National Institute of Technology Karnataka, Surathkal

Department of Computer Science and Engineering

CS350 - Cryptography and Applications

Analysis of Digital Signature, Arnold's Cat Map, & Advanced Hill Cipher Techniques for Image Encryption

Submitted to:

Mr. Alwyn Roshan Pais

Submitted by:

Abhishek Patil - 181CO102

Mohit Patil - 181CO133

Paranjaya Saxena - 181CO138

Table of Contents

1 Interpretation

1.1 Image Encryption Using Advanced Hill Cipher Algorithm

The Hill cipher algorithm is one of the symmetric key algorithms that have several advantages in data encryption. But, the inverse of the key matrix used for encrypting the plaintext does not always exist. Then if the key matrix is not invertible, encrypted text cannot be decrypted. The Advanced Hill cipher algorithm uses an Involutory key matrix for encryption. The inverse of the matrix exists when the involutory key matrix is used. Also, computational complexity can be reduced by avoiding finding the inverse of the matrix at the time of decryption as the inverse of an involutory matrix is equal to the same involutory matrix.

1.1.1 Hill Cipher

Hill cipher is a polygraphic substitution cipher based on linear algebra. Each letter is represented by a number modulo 26. Often the simple scheme a = 0, b = 1, …, z = 25 is used, but this is not an essential feature of the cipher. To encrypt a message, each block of n letters is multiplied by an invertible n × n matrix against modulus 26. To decrypt the message, each block is multiplied by the inverse of the matrix used for encryption.

$$\begin{pmatrix} C_1 \\ C_2 \\ C_3 \end{pmatrix} = \begin{bmatrix} K_{11} & K_{12} & K_{13} \\ K_{21} & K_{22} & K_{23} \\ K_{31} & K_{32} & K_{33} \end{bmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \end{pmatrix}$$

For encryption:

$$C = E_k(P) = K_p$$

For decryption:

$$P = D_k(C) = K^{-1}C = K^{-1}K_p = P$$

The inverse matrix K^{-1} of a matrix K is defined by the equation $KK^{-1} = K^{-1}K = I$, where I is the Identity matrix. But the inverse of the matrix does not always exist, and when it does, it satisfies the preceding equation. The matrix used for encryption is the cipher key, and it should be chosen randomly from the set of invertible n × n matrices (modulo 26). If the block length is m, there are 26^m different m letter blocks possible. Each of them can be regarded as a letter in a 26^m - letter alphabet.

1.1.2 Generation of Involutory Key Matrix

A is called an involutory matrix if $A = A^{-1}$. The generation of the involutory key matrix is valid for the matrix of +ve integers that are the residues of modulo arithmetic of a number. This algorithm can generate involutory matrices of order n × n where n is even.

Let $A = \begin{bmatrix} a_{11} & a_{12} & \dots & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & \dots & a_{2n} \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & \dots & a_{nn} \end{bmatrix}$ be an n × n involutory matrix partitioned to $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$

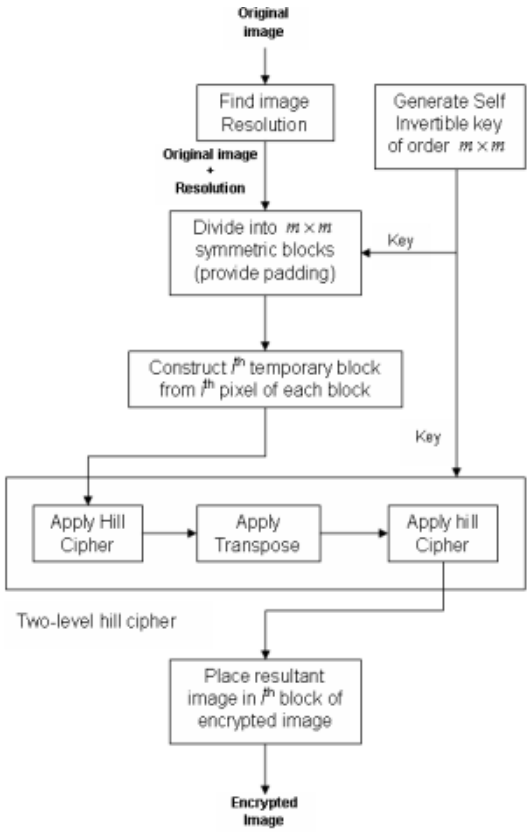
Where n is even and A_{11}, A_{12}, A_{21}, A_{22} are matrices of order $(n/2 \text{ x } n/2)$. So, $A_{12}A_{21} = I - A_{11}^2 = (I - A_{11})(I - A_{11})$. If A_{12} is one of the factors of $I - A_{11}^2$, then A_{21} is the other. Solving the 2nd matrix equation results $A_{11} + A_{22} = 0$. Then form the matrix.

Algorithm:

1. Select any arbitrary $(n/2 \text{ x } n/2)$ matrix A_{22}.
2. Obtain $A_{11} = - A_{22}$.
3. Take $A_{12} = k(I - A_{11})$ or $k(I - A_{11})$, where k is a scalar constant.
4. Then, $A_{21} = (1/k)(I - A_{11})$ or $(1/k)(I - A_{11})$.
5. Form the matrix entirely.

1.1.3 Image Encryption using Advance Hill Technique

The Advance Hill algorithm can also be used for grayscale and color images. For grayscale images, the modulus will be 256 (the number of levels is considered as the number of alphabets). In the case of color images, first, decompose the color image into (R-G-B) components. Second, encrypt each component (R-G-B) separately by the algorithm. Finally, concatenate the encrypted components together to get the encrypted color image.



Algorithm Advance Hill:

1. An involutory key matrix of dimensions m×m is constructed.
2. The plain image is divided into m×m symmetric blocks.
3. The ith pixels of each block are brought together to form a temporary block.
    a. Hill cipher technique is applied to the temporary block.
    b. The resultant matrix is transposed, and Hill cipher is again applied to this matrix.
4. The final matrix obtained is placed in the ith block of the encrypted image.

1.2 Image Encryption Using Arnold's Cat Map and Logistic Map for Secure Transmission

Data encoding provides the transformation of information in the coded form so that secure communication will be performed. But when there is the requirement to transfer some image data such as signatures, image password, biometric image, etc. A two-stage model is defined to achieve image encoding. In the first stage of this model, the chaotic map is defined. This map is able to preserve the effective image information based on frequency analysis. The mathematical modeling is applied to generate the chaotic map. Once the information is preserved, the encoding is performed using transformation operations. These operations include the phase transformation using Fast Fourier Transform.

1.2.1 Arnold's Cat Map

According to Arnold's transformation, an image is hit with the transformation that randomizes its pixels' original organization. However, if iterated enough times, eventually, the original image reappears. The number of considered iterations is known as Arnold's period. The period depends on the image size; i.e., for different size images, Arnold's period will be different.

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = A \begin{bmatrix} x_n \\ y_n \end{bmatrix} (mod\ N) = \begin{bmatrix} 1 & p \\ q & pq+1 \end{bmatrix} \begin{bmatrix} x_n \\ y_n \end{bmatrix} (mod\ N) \quad \text{.....(1)}$$

where N is the size of the image, p and q are positive integers, and det(A) = 1. (x_n, y_n) is the position of samples in the N × N data such as image, so that $\left(x_n, y_n\right) \in \{0, 1, 2, \ldots, N-1\}$

And (x_{n+1}, y_{n+1}) is the transformed position after the cat map. Cat map has two typical factors, which bring chaotic movement: tension (multiply matrix to enlarge x, y) and fold (taking mod to get x, y in-unit matrix).

Eq. (1) is used to transform each and every pixel coordinates of the image. When all the coordinates are transformed, the image resulted in a scrambled image. At a certain step of iterations, if the resulting image reaches our anticipated target (i.e., up to secret key), we have achieved the requested scrambled image. The image's decryption relies on the transformation periods (i.e., the number of iterations to be followed = Arnold's period − secret key).

1.2.2 Fast Fourier Transform

Fourier Transform is used to analyze the frequency characteristics of various filters. For images, 2D Discrete Fourier Transform (DFT) is used to find the frequency domain. A fast algorithm called Fast Fourier Transform (FFT) is used for the calculation of DFT. The image is transformed to signal form to process the encoding at an earlier stage. The expression form of this transformation is given as

$$F(k,l) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} f(i,j)\, e^{-\iota 2\pi (\frac{ki}{N} + \frac{lj}{N})}$$

Fourier Transform in Numpy

Numpy has an FFT package to do this. **np.fft.fft2()** provides the frequency transform, which will be a complex array. Its first argument is the input image, which is grayscale. The second argument is optional, which decides the size of the output array. If it is greater than the size of the input image, the input image is padded with zeros before calculation of FFT. If it is less than the input image, the input image will be cropped. If no arguments passed, the Output array size would be the same as the input.

Now after the result, the zero-frequency component (DC component) will be at the top left corner. To bring it to the center, shift the result by N/2 in both directions. This is simply done by the function, **np.fft.fftshift()**. (It is easier to analyze).

So after the frequency transform. Now reconstruct the image, i.e., find inverse DFT. For that, you simply remove the low frequencies by masking with a rectangular window of size 60x60. Then apply the inverse shift using **np.fft.ifftshift()** so that the DC component again comes at the top-left corner. Then find inverse FFT using **np.ifft2()** function. The result, again, will be a complex number. Take its absolute value.

1.3 A Technique for Image Encryption Using Digital Signature

The digital signature of an image is created by hashing the message. The digital signature of the original image is then added to the encoded version of the original image. The encoding of the image is done using error control code, such as a Bose–Chaudhuri Hochquenghem (BCH) code. At the receiver end, after the decryption

of the image, the digital signature can be used to verify the authenticity of the image. Detailed simulations have been carried out to test the encryption technique.

1.3.1 Digital Signature

Digital signatures enable the recipient of a message to authenticate the sender of a message and verify that the message is intact. The digital signature of an image is produced by using a one-way hash function. There are standard digital image algorithms that convert a message of any length into a fixed length message digest, usually 128 bits long. The standard techniques for creating a hash are MD2, MD4, MD5, and Secure Hash Algorithm (SHA). Hash functions are used because they are unique for a particular image and are very difficult to revert. A public-key encryption algorithm (such as RSA) is used in conjunction with the message digest prior to transmission.

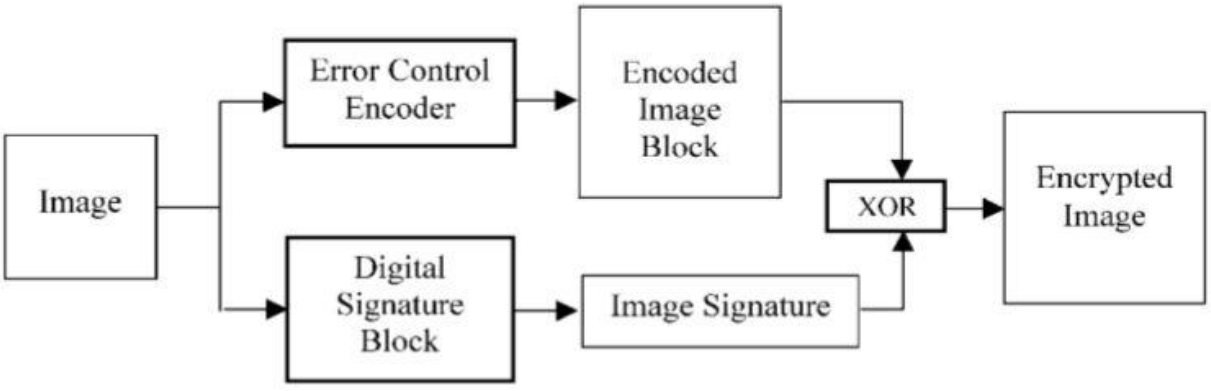
1.3.2 Bose–Chaudhuri Hochquenghem (BCH)

The Bose, Chaudhuri, and Hocquenghem (BCH) codes form a large class of powerful random error-correcting cyclic codes. This class of codes is a remarkable generalization of the Hamming code for multiple-error correction.

bchlib in Python

BCH code requires bitflip count i.e. total number of bitflips taking place and BCH polynomial. An error control code (ecc) is generated. This value of ecc is appended to data to be sent. While decrypting ecc removes all the errors in data and thus retrieves the original message.

1.3.3 Encryption Outline

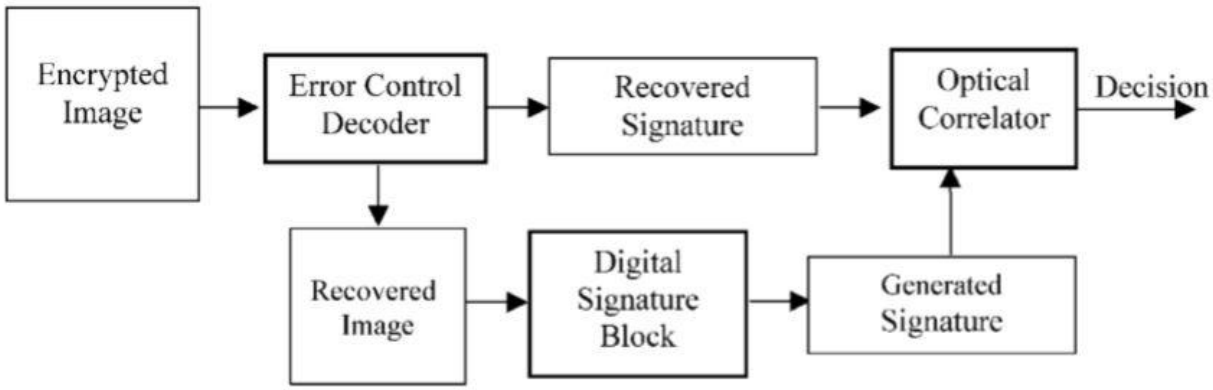


The digital signature is treated like additive noise, which can be recovered at the receiver end. To be able to recover the digital signature, an error control code is used to encode the original image. The digital signature is the noise that is added to the image after error control coding. The addition operation is equivalent to the XOR operation.

The image encryption system needs to handle all sizes of images. However, a digital signature generated by a standard algorithm produces a signature of a predefined length, usually 128 bits. Thus, the error control algorithm must be chosen depending upon the size of the input image which is done by BCH code.

The addition of the digital signature is carried out as follows. Both the encoded image and the digital signature are broken up into blocks and digital signatures are added blockwise. The size of the blocks is determined by the choice of the BCH code, which is a block code.

1.3.4 Decryption Outline



The received message is first passed through the error control decoder block. The task of this block is to recover the original image that has been corrupted (by design) by the digital signature. Once the original image is recovered, the digital signature is obtained by using the recovered image and the encrypted image. The recovered image is then used to generate its digital signature. The generated signature is then correlated with the recovered signature to reach a decision.

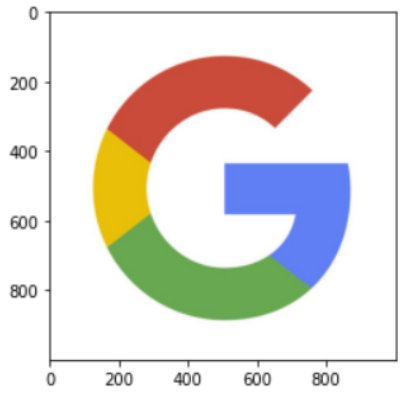
The authenticity of the transmitted image can be verified by correlating the recovered digital signature with the digital signature generated from the decrypted image. Since the digital signature is unique to an image, a correlation peak is only obtained when the decrypted image exactly coincides with the input image

2 Implementation

2.1 Image Encryption Using Advanced Hill Cipher Algorithm

Image is read using the imageio library. The image is made to have square dimensions. The image's length and width are reassigned to the smallest multiple of n greater than length and width, respectively. Imshow displays the image.

```
img = imageio.imread('0.png')
imshow(img)

nl = l = img.shape[0]
nw = w = img.shape[1]
n = 50
if l % n:
    nl = (int((l - 1) / n) + 1) * n
if w % n:
    nw = (int((w - 1) / n) + 1) * n
img2 = np.zeros((nl, nw, 3))
img2[:l, :w, :] += img
```



Encryption key involutory matrix is generated. "k" is taken as the encryption key. A random matrix is generated of the order (n/2 x n/2) named A_{22}. A_{11} = - $A_{22,}$ and similarly, A_{21} and A_{12} are generated according to the algorithm. Then final matrix A is formed by combining these four.

```
Mod = 256
k = 23  # Key for Encryption

A22 = np.random.randint(256, size=(int(n/2), int(n/2)))
I = np.identity(int(n/2))
```

```
A11 = np.mod(-A22, Mod)

A12 = np.mod((k * np.mod(I - A11, Mod)), Mod)
k = np.mod(np.power(k, 127), Mod)
A21 = np.mod((I + A11), Mod)
A21 = np.mod(A21 * k, Mod)

A1 = np.concatenate((A11, A12), axis=1)
A2 = np.concatenate((A21, A22), axis=1)
A = np.concatenate((A1, A2), axis=0)
```

Making sure that A is an involutory matrix, A*A = I

```
Test = np.mod(np.matmul(np.mod(A, Mod), np.mod(A, Mod)), Mod)
```

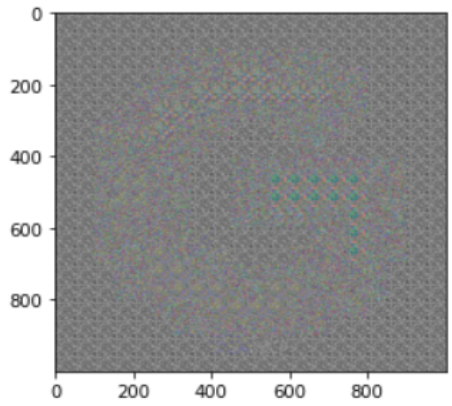
Saving key as an image. Adding dimensions of the original image within the key. Elements of the matrix should be below 256.

```
key = np.zeros((n + 1, n))
key[:n, :n] += A

key[-1][0] = int(l / Mod)
key[-1][1] = l % Mod
key[-1][2] = int(w / Mod)
key[-1][3] = w % Mod
```

Encryption is done on the image. Image is divided into three 2D arrays, one for each r, g, and b value. “A” (involutory matrix) is multiplied with the image. The transpose of the result is again multiplied with A. Then, all three results are combined to get the encrypted image. “Encrypted.png” stores the encrypted image.

```
Encrypted = np.zeros((nl, nw, 3))
for j in range(int(nw/n)):
    for i in range(int(nl/n)):
        Enc1 = (
            np.matmul(A % Mod, img2[i * n:(i + 1) * n, j * n:(j + 1) * n,
0] % Mod)) % Mod
        Enc1 = np.matmul(A % Mod, np.transpose(Enc1)) % Mod
        Enc2 = (
            np.matmul(A % Mod, img2[i * n:(i + 1) * n, j * n:(j + 1) * n,
```

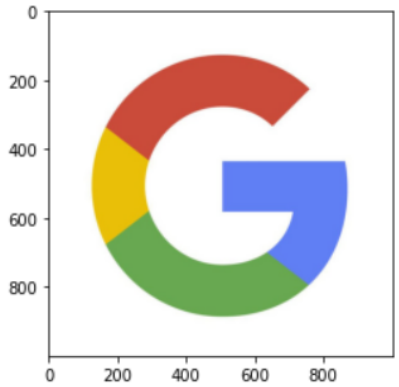
```
1] % Mod)) % Mod
        Enc2 = np.matmul(A % Mod, np.transpose(Enc2)) % Mod
        Enc3 = (
            np.matmul(A % Mod, img2[i * n:(i + 1) * n, j * n:(j + 1) * n,
2] % Mod)) % Mod
        Enc3 = np.matmul(A % Mod, np.transpose(Enc3)) % Mod

        Enc1 = np.resize(Enc1, (Enc1.shape[0], Enc1.shape[1], 1))
        Enc2 = np.resize(Enc2, (Enc2.shape[0], Enc2.shape[1], 1))
        Enc3 = np.resize(Enc3, (Enc3.shape[0], Enc3.shape[1], 1))
        Encrypted[i * n:(i + 1) * n, j * n:(j + 1) *
                  n] += np.concatenate((Enc1, Enc2, Enc3), axis=2)
imageio.imwrite('Encrypted.png', Encrypted)
imshow(imageio.imread('Encrypted.png'))
```



The decryption of the “Encrypted.png” is done here. We divide the image into three 2D arrays, one for each r, g, and b value. As we already know, $A = A^{-1}$, we multiply encrypted image by A, then take the resultant matrix transpose, and then multiply A again. The resulting matrix is then formed by combining the three 2D matrices. This is the decrypted image. This image is stored in “Decrpyted.png”.

```
Enc = imageio.imread('Encrypted.png')
nl = int(Enc.shape[0])
nw = int(Enc.shape[1])
# Loading the key
A = key
n = int(A.shape[0] - 1)
l = int(A[-1][0] * Mod + A[-1][1])  # The length of the original image
w = int(A[-1][2] * Mod + A[-1][3])  # The width of the original image
A = A[0:-1]
```

```
Decrypted = np.zeros((nl, nw, 3))
for j in range(int(nw/n)):
    for i in range(int(nl/n)):
        Dec1 = (np.matmul(A %
                          Mod, Enc[i * n:(i + 1) * n, j * n:(j + 1) * n, 0]
% Mod)) % Mod
        Dec1 = np.matmul(A % Mod, np.transpose(Dec1)) % Mod
        Dec2 = (np.matmul(A %
                          Mod, Enc[i * n:(i + 1) * n, j * n:(j + 1) * n, 1]
% Mod)) % Mod
        Dec2 = np.matmul(A % Mod, np.transpose(Dec2)) % Mod
        Dec3 = (np.matmul(A %
                          Mod, Enc[i * n:(i + 1) * n, j * n:(j + 1) * n, 2]
% Mod)) % Mod
        Dec3 = np.matmul(A % Mod, np.transpose(Dec3)) % Mod

        Dec1 = np.resize(Dec1, (Dec1.shape[0], Dec1.shape[1], 1))
        Dec2 = np.resize(Dec2, (Dec2.shape[0], Dec2.shape[1], 1))
        Dec3 = np.resize(Dec3, (Dec3.shape[0], Dec3.shape[1], 1))
        Dec = np.concatenate((Dec1, Dec2, Dec3), axis=2)
        Decrypted[i * n:(i + 1) * n, j * n:(j + 1) * n] += Dec

Final = Decrypted[:l, :w, :]  # Returning Dimensions to the real image

imageio.imwrite('Decrypted.png', Final)
imshow(imageio.imread('Decrypted.png'))
```



2.2 Image Encryption Using Arnold's Cat Map and Logistic Map for Secure Transmission

ArnoldCatTransform shuffles the pixels of the image with the help of the key. This function is used to encrypt the image. Here $(x,y)^{th}$ pixel of the image becomes ((x+y%n), (x+2y)%n) by iteration over every pixel of the image.

```
def ArnoldCatTransform(img):
    rows, cols, ch = img.shape
    n = rows
    img_arnold = np.zeros([rows, cols, ch])
    for x in range(0, rows):
        for y in range(0, cols):
            img_arnold[x][y] = img[(x+y)%n][(x+2*y)%n]
    return img_arnold
```

ArnoldCatTransform re-shuffles the pixels of the image with the help of the key. This function is used to decrypt the image. Here $(x,y)^{th}$ pixel of the image becomes ((2x-y)%n, (-x+y)%n) by iteration over every pixel of the image.

```
def ArnoldCatTransformD(img):
    rows, cols, ch = img.shape
    n = rows
    img_arnold = np.zeros([rows, cols, ch])
    for x in range(0, rows):
        for y in range(0, cols):
            img_arnold[x][y] = img[(2*x-y)%n][(-x+y)%n]
    return img_arnold
```

This function encrypts the image by calling the transform function. Image and the key is passed as arguments. "0_ArnoldCatEnc.png" stores the encrypted image.

```
def ArnoldCatEncryption(imageName, key):
    img = cv2.imread(imageName)
    for i in range (0,key):
        img = ArnoldCatTransform(img)
    cv2.imwrite(imageName.split('.')[0] + "_ArnoldCatEnc.png", img)
    return img
```

This function decrypts the image by calling the transformD function. Image and the key is passed as arguments. “img_ArnoldCatDec.png” stores the decrypted image.

```
def ArnoldCatDecryption(imageName, key):
    img = cv2.imread(imageName)
    rows, cols, ch = img.shape
    dimension = rows
    decrypt_it = dimension
    for i in range(0,key):
        img = ArnoldCatTransformD(img)
    cv2.imwrite(imageName.split('_')[0] + "_ArnoldCatDec.png",img)
    return img
```

Image and key is loaded in variables. key is set in a variable that will be used in encryption and decryption of the image.

```
image = "0"
ext = ".png"
key = 20
img = cv2.imread(image + ext,1)
pil_im = Image.open(image + ext, 'r')
imshow(np.asarray(pil_im))
```

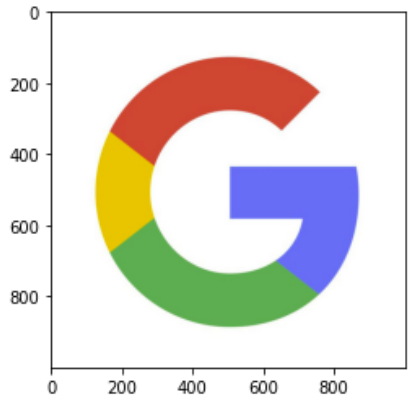
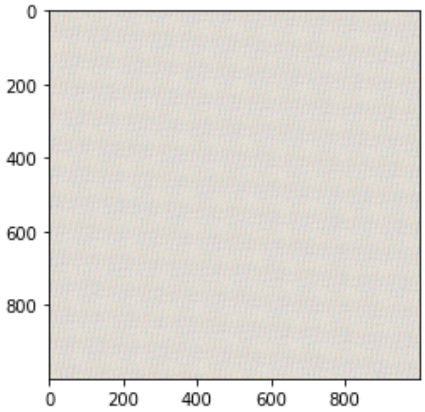


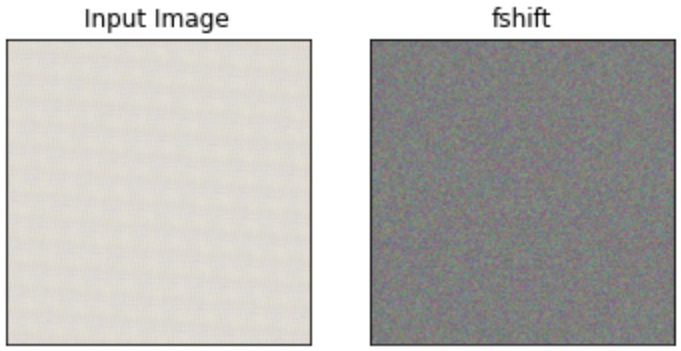
Image is encrypted by ArnoldCatEncryption with the help of a key. “0_ArnoldCatEnc.png” stores the encrypted image.

```
ArnoldCatEncryptionIm = ArnoldCatEncryption(image + ext, key)
img = Image.open("0_ArnoldCatEnc.png")
imshow(img)
```

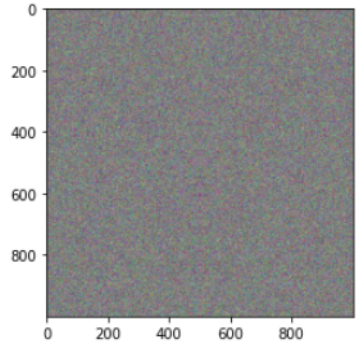


Fast Fourier transform is applied to the image. np.fft.fft2() provides the frequency transform, which will be a complex array. Its first argument is the input image. The second argument is optional, which decides the size of the output array. If it is greater than the size of the input image, the input image is padded with zeros before calculation of FFT. the zero-frequency component (DC component) will be at the top left corner. To bring it to the center, shift the result by N/2 in both directions by the function, np.fft.fftshift().

```
img = Image.open("0_ArnoldCatEnc.png")
f = np.fft.fft2(img)
fshift = np.fft.fftshift(f)

plt.subplot(121),plt.imshow(img)
plt.title('Input Image'), plt.xticks([]), plt.yticks([])
plt.subplot(122),plt.imshow((np.abs(fshift) * 255).astype(np.uint8))
plt.title('fshift'), plt.xticks([]), plt.yticks([])
plt.show()
```

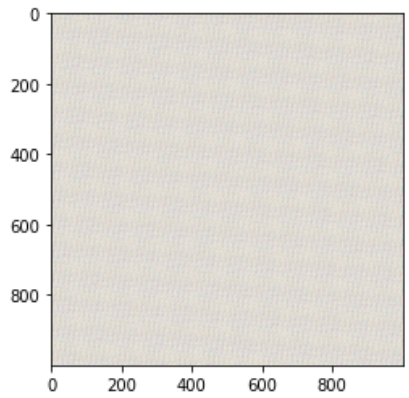


```
fft_encrypted = (np.abs(fshift)).astype(np.uint8)
cv2.imwrite("fft_encrypted" + ".png", fft_encrypted)
imshow(fft_encrypted)
```



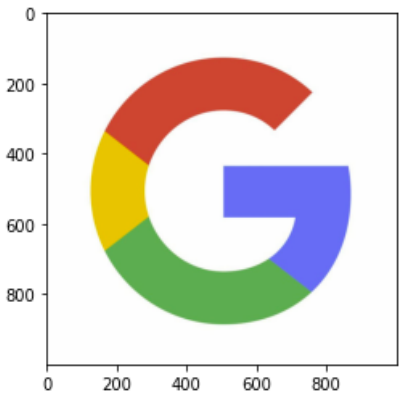
Now reconstruct the image, i.e., find inverse DFT. Apply the inverse shift using np.fft.ifftshift() so that the DC component again comes at the top-left corner. Then find inverse FFT using np.ifft2() function. The result, again, will be a complex number. Take its absolute value.

```
f_ishift = np.fft.ifftshift(fshift)
img_back = np.fft.ifft2(f_ishift)
img_back = np.abs(img_back)

# Convert into integer array
img_back=img_back.astype(int)

# 0th and 2nd rows are reversed somehow
# Therefore reverse them back
img_back[:,:,[0,2]]=img_back[:,:,[2,0]]

cv2.imwrite("img_back" + ".png",img_back)
imshow(img)
```



The image received by FFT is decrypted by the ArnoldCatDecryption function with the help of the key used for encryption. “img_ArnoldCatDec.png” stores the decrypted image.

```
ArnoldCatDecryptionIm = ArnoldCatDecryption("img_back.png", key)
img = Image.open("img_ArnoldCatDec.png")
imshow(img)
```



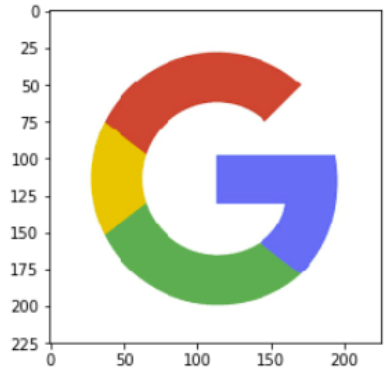
2.3 A Technique for Image Encryption Using Digital Signature

A digital signature uses public and private keys for encryption. In this snippet, we are creating RSA keys of sender and receiver.

```
from Crypto.PublicKey import RSA

#sender Keys
keyPair = RSA.generate(bits=1024)
print(f"Public key:  (n={hex(keyPair.n)}, e={hex(keyPair.e)})")
print(f"Private key: (n={hex(keyPair.n)}, d={hex(keyPair.d)})")

#Receiver Keys
keyPair2 = RSA.generate(bits=1024)
print(f"Public key2:  (n={hex(keyPair2.n)}, e={hex(keyPair2.e)})")
print(f"Private key2: (n={hex(keyPair2.n)}, d={hex(keyPair2.d)})")
```

Taking input image

```
img = imageio.imread('1.png')
imshow(img)
```



BCH code needs to specify total bitflips occurring in data, along with BCH polynomials is initiated here. Then the input array is converted into a byte array for encryption.

```
# create a bch object
BCH_POLYNOMIAL = 8219
BCH_BITS = 72
bch = bchlib.BCH(BCH_POLYNOMIAL, BCH_BITS)

#data input
a = asarray(img) #image data converted into a 3-D array
print(a.shape)
temp = bytearray(a)#3-d array converted into a byte array
l = len(temp)
final_corrupt = bytearray(0)
final_data = bytearray(0)
enc_sha = []
cor_sha = []
dec_sha = []

h=hashlib.md5(temp)
print (h.hexdigest())
```

Set the unique hash value of the image as corrupt.

```
#enter the MD5 hash value
input_corrupt = input("Enter corruption string: ")
```

Now, the encryption process has started. Divide the image array into blocks of array, then convert the corrupt array into its octane values. Create ecc for the block created

using bch.encode() function. This ecc value is appended to block. To see later that no data is lost or changed we take the hash of this block. After all this we apply XOR of Block and corrupt.

```
#image encryption starts
for n in range(0, 202500, 750):#loading a block of the bytearray
    data = bytearray(0)
    corrupt2 =''

    for i in range(n, n + 750):
        data.append(temp[i]);

    ###########Create Oct corrupt
    corrupt = bytearray(input_corrupt, 'UTF-8')
    for i in range(len(corrupt)):
        s = oct(corrupt[i])[2:]
        corrupt2 = corrupt2 +s

    corrupt = corrupt2
#     print(corrupt)

    ######### encode and make a "packet"
    ecc = bch.encode(data)
    print(len(ecc))
    packet = data + ecc

    # print hash of packet
    sha1_initial = hashlib.sha1(packet)
    print('sha1 original: %s' % (sha1_initial.hexdigest(),))
    enc_sha.append(sha1_initial.hexdigest())# saving the initial encoded
packet

    # make BCH_BITS errors
    for i in range(BCH_BITS):
        bytenum =i
        bitnum = int(corrupt[i])
        packet[bytenum] ^= (1 << bitnum)
#         print(packet[bytenum])

    #appending corrupted data of each block
    for i in range(len(packet)):
        final_corrupt=numpy.append(final_corrupt,packet[i])
       # print(len(packet))
```

```
    # print hash of packet
    sha1_corrupt = hashlib.sha1(packet)
    print('sha1 corrupted: %s' % (sha1_corrupt.hexdigest(),))
    cor_sha.append(sha1_corrupt.hexdigest())# saving the corrupted packet
into an array
```

Image hash is first encrypted using the sender's RSA private key. The method here is similar to that in RSA.

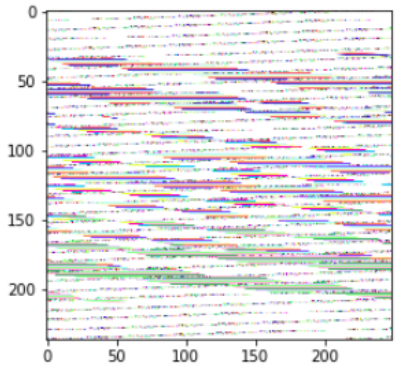
$hash^{d}(mod\ n)$

Then it is encrypted using the receiver's public key.

```
# RSA sign the message

hash= 0x354ca930b0a0070a73378cb71e298458
signature = pow(hash, keyPair.d, keyPair.n)
print("Signature:", hex(signature))

#signature pow public key of Sender
finalMessage = pow(signature, keyPair2.e, keyPair2.n)
print("Signature :",finalMessage)
```

Encrypted Image

```
pp= numpy.zeros(22, dtype= int)
final_corrupt2= numpy.append(final_corrupt, pp)

############ PRINT ENCODED IMAGE ###############
flatNumpyArray = numpy.array(final_corrupt2)
bgrImage2 = flatNumpyArray.reshape(236,248, 4)
cv2.imwrite('corrupted_image.png', bgrImage2)
imshow(Image.open('corrupted_image.png'))
```



For decryption, first, we use the reserved private key to decrypt. This step ensures that only a valid receiver can decrypt messages as the RSA private key of the receiver is only known to that particular receiver.

```
#decoding RSA signed message
recoveredSignature = pow(finalMessage, keyPair2.d, keyPair2.n)
print("recoveredSignature:", hex(recoveredSignature))
```

The encrypted image is converted into packets (packet2), these packets are then divided into data and ecc. Now using BCH we retrieve the original data and calculate total bitflips. We call this original data and ecc as ‘packet’. Then take SHA of each ‘packet’.
Now we retrieve the corrupt, which was added by doing XOR with data by doing XOR of ‘packet2’ and ‘packet’.

```
#image decryption starts
for n in range(0, 234090, 867):
    packet2 = bytearray(0)
    finalCorrect= ''
    for i in range(n, n + 867):
        packet2.append(final_corrupt[i]);# making a copy of the corrupted
bytearray

    # de-packetize
    data, ecc = packet2[:-bch.ecc_bytes], packet2[-bch.ecc_bytes:]

    # correct
    bitflips = bch.decode_inplace(data, ecc)
    print('bitflips: %d' % (bitflips)) # calculating the no. of bitflips
and printing it

    # packetize
```

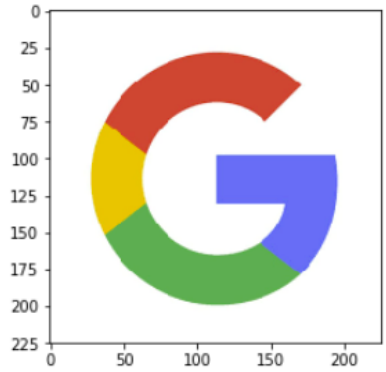
```
    packet = data + ecc
    for i in range(len(data)):
        final_data.append(data[i]) # appending each decoded data block into
a byte array

    # print hash of packet
    sha1_corrected = hashlib.sha1(packet)
    print('sha1: %s' % (sha1_corrected.hexdigest(),))
    dec_sha.append(sha1_corrected.hexdigest()) # copying the decoded data
block into an array

    print("getting back corrupt")

    for i in range(BCH_BITS):
        packet2[i]^= packet[i]
        finalCorrect= finalCorrect+str(packet2[i])
#         print(packet2[i])
```

Now by comparing the hash of the retrieved packet and the original packet. This shows BCH has retrieved the correct data.

```
############ check BCH recovered the code correctly #############

#Check if decrypted value after removing XOR is the same as enc val.
for i in range(0, 225):
    if enc_sha[i]==dec_sha[i]:
        print("Corrected!")
    else:
        print("Failed!")
```

Decrypted Image

```
#converting the byte array to a numpy array of a required dimension and
then using opencv to get the image
flatNumpyArray = numpy.array(final_data)
bgrImage = flatNumpyArray.reshape(225, 225,4)

imageio.imwrite('recovered_image.png', bgrImage)
imshow(imageio.imread('recovered_image.png'))
```



Hash of the retrieved image in octane. Compare this with corrupt which is a hash of the original image.

```
#Hashing retrieved image into Oct value
h2=hashlib.md5(temp)

# print (h2.hexdigest())
corrupt2 =''
retrieveCorrect= ''
corrupt = bytearray(h2.hexdigest(), 'UTF-8')
for i in range(len(corrupt)):
    s = oct(corrupt[i])[2:]
    corrupt2 = corrupt2 +s

corrupt = corrupt2

for i in range(len(corrupt)):
    p = int(corrupt[i])
    retrieveCorrect= retrieveCorrect+str(1<<p)

print(retrieveCorrect)

#Comparing Retrieved Image hash VS Hash recovered using ecc
if(retrieveCorrect==finalCorrect):
    print('true')
```

Take Hash of retrieved image, then using sender's public key decrypt recovered signature. Now we compare these values.

```
a = asarray(bgrImage) #image data converted into a 3-D array
temp = bytearray(a)#3-d array converted into a byte array
recoveredh=hashlib.md5(temp)
```

```
print (recoveredh.hexdigest())

# RSA verify signature
hash1= 0x354ca930b0a0070a73378cb71e298458 # This is recoveredh with 0x

#signature pow public key
hashFromSignature = pow(recoveredSignature, keyPair.e, keyPair.n)
print("Signature valid:", hash1 == hashFromSignature)
```

3 Comparison

3.1 Time

Time is an important factor to be considered for general cryptography. We can make a highly secure system but at the same time, it should be efficient. Because More time to encrypt and decrypt implies increased slowness of programs which in general usage of customers is a bad thing. Thus, the encryption algorithm is secure and requires less time for encryption. Decryption is considered better.

	Advanced hill	**Arnold's cat**	**Digital Signature**
encrypted time	time_gen_enc_key 0.03503 seconds time_enc_hill 0.39963 seconds	time_enc_arnold 1.56878 seconds time_enc_fft 0.19904 seconds	time_key_generation 0.56457 seconds time_enc_bch 0.31598 seconds
decrypted time	time_dec_hill 0.38742 seconds	time_dec_fft 0.27996 seconds time_dec_arnold 1.58020 seconds	time_dec_bch 0.38997 seconds time_compare_hash 0.04422 seconds
total	0.82208 seconds	3.62798 seconds	1.31474 seconds

3.2 Size

While considering image encryptions, we need to consider the size of the encrypted image which is sent over a medium. More the size of the encrypted implies increased slowness of programs which in general usage of customers is a bad thing. In general image encryption algorithms size of the encrypted image tends to increase thus the factor by which size increases matters most, Less the factor better the algorithm for most of the cases.

	Advanced hill	Arnold's cat	Digital Signature
original image	5.57 KB (5,706 bytes)	5.57 KB (5,706 bytes)	5.57 KB (5,706 bytes)
encrypted image	159 KB (1,63,733 bytes)	148 KB (1,52,429 bytes)	57.9 KB (59,346 bytes)

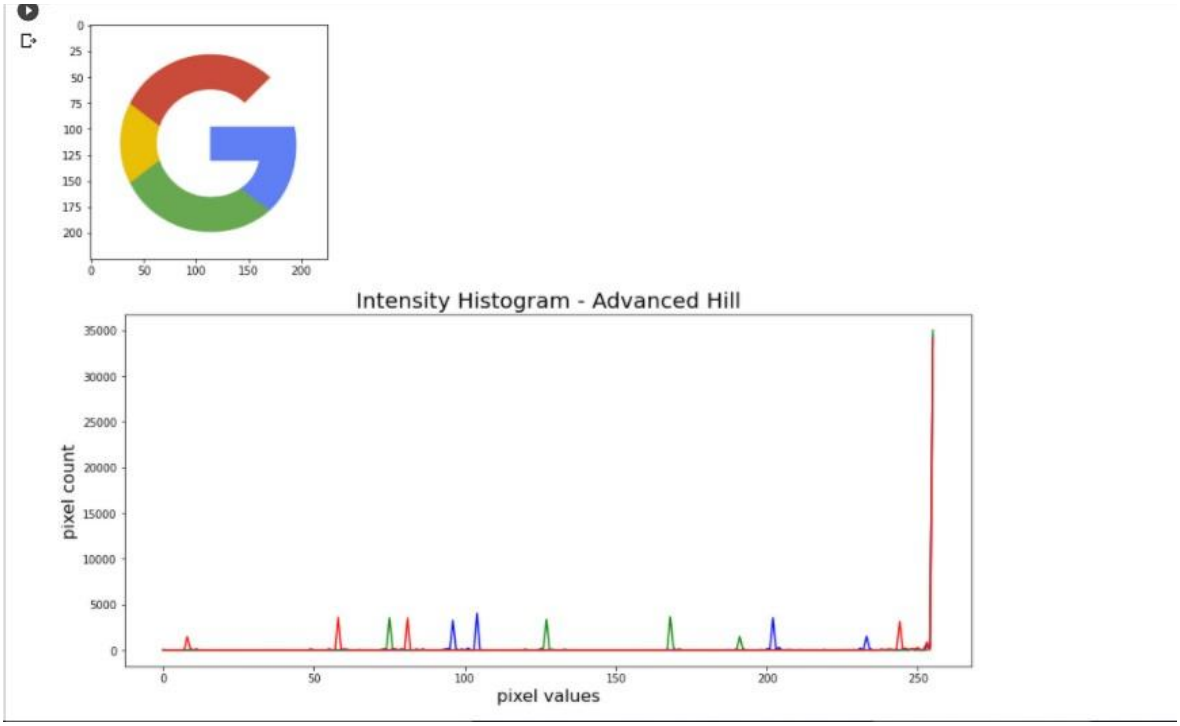
3.3 Color Science

Intensity histogram depicts the pixel values distribution in an image. It can be used to analyze the contents of an image. For eg. More blue color-related pixels may imply sky. Thus color science helps us to understand whether algorithms do retain some properties of the original image in the encrypted images or not, which can be exploited by intermediary.

3.3.1 Advanced hill

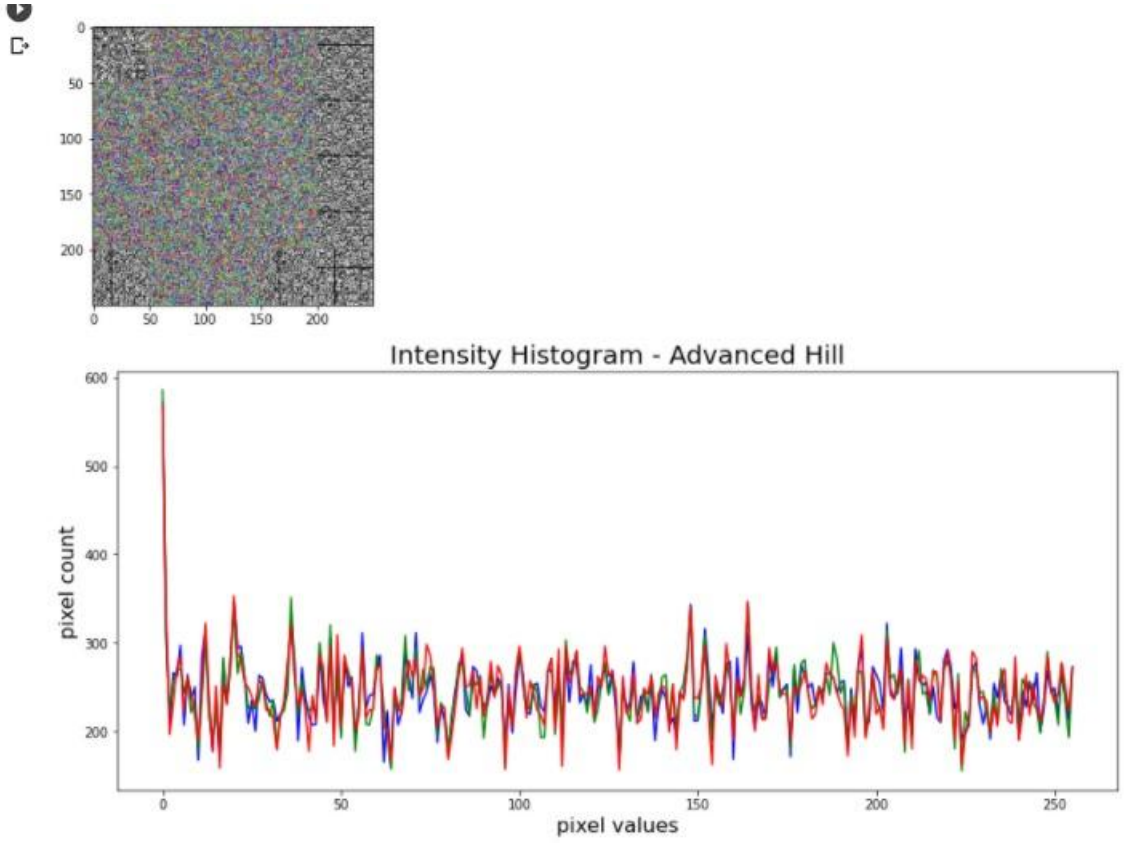
Intensity Histogram - Advanced Hill - Original Image

```
image = "0"
ext = ".png"
img = imageio.imread(image + ext)
pil_im =imageio.imread(image + ext)
imshow(np.asarray(pil_im))

plt.figure(figsize=(14,6))
histogram_blue = cv2.calcHist([img],[0],None,[256],[0,256])
plt.plot(histogram_blue, color='blue')
histogram_green = cv2.calcHist([img],[1],None,[256],[0,256])
plt.plot(histogram_green, color='green')
histogram_red = cv2.calcHist([img],[2],None,[256],[0,256])
plt.plot(histogram_red, color='red')
plt.title('Intensity Histogram - Advanced Hill', fontsize=20)
plt.xlabel('pixel values', fontsize=16)
plt.ylabel('pixel count', fontsize=16)
plt.show()
```



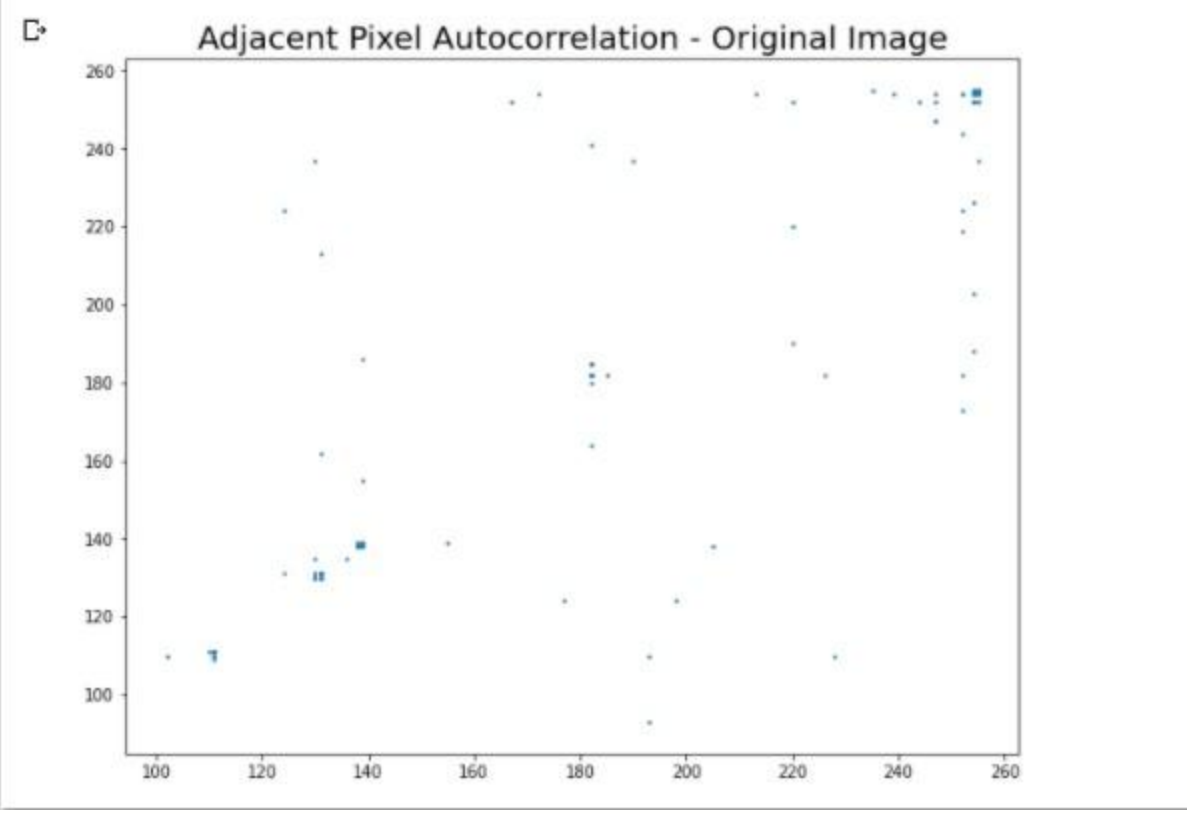
Intensity Histogram - Advanced Hill - Encrypted Image

```
image = "Encrypted"
ext = ".png"
img = cv2.imread(image + ext,1)
pil_im = Image.open(image + ext, 'r')
imshow(np.asarray(pil_im))

plt.figure(figsize=(14,6))
histogram_blue = cv2.calcHist([img],[0],None,[256],[0,256])
plt.plot(histogram_blue, color='blue')
histogram_green = cv2.calcHist([img],[1],None,[256],[0,256])
plt.plot(histogram_green, color='green')
histogram_red = cv2.calcHist([img],[2],None,[256],[0,256])
plt.plot(histogram_red, color='red')
plt.title('Intensity Histogram - Advanced Hill', fontsize=20)
plt.xlabel('pixel values', fontsize=16)
plt.ylabel('pixel count', fontsize=16)
plt.show()
```



Adjacent Pixel Autocorrelation - Advanced Hill - Original Image

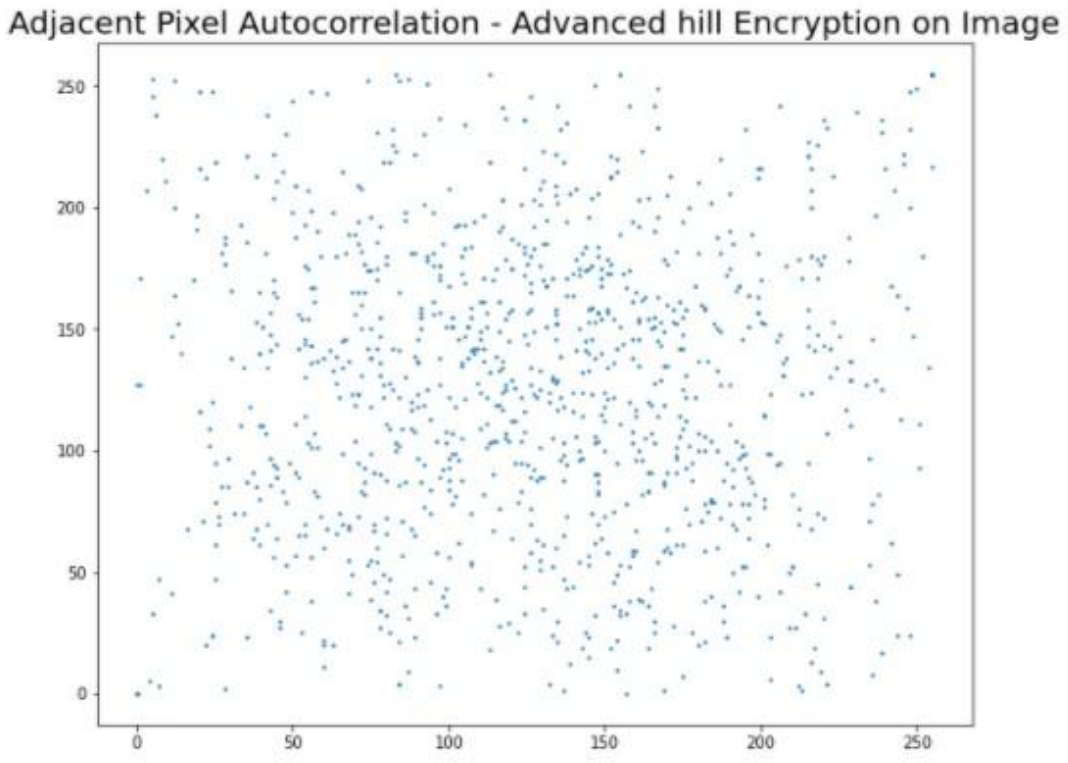
```
image = "0"
ext = ".png"
ImageMatrix,image_size = getImageMatrix_gray(image+ext)
samples_x = []
samples_y = []
for i in range(1024):
  x = random.randint(0,image_size-2)
  y = random.randint(0,image_size-1)
  samples_x.append(ImageMatrix[x][y])
  samples_y.append(ImageMatrix[x+1][y])
plt.figure(figsize=(10,8))
plt.scatter(samples_x,samples_y,s=2)
plt.title('Adjacent Pixel Autocorrelation - Original Image', fontsize=20)
plt.show()
```



Adjacent Pixel Autocorrelation - Advanced Hill - Encrypted Image

```
image = "Encrypted"
ext = ".png"
ImageMatrix,image_size = getImageMatrix_gray(image+ext)
samples_x = []
samples_y = []
print(image_size)
for i in range(1024):
  x = random.randint(0,image_size-2)
  y = random.randint(0,image_size-1)
  samples_x.append(ImageMatrix[x][y])
  samples_y.append(ImageMatrix[x+1][y])
plt.figure(figsize=(10,8))
plt.scatter(samples_x,samples_y,s=2)
plt.title('Adjacent Pixel Autocorrelation - Advanced hill Encryption on
Image', fontsize=20)
plt.show()
```

250

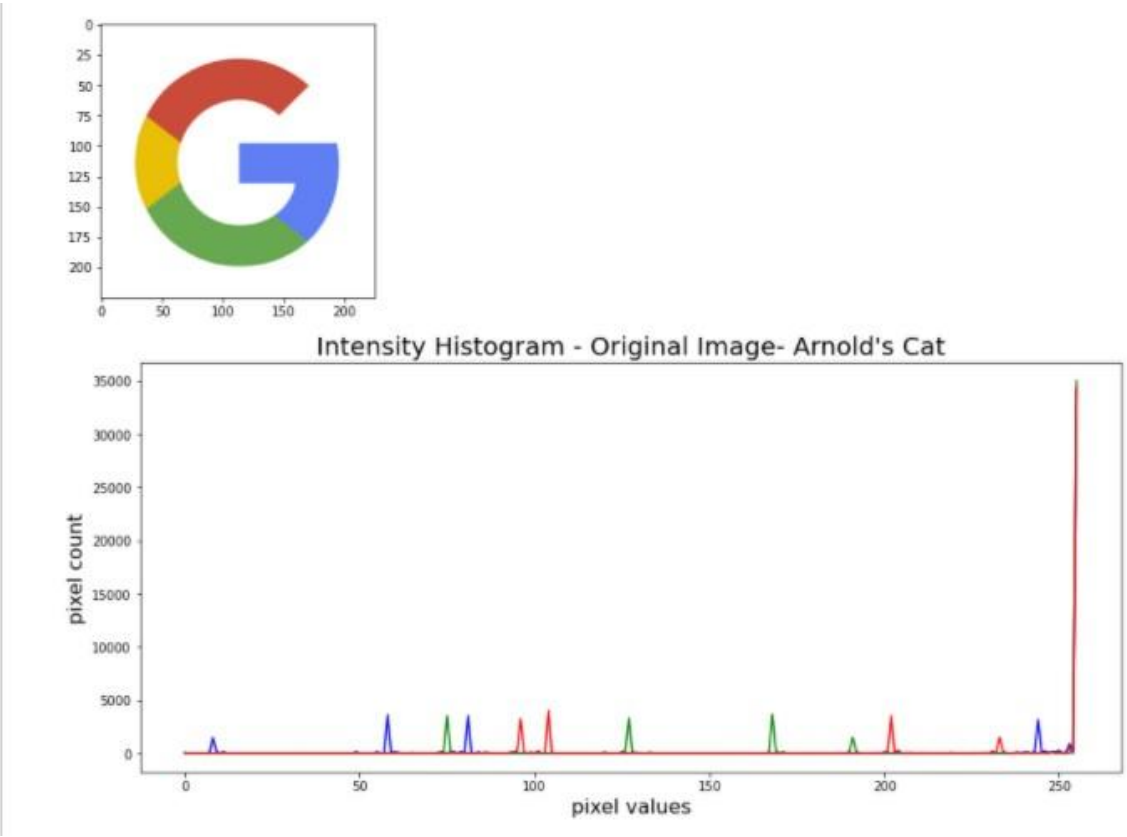


3.3.2 Arnold's cat

Intensity Histogram - Arnold's cat - Original Image

```
image = "0"
ext = ".png"
img = cv2.imread(image + ext,1)
pil_im = imageio.imread(image + ext)
imshow(np.asarray(pil_im))

plt.figure(figsize=(14,6))
histogram_blue = cv2.calcHist([img],[0],None,[256],[0,256])
plt.plot(histogram_blue, color='blue')
histogram_green = cv2.calcHist([img],[1],None,[256],[0,256])
plt.plot(histogram_green, color='green')
histogram_red = cv2.calcHist([img],[2],None,[256],[0,256])
plt.plot(histogram_red, color='red')
plt.title('Intensity Histogram - Original Image- Arnold\'s Cat',
fontsize=20)
plt.xlabel('pixel values', fontsize=16)
```

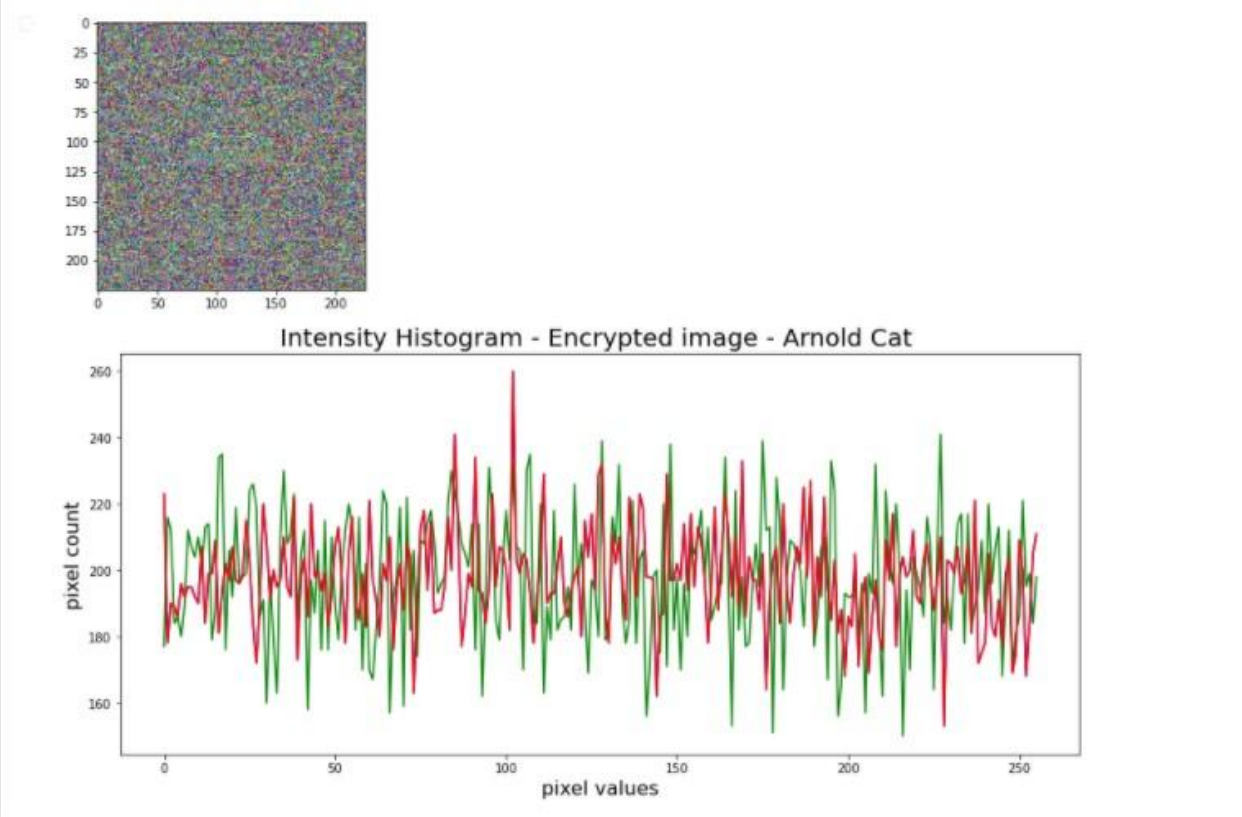
```
plt.ylabel('pixel count', fontsize=16)
plt.show()
```



Intensity Histogram - Arnold's cat - Encrypted Image

```
image = "fft_encrypted"
ext = ".png"
img = cv2.imread(image + ext,1)
pil_im = Image.open(image + ext, 'r')
imshow(np.asarray(pil_im))

plt.figure(figsize=(14,6))
histogram_blue = cv2.calcHist([img],[0],None,[256],[0,256])
plt.plot(histogram_blue, color='blue')
histogram_green = cv2.calcHist([img],[1],None,[256],[0,256])
plt.plot(histogram_green, color='green')
histogram_red = cv2.calcHist([img],[2],None,[256],[0,256])
plt.plot(histogram_red, color='red')
plt.title('Intensity Histogram - Encrypted image - Arnold Cat ',
fontsize=20)
```

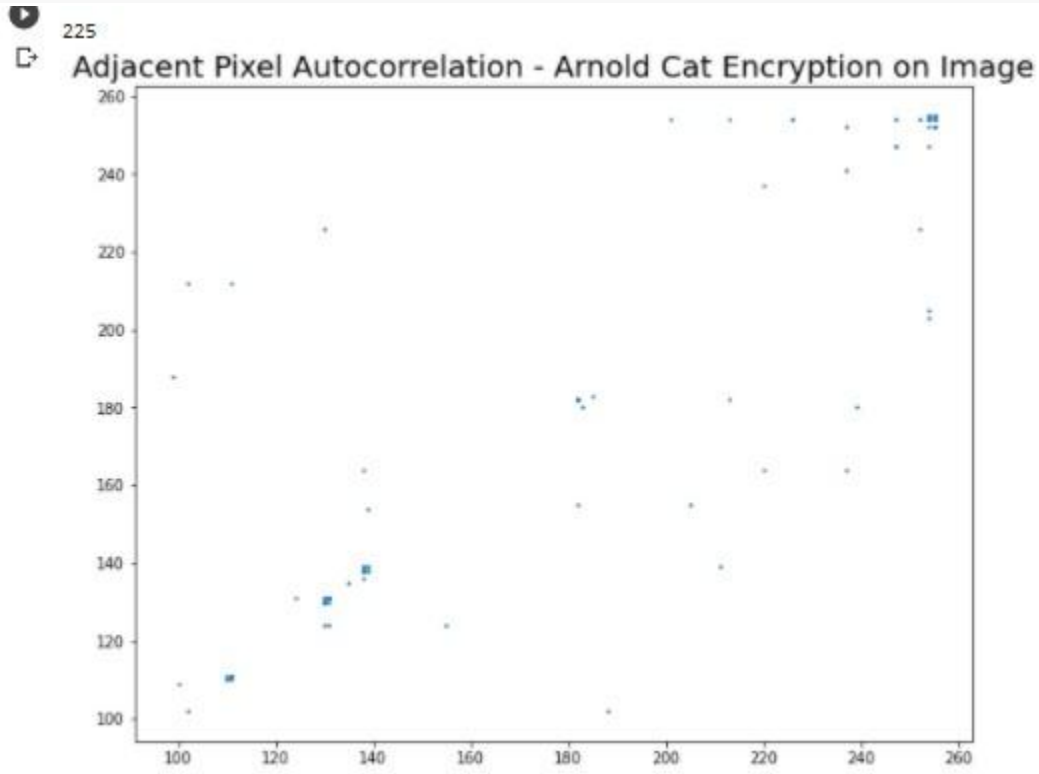
```
plt.xlabel('pixel values', fontsize=16)
plt.ylabel('pixel count', fontsize=16)
plt.show()
```



Adjacent Pixel Autocorrelation - Arnold's cat - Original Image

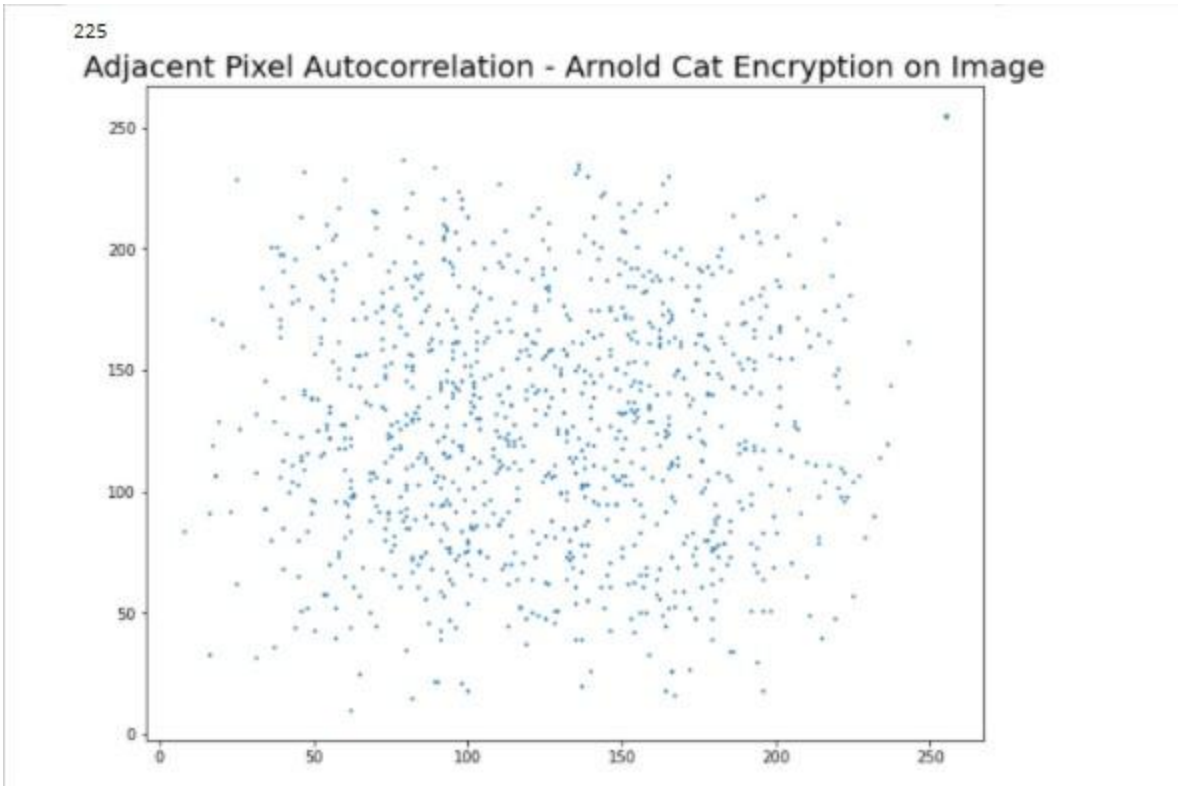
```
image = "0"
ext = ".png"
ImageMatrix,image_size = getImageMatrix_gray(image+ext)
samples_x = []
samples_y = []
print(image_size)
for i in range(1024):
  x = random.randint(0,image_size-2)
  y = random.randint(0,image_size-1)
  samples_x.append(ImageMatrix[x][y])
  samples_y.append(ImageMatrix[x+1][y])
plt.figure(figsize=(10,8))
plt.scatter(samples_x,samples_y,s=2)
plt.title('Adjacent Pixel Autocorrelation - Arnold Cat Encryption on
Image', fontsize=20)
```

```
plt.show()
```



Adjacent Pixel Autocorrelation - Arnold's cat - Encrypted Image

```
image = "fft_encrypted"
ext = ".png"
ImageMatrix,image_size = getImageMatrix_gray(image+ext)
samples_x = []
samples_y = []
print(image_size)
for i in range(1024):
  x = random.randint(0,image_size-2)
  y = random.randint(0,image_size-1)
  samples_x.append(ImageMatrix[x][y])
  samples_y.append(ImageMatrix[x+1][y])
plt.figure(figsize=(10,8))
plt.scatter(samples_x,samples_y,s=2)
plt.title('Adjacent Pixel Autocorrelation - Arnold Cat Encryption on
Image', fontsize=20)
plt.show()
```

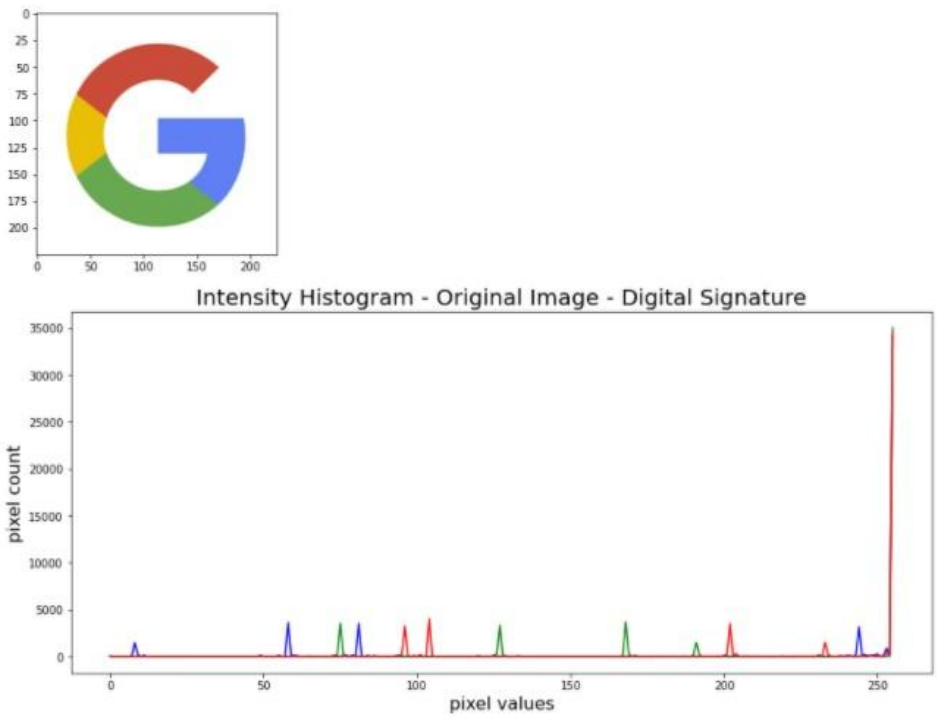


3.3.3 Digital Signature

Intensity Histogram - Digital Signature - Original Image

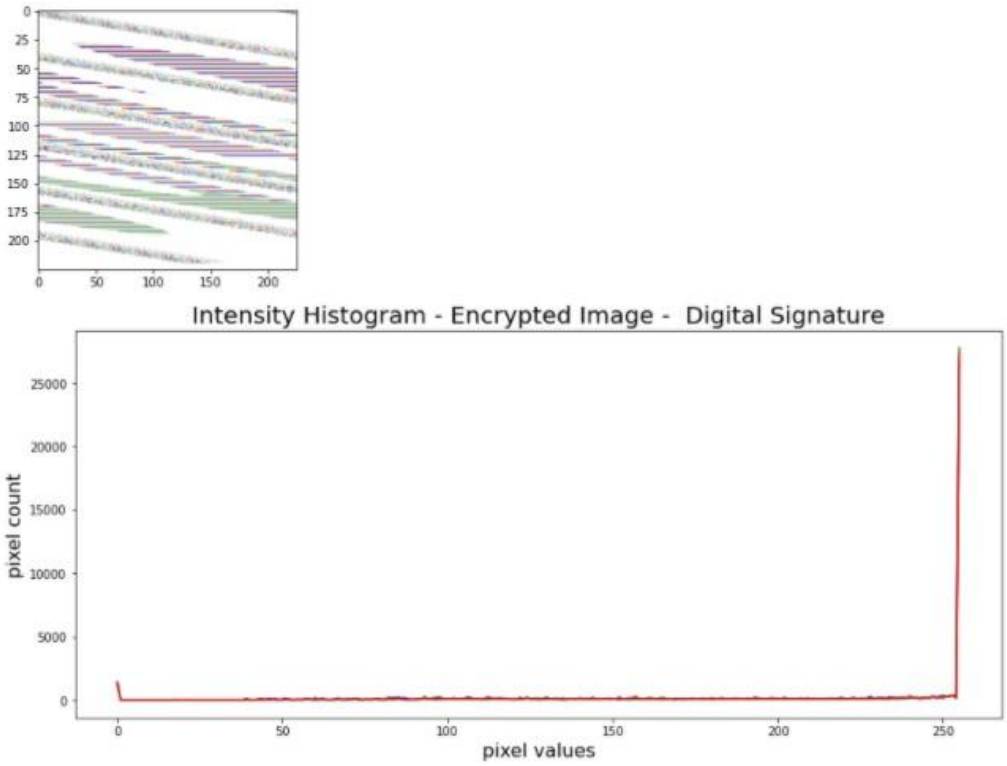
```
image = "0"
ext = ".png"
img = cv2.imread(image + ext,1)
pil_im = imageio.imread(image + ext)
imshow(np.asarray(pil_im))

plt.figure(figsize=(14,6))
histogram_blue = cv2.calcHist([img],[0],None,[256],[0,256])
plt.plot(histogram_blue, color='blue')
histogram_green = cv2.calcHist([img],[1],None,[256],[0,256])
plt.plot(histogram_green, color='green')
histogram_red = cv2.calcHist([img],[2],None,[256],[0,256])
plt.plot(histogram_red, color='red')
plt.title('Intensity Histogram - Original Image - Digital Signature',
fontsize=20)
plt.xlabel('pixel values', fontsize=16)
plt.ylabel('pixel count', fontsize=16)
```

```
plt.show()
```



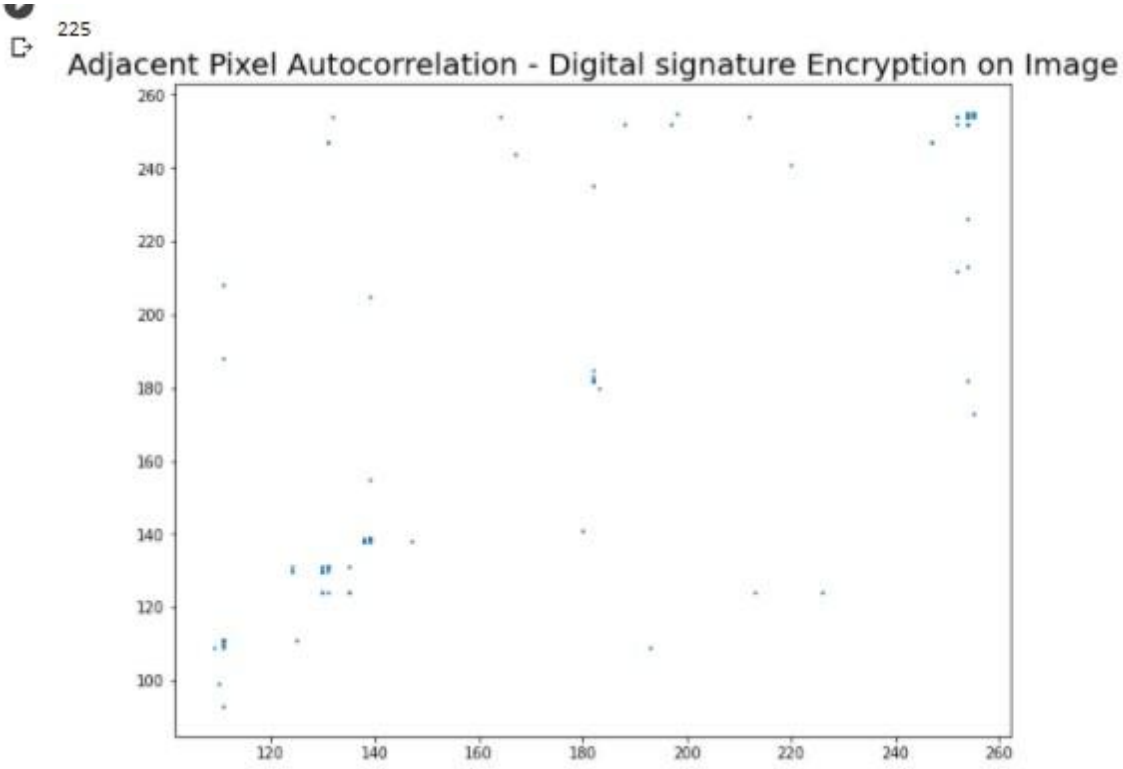
Intensity Histogram - Digital Signature - Encrypted Image

```
image = "corrupted_image"
ext = ".png"
img = cv2.imread(image + ext,1)
pil_im = imageio.imread(image + ext)
imshow(np.asarray(pil_im))

plt.figure(figsize=(14,6))
histogram_blue = cv2.calcHist([img],[0],None,[256],[0,256])
plt.plot(histogram_blue, color='blue')
histogram_green = cv2.calcHist([img],[1],None,[256],[0,256])
plt.plot(histogram_green, color='green')
histogram_red = cv2.calcHist([img],[2],None,[256],[0,256])
plt.plot(histogram_red, color='red')
plt.title('Intensity Histogram - Encrypted Image -  Digital Signature',
fontsize=20)
plt.xlabel('pixel values', fontsize=16)
plt.ylabel('pixel count', fontsize=16)
plt.show()
```



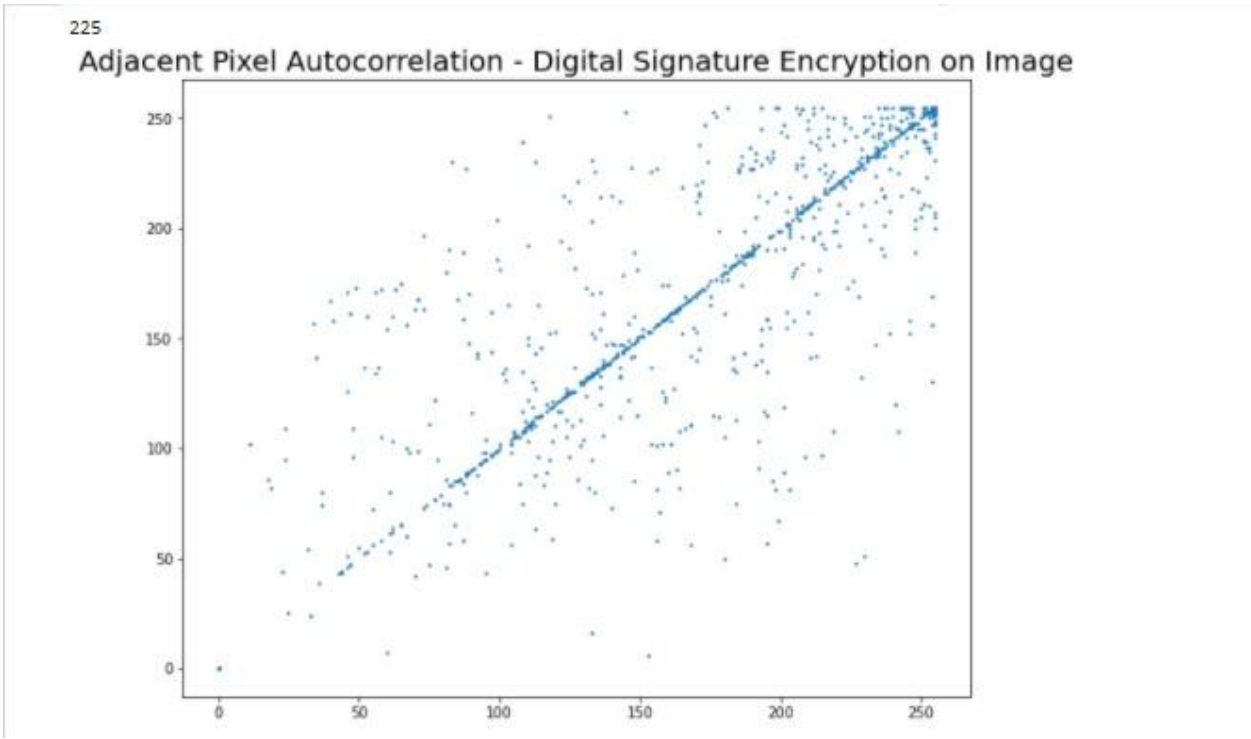
Adjacent Pixel Autocorrelation - Digital Signature - Original Image

```
image = "0"
ext = ".png"
ImageMatrix,image_size = getImageMatrix_gray(image+ext)
samples_x = []
samples_y = []
print(image_size)
for i in range(1024):
  x = random.randint(0,image_size-2)
  y = random.randint(0,image_size-1)
  samples_x.append(ImageMatrix[x][y])
  samples_y.append(ImageMatrix[x+1][y])
plt.figure(figsize=(10,8))
plt.scatter(samples_x,samples_y,s=2)
plt.title('Adjacent Pixel Autocorrelation - Digital signature Encryption on
Image', fontsize=20)
plt.show()
```



Adjacent Pixel Autocorrelation - Digital Signature - Encrypted Image

```
image = "corrupted_image"
ext = ".png"
ImageMatrix,image_size = getImageMatrix_gray(image+ext)
samples_x = []
samples_y = []
print(image_size)
for i in range(1024):
  x = random.randint(0,image_size-2)
  y = random.randint(0,image_size-1)
  samples_x.append(ImageMatrix[x][y])
  samples_y.append(ImageMatrix[x+1][y])
plt.figure(figsize=(10,8))
plt.scatter(samples_x,samples_y,s=2)
plt.title('Adjacent Pixel Autocorrelation - Digital Signature Encryption on
Image', fontsize=20)
plt.show()
```



4 References

4.1 Acharya, B., Panigrahy, S. K., Patra, S. K., and Panda, G., 2009, Image Encryption Using Advanced Hill Cipher Algorithm, International Journal of Recent Trends in Engineering and Technology, Vol. 1, No. 1, 243-247

4.2 P. Dureja, and B. Kochhar, Image Encryption Using Arnold's Cat Map and Logistic Map for Secure Transmission. International Journal of Computer Science and Mobile Computing, 4 (6), 194 – 199, (2015)

4.3 Sinha, A., Singh, K.: A Technique for Image Encryption using Digital Signature Optics Communications, 1–6 (2003)