USING SINGULAR SPECTRUM ANALYSIS FOR IMAGE PROCESSING AND REMOVING NOISE FROM DIGITAL IMAGES

Blake Hamilton, 109750167

APPM 3310-001 Spring 2022

ABSTRACT

Noise reduction is one of the most common types of digital image processing. Successful denoising is important to most types of photography and imagery. The infamous noisy "Lena" image had its noise reduced using Singular Spectrum Analysis to show a basic application of the SSA process. Digital images can be manipulated by making changes to their color intensity matrices. One application of noise reduction is the use within artistic astrophotography (photography of stars and outer space). The SSA process was used on an extremely noisy Milky Way Galaxy photograph to process and edit the picture the way a photographer would. This was done by creating a trajectory matrix through SSA, then after making column approximations with Singular Value Decomposition (SVD), the final denoised image was produced. The SSA process was done using matrix methods within MatLab, and it successfully processed the image and reduced noise similarly to professional digital image processing programs.

INTRODUCTION

In 2022, digital images are used more than ever before. Between social media, traditional media, satellite imagery, and many more applications, digital image processing and image manipulation are very important to create desirable images. In recreational astrophotography, the most common problem is having a raw image from a camera that is too noisy. This not only makes it visually unappealing, but also distorts what the stars looked like in real life. To solve this problem, image processing techniques are used to reduce noise how the photographer prefers.

Image noise is the random variation of color and light in an image, causing certain pixels to store information that is not representative of the light the camera sensor should have captured. "Finding a [noise] solution often proves to be a significant challenge in imaging, particularly in a low-light situation where the signal is already low" (1). This is because the camera must increase light sensitivity in low light situations, meaning there is more random light detected and more unwanted information stored in the image. The idea of noise reduction is to find the pixels that stand out from the others, and make them match their surrounding pixels, so they do not stand out as much. This also has the effect of blurring an image.

Two images were tested: “Lena” and “Galaxy”. Lena is the most widely used test image for digital compression algorithms and was downloaded as an open-source image (2). This image was used first to show the SSA process on a standard test image, before applying the process to a more specific application. The Galaxy image was taken by the author, an amateur photographer, in fall of 2021. It is a very noisy image and needed noise reduction to make the image representative of the Milky Way. Galaxy was a high-quality image compressed down to two megapixels to process the image on a standard laptop computer. The goal of using the Galaxy image is to use SSA, along with other basic matrix math, to process the image.

MATHEMATICAL FORMULATION

Digital color images (JPGs, PNGs) are formed by three overlayed matrices: one red, one blue, and one green. Each of these three matrices has height *h* and a width *w* that are the same as the height and width of the image (in pixels). For each color matrix $\mathbf{M}_{h\times w}$, each of the matrix entries $\boldsymbol{M}_{i,j}$ has an intensity value of $0 \leq \boldsymbol{M}_{i,j} \leq 255$, with the intensity being how bright the pixel is in that specified pixel/matrix index. (256 possible values comes from the possible numbers one byte can store). The first analysis is of a grayscale image, Lena, so the image does not have three color matrices, but rather one grayscale matrix. In a grayscale image matrix, a $\boldsymbol{M}_{i,j} = 0$ represents a black pixel and a $\boldsymbol{M}_{i,j} = 255$ is a white pixel. Values in between will be shades of gray. For the Lena image with height *h* and width *w*, its corresponding matrix will be defined as $\boldsymbol{L}_{i,j} \mid 1 \leq i \leq h, 1 \leq j \leq w$.

The Singular Spectrum Analysis method uses **L** to create a trajectory matrix **X.** The trajectory matrix is formed by choosing a “window” size, and moving this window to every location possible in **L** and saving the points as column vectors in **X**. First, a window size $u \times v$ will be chosen such that $1 \leq \mathrm{u} \leq h$ and $1 \leq \mathrm{v} \leq w$. The top left entry in the window will be used as the reference point for moving this window around **L**. The reference point $(\hat{\imath}, \hat{\jmath})$ will be moved such that $1 \leq \hat{\imath} \leq h - u + 1 \; and \; 1 \leq \hat{\jmath} \leq w - v + 1$ (1). This is most easily applied with a computer by looping through from left to right, then top to bottom by keeping *i* constant while increasing *j* from its minimum to its maximum, then increasing *i* and resetting *j*. This will ensure that the window is moved over every frame possible in the matrix. For each reference point $(\hat{\imath}, \hat{\jmath})$, there will be a correlating column in **X** storing the window values in the following fashion: for column *r* in **X** correlating to the r^{th} reference point, each entry going down the column will hold the value of $\boldsymbol{X}_{i,j}$ traversing left to right across the window, then top to bottom. The following shows this process. Let $u = v = 2$ and matrix **A** be:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

The first window of $\begin{bmatrix} a & b \\ d & e \end{bmatrix} \Longrightarrow \mathbf{X}_{i,1} = \begin{pmatrix} a \\ b \\ d \\ e \end{pmatrix}$, and the window will be moved by (1) to give

$$\boldsymbol{X} = \begin{bmatrix} a & b & d & e \\ b & c & e & f \\ d & e & g & h \\ e & f & h & i \end{bmatrix} \qquad (2)$$

In broader terms, the following is the formation of the trajectory matrix **X** from any $h \times w$ image matrix **I** with window size 2×2.

$\begin{pmatrix} [I_{1,1}] & [I_{1,2}] & \cdots & I_{1,w} \\ [I_{2,1}] & [I_{2,2}] & \cdots & I_{2,w} \\ \vdots & \vdots & \ddots & \vdots \\ I_{h,1} & I_{h,2} & \cdots & I_{h,w} \end{pmatrix}$ The matrix entries encapsuled by [ ] are selected as the window.

The reference point for the window is moved according to Equation (1) from the first indexed window shown above to the final window below.

$\begin{pmatrix} [I_{1,1}] & \cdots & \cdots & I_{1,w} \\ \vdots & \ddots & \vdots & \vdots \\ \vdots & \cdots & [I_{h-1,w-1}] & [I_{h-1,w}] \\ I_{h,1} & \cdots & [I_{h,w-1}] & [I_{h,w}] \end{pmatrix}$ This is the final window placement.

The following is the generalized trajectory matrix $\boldsymbol{X_{h \times w}}$ after moving the window to every required placement. It should be noted that there are multiples of some of the variables.

$$\boldsymbol{X} = \begin{pmatrix} I_{1,1} & I_{1,2} & \cdots & I_{1,w-1} & \cdots & I_{h-1,w-1} \\ I_{1,2} & I_{1,3} & \cdots & I_{1,w} & \cdots & I_{h-1,w} \\ I_{2,1} & I_{2,2} & \cdots & I_{2,w-1} & \cdots & I_{h,w-1} \\ I_{2,2} & I_{2,3} & \cdots & I_{2,w} & \cdots & I_{h,w} \end{pmatrix}$$

As seen in Matrix (2), the trajectory matrix is 4×4. Since each column in **X** represents all elements of a window, and each column represents a different reference point by (1), the height p and width q of $\boldsymbol{X}_{p,q}$ can be generalized to the following:

$$p = u \times v$$

$$q = (h - u + 1) \times (w - v + 1) \qquad (3)$$

The next piece of the process is to calculate the Singular Value Decomposition of $\mathbf{XX^T}$, which is to represent the matrix as $\boldsymbol{XX^T} = \boldsymbol{U\Sigma V^T}$ where $\boldsymbol{\Sigma}$ is a diagonal matrix containing the eigenvalues from largest to smallest, while **U** and **V** are orthogonal matrices storing the eigenvectors. The properties of this decomposition ensure that $U^T U = V^T V = I$. (The process

of SVD will not be covered in this paper, as it is a common decomposition and can be calculated using the SVD MatLab function). Since multiplying a matrix by its transpose gives a symmetrical matrix, and $\mathbf{\Sigma}$ is a diagonal matrix, $\mathbf{U} = \mathbf{V}$, giving the equation $\boldsymbol{XX^T} = \boldsymbol{U\Sigma U^T}$ (4). The resulting matrix $\mathbf{XX^T}$ has dimensions $u \times u$, since the long v edge is an internal edge in the matrix multiplication.

Since the first eigenvalue in $\mathbf{\Sigma}$ is the largest (from the MatLab decomposition), the first column of $\mathbf{U}$ and first row of $\mathbf{U}^T$ are multiplied by $\mathbf{X}$ in the following fashion to get an approximation to $\mathbf{X}$ (that is, each column of $\mathbf{X}$ will be approximated by using the first eigenvector). $\breve{\boldsymbol{X}} = \boldsymbol{U_{i=1:p,1}U^T_{1,j=1:p}X}$ such that i,j span the given column/vector. This SVD method is a common process to reduce data and consolidate it to save space in computing applications. It is used by itself (without SSA) in some applications to compress images, which (often unintentionally) reduces the noise of an image slightly. Only the first eigenvectors are used to create the best approximations of $\mathbf{X}$: if more vectors were used, this would create a more accurate estimation of $\mathbf{X}$, which is not desired for this application. Using more eigenvectors would be counterproductive, and if all the eigenvectors were used, this would just perfectly recreate $\mathbf{X}$, and therefore give the exact same final image.

The final step in the process is reconstructing the new image matrix $\breve{\boldsymbol{L}}$ of the same dimension as $\mathbf{L}$**,** but with depth of 2 in the third dimension (explained later). This is done using the modified transformation matrix $\breve{\boldsymbol{X}}$. This process is like reversing the process demonstrated in (2). The only difference now is that there are multiple entries in $\breve{\boldsymbol{X}}$ corresponding to the same entry in $\breve{\boldsymbol{L}}$, so there is an extra important step to merging multiple numbers into one final entry in $\breve{\boldsymbol{L}}$ (this is the most important step to remove noise in an image instead of just doing a typical SVD decomposition). When merging the entries from $\breve{\boldsymbol{X}}$ into $\breve{\boldsymbol{L}}$, $\breve{\boldsymbol{L}}$ starts as a zero matrix, and each $\breve{\boldsymbol{X}}$ entry will add itself to the current value of its corresponding entry in $\breve{\boldsymbol{L}}$. While doing this, the second layer of the matrix will keep track of the number of times the corresponding first level entry has been updated. This number will be used as the divisor to average out the sum of all the different corresponding entries. A final matrix $\mathbf{F}$ will store the results. For example, if $\breve{\boldsymbol{L}}_{42,35,1} = 200$, and its corresponding divisor $\breve{\boldsymbol{L}}_{42,35,2} = 4$, then the corresponding $\mathbf{F}_{42,35} = 50$. The following shows this idea as a continuation of (2):

"*" Represents different approximations of the same variable

$$\breve{\boldsymbol{X}} = \begin{bmatrix} a & b* & d* & e*** \\ b & c & e** & f* \\ d & e* & g & h* \\ e & f & h & i \end{bmatrix}$$ Modified trajectory matrix

$$\breve{L}1 = \begin{bmatrix} a & b+b* & c \\ d+d* & e+e*+e**+e*** & f+f* \\ g & h+h* & i \end{bmatrix}$$ This is the first level of $\breve{\boldsymbol{L}}$

$$\check{L}2 = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$ This is the second level of $\check{\boldsymbol{L}}$.

$$F = \begin{bmatrix} a & \frac{b+b*}{2} & c \\ \frac{d+d*}{2} & \frac{e+e*+e**+e***}{4} & \frac{f+f*}{2} \\ g & \frac{h+h*}{2} & i \end{bmatrix}$$ Final denoised matrix (3)

In the result of the above example, it is observed that the edge entries (first or last column/row) in the matrix do not get averaged out, if at all (in the case of corners *a, c, g, i*). This makes sense because those entries do not have surrounding data to average them out. In other words, there is not much information surrounding those pixels, so there isn't information to use to estimate what those pixels should be. On the contrary, for internal matrix entries, there is more information around them, so they get a more accurate average, as seen at entry *e*, since it is averaged out by four approximations in the trajectory matrix. This idea of averaging out a given matrix entry is what creates the denoising effect: noise is created by pixels not representing what they should be, and therefore the pixels stick out and hold vastly different values than the pixels surrounding them. Averaging them out smoothens out these sharp pixels, and therefore makes the image smoother and less noisy.

An interesting change that can be made in this process is changing the ratio between *u* and *v*. In photography, this ratio is called the *aspect ratio.* When dealing with a square image, a square window, i.e., $u = v$, the blur will appear even and the denoising properties will be equally horizontal as vertical. If *u* were to be twice the size of *v* (giving a rectangular window taller than it is wide), then this would give more of a blur in the vertical direction. This happens because a taller window means the horizontal relationship between matrix entries is weighted more, and as (4) will weigh the horizontal relationships more, there will be more blurring in the vertical direction after matrix reconstruction. This will be shown in the results later.

The process shown for grayscale matrices can be used in a similar way to denoise color images. For a given color image, as explained at the start of the section, there are three different channels: one each for red, green, and blue. To denoise a color image using SSA, repeat the outlined process for each of the color matrices and merge them back together afterwards to form the final color image. This is a more common application, as most pictures used and processed now are color images, and not monochrome (black and white).

EXAMPLES AND RESULTS

The first application was to test the SSA denoising algorithm on the Lena image, Figure (2). This image is low quality at 128×128 pixels, but the noise can be seen around the edges of shapes in the figure, and the way the image looks rough.

Figure 1: Lena

Use of Code 7 to create the trajectory matrix for the grayscale Lena matrix yielded a trajectory matrix **X**, then after use of Code 8 and Code 9, the result was an image that was a denoised version of the original Lena matrix. This process was repeated for different window sizes and the reference point moved each time as described in equation (1). The difference in window sizes (represented with vertical dimension *u* and horizontal dimension *v*) can be seen in Figure (2), as the image noise decreases, and blur increases as the window size increases until the image is almost unrecognizable with a window size of $u = v = 16$.

Figure 2: Change of Lena Window Size

When comparing the top left image in Figure (2) ($u = v = 1$) to Figure (1), there is almost no difference. This is because the window size of 1×1 does not result in multiple entries in **X** but rather just one, and therefore the modified **X** is almost an exact replica of **X**. The only difference is caused by the slight compression/blurring effects caused by SVD. As observed in Figure (2), the blur in all images appears even and smooth: it does not appear to be blurred in one direction more than the other. This visual is consistent with what is expected theoretically from example (3): as there are larger window sizes, the pixels are averaged out more and therefore the image results in looking smoother and less noisy.

Figure 3: Uneven Denoising

Figure (3) shows the way that a non-square window used with a square picture will blur an image. In Figure (3), *u* was held constant to 1, and *v* was increased to the values specified above each image, which can be visualized as a rectangular window that keeps getting taller with the same width. It is observed in Figure (3) that at higher *v* values, with a window aspect ratio progressively more different than the original image aspect ratio, the denoising is more distorted. For almost all real-world applications of denoising, an evenly distributed blur is preferred. Therefore, for the later application on the non-square Galaxy image, the window aspect ratio will be held the same as the original image to not have any distorted blurring effects like the ones in Figure (3).

Figure 4: Galaxy Image

Figure (4) shows the Galaxy image that will be used to test the denoising application on a more advanced, real-world image. The desire for a photographer processing an image like Figure (4) is to smoothen out the stars and make the color of the Milky Way stand out more and be less noisy and pixelated. The image is a 3:2 height to width aspect ratio, and as shown in Figure (3), it is optimal to choose a window with the same aspect ratio. Therefore, for the final denoising of Galaxy, $u = 3t$ and $v = 2t$ (4) for some real number *t.* To decide what window size (what value of *t*) is desirable for this image, a test case of four different window sizes was done for the image (Figure (5)). For simplicity, the test cases were done on square monochrome sections of the picture. (This helped save time when compiling the program, as Galaxy is a 2MB image, and takes much longer to process than Lena).

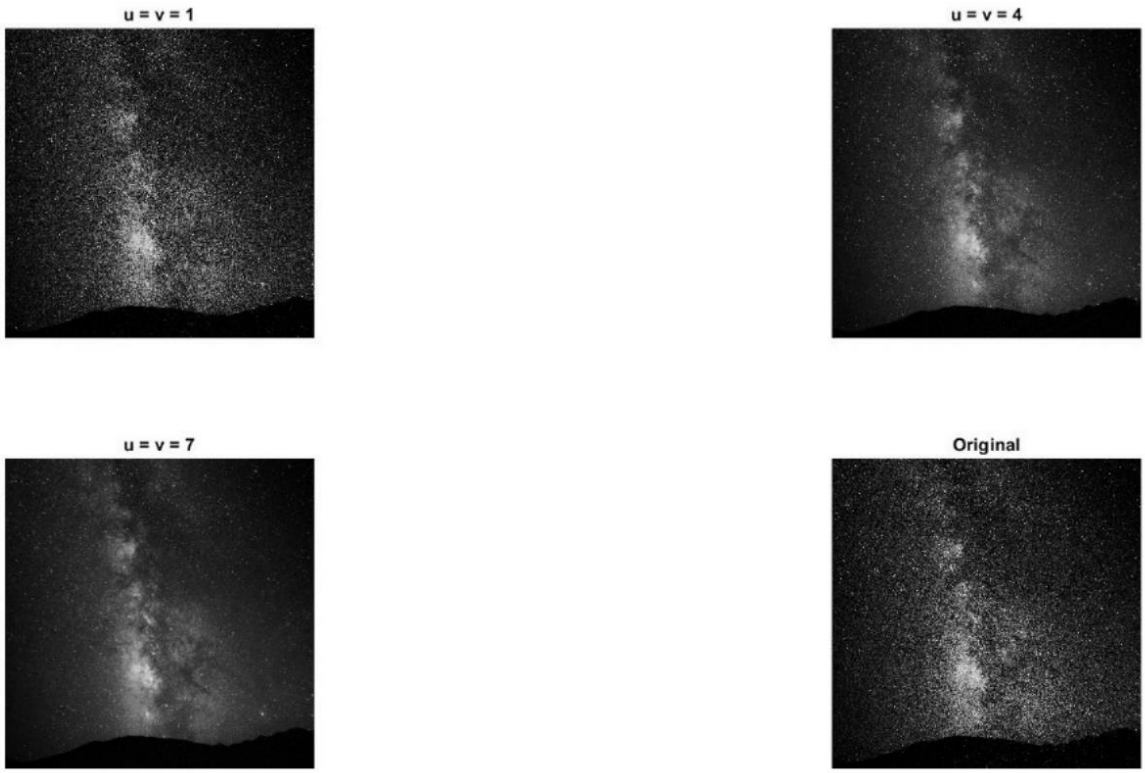

Figure 5: Galaxy Window Size Test Cases

Of the test cases in Figure (7), $u = v = 4$ was chosen as the best window size for its balance between noise and blur. To translate this same denoising factor to the full image, Equation (4) was followed to find $u = 6\ and\ v = 4$. Figure (6) shows the final product of the noise reduction.

Figure 6: Original Image and Final Noise Reduced Image

Figure 7: After/Before Zoomed In

The difference between the before and after of the SSA process in Figure (6) may seem small or insignificant, especially when viewing on a small computer monitor, but Figure (7) aims to highlight the difference. Figure (7) shows the two images spliced together, with the final image being the upper section and the original the lower. It can be seen how the areas of the original image that are primarily stars became smoother and brighter, while the areas that are primarily dark space become darker and less pixelated. This is desirable, since the SSA process successfully brings out the color and light of the Milky Way in the final picture, instead of the pixelated noisy version of the Milky Way on the bottom. This change makes sense mathematically, since a single bright pixel (a random star or random unwanted light) surrounded by dark pixels will then have many darker approximations from the transformation matrix that make this bright pixel darker. This idea across the entire image makes it look much better. Even after viewing Figure (7), it may still seem insignificant, but reducing noise will have an exaggerated benefit when printing and enlarging pictures. The following (Figure (8)) quantifies the effects of denoising the Galaxy image in case visual inspection does not show the success of the denoising.

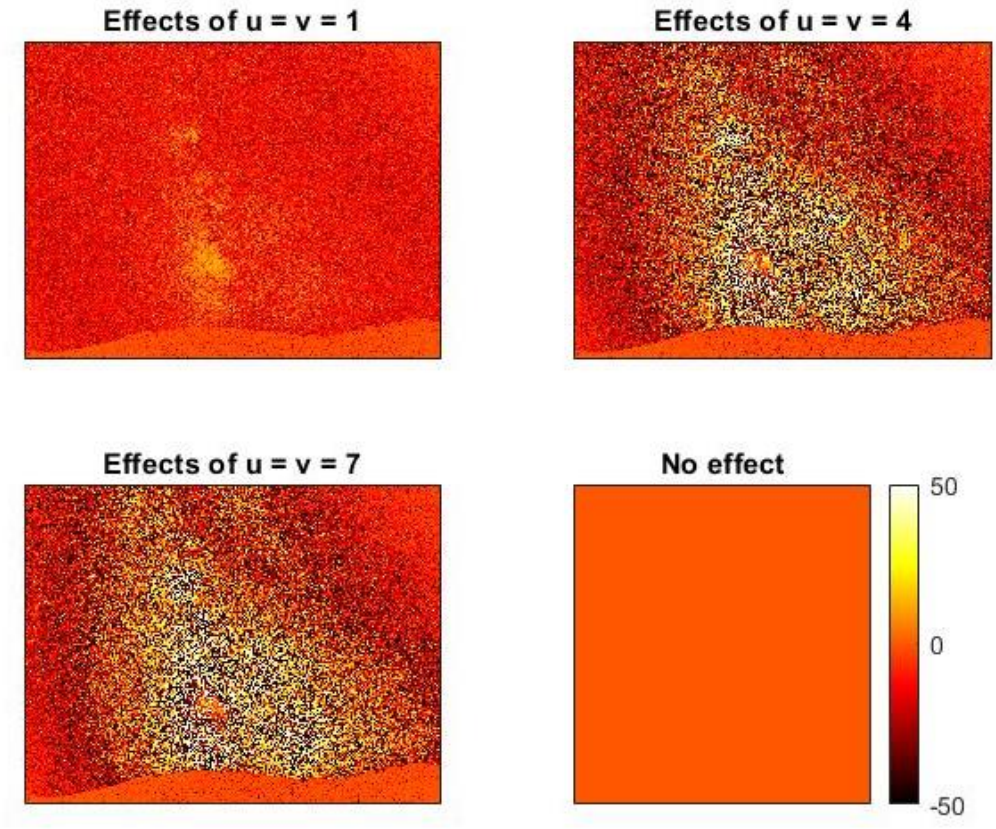

Figure 8: Effects of Denoising

Figure (8) was made by finding the difference between each of the matrices in Figure (5) and the original matrix. Each edited matrix **E** from Figure (4) was subtracted by the original unedited image matrix **G** to give a final matrix $\boldsymbol{D = E - G}$**.** The images in Figure (8) can be interpreted by observing the scale and image at the bottom right. Since the original matrix was compared to itself, the entire frame on the bottom right is the same orange color correlating to a difference of zero. Looking at the other difference matrices shows that any dark pixel shows the corresponding pixel in the original image was made darker by the denoising. On the contrary, any pixel lighter than orange means that it was made brighter by denoising.

As mentioned before, there is almost no difference between the original matrix and the denoised one with a window size of 1 (top left image in Figure (8)). This can be seen because the image is primarily the same orange color as the bottom right image. However, for window sizes of 4 and 7, there is clearly a difference between the denoised images and the original. The areas of the top right and bottom left images in Figure (8) with primarily black pixels correlate to the areas of the original image that had random stars and unwanted light diffractions creating noise, and the SSA algorithm got rid of these pixels by making them darker. In the center of the difference map, where the Galaxy image contains the Milky Way, there was the opposite effect- and most pixels were made lighter. It can also be seen that at the very bottom of the difference map frames for the 4 and 7 windows, there are random black pixels. This represents the denoising of random unwanted light that appeared below the skyline in the original image (clearly unwanted, as there are no stars below the horizon). To better understand these difference maps, compare them to Figure (7) for a pixel-by-pixel comparison.

DISCUSSION AND CONCLUSION

The use of SSA to reduce the noise in a digital image was certainly successful. An extremely noisy image could be processed with the techniques in this paper. The results of denoising are comparable to those of a professional denoising software, such as Adobe Lightroom. The denoised images showed no flaws, and the results on extended applications were as expected based on the theoretical math and test denoising on the Lena image. To the untrained eye, the denoising may seem minimal, but to a photographer, there was successful denoising.

The big problem found with using the technique is that it is very expensive computationally. Equation (3) is of time complexity $O(n^2)$, and due to some other necessary steps in the process, the overall process results in a $O(n^3)$ time complexity. The reasoning for this is beyond the scope of this paper. A time complexity this size is very large and costly in a computing power sense. This means that to denoise a standard size image on a laptop with this method will be very slow and could be impossible. The code sections referenced in the code appendix had to be run one at a time with a disk memory cleaning in between each on a standard laptop to compile without crashing. The original Galaxy image was a 15 megapixel image and had to be compressed to a 2 megapixel image to make it processable on a laptop computer with

MatLab. Even using this compressed version of the image, the largest window that could be used in the process was a 6×4 window. For many applications, it would be desirable to make the window size bigger and/or keep the original image in its uncompressed form. The requirement of a large amount of computing power is certainly a limitation of the SSA process.

The process of writing the code and implementing the computation applications of the relevant mathematical theory showed how advanced many of the image editing platforms are. Although SSA did a comparable job to Adobe Lightroom with noise reduction, it took much longer: in Lightroom, noise reduction is almost instant and can be done simply by dragging a slider back and forth. This experiment showed how important it is to keep images stored as efficiently as possible, as processing them can quickly take up an enormous amount of computing power.

If this experiment was repeated, it would be favorable to do it with larger, off-site computing power, instead of just a personal laptop. This would enable a more extensive application of the SSA process, as it would be possible to denoise much larger images and denoise them more (with larger window sizes). There would also be an advantage to surveying people to see what they found most attractive in image noise reduction, and then base the denoising of the Galaxy image on what was generally seen as better quality.

One application this research could be extended to is satellite imagery. Satellite images are often very noisy, as they are high-definition pictures taken with cameras with extremely high sensitivity. On top of this, their images are extra noisy because there is a great distance between a satellite and what it is taking a picture of, which creates more room for unwanted light to corrupt the images it takes. Satellite images are very important, not only in astrophotography fields, but also in mapping, security, and surveillance. The trouble with applying SSA to satellite imagery, though, is that satellite images are generally massive files, and as the SSA algorithm is very costly to computing power, it would take immense amounts of computing power to denoise satellite images in this way.

A future research idea stemming from this research would be figuring out a way to minimize the computing power required to apply SSA to images. It would require finding ways to make the trajectory matrix $\mathbf{X}$ smaller for given image and window sizes. This would not be an easy task, but perhaps there would be ways to find weights to the trajectory matrix rows so that there are fewer columns in $\mathbf{X}$.

CODE DESCRIPTION

The following is a brief description of each code section used and shown in the Code Appendix. For more information about each section, comments can be seen in the MatLab code in the Code Appendix.

1. Processes a specified image, splits the color channels into three one-dimensional matrices, and then combines the color channels back together to display the image.

2. Crops a matrix using a modified identity matrix (unused, but interesting application).
3. Applies the full SSA process in varying amounts, then displays results. Creates a second figure that shows the effects of vertical-only denoising. Uses Code (6).
4. Explicitly creates three variations of the Galaxy image with varying window sizes, then displays the results together. Uses Code (6).
5. Denoises the Galaxy image with slightly edited color channels for extra color clarity, and shows the final denoised image.
6. Main SSA function: calls Code (7), Code (8), and Code (9) to make applying SSA repeatedly easier in previous sections.
7. SSA destructor function: creates the trajectory matrix **X** as described at Matrix (2) from an inputted monochrome (gray) matrix.
8. SVD function: runs a SVD process on the inputted matrix, and returns the approximated version of that matrix.
9. SSA Reconstructor function: Takes an input trajectory matrix and creates a final image matrix from this. Must use the three dimensional second layer to the matrix to average out the corresponding entries to the final matrix.

CODE APPENDIX

Code 1

```
%%Showing how to read in an image and seperate color channels
%reading in an image
original_image = double(imread('Galaxy.jpg'));

%turning image into a greyscale matrix, combining each color channel
greyscale_image = original_image(:,:,1) / 3 + original_image(:,:,2) / 3 +
original_image(:,:,3) / 3;

%showing how to edit lighness of image
for i = 1:height(greyscale_image)
    for j = 1:width(greyscale_image)
        greyscale_image(i, j) = greyscale_image(i, j) + (greyscale_image(i, j) - 127)
* .15;
        if greyscale_image(i, j) < 20
            greyscale_image(i, j) = 0;
        end
    end
end
figure
imshow(1.2 * uint8(greyscale_image));
title('Greyscale')

%getting the RGB channels from the original matrix
red_channel = original_image(:,:,1);
green_channel = original_image(:,:,2);
blue_channel = original_image(:,:,3);

%showing image once
color_image = cat(3,red_channel,green_channel,blue_channel);
```

```
figure;
imshow(uint8(color_image));
title('Original image')

%editting colors- making a red boosted version
red_boosted_image = cat(3,red_channel * 1.5,green_channel ,blue_channel);
figure;
imshow(uint8(red_boosted_image));
title('Red Boosted Version')
```

Code 2

```
%creating I matrix of longer egde size
%image is 4608x3456
I_crop = eye(3456);
I_crop (1:400,1:400) = 0;
I_crop (3056:3456, 3056:3456) = 0;
figure
spy(I_crop);
xlabel(sprintf('Number of nonzero elements: %d', nnz(I_crop)))

%getting the RGB channels from the original matrix- must convert to double
%to multiply
red_channel_crop = I_crop * double(red_channel);
green_channel_crop = I_crop * double(green_channel);
blue_channel_crop = I_crop *  double(blue_channel);
cropped_image = cat(3,uint8(red_channel_crop),uint8(green_channel_crop) ,
uint8(blue_channel_crop));

figure
imshow(cropped_image);
title ('Cropped image');
```

Code 3

```
%reading in an image
Lena = double(imread('Lena.jpg'));
h = height(Lena);
w = width(Lena);
%creating L variable for the number of eigenvectors used
L = 1;

%creating an image showing different effects of blur
subplot(2,3,1)
imshow(uint8(SSA(Lena, L, 1, 1, h, w)));
title('u = v = 1')
subplot(2,3,2)
imshow(uint8(SSA(Lena, L, 4, 4, h, w)));
title('u = v = 4')
subplot(2,3,3)
imshow(uint8(SSA(Lena, L, 7, 7, h, w)));
title('u = v = 7')
subplot(2,3,4)
imshow(uint8(SSA(Lena, L, 10, 10, h, w)));
title('u = v = 10')
subplot(2,3,5)
```

```
imshow(uint8(SSA(Lena, L, 13, 13, h, w)));
title('u = v = 13')
subplot(2,3,6)
imshow(uint8(SSA(Lena, L, 16, 16, h, w)));
title('u = v = 16')


%demonstrating the change in skewing the blur
placement = 1;
figure(2)
for v = 1: 3: 16
    subplot(1,6,placement);
    imshow(uint8(SSA(Lena, L, 1, v, h, w)));
    title("v = " + v)
    placement = placement + 1;
end
```

Code 4

```
%reading in an image
Galaxy_color = double(imread('Galaxy.jpg'));
Galaxy = Galaxy_color(:,:,1) / 3 + Galaxy_color(:,:,2) / 3 + Galaxy_color(:,:,3) / 3;
hG = height(Galaxy);
wG = width(Galaxy);
%creating L variable for the number of eigenvectors used
LG = 1;


%for u = v = 1
%-----------------------------------------------
%Setting window size
uG = 1;
vG = 1;
%Creating the 'X' matrix for the galaxy matrix
XG1 = SSADeconstructor(Galaxy, uG, vG, hG, wG);
%Creating the SVD modification to the galaxy
XG1_new = SVDmodifier(XG1, LG, uG, vG, hG, wG);
G1 = SSAReconstructor(XG1_new, uG, vG, hG, wG);
subplot(2,2,1)
imshow(uint8(G1));
title('u = v = 1')


%for u = v = 4
%-----------------------------------------------
%Setting window size
uG = 4;
vG = 4;
%Creating the 'X' matrix for the galaxy matrix
XG4 = SSADeconstructor(Galaxy, uG, vG, hG, wG);
%Creating the SVD modification to the galaxy
XG4_new = SVDmodifier(XG4, LG, uG, vG, hG, wG);
G4 = SSAReconstructor(XG4_new, uG, vG, hG, wG);
subplot(2,2,2)
imshow(uint8(G4));
title('u = v = 4')



%for u = v = 7
```

```
%-----------------------------------------------
%Setting window size
uG = 7;
vG = 7;
%Creating the 'X' matrix for the galaxy matrix
XG7 = SSADeconstructor(Galaxy, uG, vG, hG, wG);
%Creating the SVD modification to the galaxy
XG7_new = SVDmodifier(XG7, LG, uG, vG, hG, wG);
G7 = SSAReconstructor(XG7_new, uG, vG, hG, wG);
subplot(2,2,3)
imshow(uint8(G7));
title('u = v = 7')

subplot(2,2,4)
imshow(uint8(Galaxy));
title('Original')

%}
%{
%Setting window size
uG = 4;
vG = 6;
%Creating the 'X' matrix for the galaxy matrix
XG = SSADeconstructor(Galaxy, uG, vG, hG, wG);
%Creating the SVD modification to the galaxy
XG_new = SVDmodifier(XG, LG, uG, vG, hG, wG);
G = SSAReconstructor(XG_new, uG, vG, hG, wG);
figure(2)
imshow(uint8(G));
title('Reduced Noise Full Image: u = 4; v = 6')
%}
```

Code 5

```
%reading in an image
Galaxy_color = double(imread('Galaxy-Full.jpg'));
Galaxy_red = 1.05 * Galaxy_color(:,:,1);
Galaxy_green = 1.05 * Galaxy_color(:,:,2);
Galaxy_blue = Galaxy_color(:,:,3);


hG = height(Galaxy_red);
wG = width(Galaxy_red);
%creating L variable for the number of eigenvectors used
LG = 1;
%Setting window size
uG = 4;
vG = 6;

disp('Red...')
%Creating the 'X' matrix for red
XG_red = SSADeconstructor(Galaxy_red, uG, vG, hG, wG);
%Creating the SVD modification to the galaxy
XG_red_new = SVDmodifier(XG_red, LG, uG, vG, hG, wG);
G_red = SSAReconstructor(XG_red_new, uG, vG, hG, wG);

disp('Green...')
%Creating the 'X' matrix for red
XG_green = SSADeconstructor(Galaxy_green, uG, vG, hG, wG);
%Creating the SVD modification to the galaxy
```

```
XG_green_new = SVDmodifier(XG_green, LG, uG, vG, hG, wG);
G_green = SSAReconstructor(XG_green_new, uG, vG, hG, wG);

disp('Blue...')
%Creating the 'X' matrix for red
XG_blue = SSADeconstructor(Galaxy_blue, uG, vG, hG, wG);
%Creating the SVD modification to the galaxy
XG_blue_new = SVDmodifier(XG_blue, LG, uG, vG, hG, wG);
G_blue = SSAReconstructor(XG_blue_new, uG, vG, hG, wG);
disp('Concatonating image...')
%showing image (color)
G_color = 1.2 * cat(3,G_red, G_green, G_blue);
figure;
imshow(uint8(G_color));
title('Color image')
```

Code 6

```
function [new] = SSA(X, L, u, v, h, w)
    Y = SSADeconstructor(X, u, v, h, w);
    X_new = SVDmodifier(Y, L, u, v, h, w);
    new = SSAReconstructor(X_new, u, v, h, w);
end
```

Code 7

```
function [X] = SSADeconstructor(M, u, v, h, w)
    disp('Deconstructing SSA...');
    %getting trajectory matrix size
    p = u * v;
    q = (h - u + 1) * (w - v + 1);

    %setting X/window index to 1
    X_index = 1;

    %creating matrix
    X = zeros(p, q);

    %going through full picture frame from left to right
    for i = 1 : h - u + 1
        %Top to bottom
        for j = 1 : w - v + 1
            %saving points left to right accross window
            for Wi = 1 : u
                %top to bottom accross window
                for Wj = 1 : v
                    %updating the trajectory matrix based on M
                    X((Wi - 1) * u + Wj, X_index) = M(i + Wi - 1, j + Wj - 1);
                end
            end
            %Moving over to the next vector
            X_index = X_index + 1;
        end
    end
end
```

Code 8

```
function [X_new] = SVDmodifier (X, L, u, v, h, w)
    disp('Modifying SVD...');
    %calculating p and q as size of X_new
    p = u * v;
    q = (h - u + 1) * (w - v + 1);


    XXT = X * transpose(X);

    %finding the SVD of XXT
    [U,S,V] = svd (XXT);
    %U and V are the same since dealing with a symmetric matrix
    U_transpose = transpose(U);

    %This code only runs once, as L=1, but will allow using multiple
    %eigenvectors if L > 1
    X_new = zeros(p, q);
    for it = 1:L
        %Each U is only using the first row/column as the first eigenvector
        X_new = X_new + (U(:, it) * U_transpose(it, :) * X);
    end
end
```

Code 9

```
function [M_new] = SSAReconstructor(X, u, v, h, w)
    disp('Reconstructing SSA...');
    %getting window size
    %finding p and q for X size
    p = u * v;
    q = (h - u + 1) * (w - v + 1);

    %setting X/window index to 1
    X_index = 1;

    %creating matrix- third dimension to take care of averaging multiples
    M_new_3D = zeros(h, w, 2);

    %going through full picture frame from left to right
    for i = 1 : h - u + 1
        %Top to bottom
        for j = 1 : w - v + 1
            %saving points left to right accross window
            for Wi = 1 : u
                %top to bottom accross window
                for Wj = 1 : v
                    %Adding the new value to the current value in M_new
                    M_new_3D(i + Wi - 1, j + Wj - 1, 1) = M_new_3D(i + Wi - 1, j + Wj
- 1, 1) + X((Wi - 1) * u + Wj, X_index);
                    %incramenting the second level by one
                    M_new_3D(i + Wi - 1, j + Wj - 1, 2) = M_new_3D(i + Wi - 1, j + Wj
- 1, 2) + 1;
                end
            end
            %Moving over to the next vector
            X_index = X_index + 1;
```

```
        end
    end

    %must now average out the values in the 3 dimensional matrix
    %Use the second layer to do this
    M_new = zeros(h, w);
    for i = 1:h
        for j = 1:w
            %dividing the dividend by the divisor
            M_new(i, j) = M_new_3D(i, j, 1) / M_new_3D(i, j, 2);
            %this makes the lights lighter, and the darks darker
            M_new(i, j) = M_new(i, j) + (M_new(i, j) - 127) * .15;
            %this helps make any pixel close to black completely black
            if M_new(i, j) < 20
                M_new(i, j) = 0;
            end
        end
    end
end
```

REFERENCES

1) The Nynomic Group. (n.d.). Image Noise. Image Engineering. Retrieved April 13, 2022, from https://www.image-engineering.de/library/image-quality/factors/1080-noise
2) The Lenna Story. Lenna History. (n.d.). Retrieved April 13, 2022, from http://www.lenna.org/
3) Rodr´ıguez-Arag´on,, L. J., & Zhigljavsky, A. (2010). SSA change point detection and eye fundus image analysis. *Statistics and Its Interface*, *3*, 419–426. https://doi.org/10.1201/b19140-9