Delhi Technological University

Department of Applied Physics

Interpolation of Digital Images using MATLAB

Mid-term Evaluation Project Report

(EP-208) Computational Methods

A Project by:-
Dhruv Tyagi (2K19/EP/032)
Ayush Kumar (2K19/EP/030)

Acknowledgement

We would like to express our sincere gratitude towards our respected professor, Dr. Ajeet Kumar for encouraging us and providing us with the opportunity to expand our subject knowledge by working on this project.

His continued support & valuable criticism along with his guidance have been huge contributions towards the successful completion of this project.

Table Of Contents

An Introduction to Image Processing

Image processing is a method of processing digital images by performing certain operations on it based on an algorithm. Common image processing methods include image enhancement, restoration, encoding, and compression. Digital Image processing offers applications in diverse fields such as medical science, consumer electronics, agriculture, computer graphics, and even in military.

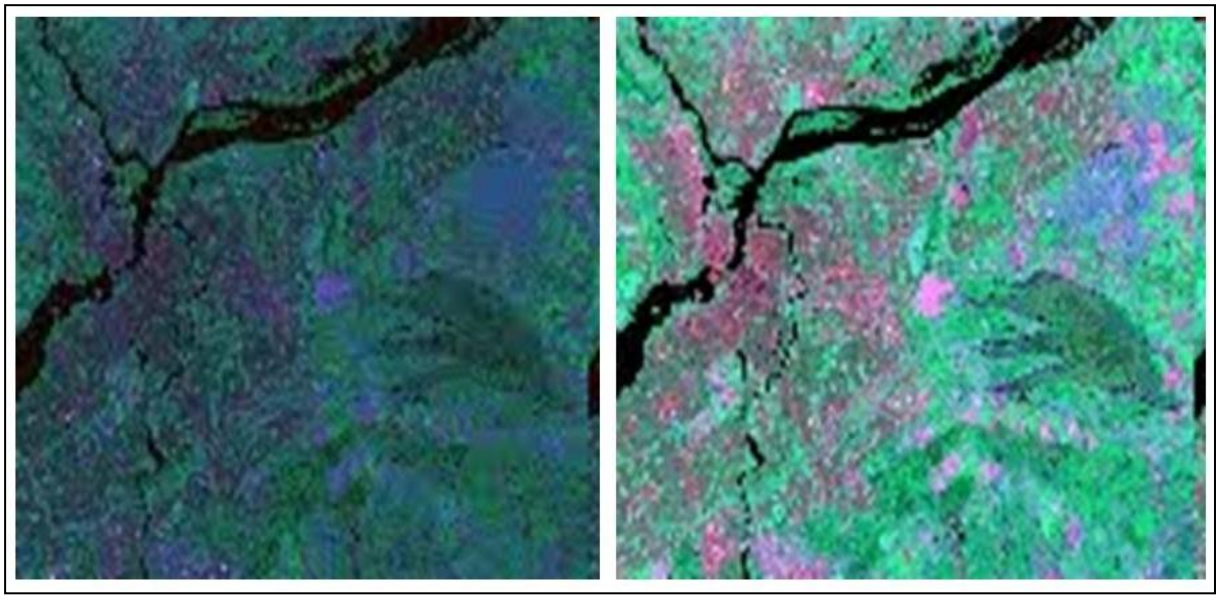


Image Processing offers numerous applications in image editing & enhancement. The images shown above are the before & after results of an image contrast enhancement operation.

The main image processing algorithms that this project focuses on are image interpolation algorithms widely used in image editing software's such as Adobe Photoshop & MATLAB. Image interpolation algorithms are mainly used to improve image quality & to resize images.

Perhaps the most common use of an image interpolation algorithm being employed is seen, when we zoom into any given image. Zooming refers to increasing the overall quantity of pixels, so that when an image is zoomed into, more detail is observable.

Image Magnification

Image Interpolation algorithms are mainly used to resize/zoom into images. Thus it is necessary to understand the working principle behind magnifying or 'zooming' into images.

The main application that image interpolation is used for in image processing, addresses the problem of generating high resolution images from its low-resolution. This process is also known as 'Image Magnification'.

A zoomed in image will typically display pixelation since when zoomed into, the image needs to be interpolated to find intermediate pixel values that did not exist previously in the original images resolution.

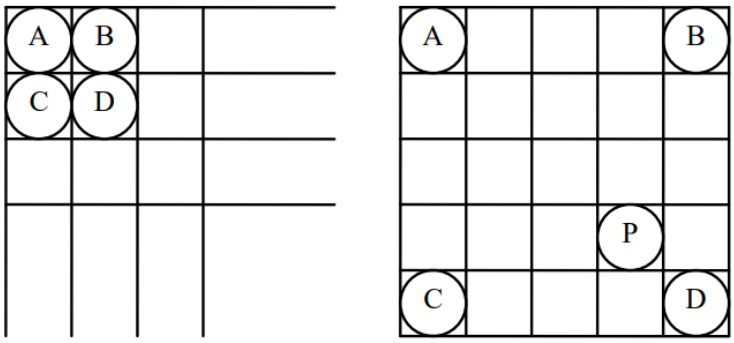
The basic idea of interpolation is that it utilizes known data to estimate unknown values between and around the values that have already been determined.

A key feature of image interpolation is that it works in two directions, and tries to achieve a best approximation of a pixel's intensity based on the values at surrounding pixels.

So how is interpolation relevant to magnification of images?

The basic principle of image magnification is to increase the image pixel number, so a low resolution image is converted to a high resolution image. Since the number of pixels is effectively being increased, new pixels are being

added between previously adjacent pixels. This effect of magnifying the image & consequent addition of new pixels between pre-existing ones is highlighted in the figure shown below.



When a small image is enlarged, such as in the example shown above, the colour values of original 4 adjacent pixels marked A, B, C, and D are mapped to corresponding new locations in accordance with the magnification factor. Due to the magnification factor, new unknown pixel spaces are introduced between A, B, C, and D, such as P. To evaluate these intermediate unknown pixel values a series of interpolation algorithms are employed.

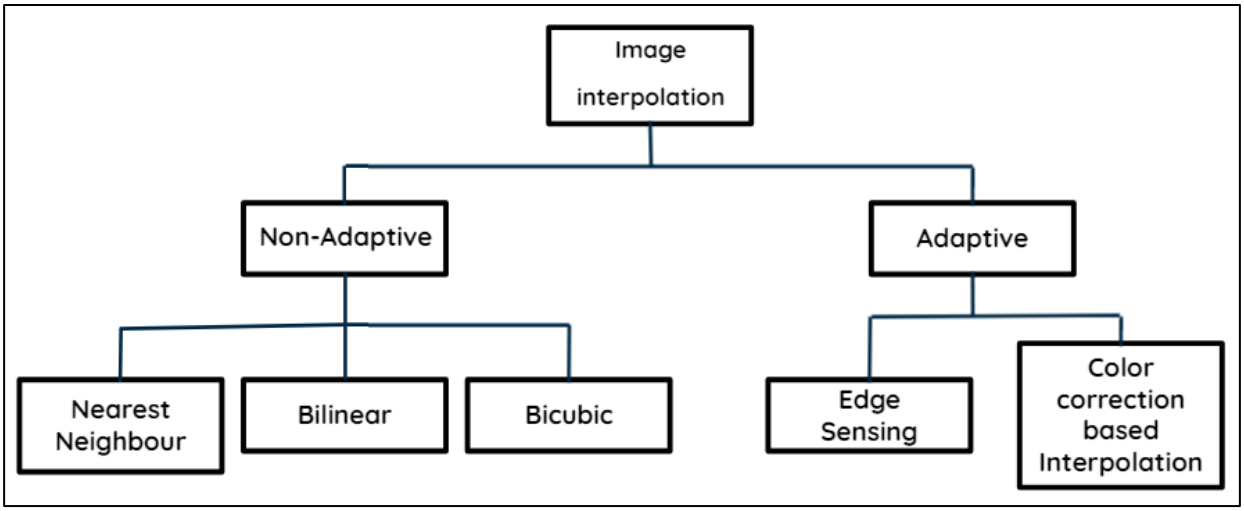
While enlarging an image it is not possible to discover any more information in the image than already exists, and thus image quality inevitably suffers. Hence the model being employed to describe the relationship between high-resolution and low resolution pixels plays a critical role in the performance of an interpolation algorithm. In this project we have attempted to discuss a few different types of non adaptive interpolation algorithms.

Adaptive & Non-adaptive Algorithms

Common interpolation algorithms can be grouped into two categories: adaptive and non-adaptive. The difference between these two categories being that non-adaptive algorithms apply a fixed pattern to each pixel of an image without considering other parameters such as features or the edges, meanwhile Adaptive algorithms employ on spectral and spatial features available in the

neighboring pixels in order to interpolate the unknown pixel as close to original as possible.

Adaptive algorithms are in general considered to be better at image processing than non adaptive however they require increasingly complex lines of code to be implemented & they also require greater computational time & processing power as compared to non-adaptive algorithms. Due to these reasons non-adaptive algorithms are not discussed in detail in this project.



Types of Image Interpolation Algorithms are roughly classified into 2 types adaptive & non adaptive

MATLAB - Image Processing Toolbox™

For the simulation aspect of this project, we designed programs capable of carrying out different types of interpolation algorithms on a set of images. The software we used to simulate these interpolation algorithms was the well renowned programming platform - MATLAB.

We chose MATLAB as the programming environment since it offers convenient toolbox packages catered for specific fields, such as the Image Processing Toolbox™ which we have utilized extensively in our programs.

Image Processing Toolbox™ provides a comprehensive set of reference-standard algorithms and workflow apps for image processing, analysis, visualization, and algorithm development.

Some useful functions provided by the Image Processing Toolbox™, that we have also implemented in our codes include:-

- Imread()

A = imread(filename) reads the image from the file specified by filename, inferring the format of the file from its contents.

- Imresize()

B = imresize(A,scale) returns image B that is scale times the size of A. The input image A can be a grayscale, RGB, or binary image.

- Im2uint8

(Note: Although instead of im2unint8 the cast function has been used, im2uint8 serves the same function as the 'cast' function.)

J = im2uint8(I) converts the grayscale, RGB, or binary image I to uint8, rescaling or offsetting the data as necessary. If the input image is of class uint8, then the output image is identical. If the input image is of class logical, then im2uint8 changes true-valued elements to 255.

Apart from these useful functions, the Image Processing Toolbox™ also offers a vast variety of diverse RGB, Gray-Scale & Binary pre-loaded demo images that users may choose from. The list of a few different types of images along with their format has been shown below:

AT3_1m4_01.tif	AT3_1m4_02.tif
AT3_1m4_03.tif	AT3_1m4_04.tif
AT3_1m4_05.tif	AT3_1m4_06.tif
AT3_1m4_07.tif	AT3_1m4_08.tif
AT3_1m4_09.tif	AT3_1m4_10.tif
autumn.tif	bag.png
blobs.png	board.tif
cameraman.tif	canoe.tif
cell.tif	circbw.tif
circles.png	circuit.tif
coins.png	concordaerial.png
concordorthophoto.png	eight.tif
fabric.png	football.jpg
forest.tif	gantrycrane.png
glass.png	greens.jpg
hestain.png	kids.tif
liftingbody.png	logo.tif
m83.tif	mandi.tif
moon.tif	mri.tif
office_1.jpg	office_2.jpg
office_3.jpg	office_4.jpg
office_5.jpg	office_6.jpg
onion.png	paper1.tif
pears.png	peppers.png
pillsetc.png	pout.tif
rice.png	saturn.png
shadow.tif	snowflakes.png
spine.tif	tape.png
testpat1.png	text.png
tire.tif	tissue.png
trees.tif	westconcordaerial.png
westconcordorthophoto.png	

List of Inbuilt demo images offered by the Image Processing Toolbox™

More than 140 built-in images come pre-loaded as a part of the Image Processing Toolbox™.

Any of these images may be called directly in a program without the need to specify which directory they are present in. To view the full list of preloaded images offered by the toolbox, one may simply open the directory given below:

C:\Program Files\MATLAB\R2012b\toolbox\images\imdemos.

As a side note, it is also relevant to discuss how images are stored in different formats since our implementation of the interpolation algorithms is largely compatible with only true RGB images. This raises the question, what exactly are RGB images?

RGB Images

Each pixel of a RGB color image corresponds to a triplet of red, blue and green color values. When an image is classified as uint8 (unsigned 8-bit integer), the range of each 3 color value is [0,255]. The imread() function mentioned earlier thus stores RGB images as 3 dimensional matrices, with each element of the matrix signifying a pixel with color values ranging between 0 & 255.

The value 255 is determined by calculating 2^8. A fully saturated color in a 24-bit RGB color space is represented by the value 255. At the other extreme, the value 0 denotes no color. For the class uint8, each pixel is represented by 1 byte. Other classes exist, such as uint16, but for the purpose of this project, the class uint8 has been used. In the field of image processing, uint8 is significant because this class is common when dealing with image files in the format of a JPEG or TIFF. The following RGB color cube, which has been normalized, represents the RGB color space

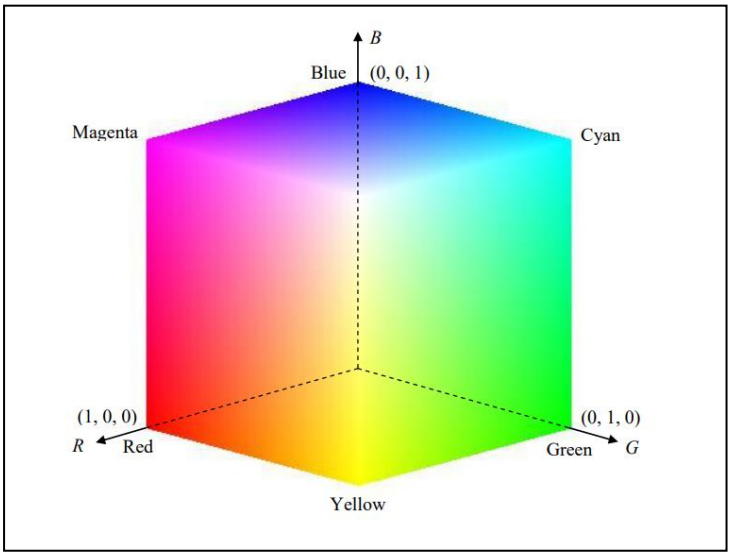


Figure: The RGB color cube, RGB images are encoded to have 3 different channels corresponding to the 3 different colors red, green & blue.

Image Interpolation Algorithms

In the previous sections, interpolation & its significance in image magnification as well as the adaptive & non-adaptive algorithms were discussed. In this section, we take a brief look at the different types of image interpolation algorithms employed in image processing methods.

The scope of our project mainly focuses on the 3 most commonly used non-adaptive interpolation algorithms, namely:-

i. Nearest Neighbour Interpolation
ii. Bilinear Interpolation
iii. Bicubic Interpolation

Each of these different types of interpolation methods results in a different look to final image. Thus, it is usually desired that the quality, or visible distinction for each pixel, is retained throughout the interpolation function.

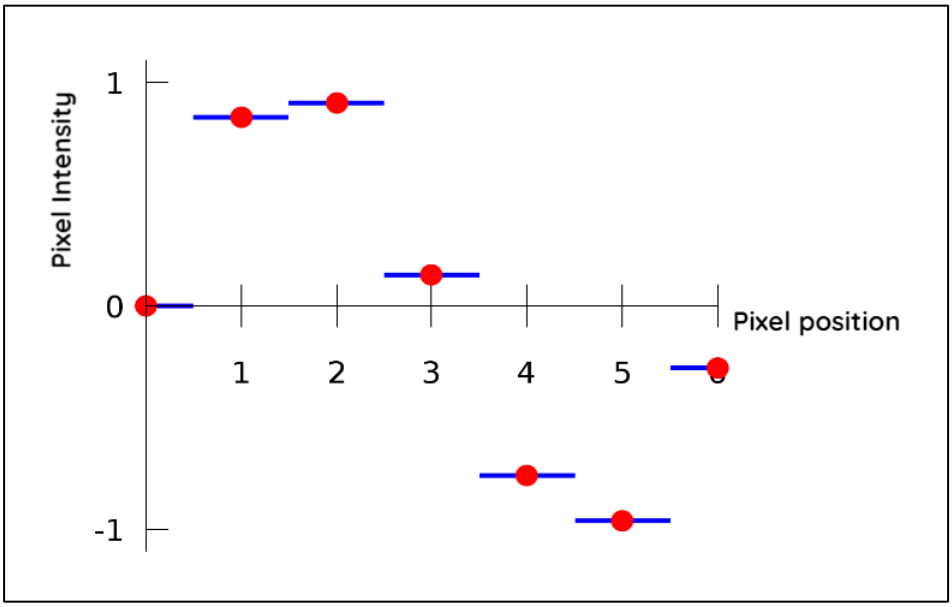
Nearest Neighbour Interpolation

Nearest neighbour interpolation is the most basic & simplest image interpolation algorithm. It also requires by far the least amount of computation time. Designing a code for nearest neighbour interpolation is also quite straightforward since it is not a computationally complex algorithm.

Nearest neighbour interpolation works by deciding the pixel intensity value from the nearest pixel to the specified input coordinates and assigns that value to the output coordinates. Technically speaking nearest neighbour interpolation does not actually carry out interpolation; it simply copies the pixel intensity values of the nearest pixel to pixels of unknown intensity.

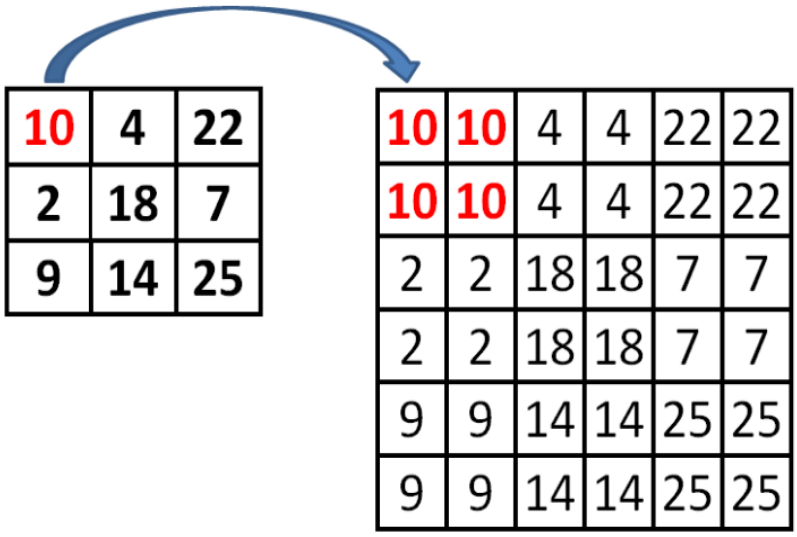
Let us imagine a random set of binary pixels in 1D all ranging in intensity between 0 and 1. If these pixels are plotted onto a graph of intensity vs. pixel

position, & nearest neighbour interpolation is employed to interpolate unknown intermediate pixel values that exist between the given set of pixels, the algorithm would interpolate them as shown in the figure below.



Nearest Neighbour Interpolation of a random 1D pixel set, plotted on a Pixel intensity vs. Pixel position graph

The blue lines in the figure show the interpolated intensity values of the intermediate pixels. In a 2D RGB image the nearest interpolation algorithm would function similarly, obtaining unknown values of pixels by considering their respective nearest neighbour as is shown in the figure below:



Interpolation kernel of it for each direction is,

$$u(x) = \begin{cases} 0, & |x| > 0.5 \\ 1, & |x| > 0.5 \end{cases}$$

Where, x represents the distance between interpolating point and grid point.

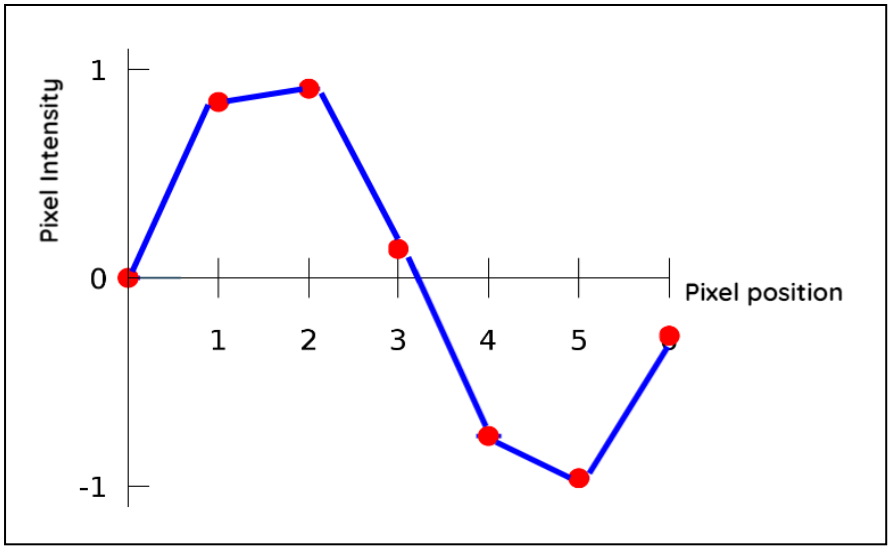
Despite being the most elementary & computationally undemanding algorithm, nearest neighbour is often not preferred over other interpolation algorithms as the image quality produced after nearest neighbour interpolation often suffer from a high degree of pixelation & jagged artifacts as compared to other interpolation algorithms.

Bilinear Interpolation

Bilinear interpolation is essentially an extension of linear interpolation, modified for the purpose of resizing or re-sampling images. The basic principle behind bilinear interpolation is that it takes a weighted average of 4 neighbourhood pixels around the unknown pixel to calculate the final interpolated value.

We are all well aware of the principle behind linear interpolation, where a straight line fit is made through any 2 given points after computing the gradient & the y intercept.

Once again, let us consider the case of a random set of binary pixels in 1D all ranging in intensity between 0 and 1. If these pixels are plotted onto a graph of intensity vs. pixel position, & linear interpolation is carried out, the graph we observe is as follows:

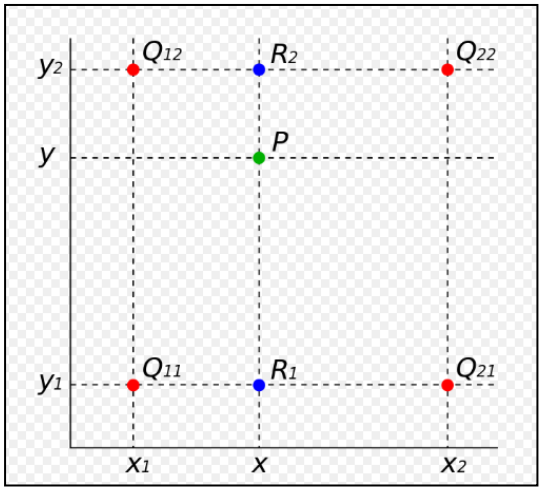


Bilinear Interpolation of 1D pixel set, plotted on a Pixel intensity vs. Pixel position graph

The blue lines in the figure above show the linearly interpolated intensity values of the intermediate pixels.

Bilinear interpolation is carried out in a similar fashion, with the significant difference being that bilinear interpolation performs interpolation in both directions, horizontal and vertical. Two linear interpolations are performed in one direction and next linear interpolation is performed in the perpendicular direction.

Suppose that we want to find the value of the unknown function f at the point P = (x, y). It is assumed that we know the value of f at the four points $Q_{11} = (x_1, y_1)$, $Q_{12} = (x_1, y_2)$, $Q_{21} = (x_2, y_1)$, and $Q_{22} = (x_2, y_2)$.



The figure above shows the four points, Q_{11}, Q_{12}, Q_{21}, & Q_{22} as well as the point which we want to interpolate, P.

As mentioned, we begin by first linearly interpolating in the x direction. This gives us:

$$f(x, y_1) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21}),$$
$$f(x, y_2) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22}).$$

Then we linearly interpolate in the y-direction to obtain the desired estimate as follows:

$$f(x, y) \approx \frac{y_2 - y}{y_2 - y_1} f(x, y_1) + \frac{y - y_1}{y_2 - y_1} f(x, y_2)$$

$$= \frac{y_2 - y}{y_2 - y_1} \left(\frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21}) \right) + \frac{y - y_1}{y_2 - y_1} \left(\frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22}) \right)$$

Opening the brackets & taking the term $\frac{1}{(x2-x1)(y2-y1)}$ common from the remaining terms, allows us to re-write the equation as:

$$= \frac{1}{(x_2 - x_1)(y_2 - y_1)} \begin{bmatrix} x_2 - x & x - x_1 \end{bmatrix} \begin{bmatrix} f(Q_{11}) & f(Q_{12}) \\ f(Q_{21}) & f(Q_{22}) \end{bmatrix} \begin{bmatrix} y_2 - y \\ y - y_1 \end{bmatrix}$$

An alternative solution that makes use of matrix manipulation also allows us to reach the same result seen above.

Bilinear interpolation gives visually better results as compared to nearest neighbour interpolation while taking lesser computation time as compared to bicubic interpolation. Bilinear interpolation has also proven to be quite useful in reducing the visual distortions that emerge as a result of resampling images.

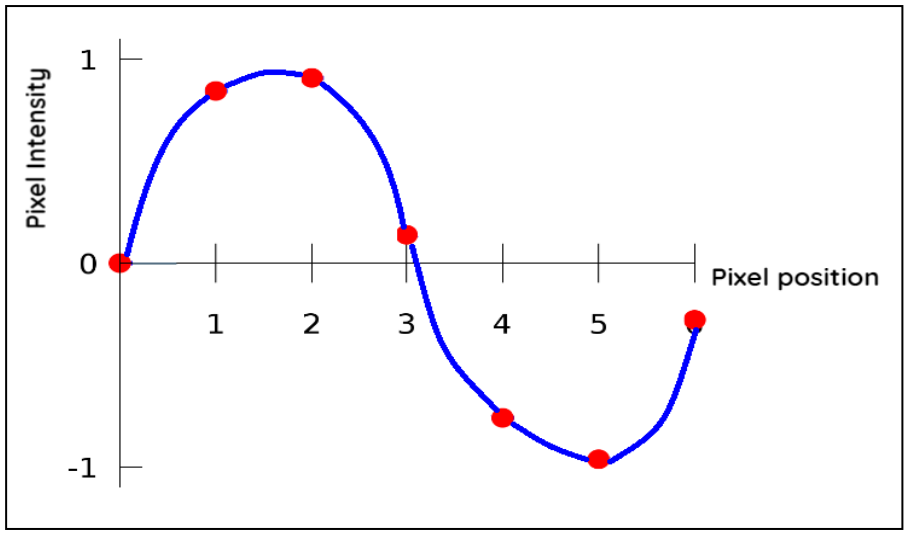
Bilinear interpolation is one of the basic & commonly used resampling techniques in computer vision and image processing, where it is also called **bilinear filtering** or **bilinear texture mapping**.

Bicubic Interpolation

Bicubic goes one step beyond bilinear by considering the closest 4x4 neighborhood of known pixels for a total of 16 pixels. Bicubic Interpolation is essentially an extension of regular cubic interpolation used for interpolating data points & hence resizing/resampling a two dimensional regular grid.

In cubic interpolation any given function, $f(x)$ may be interpolated within a certain interval if the values of the function $f(x)$ & its derivatives are known at the said interval, using a third degree polynomial approximation.

Once again let us consider the same set of binary pixels in 1D all ranging in intensity between 0 and 1. The intensity vs. pixel position graph after cubic interpolation has been carried out would look like this:



Bicubic Interpolation of a 1D pixel set, plotted on a Pixel intensity vs. Pixel position graph

The cubic polynomial fit usually provides an accurate estimate of what the pixel intensity of the unknown pixels might be since gradual gradient changes of pixel intensity around object borders in an image are quite likely.

The principle behind bicubic interpolation is similar to that behind cubic interpolation, except that bicubic interpolation employs cubic interpolation in both the horizontal & vertical axis directions. (Analogous to the case of linear bilinear interpolation as well)

Since a grid of 4x4 pixels around any single specific unknown pixel is relatively quite large, closer pixels are given a higher weighting in the calculation as compared to those that are further away.

Suppose the function values f and the derivatives f_x, f_y, & f_{xy} are known at the four corners (0, 0), (1, 0), (0, 1), & (1, 1) of a unit square. The interpolated surface can then be written as

$$p(x,y) = \sum_{i=0}^{3}\sum_{j=0}^{3} a_{ij} x^i y^j .$$

Here a_{ij} represents the weight parameter, and i & j represent the horizontal and vertical coordinates, respectively.

The interpolation problem consists of determining the 16 coefficients a_{ij}. Matching $p(x,y)$ with the function values yields four equations furthermore there are eight equations for the derivatives in the x & y directions, and four more equations corresponding to the xy mixed partial derivative.

This procedure yields a surface $p(x,y)$ on the unit square [0, 1] x [1, 0] that is continuous and has continuous derivatives.

Then, all the unknown coefficients are grouped into a row vector a_{ij}, & the 16 equations mentioned above are reformulated into the linear equation $A\alpha = x$, where x is the transpose of the row vector formed from the functional values of each of the 16 equations.

Solving the linear equation, will allow us to calculate the surface $p(x,y)$ as shown below:

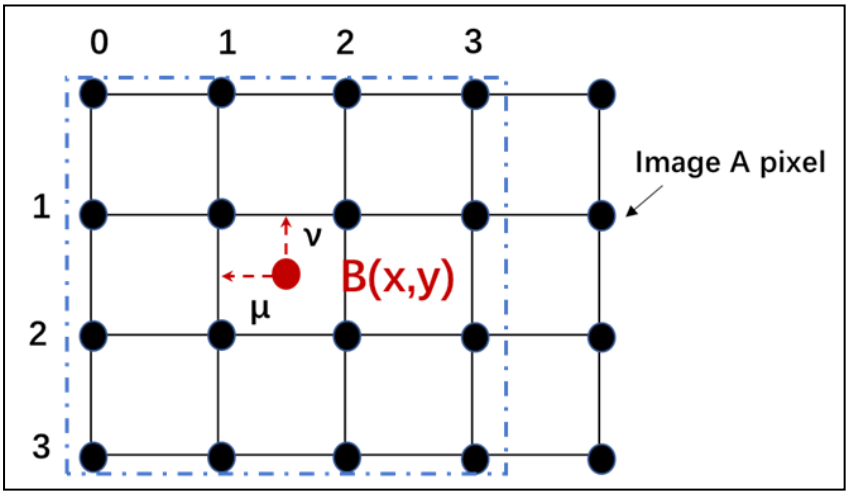
$$p(x,y) = \begin{bmatrix} 1 & x & x^2 & x^3 \end{bmatrix} \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} 1 \\ y \\ y^2 \\ y^3 \end{bmatrix}$$

Bicubic interpolation on an arbitrarily sized regular grid can then be accomplished by patching together such bicubic surfaces.

The interpolation kernel for bicubic interpolation is

$$u(x) = \begin{cases} (a+2)|x|^3 - (a+3)|x|^2 + 1, & 0 <= |x| < 1 \\ a|x|^3 - 5a|x|^2 + 8a|x| - 4a, & 1 <= |x| < 2 \\ 0, & \text{other} \end{cases}$$

Where x signifies the distance between interpolated point and grid point & a = free variable (for good effect $(-1, \frac{1}{2})$)



Bicubic Interpolation algorithm being implemented to find the pixel B(x, y), the immediate 4x4 neighbourhood around the pixel B is considered as shown by the dotted lines.

In image processing, bicubic interpolation is often chosen over bilinear or nearest-neighbor interpolation in image resampling, when speed is not an issue. The reason behind this is that Bicubic produces noticeably sharper images than the previous two methods, and is perhaps the ideal combination of processing time and output quality. For this reason it is a standard in many image editing programs including Adobe Photoshop, printer drivers and in-camera interpolation

Simulation

For the simulation aspect of this project we have designed 3 function files using MATLAB that each carry out nearest neighbour, bilinear, and bicubic interpolation algorithms on any specific image respectively. The 3 function files have been named as follows:-

i) Nearest_Neighbour_Interpolation.m

ii) Bilinear_Interpolation.m

iii) Bicubic_m2.m

The codes for the 3 interpolation algorithm functions were designed in MATLAB & each of them utilize the comprehensive set of crucial functions offered by the Image Processing Toolbox™ such as 'imread()' function.

All 3 function files were designed on the basis of the algorithms as described in the 'Image Interpolation Algorithms' section [see pages 11-18]. The codes of the 3 function files along with some examples of the implementations of these image interpolation algorithms using images provided by the Image Processing Toolbox™ have been illustrated in this section.

(Note: All the relevant information regarding how to use the function files have been added as comments into the function files & may be accessed by any user by typing in the prompt 'help' followed by the function file name into the MATLAB command window)

Nearest_Neighbour_Interpolation.m

The code for the Nearest Neighbour Interpolation algorithm is a relatively simple one as compared to the other interpolation algorithms covered under this project. The implementation of this algorithm was fairly simple. The code for the Nearest_Neighbour_Interpolation function file is displayed below as:-

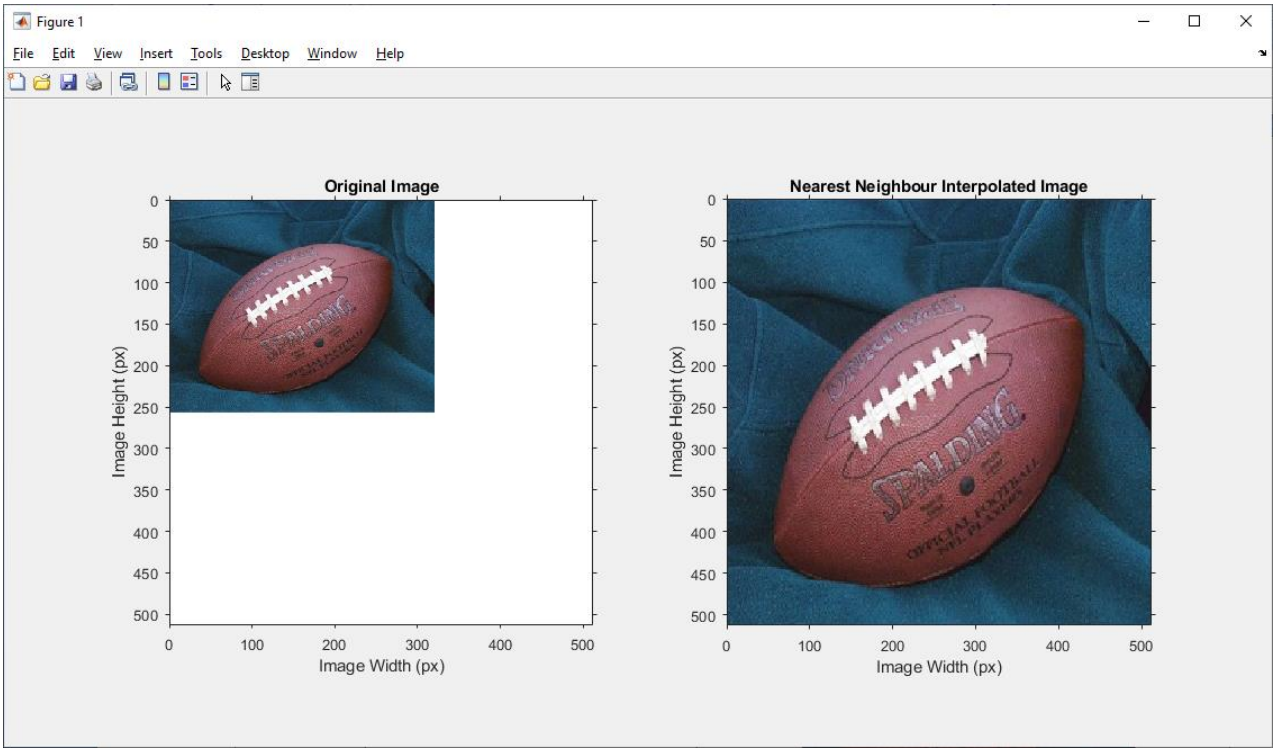
```
function [Output] = Nearest_Neighbour_Interpolation(im, out_dims)

%This function given the Inputs 'A' & 'out_dims' performs Nearest Neighbour
%Interpolation & zooms into or out of based on the input 'out_dims'. Input
%'A' must be a image stored as a matrix using preferably the 'imread'
%function. %'out_dims' variable must be defined as a [1x2] matrix with the
%two elements of this matrix defining the new height & the new width of the
%desired image, respectively.
%Note that this function only interpolates images that are in RGB mode, so
%certain images such as .tif format images do not work with this function

% Obtain the no. of Rows/Columns (or Height/Width) of the desired
% interpolated image
Row = out_dims(1);
Col = out_dims(2);

% Computing the Ratio's of the New Size to the Old Size
rtR = Row/size(im,1);
rtC = Col/size(im,2);

% Obtain the Interpolated Positions
IR = ceil([1:(size(im,1)*rtR)]./(rtR));
IC = ceil([1:(size(im,2)*rtC)]./(rtC));

% Now each channel Red, Green, & Blue of the RGB image are interpolated
% seperately as done below.

% RED CHANNEL
Temp= im(:,:,1);
% Row-Wise Interpolation
Red = Temp(IR,:);
% Columnwise Interpolation
Red = Red(:,IC);


% GREEN CHANNEL
Temp= im(:,:,2);
% Row-Wise Interpolation
Green = Temp(IR,:);
% Columnwise Interpolation
Green = Green(:,IC);


% BLUE CHANNEL
Temp= im(:,:,3);
% Row-Wise Interpolation
Blue = Temp(IR,:);
% Columnwise Interpolation
Blue = Blue(:,IC);

% The interpolated values are initialized back into the target 'output'
% variable where we want to obtain the Nearest neighbourhood interpolated
% image.
Output=zeros([Row,Col,3]);
```

```
Output(:,:,1)=Red;
Output(:,:,2)=Green;
Output(:,:,3)=Blue;

%uint8 function used to turn variable 'Output' into uint8 format
Output = uint8(Output);

end
```

This function file may be used with only a few short lines of additional code as follows:

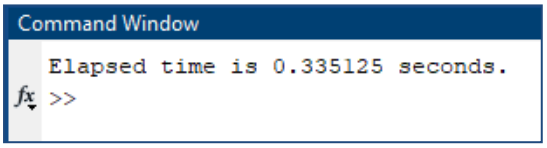
```
clc;
clear all;
tic

% Store any RGB image as a matrix in A
% Works with the following images:- onion.png, football.jpg, greens,jpg,
% hestain.png, office_3.jpg, & fabric.png
im=imread('football.jpg');

% Define the desired Re-sample size
out_dims=[512 512];

% Obtain the Nearest Neighbour Interpolated Image using the function
'Nearest_Neighbour_Interpolation'
Interpolated_Img = Nearest_Neighbour_Interpolation(im, out_dims);

% Plotting Original Image (Before Interpolation)
figure;
subplot(121)
imshow(im);
title('Original Image');
ylabel('Image Height (px)')
xlabel('Image Width (px)')
axis([0,512,0,512]);
axis on;

% Plotting the Nearest Neighbour Interpolated Image (After Interpolation)
subplot(122)
imshow(Interpolated_Img)
title('Nearest Neighbour Interpolated Image');
ylabel('Image Height (px)')
xlabel('Image Width (px)')
axis([0,512,0,512]);
axis on;

toc
```

The output obtained upon executing this code is as follows:-



From afar the nearest neighbour interpolated image as seen above might look decently clear; however this is due to the limitation of the fact that we had to resize the original screenshot of the output figure window to make it fit into this report. Upon closer inspection, noticeable pixelation & jagged artifacts may be observed.

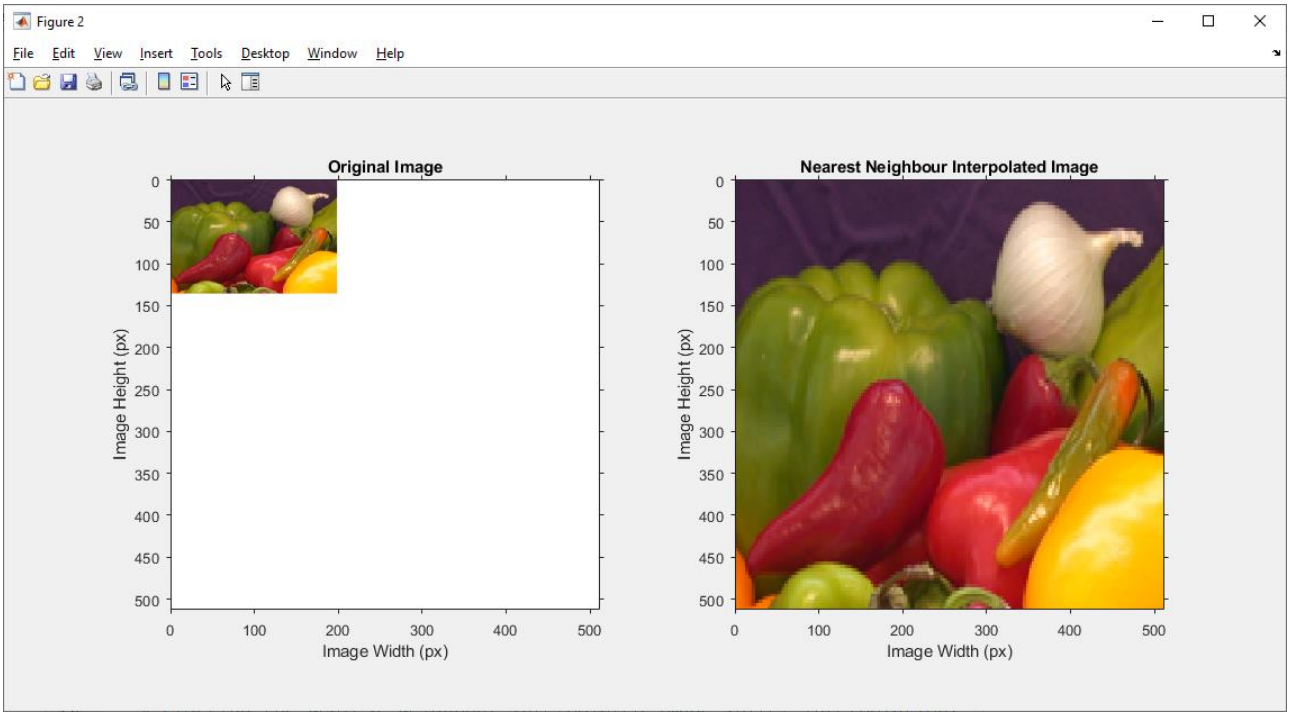
However computationally speaking the nearest neighbour interpolation method took the least amount of time out of the other interpolation algorithms. This is verified by the 'tic toc' function, which gives the following output:



(Note: This output may differ from system to system depending on factors such as the processor speed & RAM available)

Thus at the expense of heavy pixelization in images, nearest neighbour interpolation carries out the fastest resampling/resizing of images.

Another example of the nearest neighbour interpolation function being used to resample an image is shown below. Here, the image being interpolated is 'onion.png' which is another image provided in the Image Processing Toolbox™.



'onion.png' is just one of the many pre-loaded demo images offered by the MATLAB Image Processing Toolbox™

Bilinear_Interpolation.m

Bilinear _Interpolation.m carries out bilinear interpolation on any given image based on the desired dimensions specified by the user in the input variable 'out_dims'. All the relevant information regarding how to use the function may be accessed by the user by utilizing the 'help' function.

The code of the function file 'Bilinear_Interpolation.m' is given below:

```
    function [out] = Bilinear_Interpolation(im, out_dims)
%This function given the inputs 'im' & 'out_dims' carries out bilinear
```

```
%interpolation on the image stored in variable 'im'. The output image will
%either be zoomed into or zoomed out of based on the dimensions specified
%by the user in the input variable 'out_dims'.
%'out_dims' variable must be defined as a [1x2] matrix with the
%two elements of this matrix defining the new height & the new width of the
%desired image, respectively.

%Note that this interpolation algorithm has been configured to work with
%RGB images, attempting to use this function with tagged image file
%formats such as .tif might lead to ustable results.

    % Obtaining required variables
    in_rows = size(im,1); %No. of Rows of the Image
    in_cols = size(im,2); %No. of columns of the Image
    out_rows = out_dims(1); %No. of Rows desired in Interpolated Image
    out_cols = out_dims(2); %No. of columns desired in Interpolated Image

    % Let S_R = R / R'
    S_R = in_rows / out_rows; %Ratio of Rows of original Image:Rows of
Interpolated Image
    % Let S_C = C / C'
    S_C = in_cols / out_cols; %Ratio of Columns of original Image: Columns of
Interpolated Image

    % Define grid of co-ordinates in our image
    % Generate (x,y) pairs for each point in our image
    % Creating a meshgrid from the desired dimensions of the interpolated
    % image
    [cf, rf] = meshgrid(1 : out_cols, 1 : out_rows);

    % Let r_f = r'*S_R for r = 1,...,R'
    % Let c_f = c'*S_C for c = 1,...,C'
    rf = rf * S_R;
    cf = cf * S_C;

    % Let r = floor(rf) and c = floor(cf)
    % Floor used to rounds the elements to the nearest integers less than
    % or equal to rf/cf
    r = floor(rf);
    c = floor(cf);

    % Any values out of range, cap
    % Since we dont want any 2D grid coordinates to extend beyond orginal
    % image dim.
    r(r < 1) = 1;
    c(c < 1) = 1;
    r(r > in_rows - 1) = in_rows - 1;
    c(c > in_cols - 1) = in_cols - 1;

    % Let delta_R = rf - r and delta_C = cf - c
    delta_R = rf - r; %// Unfloored Values - Floored & Capped Values
    delta_C = cf - c;

    % Final line of algorithm
    % Get column major indices for each point we wish
    % to access
    % The sub2ind command determines the equivalent single index
```

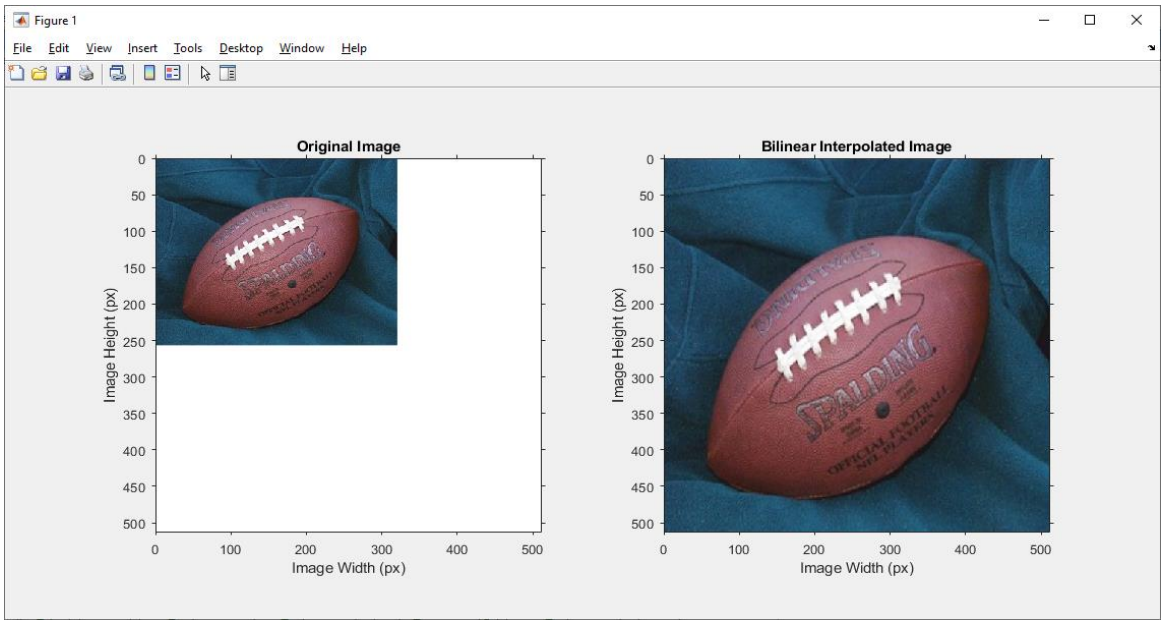
```
    % corresponding to a set of subscript values. IND = sub2ind(siz,I,J)
    % returns the linear index equivalent to the row and column subscripts
    % I and J for a matrix of size siz.
    in1_ind = sub2ind([in_rows, in_cols], r, c);
    in2_ind = sub2ind([in_rows, in_cols], r+1,c);
    in3_ind = sub2ind([in_rows, in_cols], r, c+1);
    in4_ind = sub2ind([in_rows, in_cols], r+1, c+1);

    % Now interpolation process begins
    % Go through each channel for the case of 3 different colour channels
    % Create output image that is the same class as input
    out = zeros(out rows, out cols, size(im, 3));
    out = cast(out, class(im)); %The cast function converts out to the data
type of im (data type of im= class(im))

    for idx = 1 : size(im, 3) % size(im, 3)=3 since RGB has 3 corresponding
modes/channels
        chan = double(im(:,:,idx)); % Get i'th channel (This gives the matrix
corresponding to R,G & B channels)
        % Also, double here is used to convert to double precision
% Interpolate the channel
        tmp = chan(in1_ind).*(1 - delta_R).*(1 - delta_C) + ...
                      chan(in2_ind).*(delta_R).*(1 - delta_C) + ...
                      chan(in3_ind).*(1 - delta_R).*(delta_C) + ...
                      chan(in4_ind).*(delta_R).*(delta_C);
        out(:,:,idx) = cast(tmp, class(im));
    end
```

Once again, implementing this function in a program to carry out Bilinear_interpolation is relatively simple & achieved using just a few additional lines of code as follows:-

```
clc;
clear all;
tic

% Store any RGB image as a matrix in 'im'
% Works with the following images:- onion.png, football.jpg, cameraman.tif,
% hestain.png, coins.png, office_3.jpg, & fabric.png
im = imread('football.jpg');

% Define the desired Re-sample size
out_dims = [512 512];

% Obtain the Bilinear Interpolated Image using the function
'Bilinear_Interpolation'
out = Bilinear_Interpolation(im, out_dims);

% Plotting Original Image (Before Interpolation)
figure;
```

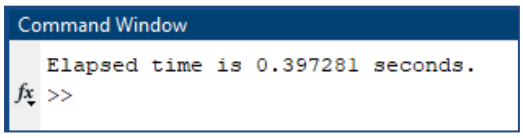
```
subplot(1,2,1)
imshow(im);
title('Original Image');
ylabel('Image Height (px)')
xlabel('Image Width (px)')
axis([0,512,0,512]);
axis on;

% Plotting the Bilinearly Interpolated Image (After Interpolation)
subplot(1,2,2)
imshow(out);
title('Nearest Bilinear Interpolated Image');
ylabel('Image Height (px)')
xlabel('Image Width (px)')
axis([0,512,0,512]);
axis on;

toc
```

The output obtained upon executing this code is as follows:-



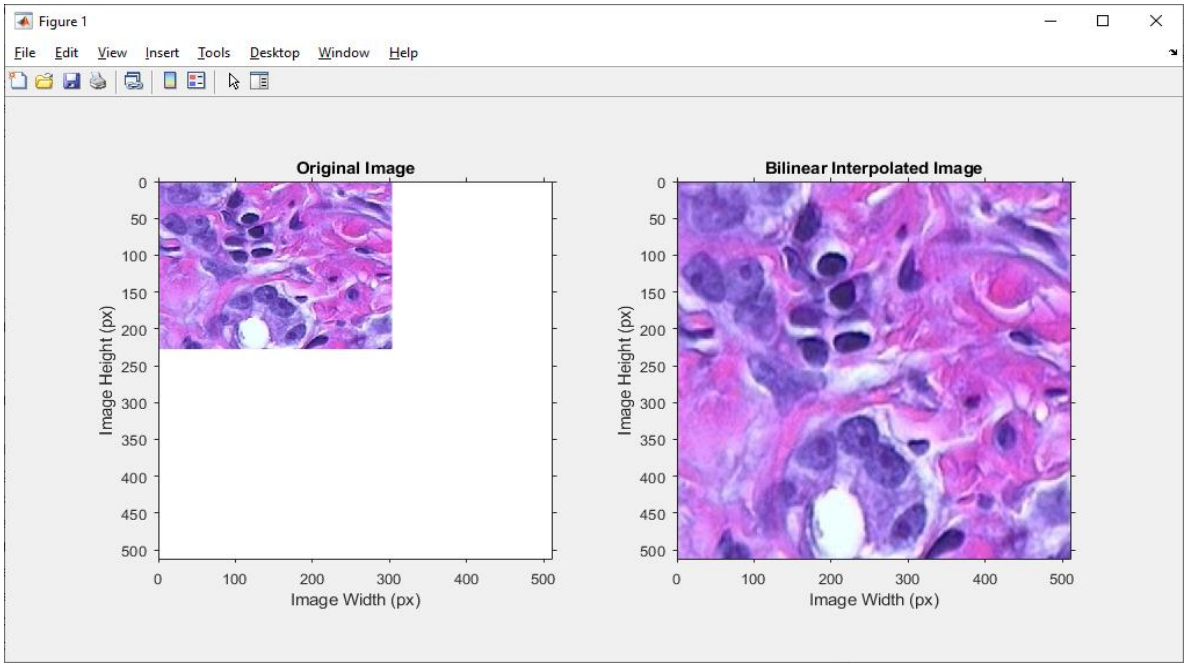
The overall image produced after the bilinear interpolation algorithm on ‘football.jpg’ is less grainy & pixelated as compared to the nearest neighbour interpolation, however since it required to actually interpolate intermediate pixel values (contrary to nearest neighbour which simply selected values of nearest neighbour pixels) the computational time was also higher relative to the nearest neighbour interpolation method.

The tic toc function used yielded the following output (shown below) confirming that while it produces more detailed & clearer images, the trade-off with the bilinear interpolation method is the increased computational time.



However, since the bilinear interpolation algorithm produces reasonably clear & detailed images while taking a suitable amount of computational time, it is considered to be perhaps the most efficient interpolation algorithm as unlike the nearest neighbour method which is incapable of preserving image quality & unlike the bicubic method which takes up the highest computational time, Bilinear interpolation attempts to find a reasonable balance between both image quality & processing time without sacrificing one or the other.

Another example of the Bilinear_Interpolation.m function being utilized to resample a different image is shown below. The image being resampled in the example below is 'hestain.png'.



Bicubic_Interpolation.m

The Bicubic_Interpolation.m function file resamples/resizes images using the bicubic interpolation algorithm. The help function provides users with all the relevant information required to use this function file. The implementation of this algorithm was perhaps the most complex out of all the 3 interpolation algorithms. The code of the function file is given below:

```
function im_zoom = Bicubic_interpolation(image, zoom);

%This function given the inputs 'image' & 'zoom' carries out bicubic
%interpolation on the image stored in variable 'image'. The output image will
%either be zoomed into or zoomed out of based on the magnification factor
%input into the variable 'zoom'
%'zoom' variable must be defined as a integer value, which defines the
%desired magnification factor of the image. (Preferably in the range 1-10
%as large magnification factors might cause low end systems to crash)

%Note that this function has been configured to interpolate images encoded
%in RGB mode. Attempting to interpolate gray-scale, binary or .tif file
%formats might lead to unstable outputs.

% Obtaining some required variables from the input image stored in 'image'
[r c d] = size(image);
rn = floor(zoom*r); % Multiplying the rows of img by zoom factor
cn = floor(zoom*c); % Multiplying the columns of img by zoom factor
s = zoom;
% Cast function used to convert zeroes(rn,cn,d) to the data class type int8
im_zoom = cast(zeros(rn,cn,d),'uint8');
im_pad = zeros(r+4,c+4,d);
im_pad(2:r+1,2:c+1,:) = image;
im_pad = cast(im_pad,'double');
% For loop used to begin mapping through each one of the 16 pixels in the
% [4 x 4] environment around target pixel. For more info behind the
% algorithm being implememted take a look at the bicubic interpolation
% algorithm explained in the previous sections
for m = 1:rn
    x1 = ceil(m/s); x2 = x1+1; x3 = x2+1;
    p = cast(x1,'uint16');
    if(s>1)
       m1 = ceil(s*(x1-1));
       m2 = ceil(s*(x1));
       m3 = ceil(s*(x2));
       m4 = ceil(s*(x3));
    else
       m1 = (s*(x1-1));
       m2 = (s*(x1));
       m3 = (s*(x2));
       m4 = (s*(x3));
    end
    X = [ (m-m2)*(m-m3)*(m-m4)/((m1-m2)*(m1-m3)*(m1-m4)) ...
```

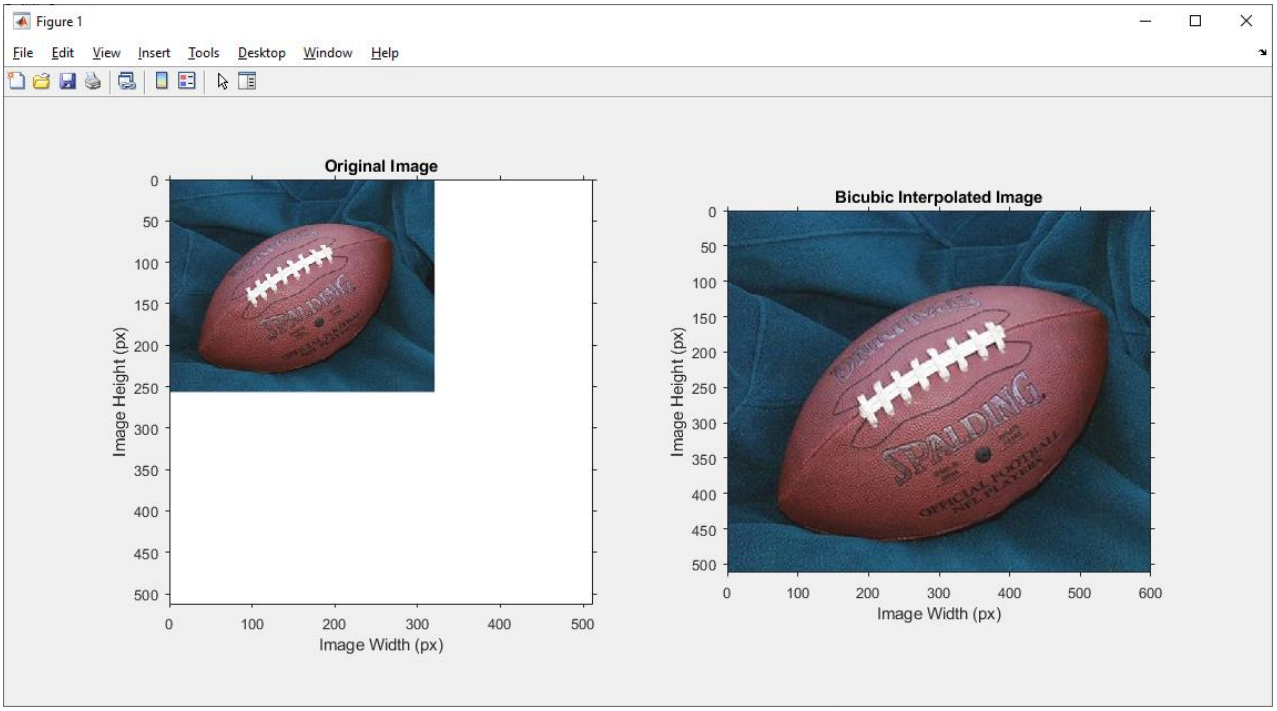
```
           (m-m1)*(m-m3)*(m-m4)/((m2-m1)*(m2-m3)*(m2-m4)) ...
           (m-m1)*(m-m2)*(m-m4)/((m3-m1)*(m3-m2)*(m3-m4)) ...
           (m-m1)*(m-m2)*(m-m3)/((m4-m1)*(m4-m2)*(m4-m3))];
    for n = 1:cn
        y1 = ceil(n/s); y2 = y1+1; y3 = y2+1;
        if (s>1)
           n1 = ceil(s*(y1-1));
           n2 = ceil(s*(y1));
           n3 = ceil(s*(y2));
           n4 = ceil(s*(y3));
        else
           n1 = (s*(y1-1));
           n2 = (s*(y1));
           n3 = (s*(y2));
           n4 = (s*(y3));
        end
        Y = [ (n-n2)*(n-n3)*(n-n4)/((n1-n2)*(n1-n3)*(n1-n4));...
              (n-n1)*(n-n3)*(n-n4)/((n2-n1)*(n2-n3)*(n2-n4));...
              (n-n1)*(n-n2)*(n-n4)/((n3-n1)*(n3-n2)*(n3-n4));...
              (n-n1)*(n-n2)*(n-n3)/((n4-n1)*(n4-n2)*(n4-n3))];
        q = cast(y1,'uint16');  % Cast function used to convert y1 to the
data class type uint8
        sample = im_pad(p:p+3,q:q+3,:); % p & q are basically x1 & y1 cast
into uint8 data type
        im_zoom(m,n,1) = X*sample(:,:,1)*Y;
        if(d~=1)
              im_zoom(m,n,2) = X*sample(:,:,2)*Y;
              im_zoom(m,n,3) = X*sample(:,:,3)*Y;
        end
    end
end
% Cast function used here to convert im_zoom to the data class type int8
im_zoom = cast(im_zoom,'uint8'); % im_zoom is the required bicubic
interpolated image
```

An example of this function being utilized to resample the image ‘football.jpg’ is shown below:

```
clc;
clear all;
tic

% Store any RGB image as a matrix in variable 'image'
% Works with the following images:- onion.png, football.jpg, cameraman.tif,
% hestain.png, coins.png, office_3.jpg, & fabric.png
image = imread('football.jpg');

% Define the desired magnification factor in variable im_zoom
im_zoom = Bicubic_Interpolation(image,2);
```

```
% Plotting Original Image (Before Interpolation)
figure;
subplot(1,2,1)
imshow(image);
title('Original Image');
ylabel('Image Height (px)')
xlabel('Image Width (px)')
axis([0,512,0,512]);
axis on;

% Plotting the Bilinearly Interpolated Image (After Interpolation)
subplot(1,2,2)
imshow(im_zoom);
title('Bicubic Interpolated Image');
ylabel('Image Height (px)')
xlabel('Image Width (px)')
axis([0,512,0,512]);
axis on;

toc
```

The output obtained upon executing this code is as follows:-



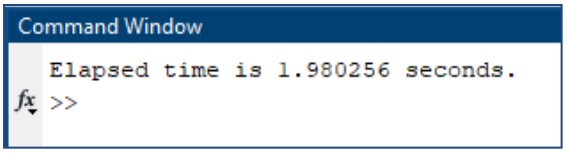
As clearly visible from the output figure window, by far the best image quality & shapness out of all the interpolation algorithms so far is seen in the bicubic

interpolated image. There is almost noticeably no pixelation or jagged artifacts in the final interpolated image.

This undoubtedly proves the bicubic interpolation algorithm as the most ideal non-adaptive image processing algorithm in terms of image output quality. For this reason it is a standard in many image editing programs including Adobe Photoshop, printer drivers and in-camera interpolation.

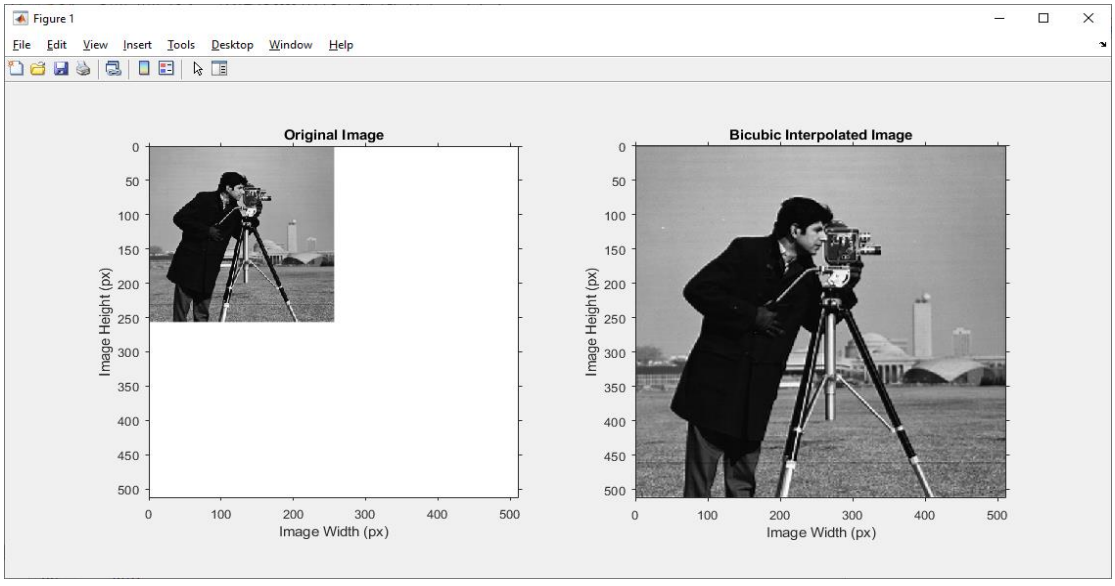
The computational time of the bicubic is also the highest out of the 3 algorithms, however the increase in image pixel quality & image clarity are the reason that bicubic is still the most widely preferred image interpolation.

The 'tic toc' function used, yields the following output:



As expected the computational time of the Bicubic_Interpolation.m function required the highest computational time attributed to the complex structure of the bicubic interpolation algorithm.

Another example of the Bicubic_Interpolation.m function being used with a different image is shown below for the sake of comaprision. The image being interpolated in this case is 'cameraman.tif'



Comparison between the 3 Interpolation Algorithms

A comparison of the time complexity of all three image interpolation algorithms is shown in the image below. Computationally, nearest neighbour interpolation is the least intensive as it considers only one pixel for interpolation. Meanwhile Bicubic is the most time complex since it considers a relatively large [4 x 4] pixel neighbourhood around the point being interpolated.

The computation times are tabulated based on the output recieved upon using the 'tic toc' function to calculate & display the elapsed time.

Interpolation Algorithm	Computation Time (in seconds)
Nearest Neighbour	0.335125
Bilinear	0.397281
Bicubic	1.980256

Thus based on the quality of the final resampled image after interpolation & the time taken by the interpolation algorithms to carry out the interpolation process, one may determine that each method has its own pros and cons. However based on the visual and mathematical analysis, the consensus points to the Bicubic as being the best interpolation method, where the image produced is comparable to the original & has high image quality. For this reason it is a standard in many image editing programs including Adobe Photoshop, printer drivers and in-camera interpolation & even in MATLAB.

Conclusion

To conclude, through this project we were able to learn about the different types of non adaptive image interpolation algorithms involved in image processing in today's world. We were able to grasp the important role that image interpolation algorithms play as a fundamental processing task in numerous computer graphics based applications.

Through simulating different image interpolation algorithms we were able to gain an insight into the utility and resourcefulness of software's such as MATLAB in image processing applications. The Image Processing Toolbox™ offered by MATLAB allowed us to develop an efficient implementation of the interpolation algorithms.

Once again we would like to thank our professor Dr. Ajeet Kumar who gave us the opportunity to work on this project & further our knowledge on the topic to its current state.

References

The references & resources used in this project have been cited below:

R. C. Gonzalez, Digital image processing. Pearson Education India; 2009.

W. Zhe, Z. Jiefu and Z. Mengchu, "A Fast Autoregression Based Image Interpolation Method", in Proc. of the IEEE International Conference on Networking, Sensing and Control, pp.1400-1404, 2008

Gonzalez, R. C., Woods, R. E. and Eddins, S. L. [2009]. Digital Image Processing using MATLAB, 2nd ed., Gatesmark Publishing, www.gatesmark.com.

V. Patel and K. Mistree, "A Review on Different Image Interpolation Techniques for Image Enhancement", International Journal of Emerging Technology and Advanced Engineering, Vol. 3, pp. 129-13, Dec. 2013.

S.D. Ruikar and D.D. Doye, "Image Denoising using Tri Nonlinear and Nearest Neighbour Interpolation with Wavelet Transform", International Journal of Information Technology and Computer Science, Vol.4, pp. 36-44, Aug.2012.

R. Keys (1981). "Cubic convolution interpolation for digital image processing".

Links:

https://en.wikipedia.org/wiki/Digital_image_processing

https://www.cambridgeincolour.com/tutorials/image-interpolation.htm

https://www.sciencedirect.com/topics/engineering/image-interpolation

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

https://core.ac.uk/download/pdf/25523628.pdf

http://scholarworks.csun.edu/bitstream/handle/10211.2/1548/Aaron_Santee_GradProj_Final.pdf?sequence=1

https://ijarcce.com/wp-content/uploads/2016/02/IJARCCE-7.pdf

https://www.researchgate.net/publication/282449606_A_Review_Image_Interpolation_Techniques_for_Image_Scaling

https://in.mathworks.com/products/image.html