Caleb Gannon
400137271

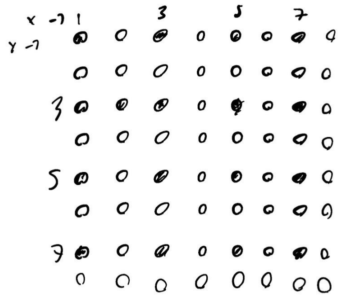
Project 1 3SK3
Image Interpolation

Part A. The Algorithm.

Parsing Solution

↳ working array is 8×8

↳ known values @ 1, 3, 5, 7



↳ $A = B^{-1} F C^{-1}$

where B is $[x^3\ x^2\ x\ 1]$

where C is $\begin{bmatrix} y^3 \\ y^2 \\ y \\ 1 \end{bmatrix}$

where F is 4×4 of function values.

↳ working from $1 \overset{\text{odd}}{-} 7$ x & y, B and C are going to be the same.

$$B: \begin{matrix} 1^3 & 1^2 & 1 & 1 \\ 3^3 & 3^2 & 3 & 1 \\ 5^3 & 5^2 & 5 & 1 \\ 7^3 & 7^2 & 7 & 1 \end{matrix}$$

$$C: \begin{matrix} 1^3 & 3^3 & 5^3 & 7^3 \\ 1^2 & 3^2 & 5^2 & 7^2 \\ 1 & 3 & 5 & 7 \\ 1 & 1 & 1 & 1 \end{matrix}$$

$$f(x,y) = [x^3\ x^2\ x\ 1]\, A \begin{bmatrix} y^3 \\ y^2 \\ y \\ 1 \end{bmatrix}$$

Solved for the 3x A matrices.

Caleb Gannon
400137271

My Algorithm -

1. The algorithm operates in 4x4 -> 8x8 chunks, using interpolation to find the missing pixels. In this way, it doubles the resolution of the image.
2. For interpolation, F=BAC is used. F is a matrix of the original image values, B and C are the x and y coordinates respectively for each of the known pixel values.
3. Since my algorithm is always interpolating in 8x8 chunks, the known indices will always be the odd numbers from 1-7.
4. Once the A matrix is obtained, the missing values at the even numbers 2,4,6,8 for (x,y) can be interpolated.
5. The newly interpolated 8x8 chunk is then inserted into the new image.
6. The loop then moves one pixel over, and repeats this process. Scanning across the entire image.

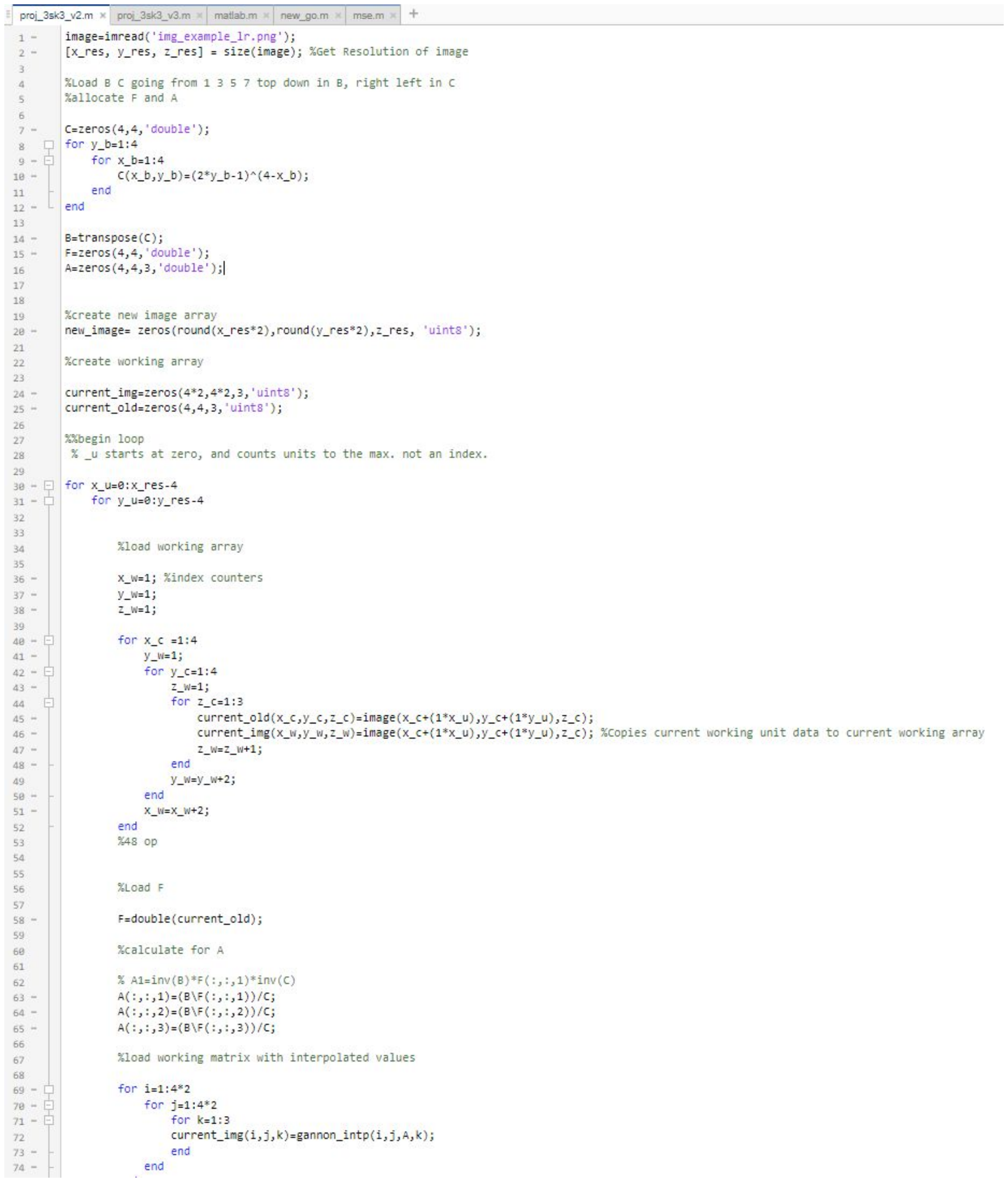
Part B. Differences between interpolated pixels.

Using this algorithm, each set of 5 new pixels within a group of 4 known pixels, is interpolated using their nearest surrounding 16 known pixels. No difference is made between the empty white, or the hashed white dots.

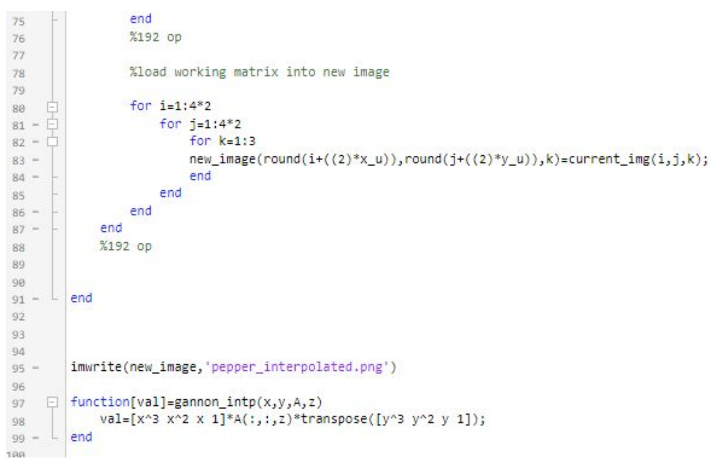
Part C. The Code.

Here is the matlab code used to interpolate an image.

Caleb Gannon
400137271



```
image=imread('img_example_lr.png');
[x_res, y_res, z_res] = size(image); %Get Resolution of image

%Load B C going from 1 3 5 7 top down in B, right left in C
%allocate F and A

C=zeros(4,4,'double');
for y_b=1:4
    for x_b=1:4
        C(x_b,y_b)=(2*y_b-1)^(4-x_b);
    end
end

B=transpose(C);
F=zeros(4,4,'double');
A=zeros(4,4,3,'double');


%create new image array
new_image= zeros(round(x_res*2),round(y_res*2),z_res, 'uint8');

%create working array

current_img=zeros(4*2,4*2,3,'uint8');
current_old=zeros(4,4,3,'uint8');

%%begin loop
 % _u starts at zero, and counts units to the max. not an index.

for x_u=0:x_res-4
    for y_u=0:y_res-4


        %load working array

        x_w=1; %index counters
        y_w=1;
        z_w=1;

        for x_c =1:4
            y_w=1;
            for y_c=1:4
                z_w=1;
                for z_c=1:3
                    current_old(x_c,y_c,z_c)=image(x_c+(1*x_u),y_c+(1*y_u),z_c);
                    current_img(x_w,y_w,z_w)=image(x_c+(1*x_u),y_c+(1*y_u),z_c); %Copies current working unit data to current working array
                    z_w=z_w+1;
                end
                y_w=y_w+2;
            end
            x_w=x_w+2;
        end
        %48 op


        %Load F

        F=double(current_old);

        %calculate for A

        % A1=inv(B)*F(:,:,1)*inv(C)
        A(:,:,1)=(B\F(:,:,1))/C;
        A(:,:,2)=(B\F(:,:,2))/C;
        A(:,:,3)=(B\F(:,:,3))/C;

        %load working matrix with interpolated values

        for i=1:4*2
            for j=1:4*2
                for k=1:3
                current_img(i,j,k)=gannon_intp(i,j,A,k);
                end
            end
```

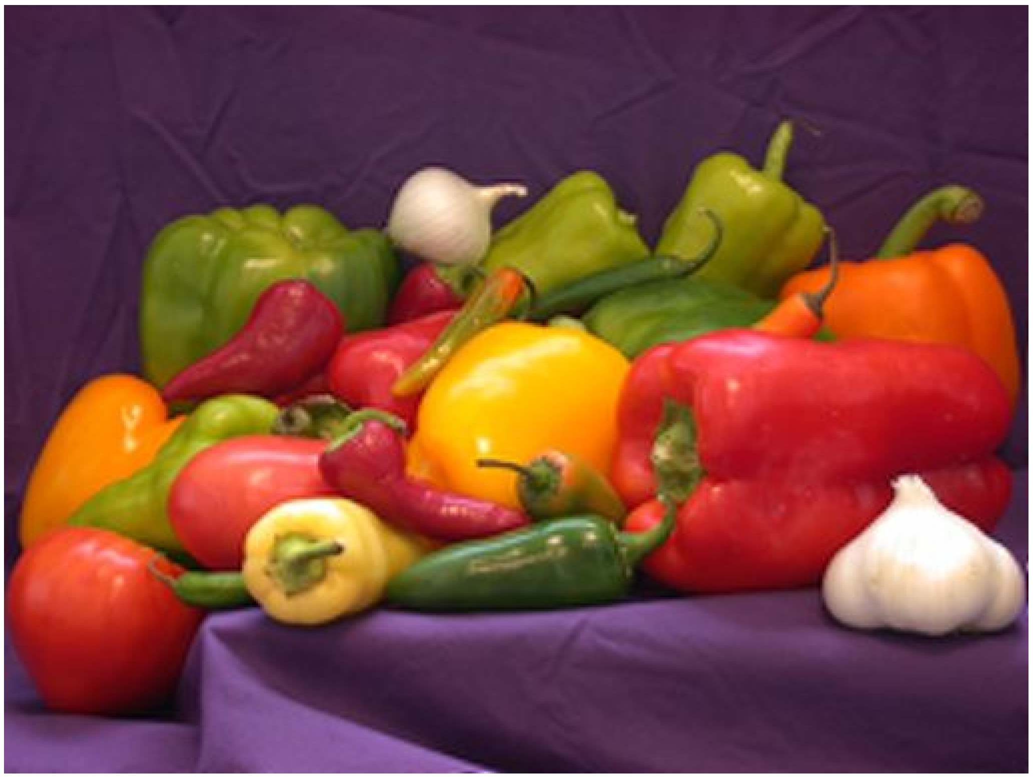


```
        end
        %192 op

        %load working matrix into new image

        for i=1:4*2
            for j=1:4*2
                for k=1:3
                new_image(round(i+((2)*x_u)),round(j+((2)*y_u)),k)=current_img(i,j,k);
                end
            end
        end
    end
    %192 op


end



imwrite(new_image,'pepper_interpolated.png')

function[val]=gannon_intp(x,y,A,z)
    val=[x^3 x^2 x 1]*A(:,:,z)*transpose([y^3 y^2 y 1]);
end
```

Caleb Gannon
400137271

Here is the original larger image.

Here is my code's larger image.



Caleb Gannon
400137271

Here is the original smaller image.

Part D. Efficiency.

I focused on efficiency in my code over other elements, choosing to overwrite known values, rather than recalculate my “A” array for each individual pixel. This method slides across the image, requiring only one calculation of “A” per pixel of the original image.

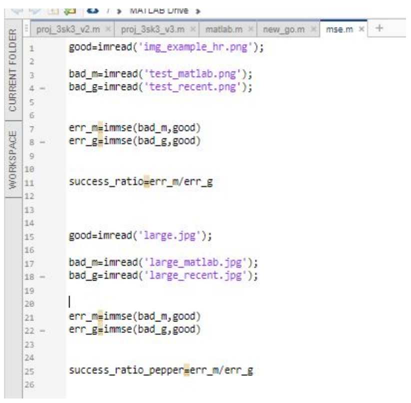
This is 4 times faster than if each pixel of the new resolution had to recalculate the same “A” matrix.

There are approximately 432 total operations, counting for-loops, to solve a 8x8 grid. In an 8x8 grid, there are 8x8-4x4 unknown pixels, or 48.

Therefore, it takes on average 9 operations to solve for a missing pixel.

Part E. Mean Square Error.

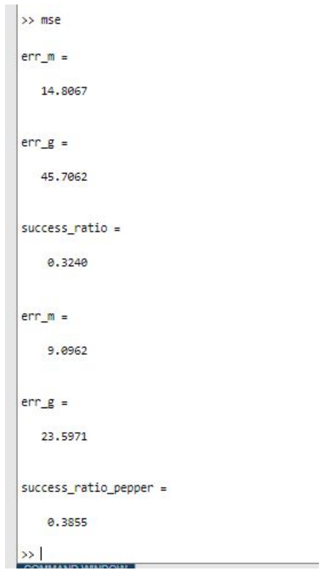
I wrote a small section of Matlab code to compare my code’s mean square error, against that of Matlab’s built-in bicubic interpolation operation.



```
good=imread('img_example_hr.png');

bad_m=imread('test_matlab.png');
bad_g=imread('test_recent.png');


err_m=immse(bad_m,good)
err_g=immse(bad_g,good)


success_ratio=err_m/err_g



good=imread('large.jpg');

bad_m=imread('large_matlab.jpg');
bad_g=imread('large_recent.jpg');


err_m=immse(bad_m,good)
err_g=immse(bad_g,good)


success_ratio_pepper=err_m/err_g

```

Caleb Gannon
400137271

The results came in as such:



```
>> mse

err_m =

   14.8067


err_g =

   45.7062


success_ratio =

    0.3240


err_m =

    9.0962


err_g =

   23.5971


success_ratio_pepper =

    0.3855

>> |
```

err_m is matlab's bicubic error against the original image, at 14.8.
err_g is my bicubic error against the original image, at 45.7.

To better interpret this data, I used a ratio of the mean squared errors.
By the mean square error metric, my code is about 32.4% as accurate as matlabs.

I ran the tests again against another image I found online.
("https://www.wallpaperflare.com/woman-wearing-tank-top-climbing-on-brown-mountain-wallpaper-184123")

By these tests, matlab is at 9.1 MSE, my code is at 23.6, and the ratio shows 38.6%

The difference is likely a result of my tradeoff for efficiency. As I worked in 8x8 grids, scrolling to solve individual 2x2 sections per "A" matrix, rather than individual pixels, a slight amount of fidelity is likely lost.

According to the documentation, matlab's bicubic interpolation method solves each pixel according to its own individual "A" matrix, increasing fidelity.

This would also explain why the MSE between our implementations decreases at higher resolutions, as individual pixels have less of an impact.