

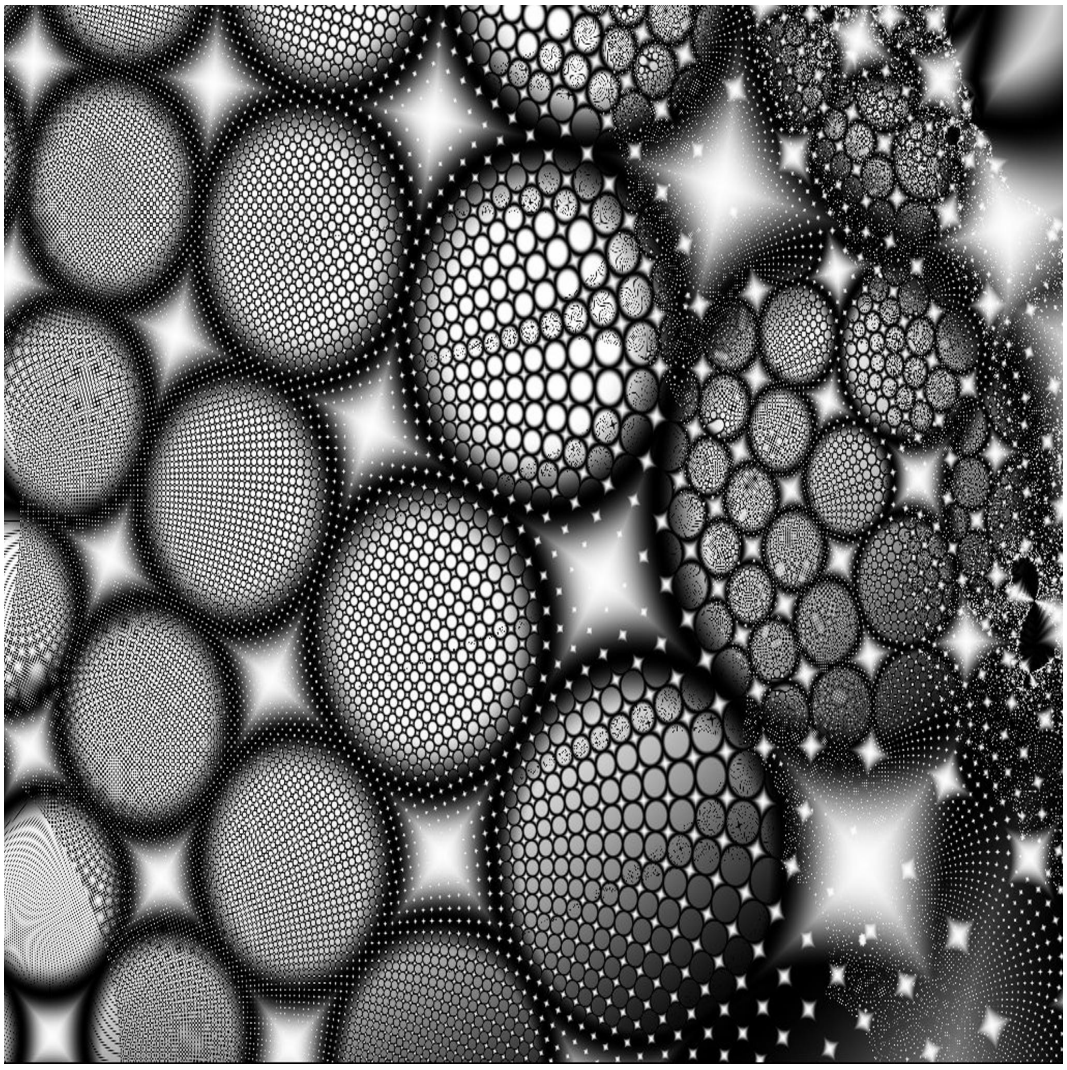
Here are some more examples of the algortihm on different images:

There is a slight difference, mainly that the original image appears to be sharper. Over all however this image faired very well after going through the downspampling and the interpolation.



The above is a demonstration of what happens when an image is downsampled by a factor of 2 **twice** and then interpolated with our algorithm twice, restoring it to its original size. As can be clearly seen, there is a significant loss of data





And finally, this is an application of our algorithm on an image to make it twice as large. The results are quite pleasing. It is almost impossible to spot a difference between the two images (other than the size). This shows that the algorithm performed as expected.