# 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.

Text

Description automatically generated with medium confidence

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.

Text

Description automatically generated

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.

Text, letter

Description automatically generated

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.

Text

Description automatically generated

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)).

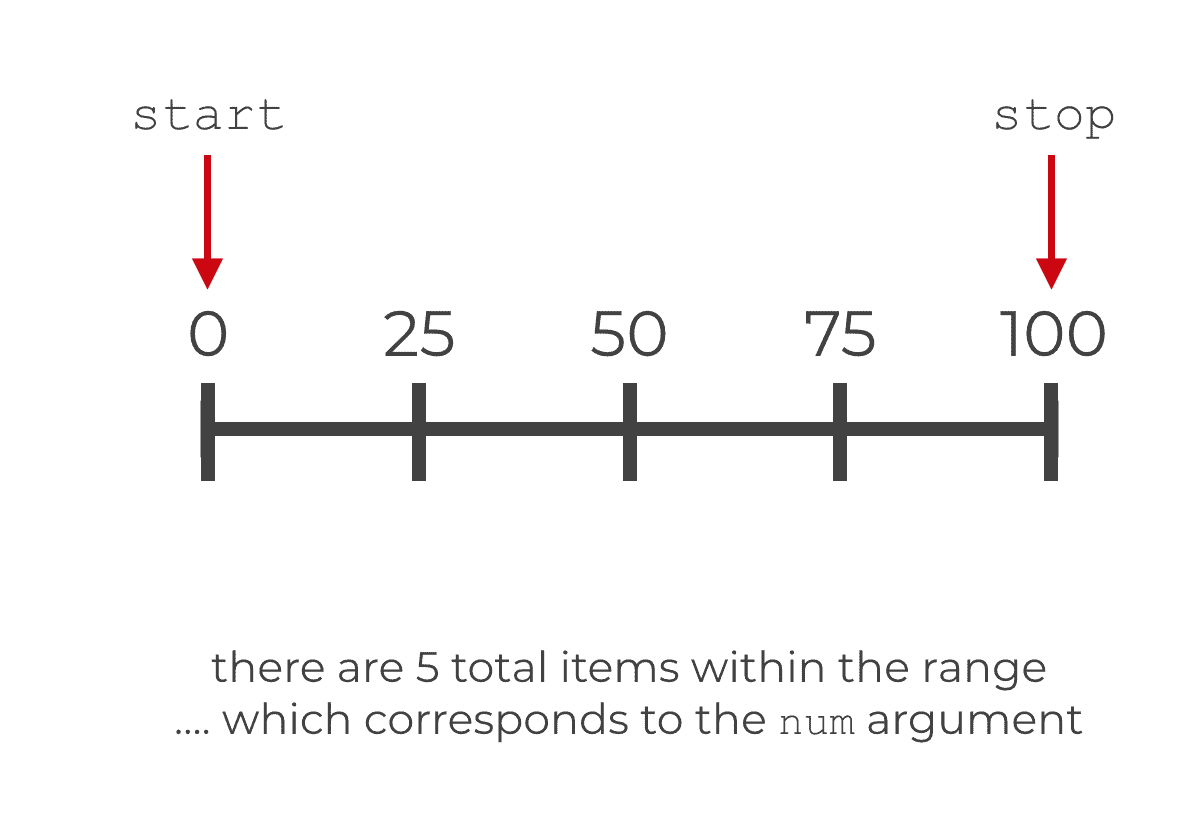


Figure 5: Visual representation of linspace(0,100,5).

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.

A picture containing chart

Description automatically generated

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 ‘yth‘ 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.

Graphical user interface, text, application

Description automatically generated

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.

Chart, timeline

Description automatically generated

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.

Text

Description automatically generated

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.

Text

Description automatically generated

# Results and Discussion