

Comparative Study of Different PIT Techniques

by

Rishik Varma (ClassRoll – 001811001050)

&

Vishal Bharti (ClassRoll – 001811001003)

Under the guidance of

Mr. Sujit Kumar Das



Department of Information Technology,

Faculty of Engineering and Technology,

Jadavpur University,

Kolkata, India

2022

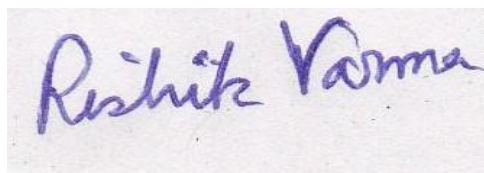
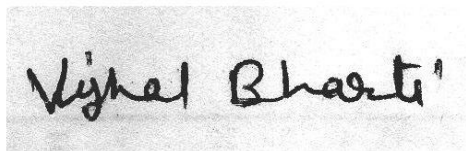
Students' Declaration

We hereby declare that this project report titled "Comparative Study of Different PIT Techniques" contains only the work completed by us as a part of Bachelors of Engineering course, during the academic year 2021-22 under the supervision and guidance of [Prof. Sujit Kumar Das](#) hailing from the department of Information Technology of Jadavpur University.

We ensure that all information, materials and methods that are not original to this work have been properly referenced and cited to the best of our knowledge.

We also declare that no part of this project has been submitted for the award of any other degree prior to this date.

Signature:



Date: 12th April, 2022

Certificate

This is to certify hereby that the project work titled “Comparative Study of Different PIT Techniques” performed by [Rishik Varma \(roll no: 001811001050\)](#) and [Vishal Bharti \(roll no: 001811001003\)](#) submitted to the Jadavpur University during the year 2021-22 for the degree of Bachelors of Engineering is a bonafide record of work done under my supervision.

The contents of this project, in full or in parts, have not been submitted to any other institute or university for fulfillment of any other degree.

Signature of guide :

Date :

Acknowledgement

We would like to take this opportunity to thank our project mentor [Mr. Sujit Kumar Das](#), Assistant Professor of Information Technology Department, Jadavpur University, Kolkata for entitling us with this project which was a great learning experience under his supervision.

We are indebted to him for his continual support and constant and invaluable feedback as our work progressed. It was a great experience working under his guidance.

Thank You!

Rishik Varma (001811001050)

Vishal Bharti (001811001003)

Abstract

In Daily Life we often require large size images transmitted over the Internet for various purposes. But everywhere it is not possible to have good and stable internet connection with high bandwidth.

In this research work we had devised three methods(Bit-Plane Method, A New Scheme Bit-Plane Method and Extended Local Binary Pattern[ELBP]) based on progressive image transmission to overcome this problem.

Keywords: Progressive Image Transmission(PIT), Bit-Plane Method(BPM), A New Scheme Bit Plane Method(NSBPM), Extended Local Binary Pattern(ELBP)

Contents

Declaration	2
Certificate	3
Acknowledgement	4
Abstract	5
Chapter 1: Progressive Image Transmission(PIT)	7
Chapter 2: Different Type of Methods Used in Progressive Image Transmission	8
Chapter 3: Bit Plane Method(BPM)	9-18
Chapter 4: A New Scheme Bit Plane Method(NSBPM)	19-29
Chapter 5: Extended Local Binary Pattern(ELBP)	30-45
Chapter 6: Comparison Between BPM, NSBPM and ELBP.	46
Chapter 7: Conclusion	47

Chapter-1

PROGRESSIVE IMAGE TRANSMISSION (PIT)

- ❖ The main idea of PIT is that it delivers an image in two or more phases.
- ❖ In the first phase, the PIT system might transmit a rough but recognizable Image.
- ❖ Subsequently, in later phases, the PIT system supplements the original rough image by transmitting detailed residual information of that image.
- ❖ The progressive image transmission (PIT) system here transmits the gross outline of an image first; subsequently, the receiver can choose to make the transmission stop at any point he or she is satisfied with.
- ❖ It is often applied to the transmission of high-resolution images over narrow bandwidth channels

Chapter-2

Different Type of Methods Used In PIT

- 1. BIT PLANE METHOD
(BPM)**
- 2. A NEW SCHEME BIT PLANE METHOD
(NSBPM)**
- 3. EXTENDED LOCAL BINARY PATTERN METHOD
(ELBP)**

Chapter-3

BIT PLANE METHOD

The Bit-Plane Method (BPM) is the simplest way to implement the progressive image transmission (PIT) system.

Explanation:

Suppose that each pixel in an image is represented by 8 bits (256 gray levels). We can decompose a gray image with eight 1-bit planes, ranging from Plane-0 for the most significant bit(MSB) to Plane-7 for the least significant bit(LSB).

In terms of 8-bit bytes, Plane 0 contains the main visual sense of an image, and the last bit plane contains subtle details of the image.

So, in the BPM, only the most significant bit of the intensity of each pixel in an image is sent in the first phase (i.e., phase 0). Subsequently, the image is repeatedly resent with the remaining bits added to it until the final image is lossless or until the receiver does not need the image quality to be further improved.

In fact, according to our observation, there are six highest orders of bits containing visually significant data. So, transmitting the six earlier planes is adequate for high image quality.

Now, let us consider the process of the BPM in detail.

- In Plane 0, the bit value 0 means that the pixel value of its corresponding pixel is equal to or smaller than 127, ranging from 0 to 127. Inversely, the bit value 1 in Plane 0 means that the pixel value of its corresponding pixel is larger than 127, ranging from 128 to 255. So in the 1st phase we will consider 0 or 128 only in the first phase for image formation.

i.e. in this phase intensity of every pixel will be 0 or 128 only.

- In Plane 1, the bit value 0 means; it has two cases as received from bit-plane 0 :
 - If Plane-0 value is 1 then we will consider the sum of value of individual bits that is $(128 + 0) = 128$.
 - If Plane-0 value is 0 then we will consider the sum of value of individual bits that is $(0 + 0) = 0$.
- In Plane 1, the bit value 1 means has two cases as received from bit-plane 0 :
 - If Plane-0 value is 1 then we will consider the sum of value of individual bits that is $(128 + 64) = 192$.
 - If Plane-0 value is 0 then we will consider the sum of value of individual bits that is $(0 + 64) = 64$.

- In Plane-2, the bit value 1 means it is decided based on the previous phase's bit value.
 - Like if we get 64 as value previously in two phases then we will take the sum of previous plus the new plane bit value. i.e. $(64 + 32) = 96$.
- And bit value 0 means it is decided based on the previous phase's bit value.
 - Like if we get 64 as value previously in two phases then we will take the sum of previous plus the new plane bit value. i.e. $(64 + 0) = 64$.
- Like that we will receive all bit planes till we get a lossless image or original image.
- We will keep on adding the bit planes till the last bit plane with their respective bit values.
- This method is the simplest method for Progressive Image Transmission(PIT).

Apply bit plane slicing on the following image size (3X3):

167	133	111
144	140	135
159	154	148

Fig 3.1 : Sample 3x3 image

Binary representation of this image ,

10100111	10000101	01101111
10010000	10001100	10000111
10011111	10011010	10010100

Fig 3.2 : Binary representation of the sample image

For 1st digit : Binary format of the 167 ,

1	0	1	0	0	1	1	1
MSB	7th bit	6th bit	5th bit	4th bit	3rd bit	2nd bit	LSB

Bit planes for the given 3X3 image :

1	1	0
1	1	1
1	1	1

a)MSB Bit Plane (Bit Plane - 0)

0	0	1
0	0	0
0	0	0

b)7th Bit Plane (Bit Plane-1)

1	0	1
0	0	0
0	0	0

a)6th Bit Plane (Bit Plane-2)

0	0	0
1	0	0
1	1	1

b)5th Bit Plane (Bit Plane-3)

0	0	1
0	1	0
1	1	0

a)4th Bit Plane (Bit Plane-4)

1	1	1
0	1	1
1	0	1

b)3rd Bit Plane (Bit Plane-5)

1	0	1
0	0	1
1	1	0

a) 2nd Bit Plane (Bit Plane-6)

1	1	1
0	0	1
1	0	0

b) LSB Bit Plane (Bit Plane-7)

Fig 3.3 : Bit Planes

- AT FIRST BIT-PLANE-0 IS SENT (BLURRED IMAGE).
- THEN PROGRESSIVELY KEEP ADDING THE OTHER BIT-PLANES TO THE BIT-PLANES SENT IN THE PREVIOUS STEPS.

After each phase the image formed :

128	128	0
128	128	128
128	128	128

a) 1st Phase

128	128	64
128	128	128
128	128	128

b) 2nd Phase

160	128	96
128	128	128
128	128	128

a)3rd Phase

160	128	96
144	128	128
144	144	144

b)4th Phase

160	128	104
144	136	128
152	152	144

a)5th Phase

164	132	108
144	140	132
156	152	148

b)6th Phase

166	132	110
144	140	134
158	154	148

a)7th Phase

167	133	111
144	140	135
159	154	148

b)8th Phase

Fig 3.4 : Image after each phase

- After the 8th phase we will get the final image.

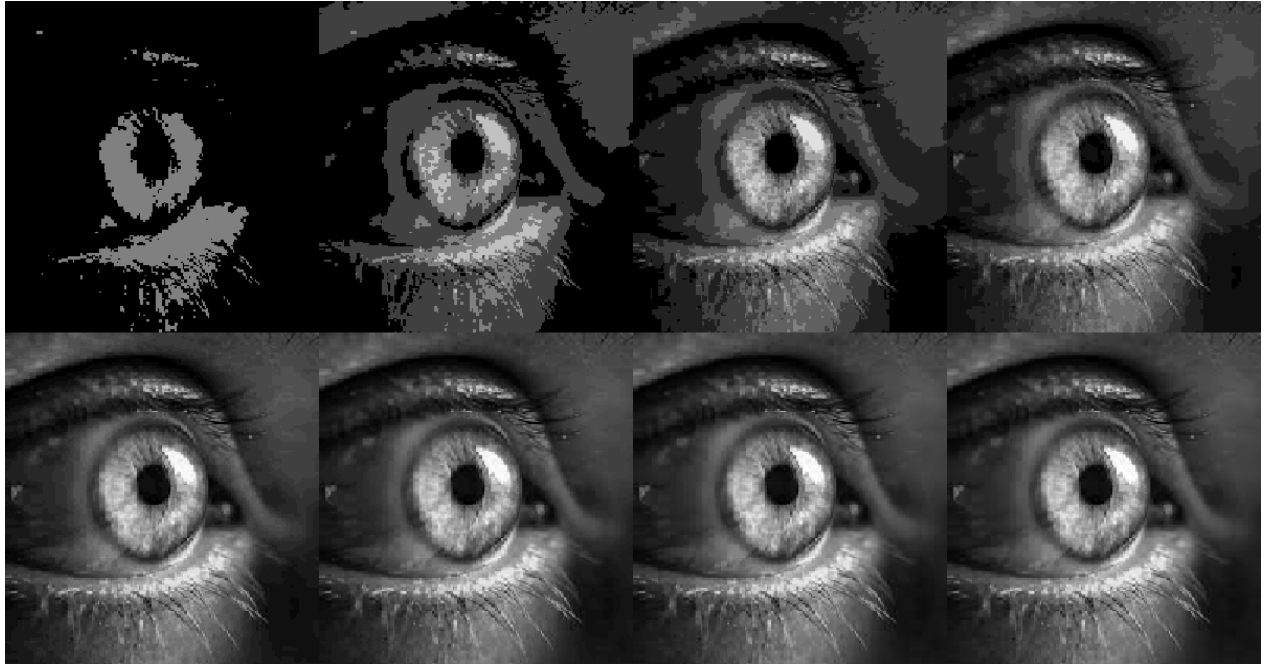


Fig 3.5 : Image Received in all 8 phases.

Code for this method :


```
1 import numpy as np
2 import cv2
3 from google.colab.patches import cv2_imshow
```

```
[ ] 1 from google.colab import files
    2 uploaded = files.upload()
```

No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.
Saving download1.jpg to download1.jpg

```
[ ] 1 image = cv2.imread('download1.jpg')
```

```
1 cv2_imshow(image)
```



```
[ ] 1 greyscale_image = cv2.imread('download1.jpg', cv2.IMREAD_GRAYSCALE)
```

```
[ ] 1 cv2_imshow(greyscale_image)
```



```
[ ] 1 row_size, column_size = greyscale_image.shape
```

```
1 def convert_binary(image) :
2     arr = []
3     for i in range(row_size) :
4         for j in range(column_size) :
5             arr.append(np.binary_repr(image[i][j], width=8))
6     return arr
```

```
[ ] 1 def get_bit_plane(plane_index, image) :
2   bit_arr = [int(pixel[plane_index]) for pixel in image]
3   return np.array(bit_arr)
```

```
[ ] 1 def get_weighted_bit_plane(plane_index, image) :
2   wt = 2**(7 - plane_index)
3   return image*wt
```

```
▶ 1 image = convert_binary(greyscale_image)
```

```
▶ 1 bit_plane_0 = get_weighted_bit_plane(0, get_bit_plane(0,image))
2 bit_plane_1 = get_weighted_bit_plane(1, get_bit_plane(1,image))
3 bit_plane_2 = get_weighted_bit_plane(2, get_bit_plane(2,image))
4 bit_plane_3 = get_weighted_bit_plane(3, get_bit_plane(3,image))
5 bit_plane_4 = get_weighted_bit_plane(4, get_bit_plane(4,image))
6 bit_plane_5 = get_weighted_bit_plane(5, get_bit_plane(5,image))
7 bit_plane_6 = get_weighted_bit_plane(6, get_bit_plane(6,image))
8 bit_plane_7 = get_weighted_bit_plane(7, get_bit_plane(7,image))
```

```
[ ] 1 def get_combination(*bitplanes) :
2   result = bitplanes[0]
3   for plane in bitplanes[1:] :
4       result = result + plane
5   return result
```

```
[ ] 1 image0 = get_combination(bit_plane_0)
2 img0 = np.reshape(image0, (row_size, column_size))
```

```
[ ] 1 image1 = get_combination(image0,bit_plane_1)
2 img1 = np.reshape(image1, (row_size, column_size))
```

```
[ ] 1 image2 = get_combination(image1,bit_plane_2)
2 img2 = np.reshape(image2, (row_size, column_size))
```

```
[ ] 1 image3 = get_combination(image2,bit_plane_3)
2 img3 = np.reshape(image3, (row_size, column_size))
```

```
▶ 1 image4 = get_combination(image3,bit_plane_4)
2 img4 = np.reshape(image4, (row_size, column_size))
```

+ Code

+ Text

```
[ ] 1 image5 = get_combination(image4,bit_plane_5)
2 img5 = np.reshape(image5, (row_size, column_size))
```

Add text cell

```
[ ] 1 image6 = get_combination(image5,bit_plane_6)
2 img6 = np.reshape(image6, (row_size, column_size))
```

```
[ ] 1 image7 = get_combination(image6,bit_plane_7)
2 img7 = np.reshape(image7, (row_size, column_size))
```

Chapter-4

A NEW SCHEME BIT PLANE METHOD (NSBPM)

This method is an extended version of bit plane method (BPM) where we do some manipulation during the bit plane addition for finding the value bit planes that we didn't receive till that time.

In this method we will improve the image quality of the BPM in each phase.

Explanation :

Suppose that each pixel in an image is represented by 8 bits (256 gray levels). We can decompose a gray image with eight 1-bit planes, ranging from Plane 0 for the most significant bit to Plane 7 for the least significant bit.

In terms of 8-bit bytes, Plane 0 contains the main visual sense of an image, and the last bit plane contains subtle details of the image. So, in the BPM, only the most significant bit of the intensity of each pixel in an image is sent in the first phase (i.e., phase 0). Subsequently, the image is repeatedly resent with the remaining bits added to it until the final image is lossless or until the receiver does not need the image quality to be further improved.

In fact, according to our observation, there are six highest orders of bits containing visually significant data. So, transmitting the six earlier planes is adequate for high image quality.

Now, let us consider the process of the BPM in detail.

- In Plane 0, the bit value 0 means,
 - that the pixel value of its corresponding pixel is equal to or smaller than 127, ranging from 0 to 127.
 - Inversely, the bit value 1 in Plane 0 means that the pixel value of its corresponding pixel is larger than 127, ranging from 128 to 255.
 - When a receiver receives Plane 0, in general, BPM applies the mean, “64”, of 0 and 127 to express the pixel whose bit value is 0.
 - And applies the mean, “192”, of 128 and 255 to express the pixel whose bit value is 1.
 - That is to say, a receiver uses the set (64, 192) to construct the image in phase 0.
 - By the same token, the receiver uses the set (32, 96, 160, 224) to construct the image in phase-1. This same procedure goes on for the later phases. Thus, we call these sets codebooks,

and call the elements the codewords in these sets.

- It is easy to organize these codebooks in the form of a tree-structured codebook. There are two codewords (i.e., 64 and 192) at level 1 of the tree-structured codebook, and there are four codewords (i.e., 32, 96, 160, and 224) at level 2 of the tree-structured codebook, and so on.

- Our new scheme, which improves the image quality of the BPM, is proposed here.
- The main difference between our scheme and the original BPM is that we send an additional tree-structured codebook from the sender to the receiver. This codebook could improve the image quality of the BPM effectively for each phase.

Apply new scheme bit plane slicing on the following image size (3X3):

167	133	111
144	140	135
159	154	148

Fig 4.1 : Sample image of size 3x3

Binary representation :

10100111	10000101	01101111
10010000	10001100	10000111
10011111	10011010	10010100

Fig 4.2 : Binary representation of sample image

- For 1st digit: Binary format of the 167 :

1	0	1	0	0	1	1	1
MSB	7th bit	6th bit	5th bit	4th bit	3rd bit	2nd bit	LSB

- Bit planes for the given 3X3 image :

1	1	0
1	1	1
1	1	1

a)MSB Bit Plane (Bit Plane - 0)

0	0	1
0	0	0
0	0	0

b)7th Bit Plane (Bit Plane-1)

1	0	1
0	0	0
0	0	0

a)6th Bit Plane (Bit Plane-2)

0	0	0
1	0	0
1	1	1

b)5th Bit Plane (Bit Plane-3)

0	0	1
0	1	0

1	1	1
0	1	1

1	1	0
---	---	---

a)4th Bit Plane (Bit Plane-4)

1	0	1
---	---	---

b)3rd Bit Plane (Bit Plane-5)

1	0	1
0	0	1
1	1	0

a)2nd Bit Plane (Bit Plane-6)

1	1	1
0	0	1
1	0	0

b)LSB Bit Plane (Bit Plane-7)

Fig 4.3 : Bit Planes

After Each Phase Image formed after mean calculation :

192	192	64
192	192	192
192	192	192

a)1st Phase

160	160	96
160	160	160
160	160	160

b)2nd Phase

176	144	112
144	144	144
144	144	144

a)3rd Phase

168	136	104
152	136	136
152	152	152

b)4th Phase

164	132	108
148	140	132
156	156	148

a)5th Phase

166	134	110
146	142	134
158	154	150

b)6th Phase

167	133	111
145	141	135
159	155	149

a)7th Phase

167	133	111
144	140	135
159	154	148

b)8th Phase

Fig 4.4 : Image after each phase



Fig 4.5 : Image Received in all 8 phases with their PSNR values.

PSNR:

Peak signal-to-noise ratio (PSNR) is the ratio between the maximum possible power of an image and the power of corrupting noise that affects the quality of its representation. To estimate the PSNR of an image, it is necessary to compare that image to an ideal clean image with the maximum possible power.

```
def PSNR(original, compressed):
```

```
    mse = np.mean((original - compressed) ** 2)
```

```
    if(mse == 0): # MSE is zero means no noise is present in the signal .
```

```
        # Therefore PSNR has no importance.
```

```
    return math.inf
```

```
max_pixel = 255
```

```
psnr = 20 * log10(max_pixel / sqrt(mse))
```

```
return psnr
```

Code for this method :

```
1 def convert_binary(image) :  
2     arr = []  
3     for i in range(row_size) :  
4         for j in range(column_size) :  
5             arr.append(np.binary_repr(image[i][j], width=8))  
6     return arr  
  
[ ] 1 def get_bit_plane(plane_index, image) :  
2     bit_arr = [int(pixel[plane_index]) for pixel in image]  
3     return np.array(bit_arr)  
  
[ ] 1 def get_weighted_bit_plane(plane_index, image) :  
2     power=(7-plane_index)  
3     wt = 2**power  
4     return image*wt  
  
[ ] 1 image = convert_binary(greyscale_image)  
  
[ ] 1 bit_plane_0 = get_bit_plane(0,image)  
2 bit_plane_1 = get_bit_plane(1,image)  
3 bit_plane_2 = get_bit_plane(2,image)  
4 bit_plane_3 = get_bit_plane(3,image)  
5 bit_plane_4 = get_bit_plane(4,image)  
6 bit_plane_5 = get_bit_plane(5,image)  
7 bit_plane_6 = get_bit_plane(6,image)  
8 bit_plane_7 = get_bit_plane(7,image)
```

```

1 def get_combination(*bitplanes) :
2     # result = bitplanes[0]
3     # for plane in bitplanes[1:] :
4     #     result = result + plane
5     # return result
6     result = None
7     length = len(bitplanes)
8     for i in range(length-1):
9         index = 7-i
10        if result is None:
11            result=bitplanes[i]*(2**index)
12        else:
13            result=result+bitplanes[i]*(2**index)
14    k = (bitplanes[-1]*(2**(8-length)))+2**(7-length)
15    if result is None:
16        result = k
17    else:
18        result = result + k
19    return np.floor(result)

```

```

[ ] 1 def PSNR(original, compressed):
2     mse = np.mean((original - compressed) ** 2)
3     if(mse == 0): # MSE is zero means no noise is present in the signal .
4         # Therefore PSNR have no importance.
5         return 100
6     max_pixel = 255.0
7     psnr = 20 * log10(max_pixel / sqrt(mse))
8     return psnr

```

```

1 image0 = get_combination(bit_plane_0)
2 img0 = np.reshape(image0, (row_size, column_size))
3 psnr0 = PSNR(greyscale_image,img0)

```

```

1 image1 = get_combination(bit_plane_0,bit_plane_1)
2 img1 = np.reshape(image1, (row_size, column_size))
3 psnr1 = PSNR(greyscale_image,img1)

```

```

1 image2 = get_combination(bit_plane_0,bit_plane_1,bit_plane_2)
2 img2 = np.reshape(image2, (row_size, column_size))
3 psnr2 = PSNR(greyscale_image,img2)

```

```

1 image3 = get_combination(bit_plane_0,bit_plane_1,bit_plane_2,bit_plane_3)
2 img3 = np.reshape(image3, (row_size, column_size))
3 psnr3 = PSNR(greyscale_image,img3)

```

```

1 image4 = get_combination(bit_plane_0,bit_plane_1,bit_plane_2,bit_plane_3,bit_plane_4)
2 img4 = np.reshape(image4, (row_size, column_size))
3 psnr4 = PSNR(greyscale_image,img4)

```

```

1 image5 = get_combination(bit_plane_0,bit_plane_1,bit_plane_2,bit_plane_3,bit_plane_4,bit_plane_5)
2 img5 = np.reshape(image5, (row_size, column_size))
3 psnr5 = PSNR(greyscale_image,img5)

```

```

1 image6 = get_combination(bit_plane_0,bit_plane_1,bit_plane_2,bit_plane_3,bit_plane_4,bit_plane_5,bit_plane_6)
2 img6 = np.reshape(image6, (row_size, column_size))

```

```
[ ] 1 font = cv2.FONT_HERSHEY_SIMPLEX
2
3 # org
4 org = (50, 280)
5
6 # fontScale
7 fontScale = 0.6
8
9 color = (255, 255, 255)
10
11 # Line thickness of 2 px
12 thickness = 2
```

```
▶ 1 img0 = cv2.resize(img0, (300,300), interpolation = cv2.INTER_NEAREST)
2 img0 = cv2.putText(img0, "IMAGE-0 PSNR: "+str(np.round(psnr0,2)), org, font, fontScale, color, thickness, cv2.LINE_AA)
3 img1 = cv2.resize(img1, (300,300), interpolation = cv2.INTER_NEAREST)
4 img1 = cv2.putText(img1, "IMAGE-1 PSNR: "+str(np.round(psnr1,2)), org, font, fontScale, color, thickness, cv2.LINE_AA)
5 img2 = cv2.resize(img2, (300,300), interpolation = cv2.INTER_NEAREST)
6 img2 = cv2.putText(img2, "IMAGE-2 PSNR: "+str(np.round(psnr2,2)), org, font, fontScale, color, thickness, cv2.LINE_AA)
7 img3 = cv2.resize(img3, (300,300), interpolation = cv2.INTER_NEAREST)
8 img3 = cv2.putText(img3, "IMAGE-3 PSNR: "+str(np.round(psnr3,2)), org, font, fontScale, color, thickness, cv2.LINE_AA)
9 img4 = cv2.resize(img4, (300,300), interpolation = cv2.INTER_NEAREST)
10 img4 = cv2.putText(img4, "IMAGE-4 PSNR: "+str(np.round(psnr4,2)), org, font, fontScale, color, thickness, cv2.LINE_AA)
11 img5 = cv2.resize(img5, (300,300), interpolation = cv2.INTER_NEAREST)
12 img5 = cv2.putText(img5, "IMAGE-5 PSNR: "+str(np.round(psnr5,2)), org, font, fontScale, color, thickness, cv2.LINE_AA)
13 img6 = cv2.resize(img6, (300,300), interpolation = cv2.INTER_NEAREST)
14 img6 = cv2.putText(img6, "IMAGE-6 PSNR: "+str(np.round(psnr6,2)), org, font, fontScale, color, thickness, cv2.LINE_AA)
15 img7 = cv2.resize(img7, (300,300), interpolation = cv2.INTER_NEAREST)
16 img7 = cv2.putText(img7, "IMAGE-7 PSNR: "+str(np.round(psnr7,2)), org, font, fontScale, color, thickness, cv2.LINE_AA)
```

```
horizontal1 = np.concatenate((img0,img1,img2,img3), axis = 1)
horizontal2 = np.concatenate((img4,img5,img6,img7), axis = 1)
final = np.concatenate((horizontal1,horizontal2), axis = 0)
cv2_imshow(final)
```



Chapter-5

EXTENDED LOCAL BINARY PATTERN METHOD (ELBP)

- In this method, a 3×3 block is encoded by k bits ($1 \leq k \leq 9$) with respect to the central pixel.
- Here image size should be multiple of 3×3 .
- Suppose we have an image of size 4×5 ;
Then we have to manipulate the image

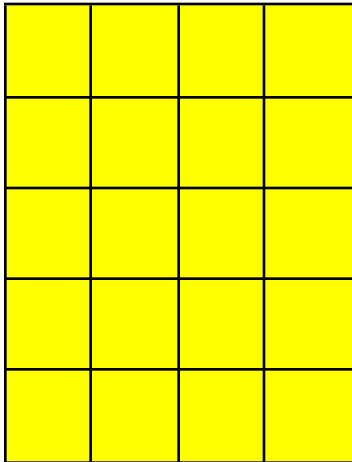
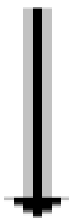


Fig 5.1 : 4x5 image



converted to a size of multiple of 3×3 . i.e 6×6 .

Filling white spaces for extra pixels generated.

Or some operations should be done to resize the image to a new size. i.e. multiple of 3x3.

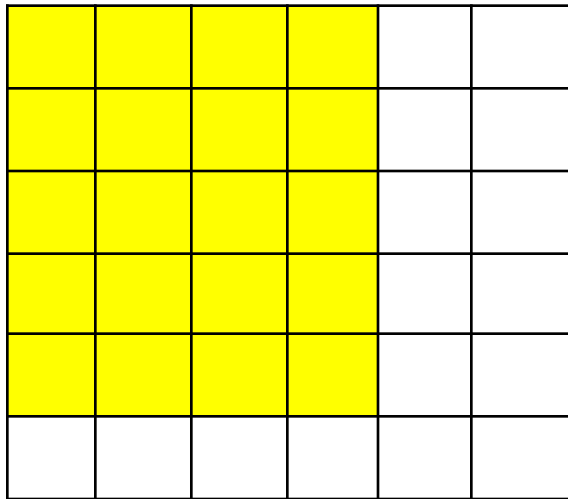


Fig 5.2 : 4x5 image converted to 6x6 image

- Local Binary Pattern (LBP) is defined as a gray scale invariant texture measure and is a useful operator to model texture images. In the LBP method, an image block of size 3×3 is considered.
- LBP operator labels each neighbor pixel G_i with respect to the central pixel G_c and, say, the label of

G_i is B_i for $i = 0, 1, 2, \dots, 7$.

1 if $G_i \geq G_c$

$B_i = \left\{ \right.$

0 if $G_i < G_c$

G_0	G_1	G_2
G_3	G_c	G_4
G_5	G_6	G_7

Fig 5.3 : 3X3 image block

B_0	B_1	B_2
B_3		B_4
B_5	B_6	B_7

Fig 5.4 : Binary levels of pixel

Let's take an example,

48	51	52
49	50	53
47	41	57

Fig 5.5 : Sample image of size 3x3



Its Binary Descriptor

0	1	1
0	50	1
0	0	1

Fig 5.6 : Binary Descriptor of Sample image

Here $G_c = 50$.

So,

$G_0 < G_c$ therefore $B_0 = 0$

$G_1 > G_c$ therefore $B_1 = 1$

$G_2 > G_c$ therefore $B_2 = 1$

$G_3 < G_c$ therefore $B_3 = 0$

$G_4 > G_c$ therefore $B_4 = 1$

$G_5 < G_c$ therefore $B_5 = 0$

$G_6 < G_c$ therefore $B_6 = 0$

$G_7 > G_c$ therefore $B_7 = 1$

Here in this we will get 1 bit value and only compare values with g_c if it is greater and smaller.

Resultant image :

49	50	50
49	50	50
49	49	50

Fig 5.7 : Resultant image

Here we can see that only one difference with G_c is considered. And there can be more than a difference of 1. So to overcome this problem we will use ELBP. i.e we will use k bits ($0 \leq k < 9$) for a total of 9 iterations and we can easily find the difference in later phases.

So for $k = 0$,

48	51	52
49	50	53
47	41	57

Fig 5.8 : Original block

0	1	1
0	50	1
0	0	1

Fig 5.9 : Descriptor block

49	50	50
49	50	50
49	49	50

Fig 5.10 : Final block

For $k = 1$,

In this case we will use 1 bit for greater than and less than and one bit for the differences between G_c and G_i .

If $p(i,k) = 1$ then $P(i,k) \dots P(i,0) = \text{binary}(\min((G_i - G_c), 2^k - 1))$

If $p(i,k) = 0$ then $P(i,k) \dots P(i,0) = \text{binary}(\min((G_c - G_i), 2^k - 1))$

i	G_i	$P(i,1)$	$P(i,1) \dots P(i,0)$	P_i
0	48	0	$\text{binary}(\min(2,1)) = 1$	01
1	51	1	$\text{binary}(\min(1,1)) = 1$	11
2	52	1	$\text{binary}(\min(2,1)) = 1$	11
3	49	0	$\text{binary}(\min(1,1)) = 1$	01
4	53	1	$\text{binary}(\min(3,1)) = 1$	11
5	47	0	$\text{binary}(\min(3,1)) = 1$	01
6	41	0	$\text{binary}(\min(9,1)) = 1$	01
7	57	1	$\text{binary}(\min(,1)) = 1$	11

Table 5.1 : Description process for $k = 1$

01	11	11
01	50	11
01	01	11

Fig 5.11 : Descriptive block

i	Pi	P(i,1)	Decimal value	Gi'
0	01	0	1	49
1	11	1	1	51
2	11	1	1	51
3	01	0	1	49
4	11	1	1	51
5	01	0	1	49
6	01	0	1	49
7	11	1	1	51

Table 5.2 : Final Image process for k = 1

49	51	51
49	50	51
49	49	51

Fig 5.12 : Final Block

For K = 2,

In this case we will use 1 bit for greater than and less than and two bits for the differences between G_c and G_i .

i	G _i	P(i,2)	P(i,2)...P(i,0)	P _i
0	48	0	binary(min(2,3)) = 10	010
1	51	1	binary(min(1,3)) = 01	101
2	52	1	binary(min(2,3)) = 10	110
3	49	0	binary(min(1,3)) = 01	001
4	53	1	binary(min(3,3)) = 11	111
5	47	0	binary(min(3,3)) = 11	011
6	41	0	binary(min(9,3)) = 11	011
7	57	1	binary(min(7,3)) = 11	111

Table 5.3 : Description process for k = 2

010	101	110
001	50	111
011	011	111

Fig 5.13 : Descriptor block

i	P _i	P(i,2)	Decimal value	G _i '
0	010	0	2	48
1	101	1	1	51
2	110	1	2	52
3	001	0	1	49

4	111	1	3	53
5	011	0	3	47
6	011	0	3	47
7	111	1	3	53

Table 5.4 : Final image process for $k = 2$

48	51	52
49	50	53
47	47	53

Fig 5.14 : Final Block

For $K = 3$,

In this case we will use 1 bit for greater than and less than and three bits for the differences between G_c and G_i .

i	G_i	$P(i,3)$	$P(i,3)...P(i,0)$	P_i
0	48	0	binary(min(2,7)) = 10	0010
1	51	1	binary(min(1,7)) = 01	1001
2	52	1	binary(min(2,7)) = 10	1010
3	49	0	binary(min(1,7)) = 01	0001

4	53	1	binary(min(3,7)) = 11	1011
5	47	0	binary(min(3,7)) = 11	0011
6	41	0	binary(min(9,7)) = 111	0111
7	57	1	binary(min(7,7)) = 111	1111

Table 5.5 : Description process for k = 3

0010	1001	1010
0001	50	1011
0011	0111	1111

Fig 5.15 : Descriptor block

i	Pi	P(i,3)	Decimal value	Gi'
0	0010	0	2	48
1	1001	1	1	51
2	1010	1	2	52
3	0001	0	1	49
4	1011	1	3	53
5	0011	0	3	47
6	0111	0	7	43
7	1111	1	7	57

Table 5.6 : Final Image process for k = 3

48	51	52
49	50	53
47	43	57

Fig 5.16 : Final Block

So likewise for all other values of $k(0 \leq k < 9)$ different descriptor blocks are created and sent to the receiver and on that basis the final block the image is created.

After the 9th Phase we will get the Final Image,





Fig 5.17 : Images after each 9 Phases with PSNR values.

Code for this method:

```
[ ] 1 import numpy as np
    2 import cv2
    3 from google.colab.patches import cv2_imshow
    4 from math import log10, sqrt
```

```
[ ] 1 from google.colab import files
    2 uploaded = files.upload()
```

No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.
Saving download.jpg to download (4).jpg

```
[ ] 1 image = cv2.imread('download.jpg')
```

```
[ ] 1 cv2_imshow(image)
```



Resizing image to multiple of 3X3 image

```
[ ] 1 row_size, column_size = greyscale_image.shape
    2 new_row_size = ((row_size+2)//3)*3
    3 new_col_size = ((column_size+2)//3)*3
    4 greyscale_image = cv2.resize(greyscale_image, (new_row_size,new_col_size), interpolation = cv2.INTER_NEAREST)
    5 # greyscale_image = cv2.resize(greyscale_image, (12,12), interpolation = cv2.INTER_NEAREST)
    6 greyscale_image = np.array(greyscale_image)
```

```
▶ 1
  2 cv2_imshow(greyscale_image)
```



```
[ ] 1
    2 row_size, column_size = greyscale_image.shape
    3
    4 print(row_size)
```

Extended LBP Local binary pattern

```
1 def getKthImage(image,k):
2     row,col = image.shape
3     gcs=[]
4     result=[["GC" for i in range(col)] for i in range(row)]
5     for i in range(1,row,3):
6         for j in range(1,col,3):
7             gcs.append([i,j])
8     if k==1:
9         for i in range(len(gcs)):
10            x = gcs[i][0]
11            y = gcs[i][1]
12            for l in range((x//3)*3,(x//3)*3+3):
13                for r in range((y//3)*3,(y//3)*3+3):
14                    if(l!=x or r!=y):
15                        result[l][r]= str(0 if image[l][r]<image[x][y] else 1)
16                    else:
17                        result[l][r]=str(image[x][y])
18
19     else:
20         for i in range(len(gcs)):
21            x = gcs[i][0]
22            y = gcs[i][1]
23            for l in range((x//3)*3,(x//3)*3+3):
24                for r in range((y//3)*3,(y//3)*3+3):
25                    if(l!=x or r!=y):
26                        result[l][r]=str(0 if image[l][r]<image[x][y] else 1)
27                        diff = min((abs(int(image[l][r])-int(image[x][y]))),2**(k-1)-1)
28                        binary_rep = np.binary_repr(diff,k-1)
29                        result[l][r]+=binary_rep
30                    else:
31                        result[l][r]=str(image[x][y])
32     return result
33
```

```
1 def getPixelValue(cell,gc):
2     if(cell == gc):
3         return int(gc)
4     ans=int(gc)
5     if cell[0]!='0':
6         if(len(cell) > 1):
7             ans+=int(cell[1:],2)
8         else:
9             ans-=1
10
11 elif(len(cell) > 1):
12     ans+=int(cell[1:],2)
13     return ans
14
```

```
[ ] 1 def getFinalImage(image):
2     row = len(image)
3     col = len(image[0])
4     result=[[0 for i in range(col)] for i in range(row)]
5     for i in range(row):
6         for j in range(col):
7             gc = image[(i//3)*3+1][(j//3)*3+1]
8             result[i][j]=getPixelValue(image[i][j],gc)
9     return np.array(result)
```

```

▶ 1 img1 = getKthImage(greyscale_image,1)
2 img2 = getKthImage(greyscale_image,2)
3 img3 = getKthImage(greyscale_image,3)
4 img4 = getKthImage(greyscale_image,4)
5 img5 = getKthImage(greyscale_image,5)
6 img6 = getKthImage(greyscale_image,6)
7 img7 = getKthImage(greyscale_image,7)
8 img8 = getKthImage(greyscale_image,8)
9 img9 = getKthImage(greyscale_image,9)

```

```

[ ] 1 image1 = getFinalImage(img1)
2 image2 = getFinalImage(img2)
3 image3 = getFinalImage(img3)
4 image4 = getFinalImage(img4)
5 image5 = getFinalImage(img5)
6 image6 = getFinalImage(img6)
7 image7 = getFinalImage(img7)
8 image8 = getFinalImage(img8)
9 image9 = getFinalImage(img9)

```

```

[ ] 1 horizontal1 = np.concatenate((image1,image2,image3,image4), axis = 1)
2 horizontal2 = np.concatenate((image5,image6,image7,image8), axis = 1)
3 final = np.concatenate((horizontal1,horizontal2), axis = 0)
4 cv2_imshow(final)
5 cv2_imshow(image9)

```

```

[ ] 1 def PSNR(original, compressed):
2     mse = np.mean((original - compressed) ** 2)
3     if(mse == 0): # MSE is zero means no noise is present in the signal .
4         # Therefore PSNR have no importance.
5         return 100
6     max_pixel = 255.0
7     psnr = 20 * log10(max_pixel / sqrt(mse))
8     return psnr

```

```

▶ 1 psnr1 = PSNR(greyscale_image,image1)
2 psnr2 = PSNR(greyscale_image,image2)
3 psnr3 = PSNR(greyscale_image,image3)
4 psnr4 = PSNR(greyscale_image,image4)
5 psnr5 = PSNR(greyscale_image,image5)
6 psnr6 = PSNR(greyscale_image,image6)
7 psnr7 = PSNR(greyscale_image,image7)
8 psnr8 = PSNR(greyscale_image,image8)
9 psnr9 = PSNR(greyscale_image,image9)

```

```

[ ] 1 font = cv2.FONT_HERSHEY_SIMPLEX
2
3 # org
4 org = (50, 280)
5
6 # fontScale
7 fontScale = 0.6
8

```

```
[ ] 1 font = cv2.FONT_HERSHEY_SIMPLEX
2
3 # org
4 org = (50, 280)
5
6 # fontScale
7 fontScale = 0.6
8
9 color = (255, 255, 255)
10
11 # Line thickness of 2 px
12 thickness = 2
```

```
1 image1 = cv2.resize(image1, (300,300), interpolation = cv2.INTER_NEAREST)
2 image1 = cv2.putText(image1, "IMAGE-1 PSNR: "+str(np.round(psnr1,2)), org, font, fontScale, color, thickness, cv2.LINE_AA)
3 image2 = cv2.resize(image2, (300,300), interpolation = cv2.INTER_NEAREST)
4 image2 = cv2.putText(image2, "IMAGE-2 PSNR: "+str(np.round(psnr2,2)), org, font, fontScale, color, thickness, cv2.LINE_AA)
5 image3 = cv2.resize(image3, (300,300), interpolation = cv2.INTER_NEAREST)
6 image3 = cv2.putText(image3, "IMAGE-3 PSNR: "+str(np.round(psnr3,2)), org, font, fontScale, color, thickness, cv2.LINE_AA)
7 image4 = cv2.resize(image4, (300,300), interpolation = cv2.INTER_NEAREST)
8 image4 = cv2.putText(image4, "IMAGE-4 PSNR: "+str(np.round(psnr4,2)), org, font, fontScale, color, thickness, cv2.LINE_AA)
9 image5 = cv2.resize(image5, (300,300), interpolation = cv2.INTER_NEAREST)
10 image5 = cv2.putText(image5, "IMAGE-5 PSNR: "+str(np.round(psnr5,2)), org, font, fontScale, color, thickness, cv2.LINE_AA)
11 image6 = cv2.resize(image6, (300,300), interpolation = cv2.INTER_NEAREST)
12 image6 = cv2.putText(image6, "IMAGE-6 PSNR: "+str(np.round(psnr6,2)), org, font, fontScale, color, thickness, cv2.LINE_AA)
13 image7 = cv2.resize(image7, (300,300), interpolation = cv2.INTER_NEAREST)
14 image7 = cv2.putText(image7, "IMAGE-7 PSNR: "+str(np.round(psnr7,2)), org, font, fontScale, color, thickness, cv2.LINE_AA)
15 image8 = cv2.resize(image8, (300,300), interpolation = cv2.INTER_NEAREST)
16 image8 = cv2.putText(image8, "IMAGE-8 PSNR: "+str(np.round(psnr8,2)), org, font, fontScale, color, thickness, cv2.LINE_AA)
17 image9 = cv2.resize(image9, (300,300), interpolation = cv2.INTER_NEAREST)
18 image9 = cv2.putText(image9, "IMAGE-9 PSNR: "+str(np.round(psnr9,2)), org, font, fontScale, color, thickness, cv2.LINE_AA)
```



Chapter-6

Comparing NSBPM & ELBP

Phase-i	NSBPM Psnr Value	ELBP Psnr Value
Phase-0	16.82	22.79
Phase-1	22.87	22.9
Phase-2	28.83	23.32
Phase-3	34.66	24.13
Phase-4	40.73	25.64
Phase-5	46.4	28.47
Phase-6	51.16	34.09
Phase-7	inf	46.83
Phase-8	—————	inf

Table 6.1 : Comparison between NSBPM & ELBP

Chapter-7

Conclusion

Observations from the Comparison table :

- In Phase-0 Image formed in ELBP is better.
- After Phase-7 we will get a lossless image in NSBPM but not in ELBP.
- Value of PSNR increase in NSBPM is more than ELBP.