Pattern Analysis of Sequences and Images Using Various Decomposition of Matrices Using Python Code

A Project Report submitted to the
Institute of Mathematics and Applications
in partial fulfillment of the requirements for the degree of

B. Sc.(Hons.)

in

Mathematics and Computing

by

Sonu Gupta (7473U186017)

Abhishek Mishra (7473U186001)

Under the supervision of

Asst. Prof. (Dr.) Sudhakar Sahoo

INSTITUTE OF MATHEMATICS AND APPLICATIONS

ANDHARUA, BHUBANESWAR - 751029, ODISHA, INDIA

Institute of Mathematics and Applications

Andharua, Bhubaneswar - 751029, Odisha, India

SUPERVISOR'S CERTIFICATE

This is to certify that the work presented in this project entitled **Pattern Analysis of Sequences and Images Using Various Decomposition of Matrices Using Python Code**, submitted by **Sonu Gupta** and **Abhishek Mishra** with roll numbers **7473U186017** and **7473U186001** respectively, Undergraduate Students of the **Institute of Mathematics and Applications** in partial fulfillment of the requirements for the degree of the **B. Sc.(Hons.) in Mathematics and Computing**. I hereby accord my approval of it as a study carried out and presented in a manner required for its acceptance for the Bachelor's Degree Certificate for which it has been submitted. The report has fulfilled all the requirements as per the regulations of the Institute and has reached the standard needed for submission. Neither this project nor any part of it has been submitted for any degree or diploma to any institute or university in India or abroad.

Supervisor's Name

Supervisor's Designation

Place:
Date:

Dedicated to

Asst. Prof. (Dr.) Sudhakar Sahoo

DECLARATION

We, **Sonu Gupta** and **Abhishek Mishra** with roll numbers **7473U186017** and **7473U186001** respectively hereby declare that this project entitled **Pattern Analysis of Sequences and Images Using Various Decomposition of Matrices Using Python Code** represents our original/review work carried out as a **B. Sc.(Hons.) in Mathematics and Computing** of IMA Bhubaneswar and, to the best of our knowledge, it is not a complete copy of previously published or written by another person, nor any material presented for award of any other degree or diploma of **Institute of Mathematics and Applications** or any other institution. Any contribution made to this research by others, with whom we have worked at IMA Bhubaneswar or elsewhere, is explicitly acknowledged in the dissertation. Works of other authors cited in this dissertation have beenduly acknowledged under the section Bibliography.

Sonu Gupta **7473U186017**
Abhishek Mishra **7473U186001**
IMA Bhubaneswar
Date: December 19, 2023

ACKNOWLEDGEMENT

At the end of my research project, it is a pleasant task and honour to express my sincere thanks to all those who contributed in many ways for making my project.

This research project could never have been possible without the great co-operation, encouragement and guidelines of many others. We take this opportunity to express our gratitude to the people who have been instrumental in the successful completion of this project.

We would like to express our sincere gratitude to our project guide **Asst. Prof. (Dr.) Sudhakar Sahoo**, **Institute of Mathematics and Applications** for his support, stimulating suggestions, and encouragement to harmonize my project especially in selecting and making this project.

We would like to gratefully acknowledge the enthusiastic guidance of, Amitav Saran, Dr. Sudhanshu Shekhar Rout, Asst. Prof. Indra Bate who were always present to answer our queries. We could not have completed the project without the help of our classmates and our friends who listened to us and guided us as well. Finally, we thank our respective Parents for being there through all thick and thins of our life and for having faith in all our decisions.

Sonu Gupta **7473U186017**
Abhishek Mishra **7473U186001**
IMA Bhubaneswar
Date: December 19, 2023

ABSTRACT

We will be studying different patterns by creating their images, decomposing and analysing them. The various decomposition used in the project are Singular Value Decomposition, L-U Decomposition, Q-R decomposition, Carry Value Transformation and Bitwise exclusive OR. The pattern formed by few sequences will also be studied using the SVD.

The sequences that are being studied in this project are:

- Fibonacci Sequence
- $\{2^n + 1\}$
- $\{2^n\}$

Also, we will be verifying the above mentioned decomposition using python codes, study the images formed and also visually inspect the decompositions.

We will see theorem related to CVT and XOR decomposition and define the theorem for matrices. Following which we will see the verification and application. Firstly, we will see the verification on matrix using python codes and then observe the same for an image.

Keywords: Singular Value Decomposition, L-U Decomposition, Q-R Decomposition, Carry Value Transformation, Bitwise Exclusive OR

Contents

Chapter 1

Introduction

1.1 Overview

Pattern recognition is the automated recognition of patterns and regularities in data. Pattern recognition has its origins in statistics and engineering. Some modern approaches to pattern recognition include the use of machine learning, due to the increased availability of big data and a new abundance of processing power. Pattern Analysis and Applications (PAA) also examines the use of advanced methods, including statistical techniques, neural networks, genetic algorithms, fuzzy pattern recognition, machine learning, and hardware implementations which are either relevant to the development of pattern analysis as a research area or detail novel pattern analysis applications [3].

We take few particular sequences and write them in a matrix form and then obtain the image of the matrix using python. We visualize the sequences using the image formed from the matrix. Further few decompositions of the matrix are done and hence the images are obtained. And then we visualize and analyse those images. We also verify various matrix decomposition techniques using the results we obtain.

1.2 Background, Preliminary, and Related Work

Carry Value Transformation (CVT) and XOR are the two different functions we are basically going to use in this project work. Referring to the work

of S. Pal, S.Sahoo and B.K. Nayak on CVT and XOR [7], we define the CVT and XOR as the bitwise AND and bitwise exclusive OR. We will be defining these functions in detail in upcoming pages and also use various matrix decomposition methods such as LU, QR and SVD, which we will see in detail in the further contents.

1.3 A Peek at the Contents

We will be defining all the different decompositions used, in the second chapter, explaining then in breif. The different python libraries and digital image basics will be then explained in chapter three. In the next two chapter following it will discuss and visualize CVT and XOR of matrices, which will lead us to discussion of the additive property of CVT and XOR and few theorems related to them. Discussion and verification of Q-R Decomposition, L-U Decomposition and Singular Value Decomposition in detail will be carried out in the further three chapters. The tenth chapter will mainly consist of study of the patterns of sequences in detail. At last we have an additional work as an application of additive property of CVT and XOR, which we discussed in the last chapter.

We have observed theorems regarding CVT and XOR of natural numbers and defined a similar theorem for matrices and proved them, following which we have a hypothesis regarding matrices based on the hypothesis of natural numbers, which might be proved in future and used for many purposes.

Chapter 2

Functions and Decompositions

In this section we will be looking at functions and decompositions we are using in the project. Namely, Carry value transformation, XOR, singular value decomposition and lower-upper decomposition.

2.1 Carry Value Transformation

Let $B_2 = \{0, 1\}$, also let a and b be decimal representation of binary number $(a_n, a_{n-1}, \ldots, a_1)_2$ and $(b_n, b_{n-1}, \ldots, b_1)_2$, where $a_i, b_i \in B_2$ for all $i = 1, 2, 3, \ldots, n$. The mapping $CVT : B_2 \times B_2 \to B_2 \times \{0\}$ defined by $CVT = (a_n \wedge b_n, a_{n-1} \wedge b_{n-1}, \ldots, a_1 \wedge b_1, 0)$.

Here, we convert the decimals in binary form and then operate the bitwise AND operator on each string of both the binary numbers. Then we put a 0 at the end of the string[Table 2.1].

For example let $a = (20)_{10} = (10100)_2$ and $b = (13)_{10} = (01101)_2$
$CVT = (1 \wedge 0, 0 \wedge 1, 1 \wedge 1, 0 \wedge 0, 0 \wedge 1, 0)_2$
$CVT = (0, 0, 1, 0, 0, 0)_2 = (8)_{10}$

a	1	0	1	0	0	-
b	0	1	1	0	1	-
CVT	0	0	1	0	0	0

Table 2.1: CVT Table

2.2 XOR - Bitwise Exclusive OR

Let $B_2 = \{0, 1\}$, also let a and b be decimal representation of binary number $(a_n, a_{n-1}, \ldots, a_1)_2$ and $(b_n, b_{n-1}, \ldots, b_1)_2$, where $a_i, b_i \in B_2$ for $i = 1, 2, 3, \ldots, n$. The mapping $XOR : B_2 \times B_2 \to B_2$ defined by $XOR = (a_n \oplus b_n, a_{n-1} \oplus b_{n-1}, \ldots, a_1 \oplus b_1)$.

Here, we convert the decimals in n-ary form and then operate the bitwise exclusive OR operator on each string of both the n-ary numbers. And then we convert the output of XOR into decimal form[Table 2.2].

For example let $a = (20)_{10} = (10100)_2$ and $b = (13)_{10} = (01101)_2$
$XOR = (1 \oplus 0, 0 \oplus 1, 1 \oplus 1, 0 \oplus 0, 0 \oplus 1,)_2$
$XOR = (1, 1, 0, 0, 1)_2 = (25)_{10}$

a	1	0	1	0	0
b	0	1	1	0	1
XOR	1	1	0	0	1

Table 2.2: XOR Table

2.3 Singular Value Decomposition

In linear algebra, the Singular Value Decomposition (SVD) of a matrix is a factorization of that matrix into three matrices i.e. $A = U\Sigma V^T$. A is a $m \times n$ matrix, U is a $m \times m$, V is a $n \times n$ and S is a $m \times n$ matrix. Here U and V^T are two unitary matrices, where is Σ is a diagonal matrix. U contains the information about the column space of A, V^T contains the information about the row space of A, and Σ contains information about how important the rows and columns of A are in presenting the data presented by A.

Although Σ is a $m \times n$ matrix, all the other elements in the matrix except the diagonals are zero. Also, if σ_i is the entry of Σ in the diagonal, then $\sigma_1 > \sigma_2 > \sigma_3 \ldots$, i.e. all the entries in Σ are hierarchically arranged. This also implies that the first column of U is more important than the second column of U and same goes with the row of V^T.

We create images of a sequence of numbers using the XOR or CVT functions. The image contains a lot of useless data which can be removed using

the singular value decomposition. So, we decompose the array of the image of the sequence such that we get only the important data of the image and ignore all the unnecessary data. A lot of space is saved using this technique. Since, SVD can be done only to 2-dimensional array, we convert the coloured image into grayscale.

2.4 L-U-P Decomposition

Any given square matrix can be written as the product of a lower triangular matrix and an upper triangular matrix. We use the Doolittle's LUP decomposition to write matrix A with partial pivoting to decompose A into PA = LU. Where P is a permutation matrix, which is used to resolve certain singularity issues. LUP-decomposition is used in certain quantitative finance algorithms.

We use the Dolittle's Algorithm to find the LU decomposition and find the upper triangular matrix using the [Eqn.2.1]

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} u_{kj} l_{ik} \tag{2.1}$$

We find the lower triangular matrix using the [Eqn.2.2]

$$l_{ij} = \frac{a_{ij} - \sum_{k=1}^{i-1} u_{kj} l_{ik}}{u_{jj}} \tag{2.2}$$

Also, we have $PA = LU$. As mentioned above P re-orders the rows of the given matrix A upon left-multiplication.

2.5 QR-Decomposition

Any given matrix can be written as the product of a unitary matrix(orthogonal matrix) and an upper triangular matrix. The QR method is a preferred iterative method to find all the eigenvalues of a matrix.

The RQ decomposition transforms a matrix A into the product of an upper triangular matrix R and an orthogonal matrix Q. The only difference from QR decomposition is the order of these matrices.

- QR decomposition is Gram–Schmidt orthogonalization of columns of A, started from the first column.
- RQ decomposition is Gram–Schmidt orthogonalization of rows of A, started from the last row.

The Gram-Schmidth method is very easy to implement. Hence, it is most widely used. Whereas we use the method of Householder Reflections to find the QR decomposition.

2.6 Splitting the Image

Every coloured image had three colour components RGB, standing for red, blue and green respectively. We can separate the colours from the image in Python, and find the relation between CVT and XOR of the separated images with that of the original image. This technique can be further modified and worked upon to obtain new ways of encrypting data in an image, leading to increased security. A very brief of this topic is studied in this project.

Chapter 3

Python Basics

3.1 Python Libraries

We are going to look into some of the functions of Python we used while writing the code.

- ***NumPy***: - NumPy is a library in Python which adds support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on arrays.
- ***PIL***: - Python Imaging Library is a free and open-source additional library for the Python programming language that adds support for opening, manipulating, and saving many different image file formats.
- ***matplotlib***: - matplotlib is a visualization library in Python for 2D plots of arrays. It is based on NumPy array. It allows us visual access to huge amounts of data in easily digestible visuals.
- ***OpenCV***: - OpenCV is a library of Python bindings designed to solve computer vision problems.
- ***SciPy***: - A free and open-source Python library used for scientific computing and technical computing. It contains modules for linear algebra and image processing, which are used in this project.
- ***Skimage***: - An open-source Python library mainly used for image processing in python.

3.2 Digital Image Basics

The image is a rectangular tiling of fundamental elements called pixels. A pixel is a small block that represents the amount of grey intensity to be displayed for that particular portion of the image. For most images, pixel values are integers that range from 0 (black) to 255 (white). The 256 possible grey intensity values are shown below.

Figure 3.1: Digital Image Basics

Color images are made up of pixels each of which holds three numbers corresponding to the red, green, and blue (RGB) levels of the image at a particular location. Assuming 256 levels for each RGB primary level, each pixel can be stored in three bytes (24 bits) of memory. This corresponds to roughly 16.7 million different possible colors. [1]

$2^8$of Red $\times$ 2^8 of Green $\times$ 2^8 of Blue = $2^{24} = 16,777,216$

Chapter 4

Carry Value Transformation

4.1 CVT of a matrix

Here, we show the program to find the CVT of a matrix and then show the image of CVT of a matrix [Figure 4.4]. [8]

In the following source code we define several functions for different purposes.

- **binary(x)**: Convert a decimal number to binary.
- **decimal(x)**: Convert a binary number to decimal.
- **convert_cvt(x, y)**: Find the carry value for a pair of numbers. Since, we find the carry value by operating the bitwise AND on the binary of a pair of numbers, we operate AND in the while loop. We assign $c = 0$ as to get the carry value we need a 0 at the end of the output we get after operating AND.

Input

```
1 import numpy as np
2 from PIL import Image
3
4 def binary(x):
5     return int(bin(x)[2:])
6
7 def decimal(x):
8     return int(str(x), 2)
9
```

```
10 def convert_cvt(x, y):
11     a = binary(x)
12     b = binary(y)
13     c = "0"
14     while a > 0 and b > 0:
15         if a % 2 == 1 and b % 2 == 1:
16             c = "1" + c
17         else:
18             c = "0" + c
19         a = a // 10
20         b = b // 10
21     return decimal(int(c))
22 
23 n = int(input("Total number of rows: "))
24 m = int(input("Total number of columns: "))
25 # First random matrix A
26 A = np.random.randint(1000, size=(n, m))
27 print("matrix_'A'")
28 print(A)
29 # Seconnd random matrix B
30 B = np.random.randint(1000, size=(n, m))
31 print("matrix_'B'")
32 print(B)
33 matrix_CVT = np.random.randint(1, size=(n, m))
34 
35 for i in range(n):
36     for j in range(m):
37         matrix_CVT[i][j] = convert_cvt(A[i][j],  B[i][j])
38 print("matrix-'CVT'")
39 print(matrix_CVT)
40 img = Image.fromarray(matrix_CVT)
41 img.show("matrix_CVT.png")
```

Listing 4.1: CVT Matrix

Output

```
Total number of rows: 1000
Total number of columns: 1000
matrix_'A'
[[537 536 644 ... 199 618 411]
 [736 993 823 ... 343 200 281]
 [975 953 589 ... 400 888 555]
 ...
 [894 315 454 ... 181  70  62]
 [254 854 428 ... 430 283 420]
 [379 646 343 ... 216 641 176]]
```

Figure 4.1: Matrix_A

```
matrix_'B'
[[988 153 402 ... 519 202 828]
 [632 390  28 ... 575 293 977]
 [232  36 480 ... 365 879 949]
 ...
 [201 631 553 ... 169 845 970]
 [562 400 115 ... 134 658 860]
 [947 671 567 ... 638 430 813]]
```

Figure 4.2: Matrix_B

```
matrix_CVT
[[1104  520 1042 ...   26  232  560]
 [ 384   26  832 ...  806  780  262]
 [  16   48 1040 ... 1062  580 1154]
 ...
 [ 288 1560  520 ... 1062   72   56]
 [  64  530  154 ...  256  352 1664]
 [   6  268  512 ...    4  640  642]]
```

Figure 4.3: Matrix_CVT

The generated pattern is

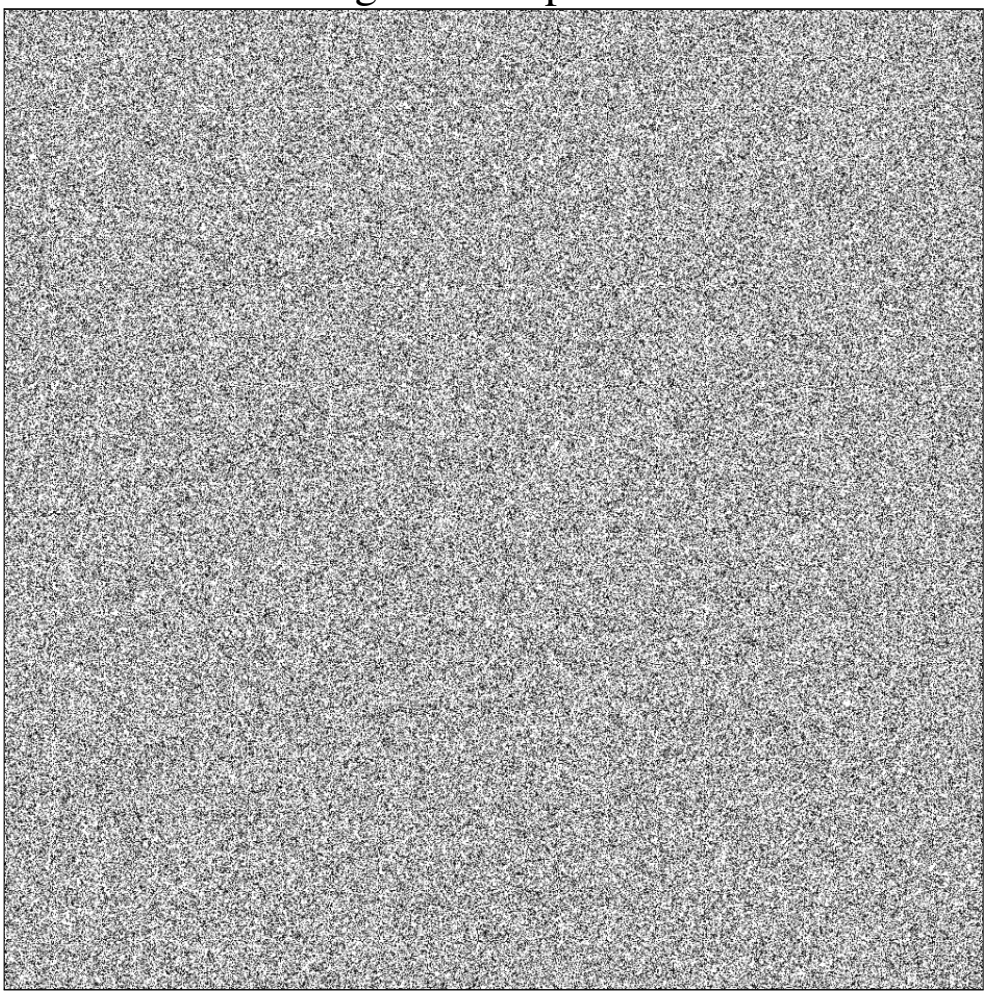


Figure 4.4: Image formed by matrix_CVT

Chapter 5

Bitwise Exclusive- OR

5.1 XOR of a matrix

In this section, the program to find the XOR of a matrix is shown and then the image of XOR of a matrix [Figure 5.4].

In the following source code we define several functions for different purposes.

- **binary(x)**: Convert a decimal number to binary.
- **decimal(x)**: Convert a binary number to decimal.
- **convert_xor(x, y)**: Find the XOR value for a pair of numbers. Since, we find the XOR value by operating the bitwise exclusive OR on the binary of a pair of numbers, we operate XOR in the while loop. We assign $c = 0$ as to get the carry value we need a 0 at the end of the output we get after operating XOR.

Input

```
1 import numpy as np
2 from PIL import Image
3
4 def binary(x):
5     return int(bin(x)[2:])
6
7 def decimal(x):
8     return int(str(x), 2)
9
```

```
10 def convert_xor(x, y):
11     a = binary(x)
12     b = binary(y)
13     c = "0"
14     while a > 0 or b > 0:
15         if (a%2==1 and b%2==1) or (a%2==0 and b%2==0):
16             c = "0" + c
17         else:
18             c = "1" + c
19         a = a // 10
20         b = b // 10
21     c = int(c) // 10
22     return decimal(c)
23 
24 n = int(input("Total number of rows: "))
25 m = int(input("Total number of columns: "))
26 A = np.random.randint(1000, size=(n, m))
27 print("matrix_'A'")
28 print(A)
29 # Second random matrix B
30 B = np.random.randint(1000, size=(n, m))
31 print("matrix_'B'")
32 print(B)
33 matrix_XOR = np.random.randint(1, size=(n, m))
34 
35 for i in range(n):
36     for j in range(m):
37         matrix_XOR[i][j] = convert_xor(A[i][j], B[i][j])
38 print("matrix_'XOR'")
39 print(matrix_XOR)
40 img = Image.fromarray(matrix_XOR)
41 img.show("matrix_XOR.png")
```

Listing 5.1: XOR Matrix

Output

```
Total number of rows: 1000
Total number of columns: 1000
matrix_'A'
[[537 536 644 ... 199 618 411]
 [736 993 823 ... 343 200 281]
 [975 953 589 ... 400 888 555]
 ...
 [894 315 454 ... 181  70  62]
 [254 854 428 ... 430 283 420]
 [379 646 343 ... 216 641 176]]
```

Figure 5.1: matrix_A

```
matrix_'B'
[[988 153 402 ... 519 202 828]
 [632 390  28 ... 575 293 977]
 [232  36 480 ... 365 879 949]
 ...
 [201 631 553 ... 169 845 970]
 [562 400 115 ... 134 658 860]
 [947 671 567 ... 638 430 813]]
```

Figure 5.2: matrix_B

```
matrix_'XOR'
[[ 453  641  790 ...  704  672  679]
 [ 152  615  811 ...  872  493  712]
 [ 807  925  941 ...  253   23  414]
 ...
 [ 951  844 1007 ...   28  779 1012]
 [ 716  710  479 ...  296  905  760]
 [ 712   25  864 ...  678  815  925]]
```

Figure 5.3: matrix_XOR

The generated pattern is

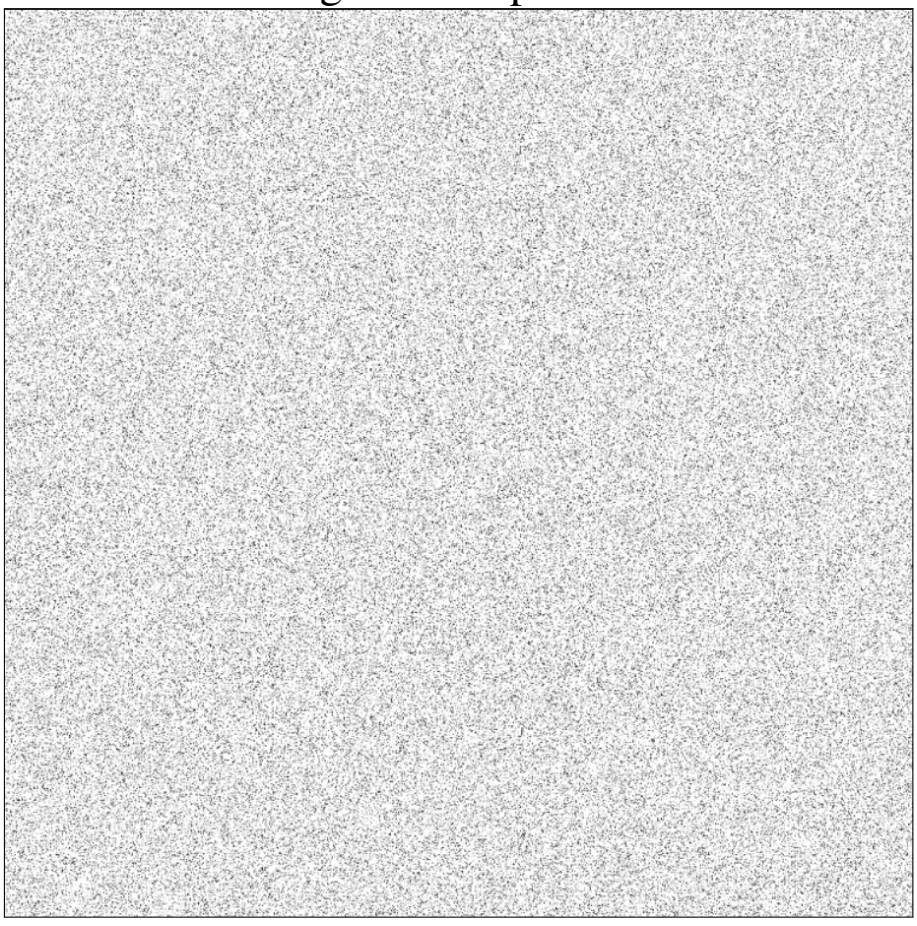


Figure 5.4: Image formed by matrix_XOR

Chapter 6

Additive Property of CVT and XOR for Matrices

6.1 Some Important Theorems

Theorem 1. *Given 2 positive integers a and b,* $a + b = CVT(a, b) + XOR(a, b)$. *[7]*

Theorem 2. *Given 2 matrices A and B with non-negative entries, we have* $B + A = CVT(A, B) + XOR(A, B)$

Proof. Define $CVT(A, B) = [c_{ij}]$, where $c_{ij} = CVT(a_{ij}, b_{ij})$.

Define $XOR(A, B) = [x_{ij}]$, where $x_{ij} = XOR(a_{ij}, b_{ij})$.

On operating $CVT \; \forall \; a_{ij} \in A$ and $b_{ij} \in B$, we get $c_{ij} \in CVT(A, B)$

On operating $XOR \; \forall \; a_{ij} \in A$ and $b_{ij} \in B$, we get $x_{ij} \in XOR(A, B)$

Let $M = CVT(A, B) + XOR(A, B)$.

Now, we have $m_{ij} = c_{ij} + x_{ij}$

$m_{ij} = CVT(a_{ij}, b_{ij}) + XOR(a_{ij}, b_{ij})$

Using previous theorem we get, $m_{ij} = a_{ij} + b_{ij}$

$M = A + B$

$CVT(A, B) + XOR(A, B) = A + B$ □

Lemma 1. *If* a *and* b *are of maximum* n *binary bits, then the number of iterations required to get* $CVT(a, b) = 0$ *is atmost* $(n + 1)$.

Theorem 3. *Let $f : Z \times Z \to Z \times Z$ be defined as $f(a, b) = CVT(a, b) + XOR(a, b)$. Then the iterative scheme $(x_{n+1}, y_{n+1}) = f(x_n, y_n)$ for $n = 1, 2, 3, \ldots$ converges to $(0, x_0+y_0)$ for any initial choice of $(x_0, y_0) \in Z \times Z$. Further, for any non-negative integers x_0 and y_0 where $x_0 \geq y_0$, the number of iterations required to get either $CVT = 0$ or $XOR = 0$ is atmost the length of x_0 when expressed as binary string [7].*

Hypothesis 1. *For given two matrices A and B with entries $a_{ij} \in Z$ and $b_{ij} \in Z$ respectively. We have defined $CVT(A, B)$ and $XOR(A, B)$ in theorem 2. Now, let $f : Z \times Z \to Z \times Z$ be defined as $f(A, B) = CVT(A, B)+XOR(A, B)$. Then the iterative scheme (CVT_{n+1}, XOR_{n+1}) = $f(CVT_n, XOR_n)$ for $n = 0, 1, 2, 3, \ldots$ converges to $(0, CVT_0 + XOR_0)$ where (CVT_0, XOR_0) is $(A, B) \in Z \times Z$. Further, for any non-negative integers a_{ij} and b_{ij} where a_{ij} is the greatest element of A and b_{ij} is the greatest element of B, the number of iterations required to get either $CVT = 0$ or $XOR = 0$ is atmost the length of a_{ij} when expressed as binary string if $a_{ij} \geq b_{ij}$ [7].*

6.2 Visualization of the theorem for matrices

We will visualize our above statement using a program in python.

Input of the program is given below, following which is the output of the program.

Firstly, we construct two random matrices A and B of order $n \times m$ with entries, non-negative integers between 0 to 999. We operate CVT and XOR on matrix A and matrix B, using the function $def\ cvt\_m(x, y)$ and $def\ xor\_m(x, y)$.

Input

```
import numpy as np
import matplotlib.pyplot as plt
def isZero(x):
    for row in x:
        for i in row:
            if i !=0:
                return False
```

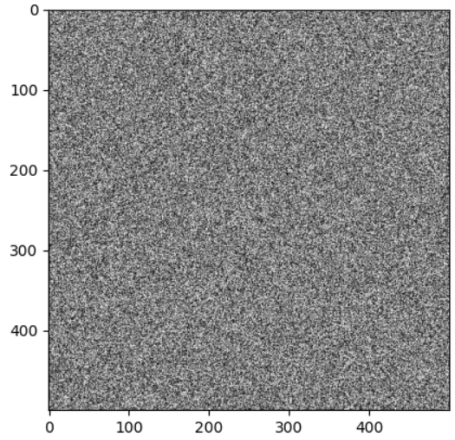
```
    return True
def cvt_m(x, y):
    res_c = np.zeros((n,m), dtype = np.int64)
    for i in range(n):
        for j in range(m):
            res_c[i][j] = (x[i][j]&y[i][j])<<1
    print(res_c)
    plt.imshow(res_c, cmap='gray')
    plt.show()
    return res
def xor_m(x, y):
    res_x = np.zeros((n,m), dtype = np.int64)
    for i in range(n):
        for j in range(m):
            res_x[i][j] = x[i][j]^y[i][j]
    print(res_x)
    plt.imshow(res_x, cmap='gray')
    plt.show()
    return res
def solve(x, y):
    if isZero(x) or isZero(y):
      global result
      result = [x, y]
    else:
      cvt = cvt_m(x, y)
      xor = xor_m(x, y)
      solve(cvt, xor)
n = int(input("Enter n : "))
m = int(input("Enter m : "))
A = np.random.randint(50, size=(n, m))
B = np.random.randint(50, size=(n, m))
print(A)
print(B)
plt.imshow(A, cmap='gray')
plt.show()
plt.imshow(B, cmap='gray')
plt.show()
solve(A, B)
```

Listing 6.1: Recursive CVT + XOR

Output

```
[[ 2 15 28 ...  7 29  1]
 [38 14 24 ... 35  4 16]
 [22 42  1 ... 33 40 36]
 ...
 [41 12 35 ... 31  8 17]
 [ 8 31 25 ... 38  1 16]
 [ 8  8 13 ... 34 38 43]]
```

(a) matrix_A

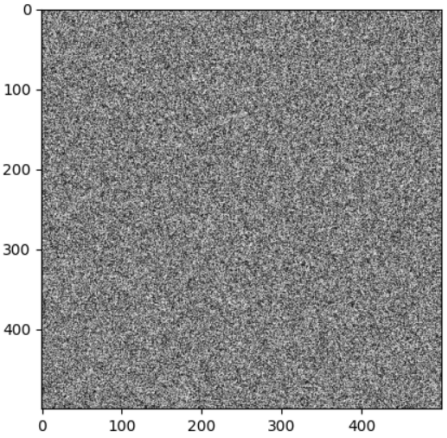


(b) Image formed by matrix_A

Figure 6.1: Output Matrix_A and its Image

```
[[40 32 33 ... 19 13 46]
 [45 22 11 ... 33 11 33]
 [ 6 30 42 ... 32 48 46]
 ...
 [16 46 17 ...  1 27 18]
 [ 4 31 15 ... 46 10 27]
 [45 32 15 ... 15 39 21]]
```

(a) matrix_B

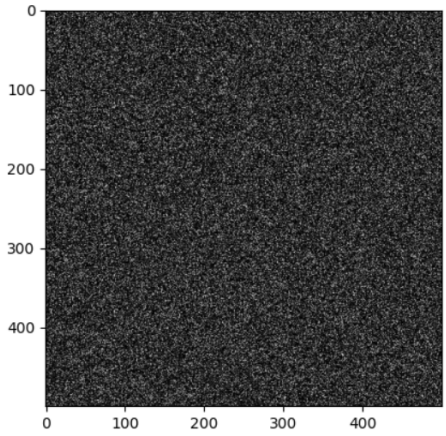


(b) Image formed by matrix_B

Figure 6.2: Output Matrix_B and its Image

```
[[ 0  0  0 ...  6 26  0]
 [72 12 16 ... 66  0  0]
 [12 20  0 ... 64 64 72]
 ...
 [ 0 24  2 ...  2 16 32]
 [ 0 62 18 ... 76  0 32]
 [16  0 26 ...  4 76  2]]
```

(a) CVT(A, B)

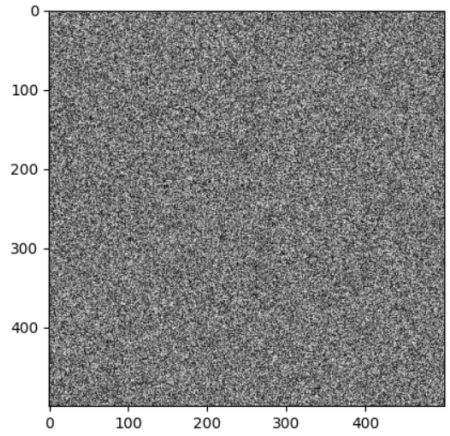


(b) Image formed by CVT(A, B)

Figure 6.3: Output CVT(A, B) and its Image

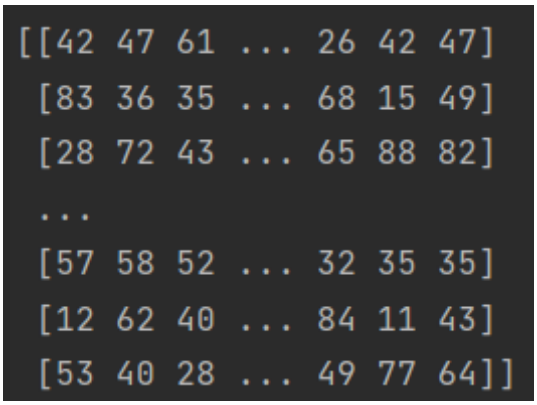
```
[[42 47 61 ... 20 16 47]
 [11 24 19 ...  2 15 49]
 [16 52 43 ...  1 24 10]
 ...
 [57 34 50 ... 30 19  3]
 [12  0 22 ...  8 11 11]
 [37 40  2 ... 45  1 62]]
```

(a) XOR(A, B)

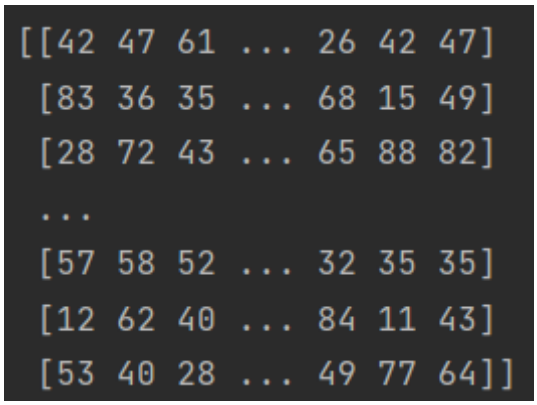


(b) Image formed by XOR(A, B)

Figure 6.4: Output XOR(A, B) and its Image

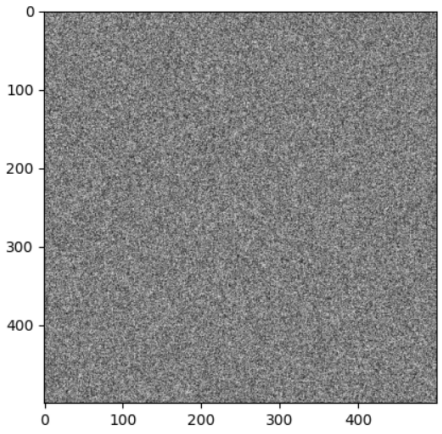


(a) matrix_A + matrix_B

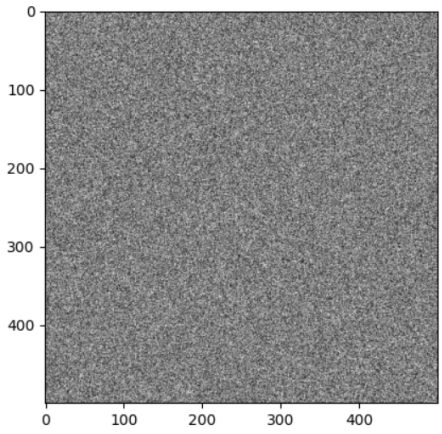


(b) CVT(A, B) + XOR(A, B)

Figure 6.5: Sum of Matrices and their CVT + XOR



(a) Image formed by matrix_A + matrix_B



(b) Image formed by CVT(A, B) + XOR(A, B)

Figure 6.6: Image formed by Sum of Matrices and their CVT + XOR

6.3 Recursive addition of CVT and XOR

We will try to check whether the hypothesis 1 is correct or wrong using the following python code. We recursively do CVT and XOR of two matrices and check whether at the end we get a pair of 0 matrix and sum matrix, where sum matrix is the sum of the initial two matrices. We will observe the output which will help us understand whether our hypothesis is true or false.

```
import numpy as np
import matplotlib.pyplot as plt
def cvt_m(x, y):
    res_c = np.zeros((n, m), dtype=np.int64)
    for i in range(n):
        for j in range(m):
            res_c[i][j] = (x[i][j] & y[i][j]) << 1
    print(res_c)
    plt.imshow(res_c, cmap='gray')
    plt.show()
    return res_c
def xor_m(x, y):
    res_x = np.zeros((n, m), dtype=np.int64)
    for i in range(n):
        for j in range(m):
            res_x[i][j] = x[i][j] ^ y[i][j]
    print(res_x)
    plt.imshow(res_x, cmap='gray')
    plt.show()
    return res_x
n = int(input("Enter n : "))
m = int(input("Enter m : "))
A = B = np.random.randint(50, size=(n, m))
print(A)
print(B)
plt.imshow(A, cmap='gray')
plt.show()
plt.imshow(B, cmap='gray')
plt.show()
cvt_m(A, B)
xor_m(A,B)
L = A + B
```

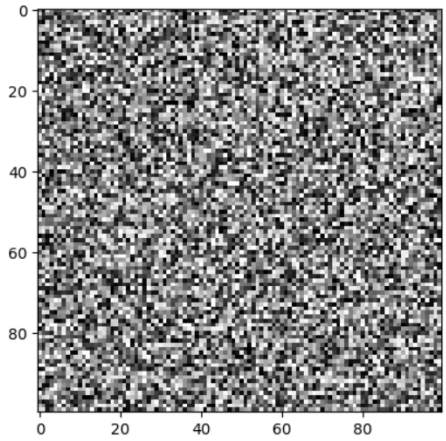
```
K = cvt_m(A,B) + xor_m(A,B)
print(K)
plt.imshow(K, cmap='gray')
plt.show()
print(L)
plt.imshow(K, cmap='gray')
plt.show()
```

Listing 6.2: CVT + XOR

Output

```
[[26  4 43 ...  7  4 28]
 [42  9 27 ... 17 16 27]
 [23  6  6 ... 38 10 30]
 ...
 [16  3  7 ... 43 13  8]
 [35 27  4 ... 43 25 32]
 [43 44  5 ...  8 37  1]]
```

(a) matrix_A



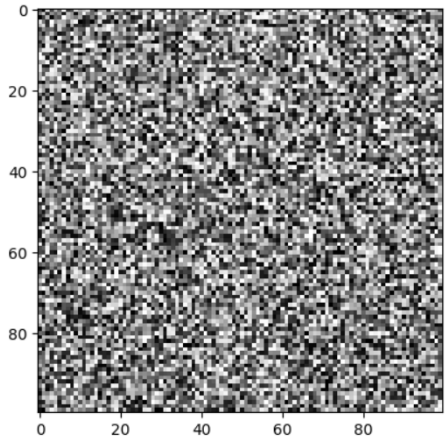
(b) Image formed by matrix_A

Figure 6.7: Output Matrix_A and its Image

Since, the upper limit of all the entries was set to be 50, the maximum element a cell could get is $(49)_{10}$, which in binary is $(110001)_2$. The length of the binary string of the greatest element is 6, so maximum number of possible iteration for CVT of both matrix to be zero as per the hypothesis is 7. Also, we observe that at 7^{th} iteration we get the CVT matrix equal to 0 and the XOR matrix equal to the sum of the initial matrices. This verifies our hypothesis.

```
[[13  1 43 ... 40 34 14]
 [25 38  8 ... 34 33 35]
 [ 0 12 35 ... 23 20 22]
 ...
 [ 5  6 35 ... 10 20 45]
 [13 28 22 ... 35 28 48]
 [ 3  0 34 ... 28 48 45]]
```

(a) matrix_B



(b) Image formed by matrix_B

Figure 6.8: Output Matrix_B and its Image

```
[[16  0 86 ...  0  0 24]
 [16  0 16 ...  0  0  6]
 [ 0  8  4 ... 12  0 44]
 ...
 [ 0  4  6 ... 20  8 16]
 [ 2 48  8 ... 70 48 64]
 [ 6  0  0 ... 16 64  2]]
```

(a) CVT of 1^{st} Iteration

```
[[32  0  0 ...  0  0 32]
 [32  0 32 ...  0  0  0]
 [ 0 16  8 ...  0  0 16]
 ...
 [ 0  8  8 ...  0 16  0]
 [ 4  0  0 ...  0  0  0]
 [ 0  0  0 ... 32  0  0]]
```

(b) CVT of 2^{nd} Iteration

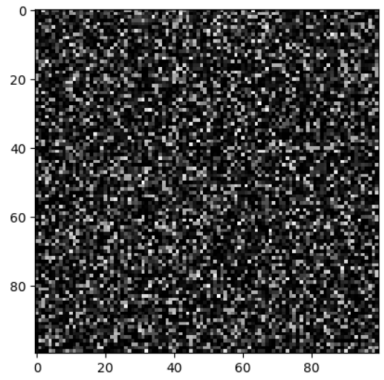
```
[[ 0  0  0 ...  0  0  0]
 [64  0  0 ...  0  0  0]
 [ 0  0  0 ...  0  0  0]
 ...
 [ 0  0  0 ...  0 32  0]
 [ 8  0  0 ...  0  0  0]
 [ 0  0  0 ...  0  0  0]]
```

(c) CVT of 3^{rd} Iteration

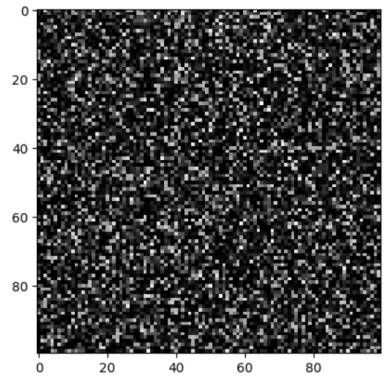
```
[[ 0  0  0 ...  0  0  0]
 [ 0  0  0 ...  0  0  0]
 [ 0  0  0 ...  0  0  0]
 ...
 [ 0  0  0 ...  0  0  0]
 [16  0  0 ...  0  0  0]
 [ 0  0  0 ...  0  0  0]]
```

(d) CVT of 4^{th} Iteration

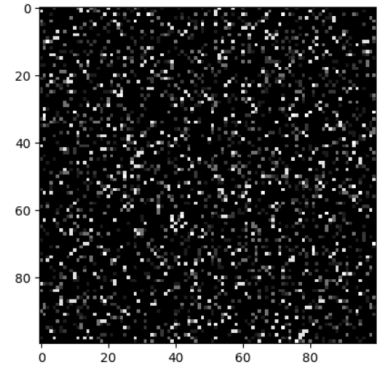
Figure 6.9: CVT matrix of first four Iterations



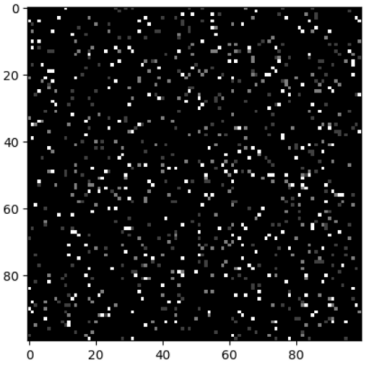
(a) Image of CVT of 1^{st} Iteration



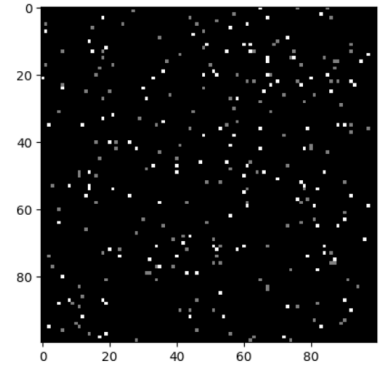
(b) Image of CVT of 2^{nd} Iteration



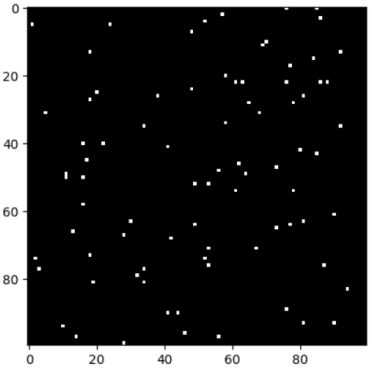
(c) Image of CVT of 3^{rd} Iteration



(d) Image of CVT of 4^{th} Iteration



(e) Image of CVT of 5^{th} Iteration

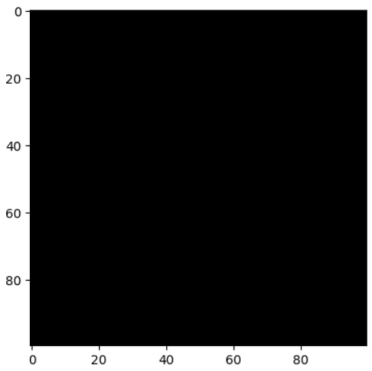


(f) Image of CVT of 6^{th} Iteration

Figure 6.10: Images of CVT of all the $n-1$ Iterations

```
[[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
```

(a) CVT of 7^{th} Iteration

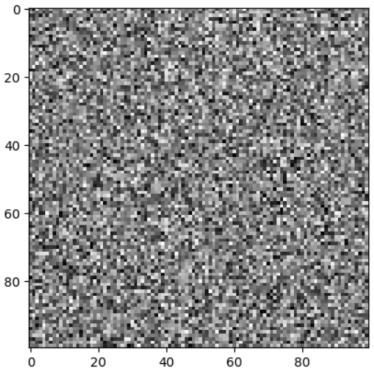


(b) Image of CVT of 7^{th} Iteration

Figure 6.11: CVT and Image, of the last Iteration

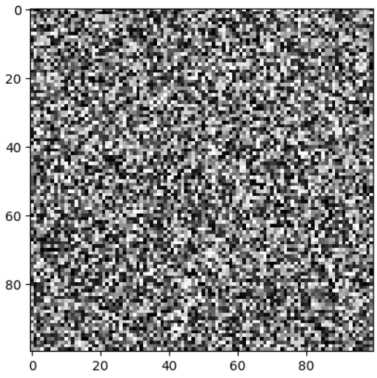
```
[[39  5 86 ... 47 38 42]
 [67 47 35 ... 51 49 62]
 [23 18 41 ... 61 30 52]
 ...
 [21  9 42 ... 53 33 53]
 [48 55 26 ... 78 53 80]
 [46 44 39 ... 36 85 46]]
```

(a) XOR of 7^{th} Iteration

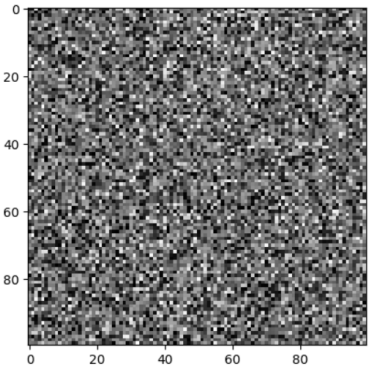


(b) Image of XOR of 7^{th} Iteration

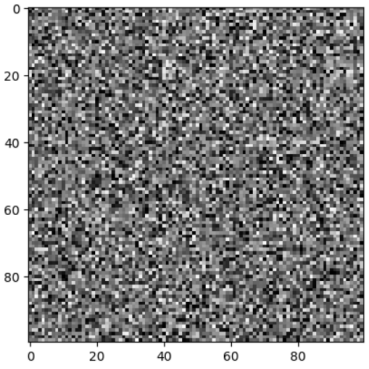
Figure 6.12: XOR and Image, of the last Iteration



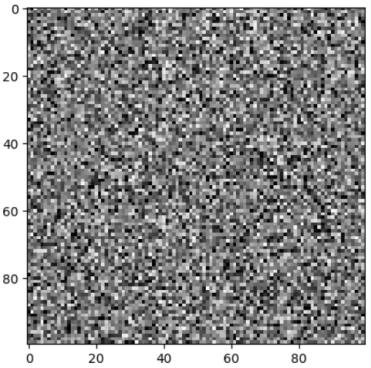
(a) XOR of 1^{st} Iteration



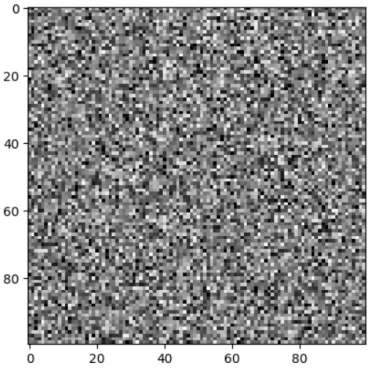
(b) XOR of 2^{nd} Iteration



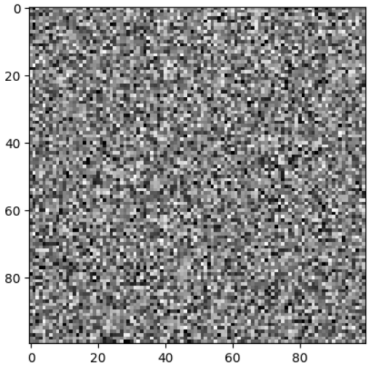
(c) XOR of 3^{rd} Iteration



(d) XOR of 4^{th} Iteration

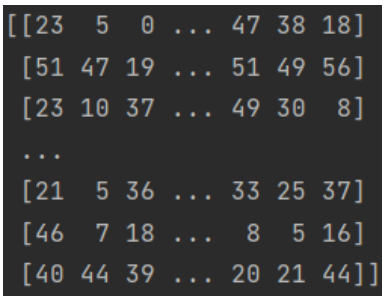


(e) XOR of 5^{th} Iteration

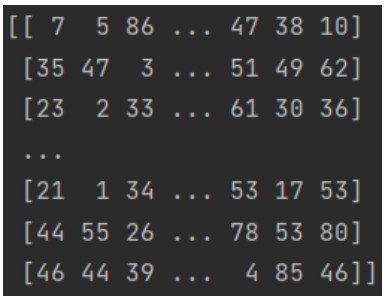


(f) XOR of 6^{th} Iteration

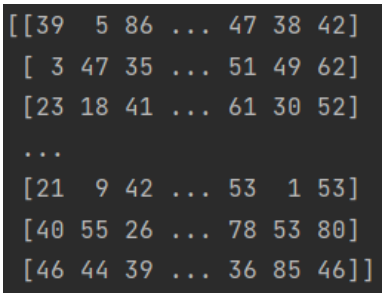
Figure 6.13: XOR of all the $n-1$ Iterations



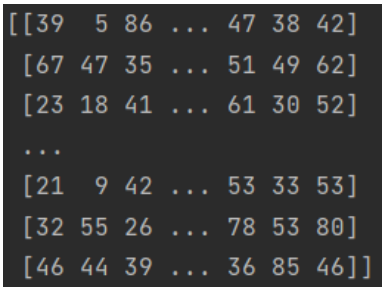
(a) XOR of 1^{st} Iteration



(b) XOR of 2^{nd} Iteration



(c) XOR of 3^{rd} Iteration



(d) XOR of 4^{th} Iteration

Figure 6.14: XOR matrix of first four Iterations

Chapter 7

Q-R Decomposition

For a square matrix A the QR Decomposition converts A into the product of an orthogonal matrix Q (i.e. $Q^TQ = I$) and an upper triangular matrix R. Hence, $A = QR$ [4].

Similar matrices have same eigenvalues and associated eigenvectors. Two matrices A and B are said to be similar if $A = C^{-1}BC$, where C is an invertible matrix.

Let's take a matrix A_0, and we have to determine it's eigenvalues. At the k^{th} step (starting with $k = 0$), we perform the QR decomposition and get $A_k = Q_kR_k$, Q_k being orthogonal matrix and R_k being upper trianguar matrix. We then form $A_{k+1} = R_kQ_k$.

$A_{k+1} = R_kQ_k$
$A_{k+1} = Q_k^{-1}Q_kR_kQ_k$
$A_{k+1} = Q_k^{-1}A_kQ_k$

Therefor, all the A_k are similar. As we continue to iterate, we eventually converge to an upper triangular form of the matrix A_k, figure 7.1.

$$A_k = R_kQ_k = \begin{bmatrix} \lambda_1 & X & \dots & X \\ 0 & \lambda_2 & \dots & X \\ & & \dots & \\ 0 & 0 & \dots & \lambda_n \end{bmatrix}$$

Figure 7.1: A_k in the upper triangular form

7.1 Q-R Decomposition of a Matrix

Input

```
import pprint
import numpy as np
import scipy.linalg

n = int(input("total number of rows : "))
m = int(input("total number of column : "))
A = np.random.randint(1000, size=(n, m))
Q, R = scipy.linalg.qr(A)

print("A:")
pprint.pprint(A)

print("Q:")
pprint.pprint(Q)

print("R:")
pprint.pprint(R)
```

Listing 7.1: Q-R Decomposition

Output

```
[[321, 942, 459, ..., 593,  78, 123],
 [551, 296, 816, ..., 676, 685, 688],
 [275, 533, 909, ..., 576, 301, 496],
 ...,
 [850, 116, 760, ..., 779, 876,  75],
 [178, 743, 992, ..., 753, 252,  36],
 [214, 381, 520, ..., 967, 497, 174]]
```

Figure 7.2: Matrix A

```
([[-0.01733654,  0.05847761,  0.00784209, ..., -0.04185719,
    0.01200556, -0.0221213 ],
  [-0.02975837, -0.0099128 , -0.04225696, ...,  0.04908506,
   -0.00363383, -0.00679906],
  [-0.01485218,  0.0272198 , -0.05198221, ...,  0.04445087,
   -0.00357294, -0.03057339],
  ...,
  [-0.04590674, -0.04373171, -0.03277202, ...,  0.01133225,
    0.00895464, -0.02248883],
  [-0.00961341,  0.05084912, -0.05495651, ...,  0.03361813,
   -0.02778189, -0.00822407],
  [-0.0115577 ,  0.01836232, -0.02448802, ..., -0.00196225,
   -0.01532196,  0.01544566]])
```

Figure 7.3: Q

```
([[-18515.80100887, -13936.45907495, -13796.11602424, ...,
   -13705.88563133, -13534.68688068, -13724.75411019],
  [     0.        ,  11977.06163682,   5209.69955128, ...,
     5188.31619118,   5077.54447954,   5239.29206781],
  [     0.        ,      0.        , -10816.98267685, ...,
    -3309.48344392,  -3613.53747283,  -3058.48926131],
  ...,
  [     0.        ,      0.        ,      0.        , ...,
     -199.69466713,   -197.61253324,    -29.40120864],
  [     0.        ,      0.        ,      0.        , ...,
        0.        ,    504.93403824,    512.02849275],
  [     0.        ,      0.        ,      0.        , ...,
        0.        ,      0.        ,    179.64029876]])
```

Figure 7.4: R

7.2 Pattern formed by Q-R Decomposition of a Matrix

We see from the previous section that the the matrix Q has entries very small and near to zero. So, we will observe that the image of the Q matrix will be nothing but a black image, as all components round off to 0. The matrix R is an upper triangular matrix, hence the image formed by the R will have a pattern in upper half and black in lower half.

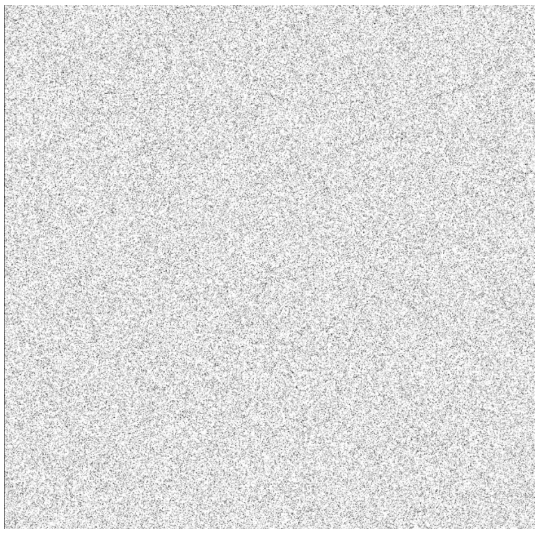
We now see the input code and the output.

Input

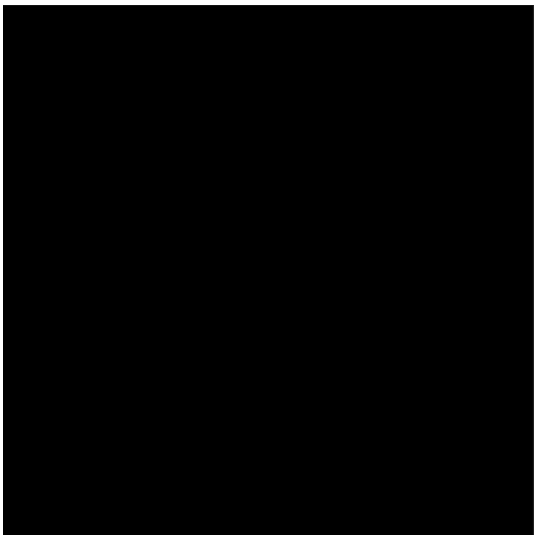
```
import numpy as np
import scipy.linalg
from PIL import Image
import matplotlib.pyplot as plt
array = np.zeros([1000, 1000, 3], dtype=np.uint8)
A = np.random.randint(1000, size=(1000, 1000))
Q, R = scipy.linalg.qr(A)
for i in range(1000):
    for j in range(1000):
        array[i][j] = Q[i][j]
img = Image.fromarray(array)
img.show("Matrix_Q")
plt.show()
for i in range(1000):
    for j in range(1000):
        array[i][j] = R[i][j]
img = Image.fromarray(array)
img.show("Matrix_R")
plt.show()
img = plt.imshow(Q)
img = plt.imshow(R)
```

Listing 7.2: Pattern formation of Q-R Decomposition

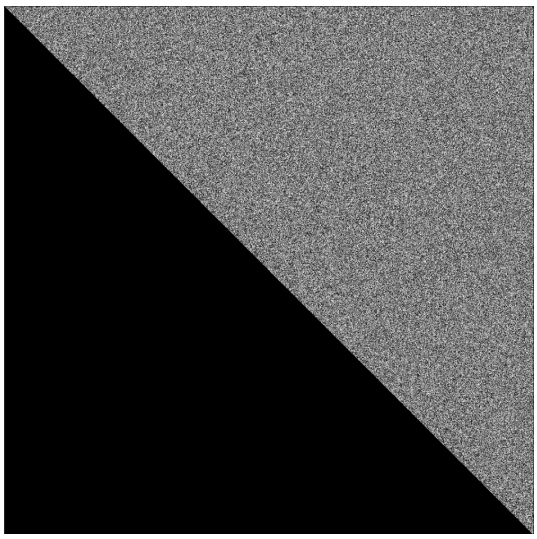
Output



(a) Image of Matrix A



(b) Image of Matrix Q



(c) Image of Matrix R

Figure 7.5: Q-R Decomposed matrix A

Chapter 8

L-U Decomposition

8.1 L-U Decomposition

We will see the steps of the Gaussian-Elimination method to find the L-U decomposition of a matrix [2].

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots a_{1j} & \dots a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots a_{2j} & \dots a_{2n} \\ . & . & . & . & . \\ : & : & : & : & : \\ a_{i1} & a_{i2} & a_{i3} & \dots a_{ij} & \dots a_{im} \\ a_{m1} & a_{m2} & a_{m3} & \dots a_{mj} & \dots a_{mn} \end{bmatrix}$$

Steps:

- We fave convert all a_{ij} for $i > j$ to zero using row operation. The matrix is visualized as in [figure 8.1].
- The multiplier coefficients because of which the respective positions became zero in the U matrix, are then put in the L matrix in the corresponding positions. The matrix is visualized as in [figure 8.2].

$$\mathbf{U} = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & & u_{2n} \\ 0 & 0 & u_{33} & & u_{3n} \\ \vdots & & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & u_{nn} \end{bmatrix}$$

Figure 8.1: Upper Matrix

$$\mathbf{L} = \begin{bmatrix} l_{11} & 0 & 0 & \cdots & 0 \\ l_{21} & l_{22} & 0 & & 0 \\ \vdots & & & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & l_{nn} \end{bmatrix}$$

Figure 8.2: Lower Matrix

8.2 Python program for L-U Decomposiiton of a matrix

We use the SciPy.lingal library of Python to perform LU decomposition of a matrix. The input source code and the output are given below [6].

Input

```
import pprint
import scipy.linalg
import numpy as np
n = int(input("Total number of rows: "))
m = int(input("Total number of columns: "))
A = np.random.randint(10, size=(n, m))
P, L, U = scipy.linalg.lu(A)
print("A:")
pprint.pprint(A)
print("P: ")
pprint.pprint(P)
print("L:")
pprint.pprint(L)
print("U:")
pprint.pprint(U)
```

```
img = plt.imshow(A)
plt.show()
img = plt.imshow(L)
plt.show()
img = plt.imshow(P)
plt.show()
img = plt.imshow(U)
plt.show()
```

Listing 8.1: L-U Decomposition

Output

```
Total number of rows: 1000
Total number of columns: 1000
A:
array([[8, 6, 3, ..., 2, 9, 4],
       [0, 8, 3, ..., 5, 7, 5],
       [2, 7, 4, ..., 2, 8, 4],
       ...,
       [6, 7, 3, ..., 6, 1, 7],
       [9, 2, 7, ..., 1, 1, 8],
       [5, 6, 2, ..., 5, 8, 2]])
```

Figure 8.3: Matrix A

```
P:
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]])
```

(a) Permutation matrix(P)

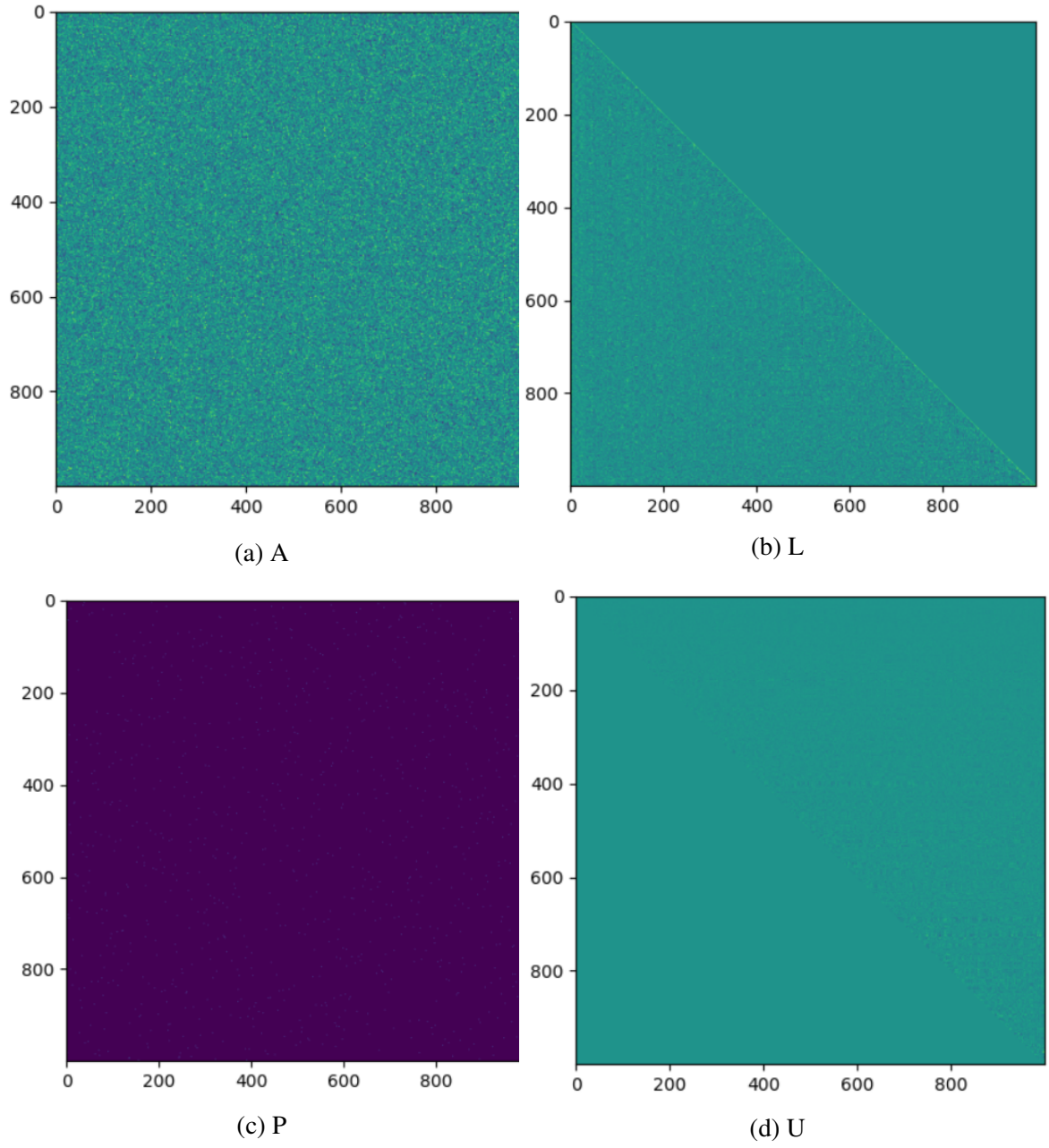
```
L:
array([[ 1.        ,  0.        ,  0.        , ...,  0.        ,
         0.        ,  0.        ],
       [ 0.        ,  1.        ,  0.        , ...,  0.        ,
         0.        ,  0.        ],
       [ 0.55555556,  0.87654321,  1.        , ...,  0.        ,
         0.        ,  0.        ],
       ...,
       [ 0.88888889,  0.69135802,  0.55555556, ...,  1.        ,
         0.        ,  0.        ],
       [ 0.44444444,  0.34567901,  0.11111111, ...,  0.825846  ,
         1.        ,  0.        ],
       [ 0.44444444,  0.2345679 , -0.66666667, ...,  0.46183505,
        -0.02390515,  1.        ]])
```

(b) Lower triangular matrix(L)

```
U:
array([[   9.        ,    2.        ,    2.        , ...,    0.        ,
           5.        ,    1.        ],
       [   0.        ,    9.        ,    9.        , ...,    9.        ,
           1.        ,    4.        ],
       [   0.        ,    0.        ,   -9.        , ...,   -3.88888889,
          -3.65432099,    3.9382716 ],
       ...,
       [   0.        ,    0.        ,    0.        , ...,   22.86116296,
           7.8376432 , -151.71025882],
       [   0.        ,    0.        ,    0.        , ...,    0.        ,
         -30.92552743,  124.96952671],
       [   0.        ,    0.        ,    0.        , ...,    0.        ,
           0.        ,   73.40141596]])
```

(c) Upper triangular matrix(U)

Figure 8.4: P-L-U Decomposed matrix A



(a) A

(b) L

(c) P

(d) U

Figure 8.5: Plot of A, L, P, U

8.3 Python program for L-U Decomposition of an image

Here, we are going to do a visual verification of the LU decomposition. Since, we know that any image can be written as a matrix in Python. We verify the equation $PA = LU$ using a random image, where P is permuation matrix, $L\ U$ are lower triangular and upper triangular matrices respectively.

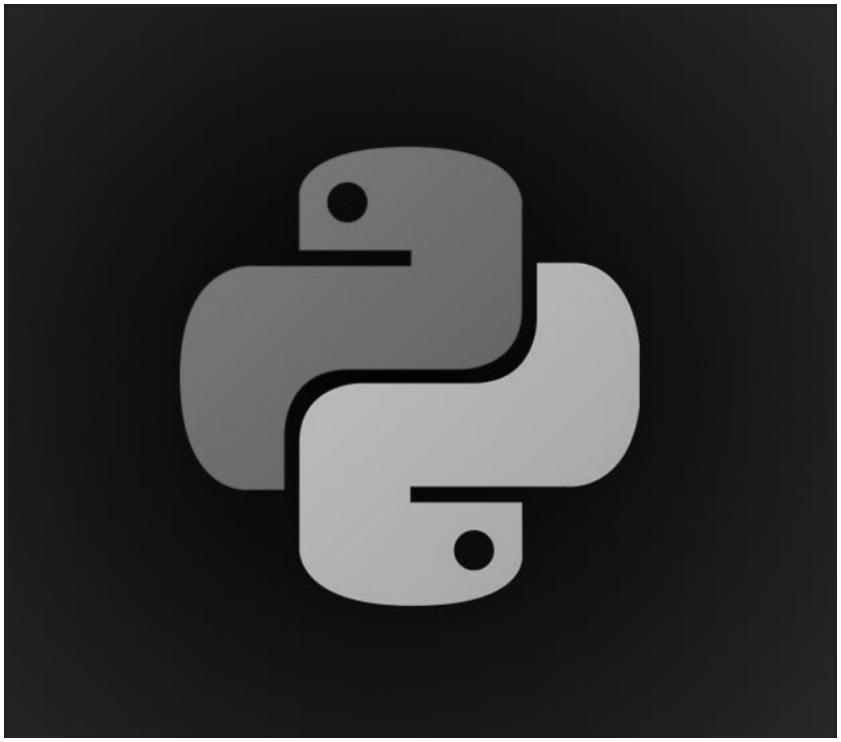
Input

```
import numpy as np
from PIL import Image
import scipy.linalg
import cv2

A = cv2.imread(r'D:\Documents\python.jpg')
X = np.mean(A, -1)
Z = Image.fromarray(X)
P, L, U = scipy.linalg.lu(X)
K = np.matmul(P, L)
J = np.matmul(K, U)
im = Image.fromarray(J)
Z.show()
im.show()
```

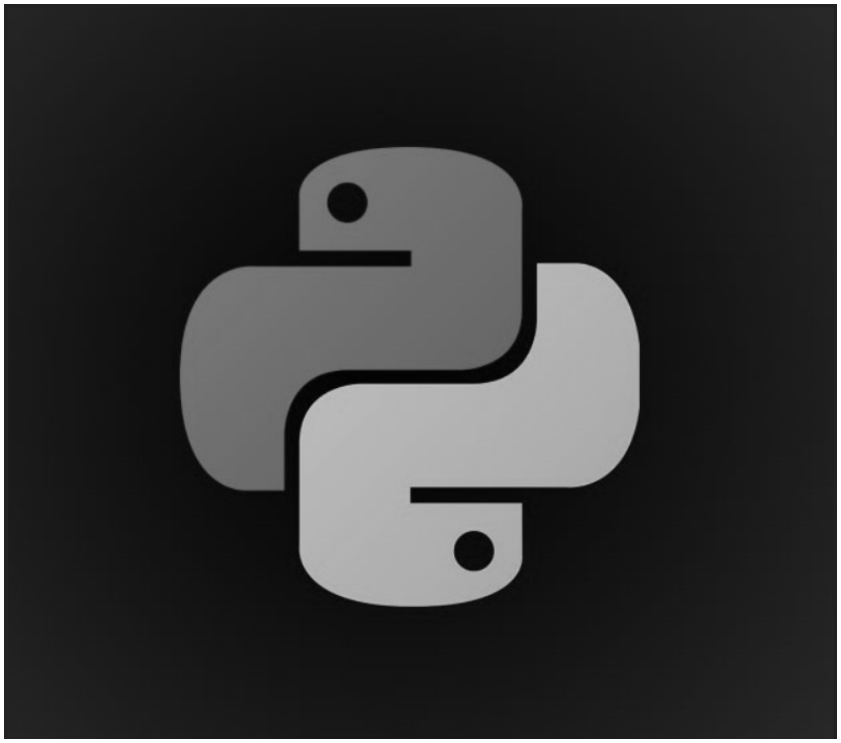
Listing 8.2: L-U Decomposition of Image

Output

The output of the above code helps us visualize the PLU decomposition. We can see in [Figure 8.6], given [Figure 8.6a] is image obtained my matrix X, i.e. the original matrix, and [Figure 8.6b] shows the image formed by product of matrix P, L, U in the given order. We clearly observe that the image in both cases are same, hence, verifying the LU decomposition of a matrix.



(a) Original Image



(b) PLU

Figure 8.6: $A = PLU$

Chapter 9

Singular Value Decomposition

Any matrix A can be decomposed as a product of three different matrices. $A_{n\times m} = U_{n\times n}.\Sigma_{n\times m}.{V^T}_{m\times m}$. Where the columns of U are the left singular vectors (gene coefficient vectors), Σ has singular values (mode amplitudes) and is diagonal, and V^T has rows that are the right singular vectors (expression level vectors). The SVD represents an expansion of the original data in a coordinate system where the covariance matrix is diagonal.

9.1 Steps to find out SVD of a Matrix

Calculating the SVD consists of finding the eigenvalues and eigenvectors of AA^T and A^TA. The eigenvectors of A^TA make up the columns of V , the eigenvectors of AA^T make up the columns of U. Also, the singular values in Σ are square roots of eigenvalues from AA^T or A^TA. The singular values are the diagonal entries of the Σ matrix and are arranged in descending order.

We know that for an $n \times n$ matrix W, then a nonzero vector x is the eigenvector of W if $Wx = \lambda x$

- We find the matrix AA^T and A^TA, followed by their eigenvalues.
- We then find the square root of the eigenvalues to get the singular values and then place it on the diagonal of the matrix Σ in descending order.
- Now, find the eigenvectors corresponding to the eigenvalues.

- We put eigenvectors of A^TA in V and eigenvectors of AA^T in U corresponding to the eigenvalues in Σ.

9.2 Python program for SVD of a Matrix

We are going to verify the singular value decomposition of a matrix using a Python program, and the image of the matrix [5].

Input

```
import numpy as np
import numpy as geek
from PIL import Image
array = np.zeros([1000, 1000, 3], dtype=np.uint8)

n = int(input("Total number of rows: "))
# total number of rows
m = int(input("Total number of columns: "))
# total number of columns
X = np.random.randint(100, size=(n, m))
img = Image.fromarray(X)
img.show("X")

U, S, VT = np.linalg.svd(X)
S = np.diag(S)
print(X)
print(U)
print(S)
print(VT)
A = geek.zeros([n, m])
for i in range(n):
    for j in range(m):
        for k in range(n):
            A[i][j] += U[i][k] * S[k][j]

B = geek.zeros([n, m])
for i in range(n):
    for j in range(m):
        for k in range(m):
            B[i][j] += A[i][k] * VT[k][j]
print(B)
img = Image.fromarray(B)
```

```
33 img.show("B")
```

Listing 9.1: SVD Of Matrix

Output

```
Total number of rows: 500
Total number of columns: 500
[[49 99 37 ... 76 82  9]
 [19 86 90 ... 92 23 72]
 [33 72  6 ... 50 85 61]
 ...
 [47 59  4 ... 37 82 73]
 [33 36  4 ...  9  7 61]
 [73 14 83 ... 60 35 17]]
```

Figure 9.1: X

```
[[-0.04475294 -0.0134806   0.00419457 ...  0.05480004  0.00131507
  -0.01246431]
 [-0.04696724  0.02117294 -0.08970701 ... -0.09138692  0.03460514
   0.03774533]
 [-0.0445838  -0.0059431   0.04623022 ...  0.01784501  0.04113214
  -0.00034019]
 ...
 [-0.04422551  0.02938561 -0.02677886 ... -0.04721066 -0.05697856
  -0.02716013]
 [-0.04513346 -0.01380737  0.00369248 ...  0.02113172 -0.04617784
   0.05380599]
 [-0.04568573 -0.01572959  0.02020698 ... -0.03440181  0.00714916
   0.00524433]]
```

Figure 9.2: U

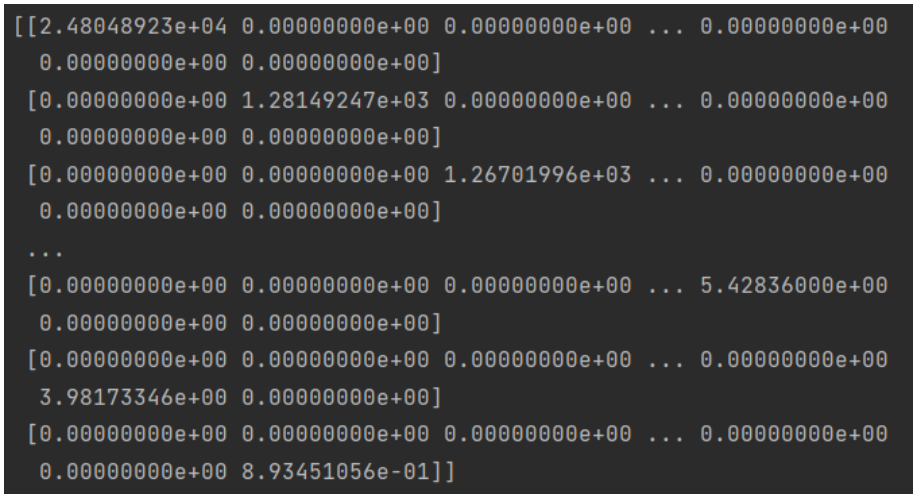


Figure 9.3: Σ

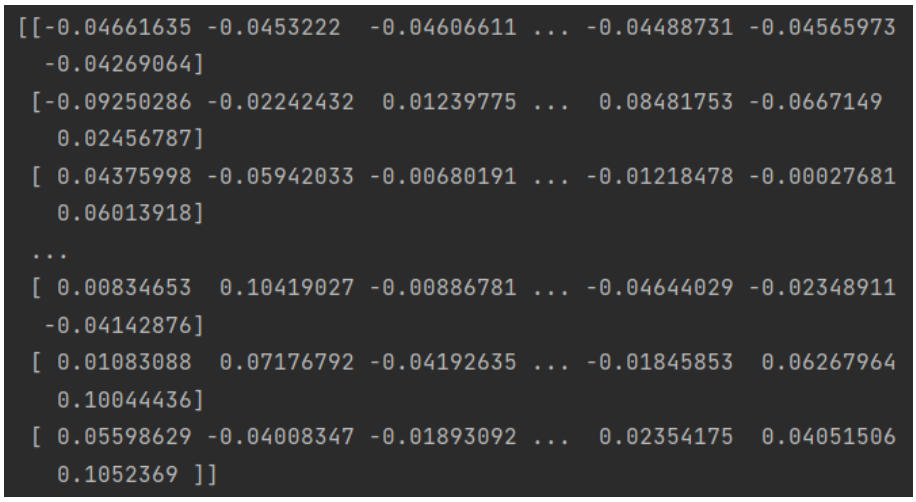


Figure 9.4: V^T

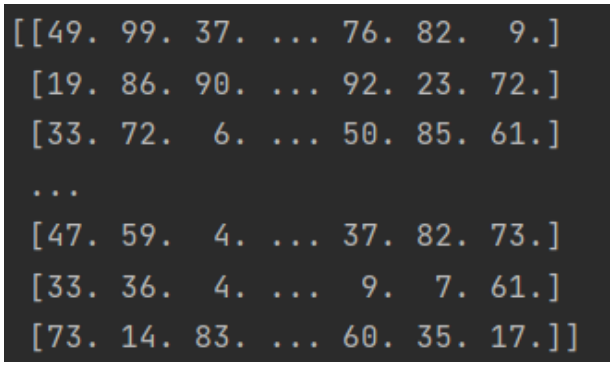
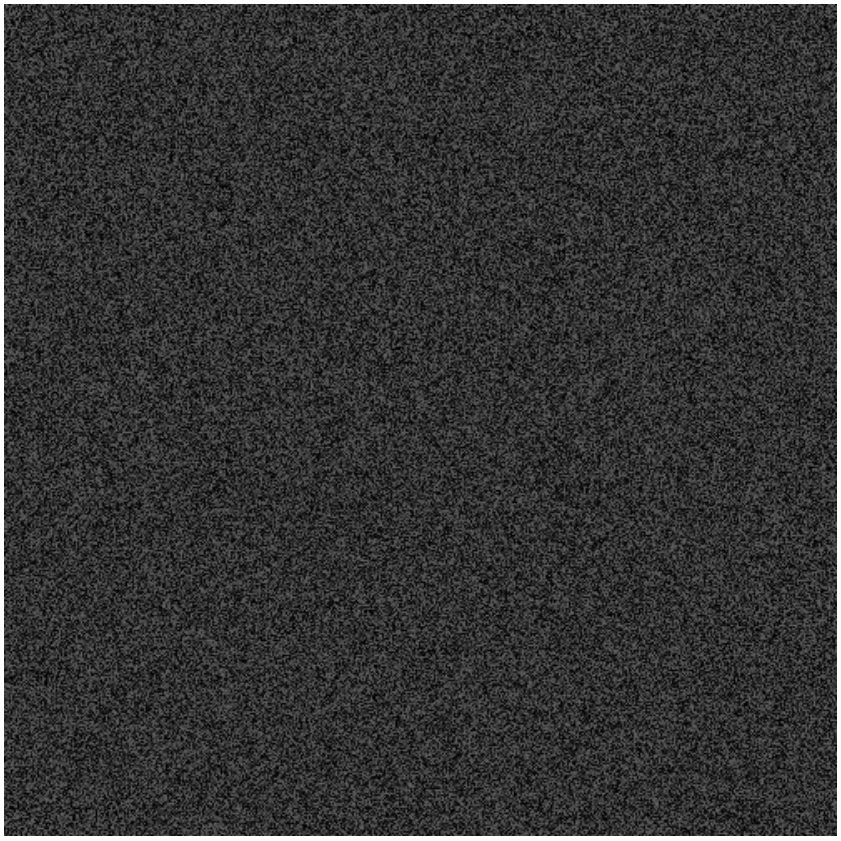
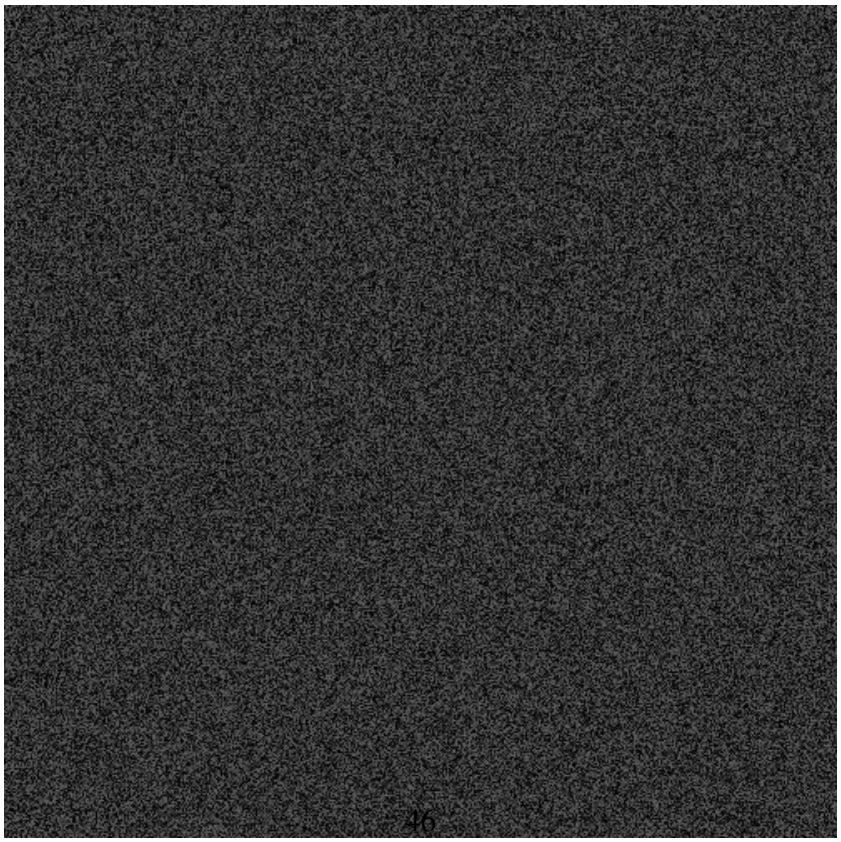


Figure 9.5: $U\Sigma V^T$



(a) Original Image



(b) $U\Sigma V^T$

Figure 9.6: Comparing Images

9.3 Python program for SVD of an Image

Since, SVD can be done only to a 2D image, we use $X = np.mean(A, -1)$, in order to convert the coloured image (3D image) into black & white image (2D image).

The "$full\_matrices = 0$" signifies that we are doing the economy singular value decomposition, i.e. we extract the first "r columns" of U, first "r rows" of V^T and first "$r \times r$" block of Σ. We then extract the image upto rank 5, 21, 40, 488 and analyse them in the output.

Input

```
from matplotlib.image import imread
import matplotlib.pyplot as plt
import numpy as np

A = imread('D:\Documents\python.jpg')
X = np.mean(A, -1);

img = plt.imshow(256-X)
img.set_cmap('gray')
plt.axis('off')
plt.show()

U, S, VT = np.linalg.svd(X, full_matrices=0)
S = np.diag(S)

j = 0
for r in (5, 21, 40, 488):
  Xapprox = U[:, :r] @ S[0:r, :r] @ VT[:r,:]
  plt.figure(j+1)
  j += 1
  img = plt.imshow(256-Xapprox)
  img.set_cmap('gray')
  plt.axis('off')
  plt.title('r = ' + str(r))
  plt.show()

plt.figure(1)
plt.semilogy(np.diag(S))
plt.title('Singular Value')
plt.show()
```

```
31
32 plt.figure(2)
33 plt.plot(np.cumsum(np.diag(S))/np.sum(np.diag(S)))
34 plt.title('Singular Value: Cumulative Sum')
35 plt.show()
```

Listing 9.2: SVD of Image

Output

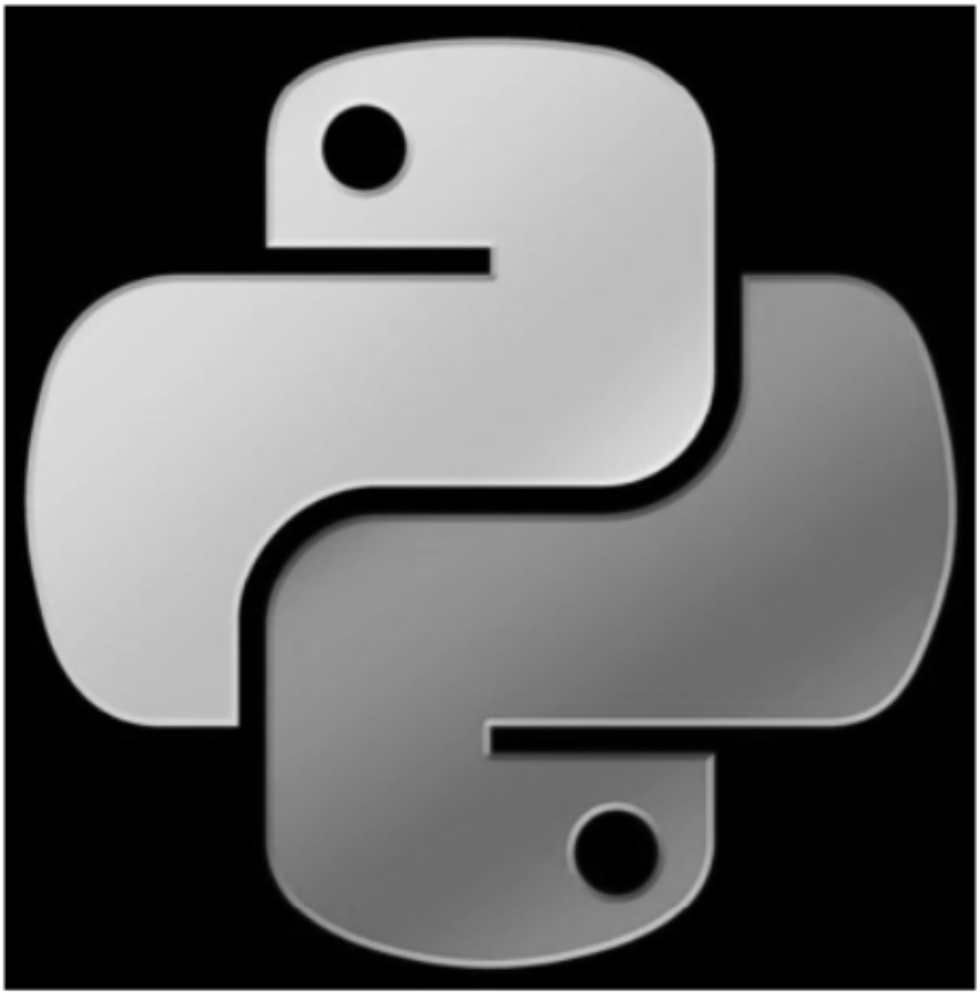
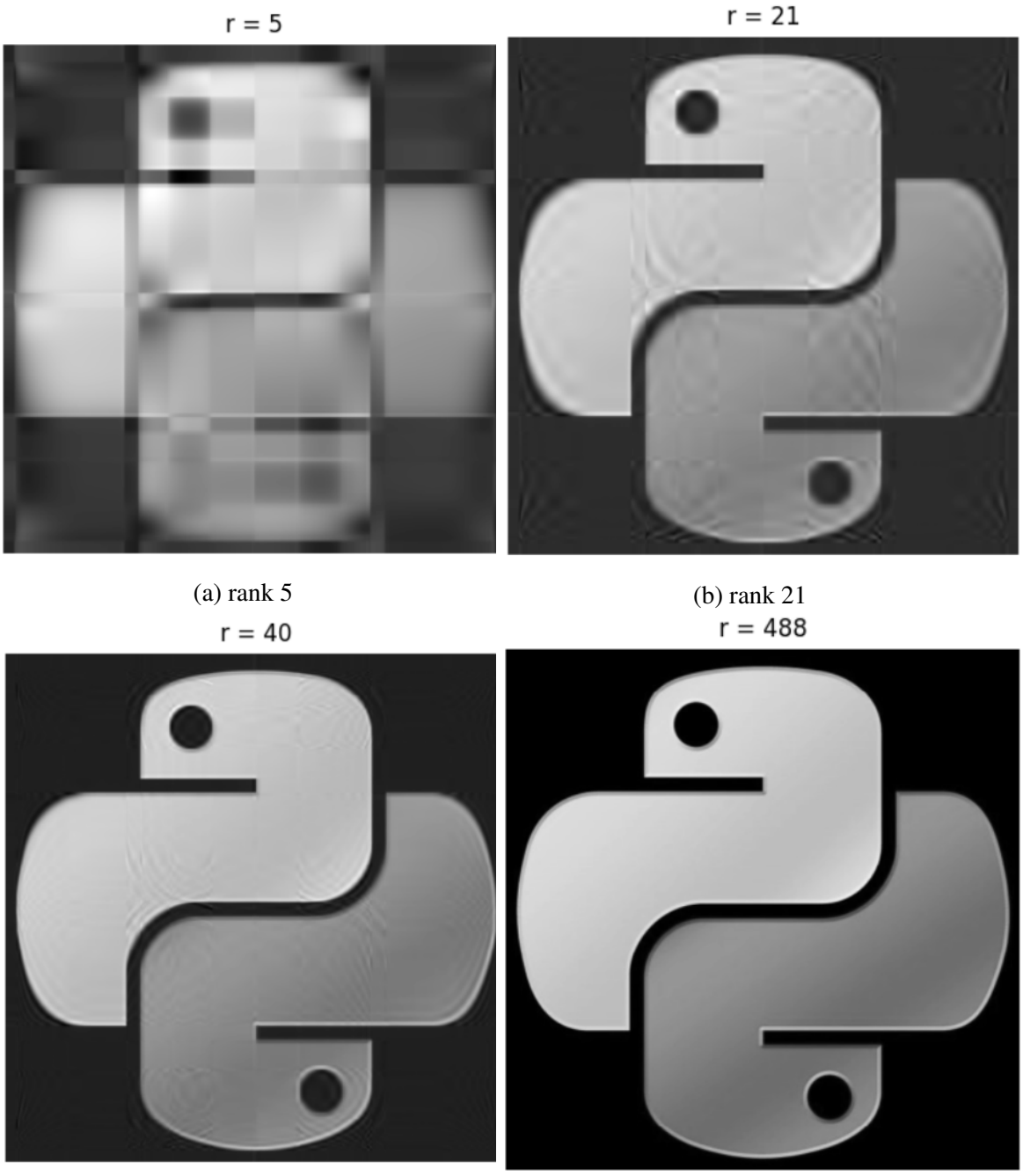


Figure 9.7: Initial Image



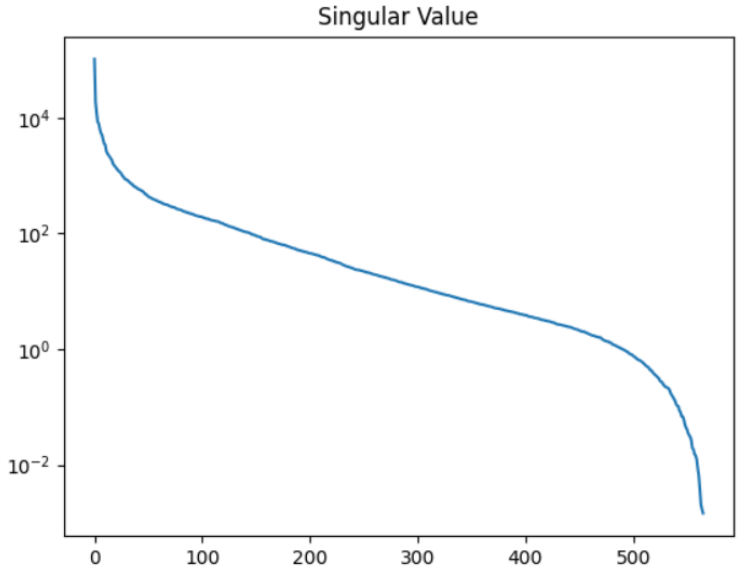
(a) rank 5

(b) rank 21

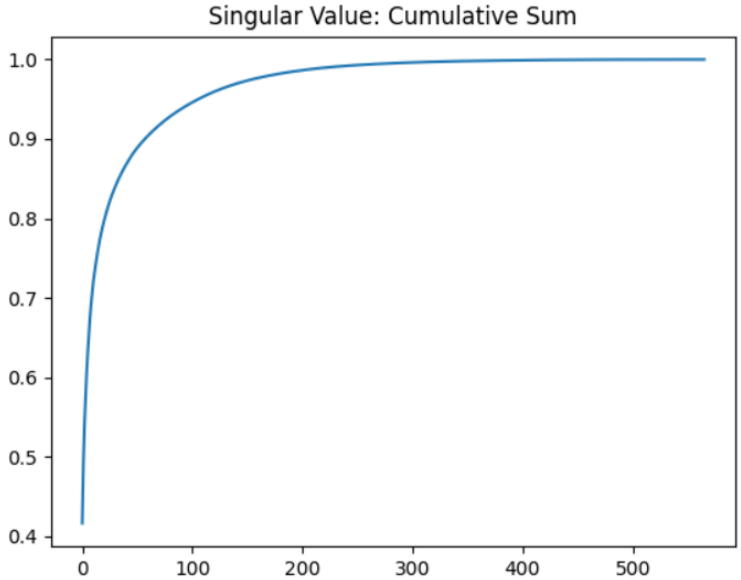
(c) rank 40

(d) rank 488

Figure 9.8: Decomposed Image



(a) $log(\sigma_j)$ vs j



(b) $\frac{(\sum_{j=1}^{r} \sigma_j)}{(\sum_{j=1}^{m} \sigma_j)}$ vs j

Figure 9.9: Graphs showing importance of Singular Values

We can see that the image in [Figure 9.8] at rank 21 can be easily recognized and is even clearer at rank 40. From this we can conclude that the important data of the image is stored in the matrix somewhere between rank 21 and 40. Also, even if we exclude the further rows, we still get the required image. So, we can easily reduce the amount of data used by the image.

The graph in [Figure 9.9a], shows the importance of σ i.e the singular values. The graph between $log(\sigma_j)$ vs j shows the importance of singular values with increasing rank of the matrix. We can clearly conclude that it is reducing, hence, we get to understand from the graph that important parts of an image lie in few initial rows and singular values of the matrix.

The graph in [Figure 9.9b], shows the ratio of matrix covered by the first i singular values and by all the singular values. We can observe from the graph that somewhere around 100, we get almost the complete matrix.

Chapter 10

Analyzing few Sequence

We have seen various types of decomposition of matrices and image formed by them. Now, we will be using all of the above decompositions to create image of sequences, decompose them and analyze them.

10.1 Fibonacci Sequence

The Fibonacci numbers, commonly denoted F_n , form a sequence, called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1. That is, $F_0 = 0$, $F_1 = 1$
And $F_n = F_{n-1} + F_{n-2}$ for $n > 2$
The beginning of the sequence is thus:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

10.1.1 CVT Of Fibonacci Sequence

Input

```
42 from matplotlib.image import imread
43 import matplotlib.pyplot as plt
44 import numpy as np
45 from PIL import Image, ImageEnhance
46 import cv2
47 c = np.zeros([1000, 1000, 3], dtype=np.uint8)
48
49 def cvt(x, y):
50   return (x & y) << 1
```

```

fibonacci_sequence = [0, 1]
m = 0
n = 1
while True:
    z = m + n
    if z < 1000:
        fibonacci_sequence.append(z)
    else:
        break
    m, n = n, z

for i in fibonacci_sequence:
    for j in fibonacci_sequence:
        c[i][j] = cvt(i, j)

img = Image.fromarray(c)
img2 = ImageEnhance.Contrast(img)
img2.enhance(200).show()
img2.enhance(200).save("fibo_1.png")
im3 = cv2.imread("fibo_1.png")
inv = 255 - im3
cv2.imwrite("fibo1.png", inv)
cv2.imshow("Inverted Image", inv)
cv2.waitKey(1000)
cv2.destroyWindow("Inverted Image")

```

Listing 10.1: CVT of Fibonacci Sequence

Output

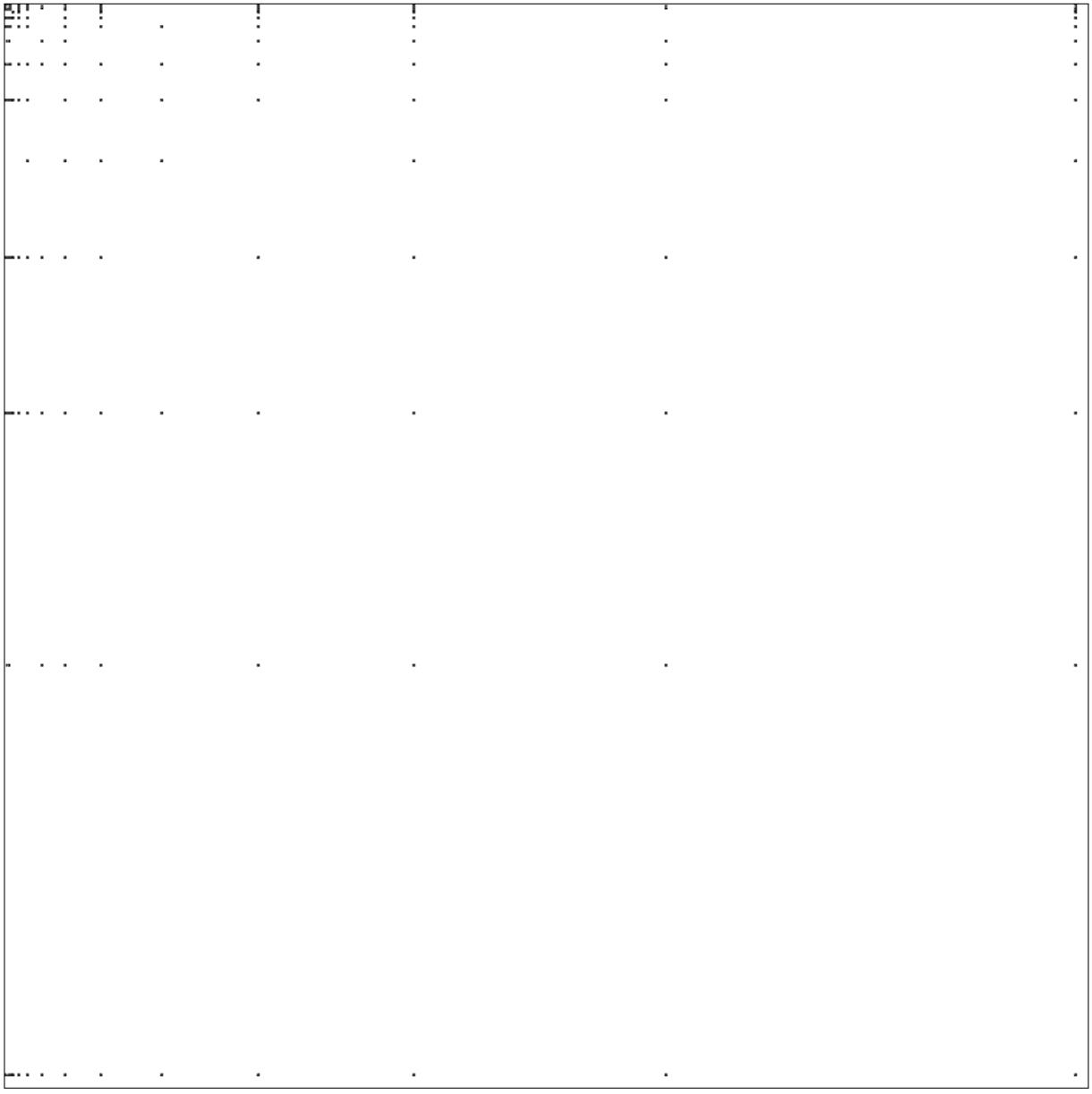


Figure 10.1: Image of CVT of Fibonacci Sequence

10.1.1.1 SVD of CVT of Fibonacci Sequence

Input

```
A = imread('fibo1.png')
X = np.mean(A, -1);

img = plt.imshow(X)
img.set_cmap('gray')
plt.axis('off')
plt.show()

U, S, VT = np.linalg.svd(X, full_matrices=0)
S = np.diag(S)

j = 0
for r in (1, 5, 10):
  Xapprox = U[:, :r] @ S[0:r, :r] @ VT[:r,:]
  plt.figure(j+1)
  j += 1
  img = plt.imshow(Xapprox)
  plt.xlim(0, 650)
  plt.ylim(650, 0)
  img.set_cmap('gray')
  plt.title('r = ' + str(r))
  plt.show()


plt.figure(1)
plt.xlim(0, 100)
plt.semilogy(np.diag(S))
plt.title('Singular Value')
plt.show()

plt.figure(2)
plt.xlim(0, 100)
plt.plot(np.cumsum(np.diag(S))/np.sum(np.diag(S)))
plt.title('Singular Value: Cumulative Sum')
plt.show()
```

Listing 10.2: SVD of CVT of Fibonacci Sequence

Output

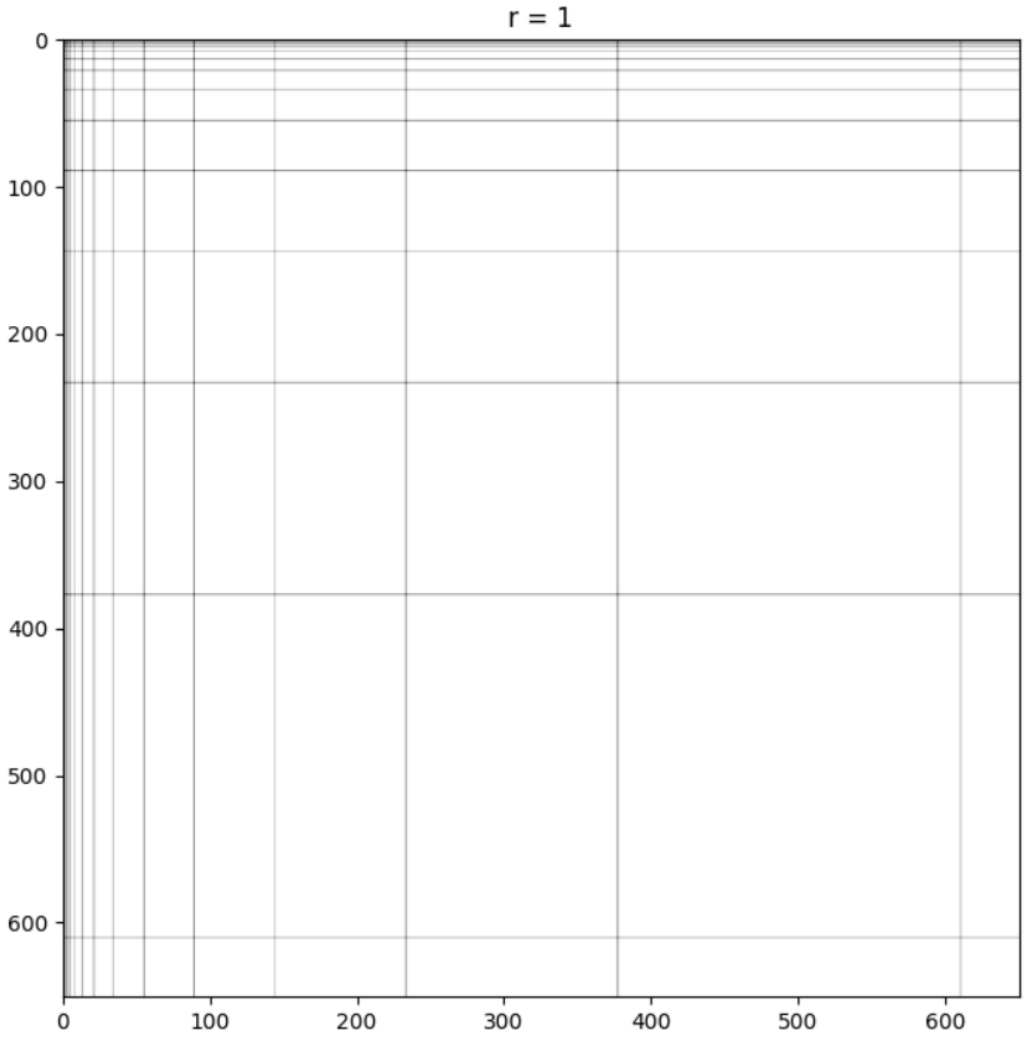


Figure 10.2: SVD at Rank 1

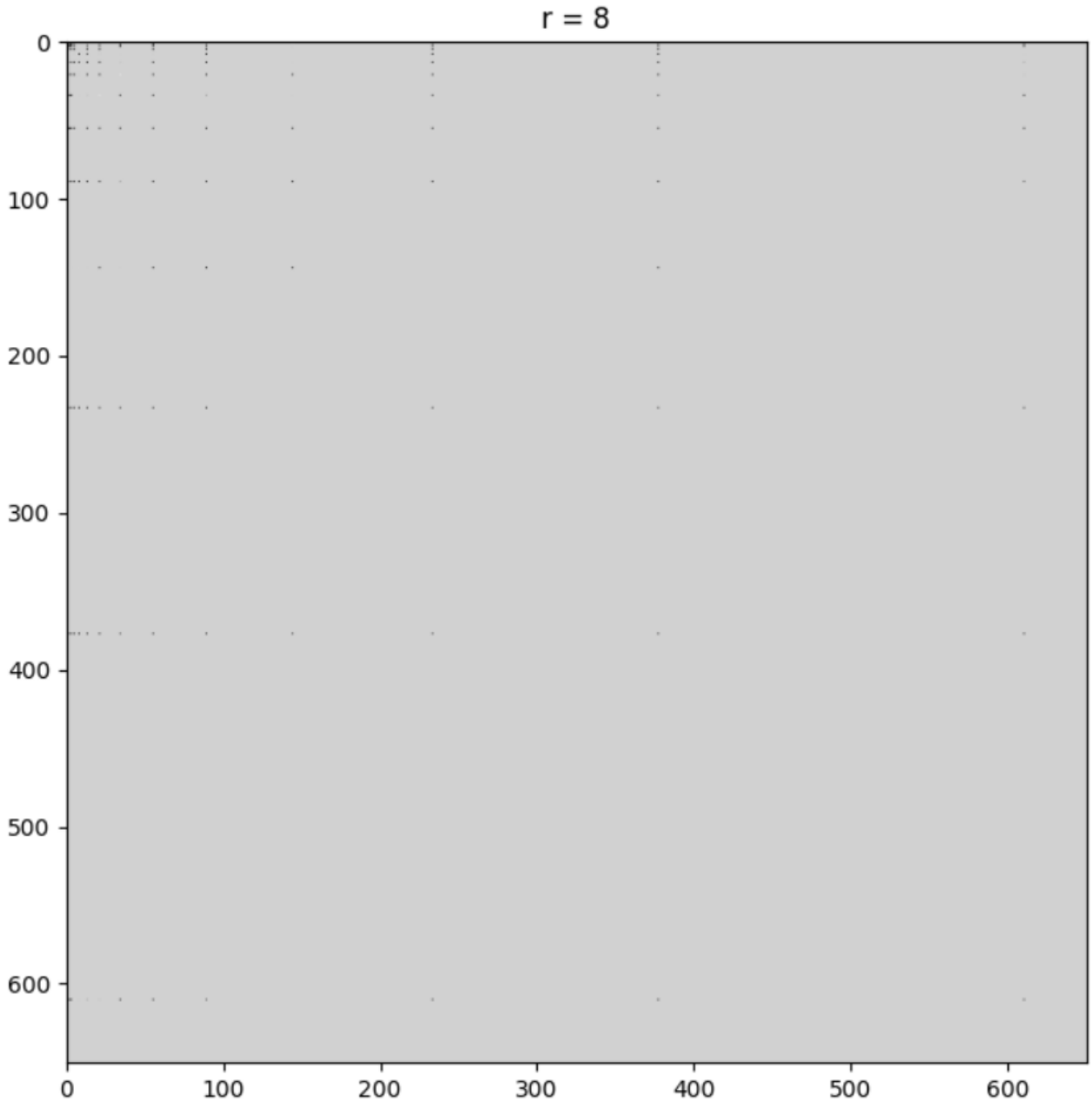


Figure 10.3: SVD at Rank 8

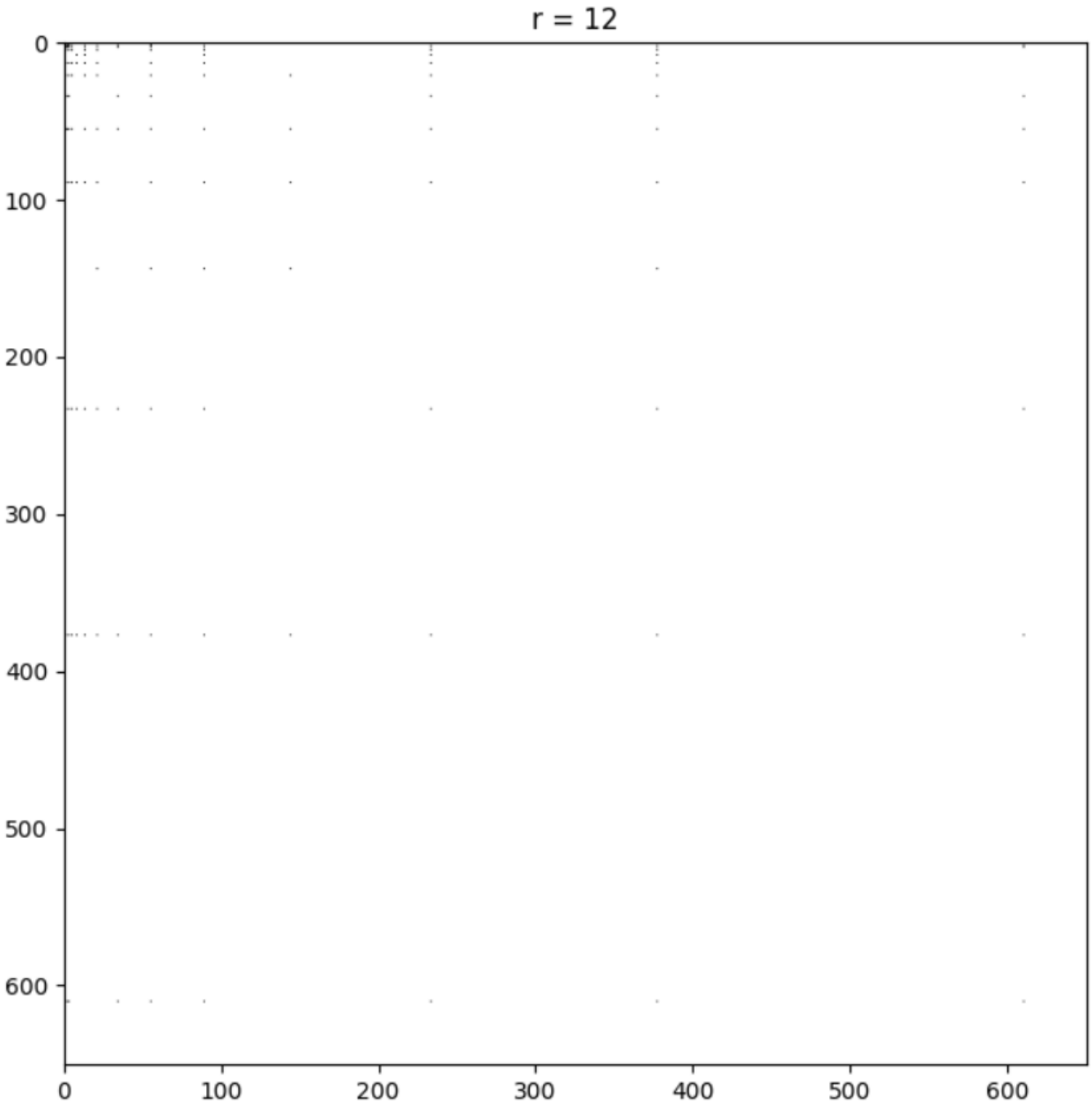
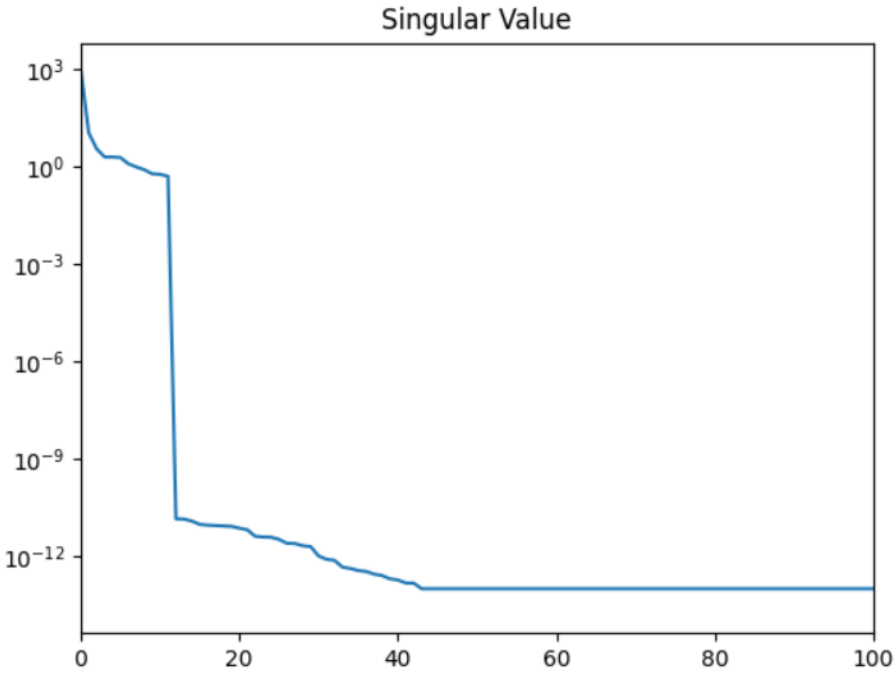
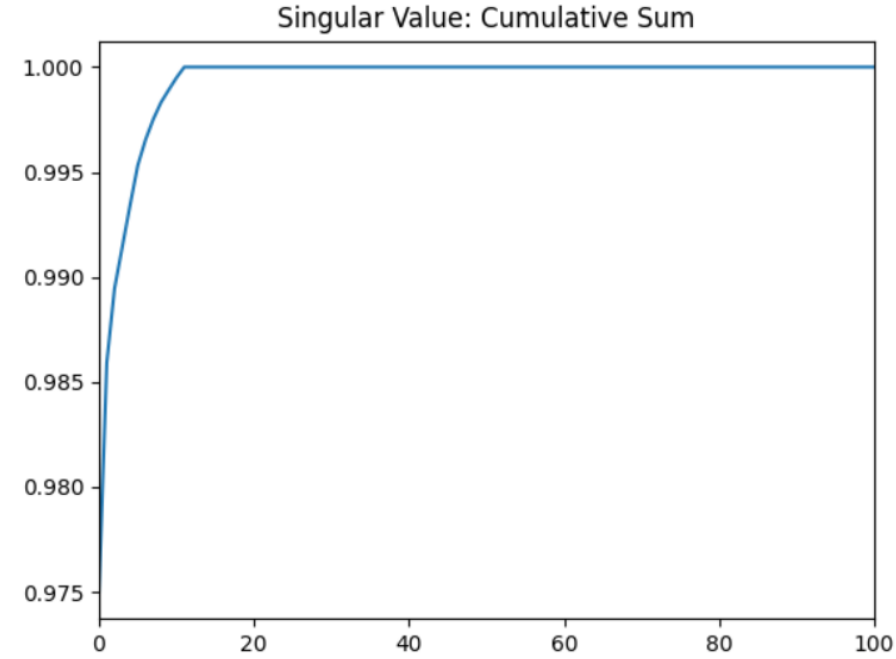


Figure 10.4: SVD at Rank 12



(a) $log(\sigma_j)$ vs j



(b) $\frac{(\sum_{j=1}^{r} \sigma_j)}{(\sum_{j=1}^{m} \sigma_j)}$ vs j

Figure 10.5: Graphs showing importance of Singular Values

10.1.2 XOR Of Fibonacci Sequence

Input

```
78 from matplotlib.image import imread
79 import matplotlib.pyplot as plt
80 import numpy as np
81 from PIL import Image, ImageEnhance
82 import cv2
83 c = np.zeros([1000, 1000, 3], dtype=np.uint8)
84
85 def xor(x, y):
86   return x^y
87
88 fibonacci_sequence = [0, 1]
89 m = 0
90 n = 1
91 while True:
92     z = m + n
93     if z < 1000:
94         fibonacci_sequence.append(z)
95     else:
96         break
97     m, n = n, z
98
99 for i in fibonacci_sequence:
100     for j in fibonacci_sequence:
101         c[i][j] = xor(i, j)
102
103 img = Image.fromarray(c)
104 img2 = ImageEnhance.Contrast(img)
105 img2.enhance(200).save("fibo_2.png")
106 im3 = cv2.imread("fibo_2.png")
107 inv = 255 - im3
108 cv2.imwrite("fibo2.img", inv)
109 cv2.imshow("Inverted Image", inv)
110 cv2.waitKey(1000)
111 cv2.destroyWindow("Inverted Image")
112
```

Listing 10.3: XOR of Fibonacci Sequence

Output

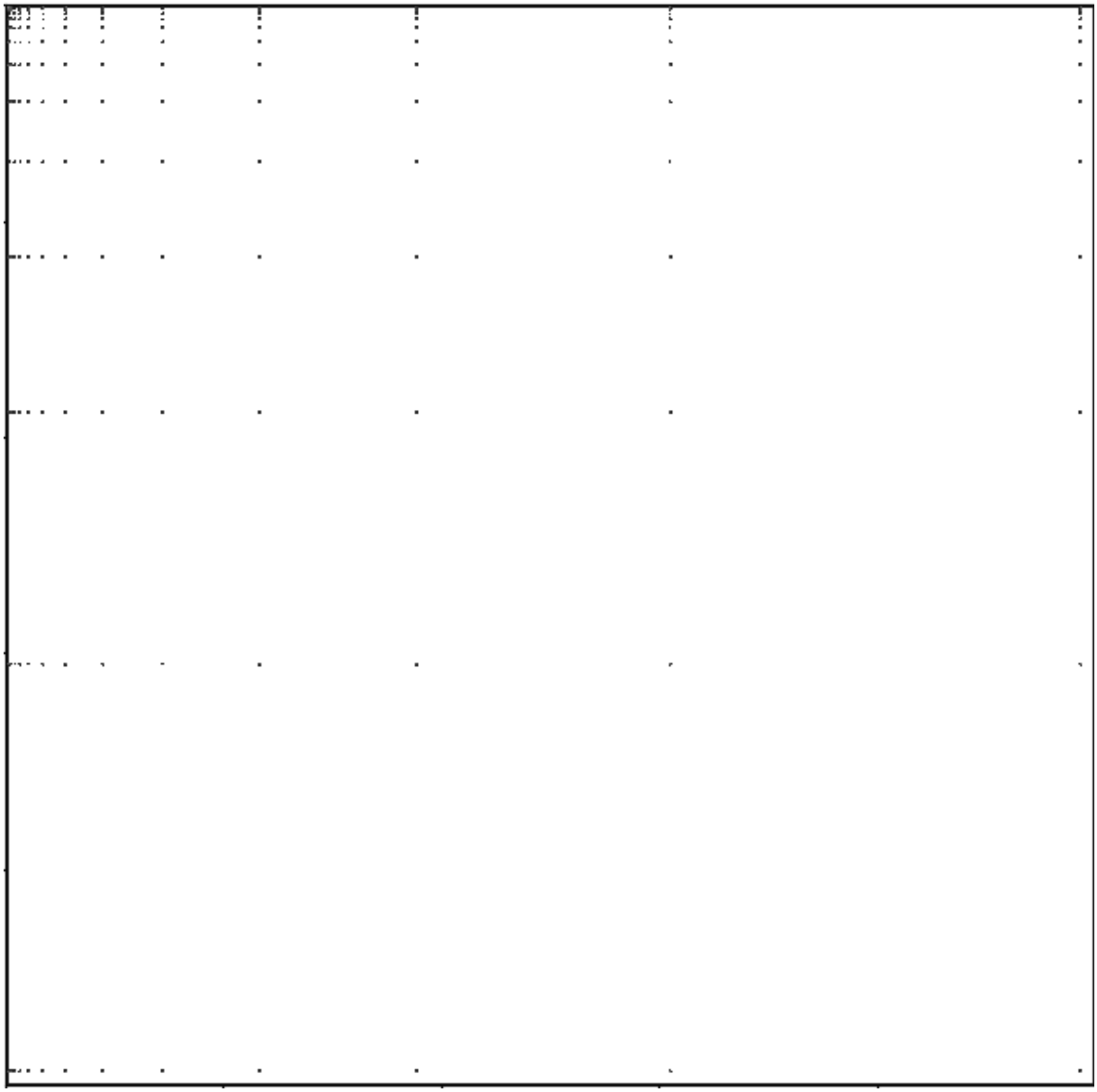


Figure 10.6: Image of XOR of Fibonacci Sequence

10.1.2.1 SVD of XOR of Fibonacci Sequence

Input

```
A = imread('fibo.png')
X = np.mean(A, -1);

img = plt.imshow(X)
img.set_cmap('gray')
plt.axis('off')
plt.show()

U, S, VT = np.linalg.svd(X, full_matrices=0)
S = np.diag(S)

j = 0
for r in (3, 8, 16):
  Xapprox = U[:, :r] @ S[0:r, :r] @ VT[:r,:]
  plt.figure(j+1)
  j += 1
  img = plt.imshow(Xapprox)
  plt.xlim(0, 650)
  plt.ylim(650, 0)
  img.set_cmap('gray')
  plt.title('r = ' + str(r))
  plt.show()

plt.figure(1)
plt.xlim(0, 100)
plt.semilogy(np.diag(S))
plt.title('Singular Value')
plt.show()

plt.figure(2)
plt.xlim(0, 100)
plt.plot(np.cumsum(np.diag(S))/np.sum(np.diag(S)))
plt.title('Singular Value: Cumulative Sum')
plt.show()
```

Listing 10.4: SVD of XOR of Fibonacci Sequence

Output

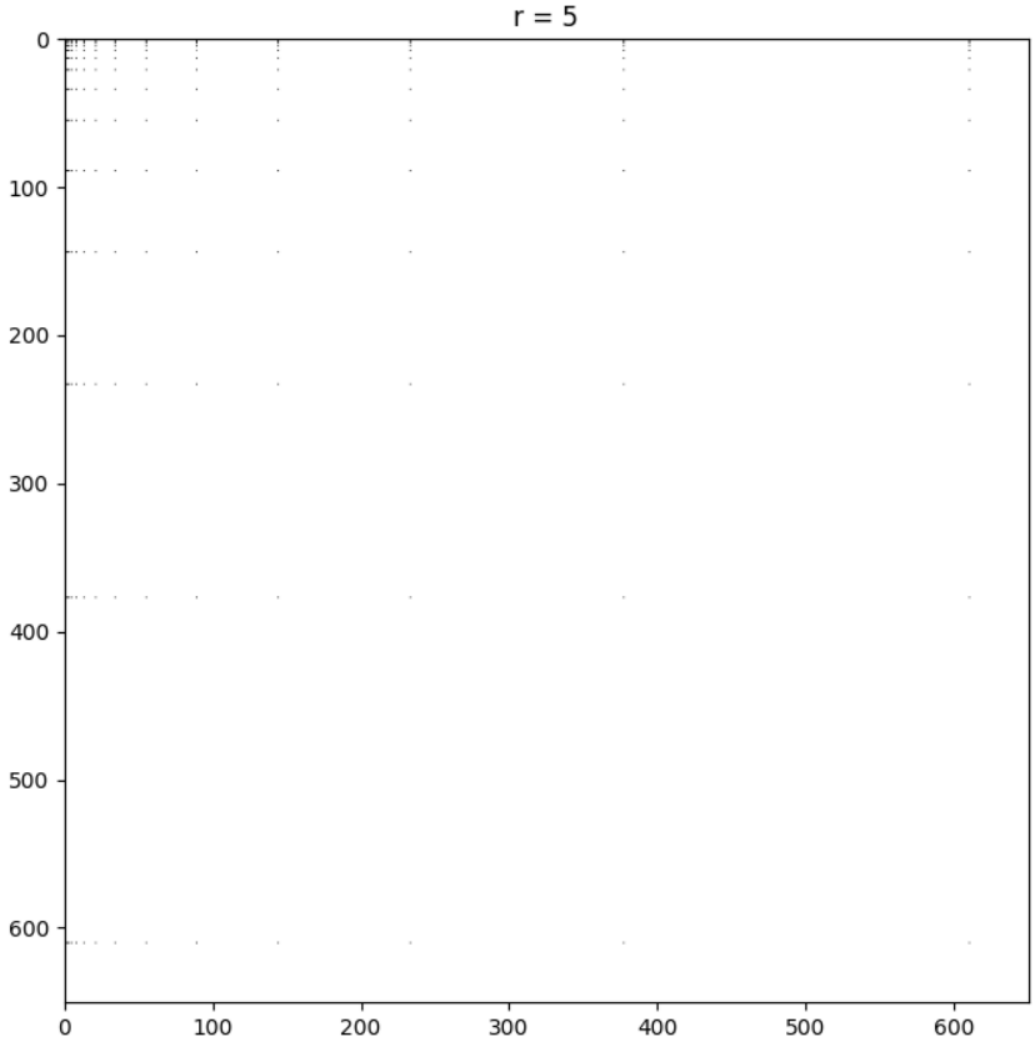


Figure 10.7: SVD at Rank 5

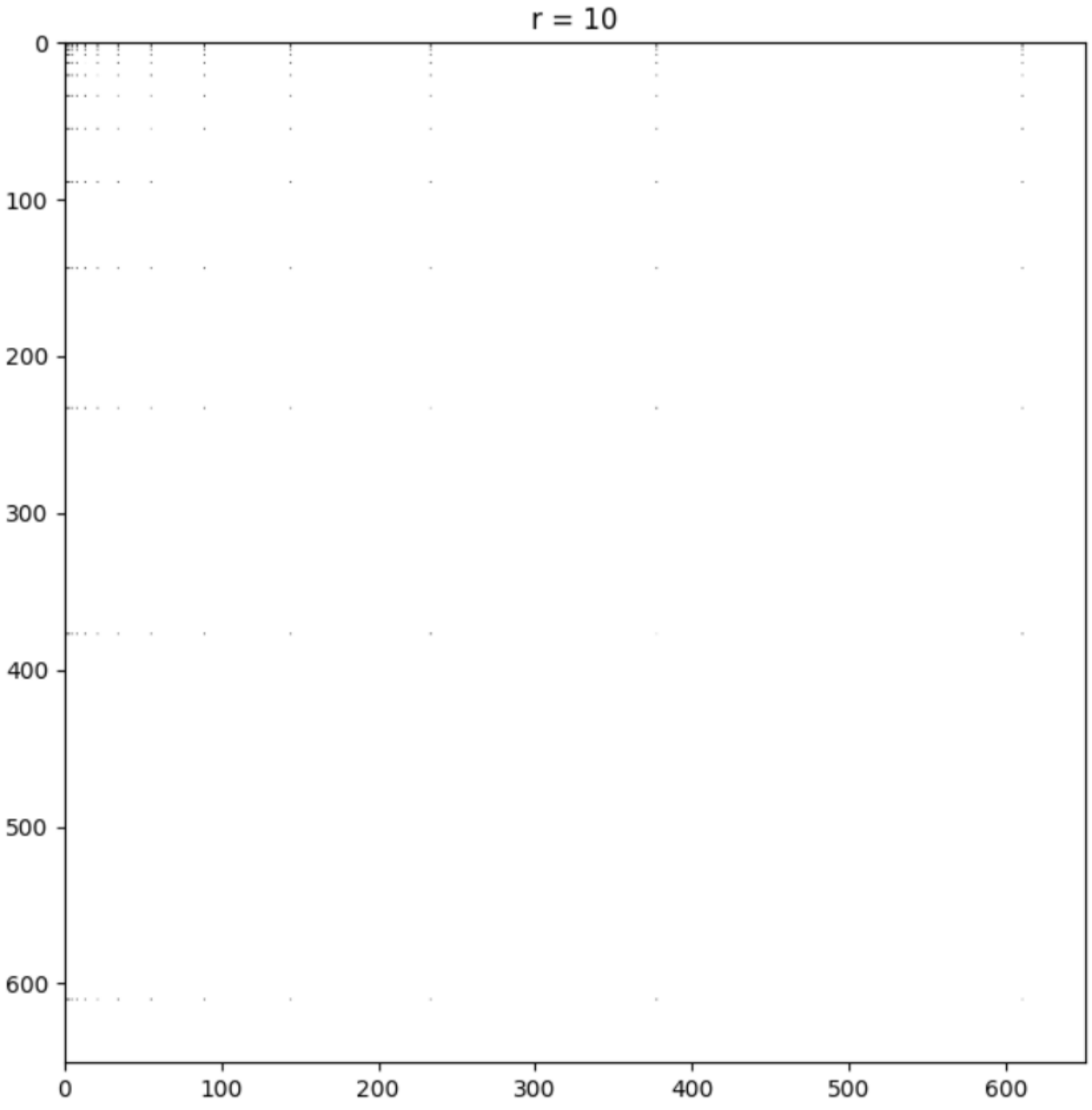


Figure 10.8: SVD at Rank 10

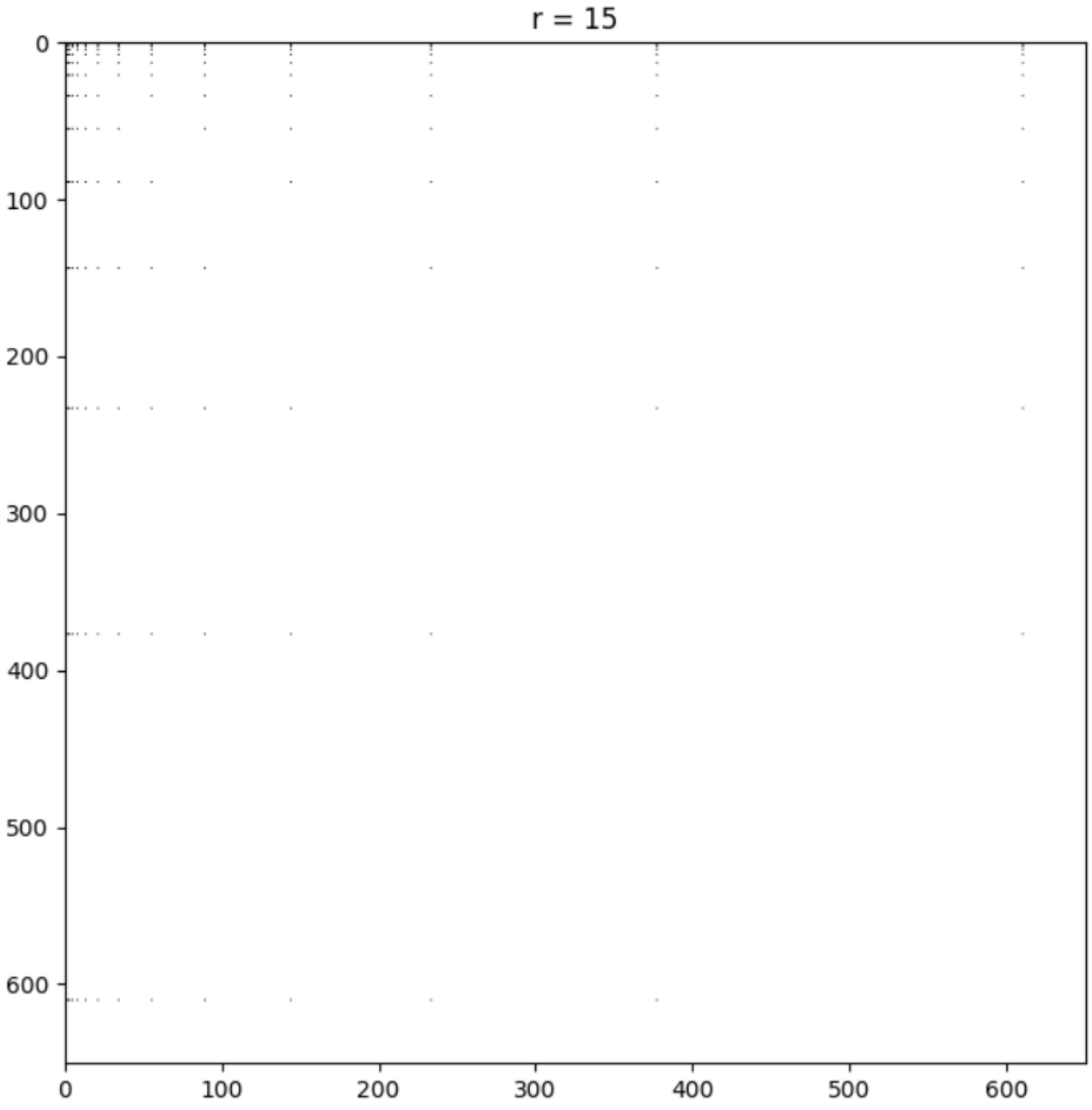
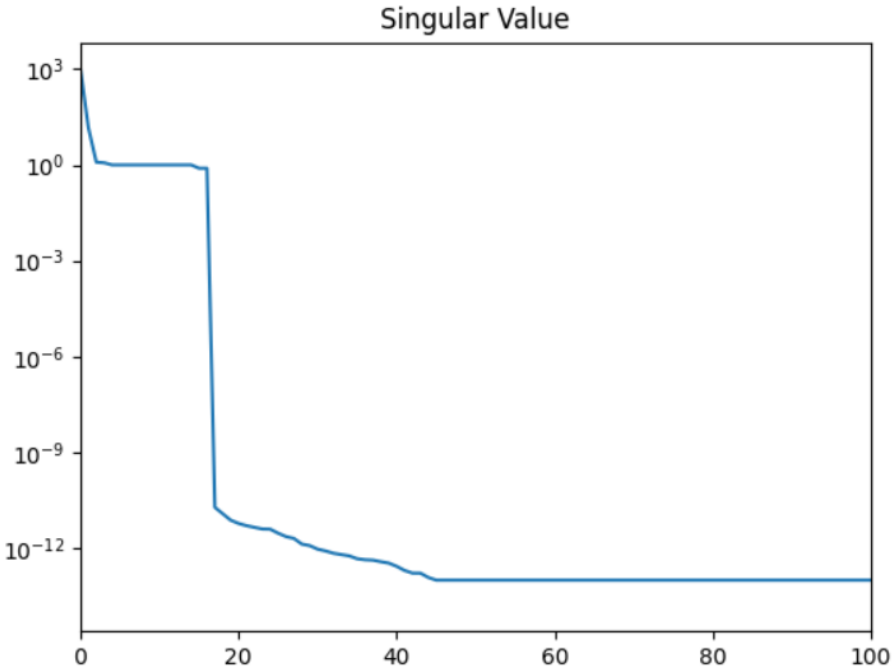
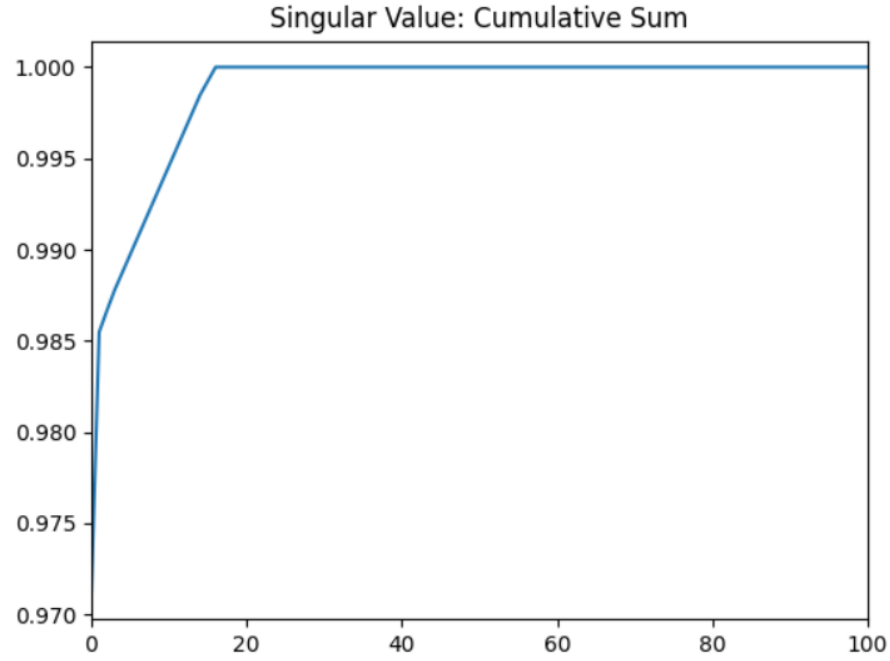


Figure 10.9: SVD at Rank 15



(a) $log(\sigma_j)$ vs j



(b) $\frac{(\sum_{j=1}^{r} \sigma_j)}{(\sum_{j=1}^{m} \sigma_j)}$ vs j

Figure 10.10: Graphs showing importance of Singular Values

10.2 The sequence $\{2^n + 1\}$

The sequence $3, 5, 9, 17, \ldots$, in binary form $11, 101, 1001, \ldots$. For each value of $0 \leq i \leq 9$, each value of $0 \leq j \leq 9$, the array position $(2^i+1, 2^j+1)$ gives the XOR value $(2^i + 1, 2^j + 1)$.

10.2.1 XOR Of the sequence $\{2^n + 1\}$

Input

```
import numpy as np
from PIL import Image, ImageEnhance
import matplotlib.pyplot as plt
from matplotlib.image import imread
import cv2
c = np.zeros([600, 600, 3], dtype=np.uint8)

def xor(x, y):
  return x ^ y

for i in range(0, 10):
  for j in range(0, 10):
    c[2 ** i + 1][2 ** j + 1] = xor(2 ** i + 1, 2 ** j + 1)

im1 =Image.fromarray(c)
im2 = ImageEnhance.Contrast(im1)
im2.enhance(200).save(('XOR_2.png'))
im3 = cv2.imread("XOR_2.png")
inv = 255 - im3
cv2.imwrite("XOR2.png", inv)
cv2.imshow("Inverted Image", inv)
cv2.waitKey(1000)
cv2.destroyWindow("Inverted Image")
```

Listing 10.5: XOR of sequence $\{2^n + 1\}$

Output

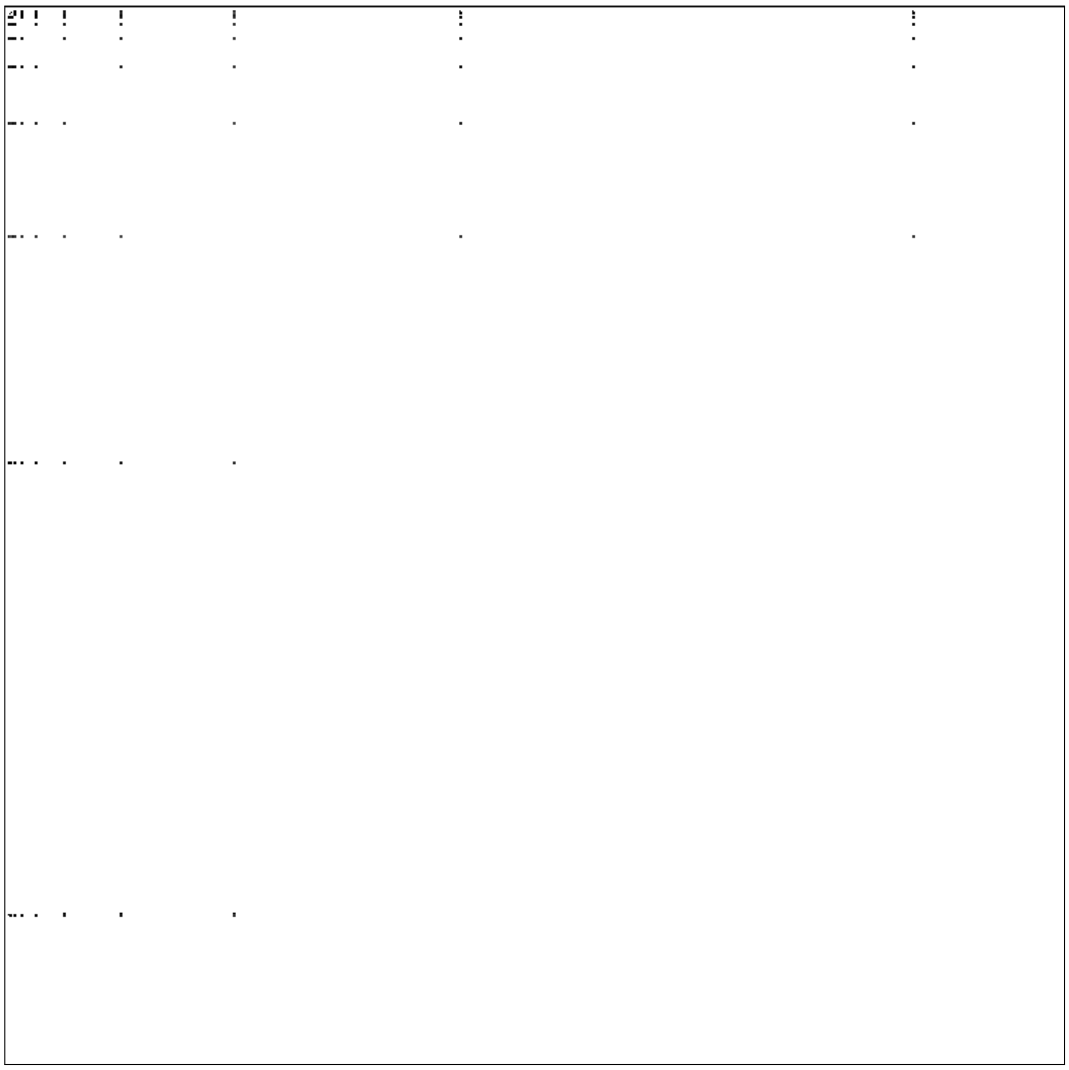


Figure 10.11: Image formed by XOR of $2^n + 1$

10.2.1.1 SVD of XOR of the sequence $\{2^n + 1\}$

Input

```
A = imread('XOR2.png')
X = np.mean(A, -1);

img = plt.imshow(X)
img.set_cmap('gray')
plt.axis('off')
plt.show()

U, S, VT = np.linalg.svd(X, full_matrices=0)
S = np.diag(S)

j = 0
for r in (2, 4, 6):
  Xapprox = U[:, :r] @ S[0:r, :r] @ VT[:r,:]
  plt.figure(j+1)
  j += 1
  img = plt.imshow(Xapprox)
  plt.xlim(0, 300)
  plt.ylim(300, 0)
  img.set_cmap('gray')
  plt.title('r = ' + str(r))
  plt.show()


plt.figure(1)
plt.xlim(0, 100)
plt.semilogy(np.diag(S))
plt.title('Singular Value')
plt.show()

plt.figure(2)
plt.xlim(0, 100)
plt.plot(np.cumsum(np.diag(S))/np.sum(np.diag(S)))
plt.title('Singular Value: Cumulative Sum')
plt.show()
```

Listing 10.6: SVD of XOR of sequence $\{2^n + 1\}$

Output

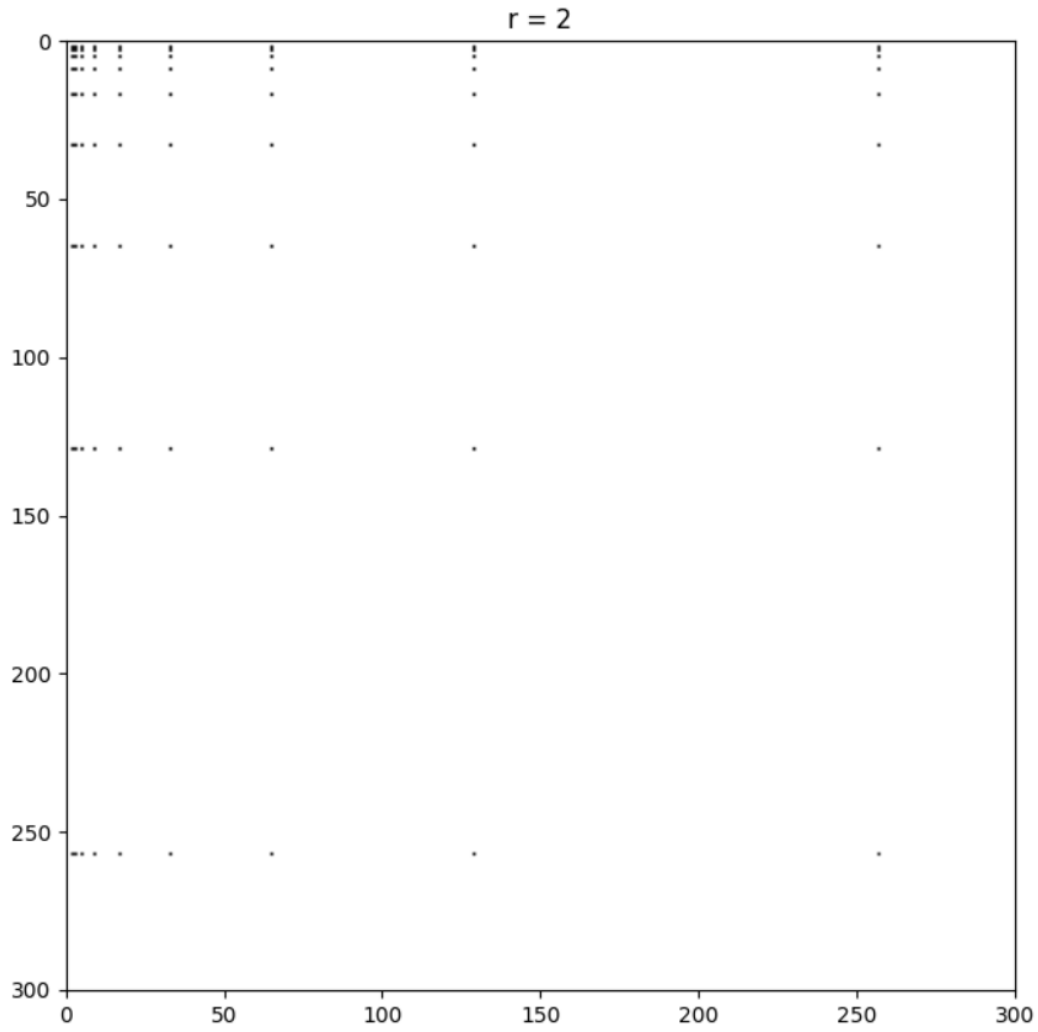


Figure 10.12: SVD at rank 2

.

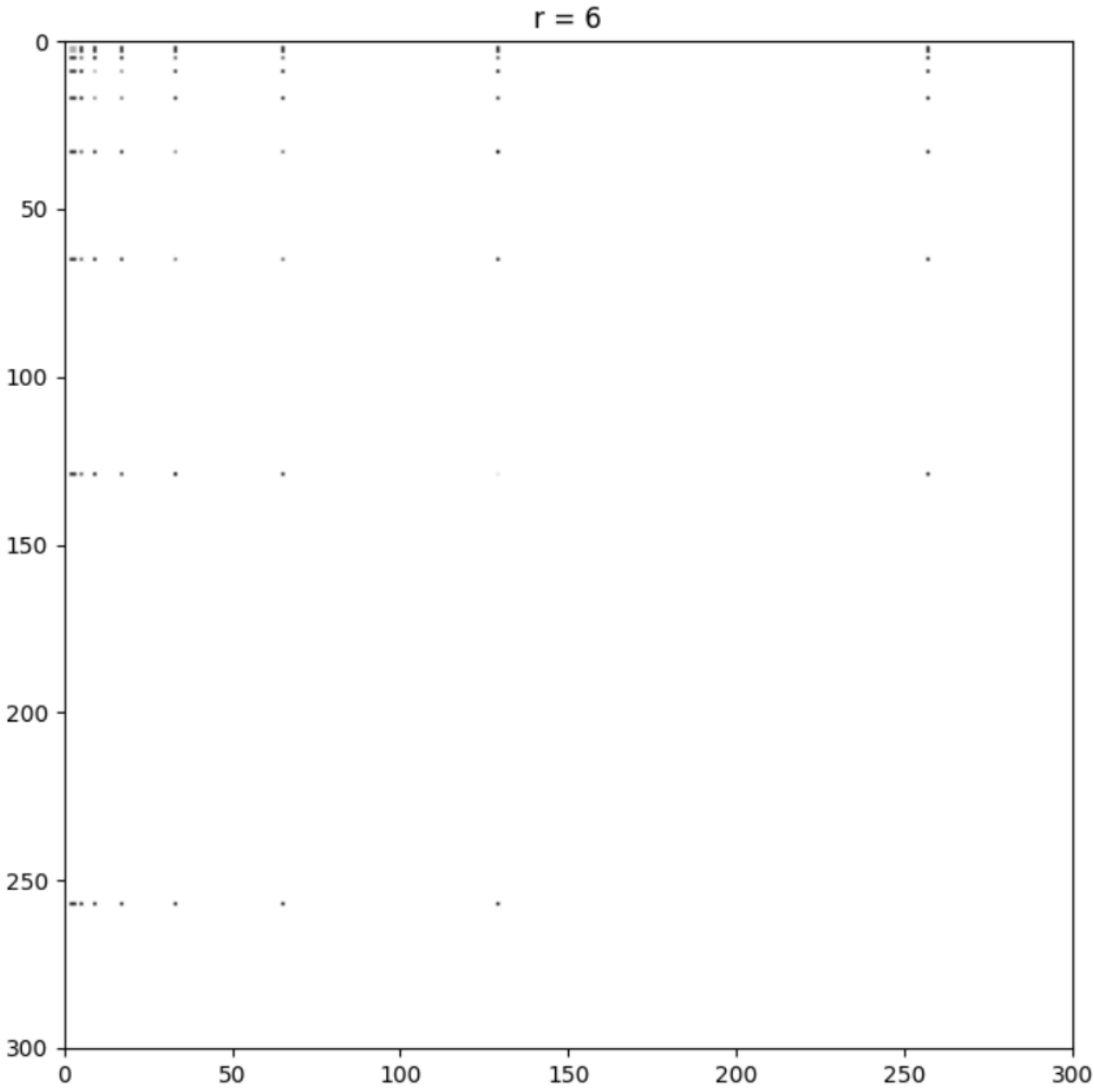


Figure 10.13: SVD at rank6

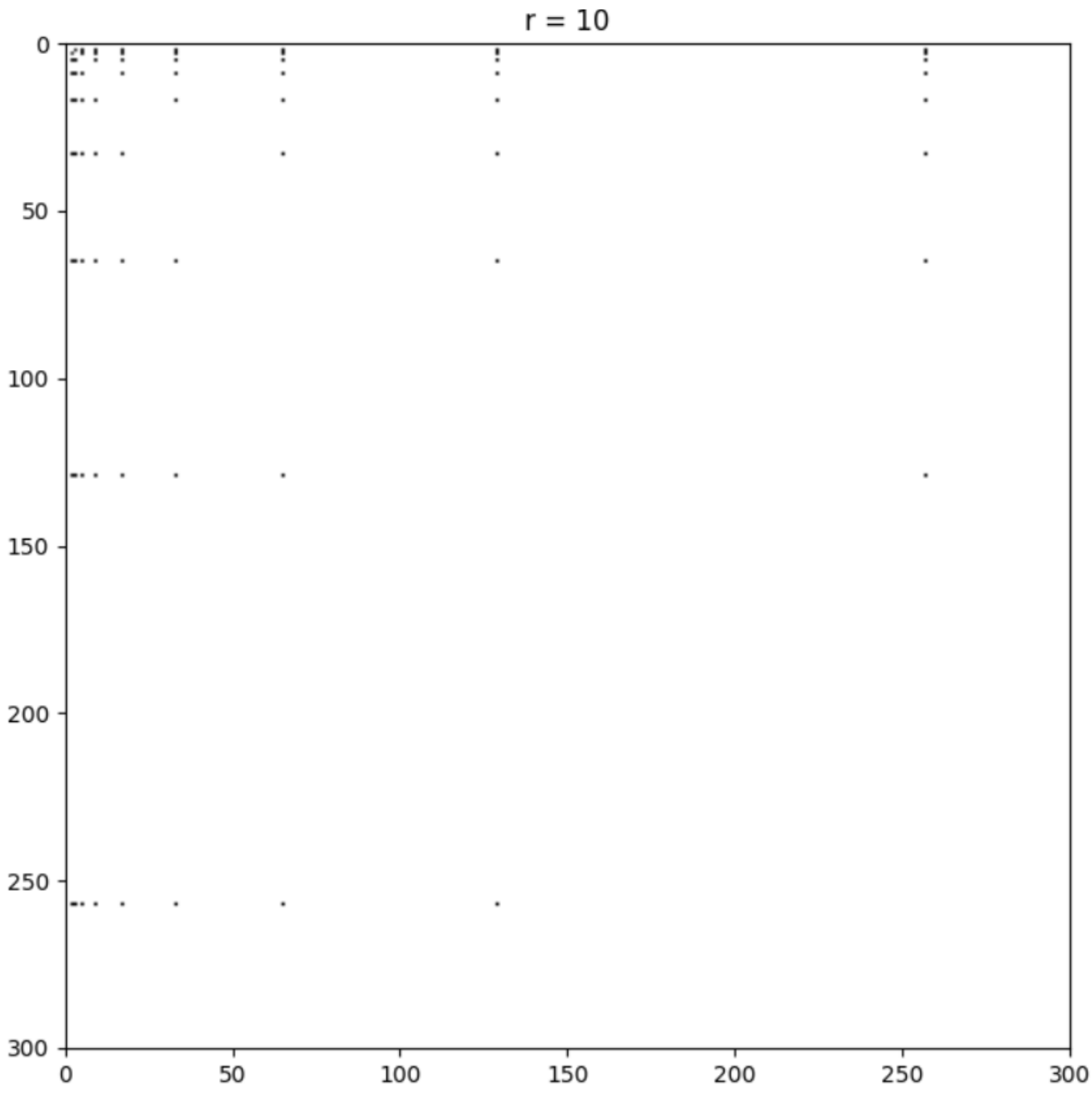
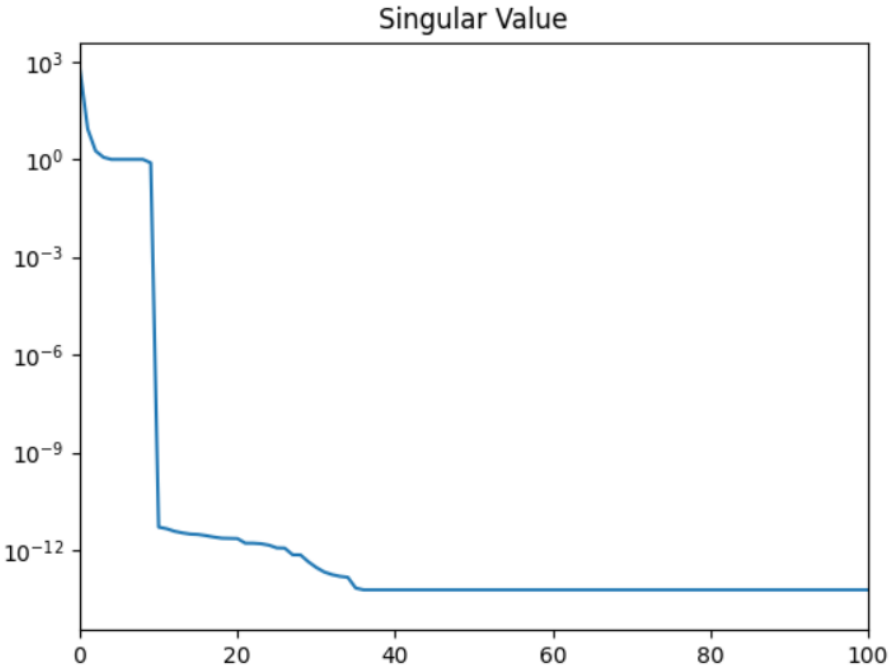
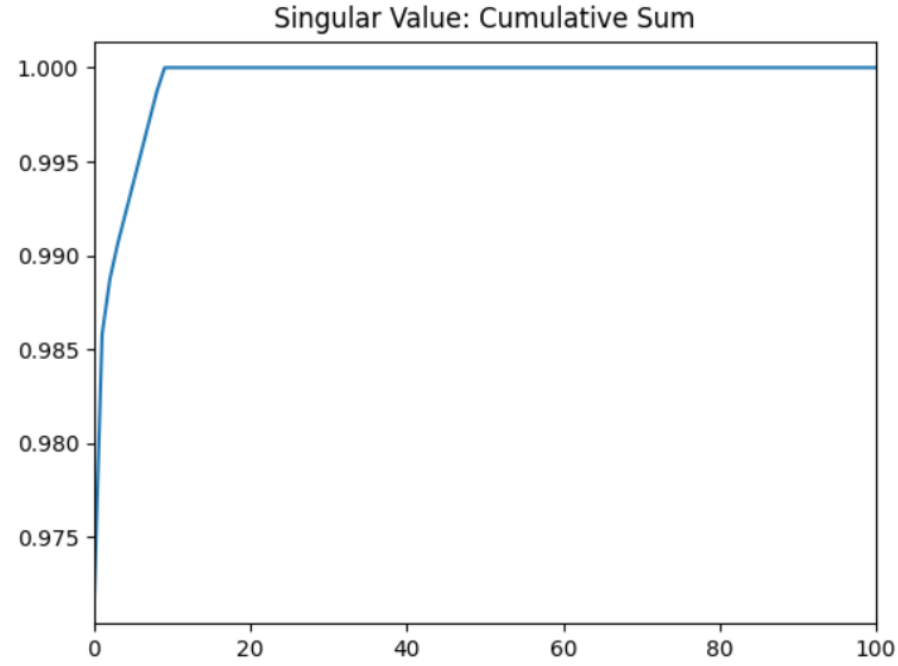


Figure 10.14: SVD at rank 10



(a) $log(\sigma_j)$ vs j



(b) $\frac{(\sum_{j=1}^{r} \sigma_j)}{(\sum_{j=1}^{m} \sigma_j)}$ vs j

Figure 10.15: Graphs showing importance of Singular Values

10.2.2 CVT Of the sequence $\{2^n + 1\}$

Input

```
from matplotlib.image import imread
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image, ImageEnhance
import cv2

c = np.zeros([600, 600, 3], dtype=np.uint8)


def cvt(x, y):
  return (x & y) << 1

for i in range(0, 10):
  for j in range(0, 10):
    c[2**i+1][2**j+1]=cvt(2**i+1,2**j+1)

im1 =Image.fromarray(c)
im2 = ImageEnhance.Contrast(im1)
im2.enhance(200).save(('CVT_2.png'))
im3 = cv2.imread("CVT_2.png")
inv = 255 - im3
cv2.imwrite("CVT2", inv)
cv2.imshow("Inverted Image", inv)
cv2.waitKey(1000)
cv2.destroyWindow("Inverted Image")
```

Listing 10.7: CVT of sequence $\{2^n + 1\}$

Output

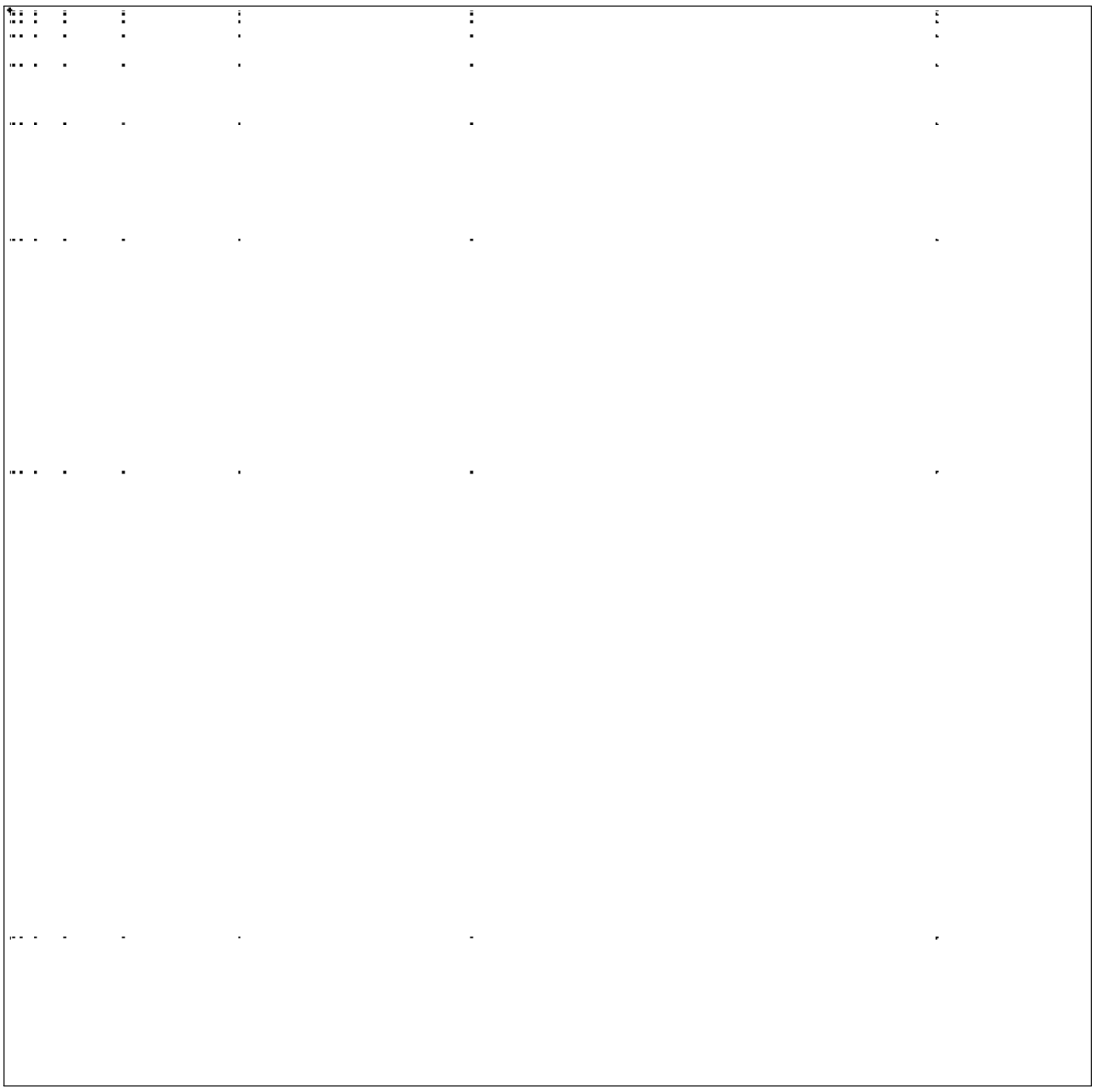


Figure 10.16: Image of CVT of $(2^i + 1, 2^j + 1)$

10.2.2.1 SVD of CVT of the sequence $\{2^n + 1\}$

Input

```
A = imread('CVT2.png')
X = np.mean(A, -1);

img = plt.imshow(X)
img.set_cmap('gray')
plt.axis('off')
plt.show()

U, S, VT = np.linalg.svd(X, full_matrices=0)
S = np.diag(S)

j = 0
for r in (2, 4, 6):
  Xapprox = U[:, :r] @ S[0:r, :r] @ VT[:r,:]
  plt.figure(j+1)
  j += 1
  img = plt.imshow(Xapprox)
  plt.xlim(0, 300)
  plt.ylim(300, 0)
  img.set_cmap('gray')
  plt.title('r = ' + str(r))
  plt.show()


plt.figure(1)
plt.xlim(0, 100)
plt.semilogy(np.diag(S))
plt.title('Singular Value')
plt.show()

plt.figure(2)
plt.xlim(0, 100)
plt.plot(np.cumsum(np.diag(S))/np.sum(np.diag(S)))
plt.title('Singular Value: Cumulative Sum')
plt.show()
```

Listing 10.8: SVD of CVT of sequence $\{2^n + 1\}$

Output

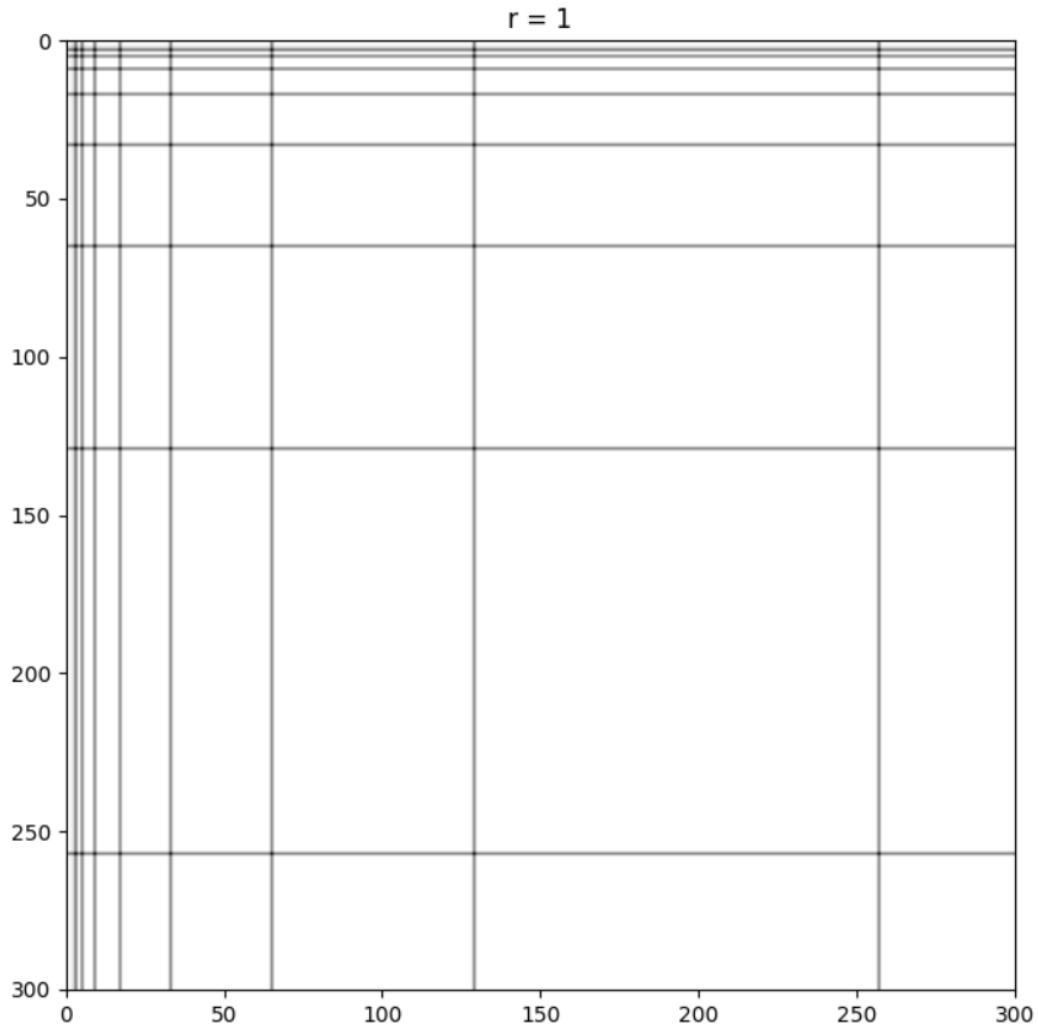


Figure 10.17: SVD at rank 1

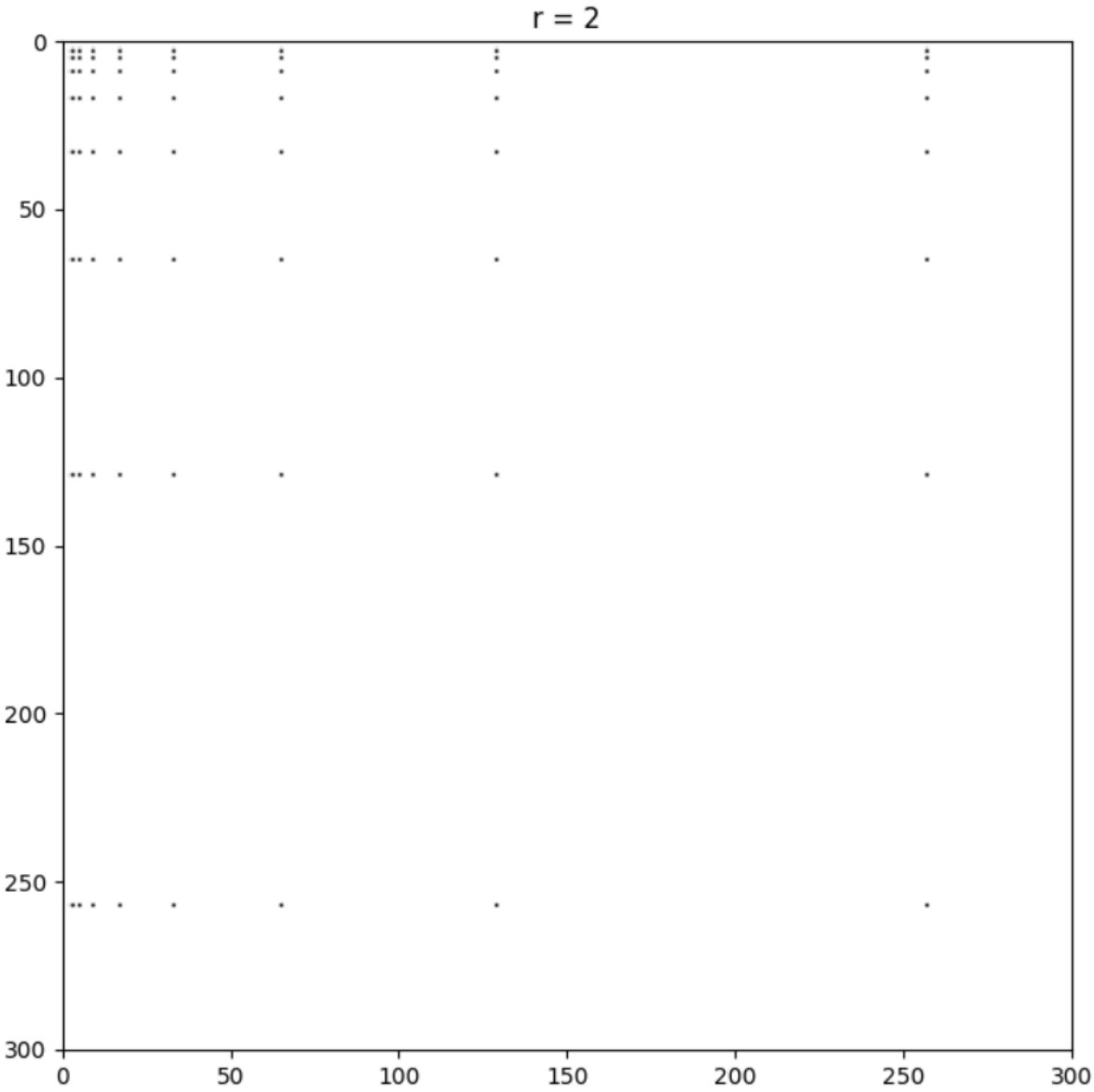


Figure 10.18: SVD at rank 2

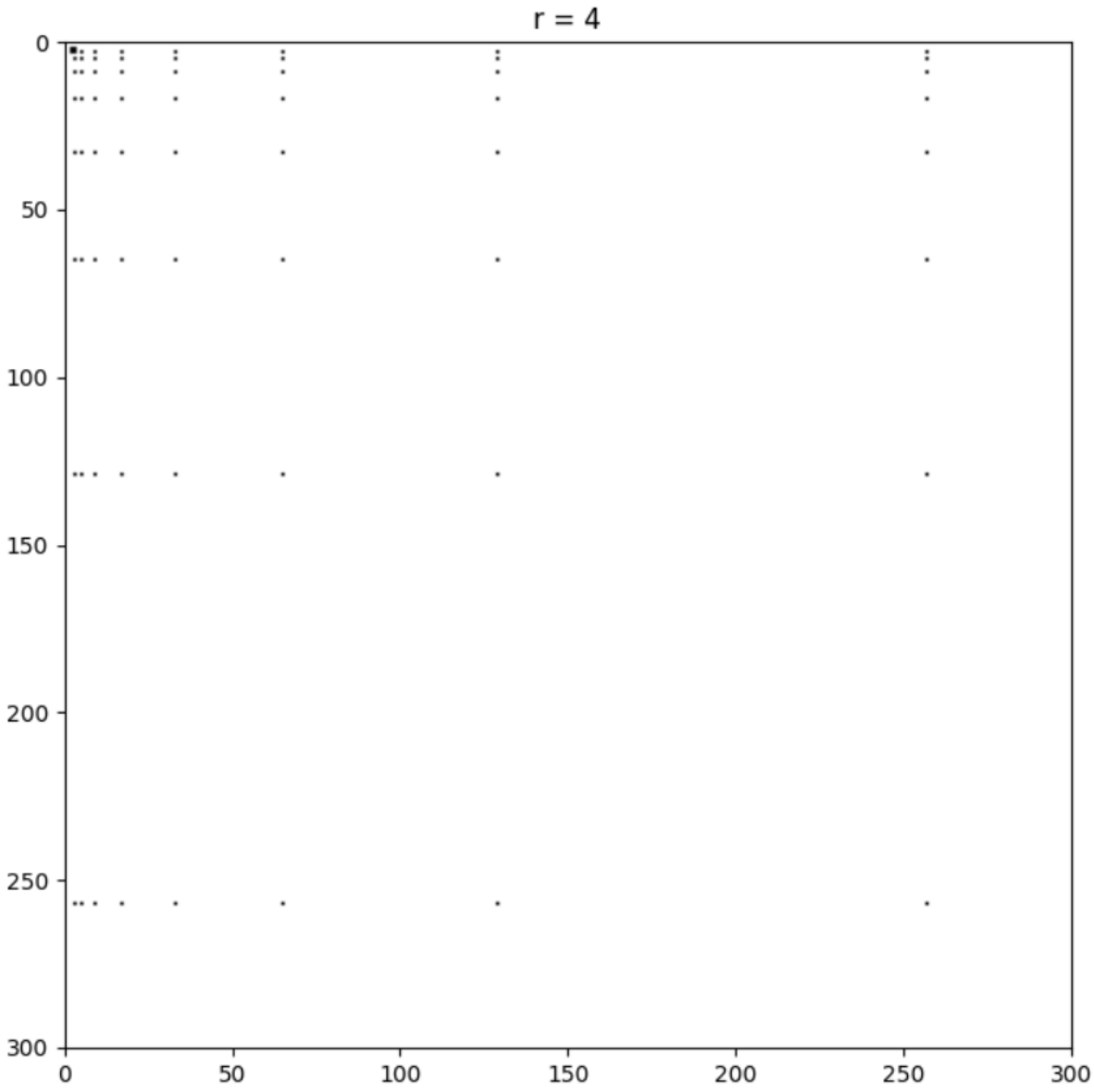
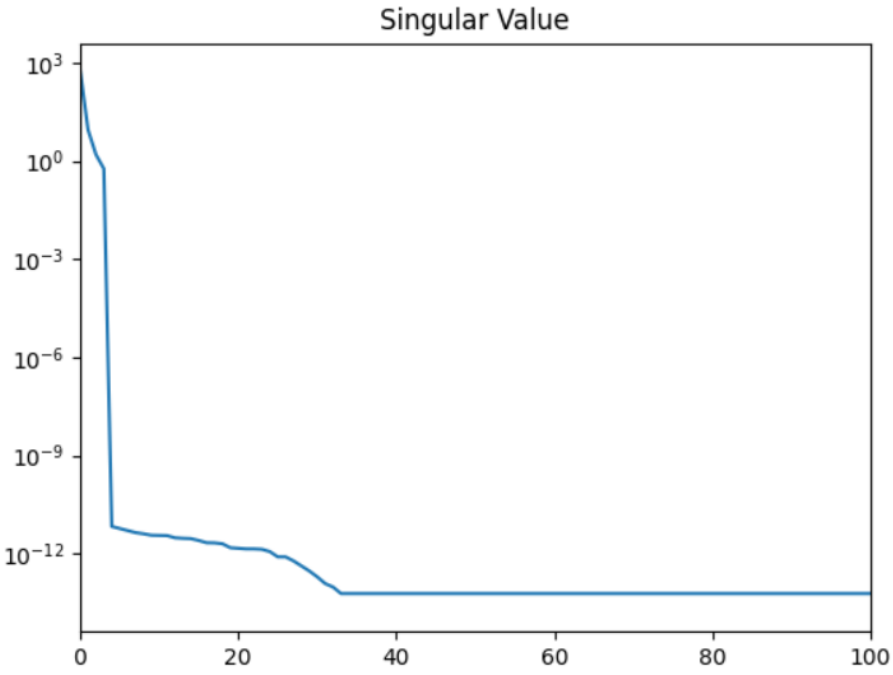
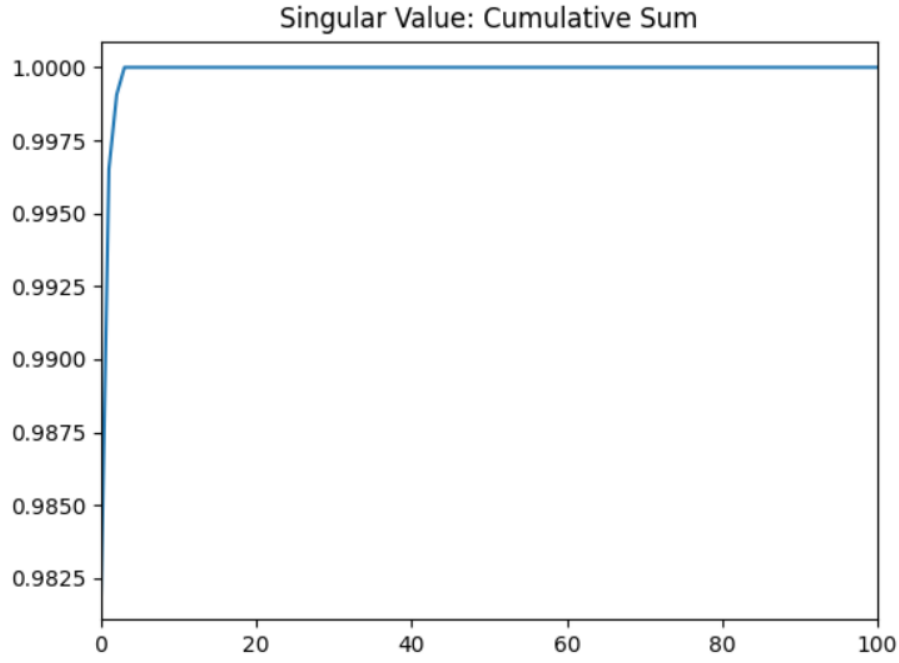


Figure 10.19: SVD at rank 4



(a) $log(\sigma_j)$ vs j



(b) $\frac{(\sum_{j=1}^{r} \sigma_j)}{(\sum_{j=1}^{m} \sigma_j)}$ vs j

Figure 10.20: Graphs showing importance of Singular Values

10.3 The sequence $\{2^n\}$

The sequence 2^n is $2, 4, 8, 16, \ldots$, in binary form $10, 100, 1000, \ldots$.

For each value of $0 \leq i \leq 9$, each value of $0 \leq j \leq 9$, the array position $(2^i, 2^j)$ gives the XOR value $(2^i, 2^j)$. And XOR value is defined as given below,

$$XOR(2^i, 2^j) = \begin{cases} 2^i + 2^j & \text{if } i \neq j \\ 0 & \text{if } i = j \end{cases}$$

10.3.1 XOR Of the sequence $\{2^n\}$

Input

```
from matplotlib.image import imread
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image, ImageEnhance
import cv2

c = np.zeros([600, 600, 3], dtype=np.uint8)

def xor(x, y):
  return x ^ y

for i in range(0, 10):
  for j in range(0, 10):
    c[2**i][2**j]=xor(2**i,2**j)

im1 =Image.fromarray(c)
im2 = ImageEnhance.Contrast(im1)
im2.enhance(200).save(('XOR_1.png'))
im3 = cv2.imread("XOR_1.png")
inv = 255 - im3
cv2.imwrite("XOR1", inv)
cv2.imshow("Inverted Image", inv)
cv2.waitKey(1000)
cv2.destroyWindow("Inverted Image")
```

Listing 10.9: XOR of the sequdnce $\{2^n\}$

Output

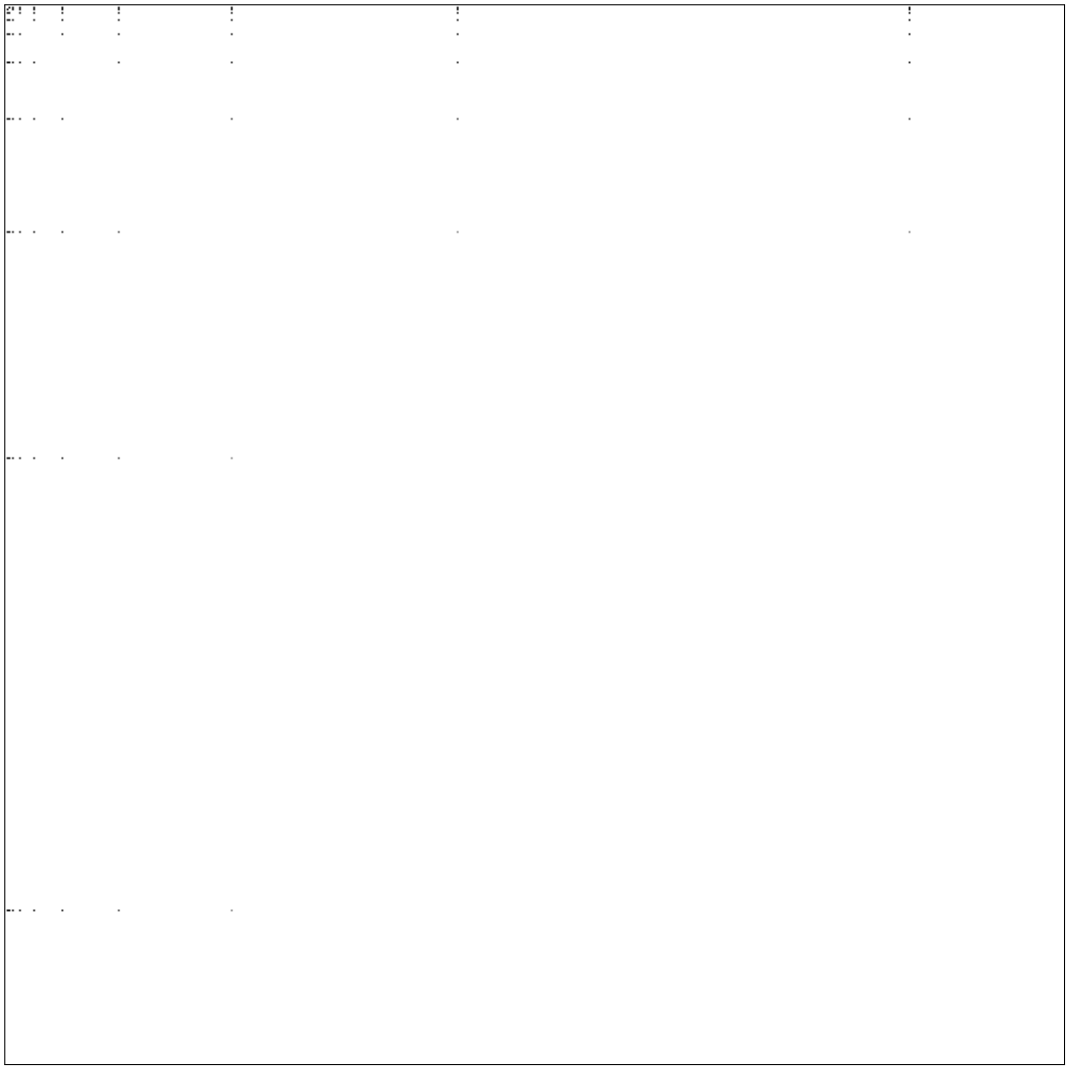


Figure 10.21: Image formed by XOR of $(2^i, 2^j)$

10.3.1.1 SVD of XOR of the sequence $\{2^n\}$

Input

```
A = imread('XOR1.png')
X = np.mean(A, -1);

img = plt.imshow(X)
img.set_cmap('gray')
plt.axis('off')
plt.show()

U, S, VT = np.linalg.svd(X, full_matrices=0)
S = np.diag(S)

j = 0
for r in (1, 3, 6):
  Xapprox = U[:, :r] @ S[0:r, :r] @ VT[:r,:]
  plt.figure(j+1)
  j += 1
  img = plt.imshow(Xapprox)
  plt.xlim(0, 280)
  plt.ylim(280, 0)
  img.set_cmap('gray')
  plt.title('r = ' + str(r))
  plt.show()

plt.figure(1)
plt.xlim(0, 100)
plt.semilogy(np.diag(S))
plt.title('Singular Value')
plt.show()

plt.figure(2)
plt.xlim(0, 100)
plt.plot(np.cumsum(np.diag(S))/np.sum(np.diag(S)))
plt.title('Singular Value: Cumulative Sum')
plt.show()
```

Listing 10.10: SVD of XOR of $\{2^n\}$

Output

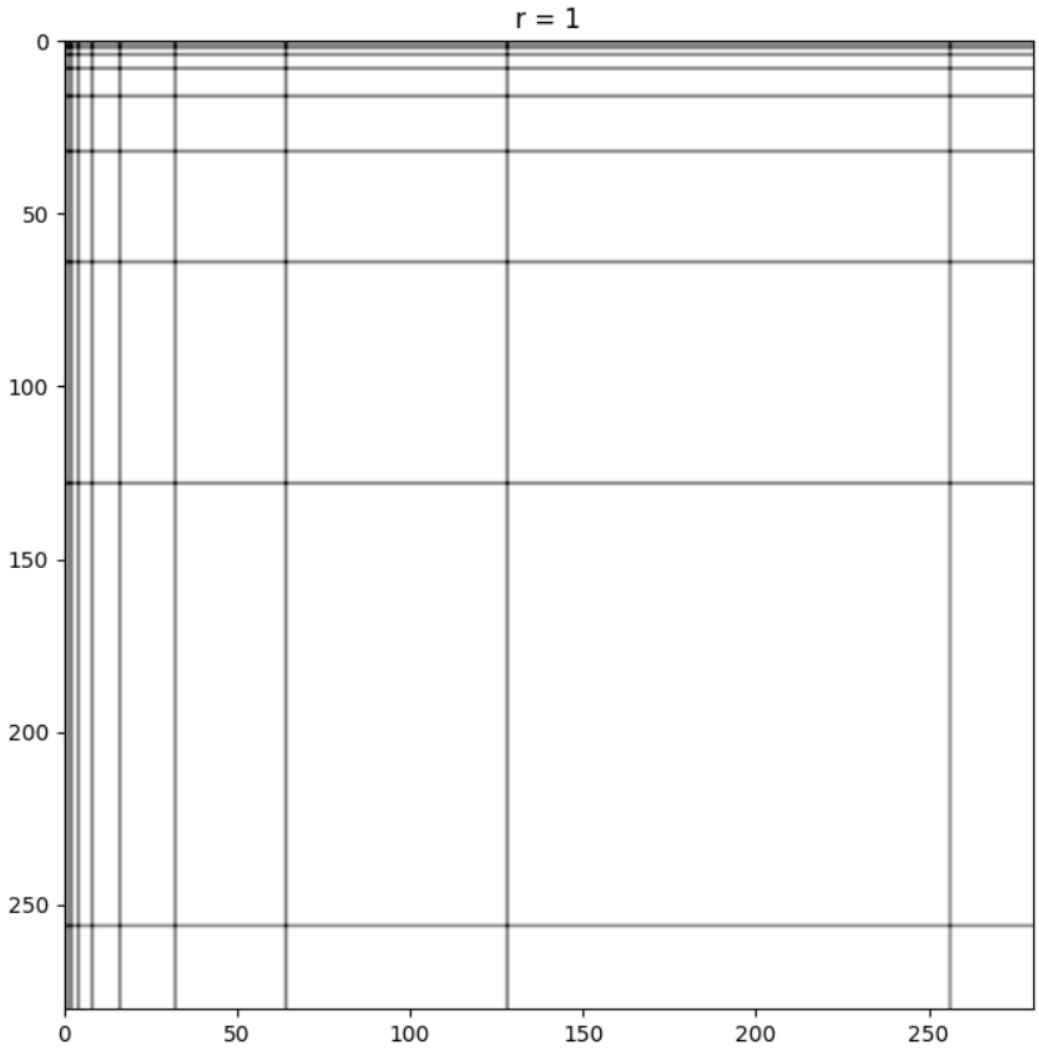


Figure 10.22: SVD at rank 1

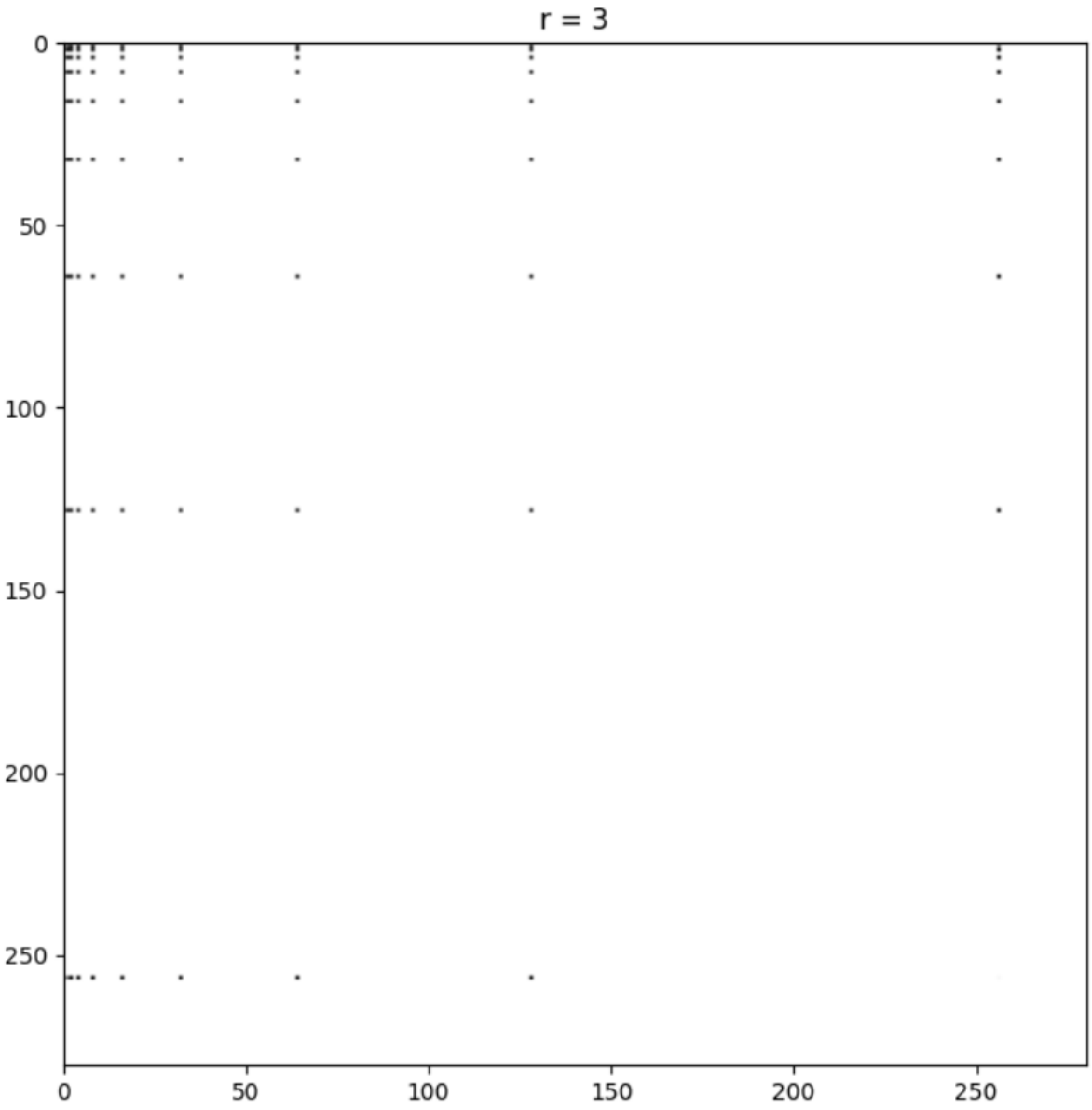


Figure 10.23: SVD at rank 7

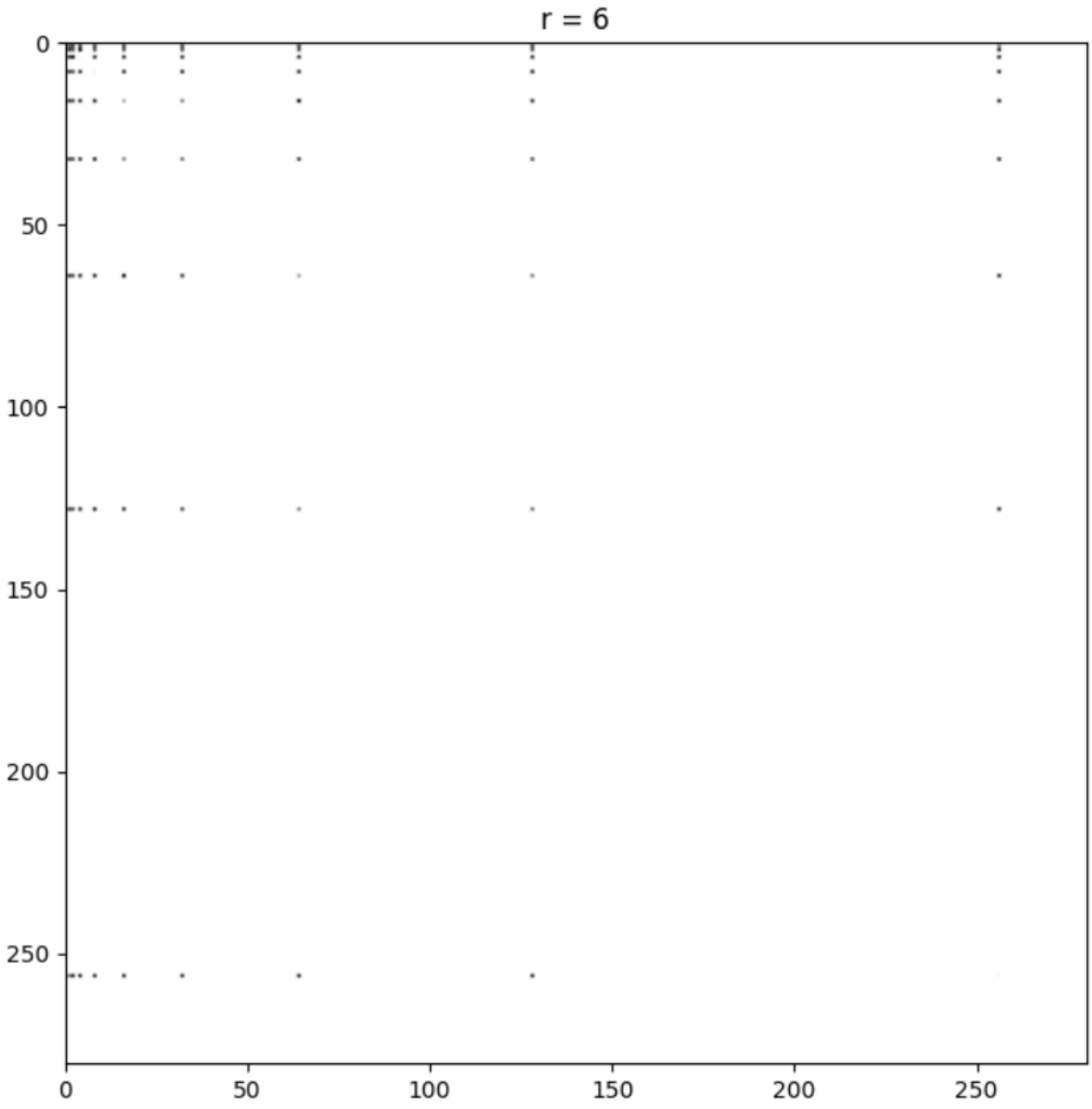
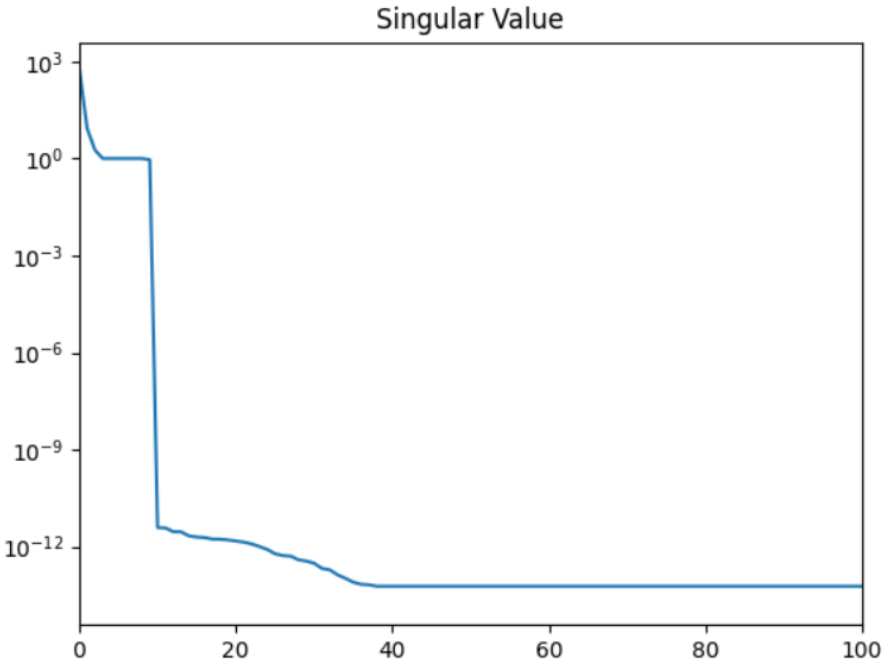
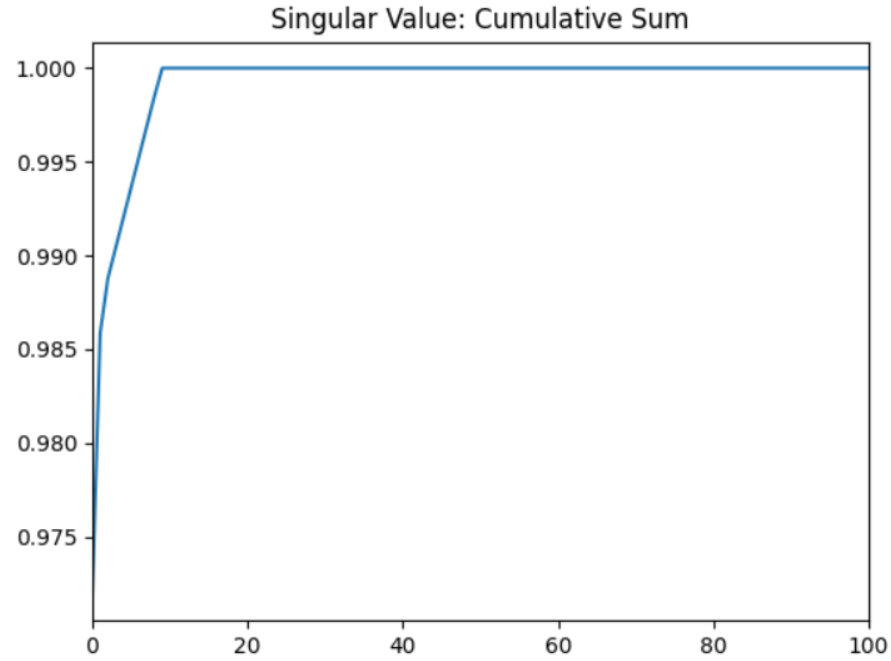


Figure 10.24: SVD at rank 12



(a) $log(\sigma_j)$ vs j



(b) $\frac{(\sum_{j=1}^{r} \sigma_j)}{(\sum_{j=1}^{m} \sigma_j)}$ vs j

Figure 10.25: Graphs showing importance of Singular Values

10.3.2 CVT Of the sequence $\{2^n\}$

Input

```
from matplotlib.image import imread
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image, ImageEnhance
import cv2

c = np.zeros([600, 600, 3], dtype=np.uint8)

def cvt(x, y):
  return (x & y) << 1

for i in range(0, 10):
  for j in range(0, 10):
    c[2**i][2**j]=cvt(2**i,2**j)

im1 =Image.fromarray(c)
im2 = ImageEnhance.Contrast(im1)
im2.enhance(200).save(('CVT_1.png'))
im3 = cv2.imread("CVT_1.png")
inv = 255 - im3
cv2.imwrite("CVT1", inv)
cv2.imshow("Inverted Image", inv)
cv2.waitKey(1000)
cv2.destroyWindow("Inverted Image")
```

Listing 10.11: CVT of $\{2^n\}$

Output

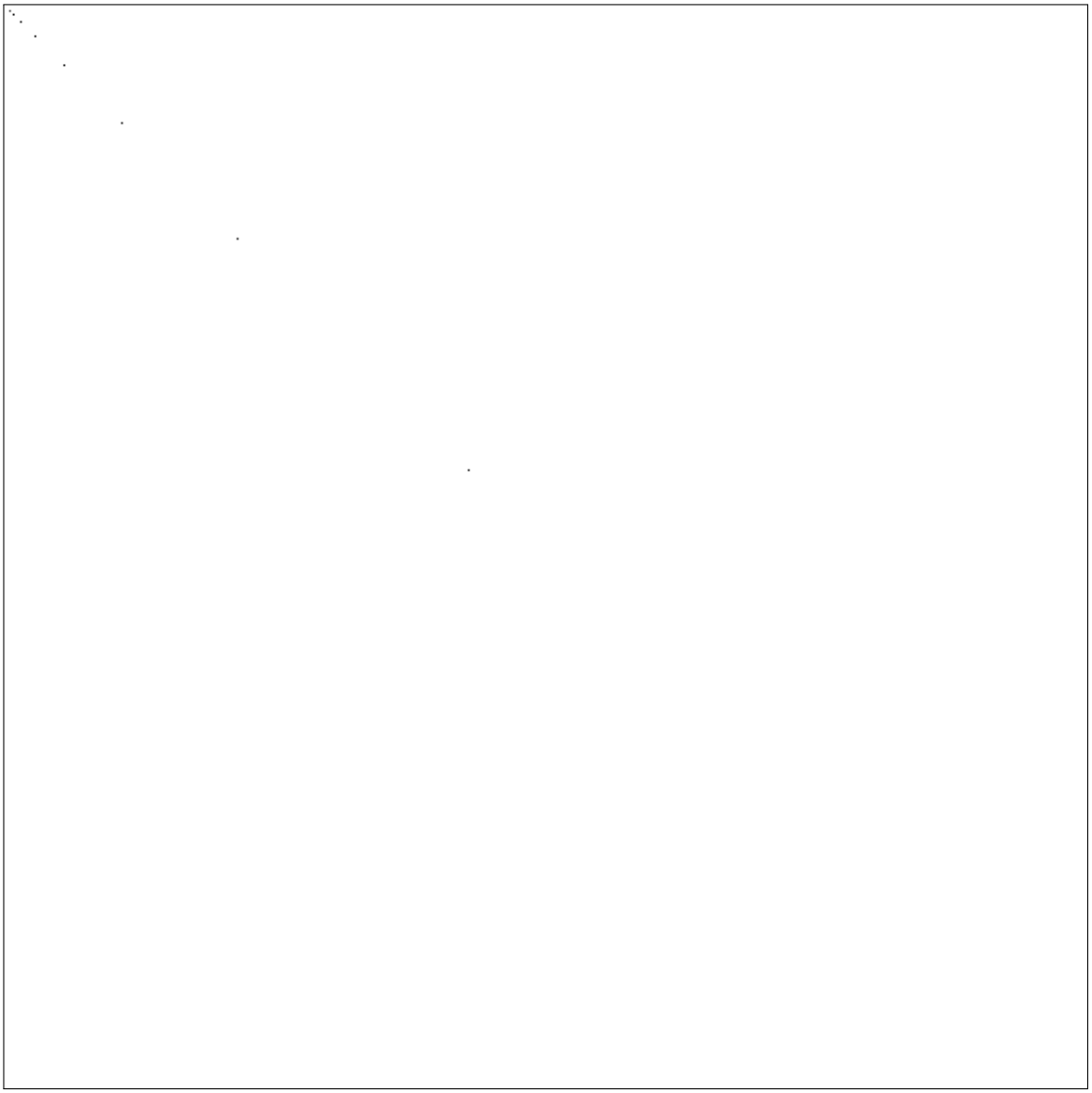


Figure 10.26: Image of CVT of $(2^i, 2^j)$

10.3.2.1 SVD of CVT of $(2^i, 2^j)$

Input

```
A = imread('CVT1.png')
X = np.mean(A, -1);

img = plt.imshow(X)
img.set_cmap('gray')
plt.axis('off')
plt.show()

U, S, VT = np.linalg.svd(X, full_matrices=0)
S = np.diag(S)

j = 0
for r in (1, 3, 6):
  Xapprox = U[:, :r] @ S[0:r, :r] @ VT[:r,:]
  plt.figure(j+1)
  j += 1
  img = plt.imshow(Xapprox)
  plt.xlim(0, 280)
  plt.ylim(280, 0)
  img.set_cmap('gray')
  plt.title('r = ' + str(r))
  plt.show()


plt.figure(1)
plt.xlim(0, 100)
plt.semilogy(np.diag(S))
plt.title('Singular Value')
plt.show()

plt.figure(2)
plt.xlim(0, 100)
plt.plot(np.cumsum(np.diag(S))/np.sum(np.diag(S)))
plt.title('Singular Value: Cumulative Sum')
plt.show()
```

Listing 10.12: SVD Of CVT of 2^i

Output

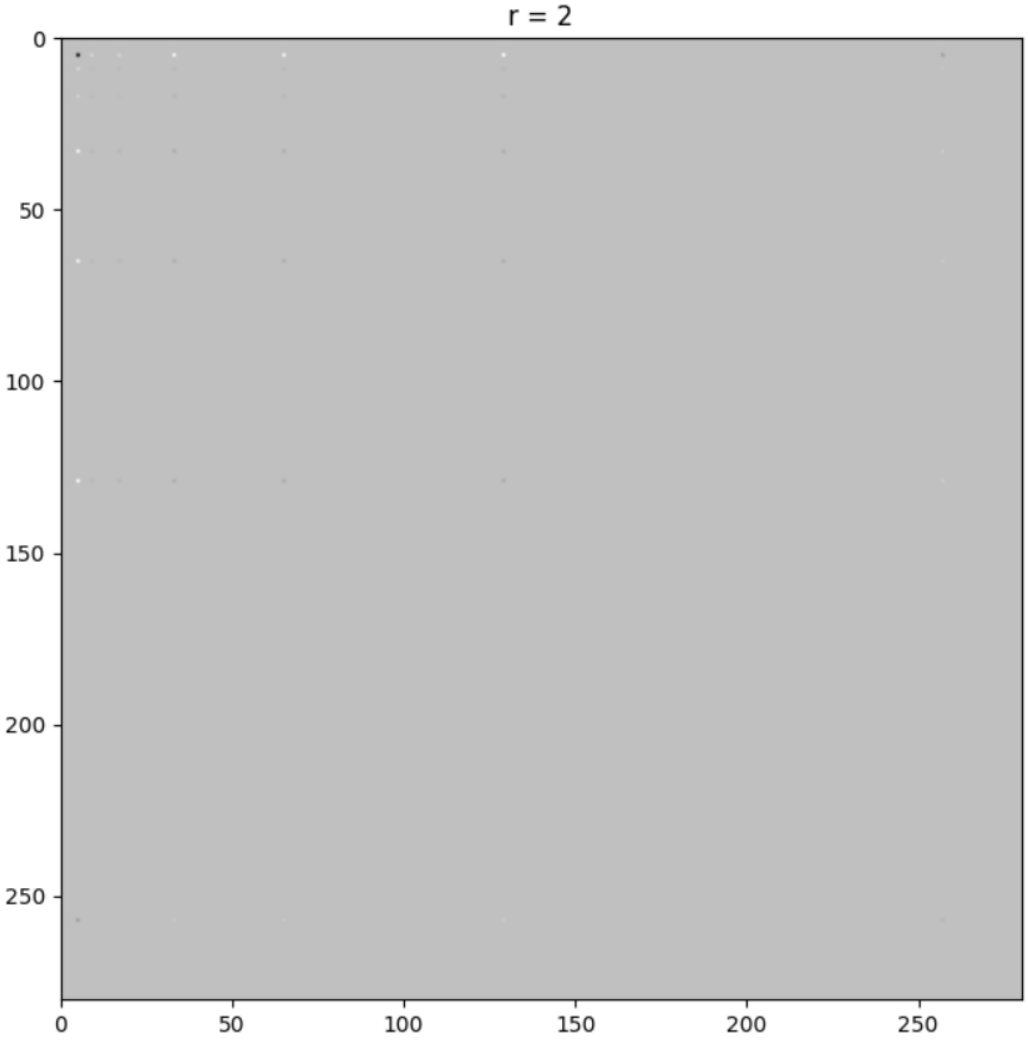


Figure 10.27: SVD at rank 2

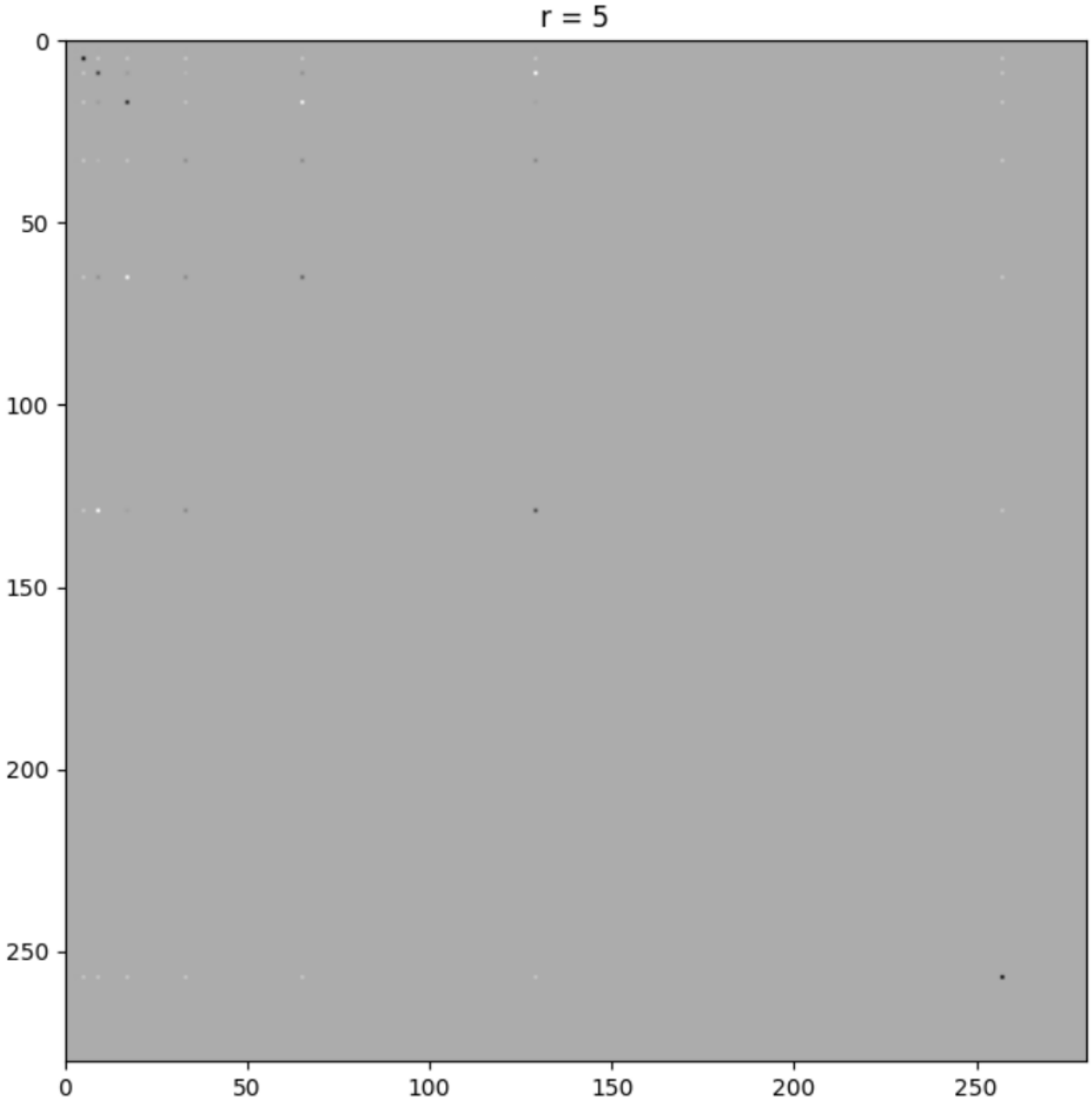


Figure 10.28: SVD at rank

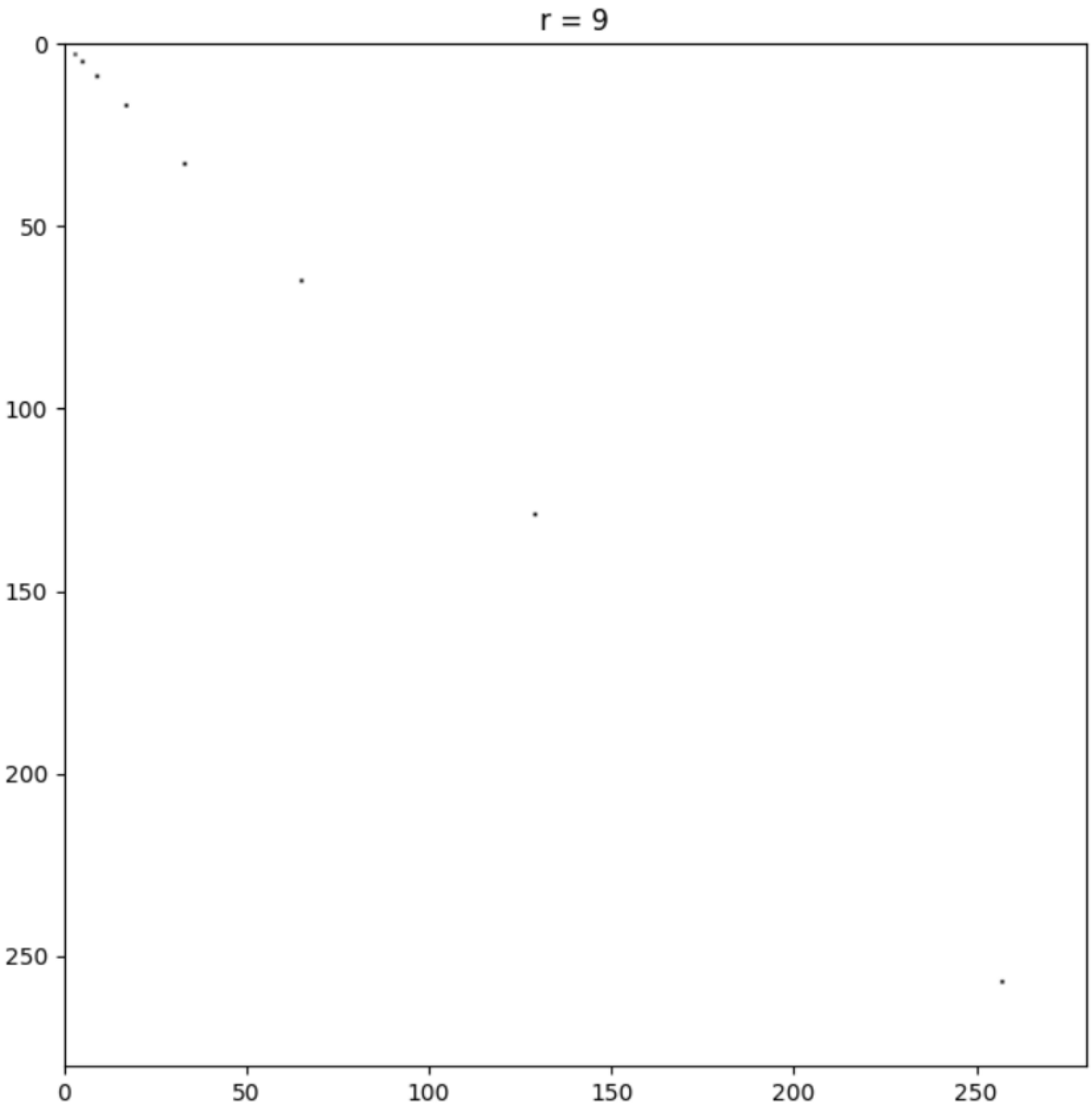
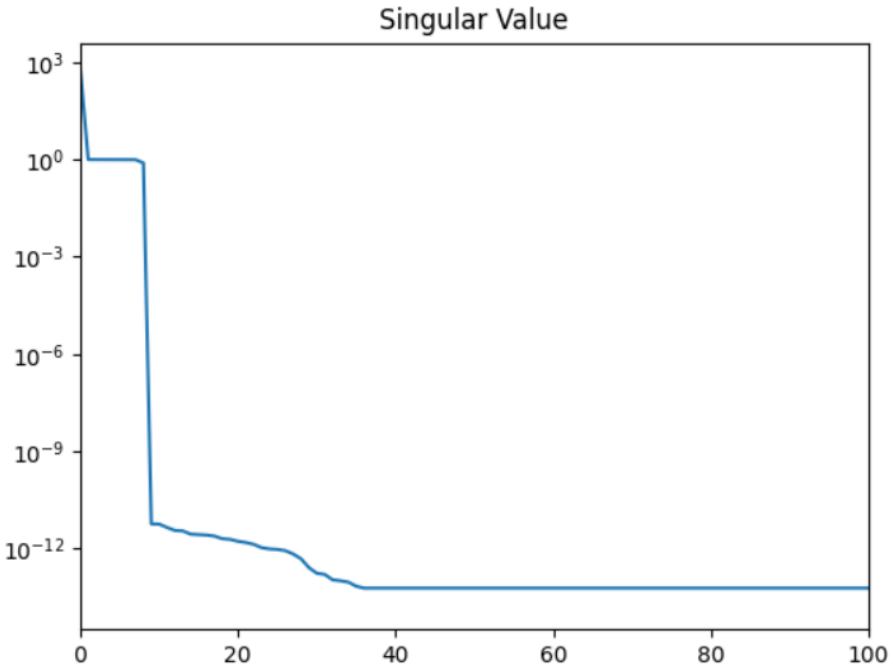
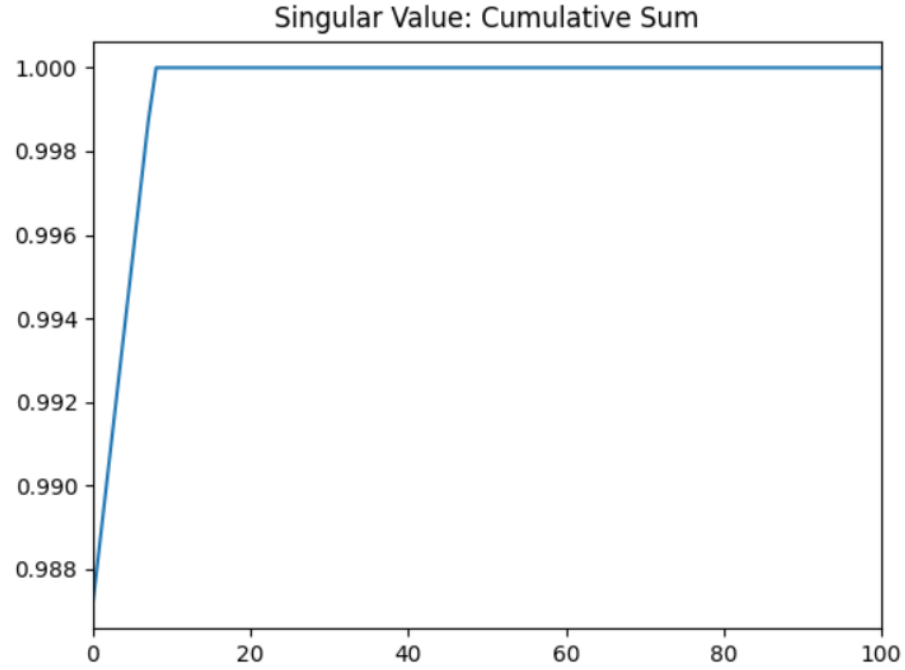


Figure 10.29: SVD at rank 9



(a) $log(\sigma_j)$ vs j



(b) $\frac{(\sum_{j=1}^{r} \sigma_j)}{(\sum_{j=1}^{m} \sigma_j)}$ vs j

Figure 10.30: Graphs showing importance of Singular Values

Chapter 11

Additional Work

11.1 Splitting the Image

Here, we take an image and we create three different images by converting the r, g and b values to 0 in respective images [9]. These three images are then read as array and we operate CVT and XOR on these matrices taken 2 at a time. We then try to obtain the original image using them.

Input

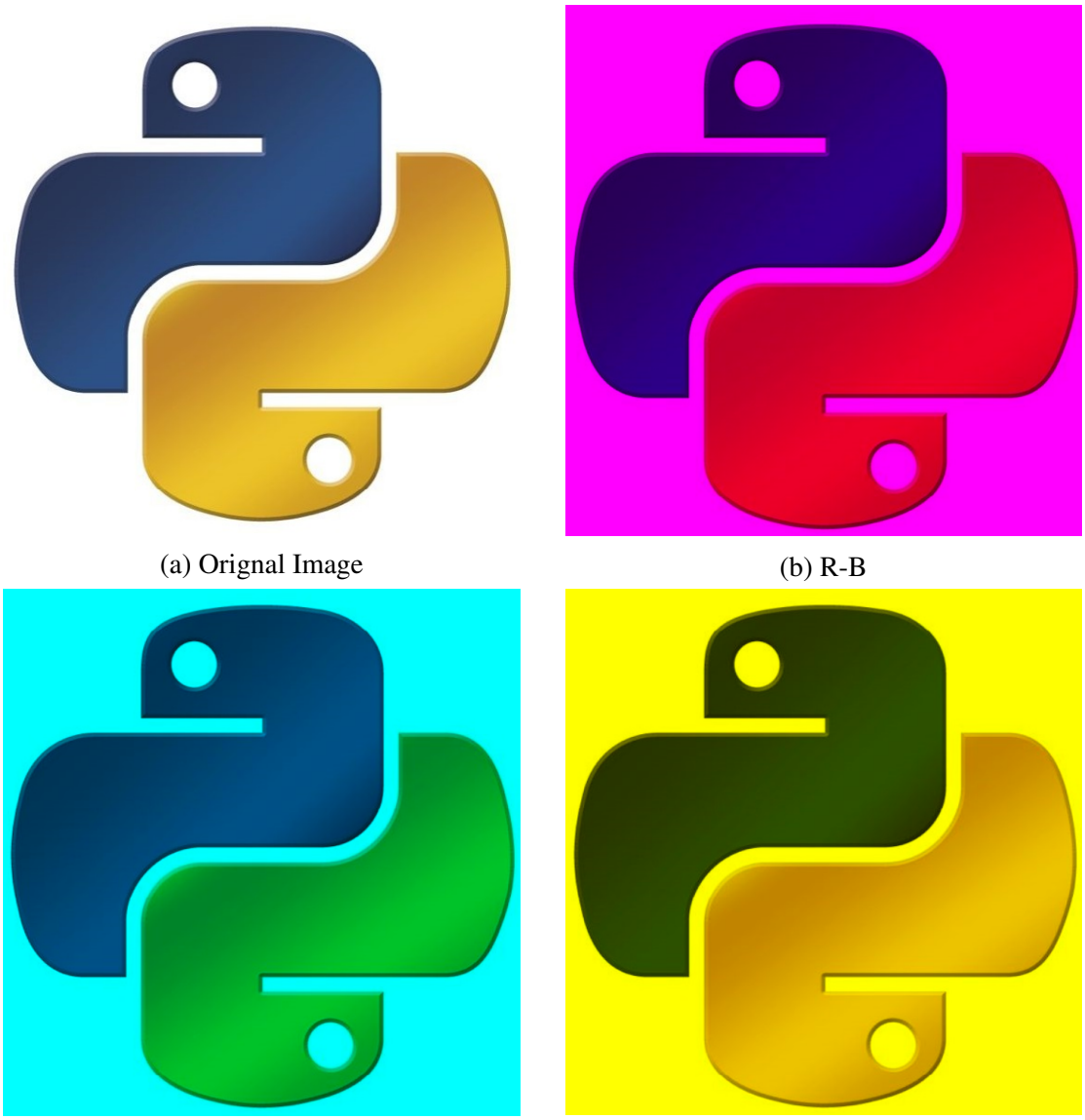
```
from matplotlib.image import imread
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image, ImageEnhance

img = Image.open('D:\Documents\python.jpg')
img1 = img.copy()
img2 = img.copy()
img3 = img.copy()

img1_data = img1.load()
img2_data = img2.load()
img3_data = img3.load()

height,width = img1.size
height,width = img2.size
height,width = img3.size

for loop1 in range(height):
    for loop2 in range(width):
```

```
21          r,g,b = img1_data[loop1,loop2]
22          img1_data[loop1,loop2] = 0,g,b
23
24 img1.show()
25 GB = np.asarray(img1)
26
27 for loop1 in range(height):
28     for loop2 in range(width):
29         r,g,b = img2_data[loop1,loop2]
30         img2_data[loop1,loop2] = r,0,b
31
32 img2.show()
33 RB = np.asarray(img2)
34
35 for loop1 in range(height):
36     for loop2 in range(width):
37         r,g,b = img3_data[loop1,loop2]
38         img3_data[loop1,loop2] = r,g,0
39
40 img3.show()
41 RG = np.asarray(img3)
```

Listing 11.1: Splitting the Image

Output

We see in the output images [Figure 11.1b] has only red and blue component, [Figure 11.1c] has only green and blue component and [Figure 11.1d] has only red and green component, whereas in [Figure 11.1a] we see the image with all the three components.



(a) Orignal Image

(b) R-B

(c) G-B

(d) R-G

Figure 11.1: Output Images

11.2 Applying CVT and XOR

Now, we operate CVT and XOR on the obtained matrices and observe the output image. First we define the CVT and XOR function. Then we operate CVT and XOR on the matrices obtained.

Input

```
import numpy as np
from PIL import Image, ImageEnhance
from numpy import array

img = Image.open('D:\Documents\python.jpg')
img1 = img.copy()
img2 = img.copy()
img3 = img.copy()
img.show()

img1_data = img1.load()
img2_data = img2.load()
img3_data = img3.load()

height, width = img1.size
height, width = img2.size
height, width = img3.size

for loop1 in range(height):
    for loop2 in range(width):
        r, g, b = img1_data[loop1, loop2]
        img1_data[loop1, loop2] = 0, g, b

for loop1 in range(height):
    for loop2 in range(width):
        r, g, b = img2_data[loop1, loop2]
        img2_data[loop1, loop2] = r, 0, b

for loop1 in range(height):
    for loop2 in range(width):
        r, g, b = img3_data[loop1, loop2]
        img3_data[loop1, loop2] = r, g, 0

A = array(img1)
B = array(img2)
```

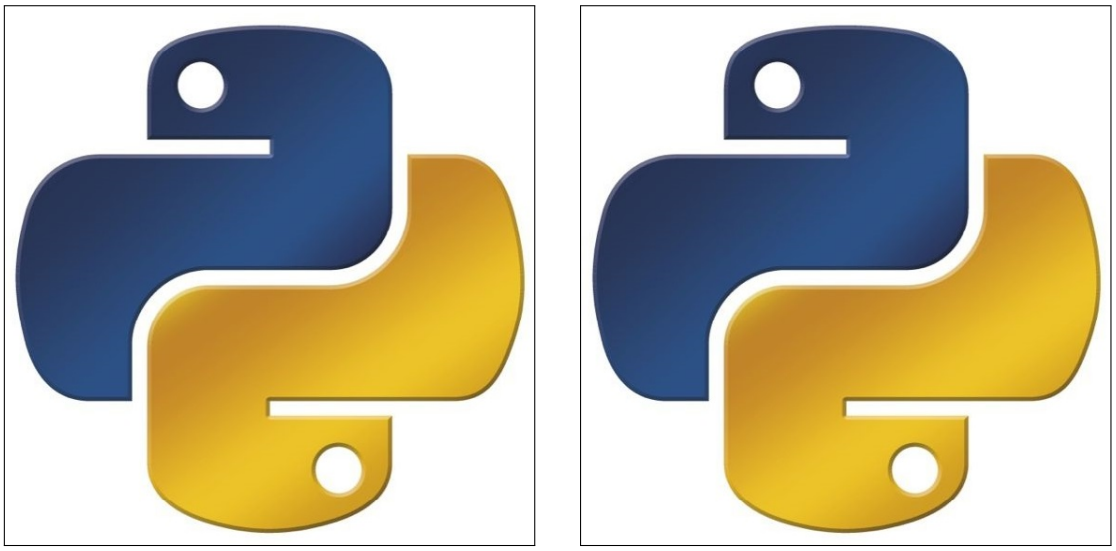
```
C = array(img3)
X = array(img)
def binary(x):  # from decimal to binary
    return int(bin(x)[2:])

def decimal(x):  # from binary to decimal
    return int(str(x), 2)

def c_cvt(x, y):
    a = binary(x)
    b = binary(y)
    c = "0"
    while a > 0 and b > 0:
        if a % 2 == 1 and b % 2 == 1:
            c = "1" + c
        else:
            c = "0" + c
        a = a // 10
        b = b // 10
    return decimal(int(c))


def c_xor(x, y):
    a = binary(x)
    b = binary(y)
    c = "0"
    while a > 0 or b > 0:
        if (a % 2 == 1 and b % 2 == 1) or (a % 2 == 0 and b
   % 2 == 0):
            c = "0" + c
        else:
            c = "1" + c
        a = a // 10
        b = b // 10
    c = int(c) // 10
    return decimal(c)


n = len(RGB)
m = len(RGB[0])
CVT1 = np.zeros((n, m, 3))
XOR1 = np.zeros((n, m, 3))
```

```
77
78 for i in range(n):
79     for j in range(m):
80         for k in range(3):
81             CVT1[i][j][k] = c_cvt(GB[i][j][k], RB[i][j][k])
82 for i in range(n):
83     for j in range(m):
84         for k in range(3):
85             XOR1[i][j][k] = c_xor(GB[i][j][k], RB[i][j][k])
86
87 CVT2 = np.zeros((n, m, 3))
88 XOR2 = np.zeros((n, m, 3))
89
90 for i in range(n):
91     for j in range(m):
92         for k in range(3):
93             CVT2[i][j][k] = c_cvt(RB[i][j][k], RG[i][j][k])
94 for i in range(n):
95     for j in range(m):
96         for k in range(3):
97             XOR2[i][j][k] = c_xor(RB[i][j][k], RG[i][j][k])
98
99 CVT3 = np.zeros((n, m, 3))
100 XOR3 = np.zeros((n, m, 3))
101
102 for i in range(n):
103     for j in range(m):
104         for k in range(3):
105             CVT3[i][j][k] = c_cvt(RG[i][j][k], GB[i][j][k])
106 for i in range(n):
107     for j in range(m):
108         for k in range(3):
109             XOR3[i][j][k] = c_xor(RG[i][j][k], GB[i][j][k])
110
111 m_sum = XOR1 + XOR2 + XOR3 + CVT1 + CVT2 + CVT3
112 m_og = np.divide(m_sum, 4)
113 result_sum = Image.fromarray(m_og.astype('uint8'), 'RGB')
114 result_sum.show()
115
116 a = Image.fromarray(m_CVT1.astype('uint8'), 'RGB')
117 b = Image.fromarray(m_CVT2.astype('uint8'), 'RGB')
118 c = Image.fromarray(m_CVT3.astype('uint8'), 'RGB')
```

```
a.show()
b.show()
c.show()

d = Image.fromarray(m_XOR1.astype('uint8'), 'RGB')
e = Image.fromarray(m_XOR2.astype('uint8'), 'RGB')
f = Image.fromarray(m_XOR3.astype('uint8'), 'RGB')
d.show()
e.show()
f.show()
```

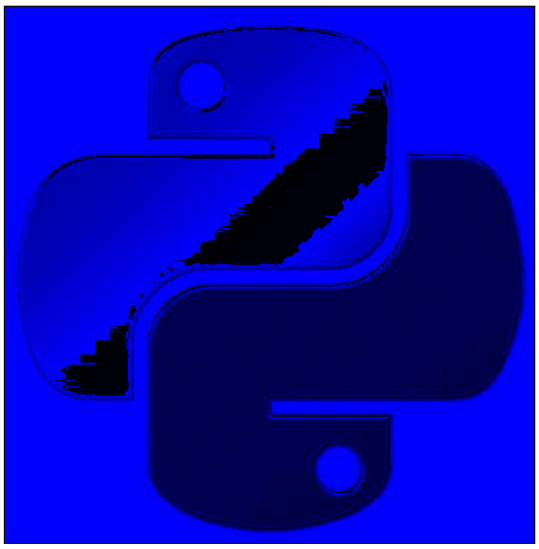
Listing 11.2: Applying CVT and XOR

Output



(a) img (b) result_sum

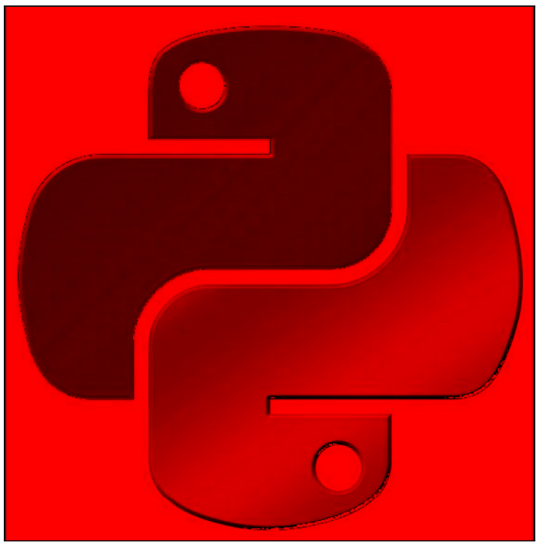
Figure 11.2: Image Comparison



(a) CVT(A,B)

(b) XOR(A,B)

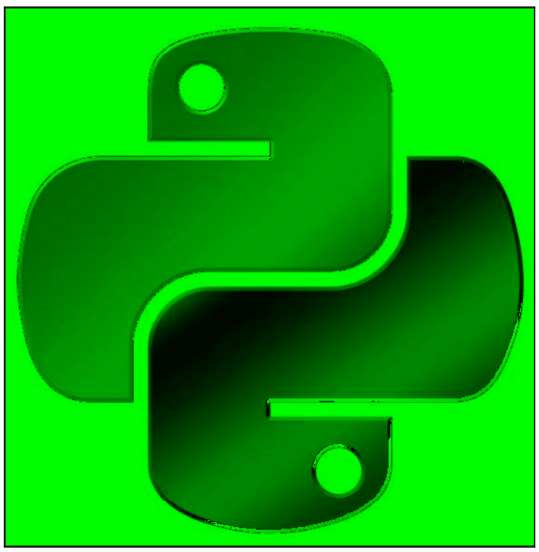
Figure 11.3: MCX(A,B)



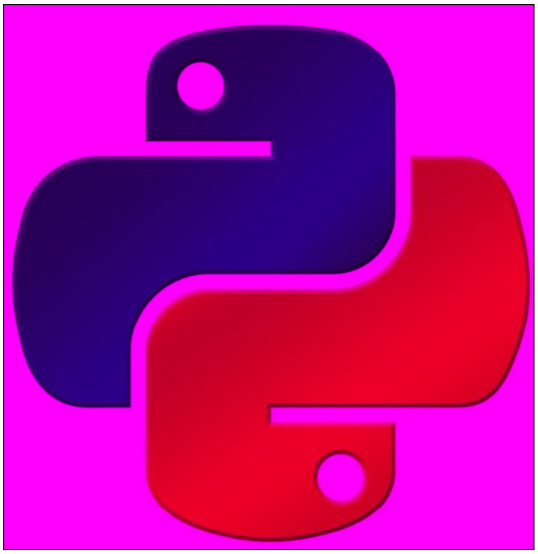
(a) CVT(B,C)

(b) XOR(B,C)

Figure 11.4: MCX(B,C)



(a) CVT(A,C)



(b) XOR(A,C)

Figure 11.5: MCX(A,C)

11.2.1 Why the code works?

We will see mathematically, how the above code works.

$m\_sum = XOR1 + XOR2 + XOR3 + CVT1 + CVT2 + CVT3$

Using theorem 2, section 6.1, we get the following three equation.

$XOR1 + CVT1 = A + B$

$XOR2 + CVT2 = B + C$

$XOR3 + CVT3 = A + C$

Using the above three equation we get,

$m\_sum = A + B + B + C + A + C$

$m\_sum = 2 \times (A + B + C)$

Since, A has components of green and blue colour, B has components of red and blue colour, C has components of red and green colour, when we add these three matrices we get twice the component of red, green and blue colour. Hence, we get twice the matrix of the original image.

$m\_sum = 2 \times (2 \times X)$

$m\_sum = 4 \times X$

$\frac{m\_sum}{4} = X$

$X = \frac{m\_sum}{4}$

CONCLUSION

In this project, we have proved some important results on Carry Value Transformation (CVT) and Bitwise exclusive OR (XOR). Using the proof of the theorem for any base of the number system, the sum of any two nonnegative integers is the same as the sum of their CVT and XOR values [7], we define a similar theorem for matrix with non-negative entries and prove it both mathematically and using python code with random matrices and their images. We have seen the authenticity of hypothesis 1 using python code in section 6.3.

After studying the pattern formed by different sequences using CVT, XOR and SVD, we observed that most of the data stored in the image formed by them of dimension (600×600), all of the important data was stored in first 10 to 20 rank of the matrix i.e. the first (10×10) to first (20×20) size of the matrix, hence, we used SVD to remove the unnecessary data, so that we do not consume unwanted space. We also used images to visualize and verify various decomposition, using python source codes.

Our hypothesis 1 can be proved in future. The proof of this hypothesis will be used for analyzing various matrices and sum of matrices, hence helping us to work with images and studying different patterns and recursions. Our work in [chapter 11] can be modified and used in encrypting images using steganography and further useful in cyber-security and other security purposes. All of our project can be used in pattern recognition and data interpretation, helping many statisticians, scientists and companies to work for further development of their own self and the world, by recognizing their benifits and loss and working on them.

REFERENCES

[1] Digital image basics. *https://visualresources.princeton.edu/*.

[2] Lu decomposition. *https://www.quantstart.com/*.

[3] *Pattern Anal. Appl.*, 20(2), 2017.

[4] Calculate the qr decomposition of a given matrix using numpy. *geeksforgeeks.com*, 2020.

[5] Steve Brunton. Singular value decomposition. *youtube.com*, 2020.

[6] Asma Khan. Doolittle algorithm : Lu decomposition. *codespeedy.com*.

[7] Suryakanta Pal, Sudhakar Sahoo, and Birendra Kumar Nayak. Properties of carry value transformation. *International Journal of Mathematics and Mathematical Sciences*, 2012, 2012.

[8] Souvik Saha. Analysis of algorithms using images. *Institute of Mathematics and Applications*, 2019.

[9] Shushant Shaw. Remove a specific color from an image in python. *codespeedy.com*.