Team 1 Project Report

Table of Contents

1: Project Purpose

The use of technology to store/share images has increased greatly since its original creation. However, with society's increasing reliance on such mediums, individuals' ability to crack and steal images, the need for encryption and decryption methods has become increasingly necessary. The idea of encryption, or cryptography in a more general term, has been practiced for centuries, historically being used to encode messages.

The use of encryption is commonly used to protect money and personal information. However, a poor selection of encryption techniques may leave data vulnerable by simple exploits. Here, we demonstrate a simple encryption method and a more complex one and show how the more complex method does a better job of concealing the obfuscated image.

XOR-ing data against a known PN sequence of bits is a common practice for scrambling data (when the data and the sequence is one-for-one). So, each bit either is/or is not flipped. When the PN sequence is faster than the data (so each data bit is broken up into multiple chips), then the data has been spread in the frequency domain. When the bit rate is much faster than the PN sequence, then the data is frequency hopping and the PN sequence controls the frequency hop (Sklar).

Even when you have an image, there are different methods to extract information. One technique to simplify the work is to convert the color image (with 3 different primary colors: red, green, blue) into a single value, thus reducing the processing by a factor of three. Thus, converting an RGB image into gray scaling improves the detection of edges (Saravanan).

Finally, the use of the Sobel filter to further process an image, and enhance contrast helps to select the brightest spot in the image (Vincent). The description of Sobel notes that filtering out noise (with a Gaussian filter) is recommended, to prevent possible false detection of edges.

The use of edge detection is often used in machine vision applications to help identify the boundaries of objects for a robot to handle (Nixon).

2: Project Overview and Methods

Scrambling the image using a PN sequence helps to scatter the pixel values. Using a keystring for the key generator shows a distinct pattern in the encrypted image. That is, the phrase “COME AND GET YOUR LOVE” has 18 letters, and a pattern seems to appear every 18 columns and every 18 rows. The contents of the keystring are irrelevant, except that there must be 18 letters for the key generator to work.

The use of a ciphering technique with an extremely long length means that it is less likely for someone to decrypt, when they only have an encrypted image available. This makes the LFSR akin to a one-time pad (Sklar); since the pattern will not repeat for an n-bit shift register for 2^n-1 bits. For this program, we are only using a 16-bit shift register (for 65,535 bits). However, it would be easy to scale up to 32-bits (for over 4 billion bits).

After decrypting the image, we reduce the task complexity by reducing the red, green, blue pixels into a single gray scale. Thus, later image processing only needs to act on gray; not on three colors.

We use a Gaussian filter on the image to reduce the noise, which might cause false edge detection with the Sobel filter. Finally, we use the Sobel filter to find the edges of the image. That is, it does a 2-D spatial gradient measurement across the image by convolving at each pixel, a 3 x 3 matrix (the surrounding pixels) with a 3 x 3 mask for the X-direction gradient, then, with the 3 x 3 mask flipped for the Y-direction gradient. The gradient magnitudes for X and for Y are combined to yield a single magnitude. The magnitude represents an edge.

3: Discussion of Algorithm Design

3.1: Overview

Program flowchart is shown in Figure 1 below. More details in the corresponding sections.

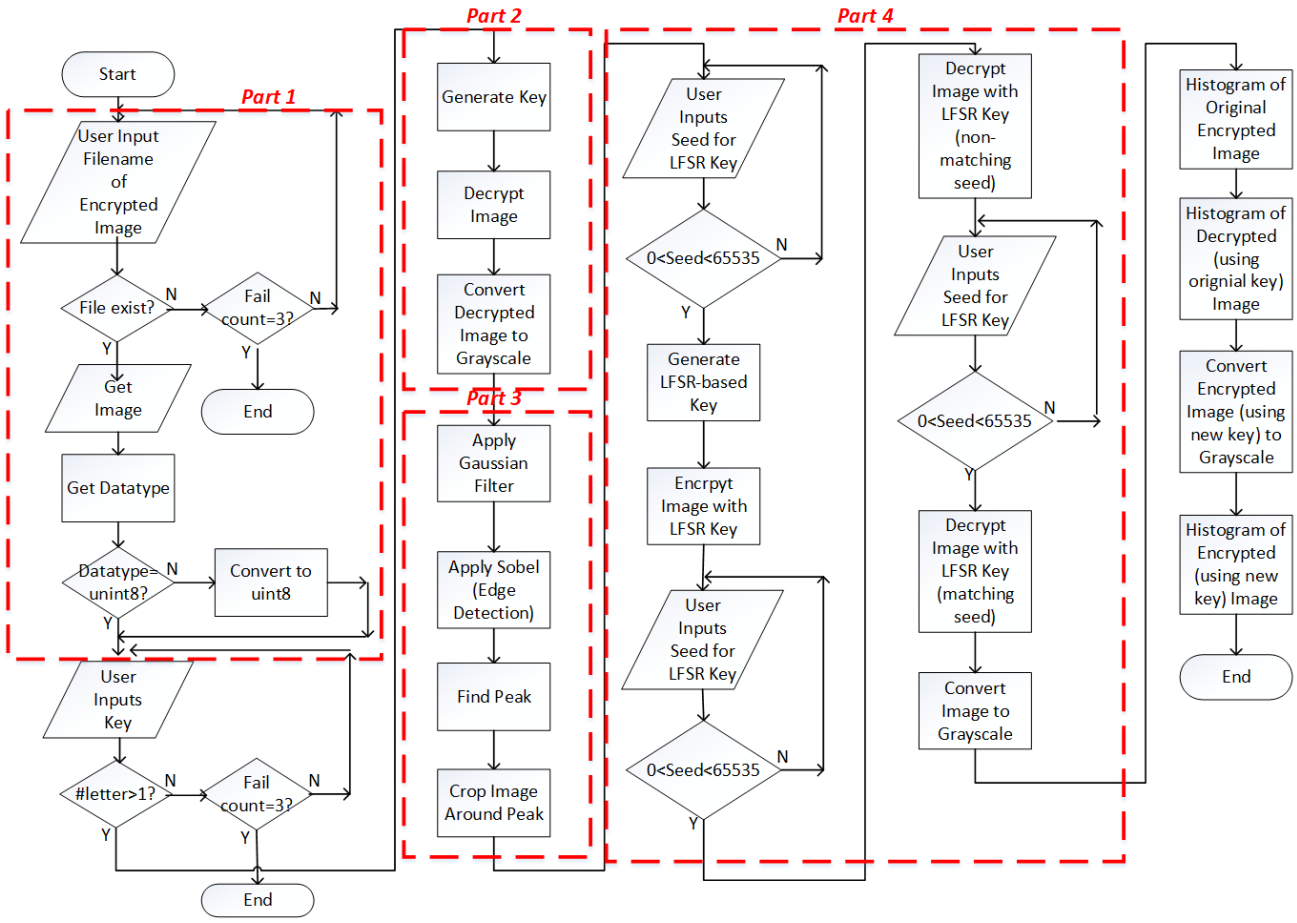


Figure 1 Python Program Flowchart

.

3.2: Code/Methods Explained — Part 1 Image Importation

To begin learning how to encrypt an image file, our group first did some background research on what kind of data types these files utilized in order to fully understand how these file types worked. We found that PNG image files utilize the float32 data type, ranging from -3.4E+38 to +3.4E+38. JPG image files on the other hand use uint8 units, ranging from 0 to 255. JPG is actually a lossy compression method that utilizes Discrete Cosine Transform (DCT), working best with colors and grayscale. Similarly, TIFF files also work with uint8 units and range from 0 to 255. Our program checks the datatype of the imported file to make sure it is uint8 and uses the *astype* method to cast the date array to uint8 if it is not.

Our user-defined function, “get_image”, asks the user to input the filename of the encrypted image they would like to decrypt. It will then check to confirm the file exists. If it does not, the user has 3 tries to enter the filename correctly before the program exits. The code grabs the image from files using the *imread* method from the *matplotlib.pyplot* library and converts it to a 3d array, preparing it for later use. As explained later, this will be the most universal method of the entirety of the code as this image array is called back several times, for both comparison and manipulation. Figure 2 shows our inputted image.

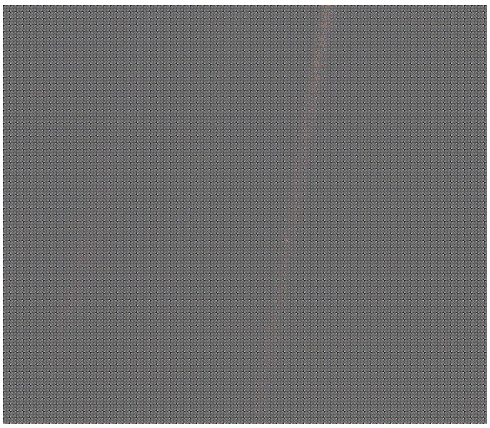


Figure 2. Original Encrypted Image

3.3: Code/Methods Explained — Part 2 Image Decryption

The image imported from the previous section was encrypted using XOR cipher whereby 2 binary strings (in our case one string is the decrypted image and the second, the key array) are compared utilizing the XOR operation whereby the output is a "0" if the two bits are the same or a "1" if the two are different. The first thing we needed to do was to generate a key array using the string "COME AND GET YOUR LOVE," given by the project's instructions. This proved to be more challenging than expected. First, before we can count the number of letters using the len function, we need to strip spaces. Originally, we used the strip method; however, upon further investigation and debugging we learned this only removes spaces at the beginning and at the end of the string and does not remove spaces if in the middle of the string. We resorted to using the replace method by replacing spaces (" ") with an empty character (""). And finally, we had to make sure the key array was the same dimensions as the image as the XOR operation operates on a bit-by-bit basis. We could have looped on all the characters across all the dimensions of the image and converted them to a binary representation prior to performing the XOR operation on each bit but we luckily found the ^ operator. Alternatively, we would have used the *bitwise_xor* method within the *numpy* library. This method does the bitwise XOR of two integers arrays.

Using the encrypted NASA image given in the project instructions (link below), we tested our decryption method and visually compared the image to the original. Originally our decrypted image (Figure 3) did not match what was in the project document ("ENGR 13300

Python Group Project Fall 2021" revised 10/5/2021) (Figure 6). But we found the unencrypted image from NASA (Figure 4) ("Pale Blue Dot Revisited") and visually it was really close so we felt confident we were on the right track.

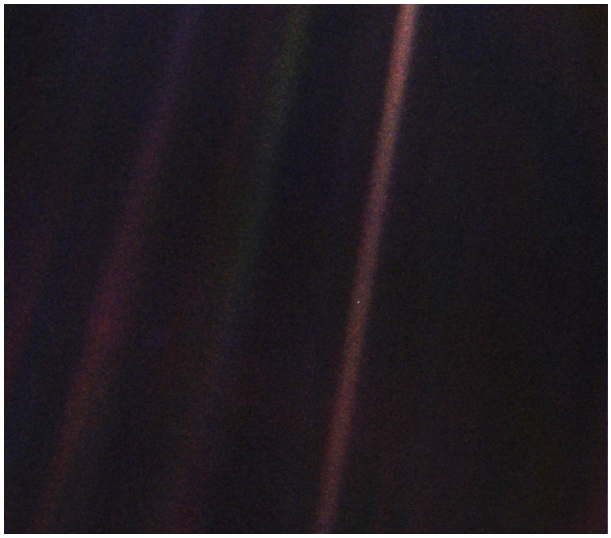


Figure 3. Our Decrypted Image

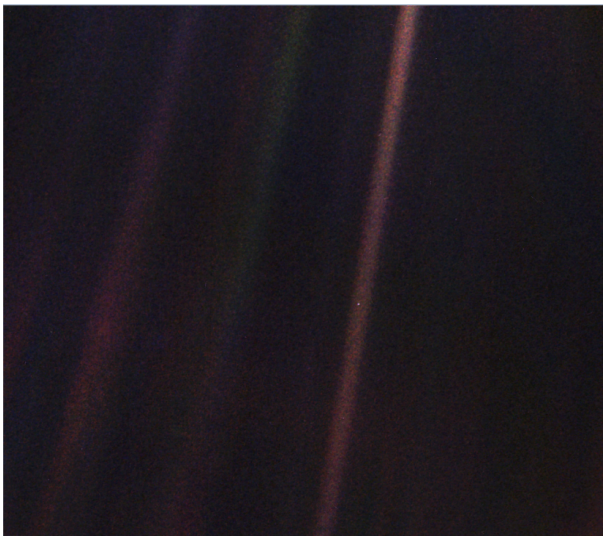


Figure 4. NASA 'Pale Blue Dot' Image

We fixed this by converting to grayscale using the formula

grayscale = 0.2989 * red + 0.5870 * green + 0.1140 * blue (Saravanan)

And our resulting image (Figure 4) very closely matched what was expected (Figure 6)

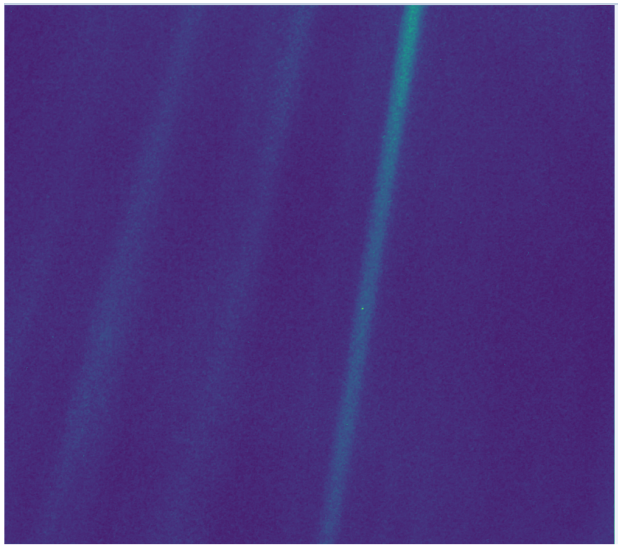


Figure 5. Our Image in GrayScale

Figure 6. Expected Decrypted Image

With no noticeable errors, the decryption method seems to have run as planned, outputting a pale blue dot image.

Our XOR cipher's runtime increases as the number of pixels increases due to an increased number of XOR cipher computations. Ideally, the key length should increase simultaneously. While the XOR cipher is relatively easy to implement into our code, its short repeat length has proven to lead to weaker encryption and requires a relatively long length about the same as the image inputted.

3.4: Code/Methods Explained — Part 3 Feature Detection

The next part of our code was to find the location of Earth from the encrypted image. This took several steps. The first was to smooth out the noise using a Gaussian filter using our "blur" function. The Gaussian filter was implemented using the *copy*, *roll*, and *array* methods from the *numpy* library to implement the convolution of the image with a 3x3 Gaussian kernel (Getreuer)

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Figure 7 shows our output from this step.

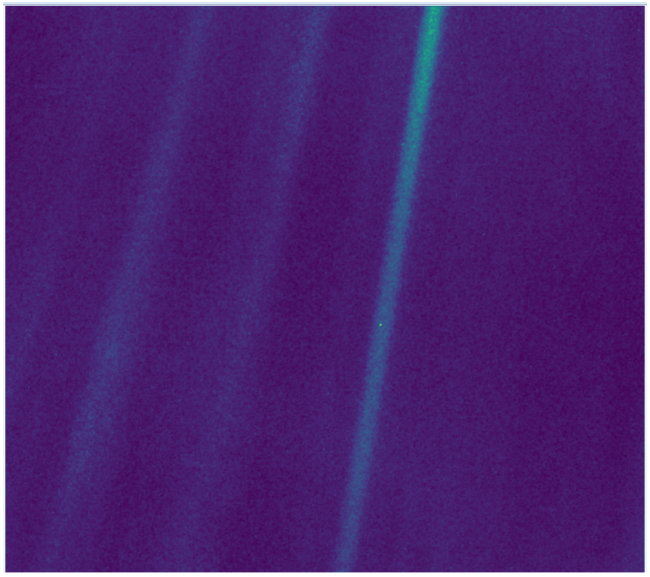


Figure 7. Output of Gausian Filter

The next step was to calculate a gradient of the image using the Sobel edge detector using our "sobel_filter" function. Essentially, we're looking for discontinuities in the first derivative by using the Sobel operator or 2D derivative mask on the image (Vincent). In this part of the code,

we used the *array, sum,* and *sqrt* methods from the *numpy* library. We first convolved the image with the Sobel filter in both the x- and y- dimension and combining the 2 outputs using

$$G = \sqrt{G_x^2 + G_y^2}$$

before normalizing the output to be between 0 and 255 (Vincent). Figure 8 shows our output from this step.

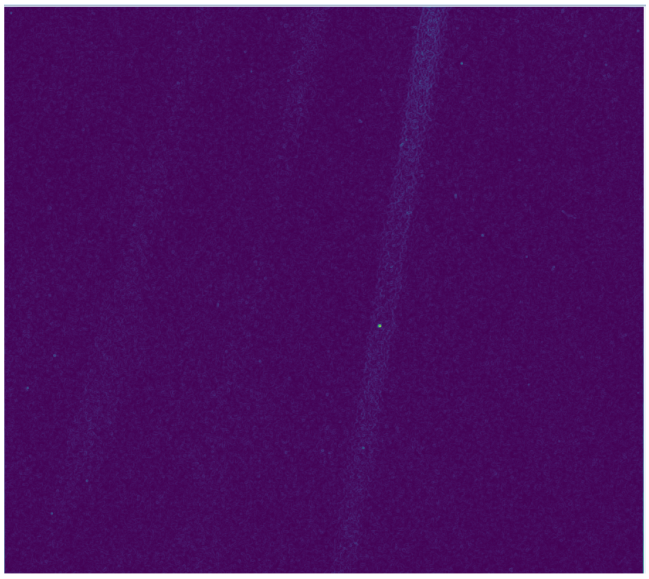


Figure 8. Output of Sobel Filter

The theoretical range of a gradient map is 0 to -255 if RGB are all at 255. Our actual range was 0.0 to 202.2894.

Once we get the gradient map, we can search for the location of Earth using our "find_peak" function. Essentially we search every element in the gradient map for the brightest one (largest value) and save the row and column of this value. This is the location and we can now crop our image around the location of Earth using our "crop_image" function making sure Earth is at the center of the cropped image. Figure 8 shows the cropped image with Earth at the center.

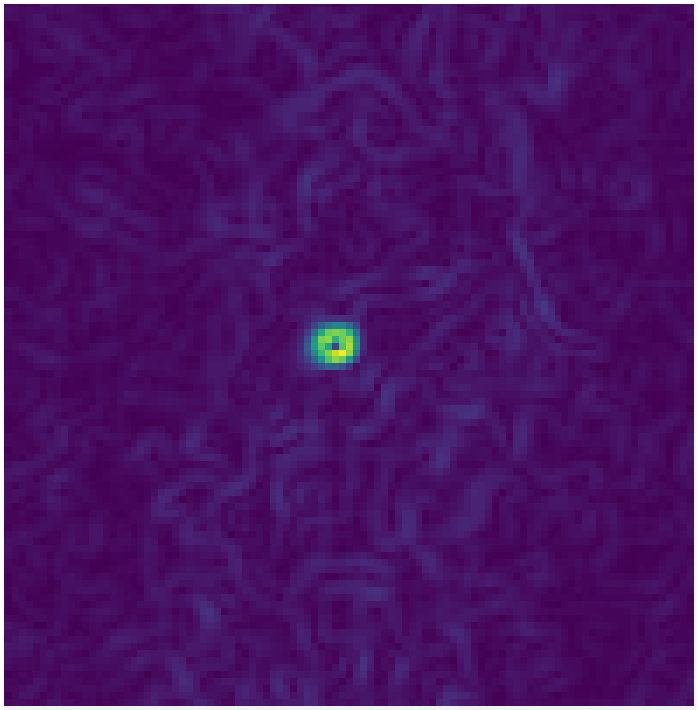


Figure 9, Cropped Image Centering Around Earth

We did not find the Gaussian filter to affect our results and we consistently found Earth at the same location (row=648, column=765) although our research showed blurring should have helped as noise will affect the derivative, so smoothing (Gaussian filter) the image first is recommended.

3.5: Code/Methods Explained — Part 4 Image Encryption

We began by looking at the histogram of the encrypted and decrypted image shown in Figures 10 and 11.

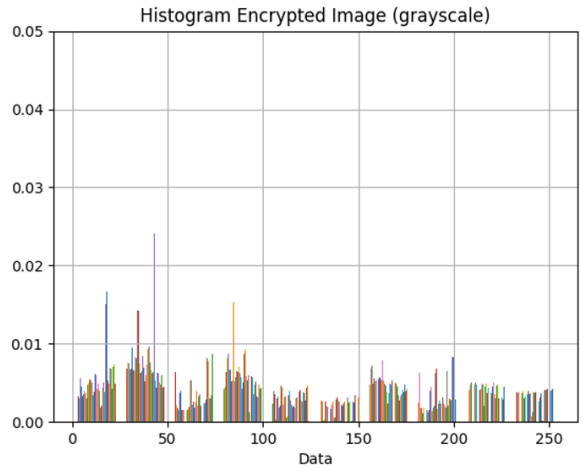


Figure 10. Encrypted (grayscale) Histogram

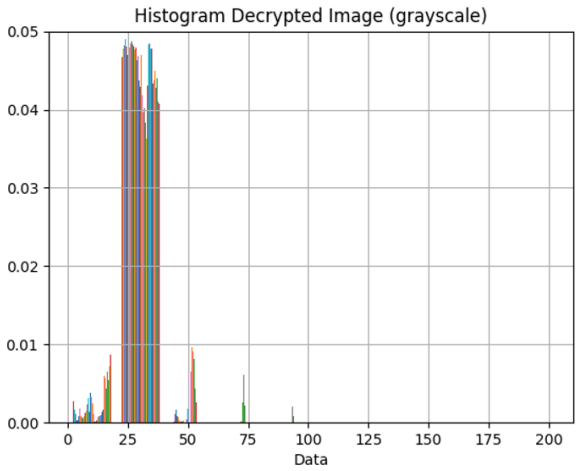


Figure 11. Decrypted (grayscale) Histogram

Our first observation is the large cluster in the decrypted image and even though the encrypted image histogram is more spread out, we still see peaks. XORing the data should randomize all bits, and looking at the histogram of the encrypted image should show a flat histogram. So, the generated key should have the same number of ones and zeros and the distribution of 8-bit values should be evenly distributed. We can possibly use a randomness test to test how well a key generator is working; the more random the better. Histogram shows distribution of the data (how often each different value in a set of data occurs) so the more spread out and flat the better.

A random seed was recommended as a possible key generator. After researching, we learned some generators are better than others and learned about the Linear Feedback Shift Register (LFSR) algorithm (Goresky).

We chose to implement a 16-bit shift register with taps at 4, 13, 15, and 16 which gives a maximal length sequence for 16-bit shift registers (Goresky). Our function"lfsr_keygen" first initializes the shift registers with a seed inputted from the user. Next, a new bit to the key is generated by XOR shift registers 4, 13, 15, and 16 as shown in Figure 12. The register values are then shifted (bit in shift register 15 moves to shift register 16, bit in shift register 14 moves to shift register 15, and so on until bit in shift register 1 moves into shift register 2 and the new bit is shifted into shift register 1). This process is repeated until the needed number of bits needed for the new key is generated.

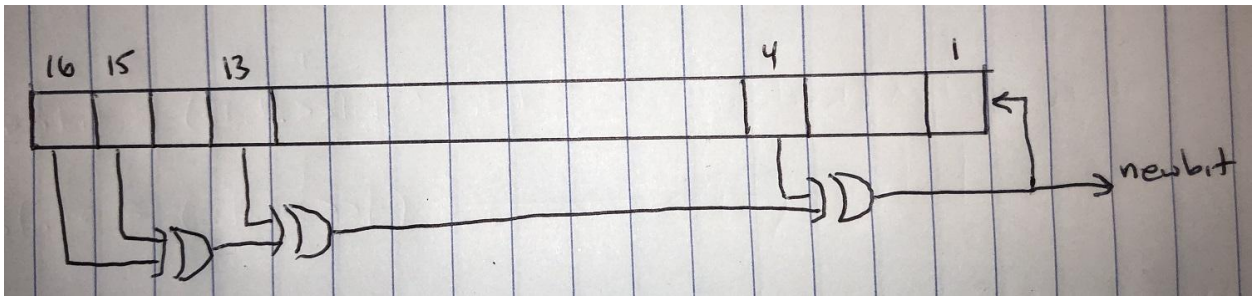


Figure 12 16-bit LFSR

Figure 13 shows the encrypted image using the new key and the corresponding histogram in Figure 14.

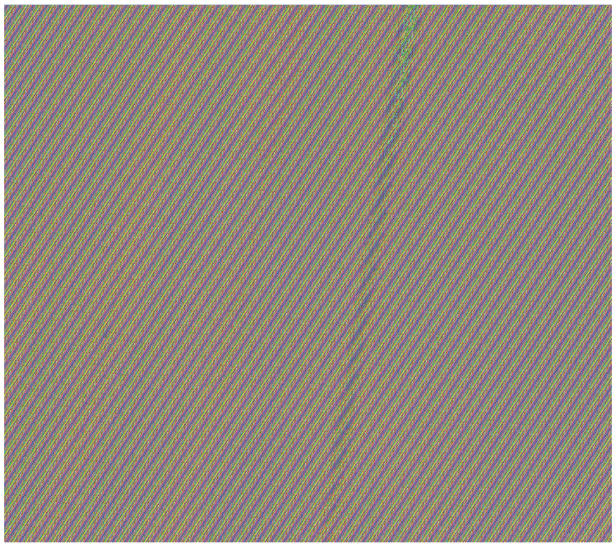


Figure 13. Encrypted (New Key) Image

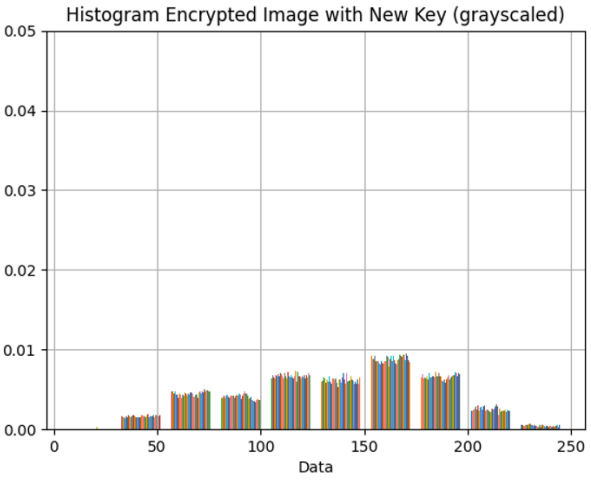


Figure 14. Encrypted (New Key) Histogram

Figures 15 and 16 show decrypted images when the wrong seed is used to generate the key and when the correct seed is used; respectively.

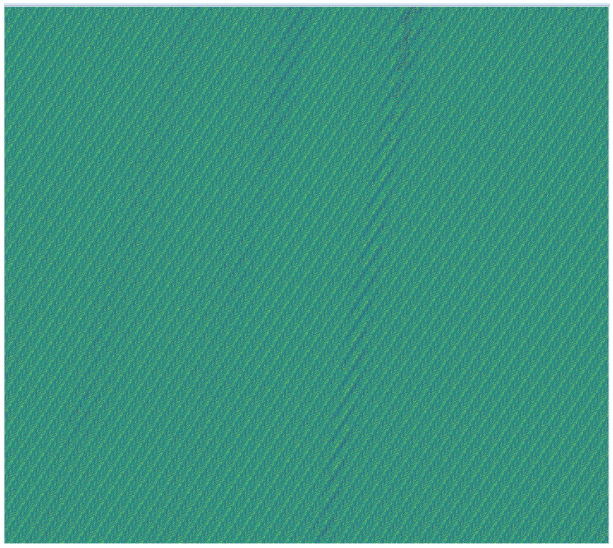


Figure 15. Decrypted (Wrong Seed) Image

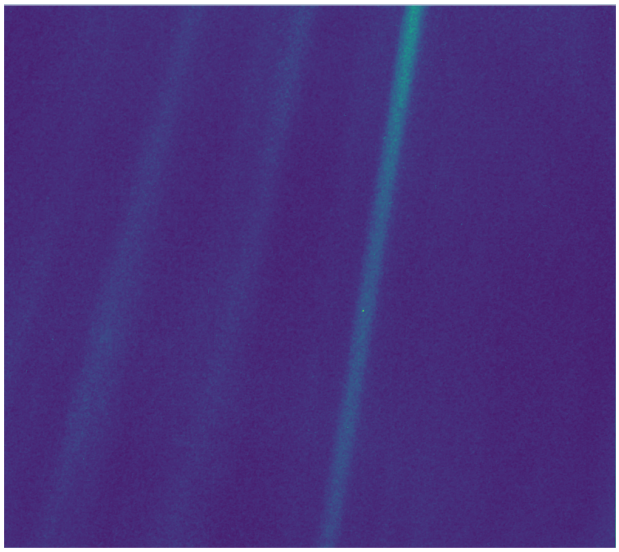


Figure 16. Decrypted (Correct Seed) Image

Compared to Figure 10 (histogram of original encrypted image), Figure 14 seems to be a bit more spread out and there are no significant peaks and we feel this is a better key in comparison. Furthermore, the original key generator will generate the same XOR pattern regardless of key string (as long as there are 18 letters). This length can be guessed by examining the pixel pattern along the edge of the image. Using a LFSR-based key generator does not have as frequent repetition especially if a maximal length sequence is used. It makes for a stronger key that much more difficult key to guess.

3.6: Code/Methods Explained — Part 5 Symmetric-Key Cryptosystem

We met with group 13 to demo our encryption and decryption. Using their given seed, we were able to decipher their image using our own decryption function. While there were a few bugs running the decryption method the first time around, we were able to run it the second time without trouble. (See Figures 17-20 below for before and after images). After consulting our partner team again, we confirmed that our outputted images were the same as the ones that they had originally encrypted.

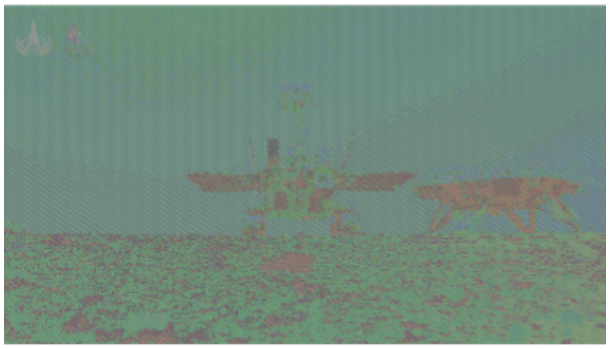


Figure 17. Encrypted Image 1

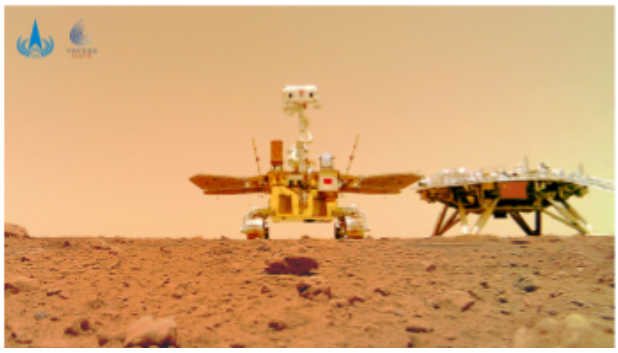


Figure 18. Decrypted Image 1

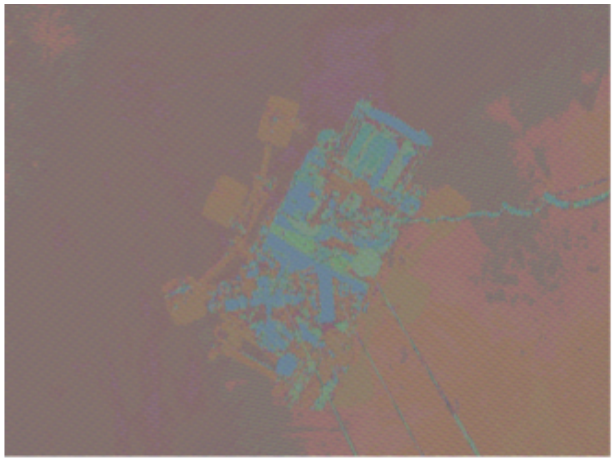


Figure 19. Encrypted Image 2

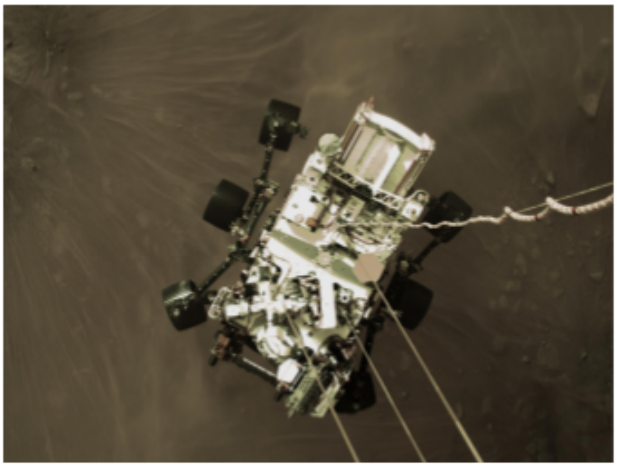


Figure 20. Decrypted Image 2

4: References

Getreuer, Pascal. “A Survey of Gaussian Convolution Algorithms.” Image Processing On Line, vol. 3, 2013, pp. 286–310., https://doi.org/10.5201/ipol.2013.87.

Dumoulin, Vincent, and Francesco Visin. “A Guide to Convolution Arithmetic for Deep Learning.” *ArXiv.org*, Cornell University, 11 Jan. 2018, https://arxiv.org/abs/1603.07285.

Vincent, Olufunke, and Olusegun Folorunso. “A Descriptive Algorithm for Sobel Image Edge Detection.” *Proceedings of the 2009 InSITE Conference*, 2009, https://doi.org/10.28945/3351.

“Pale Blue Dot Revisited.” Edited by Tony Greicius and Naomi Hartono, NASA, NASA, 2020, https://www.jpl.nasa.gov/images/pale-blue-dot-revisited.

Saravanan, C. “Color Image to Grayscale Image Conversion.” 2010 Second International Conference on Computer Engineering and Applications, 2010, https://doi.org/10.1109/iccea.2010.192.

Sklar, Bernard, and Fredric Harris. *Digital Communications: Fundamentals and Applications*. Pearson, 2021.

Goresky, M. and A. M. Klapper, A.M. "Fibonacci and Galois representations of feedback-with-carry shift registers," in *IEEE Transactions on Information Theory*, vol. 48, no. 11, pp. 2826-2836, Nov. 2002, doi: 10.1109/TIT.2002.804048.

Nixon, Mark S., and Alberto S. Aguado. *Feature Extraction & Image Processing for Computer Vision*. CPI Anthony Rowe, 2016.

5: Appendix

5.1: User Manual

Protect Earth Python User Manual

Introduction

Many probes have been launched from Earth to explore our solar system and beyond. These probes will transmit images back to Earth to improve our understanding of the universe and our place within it. Some of these images are focused on the Earth and show our location in the galaxy.

To prevent Thanos from determining our location from these images, we explore the possibility of encrypting the images before transmission, so even if an image is intercepted, he will not be able to use them to locate the Earth. This program explores several levels of encryption to determine a preferred method which prevents "reverse engineering" the encrypted image.

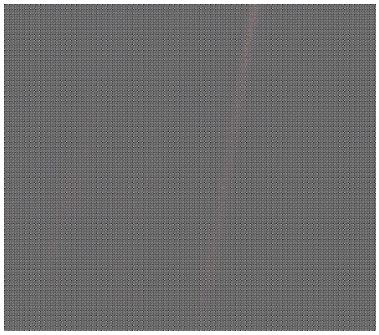
This program demonstrates XOR ciphering to obscure the image data. The cipher key generation uses two methods and their ability to obscure the image is compared using a simple grayscale to demonstrate whether a method can obfuscate the position of the Earth.

Simple Key Cipher

The program begins by demonstrating the ease of decrypting an image using a simple keystring cipher. This keystring is used to generate a key array which simply cycles through the keystring length.

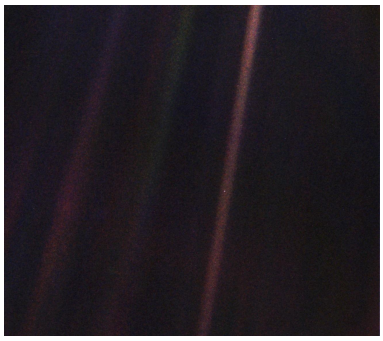
Execution begins with:

```
> py fall_2021.py
Getting encrypted image...
      What is the filename of the image? Pale_Blue_Dot_Encrypted.tiff
      Image is uint8 format
      row=1152 col=1304 depth=3
      [Exit image to continue]
```



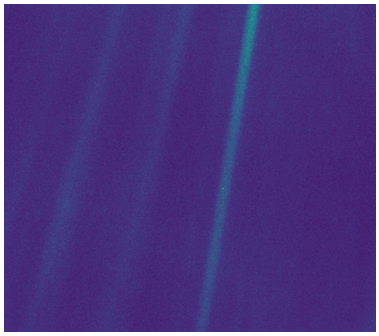
Note: This part will show the encrypted image. If the file does not exist, the user will be asked to enter the filename again. Killing the image window is necessary for the program to continue.

```
Decrypting image...
      Enter key for decryption: COME AND GET YOUR LOVE
      key=COMEANDGETYOURLOVE len=18
      [Exit image to continue]
```

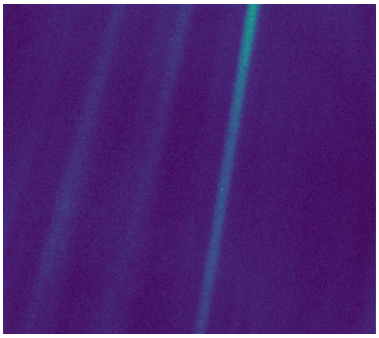


Note: The key needs to be at least 2 letters or the user will be asked to try again. The decrypted image should be displayed. Must kill the image window for the program to continue.

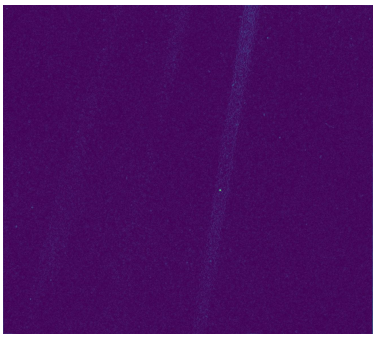
```
Grayscaling Decrypted Image...
      [Exit image to continue]
Blurring image...
      [Exit image to continue]
Applying Sobel (edge detector)... please be patient
      [Exit image to continue]
Cropping image around peak pixel...
      peak at (648,765)
      crop row=598 col=715
      [Exit image to continue]
```



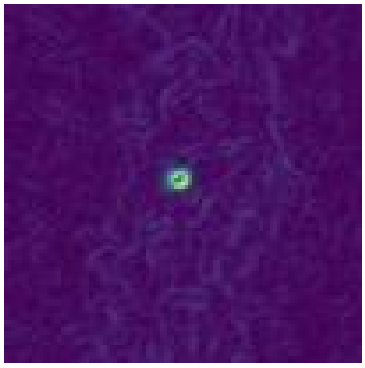
Gray scale



Blurred



Sobel filtered



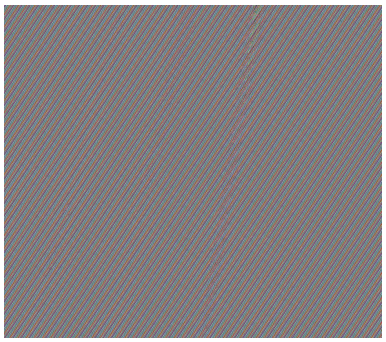
Cropped

In the above, the next set of images shows the processing steps on the decrypted image The last step shows the results of a simple search in the Sobel filtered image. The small cropped image is the brightest pixel and some of the pixels around this pixel.

Using a Linear Feedback Shift Register for Key Generation

The next steps demonstrate using a Linear Feedback Shift Register (LFSR) to generate the encryption key.

```
Encrypting image with new key...
      Enter seed (1-65535)? 12
      Generating key...
      [Exit image to continue]
```

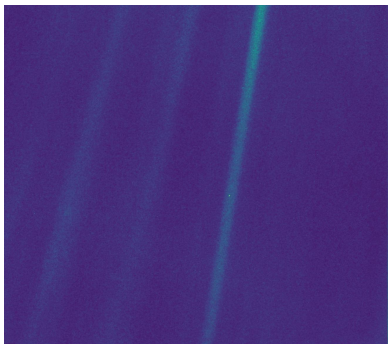


Here, we have encrypted the previously decrypted image using the LFSR. You should observe no clear pattern in the scrambled image

```
Decrypting image with new key but different seed...
      Enter seed (1-65535)? 13
      Generating key...
      [Exit image to continue]
```

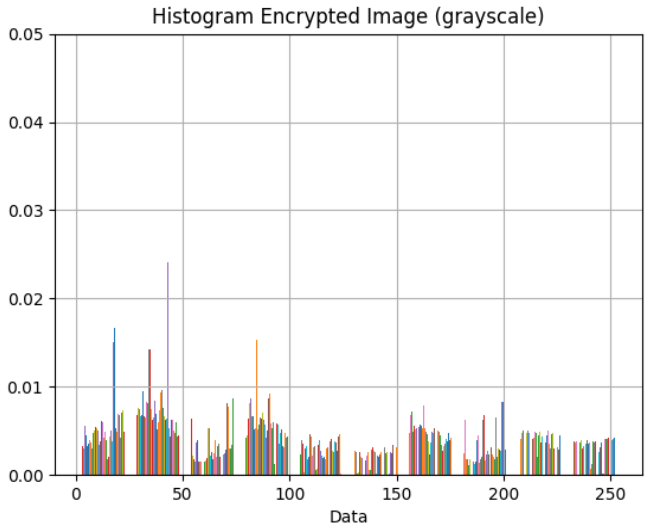
Here we show that a different seed does not decrypt an image using LFSR. This is in marked contrast to the keystring cipher, which any string with 18 letters would decrypt.

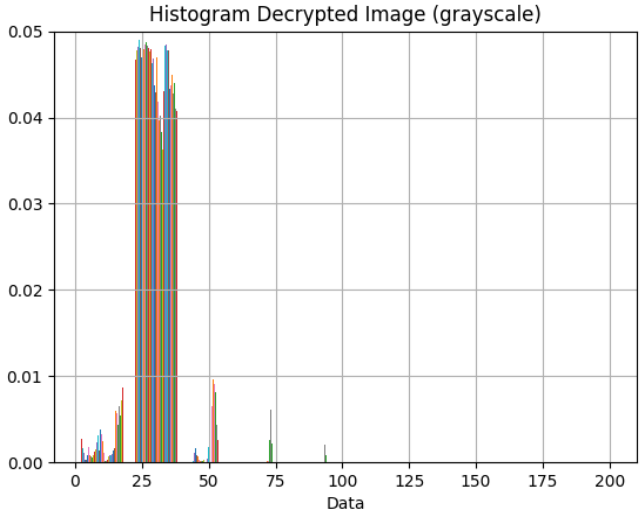
```
Decrypting image with new key and original seed...
      Enter seed (1-65535)? 12
      Generating key...
      [Exit image to continue]
```

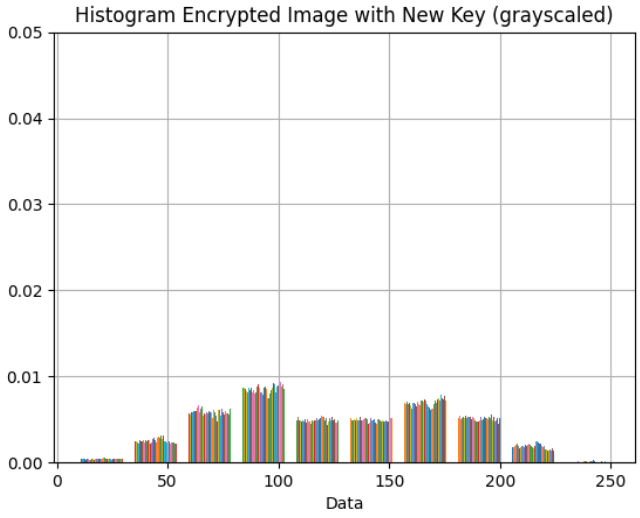


Now, we show a properly decrypted image

```
Plotting histograms using grayscale
      Original encrypted image
      [Exit image to continue]
      Decrypted image
      [Exit image to continue]
      Encrypted image using new key
      [Exit image to continue]
```







Finally, we generate some histograms that show the spread of grayscale values. Comparing the simple cipher to the actual image to the LFSR cipher. The LFSR cipher shows a flatter histogram, which means there is less information that can be gleaned. One issue that remains outstanding is the apparent gaps in the encrypted image histograms. We assume that it is an artifact of only using 16-bits on the LFSR.

5.2: Project Management Plan

Our group decided to take a more split approach to this project, originally splitting into four parts per the document's request. Geoffrey was assigned to part 1, Elliot to part 2, Christine to part 3, and Cat to part 4. However, upon testing Elliot's decryption method, we noticed several bugs, prompting Cat to take over part 2, and Elliot part 4.

Part 1 and 3 went as planned, with Geoffrey and Christine sending Elliot usable code to implement into our larger file later. Approaching expected submission of the project, Elliot compiled all the code into one document, noticing a few major issues that caused the code not to run, the first being a faulty decryption function and several other minor details in other parts of the code. Utilizing outside help, Geoffrey and Elliot were able to fix the bugs interrupting our code and ran a successful test with group 13.

5.3: Discussion of Design Process

For the most part, our group left each part to its respective author, interacting with each other once in a while to discuss minute details of our parts. Most of the parts were written using fairly standard procedure, following the guides given in the instructions and found online. The part most up to interpretation, part 4, ended up utilizing an LFSR algorithm to create a pseudo random key. Several algorithms were analyzed to find the best method of randomizing our key possible, deciding on LFSR in the end.

5.4: Python Code

```
def main():
      # Get image
      print("Getting encrypted image...")
      image = get_image()
      row, col, depth = len(image), len(image[0]), len(image[0][0])
      print('     row='+str(row),'col='+str(col),'depth='+str(depth))
      plt.imshow(image)
      plt.title("Original Encrypted Image")
      plt.ylabel("Row")
      plt.xlabel("Col")
      print("     [Exit image to continue]")
      plt.show()

      # Decrypt image using XOR cipher
      print("Decrypting image...")
      # Get key
      key = get_key()
      keylen = len(key)
      print('     key='+key,'len='+str(keylen))
      # Generate key
      keyarr = keygen(row, col, depth, key, keylen)
      decrypt_image = np.bitwise_xor(image, keyarr)
      filename = 'decryptedImage.tiff'
      plt.imshow(decrypt_image)
      plt.imsave(filename, decrypt_image)
      plt.title("Decrypted Image")
      plt.ylabel("Row")
      plt.xlabel("Col")
      plt.imsave(filename, decrypt_image)
      print("     [Exit image to continue]")
      plt.show()

      # Convert decrypted to gray scale and plot
      print("Grayscaling Decrypted Image...")
      grayscale = rgb2gray(decrypt_image)
      filename = 'grayImage.tiff'
      plt.imshow(grayscale)
      plt.title("Decrypted Image (Grayscale)")
      plt.ylabel("Row")
      plt.xlabel("Col")
      plt.imsave(filename, grayscale)
      print("     [Exit image to continue]")
      plt.show()

      # Convert to float64
      floatscale = np.float64(grayscale)

      # Apply gaussian filter
      print("Blurring image...")
      blurred = blur(floatscale)
      filename = 'blurImage.tiff'
      plt.imshow(blurred)
      plt.title("Blurred Image")
      plt.ylabel("Row")
```

```
plt.xlabel("Col")
plt.imsave(filename, blurred)
print("     [Exit image to continue]")
plt.show()

# Apply sobel (edge detector)
print("Applying Sobel (edge detector)... please be patient")
edges = sobel_filter(blurred)
filename = 'edgeImage.tiff'
plt.imshow(edges)
plt.title("Sobel Image")
plt.ylabel("Row")
plt.xlabel("Col")
plt.imsave(filename, edges)
print("     [Exit image to continue]")
plt.show()

# Crop image around peak pixel
print("Cropping image around peak pixel...")
# Search for peak pixel
hi_row, hi_col = find_peak(edges)
print('     peak at (' + str(hi_row) + ',' + str(hi_col) + ')')
#cropped = crop_image(decrypt_image, hi_row, hi_col, depth)
cropped = crop_image(edges, hi_row, hi_col)
filename = 'croppedImage.tiff'
plt.imshow(cropped)
plt.title("Cropped Image")
plt.yticks([0, 20, 40, 60, 80, 100], [str(hi_row-50),str(hi_row-50+20),
str(hi_row-50+40),str(hi_row-50+60),str(hi_row-50+80),
str(hi_row-50+100)])
plt.ylabel("Row")
plt.xticks([0, 20, 40, 60, 80, 100], [str(hi_col-50),str(hi_col-50+20),
str(hi_col-50+40),str(hi_col-50+60),str(hi_col-50+80),
str(hi_col-50+100)])
plt.xlabel("Col")
plt.imsave(filename, cropped)
print("     [Exit image to continue]")
plt.show()

# Encrypt image using new key
#encrypt_image = np.bitwise_xor(image, keyarr)
print("Encrypting image with new key...")
# Generate LFSR-based key
seed = "-1"
while int(seed) < 1 or int(seed) > 65535:
seed = input('    Enter seed (1-65535)? ')
print("     Generating key...")
keyarr = lfsr_keygen(row, col, depth, int(seed))
encrypt_image = decrypt_image ^ keyarr
filename = 'encryptedNew.tiff'
plt.imshow(encrypt_image)
plt.title(" Encrypt using New Key and Seed="+seed)
plt.ylabel("Row")
plt.xlabel("Col")
plt.imsave(filename, encrypt_image)
print("     [Exit image to continue]")
plt.show()
```

```
# Decrypt image using new key but different seed
# If seed not same as before, will not succesfully decrypt.  Try it.
print("Decrypting image with new key but different seed...")
seed2 = "-1"
while int(seed2) < 1 or int(seed2) > 65535:
seed2 = input('    Enter seed (1-65535)? ')
print("     Generating key...")
keyarr2 = lfsr_keygen(row, col, depth, int(seed2))
decrypt_image2 = encrypt_image ^ keyarr2
grayscale2 = rgb2gray(decrypt_image2)
filename = 'decryptedNewDiffSeed.tiff'
plt.imshow(grayscale2)
plt.title("Decrypted using New Key and Seed="+seed2+" (Grayscale)")
plt.ylabel("Row")
plt.xlabel("Col")
plt.imsave(filename, grayscale2)
print("     [Exit image to continue]")
plt.show()

# Decrypt image using new key and seed
print("Decrypting image with new key and original seed...")
seed3 = "-1"
while int(seed3) < 1 or int(seed3) > 65535:
seed3 = input('    Enter seed (1-65535)? ')
print("     Generating key...")
keyarr3 = lfsr_keygen(row, col, depth, int(seed3))
decrypt_image3 = encrypt_image ^ keyarr3
grayscale3 = rgb2gray(decrypt_image3)
filename = 'decryptedNewSameSeed.tiff'
# Convert to gray scale and plot
plt.imshow(grayscale3)
plt.title("Decrypted using New Key and Seed="+seed3+" (Grayscale)")
plt.ylabel("Row")
plt.xlabel("Col")
plt.imsave(filename, grayscale3)
print("     [Exit image to continue]")
plt.show()

# Plot histograms using grayscale
yaxis = 0.05
print("Plotting histograms using grayscale")
print("     Original encrypted image")
print("     [Exit image to continue]")
grayscale_encrypted = rgb2gray(image)
plot_histogram(grayscale_encrypted,
"Histogram Encrypted Image (grayscale)", yaxis, "fig1")
print("     Decrypted image")
print("     [Exit image to continue]")
plot_histogram(grayscale, "Histogram Decrypted Image (grayscale)",
yaxis, "fig2")
print("     Encrypted image using new key")
print("     [Exit image to continue]")
grayscale_encrypted_lfsr = rgb2gray(encrypt_image)
plot_histogram(grayscale_encrypted_lfsr,
"Histogram Encrypted Image with New Key (grayscaled)",
yaxis, "fig3")
```

```
# End Main


# Get Image Filename and Load into Array
def get_image():
      for file_tries in range(3):
      image_name = input('     What is the filename of the image? ')
      try:
            f = open(image_name)
            f.close()
            break
      except FileNotFoundError:
            print("     Error. No such file, ", end="")
            if file_tries == 2:
            print("exiting")
            exit()
            print("try again")
            print("")

      image = plt.imread(image_name)[:,:,:3]
      # Check Datatype and Convert if Needed
      if image.dtype == np.uint8:
      print("     Image is", image.dtype, "format")
      else:
      print("     Image is", image.dtype, " format. Converting to uint8...")
      image = image.astype(np.uint8)
      return image

# RGB to Grayscale
def rgb2gray(rgb):
      r, g, b = rgb[:,:,0], rgb[:,:,1], rgb[:,:,2]
      grayscale = 0.2989 * r + 0.5870 * g + 0.1140 * b
      return grayscale

# Get Key
def get_key():
      for tries in range(3):
      key = str(input('  Enter key for decryption: '))
      key=key.replace(" ","")  # Remove whitespace
      if len(key) > 1:
            break;

      print("     Error. Key needs to be at least 2 letters, ",end="")
      if (tries < 2):
            print("try again")
            print("")
      else:
      print("exiting")

      return key

# Generate Symmetric Key
def keygen(row, col, depth, key, keylen):
      A = np.zeros((row, col, depth), dtype=np.uint8)
      for i in range(row):
      for j in range(col):
            for k in range(depth):
```

```
            a = (i*j) % keylen
            A[i][j][k] = a
      A = A*(256//keylen)
      return A

def lfsr_keygen(row, col, depth, seed):
      A = np.zeros((row, col, depth), dtype=np.uint8)
      state = seed
      for i in range(row):
      for j in range(col):
            for k in range(depth):
            new_8bit = 0
            for b in range(8):
                  # Calculate the new bit by XOR'ing bits: 4, 11, 14, 15
                  newbit = (state ^
                              (state >> 4) ^
                              (state >> 11) ^
                              (state >> 14) ^
                              (state >> 15)) & 1
                  # Shift all bits to the right and
                  # add the new bit on the left
                  state = (state >> 1) | (newbit << 15)
                  new_8bit = new_8bit | (newbit << b)
            A[i][j][k] = new_8bit
      return A

def blur(a):
      kernel = np.array([[1.0,2.0,1.0], [2.0,4.0,2.0], [1.0,2.0,1.0]])
      kernel = kernel / np.sum(kernel)
      arraylist = []
      for y in range(3):
      temparray = np.copy(a)
      temparray = np.roll(temparray, y - 1, axis=0)
      for x in range(3):
            temparray_X = np.copy(temparray)
            temparray_X = np.roll(temparray_X, x - 1, axis=1)*kernel[y,x]
            arraylist.append(temparray_X)

      arraylist = np.array(arraylist)
      arraylist_sum = np.sum(arraylist, axis=0)
      return arraylist_sum

def sobel_filter(image):
      filter = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
      new_image_x = convolution2d(image, filter, 1, 0)
      new_image_y = convolution2d(image, np.flip(filter.T, axis=0), 1, 0)
      gradient_magnitude = np.sqrt(np.square(new_image_x) +
      np.square(new_image_y))
      gradient_magnitude *= 255.0 / gradient_magnitude.max()
      return gradient_magnitude

def convolution2d(image, kernel, stride, padding):
      image = np.pad(image, [(padding, padding), (padding, padding)],
      mode='constant', constant_values=0)

      kernel_height, kernel_width = kernel.shape
      padded_height, padded_width = image.shape
```

```
        output_height = (padded_height - kernel_height) // stride + 1
        output_width = (padded_width - kernel_width) // stride + 1

        new_image = np.zeros((output_height, output_width)).astype(np.float32)

        for y in range(0, output_height):
        for x in range(0, output_width):
              new_image[y][x] = np.sum(image[y * stride:y * stride +
              kernel_height, x * stride:x * stride + kernel_width] *
              kernel).astype(np.float32)
        return new_image

def find_peak(image):
        row, col = len(image), len(image[0])
        peak_row = 0
        peak_col = 0
        peak_value = 0
        for i in range(row):
        for j in range(col):
              if image[i][j] > peak_value:
              peak_value = image[i][j]
              peak_row = i
              peak_col = j
        return peak_row, peak_col

def crop_image(image, hi_row, hi_col):
        A = np.zeros((101, 101), dtype=np.uint8)
        row, col = len(image), len(image[0])
        crop_row = hi_row - 50
        crop_col = hi_col - 50
        # Ensure the cropped image does not map outside of the image
        if crop_row < 0:
        crop_row = 0
        if crop_col < 0:
        crop_col = 0
        if (crop_row + 100) >= row:
        crop_row = row - 100
        if (crop_col + 100) >= col:
        crop_col = col - 100
        print('     crop row='+str(crop_row),'col='+str(crop_col))
        for i in range(101):
        tmp_col = crop_col
        for j in range(101):
              A[i][j] = image[crop_row][tmp_col]
              tmp_col = tmp_col + 1
        crop_row = crop_row + 1
        return A

def plot_histogram(x, title, yaxis, filename):
        plt.hist(x, density=True, bins=10, label="Data")
        plt.xlabel("Data")
        plt.title(title)
        if ( yaxis > 0 ):
        plt.ylim((0,yaxis))
        plt.grid(True)
        plt.savefig(filename)
```

```
        plt.show()
        return 1


if __name__ == '__main__':
        main()
```