Image Processing

Project

Zooming and shrinking in digital images

Chereji Iulia

30434

2022

1. Project Description

The project consist in implementing different algorithms for zooming and shrinking digital images (grayscale with 8 bits/pixel). The advantages and disadvantages of each algorithm will be evaluated.

1. Algorithms

The 3 algorithms that were implemented are: nearest neighbor interpolation, bilinear interpolation and bicubic interpolation. They all consist of creating a new image with the appropriate dimensions and iterating on its pixels. Then for each pixel’s coordinates the coordinates in the original image are computed. They will be real numbers (not integers pointing exactly to original pixels), and each of the algorithms try a different approach to compute the new pixel’s gray level of intensity based on its neighbors in the original image.

2.1. Nearest Neighbor Interpolation

We have the original images of dimensions lets say 3x4. And we want to make it 2.5 times larger. This mans a new image of 7.5x10 rounded => 8x10. After we read the image (src) and the scale (s) we create the output image (dest) and iterate on it:

For i=0:dest.y

For j:dest:x

dest(i,j)=src(round(i/s),round(j/s))

This also works for making an image smaller (scale < 0).

In this approach we just take the value of the closest pixel using the round function. This should be very fast.

2.2. Bilinear Interpolation

It uses linear interpolation to calculate the values between the neighboring pixels’ colors.

For i=0:dest.y

For j:dest:x

isrc=i/s, jsrc=j/s

ifloor=floor(isrc), iceil=ceil(isrc), jfloor=floor(jsrc), jceil=ceil(jsrc)

ul=src(iceil,jceil), ur=src(ifloor,jceil), dl=src(iceil,jfloor),

dr=src(iceil,jceil); //u-up,d-down,l-left,r-right

Calculate the interpolated values on the horizontal lines first:

u=ul\*(jceil-jsrc) + ur\*(jsrc-jfloor)

d=dl\*(jceil-jsrc) + dr\*(jsrc-jfloor)

val=u\*(iceil-isrc)+d\*(isrc-ifloor)

dest(i,j)=val

We also need to check if the values for isrc and jsrc are integers, in which case we don’t need to interpolate, we just assign the value.

This algorithm uses 4 nearest neighbors to determine the output.

Basically pixels get weights in the final computation of the color based on how close they are to the wanted position (which is a real number).

2.3. Bicubic Interpolation

Bicubic uses 16 neighbors and weights distribution is done differently.

For example we take an image of height 241 and width 218 and the scale 1.5 => height 362, width 327

We iterate on the new bigger image and reach i=8, j=242. We compute the x and y equivalent in the original image x= 161.33 y=5.33.

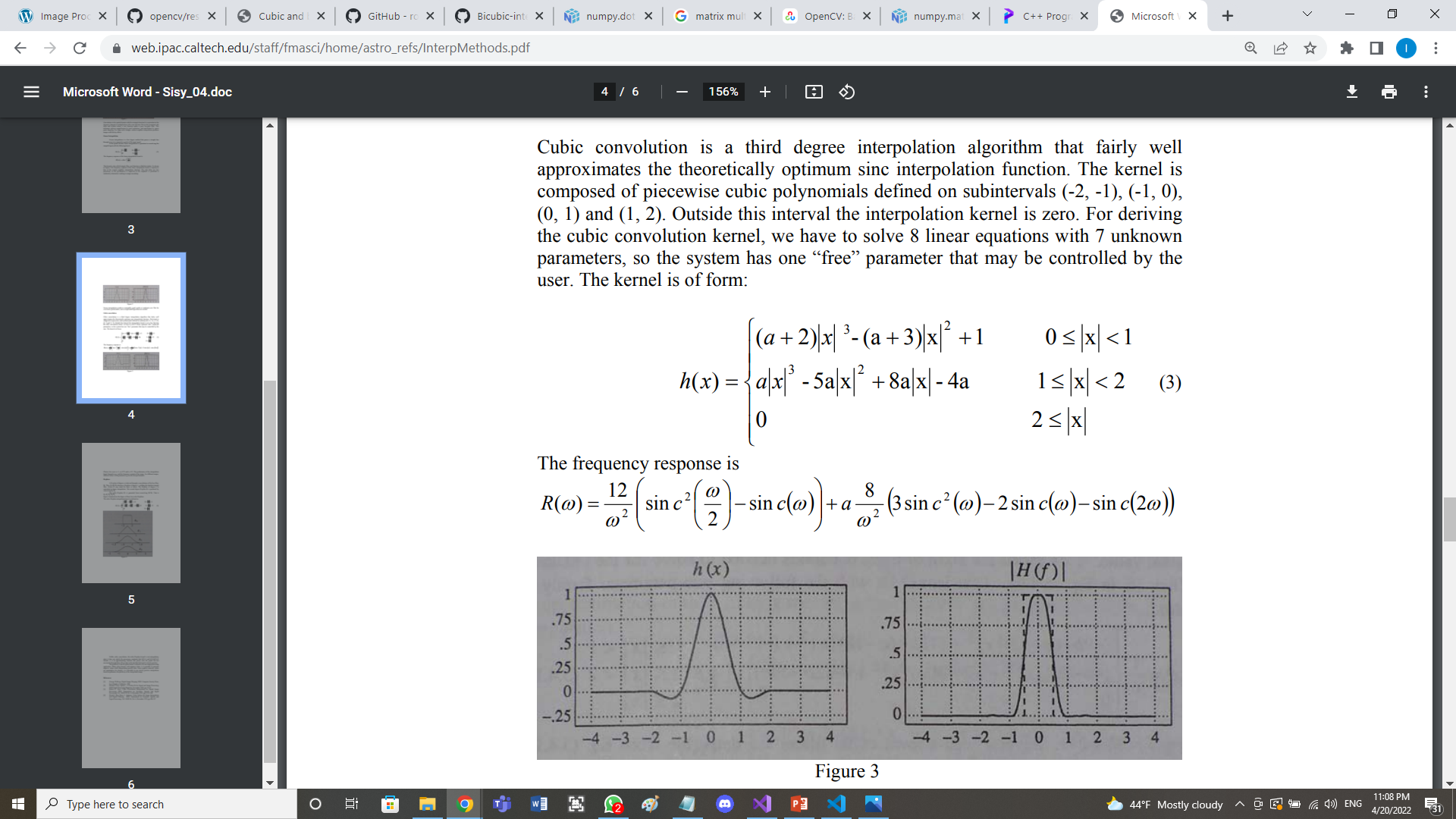
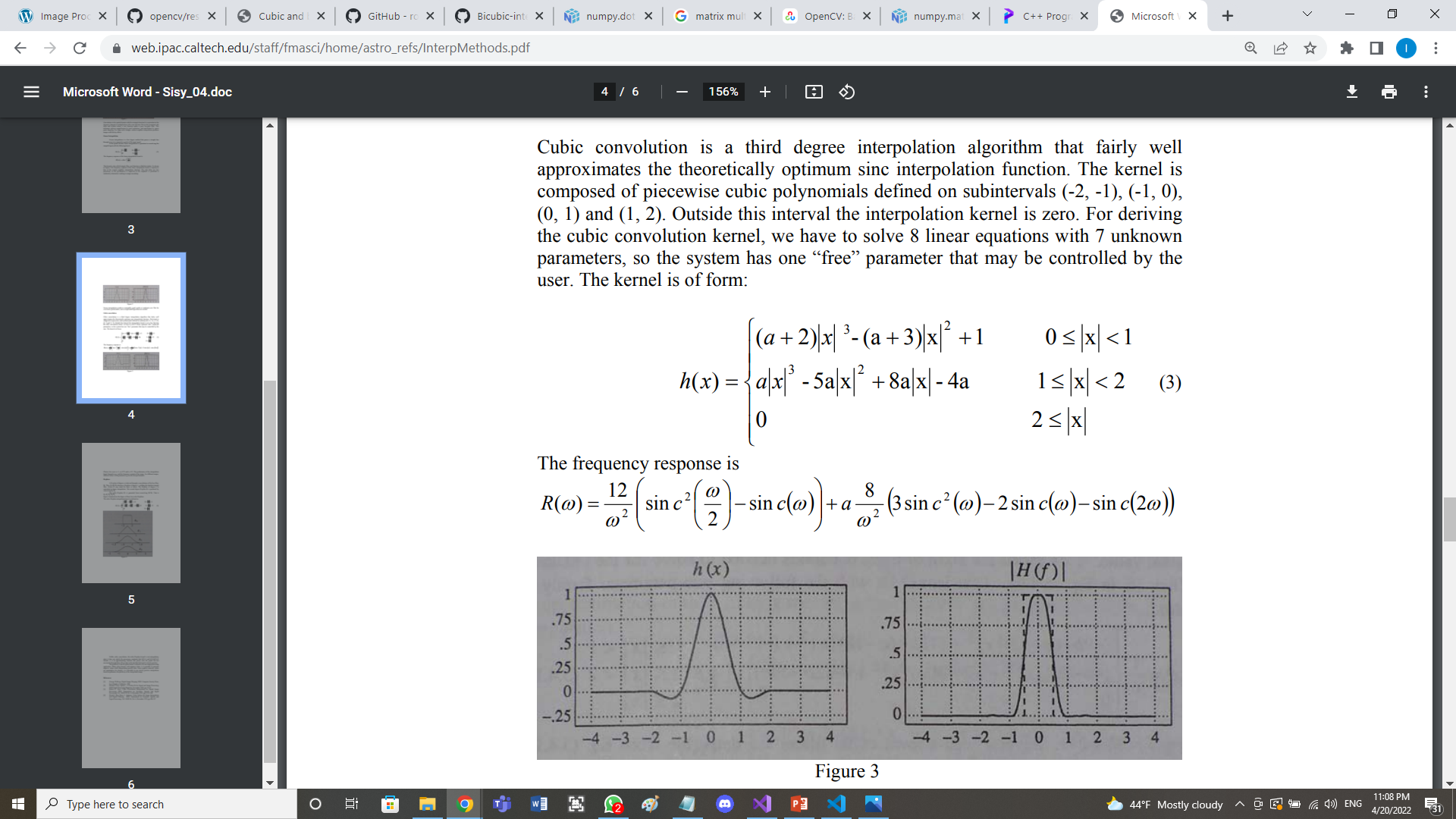
We take 16 points around the point; their x values will be: 160,161,162,163 and their y values: 4,5,6,7.

We compute the distances between our projected point (real nr) coordinates and the 16 points’ coordinates.

So x1 = 161.33-160 = 1.33, x2= 0.33, x3=0.66, x4=1.66

y1=1.33, y2=0.33,y3=0.66,y4=1.66. We need these distances to compute the weights of the 16 points.

We use this formula to compute the weights:

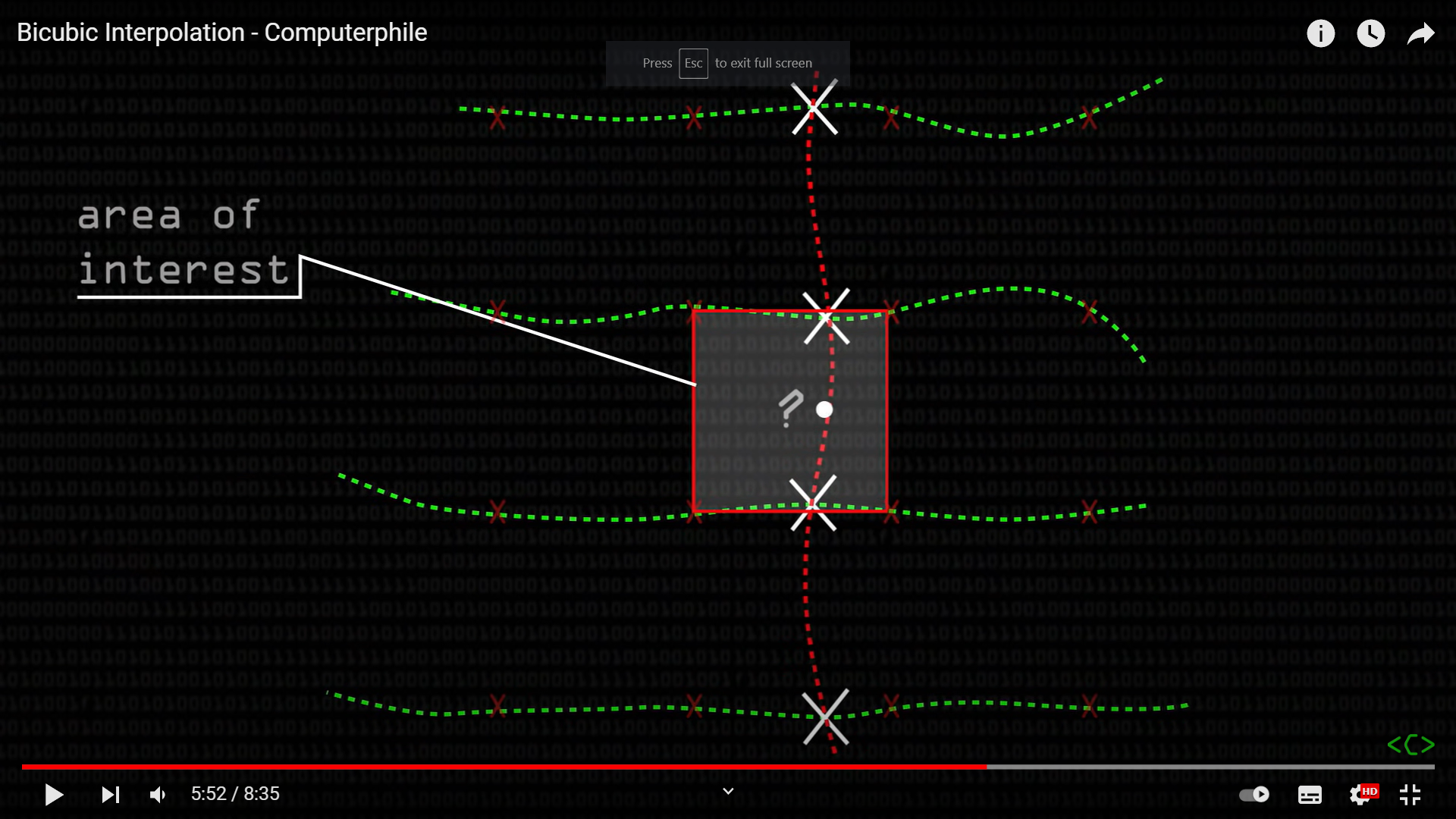


a is a chosen value (-0.5, -0.75, -1) we choose it -0.5.

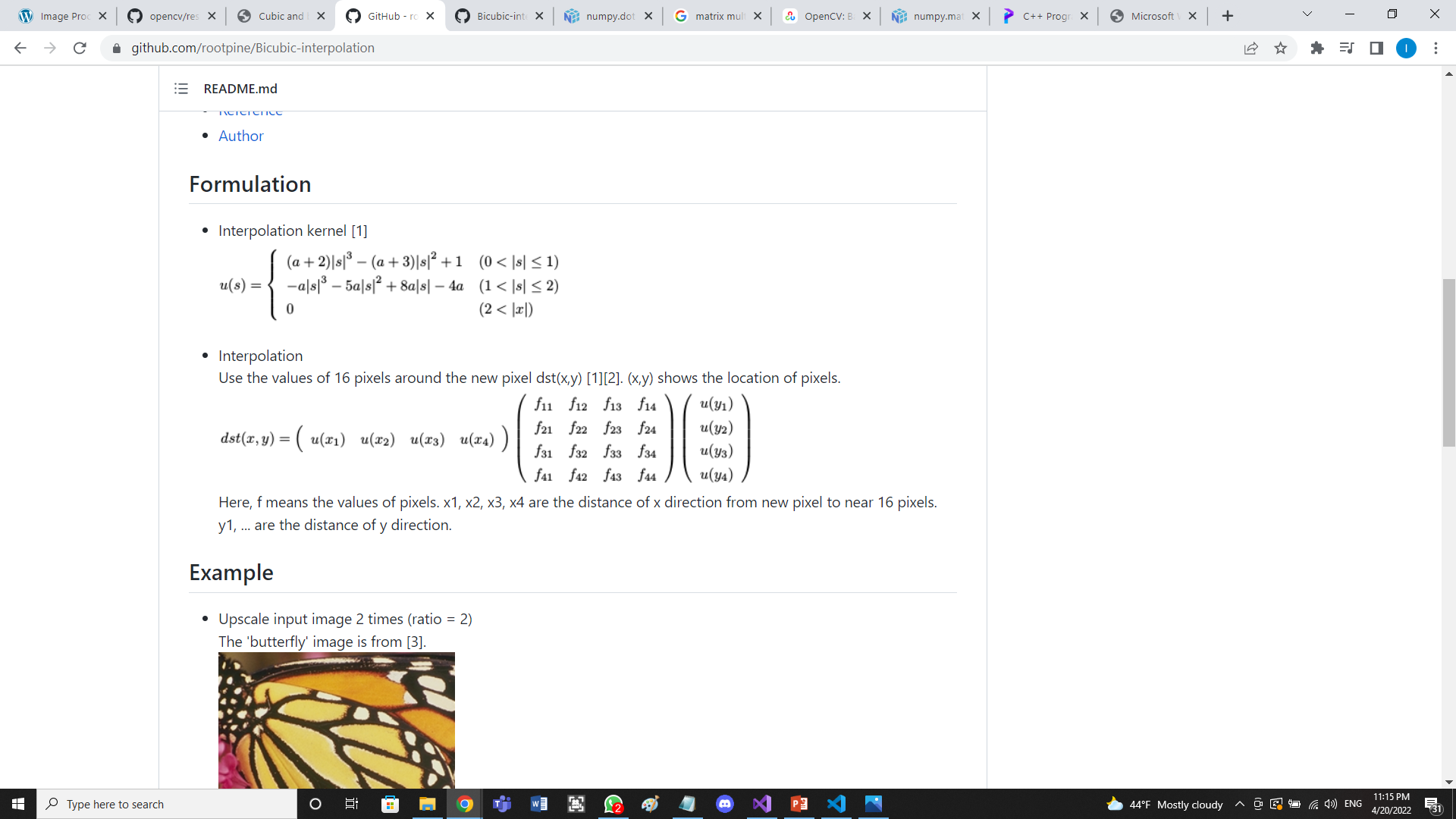
So we get h(x1)=-0.07, h(x2)=0.77, h(x3)=0.33, h(x4)=-0.03

h(y1)=-0.07, h(y2)=0.77, h(y3)=0.33, h(y4)=-0.03

Then we use these values to interpolate first horizontally (on 4 lines) then vertically the previous results to get the final value.



Doing this computation is exactly this matrix multiplication:



Where f are the pixel colors (level of grays) in the source image.

1. Implementation

3.1. Nearest Neighbor Interpolation

Mat\_<uchar> nearestNeighbor(Mat\_<uchar> src, float s) {

int height = src.rows;

int width = src.cols;

int newheight = round(s \* height);

int newwidth = round(s \* width);

Mat\_<uchar> dest = Mat(newheight, newwidth, CV\_8UC1);

for (int i = 0; i < newheight; i++) {

for (int j = 0; j < newwidth; j++) {

dest(i, j) = src(min(round(i / s),height-1), min(round(j / s),width-1));

}

}

return dest;

}

3.2. Bilinear Interpolation

Mat\_<uchar> bilinear(Mat\_<uchar> src, float s) {

int height = src.rows;

int width = src.cols;

int newheight = round(s \* height);

int newwidth = round(s \* width);

Mat\_<uchar> dest = Mat(newheight, newwidth, CV\_8UC1);

float isrc, jsrc;

int ifloor, iceil, jfloor, jceil, ul, ur, dl, dr,u,d,val;

for (int i = 0; i < newheight; i++) {

for (int j = 0; j < newwidth; j++) {

isrc = i / s;

jsrc = j / s;

ifloor = floor(isrc);

iceil = min(ceil(isrc),height-1);

jfloor = floor(jsrc);

jceil = min(ceil(jsrc),width-1);

if (ifloor == iceil && jfloor == jceil) {

val = src((int)isrc, (int)jsrc);

}

else

if (jfloor == jceil) {

u = src(ifloor, (int)jsrc);

d = src(iceil, (int)jsrc);

val = u \* (iceil - isrc) + d \* (isrc - ifloor);

}

else

if (ifloor == iceil) {

u = src((int)isrc, jfloor);

d = src((int)isrc, jceil);

val = u \* (jceil - jsrc) + d \* (jsrc - jfloor);

}

else {

ul = src(iceil, jceil);

ur = src(ifloor, jceil);

dl = src(iceil, jfloor);

dr = src(iceil, jceil); //u-up,d-down,l-left,r-right

u = ul \* (jceil - jsrc) + ur \* (jsrc - jfloor);

d = dl \* (jceil - jsrc) + dr \* (jsrc - jfloor);

val = u \* (iceil - isrc) + d \* (isrc - ifloor);

}

dest(i, j) = val;

}

}

return dest;

}

3.3. Bicubic Interpolation

Mat\_<uchar> bicubic(Mat\_<uchar> src, float s, float a) {

int height = src.rows;

int width = src.cols;

int newheight = round(s \* height);

int newwidth = round(s \* width);

Mat\_<uchar> dest = Mat(newheight, newwidth, CV\_8UC1);

float x, y, x1, x2, x3, x4, y1,y2,y3,y4;

for (int i = 0; i < newheight; i++) {

for (int j = 0; j < newwidth; j++) {

y = i / s;

x = j / s;

x1 = 1 + x - floor(x); //distance from x to point x1

x2 = x - floor(x);

x3 = floor(x) + 1 - x;

x4 = floor(x) + 2 - x;

y1 = 1 + y - floor(y);

y2 = y - floor(y);

y3 = floor(y) + 1 - y;

y4 = floor(y) + 2 - y;

// p1 p2 p3 p4

// p5 p6 p7 p8

// p9 p10 p11 p12

// p13 p14 p15 p16

//instead of addig a margin to the src I check not to get out of the image

//and if it would get out then just take the margin pixel like it would be

//duplicated on the border

Point p1(max(min((x - x1), width - 1), 0), max(min((y - y1), height - 1), 0));

Point p2(max(min((x - x2), width - 1), 0), max(min((y - y1), height - 1), 0));

Point p3(max(min((x + x3), width - 1), 0), max(min((y - y1), height - 1), 0));

Point p4(max(min((x + x4), width - 1), 0), max(min((y - y1), height - 1), 0));

Point p5(max(min((x - x1), width - 1), 0), max(min((y - y2), height - 1), 0));

Point p6(max(min((x - x2), width - 1), 0), max(min((y - y2), height - 1), 0));

Point p7(max(min((x + x3), width - 1), 0), max(min((y - y2), height - 1), 0));

Point p8(max(min((x + x4), width - 1), 0), max(min((y - y2), height - 1), 0));

Point p9(max(min((x - x1), width - 1), 0), max(min((y + y3), height - 1), 0));

Point p10(max(min((x - x2), width - 1), 0), max(min((y + y3), height - 1), 0));

Point p11(max(min((x + x3), width - 1), 0), max(min((y + y3), height - 1), 0));

Point p12(max(min((x + x4), width - 1), 0), max(min((y + y3), height - 1), 0));

Point p13(max(min((x - x1), width - 1), 0), max(min((y + y4), height - 1), 0));

Point p14(max(min((x - x2), width - 1), 0), max(min((y + y4), height - 1), 0));

Point p15(max(min((x + x3), width - 1), 0), max(min((y + y4), height - 1), 0));

Point p16(max(min((x + x4), width - 1), 0), max(min((y + y4), height - 1), 0));

//matrix multiplications

float m1[4] = { u(x1,a), u(x2,a), u(x3,a), u(x4,a) };

float m2[4][4] = { src(p1), src(p5), src(p9), src(p13),

src(p2), src(p6), src(p10), src(p14),

src(p3), src(p7), src(p11), src(p15),

src(p4), src(p8), src(p12), src(p16)

};

float m3[4] = { u(y1,a), u(y2,a), u(y3,a), u(y4,a) };

float intermediate[4] = { 0,0,0,0 };

for (int k = 0; k < 4; k++) {

float sum = 0;

for (int l = 0; l < 4; l++) {

sum += m1[l] \* m2[l][k];

}

intermediate[k] = sum;

}

float val = 0;

for (int k = 0; k < 4; k++) {

val += intermediate[k] \* m3[k];

}

dest(i, j) = (int)val;

}

}

return dest;

}

1. Results

We use the cameraman image to make a comparison between the algorithms.

First we zoom it with a scale of 5; the results are these:

4.1. Nearest Neighbor Interpolation



4.2. Bilinear Interpolation



4.3. Bicubic Interpolation



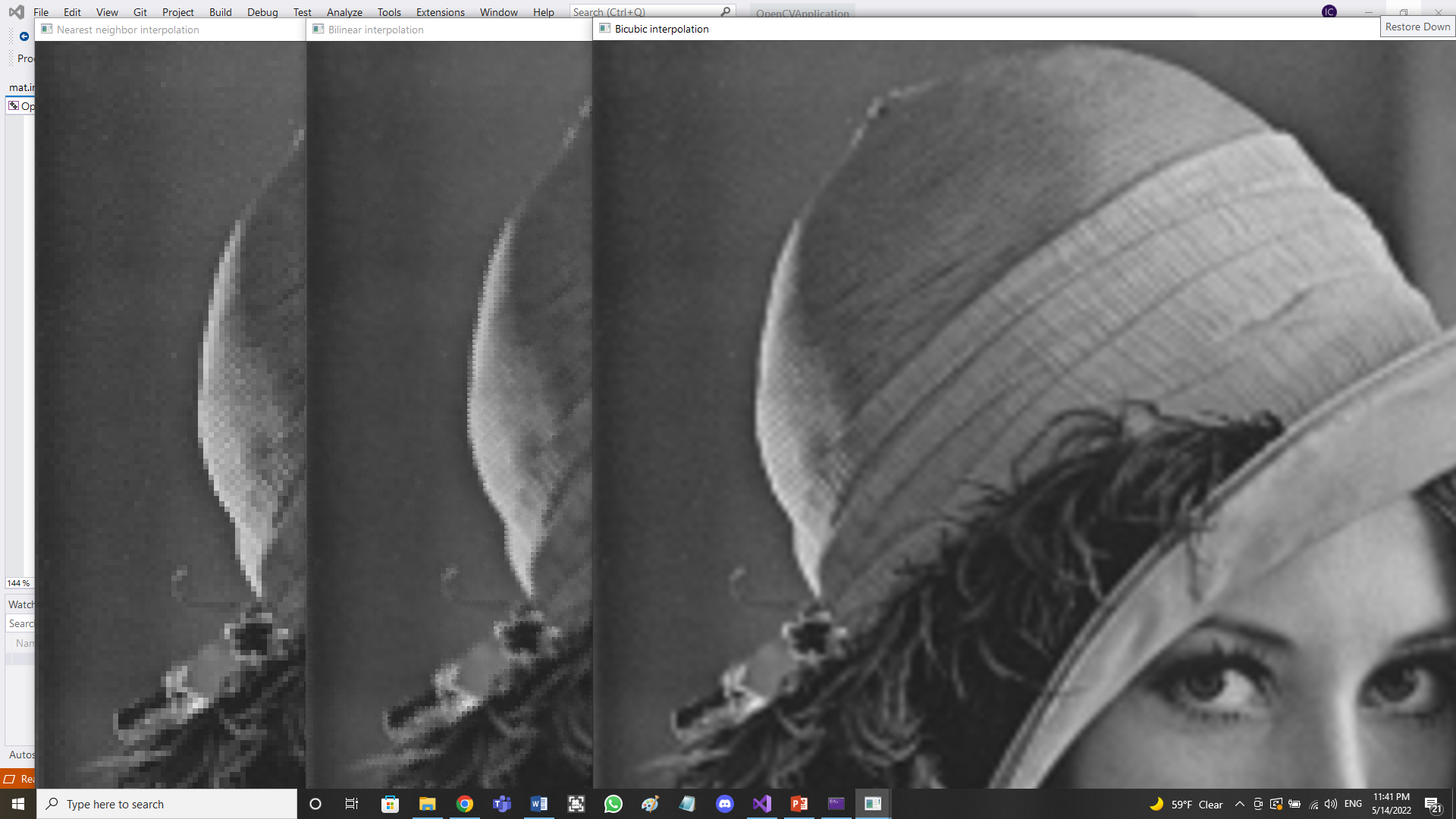
With the times:

Nearest neighbor interpolation time: 26.154 ms

Bilinear interpolation time: 24.648 ms

Bicubic interpolation time: 106.118 ms

The quality of the resulted image obviously increases in order: nearest neighbor, bilinear, bicubic and so does the time. The difference between biubic and the two other is very significant, both in quality and in time.

For the lena image we get these results:

1. Bibliography

<https://www.giassa.net/?page_id=207>

<https://github.com/rootpine/Bicubic-interpolation>

<https://github.com/rootpine/Bicubic-interpolation/blob/master/bicubic.py>

<https://web.ipac.caltech.edu/staff/fmasci/home/astro_refs/InterpMethods.pdf>

<https://www.youtube.com/watch?v=poY_nGzEEWM>

<https://www.mssc.mu.edu/~daniel/pubs/RoweTalkMSCS_BiCubic.pdf>

<https://www.youtube.com/watch?v=syH8ASkotFg&t=231s>

<https://meghal-darji.medium.com/implementing-bilinear-interpolation-for-image-resizing-357cbb2c2722>