IMAGE INTERPOLATION
(Resizing using bilinear method)

Introduction

Bilinear interpolation considers the *closest 2x2 neighborhood* of known pixel values surrounding the unknown pixel. It then takes a *weighted average* of these 4 pixels to arrive at its *final interpolated* value. This results in much smoother looking images than the *nearest neighbor*. N*earest neighbor* is another method where we take values according to the *nearest pixel*.

Algorithm

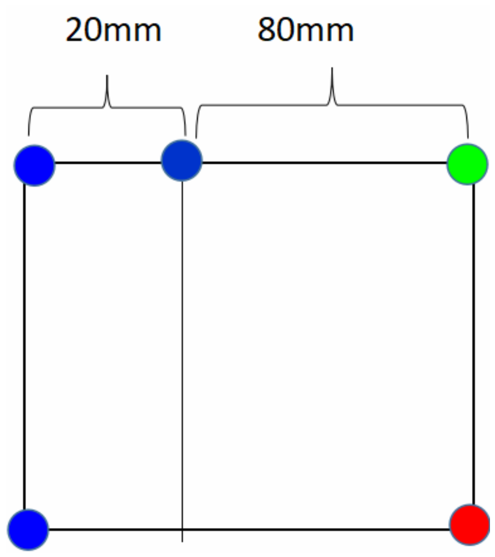
Iterate over all points in the source image. For each pixel:

1. Compute the *weighted average* color with respect to the distance to the top-left and top-right pixel.
2. Compute the weighted average color with respect to the distance to the bottom-left and bottom-right pixel.
3. Compute the weighted average color with respect to the y-distance of the two generated colors.

Explanation

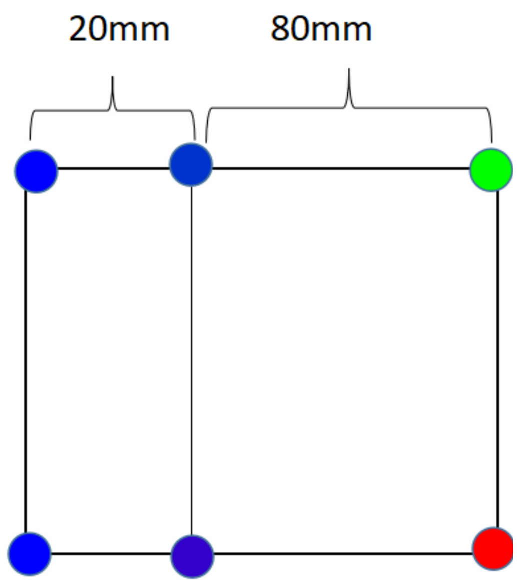
Step 1: Compute the upper weighted average

Assume the RGB colors are:
Left rgb(0,0,255)
Right rgb(0,255,0)
We have rgb(0,(0*80+255*20)/100,(255*80+0*20)/100) = rgb(0,51,204)



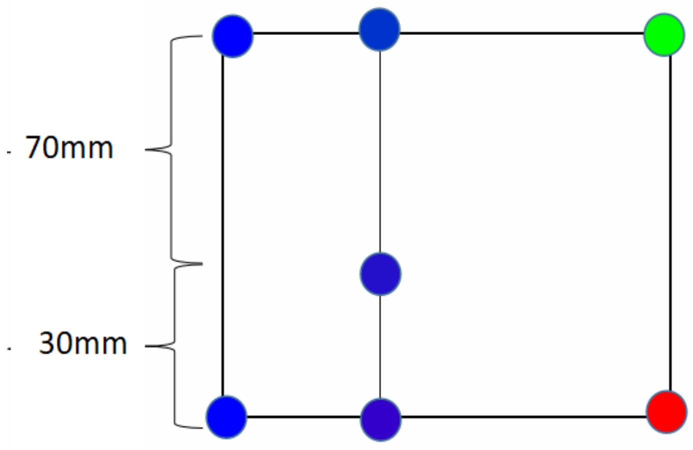
Step 2: Compute the lower weighted average

Assume the RGB colors are:
Left rgb(0,0,255)
Right rgb(255,0,0)
We have rgb((0*80+255*20)/100,0,(255*80+0*20)/100) = rgb(51,0,204)



Step 3: Compute the vertical weighted average

Assume the RGB colors are:
Top rgb(0,51,204)
Bottom rgb(51,0,204)
We have rgb((51*70)/100,(51*30)/100,204) = rgb(35.7,15.3,204)



Serially Bilinear Interpolation Implementation :

```
// x_scale ,y_scale are  scaling factor
void Inter_Linear(Mat& original, Mat& interpolated, double x_scale, double y_scale )
{
   // new dimensions for interpolated image
   int interpolated_rows = round( x_scale * original.rows );
   int interpolated_cols = round( y_scale * original.cols );
   //Creating empty Mat of new scale size
   interpolated = Mat(interpolated_rows, interpolated_cols, original.type());

   for (int i = 0; i < interpolated.rows; i++)
   {
      //geometric center alignment
      double index_i = (i + 0.5 ) / x_scale + 0.5;

      //prevent cross-border
         index_i = index_i < 0 ? 0 : index_i ;
         index_i = index_i > original.rows - 1 ? original.rows - 1 : index_i ;

      //adjacent 4*4 pixel rows (coordinates)
      int i1 = floor(index_i);
      int i2 = ceil(index_i);

      //u is the decimal part of the floating-point coordinate line
      double u = index_i - i1;
      for (int j = 0; j < interpolated.cols; j++)
      {
         //Geometric center alignment
         double index_j = ( j + 0.5 ) / y_scale + 0.5 ;

         //Prevent cross-border
         Index_j = index_j < 0 ? 0 : index_j ;
         Index_j = index_j > original.cols - 1 ? original.cols - 1 : index_j ;

         //Columns of adjacent 4*4 pixels (coordinates)
         int j1 = floor(index_j);
         int j2 = ceil(index_j);

         //v is the decimal part of the floating-point coordinate column
         double v = index_j - j1;
```

```
  // Using I(x,y) = (1-a)(1-b)I(x’,y’) + (1-a)(b)I(x’,y’+1) + (a)(1-b)I(x+1’,y’) + (1-a)(1-b)I(x’+1,y’+1)
          interpolated.at<Vec3b>(i, j)[0] = (1 - u) * (1 - v) * original.at<Vec3b>(i1, j1)[0] + (1 - u) * v *
original.at<Vec3b>(i1, j2)[0] + u * (1 - v) * original.at<Vec3b>(i2, j1)[0] + u * v * original.at<Vec3b>(i2, j2)[0];
          interpolated.at<Vec3b>(i, j)[1] = (1 - u) * (1 - v) * original.at<Vec3b>(i1, j1)[1] + (1 - u) * v *
original.at<Vec3b>(i1, j2)[1] + u * (1 - v) * original.at<Vec3b>(i2, j1)[1] + u * v * original.at<Vec3b>(i2, j2)[1];
          interpolated.at<Vec3b>(i, j)[2] = (1 - u) * (1 - v) * original.at<Vec3b>(i1, j1)[2] + (1 - u) * v *
original.at<Vec3b>(i1, j2)[2] + u * (1 - v) * original.at<Vec3b>(i2, j1)[2] + u * v * original.at<Vec3b>(i2, j2)[2];
// Mat is for colored image so, calculating it for 0,1,2 index
       }
   }
}
```

** code is implemented in bilinear_interpolation.exe file attached with pdf.
** opencv is required .

Why to do Bilinear Interpolation Parallely ?

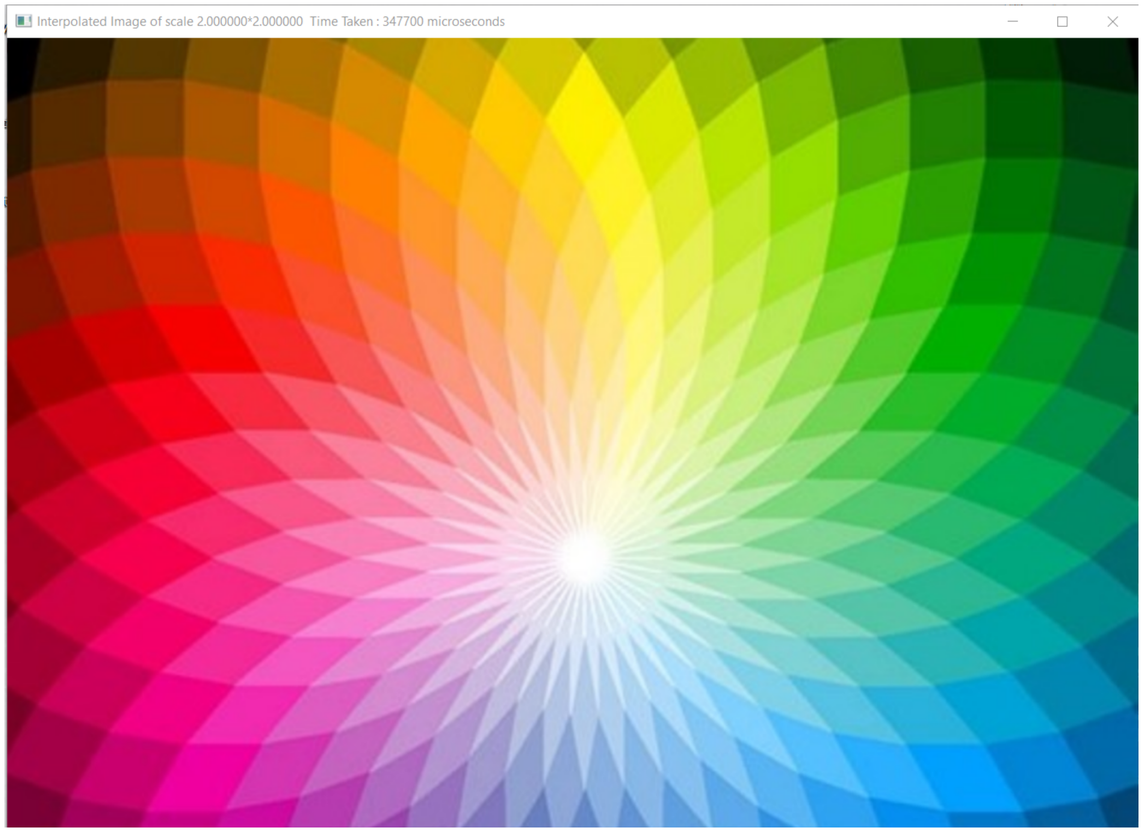
This method of bilinear interpolation takes a longer time when the interpolation scalar is large or the grid size is large. Bilinear interpolation can be approximated as the extension of the linear interpolation function with two variables, which carry out a linear interpolation in two directions of x, y . This can be reduced to 10 times using a parallel method of interpolation.

Test cases :

Converting image by scale 2*2 takes 0.3 sec
Converting image by scale 10*10 takes 8.4 sec

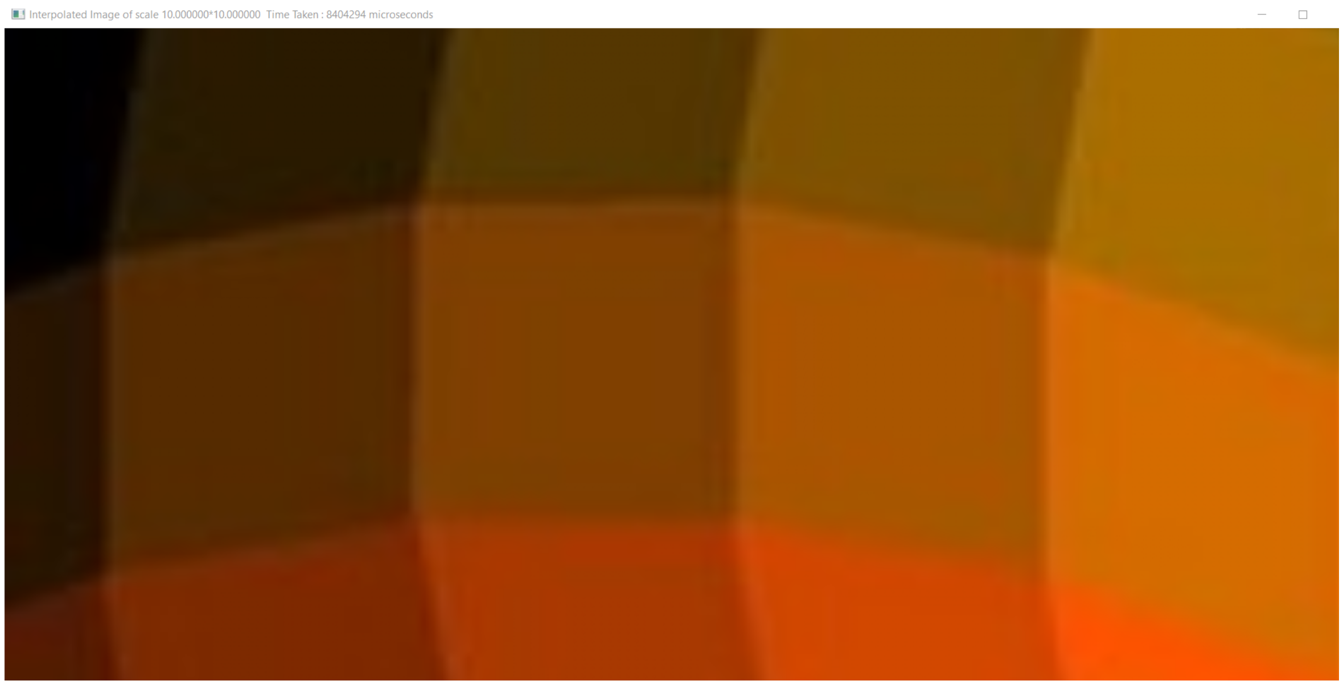
Output from bilinear_interpolation.exe is :

Test case 1 : 5 times



Interpolated Image of scale 2.000000*2.000000 Time Taken : 347700 microseconds

Test case 2 : 10 times



Interpolated Image of scale 10.000000*10.000000 Time Taken : 8404294 microseconds