RMIT University

School of Engineering

EEET2248 – Electrical Engineering Analysis

Group Lectorial Task 3

Interpolation and Mean-Squared Error

Lecturer: Dr. Katrina Neville

Student Names: Nathan Williams, Jake Weiner, Brendan Vo, Thư Nguyễnn, James Howard

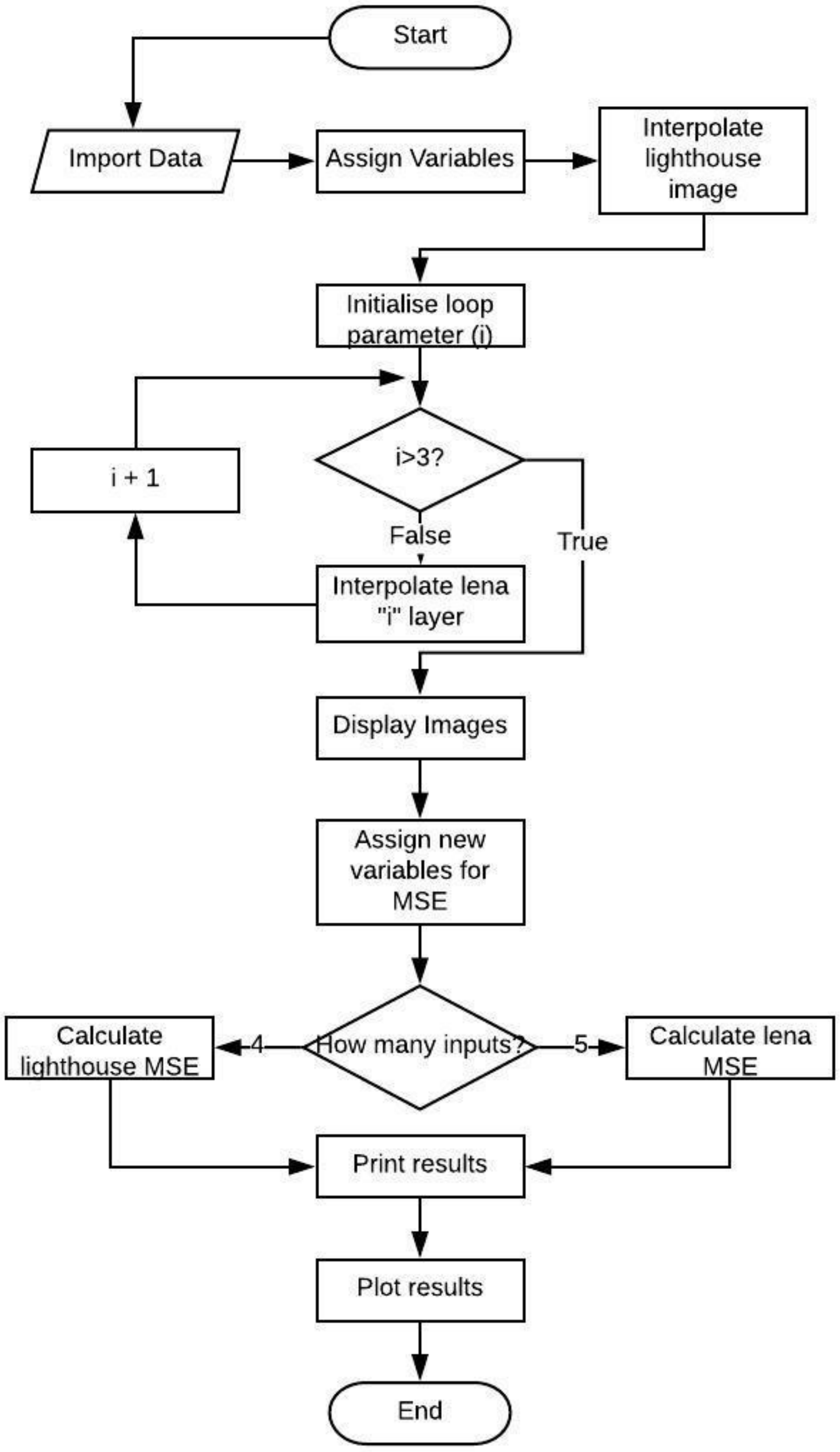
Student Number: s3707244, s3720634, s3717676, s3654756, s3287047

Submission Due Date: 1st June 2018

Introduction

The first set of group tasks involved the exploration of interpolation. Interpolation is the process by which a small set of data, such as an array or matrix, is expanded to a larger size. When this data is expanded, many blank spaces are created. By interpolation, these spaces are filled with estimated data derived from surrounding, existing data. In this task, we were provided two images, one grayscale and one color, in three different sizes. For the grayscale image, we were given a full-size reference image (512x512), an image half that size (256x256), and an image a quarter of the size of the original (128x128). The colored image was also provided in these sizes, however they contained a third dimension with a length of three. We were to expand these half and quarter-size images to the size of the reference image using three interpolation methods: linear, cubic-spline, and nearest-neighbor. Since interpolation uses informed estimation to derive unknown values, a level of error is to be expected. The second part of this task was to quantify this error using the mean-squared error method. Finding the mean-squared error involves comparing the new, interpolated image to the original, full resolution image to find the average variation, or “error,” among their data values. To perform this task, a standard equation was provided.

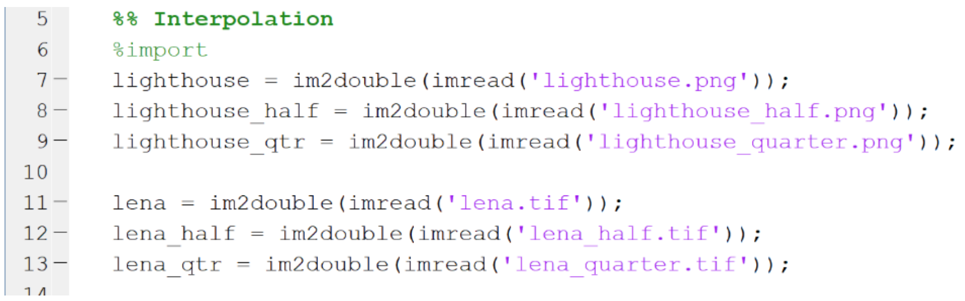
Algorithm Design



Interpolation Design

The first step in the interpolation process was to import the images, ‘lighthouse’ and ‘lena’, and their respective half and quarter-size variations as double-precision arrays. To do this, we used the “im2double” and “imread” functions. The full-size ‘lighthouse’ image, or reference image, was imported in the form of a 512x512 array, and its half and quarter-size versions were imported as 256x256 and 128x128 arrays respectively. The ‘lighthouse’ arrays are two-dimensional because they are greyscale image. The ‘lena’ images, however, are in color and, therefore, are imported as three-dimensional arrays; the full, half, and quarter-size ‘lena’ images were imported as 512x512x3, 256x256x3, and 128x128x3 arrays respectively. The third dimension in these arrays contains the images’ color data. The portion of code responsible for the importing of these images can be seen in figure 1.1. Our entire script can be found in appendix A-i.

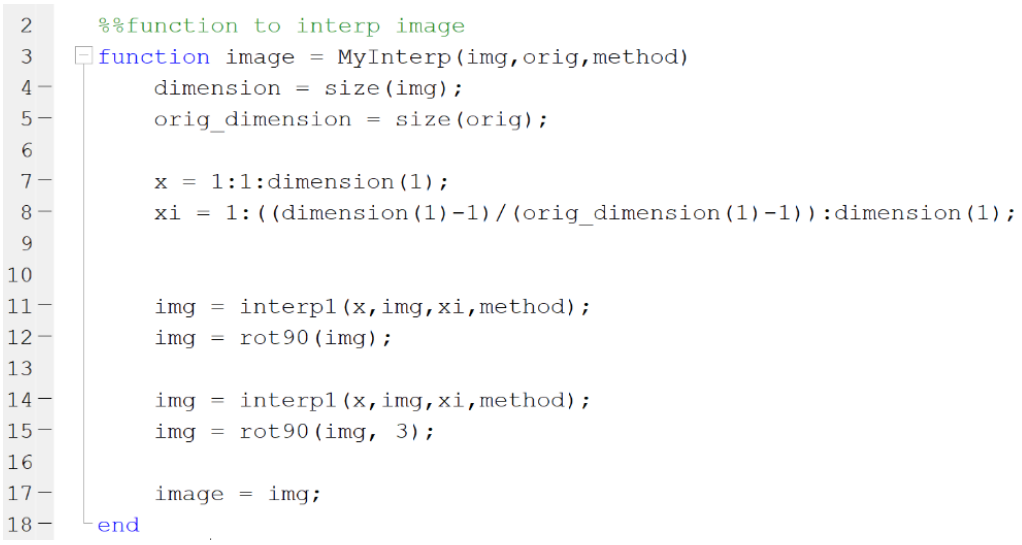
Figure 1.1: Import Code



```
%% Interpolation
%import
lighthouse = im2double(imread('lighthouse.png'));
lighthouse_half = im2double(imread('lighthouse_half.png'));
lighthouse_qtr = im2double(imread('lighthouse_quarter.png'));

lena = im2double(imread('lena.tif'));
lena_half = im2double(imread('lena_half.tif'));
lena_qtr = im2double(imread('lena_quarter.tif'));
```

Next, we created a user-defined function to interpolate the images using the linear, cubic-spline, and nearest-neighbor methods of interpolation. Our function used three arguments: ‘img’, ‘orig’, and ‘method’. ‘Img’ refers to the image we are interpolating, ‘orig’ refers to the reference image, and ‘method’ denotes the method of interpolation we are using. Next, we used the “size” function on the ‘img’ to extract the dimensions of the image. We repeated this process with ‘orig’ to extract the dimensions of the original image, which is the post-interpolation, or ‘goal’, size of the smaller image. Next, we created a one-dimensional array the length of the smaller image. This is our low-resolution data. Then, we created a one-dimensional array of high-resolution data the length of the goal size. To accomplish this, we divided the current dimensions of the smaller image by the dimensions of the reference image and used the result as the step-size for the high-resolution array. We had to subtract 1 from each array in order to ensure the new array would be the correct size. If we did not subtract 1, the resulting array would be one value short. Since the images are all square, we only had to do this for one dimension. Next, we interpolated the image about the x-axis using the “interp1” function in conjunction with the desired method, which is gained from the main script. Then, we rotated the image 90 degrees using the “rot90” function. With the image in this new position, what was previously its y-axis is now the x-axis. We interpolated the image a second time about its new x-axis. Finally, we rotated the image 90 degrees three more times, a total of 270 degrees, to orient it in its original position. Our user-defined function can be seen in figure 1.2 and found in appendix A-ii.

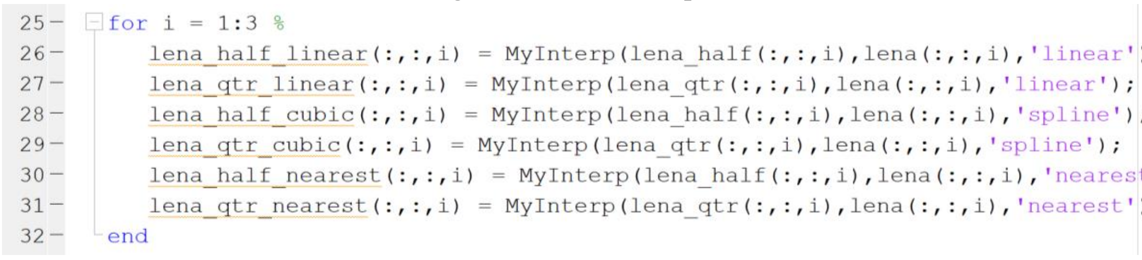
Figure 1.2: Interpolation Function



```
%%function to interp image
function image = MyInterp(img,orig,method)
    dimension = size(img);
    orig_dimension = size(orig);

    x = 1:1:dimension(1);
    xi = 1:((dimension(1)-1)/(orig_dimension(1)-1)):dimension(1);


    img = interp1(x,img,xi,method);
    img = rot90(img);

    img = interp1(x,img,xi,method);
    img = rot90(img, 3);

    image = img;
end
```

In order to successfully interpolate the images, we called in our user-defined function in the main script. For the grayscale ‘lighthouse’ images, this simply involved calling in the function with the variables corresponding to the small-sized and reference image as the first two arguments, and denoting the method as ‘linear’, ‘spline’, or ‘nearest’ as the third argument. However, the process was a bit more complicated for the colored ‘lena’ image. As mentioned earlier, colored images are imported into MATLAB in three-dimensional arrays. In this case, the ‘lena’ half-size image, is a 256x256x3 array. This composition can be interpreted as three layers of individual two-dimensional 256x256 arrays. In order to interpolate this image, we must interpolate each two-dimensional layer separately. To accomplish this, we included a “for” loop. This loop can be seen in figure 1.3.

Figure 1.3: For-Loop Code

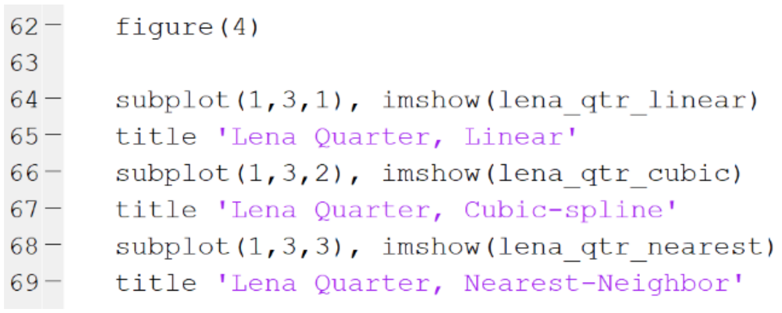


```
for i = 1:3 %
    lena_half_linear(:,:,i) = MyInterp(lena_half(:,:,i),lena(:,:,i),'linear');
    lena_qtr_linear(:,:,i) = MyInterp(lena_qtr(:,:,i),lena(:,:,i),'linear');
    lena_half_cubic(:,:,i) = MyInterp(lena_half(:,:,i),lena(:,:,i),'spline');
    lena_qtr_cubic(:,:,i) = MyInterp(lena_qtr(:,:,i),lena(:,:,i),'spline');
    lena_half_nearest(:,:,i) = MyInterp(lena_half(:,:,i),lena(:,:,i),'nearest');
    lena_qtr_nearest(:,:,i) = MyInterp(lena_qtr(:,:,i),lena(:,:,i),'nearest');
end
```

In this loop, the variable ‘i’ corresponds with the layer number, which is placed in the z-axis position of each ‘lena’ image variable. Then, we use a for-loop to ensure that our interpolation function is run for each of the three layers.

The last step in the interpolation process was to display the results. Since there are twelve images to be displayed, it could be hectic to display them individually. Instead, since there were four images being interpolated, we organized the images into 4 figures. Each figure contains three subplots showing the three methods of interpolation, which is ideal for comparison purposes. A portion of the code responsible for displaying our results can be seen in figure 1.4.

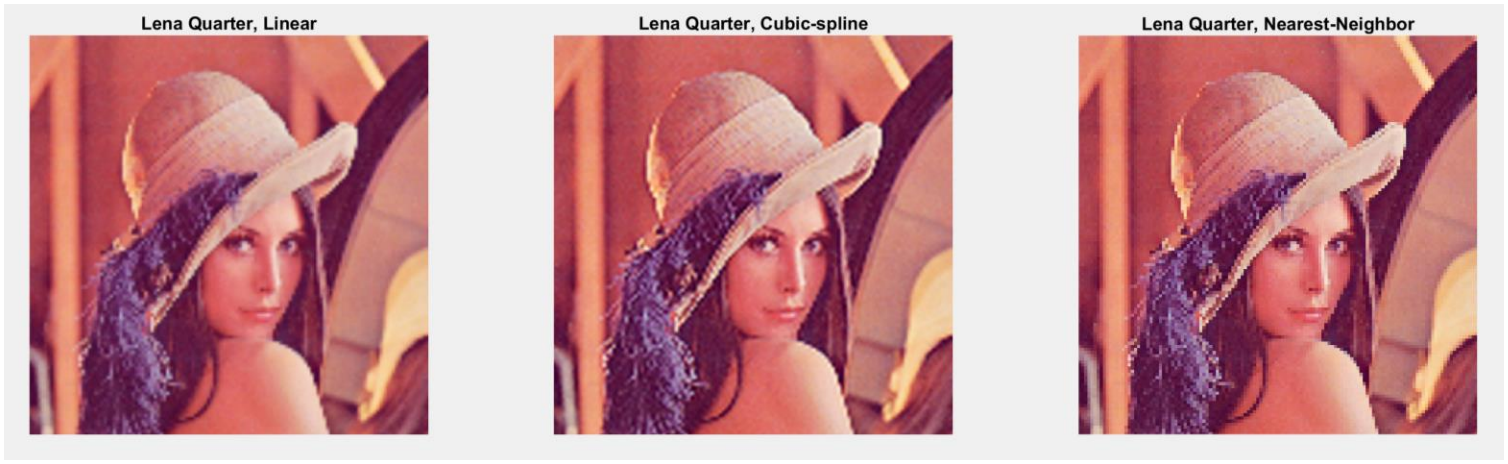
Figure 1.4: Result code



```
figure(4)

subplot(1,3,1), imshow(lena_qtr_linear)
title 'Lena Quarter, Linear'
subplot(1,3,2), imshow(lena_qtr_cubic)
title 'Lena Quarter, Cubic-spline'
subplot(1,3,3), imshow(lena_qtr_nearest)
title 'Lena Quarter, Nearest-Neighbor'
```

We used the “subplot” function to create the layout for the figure. In this case, we have 1 row with 3 columns to display the three images side-by-side. Then, for each image we denoted its position from 1 to 3. To display this image, we used the “imshow” function. Finally, we used the “title” function to clearly label each image and its respective interpolation method. This code’s output can be seen in figure 1.5.

Figure 1.5: Results

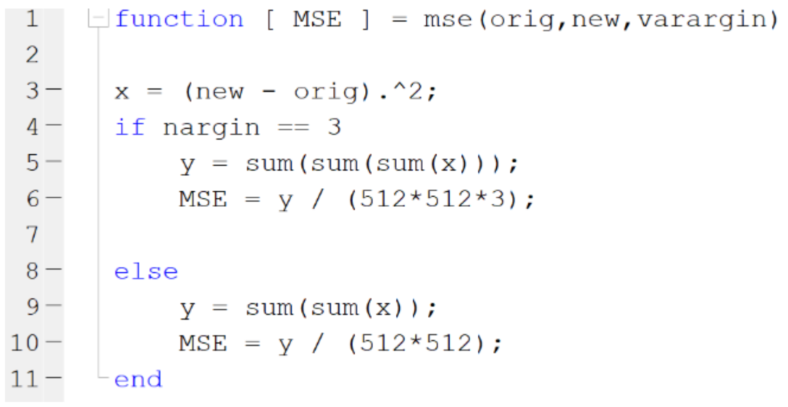


There are some differences to be noticed among the various interpolation methods. Linear interpolation appears to be the most similar to the reference image, however it is blurry in some spots, and has a slightly muted color. Cubic-spline interpolation appears to be more sharply rendered and more color-saturated than the linear method, however there is noticeable pixilation. The nearest-neighbor method is the least similar to the reference image, as the pixilation is very evident, and there is a dramatic aliasing effect. The variations among interpolation methods is far more apparent in the quarter-size images as opposed to the half-size images because much more data is being estimated, and, therefore, there is more room for error.

Mean-Squared Error Design

Once the interpolated data was created, we could calculate the mean-squared error (MSE) among the estimated values using the appropriate equation. The function we designed to run this calculation can be seen in figure 2.1 and found in appendix A-iii.

Figure 2.1: MSE Function



```
function [ MSE ] = mse(orig,new,varargin)

x = (new - orig).^2;
if nargin == 3
    y = sum(sum(sum(x)));
    MSE = y / (512*512*3);

else
    y = sum(sum(x));
    MSE = y / (512*512);
end
```

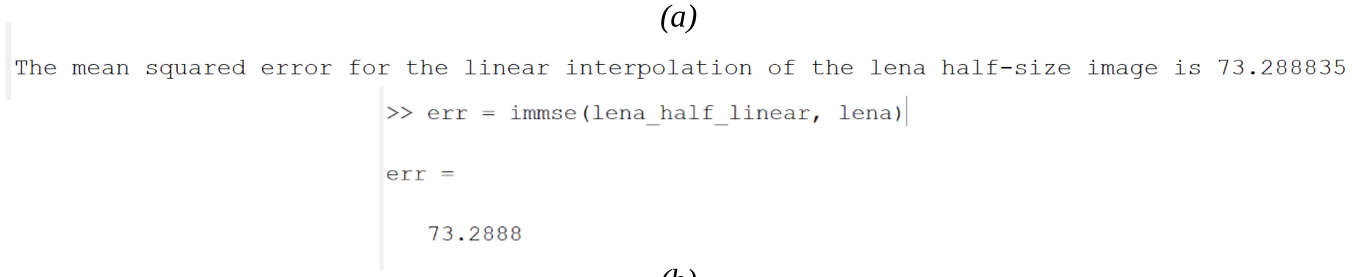
Since the ‘lena’ image is three-dimensional, we must use the “sum” function thrice, whereas we only need to use it twice with the two-dimensional ‘lighthouse’ image. We differentiate between the two processes by listing “varargin” as an argument so we can use the “nargin” function to select the appropriate calculation. When calling this function into the main script to calculate the ‘lighthouse’ MSE, we simply list the ‘orig’ and ‘new’ as the arguments. When calling in the function for the ‘lena’ image, however, we list a third argument in the form of an arbitrary string to ensure that the “if nargin == 3” condition is fulfilled.

Solution and Testing

In order to test the accuracy of our MSE results, we used MATLAB’s inbuilt MSE-calculating function, “immse”. We compared the results using the “immse” function with the results obtained using our user-defined function to determine that were producing accurate data. It is worth noting that using the “immse” function may be a quicker way of completing this task. However, creating this function by hand allows us to develop a better understanding of the mathematical process. In order to use a function properly, it is best to understand how it operates. A sample of this testing can be seen in figure 2.2

Figure 2.2: MSE Testing

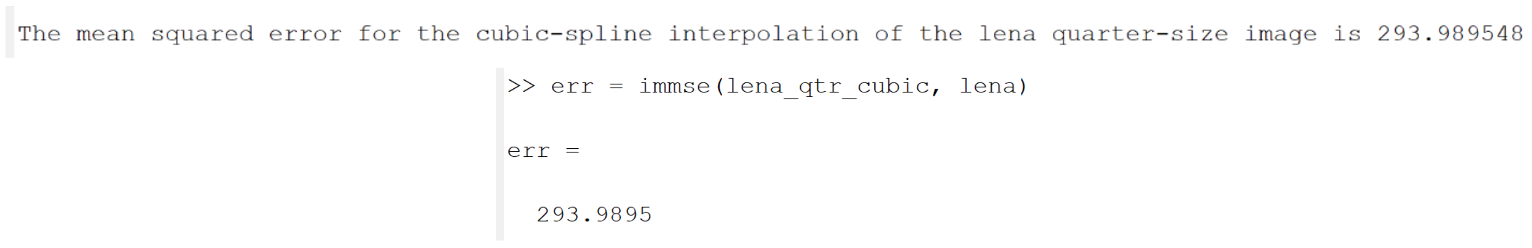
(a)



```
The mean squared error for the linear interpolation of the lena half-size image is 73.288835
```

```
>> err = immse(lena_half_linear, lena)

err =

   73.2888
```

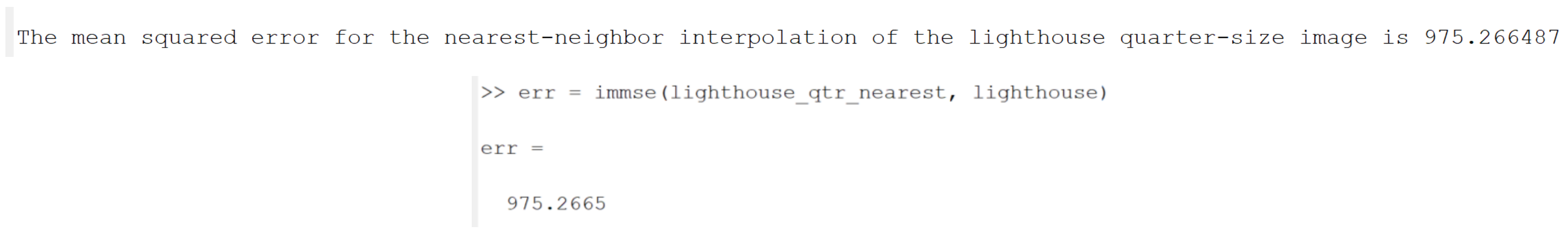
(b)



```
The mean squared error for the cubic-spline interpolation of the lena quarter-size image is 293.989548
```

```
>> err = immse(lena_qtr_cubic, lena)

err =

  293.9895
```

(c)

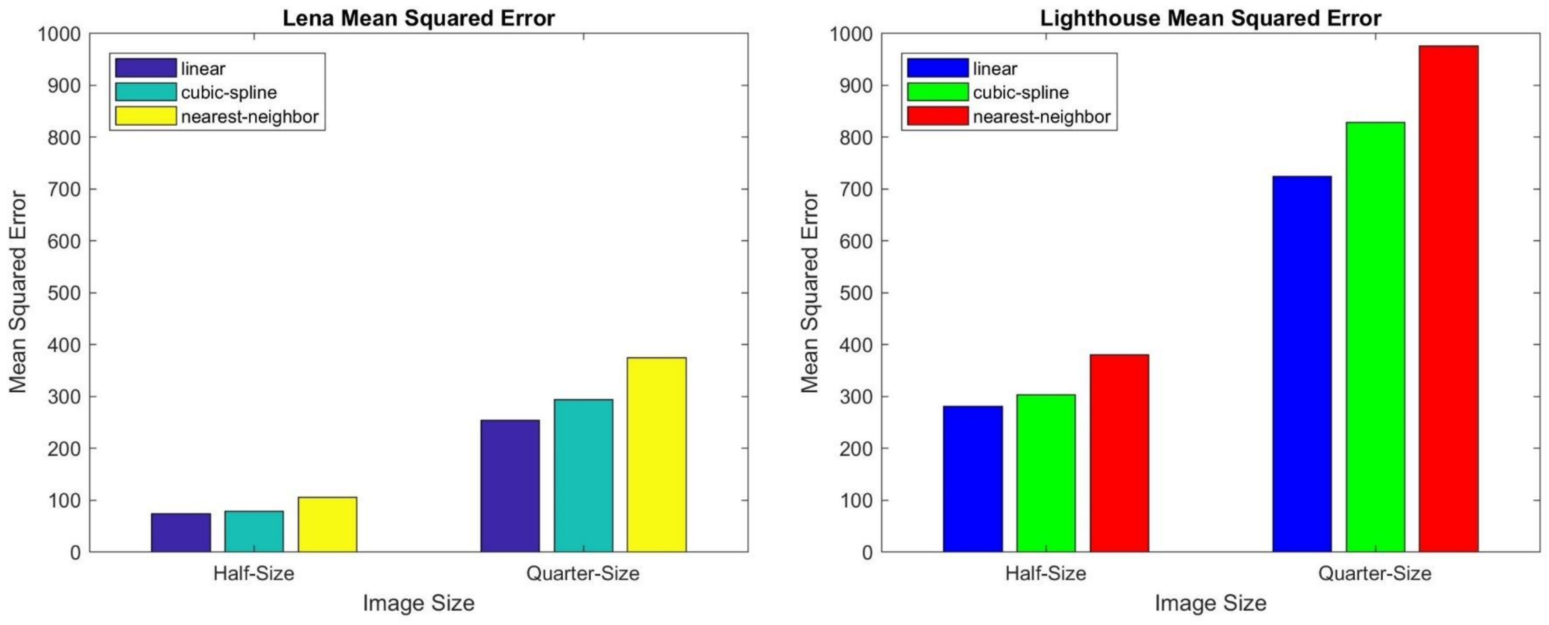


```
The mean squared error for the nearest-neighbor interpolation of the lighthouse quarter-size image is 975.266487
```

```
>> err = immse(lighthouse_qtr_nearest, lighthouse)

err =

  975.2665
```

Once we established that our results were accurate, we displayed our MSE results in two ways. First, we used the “fprintf” function to textually display the MSE values in a clear, easy to understand message as shown above in figure 2.2. While this shows the precise values of the MSEs for each image, it does not illustrate a comparison between MSE, image size and type, and interpolation method. In order to graphically display our results, we created the bar graphs shown in figure 2.3. In order to highlight the differences in the MSEs between the two images, we ensured the axes of both graphs were the same.

Figure 2.3: MSE Graphs



In each individual image, the error produced when interpolating the half-size images is much less than that of the quarter-size images. This is because far more estimation is required to increase the size of the smaller image to 512x512, and estimation begets error. Furthermore, regarding the various interpolation methods, linear interpolation is the most accurate, as it produces the smallest error, followed by cubic-spline, then nearest-neighbor. Nearest-neighbor interpolation is the least accurate because it operates by calculating the distance in between unknown points, and assigning the value of the closest known point to the point being interpolated; this does not create any new data [1]. Linear and cubic-spline methods are more accurate because they fit polynomial functions between the know points and use these functions to generate new data [1].

When comparing the ‘lena’ image to the ‘lighthouse’ image, a large discrepancy in the MSE can be seen. For each image size and method, the MSE for ‘lena’ is less than half that of the ‘lighthouse’ image. This is because there is less variation among the points in the ‘lena’ image, so there is less room for error.

References:

[1]"Interpolation Methods- MATLAB & Simulink- MathWorks Australia", *Au.mathworks.com*, 2018. [Online]. Available: https://au.mathworks.com/help/curvefit/interpolation-methods.html. [Accessed: 28- May- 2018].

Appendix A

i. Interpolation and MSE Script

```
clc;
clear;

%% Interpolation
%import
lighthouse = double(imread('lighthouse.png'));
lighthouse_half = double(imread('lighthouse_half.png'));
lighthouse_qtr = double(imread('lighthouse_quarter.png'));

lena = double(imread('lena.tif'));
lena_half = double(imread('lena_half.tif'));
lena_qtr = double(imread('lena_quarter.tif'));


%interp lighthouse
lighthouse_half_linear =
MyInterp(lighthouse_half,lighthouse,'linear'); %calls in
function to interp half-size lighthouse image (256x256) to full
size 512x512
lighthouse_qtr_linear =
MyInterp(lighthouse_qtr,lighthouse,'linear');
lighthouse_half_cubic =
MyInterp(lighthouse_half,lighthouse,'spline');
lighthouse_qtr_cubic =
MyInterp(lighthouse_qtr,lighthouse,'spline');
lighthouse_half_nearest =
MyInterp(lighthouse_half,lighthouse,'nearest');
lighthouse_qtr_nearest =
MyInterp(lighthouse_qtr,lighthouse,'nearest');

%interp lena
for i = 1:3 %
```

```
   lena_half_linear(:,:,i) = 
MyInterp(lena_half(:,:,i),lena(:,:,i),'linear');
   lena_qtr_linear(:,:,i) = 
MyInterp(lena_qtr(:,:,i),lena(:,:,i),'linear');
   lena_half_cubic(:,:,i) = 
MyInterp(lena_half(:,:,i),lena(:,:,i),'spline');
   lena_qtr_cubic(:,:,i) = 
MyInterp(lena_qtr(:,:,i),lena(:,:,i),'spline');
   lena_half_nearest(:,:,i) = 
MyInterp(lena_half(:,:,i),lena(:,:,i),'nearest');
   lena_qtr_nearest(:,:,i) = 
MyInterp(lena_qtr(:,:,i),lena(:,:,i),'nearest');
end

figure(1)

title 'Lighthouse Half,'
subplot(1,3,1), imshow(uint8(lighthouse_half_linear))
title 'Lighthouse Half, Linear'
subplot(1,3,2), imshow(uint8(lighthouse_half_cubic))
title 'Lighthouse Half, Cubic-Spline'
subplot(1,3,3), imshow(uint8(lighthouse_half_nearest))
title 'Lighthouse Half, Nearest-Neighbor'

figure(2)

subplot(1,3,1), imshow(uint8(lighthouse_qtr_linear))
title 'Lighthouse Quarter, Linear'
subplot(1,3,2), imshow(uint8(lighthouse_qtr_cubic))
title 'Lighthouse Quarter, Cubic-Spline'
subplot(1,3,3), imshow(uint8(lighthouse_qtr_nearest))
title 'Lighthouse Quarter, Nearest-Neighbor'

figure(3)

subplot(1,3,1), imshow(uint8(lena_half_linear))
title 'Lena Half, Linear'
subplot(1,3,2), imshow(uint8(lena_half_cubic))
title 'Lena Half, Cubic-spline'
subplot(1,3,3), imshow(uint8(lena_half_nearest))
title 'Lena Half, Nearest-Neighbor'

figure(4)

subplot(1,3,1), imshow(uint8(lena_qtr_linear))
title 'Lena Quarter, Linear'
subplot(1,3,2), imshow(uint8(lena_qtr_cubic))
```

```
title 'Lena Quarter, Cubic-spline'
subplot(1,3,3), imshow(uint8(lena_qtr_nearest))
title 'Lena Quarter, Nearest-Neighbor'


%% Mean Squared Error Lena

% Lena Half
orig = (lena);

new = (lena_half_linear);
mse_lenhalf_lin = mse(orig,new,'clr');
fprintf ('Mean Squared error for Lena half-sized:\n\nThe mean
squared error for the linear interpolation of the lena half-size
image is %f', mse_lenhalf_lin)

new = (lena_half_cubic);
mse_lenhalf_cube = mse(orig,new,'clr'); %takes advantage of
nargin function by adding a redundant third argument to indicate
that a 3D array is being used
fprintf ('\n\nThe mean squared error for the cubic-spline
interpolation of the lena half-size image is %f',
mse_lenhalf_cube)

new = (lena_half_nearest);
mse_lenhalf_near = mse(orig,new,'clr');
fprintf ('\n\nThe mean squared error for the nearest-neighbor
interpolation of the lena half-size image is %f',
mse_lenhalf_near)

%Lena Quarter
new = (lena_qtr_linear);
mse_lenqtr_lin = mse(orig,new,'clr');
fprintf ('\n\n\nMean Squared error for Lena quarter-
sized:\n\nThe mean squared error for the linear interpolation of
the lena quarter-size image is %f', mse_lenqtr_lin)

new = (lena_qtr_cubic);
mse_lenqtr_cube = mse(orig,new,'clr');
fprintf ('\n\nThe mean squared error for the cubic-spline
interpolation of the lena quarter-size image is %f',
mse_lenqtr_cube)

new = (lena_qtr_nearest);
mse_lenqtr_near = mse(orig,new,'clr');
```

```
fprintf ('\n\nThe mean squared error for the nearest-neighbor
interpolation of the lena quarter-size image is %f',
mse_lenqtr_near)

figure (5)
lhx = categorical({'Half-Size','Quarter-Size'});
lhy = [mse_lenhalf_lin mse_lenhalf_cube mse_lenhalf_near;
mse_lenqtr_lin mse_lenqtr_cube mse_lenqtr_near];
bar(lhx,lhy)
title 'Lena Mean Squared Error'
xlabel 'Image Size'
ylabel 'Mean Squared Error'
ylim([0 1000])
legend('linear','cubic-spline','nearest-neighbor', 'Location',
'northwest')

%% Mean Squared Error Lighthouse

%Lighthouse Half
orig = (lighthouse);

new = (lighthouse_half_linear);
mse_lighthalf_lin = mse(orig,new);
fprintf ('\n\n\nMean Squared error for Lighthouse half-
sized:\n\nThe mean squared error for the linear interpolation of
the lighthouse half-size image is %f', mse_lighthalf_lin)

new = (lighthouse_half_cubic);
mse_lighthalf_cube = mse(orig,new);
fprintf ('\n\nThe mean squared error for the cubic-spline
interpolation of the lighthouse half-size image is %f',
mse_lighthalf_cube)

new = (lighthouse_half_nearest);
mse_lighthalf_near = mse(orig,new);
fprintf ('\n\nThe mean squared error for the nearest-neighbor
interpolation of the lighthouse half-size image is %f',
mse_lighthalf_near)

%lighthouse Quarter
new = (lighthouse_qtr_linear);
mse_lightqtr_lin = mse(orig,new);
fprintf ('\n\n\nMean Squared error for Lighthouse quarter-
sized:\n\nThe mean squared error for the linear interpolation of
the lighthouse quarter-size image is %f', mse_lightqtr_lin)

new = (lighthouse_qtr_cubic);
```

```
mse_lightqtr_cube = mse(orig,new);
fprintf ('\n\nThe mean squared error for the cubic-spline
interpolation of the lighthouse quarter-size image is %f',
mse_lightqtr_cube)

new = (lighthouse_qtr_nearest);
mse_lightqtr_near = mse(orig,new);
fprintf ('\n\nThe mean squared error for the nearest-neighbor
interpolation of the lighthouse quarter-size image is %f\n',
mse_lightqtr_near)

figure(6)
lix = categorical({'Half-Size','Quarter-Size'});
liy = [mse_lighthalf_lin mse_lighthalf_cube mse_lighthalf_near;
mse_lightqtr_lin mse_lightqtr_cube mse_lightqtr_near];
b2 = bar(lix,liy);
title 'Lighthouse Mean Squared Error'
xlabel 'Image Size'
ylabel 'Mean Squared Error'
legend('linear','cubic-spline','nearest-neighbor', 'Location',
'northwest')
b2(1).FaceColor = 'b';
b2(2).FaceColor = 'g';
b2(3).FaceColor = 'r';
```

ii. Interpolation Function

```
%%function to interp image
function image = MyInterp(img,orig,method)
    dimension = size(img);
    orig_dimension = size(orig);

    x = 1:1:dimension(1);
    xi = 1:((dimension(1)-1)/(orig_dimension(1)-
1)):dimension(1);


    img = interp1(x,img,xi,method);
    img = rot90(img);

    img = interp1(x,img,xi,method);
    img = rot90(img, 3);

    image = img;
end
```

iii. MSE Function

```
function [ MSE ] = mse(orig,new,varargin)

x = (new - orig).^2;
if nargin == 3
    y = sum(sum(sum(x)));
    MSE = y / (512*512*3);

else
    y = sum(sum(x));
    MSE = y / (512*512);
end
```