Abstract

The goal of this lab is to implement nearest neighbor and bilinear interpolation functions to generate a resized image. Resizing an image to a smaller size is called “down sampling”, whereas resizing an image to a larger size is called “up sampling”. This report will discuss the techniques used to successfully create down-sampled and up-sampled images and what the accuracy of each output image tells us about the performance of the program.

Technical Discussion

How Images are Loaded into the Program

The program begins by reading an image using the imread() function. The image that is used in this lab is “Lab_02_image1.tif”. The image file is finally converted to a type “uint8” matrix using the im2uint8() function. This process is shown below in Figure 1.

```
A = imread("Lab_02_image1.tif");
A = im2uint8(A);

figure()
imshow(A);
title('Original');
```

Figure 1: Importing the lab image and typecasting to type uint8.

After matrix A is created, a printout of the sample image is made before performing any up- or down- sizing operations. Next, the matrix is sent to the “myimresize()” routine where it is transformed and returned. It is important to note that the original image A is not altered in any way. It will be used as a reference and a new image will be produced. The output of the imresize() function is assigned to a matrix of type uint8.

The imresize() function takes as input a grayscale image (a matrix), the size (number of rows and columns) of the resized image, and a string with values ‘nearest’ or ‘bilinear’. The size of the resized image will identify the dimensions of the new image and the string determines what kind of resizing operations will be done to the original image.

The first four matrices created in this program are called downSample1, downSample2, upSample1, and upSample2. They are assigned to the return value of the imresize() function, as shown below in Figure 2.

```
% Sample downsampled and upsampled test
downSample1 = myimresize(A, [40 75], 'nearest');
downSample2 = myimresize(A,[40 75],'bilinear');
upSample1 = myimresize(A, [425 600], 'nearest');
upSample2 = myimresize(A,[425 600],'bilinear');
```

Figure 2: Assigning matrices as result of the myimresize() function.

In the figure above (Figure 2), downSample1 represents the original image A downsized to an image of pixel size 40x75. 40 and 75 are entered as arguments in the myimresize() and they identify the row and column size that the user wants for the new image, respectively. The third and final argument 'nearest' identifies which kind of resizing algorithm to use. In this case, it is asking the myimresize() function to perform nearest-neighbor interpolation. If this argument is 'bilinear', myimresize() will perform bilinear interpolation.

The Structure of myimresize()

This section will cover the workflow and methodology of the myimresize() function. The function begins by assigning variables M and N to the row and column size of the new image, respectively. Next, the function compares the string argument to see what interpolation method should be performed on the original image. The options are either 'nearest' or 'bilinear', which stand for nearest-neighbor interpolation and bilinear interpolation, respectively. If the string argument is equal to either of these, the function will call their respective helper functions. The purpose of the helper functions is to return an image that is an upsized or downsized version of the original, based on the input sizes, M and N. This newly produced image is assigned to 'outputImage'. The myimresize() function is shown below in Figure 3.

```
function outputImage = myimresize(inputImage, inputSize, method)
    % Resize inputImage to get outputImage
    % inputImage = Input image, matrix
    % inputSize = Target imgage size
    % method = Nearest neighbor or bilinear interpolation

    M = inputSize(1);
    N = inputSize(2);

    if (strcmp(method, 'nearest'))
        outputImage = nearest(inputImage, M, N);
    elseif (strcmp(method, "bilinear"))
        outputImage = bilinear(inputImage, M, N);
    end
end
```

Figure 3: The myimresize() function

The Structure of the nearest() function

This section covers the methodology and precision of the nearest() function. Nearest-neighbor interpolation (also known as proximal interpolation or, in some contexts, point sampling – Wikipedia) is a simple method of multivariate interpolation in one or more dimensions. This function creates a new image of size M and N dimensions and relies on the image pixel values of the original image.

In the first few lines of the nearest() function, structs Im1 and Im2 are created. These structs contain information about the images they reference. Im1 pertains to the original image: image A from the main function, and Im2 pertains to new image dimensions. Im1.rows contains the row count of the original image and Im1.cols pertains to the column count of the original image. Similarly, Im2.rows and Im2.cols pertain to the row and column count of the new image, defined by the user in the main function when myimresize() is called. After the row and column size of

the new image are collected by the input arguments, a new matrix called ‘output’ is created and initialized as a zero matrix of size Im2.rows and Im2.cols. Afterward, two row matrices are initialized and their values correspond to row and column sizes in the original image.

This ‘rowCoords’ is created using the linspace() function. linspace() generates a linearly spaced vector along an interval. For example, linspace(1, Im2.rows, Im1.rows) returns a vector of evenly spaced out values from 1 to the value of ‘Im2.rows’. However, the size of this evenly spaced vector will have ‘Im1.rows’ number of numbers. What this is doing is mapping the indices of the row vector in the original image to the indices of the row vector in the new image. For example, if the original image contained 3 rows (three pixels in height), and the new image contained 100 rows, this linspace() function implementation would look like linspace(1,100,3), and would return 1, 50.5, and 100, in this order. The goal of this method is to identify the key index values that would serve as reference points when performing nearest-neighbor interpolation later in the function. Similarly, the variable ‘colCoords’ will be a mapping of the indices of the column vector in the original image to the indices of the column vector in the new image. Since some mappings will not perfect, as described in the previous example, rowCoords and colCoords are rounded using the round() function. This will also prevent indices from being non-integer values. This will be useful in the nearest-neighbor interpolation algorithm later. In Figure 4, shown below, this process happens in the beginning of the nearest() function.

```
function output = nearest(input, M, N)
    % This function will perform nearest neighbor interpolation.
    Im1.rows = size(input,1); Im1.cols = size(input,2);
    Im2.rows = M; Im2.cols = N;
    % Create the first instance of the output array
    output = zeros(Im2.rows, Im2.cols);
    % We will want to floor these values (rowCoords and colCoords)
    rowCoords=round(linspace(1,Im2.rows,Im1.rows)); colCoords=round(linspace(1,Im2.cols,Im1.cols));
```

Figure 4: Creating helper variables used in nearest-neighbor approach.

The next step of the nearest() function is performing the nearest-neighbor interpolation procedure. This is done with the help of the variables created previously. The first iteration starts by going through each row and column of the new “blank” image – previously instantiated as a zero matrix – and output variable. The variable ‘x’ corresponds to the new image’s row index value and ‘y’ corresponds to the new image’s column index. Next, the function goes through all the values in rowCoords. For every iteration in this for loop, the variable ‘i’ is the index in the original image. This is thanks to the linspace() function because every value in rowCoords is mapped to each value in the original image. Since the linspace() function returns a scaled row vector, indices in the original and new image can be accessed simultaneously. Another example of linspace is shown in Figure 5 below, except the function returns a row vector of evenly spaced values from 0 to 100 with 5 values in the vector (i.e. linspace(0,100,5)).

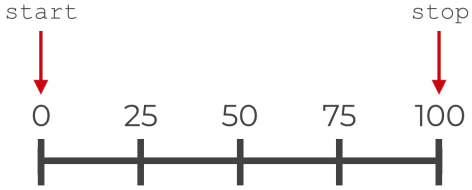

Figure 5: Visual representation of linspace(0,100,5).

Using this scaled property of rowCoords and colCoords, the function access four pixels and their indices. The indices of each of these pixels in the x- and y-direction are assigned to variables (‘last-i-Val’, ‘last_j_Val’), (‘last-i-Val’, ‘current_j_Val’), (‘current_i_Val’, ‘last_j_Val’), and (‘current_i_Val’, ‘current_j_Val’), respectively. These coordinates are represented in Figure 6 below.

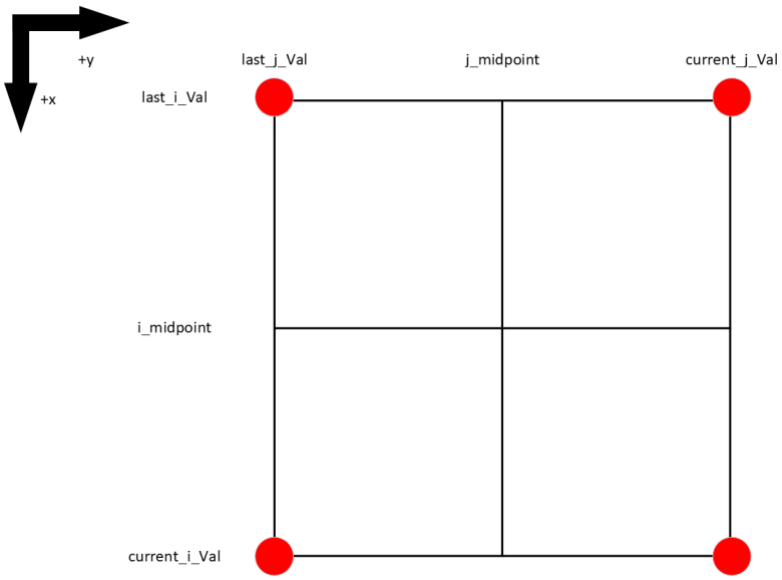

Figure 6: Nearest-neighbor pixel coordinates used in nearest() function.

The ‘i_midpoint’ and ‘j_midpoint’ values represent the threshold that the value of x and y must reach in order to be considered closer to current_i_Val and current_j_Val, respectively. Using this technique, the function creates ranges of values that x and y must be within. After x and y fall within one of the four quadrants, the values of x_idx and y_idx are assigned the value of i and j. This will determine what the pixel value of (x,y) will be in the new image. Once the values of x_idx and y_idx are assigned, their respective for loop iterating through rowCoords and colCoords will stop by the ‘break’ command, and conclude the ‘y^{th}‘ iteration by assigning the value of output(x,y) to the value of input(x_idx, y_idx). Using nearest-neighbor interpolation, the function uses complete information to fill in the gaps of “unknown” information in the new image. After the matrix input has been assigned values for all its coordinates in the x- and y-direction, it is converted to type ‘uint8’ and returned to the myimresize() routine.
The nearest-neighbor interpolation algorithm is shown below in Figure 7.

```
    for x=1:Im2.rows
        for y=1:Im2.cols
            for i=2:length(rowCoords)
                last_i_Val=rowCoords(1,i-1);
                current_i_Val=rowCoords(1,i);
                i_midpoint = round((current_i_Val + last_i_Val) / 2);
                if (x>=last_i_Val && x<=i_midpoint)
                    x_idx=i-1;
                    break;
                elseif (x>i_midpoint && x<=current_i_Val)
                    x_idx=i;
                    break;
                end
            end
            for j=2:length(colCoords)
                last_j_Val=colCoords(1,j-1);
                current_j_Val=colCoords(1,j);
                j_midpoint = round((current_j_Val + last_j_Val) / 2);
                if (y>=last_j_Val && y<=j_midpoint)
                    y_idx=j-1;
                    break;
                elseif (y>j_midpoint && y<=current_j_Val)
                    y_idx=j;
                    break;
                end
            end
            output(x,y)=input(x_idx,y_idx);
        end
    end
    output = uint8(output);
end
```

Figure 7: Nearest-neighbor interpolation algorithm.

The Structure of the `bilinear()` function

This section covers the methodology and precision of the bilinear() function. Bilinear interpolation is a method for interpolation functions of two variables (e.g., x and y) using repeated linear interpolation. Bilinear interpolation is performed using linear interpolation first in one direction (+x direction), and then again in the other direction (+y direction). Although each step is linear in the sampled values and in the position, the interpolation as a whole is not linear bur rather quadratic in the sample location – Wikipedia.

The method used to determine pixel values in up- and down-scaled images involves the "polynomial fit". Using the polynomial fit, the solution to linear interpolation in the x- and y-axis can be solved as a multilinear polynomial. This is especially important because there are multiple rows and columns to work with in a new-sized image. Similar to the nearest-neighbor interpolation method, bilinear interpolation uses four pixels and interpolations among them to determine a pixel value within their quadrant. In Figure 8 below, it is shown what kind of problem bilinear interpolation is solving.

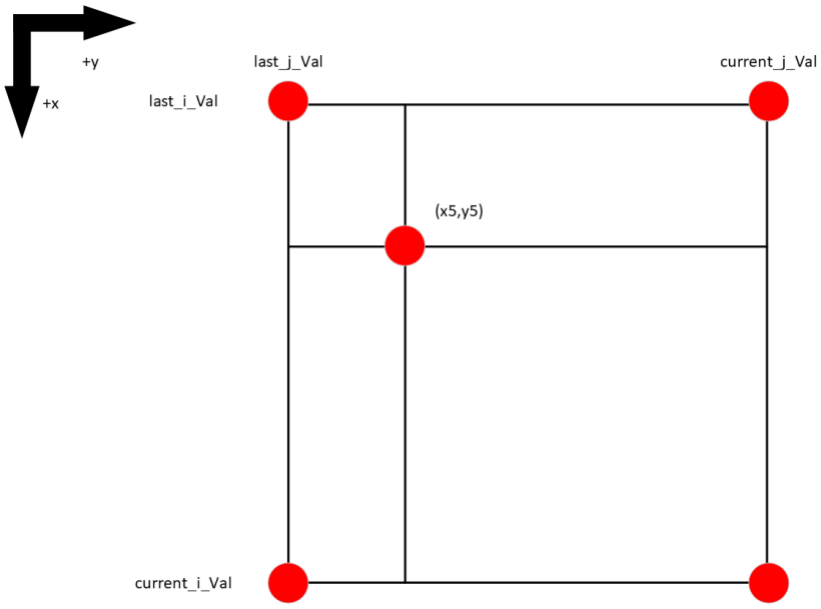

Figure 8: Bilinear pixel coordinates used in bilinear() function.

Just like in the nearest() function, structs ‘Im1’ and ‘Im2’ are created and contain information about the original and new image, respectively. After the row and column size of the new image are collected into these structs, the matrix ‘output’ is instantiated as a zero matrix. This process is shown below in Figure 9.

```
function output = bilinear(input, M, N)
    % This function will perform nearest neighbor interpolation.
    Im1.rows = size(input,1); Im1.cols = size(input,2);
    Im2.rows = M; Im2.cols = N;
    % Create the first instance of the output array
    output = zeros(Im2.rows, Im2.cols);
    % We will want to floor these values (rowCoords and colCoords)
    rowCoords=round(linspace(1,Im2.rows,Im1.rows)); colCoords=round(linspace(1,Im2.cols,Im1.cols));
```

Figure 9: Creating helper variables used in bilinear approach.

The next step of the bilinear function is performing the bilinear interpolation procedure. Variables ‘rowCoords’ and ‘colCoords’ are created and assigned as index mappings from the original image to the new image. The reasoning for this is explained in the nearest() function. These variables are used in the same fashion as before. Using their scaled property, rowCoords and colCoords serve as a reference to the original and new image indices and provide their respective pixel values.

The first iteration starts by going through each row and column of the new “blank” image – previously instatiated as a zero matrix – and output variable. Like in the nearest-neighbor approach, the variable ‘x’ corresponds to the new image’s row index value and ‘y’ corresponds to the new image’s column index. However, in the bilinear approach, no midpoint threshold is used to compare the pixel locations of the new image to the original image. Instead, variables ‘X1’, ‘X2’, ‘X3’, ‘X4’, ‘Y1’, ‘Y2’, ‘Y3’, and ‘Y4’ are created. These are references to the pixel locations and their respective values in the original image. These values are accessed using the rowCoords and colCoords variables. The mapping is the same as in the nearest-neighbor

approach since the original image is used here and has not been altered. Assuming the user gave a new-image-size input, the mapping will work if the new image is larger or smaller than the original. This applies for the nearest-neighbor approach as well thanks to the linspace() function since it can take a range of values and map them to larger or smaller ranges. The positions of these newly created variables are illustrated in Figure 10 below.

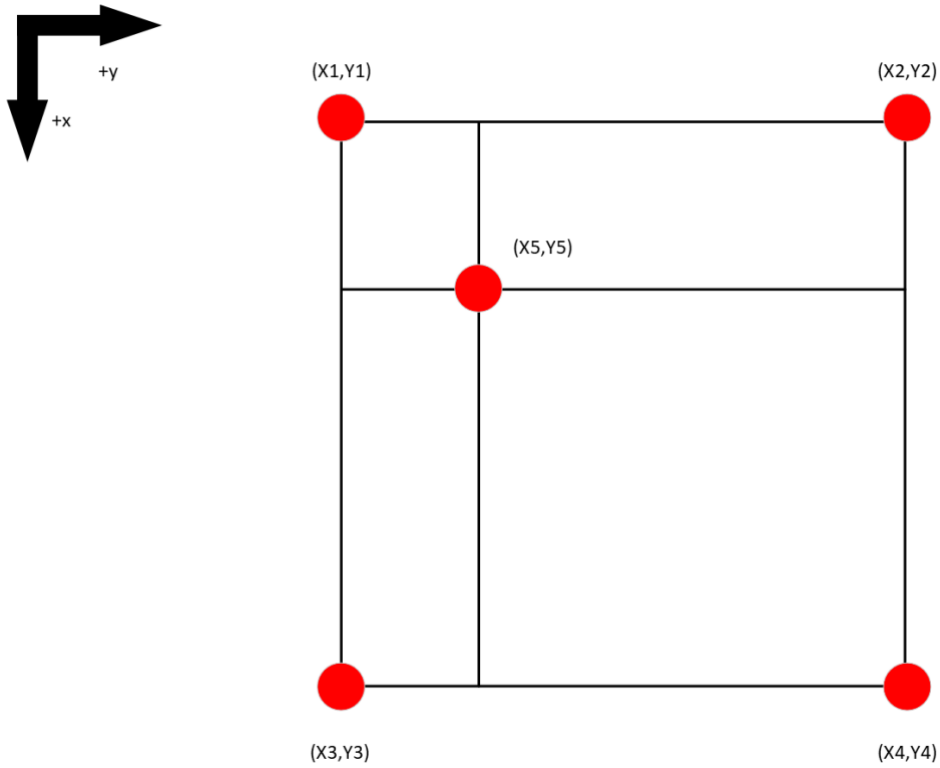

Figure 10: X and Y coordinate mappings used in the bilinear-interpolation algorithm.

After these new coordinates are assigned, the function accesses their pixel values from the original image and assigned to points ‘p1’, ‘p2’, ‘p3’, and ‘p4’. Using these “known” pixel values in the original image, and assuming the pixel coordinates ‘x’ and ‘y’ in the new image are mapped correspondingly, linear interpolation can be performed in both the x- and y-directions using the polynomial fit method by looking at the interpolation problem as a “multilinear polynomial”. A multilinear polynomial is a multivariate polynomial that is affine in each of its variables separately, but not necessarily simultaneously.

Using p1, p2, p3, and p4, polynomial fit can solve the linear system shown in figure 11 below.

$$\begin{bmatrix} 1 & x_1 & y_1 & x_1 y_1 \\ 1 & x_1 & y_2 & x_1 y_2 \\ 1 & x_2 & y_1 & x_2 y_1 \\ 1 & x_2 & y_2 & x_2 y_2 \end{bmatrix} \begin{bmatrix} a_{00} \\ a_{10} \\ a_{01} \\ a_{11} \end{bmatrix} = \begin{bmatrix} f(Q_{11}) \\ f(Q_{12}) \\ f(Q_{21}) \\ f(Q_{22}) \end{bmatrix}$$

Figure 10: A linear system of equations represented as matrices.

Assuming that the system is solvable and that all the conditions in the system satisfy the definition of a multivariate polynomial, a solution can be found for the second matrix containing coefficients. In the function, these coefficients are found using the linsolve() function. Next, the value of the pixel in question can be calculated as a multivariable polynomial solution, as shown in Figure 11 below.

$$f(x,y) \approx a_{00} + a_{10}x + a_{01}y + a_{11}xy.$$

Figure 11: A multilinear polynomial.

The solution of this equations is equal to the interpolated pixel value 'p5' at location (X5,Y5). After the matrix output has been assigned values for all its coordinates in the x- and y-direction, it is converted to type 'uint8' and returned to the myimresize() routine. The bilinear interpolation algorithm interpolation algorithm is shown below in Figure 12.

```
    for x=1:Im2.rows
        for i=2:length(rowCoords)
            lowerX=rowCoords(1,i-1);
            upperX=rowCoords(1,i);
            if (x>=lowerX && x<upperX)
                X1=rowCoords(1,i-1);X2=rowCoords(1,i-1);
                X3=rowCoords(1,i);X4=rowCoords(1,i);
                break;
            end
        end
        for y=1:Im2.cols
            for j=2:length(colCoords)
                lowerY=colCoords(1,j-1);
                upperY=colCoords(1,j);
                if (y>=lowerY && y<upperY)
                    Y1=colCoords(1,j-1);Y2=colCoords(1,j);
                    Y3=colCoords(1,j-1);Y4=colCoords(1,j);
                    break;
                end
            end
            % Interpolation
            p1=input(i-1,j-1);
            p2=input(i-1,j);
            p3=input(i,j-1);
            p4=input(i,j);
            X5=x;Y5=y;
            A = [X1 Y1 X1*Y1 1;
                 X2 Y2 X2*Y2 1;
                 X3 Y3 X3*Y3 1;
                 X4 Y4 X4*Y4 1];
            B=[p1; p2; p3; p4];
            coeff=linsolve(A,double(B));
            p5=round(coeff(1,1)*X5 + coeff(2,1)*Y5 + coeff(3,1)*(X5*Y5) + coeff(4,1));
            output(x,y)=p5;
        end
    end
    output=uint8(output);
end
```

Figure 12: Bilinear interpolation algorithm.

The Resized Images

The original image is first resized to a smaller image of size 40x75, by using both nearest-neighbor and bilinear interpolation. The nearest-neighbor result is assigned to image ‘downSample1’, and the bilinear result is assigned to image ‘downSample2’. The original image is then resized to a larger image of size 425x600, by again using both nearest-neighbor and bilinear interpolation. The nearest-neighbor result is assigned to image ‘upSample1’, and the bilinear result is assigned to image ‘upSample2’. The result of each image is shown below in Figures 14, 15, 16, and 17 respectively.

Figure 13: Original image size before resizing.

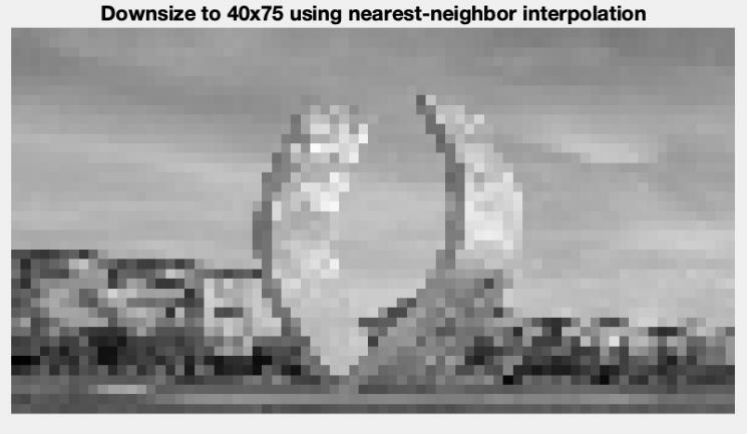

Figure 14: Nearest-neighbor interpolation downsize result.

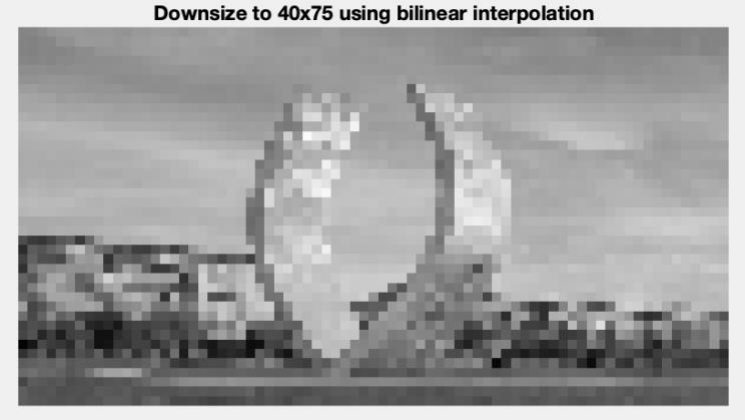

Figure 15: Bilinear interpolation downsize result.

Figure 16: Nearest-neighbor interpolation upsize result.

Figure 17: Bilinear interpolation upsize result.

Comparing the downsize result of nearest-neighbor and bilinear interpolation, the images look very much alike, and it appears that the image was resized accordingly. Additionally, the new-sized images prove that the resizing algorithms functioned as intended. Comparing the upsized results of nearest-neighbor and bilinear interpolation, the images look very much alike, and the images seem to have been resized to their desirable sizes. One small note in the bilinear interpolation upsized image is that edges appear blurrier and softer than in the nearest-neighbor approach. This may be a result of the interpolation method that takes an estimate between boundaries in high-contrast pixel regions. As a result of bilinear interpolation and its more precise approach, the edges are expected to come out blurrier and high-contrast regions will appear to blend more. This also applies to downsized images.

To analyze the effectiveness of the nearest-neighbor and bilinear interpolation algorithms, the downsized and upsized images will be returned to the original image's dimensions of 300x300. The purpose of reconstructing the images is to see how much information has been lost as a result of nearest-neighbor and bilinear interpolation. After resizing the images, they are compared to the original image using a root-mean-squared error (RMSE) value. The RMSE value will evaluate the effectiveness of image reconstruction by computing the pixelwise difference between the original and the reconstructed images. The equation for the RMSE value is shown in Figure 18 below.

$$RMSE = \sqrt{\frac{1}{MN}\sum_{m=0}^{M-1}\sum_{n=0}^{N-1}\left(I1(m,n) - I2(m,n)\right)^2}$$

Figure 18: Root-mean-squared error equation.

The function that handles this calculation is called ‘myRMSE’ and it is shown below in Figure 19.

```
function value = myRMSE(Img1, Img2)
    SUM = 0;
    M=size(Img1,1);
    N=size(Img1,2);
    for i=1:M
        for j=1:N
            q1 = double(Img1(i,j));
            q2 = double(Img2(i,j));
            SUM = SUM + (q1-q2).^2;
        end
    end
    value = sqrt((1/(M*N)) * SUM);
end
```

Figure 19: The myRMSE() function.

The myRMSE() function takes two images as arguments and computes the root-mean-squared error value between them. The value is returned to the main function.

The reconstruction process is simple, such that the upsized and downsized images are resized to the original image’s size (in respect to rows and columns), using the same interpolation method that was used to originally produce them. This process is shown below in Figure 20. Note: the reconstructed images for each downsized and upsized image are ‘reconDownSample1’, ‘reconDownSample2’, ‘reconUpSample1’, and ‘reconUpSample2’, respectively.

```
% Reconstructing images
reconDownSample1 = myimresize(downSample1, [300 300], 'nearest');
reconDownSample2 = myimresize(downSample2, [300 300], 'bilinear');
reconUpSample1 = myimresize(upSample1, [300 300], 'nearest');
reconUpSample2 = myimresize(upSample2, [300 300], 'bilinear');

% Comparing downsample tests
rmseValue1 = myRMSE(A,reconDownSample1);
rmseValue2 = myRMSE(A,reconDownSample2);
fprintf("RMSE of downsampled reconstructed nearest interpolation: %f\n",rmseValue1);
fprintf("RMSE of downsampled reconstructed bilinear interpolation: %f\n",rmseValue2);
% Comparing upsample tests
rmseValue3 = myRMSE(A, reconUpSample1);
rmseValue4 = myRMSE(A, reconUpSample2);
fprintf("RMSE of upsampled reconstructed nearest interpolation: %f\n",rmseValue3);
fprintf("RMSE of upsampled reconstructed bilinear interpolation: %f\n",rmseValue4);
```

Figure 20: Reconstruction and calculating RMSE values.

Results and Discussion

RMSE Values

The result of the RMSE test is shown in Figure 21 below.

```
>> imageResize
RMSE of downsampled reconstructed nearest interpolation: 28.339039
RMSE of downsampled reconstructed bilinear interpolation: 21.115943
RMSE of upsampled reconstructed nearest interpolation: 19.974723
RMSE of upsampled reconstructed bilinear interpolation: 6.038807
```

Figure 21: RMSE values of original vs. reconstructed images.

Upon inspection, it appears that nearest-neighbor interpolation when down sampling an image is less accurate than the bilinear interpolation approach. RMSE value of bilinear interpolation came out to be 21.115943, whereas the RMSE value of nearest-neighbor interpolation came out as 28.339039 – roughly a 7 pixelwise-point difference. Although this is not too high of a difference, this may be as a result of the idea that nearest-neighbor interpolation uses a more-robust rounding approach than bilinear interpolation. Bilinear interpolation uses a multivariate polynomial equation to closely estimate a pixel value among four pixels. Since a polynomial fit may be more accurate than rounding to a "custom" midpoint threshold, is makes sense that bilinear interpolation is more accurate when down sampling. When comparing the RMSE values in the up sampled test, it appears that bilinear interpolation is more accurate than nearest-neighbor interpolation again. The RMSE values came out to be 6.038807 vs. 19.974723 – an approximate 14-point difference.

Conclusion

After reviewing the result of the interpolation methods, it appears that bilinear interpolation is more accurate and worked well when upsizing and downsizing images. Due to the robustness of nearest-neighbor interpolation and the multivariate properties of bilinear interpolation, it makes sense that bilinear interpolation is more accurate than the latter. This lab was successful in

demonstrating the differences between these approaches and the functions help identify the key differences among these two algorithms.

Appendix B

Below is the script used to generate all relative figures and contains the interpolation functions.

```
%{
References:

The linspace() function
https://www.mathworks.com/help/matlab/ref/linspace.html

Nearest-neighbor interpolation
https://en.wikipedia.org/wiki/Nearest-neighbor_interpolation

Structure arrays in MATLAB
https://www.mathworks.com/help/matlab/ref/struct.html

Bilinear interpolation
https://en.wikipedia.org/wiki/Bilinear_interpolation

%}

clear all;
close all;
A = imread("Lab_02_image1.tif");
A = im2uint8(A);

figure()
imshow(A);
title('Original');

% Sample downsampled and upsampled test
downSample1 = myimresize(A, [40 75], 'nearest');
downSample2 = myimresize(A,[40 75],'bilinear');
upSample1 = myimresize(A, [425 600], 'nearest');
upSample2 = myimresize(A,[425 600],'bilinear');
```

```
% Show original images
figure()
imshow(downSample1);
title('Downsize to 40x75 using nearest-neighbor interpolation')
figure()
imshow(downSample2);
title('Downsize to 40x75 using bilinear interpolation')
figure()
imshow(upSample1);
title('Upsize to 425x600 using nearest-neighbor interpolation')
figure()
imshow(upSample2);
title('Upsize to 425x600 using bilinear interpolation')

% Reconstructing images
reconDownSample1 = myimresize(downSample1, [300 300], 'nearest');
reconDownSample2 = myimresize(downSample2, [300 300], 'bilinear');
reconUpSample1 = myimresize(upSample1, [300 300], 'nearest');
reconUpSample2 = myimresize(upSample2, [300 300], 'bilinear');

% Comparing downsample tests
rmseValue1 = myRMSE(A,reconDownSample1);
rmseValue2 = myRMSE(A,reconDownSample2);
fprintf("RMSE of downsampled reconstructed nearest interpolation: %f\n",rmseValue1);
fprintf("RMSE of downsampled reconstructed bilinear interpolation: %f\n",rmseValue2);
% Comparing upsample tests
rmseValue3 = myRMSE(A, reconUpSample1);
rmseValue4 = myRMSE(A, reconUpSample2);
fprintf("RMSE of upsampled reconstructed nearest interpolation: %f\n",rmseValue3);
fprintf("RMSE of upsampled reconstructed bilinear interpolation: %f\n",rmseValue4);

%{
This is the bilinear routine. This function is called from
the "myimresize" routine and performs up- and down-sampling
procedures. The routine takes an input image (type uint8),
and integer values M and N. M represents the number of rows
in the new image and N represents the number of columns in
```

```
the new image. The result of the transformation is stored
in "output" and is returned to "myimresize".
%}
function output = bilinear(input, M, N)
    % This function will perform nearest neighbor interpolation.
    Im1.rows = size(input,1); Im1.cols = size(input,2);
    Im2.rows = M; Im2.cols = N;
    % Create the first instance of the output array
    output = zeros(Im2.rows, Im2.cols);
    % We will want to floor these values (rowCoords and colCoords)
    rowCoords=round(linspace(1,Im2.rows,Im1.rows)); colCoords=round(linspace(1,Im2.cols,Im1.cols));

    for x=1:Im2.rows
        for i=2:length(rowCoords)
            lowerX=rowCoords(1,i-1);
            upperX=rowCoords(1,i);
            if (x>=lowerX && x<upperX)
                X1=rowCoords(1,i-1);X2=rowCoords(1,i-1);
                X3=rowCoords(1,i);X4=rowCoords(1,i);
                break;
            end
        end
        for y=1:Im2.cols
            for j=2:length(colCoords)
                lowerY=colCoords(1,j-1);
                upperY=colCoords(1,j);
                if (y>=lowerY && y<upperY)
                    Y1=colCoords(1,j-1);Y2=colCoords(1,j);
                    Y3=colCoords(1,j-1);Y4=colCoords(1,j);
                    break;
                end
            end
            % Interpolation
            p1=input(i-1,j-1);
            p2=input(i-1,j);
            p3=input(i,j-1);
            p4=input(i,j);
```

```
            X5=x;Y5=y;
            A = [X1 Y1 X1*Y1 1;
                 X2 Y2 X2*Y2 1;
                 X3 Y3 X3*Y3 1;
                 X4 Y4 X4*Y4 1];
            B=[p1; p2; p3; p4];
            coeff=linsolve(A,double(B));
            p5=round(coeff(1,1)*X5 + coeff(2,1)*Y5 + coeff(3,1)*(X5*Y5) + coeff(4,1));
            output(x,y)=p5;
        end
    end
    output=uint8(output);
end

%{
This is the nearest routine. This function is called from the
"myimresize" routine and performs up- and down-sampling
procedures. The routine takes an input image (type uint8),
and integer values M and N. M represents the number of rows
in the new image and N represents the number of columns in
the new image. The result of the transformation is stored
in "output" and is returned to "myimresize".
%}
function output = nearest(input, M, N)
    % This function will perform nearest neighbor interpolation.
    Im1.rows = size(input,1); Im1.cols = size(input,2);
    Im2.rows = M; Im2.cols = N;
    % Create the first instance of the output array
    output = zeros(Im2.rows, Im2.cols);
    % We will want to floor these values (rowCoords and colCoords)
    rowCoords=round(linspace(1,Im2.rows,Im1.rows)); colCoords=round(linspace(1,Im2.cols,Im1.cols));

    for x=1:Im2.rows
        for y=1:Im2.cols
            for i=2:length(rowCoords)
                last_i_Val=rowCoords(1,i-1);
                current_i_Val=rowCoords(1,i);
```

```
            i_midpoint = round((current_i_Val + last_i_Val) / 2);
            if (x>=last_i_Val && x<=i_midpoint)
               x_idx=i-1;
               break;
            elseif (x>i_midpoint && x<=current_i_Val)
               x_idx=i;
               break;
            end
         end
         for j=2:length(colCoords)
            last_j_Val=colCoords(1,j-1);
            current_j_Val=colCoords(1,j);
            j_midpoint = round((current_j_Val + last_j_Val) / 2);
            if (y>=last_j_Val && y<=j_midpoint)
               y_idx=j-1;
               break;
            elseif (y>j_midpoint && y<=current_j_Val)
               y_idx=j;
               break;
            end
         end
         output(x,y)=input(x_idx,y_idx);
      end
   end
   output = uint8(output);
end

%{
This is the myRMSE routine. This function computes the
root mean squared error between two images. This
function takes input two grayscale images of the same size
and outputs a single floating point value, the RMSE.
RMSE is a numeric method for computing the difference between
two images. It is used in this program to evaluate the
effectiveness of the two image reconstruction algorithm by
computing the pixelwise difference between the original and
the reconstructed images.
```

```
%}
function value = myRMSE(Img1, Img2)
   SUM = 0;
   M=size(Img1,1);
   N=size(Img1,2);
   for i=1:M
      for j=1:N
         q1 = double(Img1(i,j));
         q2 = double(Img2(i,j));
         SUM = SUM + (q1-q2).^2;
      end
   end
   value = sqrt((1/(M*N)) * SUM);
end


%{
This is the myimresize routine. This is a function that calls to other
functions "nearest" and "bilinear" to perform up- and down-sampling
operations. The output of each of these function calls is stored into the
"outputImage" matrix and is returned at the end of the routine to main.
%}
function outputImage = myimresize(inputImage, inputSize, method)
   % Resize inputImage to get outputImage
   % inputImage = Input image, matrix
   % inputSize = Target imgage size
   % method = Nearest neighbor or bilinear interpolation

   M = inputSize(1);
   N = inputSize(2);

   if (strcmp(method, 'nearest'))
      outputImage = nearest(inputImage, M, N);
   elseif (strcmp(method, "bilinear"))
      outputImage = bilinear(inputImage, M, N);
   end
end
```

For editing records, please refer to the GitHub repository here:
https://github.com/alejandrohsanchez/Digital-Image-Processing-Projects/tree/main/Labs/02