Software Engineering
Lab - 7
202201404
Swapnil Shukla

Part 1:

The Code Is provided after the answers for your reference

The Answers To The Following Questions :-

1. How many errors are there in the program? Mention the errors you have identified.

- **Data Reference Errors**:
  - Uninitialized variables may lead to undefined behavior, particularly when input is not validated.
  - Integer division may cause precision loss, such as `z = x / y` yielding `0` for integer inputs.
- **Data-Declaration Errors**:
  - While all variables are declared, some initializations can lead to unexpected outcomes (e.g., uninitialized array elements).
- **Computation Errors**:
  - Mixing integer division with floating-point arithmetic can result in confusion, illustrated by `z = x / y` when both x and y are integers.
- **Comparison Errors**:
  - Errors can occur from comparisons involving different data types or insufficient validation of input types (e.g., array index or user input comparisons).

- **Control-Flow Errors**:
  - Loops must be designed to ensure they terminate correctly to prevent infinite loops.
- **Interface Errors**:
  - It's essential to confirm that functions are called with the correct number and types of parameters to avoid runtime issues.
- **Input/Output Errors**:
  - User input must be validated to avert potential crashes or unintended behaviors, especially during file or console operations.
- **Overall Count**:
  - A minimum of 5-10 potential issues can be pinpointed based on the code fragments and the inspection checklist provided.

2. Which category of program inspection would you find more effective?

- **Data Reference Errors**:
  - This category is likely the most effective, as these errors can lead to runtime exceptions or undefined behavior, which are often hard to debug.

3. Which type of error are you not able to identify using the program inspection?

- **Logical Errors**:
  - These types of errors are challenging to spot using inspections since the code may run without any syntax issues but still produce incorrect results due to flawed logic.

4. Is the program inspection technique worth applying?

Absolutely, it is worthwhile:

- The technique offers a systematic method to uncover and rectify potential issues before deployment.
- Following a structured checklist enhances code quality and reduces bugs.
- Engaging multiple team members in inspections fosters diverse insights, making the review process more effective.

Part 2: Debugging

The numbers represent the codes:

Code 1:-

1. Errors Identified:

- Incorrect remainder calculation: Should be `num % 10` instead of `num / 10`.
- Incorrect number reduction: Should be `num / 10` instead of `num % 10`.

2. Number of Breakpoints:

- 2 breakpoints:
  - At the remainder calculation.
  - At the number reduction.

2(a). Steps to Fix:

- Step 1: Change `remainder = num / 10` to `remainder = num % 10`.
- Step 2: Change `num = num % 10` to `num = num / 10`.

(3)

```
class Armstrong {
public static void main(String args[]) {
    int num = Integer.parseInt(args[0]);
    int n = num;
    int check = 0, remainder;
    while (num > 0) {
        remainder = num % 10;
        check = check + (int)Math.pow(remainder, 3);
```

```
        num = num / 10;
      }
      if (check == n)
        System.out.println(n + " is an Armstrong Number");
      else
        System.out.println(n + " is not an Armstrong Number");
   }
}
```

<u>Code 2:-</u>

1. Errors Identified:

- Incorrect condition in GCD loop: In the `gcd` method, the while condition should be `a % b != 0` instead of `a % b == 0`.
- Incorrect LCM logic: In the `lcm` method, the condition should check for `a % x == 0 && a % y == 0` (both should divide a) instead of `a % x != 0 && a % y != 0`.

2. Number of Breakpoints:

- 2 breakpoints:
  - At the GCD loop condition.
  - At the LCM condition.

2(a). Steps to Fix:

- Step 1: In the `gcd` method, replace `while(a % b == 0)` with `while(a % b != 0)`.
- Step 2: In the `lcm` method, change the condition `if(a % x != 0 && a % y != 0)` to `if(a % x == 0 && a % y == 0)`.

3. Corrected Code:

```
import java.util.Scanner;

public class GCD_LCM
{
    static int gcd(int x, int y)
    {
        int r=0, a, b;
        a = (x > y) ? y : x; // a is smaller number
        b = (x < y) ? x : y; // b is larger number

        while(a % b != 0) // Fix: correct condition
        {
            r = a % b;
            a = b;
            b = r;
        }
        return b;
    }

    static int lcm(int x, int y)
    {
        int a;
        a = (x > y) ? x : y; // a is greater number
        while(true)
        {
            if(a % x == 0 && a % y == 0) // Fix: check both divisions
                return a;
            ++a;
        }
    }
```

```
    public static void main(String args[])
    {
        Scanner input = new Scanner(System.in);
        System.out.println("Enter the two numbers: ");
        int x = input.nextInt();
        int y = input.nextInt();

        System.out.println("The GCD of two numbers is: " + gcd(x, y));
        System.out.println("The LCM of two numbers is: " + lcm(x, y));
        input.close();
    }
}
```

<u>Code 3:-</u>

1. Errors Identified:

- Incorrect increment for n in the loop: The line `int option1 = opt[n++][w];` mistakenly increments n. It should be `opt[n-1][w]` to avoid skipping iterations.
- Incorrect profit calculation when taking the item: The line `int option2 = profit[n-2] + opt[n-1][w-weight[n]];` wrongly accesses `profit[n-2]`. It should access `profit[n]` to get the current item's profit.

2. Number of Breakpoints:

- 2 breakpoints:
    - At the calculation of `option1`.
    - At the calculation of `option2`.

2(a). Steps to Fix:

- Step 1: Replace `opt[n++][w]` with `opt[n-1][w]` to fix incorrect item selection.
- Step 2: Replace `profit[n-2]` with `profit[n]` to correctly add the current item's profit.

3. Corrected Code:

```
public class Knapsack {

    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);   // number of items
        int W = Integer.parseInt(args[1]);   // maximum weight of
knapsack

        int[] profit = new int[N+1];
        int[] weight = new int[N+1];

        // generate random instance, items 1..N
        for (int n = 1; n <= N; n++) {
            profit[n] = (int) (Math.random() * 1000);
            weight[n] = (int) (Math.random() * W);
        }

        // opt[n][w] = max profit of packing items 1..n with weight limit w
        // sol[n][w] = does opt solution to pack items 1..n with weight limit
w include item n?
        int[][] opt = new int[N+1][W+1];
        boolean[][] sol = new boolean[N+1][W+1];

        for (int n = 1; n <= N; n++) {
            for (int w = 1; w <= W; w++) {
```

```
            // don't take item n
            int option1 = opt[n-1][w];  // Fix: use n-1

            // take item n
            int option2 = Integer.MIN_VALUE;
            if (weight[n] <= w) {
                option2 = profit[n] + opt[n-1][w-weight[n]];  // Fix: use
profit[n]
            }

            // select better of two options
            opt[n][w] = Math.max(option1, option2);
            sol[n][w] = (option2 > option1);
        }
    }

    // determine which items to take
    boolean[] take = new boolean[N+1];
    for (int n = N, w = W; n > 0; n--) {
        if (sol[n][w]) {
            take[n] = true;
            w = w - weight[n];
        } else {
            take[n] = false;
        }
    }

    // print results
    System.out.println("item" + "\t" + "profit" + "\t" + "weight" + "\t" +
"take");
    for (int n = 1; n <= N; n++) {
```

```
            System.out.println(n + "\t" + profit[n] + "\t" + weight[n] + "\t" +
take[n]);
        }
    }
}
```

Code 4:-

1. Errors Identified:

- Incorrect while condition in inner loop: The condition `while(sum == 0)` is incorrect. It should be `while(sum > 0)` to process the digits.
- Incorrect multiplication in inner loop: The line `s = s * (sum / 10);` is incorrect. It should be `s = s + (sum % 10);` to sum up the digits.
- Missing semicolon after `sum = sum % 10;`.

2. Number of Breakpoints:

- 3 breakpoints:
  - At the inner loop condition.
  - At the digit summation.
  - After the missing semicolon.

2(a). Steps to Fix:

- Step 1: Change `while(sum == 0)` to `while(sum > 0)`.

- Step 2: Replace `s = s * (sum / 10);` with `s = s + (sum % 10);`.
- Step 3: Add a semicolon after `sum = sum % 10;`.

3. Corrected Code:

```
import java.util.*;
public class MagicNumberCheck
{
    public static void main(String args[])
    {
        Scanner ob = new Scanner(System.in);
        System.out.println("Enter the number to be checked.");
        int n = ob.nextInt();
        int sum = 0, num = n;

        while(num > 9)
        {
            sum = num;
            int s = 0;
            while(sum > 0)  // Fix: change condition to sum > 0
            {
                s = s + (sum % 10);  // Fix: sum digits
                sum = sum / 10;  // Fix: divide sum by 10 to move to next
digit
            }
            num = s;  // update num to new sum of digits
        }

        if(num == 1)
        {
            System.out.println(n + " is a Magic Number.");
```

```
        }
        else
        {
            System.out.println(n + " is not a Magic Number.");
        }
    }
}
```

Code 5:-

. Errors Identified:

- Incorrect array references in `mergeSort`:
  - `leftHalf(array+1)` and `rightHalf(array-1)` are incorrect operations on arrays. It should just pass `array` to both `leftHalf` and `rightHalf`.
  - The operations `merge(array, left++, right--)` are invalid because you cannot increment/decrement arrays. You should pass `left` and `right` as they are.

2. Number of Breakpoints:

- 2 breakpoints:
  - When splitting the array into halves.
  - When merging the sorted arrays.

2(a). Steps to Fix:

- Step 1: Replace `leftHalf(array+1)` with `leftHalf(array)` and `rightHalf(array-1)` with `rightHalf(array)` in the `mergeSort` method.

- Step 2: Change `merge(array, left++, right--)` to `merge(array, left, right)` to correctly pass the arrays.

3. Corrected Code:

```
import java.util.*;

public class MergeSort {
    public static void main(String[] args) {
        int[] list = {14, 32, 67, 76, 23, 41, 58, 85};
        System.out.println("before: " + Arrays.toString(list));
        mergeSort(list);
        System.out.println("after:  " + Arrays.toString(list));
    }

    // Places the elements of the given array into sorted order
    // using the merge sort algorithm.
    // post: array is in sorted (nondecreasing) order
    public static void mergeSort(int[] array) {
        if (array.length > 1) {
            // split array into two halves
            int[] left = leftHalf(array);  // Fix: pass array
            int[] right = rightHalf(array); // Fix: pass array

            // recursively sort the two halves
            mergeSort(left);
            mergeSort(right);

            // merge the sorted halves into a sorted whole
            merge(array, left, right);  // Fix: pass left and right
        }
    }
```

```java
// Returns the first half of the given array.
public static int[] leftHalf(int[] array) {
    int size1 = array.length / 2;
    int[] left = new int[size1];
    for (int i = 0; i < size1; i++) {
        left[i] = array[i];
    }
    return left;
}

// Returns the second half of the given array.
public static int[] rightHalf(int[] array) {
    int size1 = array.length / 2;
    int size2 = array.length - size1;
    int[] right = new int[size2];
    for (int i = 0; i < size2; i++) {
        right[i] = array[i + size1];
    }
    return right;
}

// Merges the given left and right arrays into the given
// result array.
// pre : result is empty; left/right are sorted
// post: result contains result of merging sorted lists;
public static void merge(int[] result,
                         int[] left, int[] right) {
    int i1 = 0;   // index into left array
    int i2 = 0;   // index into right array

    for (int i = 0; i < result.length; i++) {
```

```
            if (i2 >= right.length || (i1 < left.length &&
                    left[i1] <= right[i2])) {
                result[i] = left[i1];    // take from left
                i1++;
            } else {
                result[i] = right[i2];   // take from right
                i2++;
            }
        }
    }
}
```

Code 6:-

1. Errors Identified:

- Incorrect indexing in the multiplication loop:
  - In the statement `first[c-1][c-k]`, `second[k-1][k-d]`, the index should not involve `-1`. The correct form should be `first[c][k]` and `second[k][d]`.
- Incorrect prompt for second matrix input: The program asks twice for the "number of rows and columns of the first matrix" instead of the second matrix in the second prompt.

2. Number of Breakpoints:

- 2 breakpoints:
  - Fix incorrect array index calculation in the multiplication.
  - Correct the second matrix input prompt.

2(a). Steps to Fix:

- Step 1: Remove `-1` in the indices in the multiplication loop, replacing `first[c-1][c-k]` with `first[c][k]` and `second[k-1][k-d]` with `second[k][d]`.
- Step 2: Correct the prompt to ask for the "number of rows and columns of second matrix."

3. Corrected Code:

```
import java.util.Scanner;

class MatrixMultiplication {
   public static void main(String args[]) {
      int m, n, p, q, sum = 0, c, d, k;

      Scanner in = new Scanner(System.in);
      System.out.println("Enter the number of rows and columns of first
matrix");
      m = in.nextInt();
      n = in.nextInt();

      int first[][] = new int[m][n];

      System.out.println("Enter the elements of first matrix");

      for (c = 0; c < m; c++)
         for (d = 0; d < n; d++)
            first[c][d] = in.nextInt();

      System.out.println("Enter the number of rows and columns of
second matrix");  // Fix: second matrix prompt
      p = in.nextInt();
```

```
        q = in.nextInt();

        if (n != p)
            System.out.println("Matrices with entered orders can't be
multiplied with each other.");
        else {
            int second[][] = new int[p][q];
            int multiply[][] = new int[m][q];

            System.out.println("Enter the elements of second matrix");

            for (c = 0; c < p; c++)
                for (d = 0; d < q; d++)
                    second[c][d] = in.nextInt();

            for (c = 0; c < m; c++) {
                for (d = 0; d < q; d++) {
                    for (k = 0; k < n; k++) {  // Fix: correct indexing for
multiplication
                        sum = sum + first[c][k] * second[k][d];
                    }
                    multiply[c][d] = sum;
                    sum = 0;
                }
            }

            System.out.println("Product of entered matrices:");

            for (c = 0; c < m; c++) {
                for (d = 0; d < q; d++)
                    System.out.print(multiply[c][d] + "\t");
```

```java
            System.out.print("\n");
        }
    }
    in.close();
  }
}
```

<u>Code 7:-</u>

1. Errors Identified:

- Syntax Error: The statement `i + = (i + h / h--) % maxSize;` should be corrected to `i = (i + h * h++) % maxSize;`. This is a misplaced operator and should use `*` for quadratic probing, and the increment of h should be done correctly.
- Logic Error in Rehashing: In the rehashing logic after removal, the statement `currentSize--;` is written twice, which will incorrectly reduce the current size of the hash table.

2. Number of Breakpoints:

- 2 breakpoints:
    - Fix the syntax error in the probing formula.
    - Correct the rehashing logic to avoid decrementing `currentSize` twice.

2(a). Steps to Fix:

Step 1: Replace `i + = (i + h / h--) % maxSize;` with `i = (i + h * h++) % maxSize;` in the `insert` method.

- Step 2: Remove the duplicate `currentSize--;` in the `remove` method.

3. Corrected Code:

```
import java.util.Scanner;

/** Class QuadraticProbingHashTable **/
class QuadraticProbingHashTable {
    private int currentSize, maxSize;
    private String[] keys;
    private String[] vals;

    /** Constructor **/
    public QuadraticProbingHashTable(int capacity) {
        currentSize = 0;
        maxSize = capacity;
        keys = new String[maxSize];
        vals = new String[maxSize];
    }

    /** Function to clear hash table **/
    public void makeEmpty() {
        currentSize = 0;
        keys = new String[maxSize];
        vals = new String[maxSize];
    }

    /** Function to get size of hash table **/
    public int getSize() {
        return currentSize;
    }

    /** Function to check if hash table is full **/
    public boolean isFull() {
        return currentSize == maxSize;
    }

    /** Function to check if hash table is empty **/
    public boolean isEmpty() {
        return getSize() == 0;
    }


        return null;
    }
```

```
/** Function to remove key and its value **/
public void remove(String key) {
    if (!contains(key))
        return;

    /** find position key and delete **/
    int i = hash(key), h = 1;
    while (!key.equals(keys[i]))
        i = (i + h * h++) % maxSize;
    keys[i] = vals[i] = null;

    /** rehash all keys **/
    for (i = (i + h * h++) % maxSize; keys[i] != null; i = (i + h * h++) % maxSize) {
        String tmp1 = keys[i], tmp2 = vals[i];
        keys[i] = vals[i] = null;
        currentSize--;
        insert(tmp1, tmp2);
    }
    // Fix: Remove the     /** Function to check if hash table contains a key **/
public boolean contains(String key) {
    return get(key) != null;
}

/** Function to get hash code of a given key **/
private int hash(String key) {
    return key.hashCode() % maxSize;
}

/** Function to insert key-value pair **/
public void insert(String key, String val) {
    int tmp = hash(key);
    int i = tmp, h = 1;
    do {
        if (keys[i] == null) {
            keys[i] = key;
            vals[i] = val;
            currentSize++;
            return;
        }
        if (keys[i].equals(key)) {
            vals[i] = val;
            return;
        }
        i = (i + h * h++) % maxSize;   // Fix: Corrected probing formula
    } while (i != tmp);
}

/** Function to get value for a given key **/
public String get(String key) {
    int i = hash(key), h = 1;
    while (keys[i] != null) {
        if (keys[i].equals(key))
            return vals[i];
        i = (i + h * h++) % maxSize;
    }
```

Code 8:-

Errors in the Code:

1. Class Name: `Ascending _Order` has a space in the class name, which is invalid. It should be `AscendingOrder`.
2. Condition in Sorting Loop: The loop condition `for (int i = 0; i >= n; i++);` is incorrect. It should be `for (int i = 0; i < n; i++)` to iterate over the array.
3. Incorrect Comparison in Sorting Logic: In the `if` statement `if (a[i] <= a[j])`, it should be `if (a[i] > a[j])` for ascending order sorting.
4. Array Traversal in Output: The last element of the array should be printed after the loop, and there should be no extra `,` after the last element.

Corrected Code:

```
import java.util.Scanner;

public class AscendingOrder {
    public static void main(String[] args) {
        int n, temp;
        Scanner s = new Scanner(System.in);
        System.out.print("Enter no. of elements you want in array: ");
        n = s.nextInt();

        int a[] = new int[n];
        System.out.println("Enter all the elements:");
        for (int i = 0; i < n; i++) {
            a[i] = s.nextInt();
        }

        // Sorting array in ascending order
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                if (a[i] > a[j]) { // Corrected condition for ascending order
                    temp = a[i];
                    a[i] = a[j];
```

```
                a[j] = temp;
            }
        }
    }

    // Display sorted array
    System.out.print("Ascending Order: ");
    for (int i = 0; i < n - 1; i++) {
        System.out.print(a[i] + ", ");
    }
    System.out.print(a[n - 1]); // Print last element without a trailing comma
  }
}
```

Code 9:-

1. Number of Errors Identified:

- Total Errors: 1 error
- Identified Error:
  - Print Loop Issue: The print loop incorrectly iterates until `n - 1`, which could lead to confusion when displaying the last element. Although this does not cause a runtime error, it can result in an incorrect display format if not handled properly.

2. Number of Breakpoints to Fix Errors:

- Total Breakpoints Needed: 1 breakpoint
- Steps to Fix the Identified Error:
  - Change the print loop to correctly display the last element without a trailing comma. Modify the code in the display section as follows:
    - Instead of using `for (int i = 0; i < n - 1; i++)`, simply iterate through all elements and conditionally add a comma after each element except the last.

3. Complete Executable Code:

```
import java.util.Scanner;

public class AscendingOrder {
    public static void main(String[] args) {
        int n, temp;
        Scanner s = new Scanner(System.in);
        System.out.print("Enter no. of elements you want in array: ");
        n = s.nextInt();

        int a[] = new int[n];
        System.out.println("Enter all the elements:");
        for (int i = 0; i < n; i++) {
            a[i] = s.nextInt();
        }

        // Sorting array in ascending order
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                if (a[i] > a[j]) { // Corrected condition for ascending order
                    temp = a[i];
                    a[i] = a[j];
                    a[j] = temp;
                }
            }
        }

        // Display sorted array
        System.out.print("Ascending Order: ");
        for (int i = 0; i < n; i++) { // Updated loop to include all elements
            System.out.print(a[i]);
            if (i < n - 1) { // Print comma only if it's not the last element
                System.out.print(", ");
            }
        }
    }
}
```

Code 10:

1. Errors Identified:

- Incorrect Increment/Decrement Usage:
    - The use of `topN++`, `inter--`, `from+1`, and `to+1` in the recursive calls is incorrect. These expressions do not

modify the values as intended. Instead, they should pass the correct arguments directly without modifying them.

- Incorrect Logic for Recursive Calls:
  - The recursion for moving disks does not properly implement the Tower of Hanoi logic, leading to incorrect moves.
- Missing Semicolon:
  - There's a missing semicolon at the end of the line with `doTowers(...)` inside the else block.

2. Breakpoints Needed:

- Total Breakpoints: You can set breakpoints on the lines where you have the recursive calls and where the output statements are to trace the logic.
- Steps to Fix Errors:
  - Replace `topN++` with `topN - 1` in the recursive calls.
  - Replace `inter--` with `inter` and `from + 1` and `to + 1` with `from` and `to` respectively.
  - Ensure all necessary semicolons are included at the end of statements.

3. Corrected Executable Code:

```
// Tower of Hanoi
public class MainClass {
  public static void main(String[] args) {
    int nDisks = 3; // Number of disks
    doTowers(nDisks, 'A', 'B', 'C'); // A, B and C are names of rods
  }

  public static void doTowers(int topN, char from, char inter, char to) {
    if (topN == 1) {
      System.out.println("Disk 1 from " + from + " to " + to);
    } else {
      // Move topN - 1 disks from source to auxiliary
      doTowers(topN - 1, from, to, inter);
      // Move the largest disk from source to destination
```

```
            System.out.println("Disk " + topN + " from " + from + " to " + to);
            // Move the disks from auxiliary to destination
            doTowers(topN - 1, inter, from, to);
        }
    }
}
```

Part 3:-

Static Analysis:

Excel Sheet provided.

Github Code:

https://github.com/zhangyilang/jpeg2000/blob/master/code/compress.py

```
#coding:utf-8
from PIL import Image
import numpy as np
import cv2
import pywt
import math
import re
import struct

def bgr2rgb(img):
    #把bgr顺序换为rgb顺序
    #此函数同样可以把rgb换成bgr！反正就是第2个和第0个换顺序
    img=img.copy()
    temp=img[:,:,0].copy()
    img[:,:,0]=img[:,:,2].copy()
    img[:,:,2]=temp
    return img
def rgb2bgr(img):
    img=img.copy()
    temp=img[:,:,0].copy()
    img[:,:,0]=img[:,:,2].copy()
    img[:,:,2]=img[:,:,1].copy()
    img[:,:,1]=temp
    return img


class Encoder(object):
    def __init__(self):
        self.C = np.uint32(0)
        self.A = np.uint16(32768)
        self.t = np.uint8(12)
        self.T = np.uint8(0)
        self.L = np.int32(-1)
        self.stream = np.uint8([])
```

```
class Tile(object):
    def __init__(self, tile_image):
        self.tile_image = tile_image
        self.y_tile, self.Cb_tile, self.Cr_tile = None, None, None


class JPEG2000(object):
    """compression algorithm, jpeg2000"""

    def __init__(self, file_path="./test.png", lossy=True, debug=False, tile_size=2**10):
        """
        JPEG2000 algorithm
        Initial parameters:
        file_path: path to image file to be compressed (string)
        quant: include quantization step (boolean)
        lossy: perform lossy compression (boolean)
        debug: whether to debug (boolean)
        tile_size: size of tile, default 1024 (int)
        """
        self.file_path = file_path
        self.debug = debug
        self.lossy = lossy

        # the digits of image
        self.digits = None

        # list of Tile objects of image and tile size
        self.tiles = []
        self.tile_size = tile_size
        self.deTiles = []

        # lossy or lossless compression component transform matrices
        if lossy:
            self.component_transformation_matrix = np.array([[0.2999, 0.587, 0.114],
                [-0.16875, -0.33126, 0.5], [0.5, -0.41869, -0.08131]])
            self.i_component_transformation_matrix = ([[1.0, 0, 1.402], [1.0, -0.34413, -0.71414], [1.0, 1.772, 0]])
        else:
            self.component_transformation_matrix = np.array([[0.25, 0.5, 0.25],
                [0, -1.0, 1.0], [1.0, -1.0, 0]])
            self.i_component_transformation_matrix = ([[1.0, -0.25, -0.25], [1.0, -0.25, 0.75], [1.0, 0.75, -0.25]])

        # Daubechies 9/7coefficients(lossy case)
        self.dec_lo97 = [0, 0.02674875741080976, -0.01686411844287495, -0.07822326652898785, 0.2668641184428723,
                    0.6029490182363579, 0.2668641184428723, -0.07822326652898785, -0.01686411844287495,
                    0.02674875741080976]
        self.dec_hi97 = [0, 0.09127176311424948, -0.05754352622849957, -0.5912717631142470, 1.115087052456994,
                    -0.5912717631142470, -0.05754352622849957, 0.09127176311424948, 0, 0]
        self.rec_lo97 = [0, -0.09127176311424948, -0.05754352622849957, 0.5912717631142470, 1.115087052456994,
                    0.5912717631142470, -0.05754352622849957, -0.09127176311424948, 0, 0]
        self.rec_hi97 = [0, 0.02674875741080976, 0.01686411844287495, -0.07822326652898785, -0.2668641184428723,
                    0.6029490182363579, -0.2668641184428723, -0.07822326652898785, 0.01686411844287495,
                    0.02674875741080976]

        # Le Gall 5/3 coefficients (lossless case)
        self.dec_lo53 = [0, -1/8, 2/8, 6/8, 2/8, -1/8]
        self.dec_hi53 = [0, -1/2, 1, -1/2, 0, 0]
        self.rec_lo53 = [0, 1/2, 1, 1/2, 0, 0]
        self.rec_hi53 = [0, -1/8, -2/8, 6/8, -2/8, -1/8]

        # wavelet
```

```
        self.wavelet = None

        # quantization
        self.quant = lossy
        self.step = 30

    def init_image(self, path):
        """ return the image at path """
        img = cv2.imread(path)
        self.digits = int(re.split(r'([0-9]+)', str(img.dtype))[1])
        return img

    def image_tiling(self, img):
        """
        tile img into square tiles based on self.tile_size (default 1024 * 1024) tiles from bottom and right edges will
        be smaller if image w and h are not divisible by self.tile_size
        """
        tile_size = self.tile_size
        (h, w, d) = img.shape  # size of original image

        # change w and h to be divisible by tile_size
        left_over = w % tile_size
        w += (tile_size - left_over)
        left_over = h % tile_size
        h += (tile_size - left_over)

        # create the tiles by looping through w and h to stop on every pixel that is the top left corner of a tile
        for i in range(0, w, tile_size):  # loop through the width of img, skipping tile_size pixels every time
            for j in range(0, h, tile_size):  # loop through the height of img, skipping tile_size pixels every time
                # add the tile starting at pixel of row j and column i
                tile = Tile(img[j:j + tile_size, i:i + tile_size])
                self.tiles.append(tile)

                # if self.debug:
                #    cv2.imshow("tile" + str(counter), tile.tile_image)
                #    cv2.imwrite("tile " + str(counter) + ".jpg", tile.tile_image)
                #    counter += 1

    def image_splicing(self):

        tile_size = self.tile_size

        h = 0
        w = 0
        for tile in self.deTiles:
            (h_tile, w_tile) = tile.y_coeffs.shape
            h += h_tile
            w += w_tile
        d = 3

        recovered_img = np.empty((h, w, d))
        k = 0
        for i in range(0, w, tile_size):  # loop through the width of img, skipping tile_size pixels every time
            for j in range(0, h, tile_size):  # loop through the height of img, skipping tile_size pixels every time
                recovered_img[j:j + tile_size, i:i + tile_size] = self.deTiles[k].recovered_tile
                k += 1

        bgr_img = np.floor(rgb2bgr(recovered_img))
        cv2.imwrite("recovered_img.jpg", bgr_img, [int(cv2.IMWRITE_JPEG_QUALITY), 100])
        cv2.namedWindow("RECOVERED_IMG")
        RECOVERED_IMG = cv2.imread("recovered_img.jpg")
```

```
        cv2.imshow("RECOVERED_IMG",RECOVERED_IMG)
        cv2.waitKey(0)
        cv2.destroyAllWindows()

    def dc_level_shift(self):
        # dc level shifting
        for t in self.tiles:
            #  normalization for lossy compress
            if self.lossy:
                t.tile_image = t.tile_image.astype(np.float64)
                t.tile_image -= 2 ** (self.digits - 1)
                t.tile_image /= 2 ** self.digits
            # shift for lossless compress
            else:
                t.tile_image -= 2 ** (self.digits - 1)

    def idc_level_shift(self, img):
        # inverse dc level shifting
        for t in self.deTiles:
            if self.lossy:
                t.recovered_tile *= 2 ** self.digits
            t.recovered_tile += 2 ** (self.digits - 1)

    def component_transformation(self):
        """
        Transform every tile in self.tiles from RGB colorspace
        to either YCbCr colorspace (lossy) or YUV colorspace (lossless)
        and save the data for each color component into the tile object
        """
        # loop through tiles
        for tile in self.tiles:
            (h, w, _) = tile.tile_image.shape  # size of tile

            # transform tile to RGB colorspace (library we use to view images uses BGR)
            rgb_tile = cv2.cvtColor(tile.tile_image, cv2.COLOR_BGR2RGB)
            Image_tile = Image.fromarray(rgb_tile, 'RGB')

            # create placeholder matrices for the different colorspace components
            # that are same w and h as original tile
            # tile.y_tile, tile.Cb_tile, tile.Cr_tile = np.empty_like(tile.tile_image), np.empty_like(tile.tile_image),
np.empty_like(tile.tile_image)
            tile.y_tile, tile.Cb_tile, tile.Cr_tile = np.zeros((h, w)), np.zeros((h, w)), np.zeros((h, w))
            # tile.y_tile, tile.Cb_tile, tile.Cr_tile = np.zeros_like(tile.tile_image), np.zeros_like(tile.tile_image), np.zeros_like(tile.tile_image)

            # loop through every pixel and extract the corresponding
            # transformed colorspace values and save in tile object
            for i in range(0, w):
                for j in range(0, h):
                    r, g, b = Image_tile.getpixel((i, j))
                    rgb_array = np.array([r, g, b])
                    if self.lossy:
                        # use irreversible component transformation matrix to transform to YCbCr
                        yCbCr_array = np.matmul(self.component_transformation_matrix, rgb_array)
                    else:
                        # use reversible component transform to get YUV components
                        yCbCr_array = np.matmul(self.component_transformation_matrix, rgb_array)

                    # y = .299 * r + .587 * g + .114 * b
                    # Cb = 0
                    # Cr = 0
                    tile.y_tile[j][i], tile.Cb_tile[j][i], tile.Cr_tile[j][i] = int(yCbCr_array[0]), int(
```

```
                yCbCr_array[1]), int(yCbCr_array[2])
            # tile.y_tile[j][i], tile.Cb_tile[j][i], tile.Cr_tile[j][i] = int(y), int(Cb), int(Cr)

    # if self.debug:
    #    tile = self.tiles[0]
    #    Image.fromarray(tile.y_tile).show()

    #     # Image.fromarray(tile.y_tile).convert('RGB').save("my.jpg")

    #     # cv2.imshow("y_tile", tile.y_tile)
    #     # cv2.imshow("Cb_tile", tile.Cb_tile)
    #     # cv2.imshow("Cr_tile", tile.Cr_tile)
    #     # print tile.y_tile[0]
    #     cv2.waitKey(0)

def i_component_transformation(self):
    """
    Inverse component transformation:
    transform all tile back to RGB colorspace
    """
    # loop through tiles, converting each back to RGB colorspace
    for tile in self.deTiles:
        #(h, w, _) = tile.tile_image.shape  # size of tile
        (h, w) = tile.y_coeffs.shape  # size of tile
        # (h, w) = tile.y_coeffs.shape

        # initialize recovered tile matrix to same size as original 3 dimensional tile
        tile.recovered_tile = np.empty((h,w,3))


        # loop through every pixel of the tile recovered from iDWT and use
        # the YCbCr values (if lossy) or YUV values (is lossless)
        # to transfom back to single RGB tile
        for i in range(0, w):
            for j in range(0, h):
                y, Cb, Cr = tile.y_coeffs[j][i], tile.Cb_coeffs[j][i], tile.Cr_coeffs[j][i]
                yCbCr_array = np.array([y, Cb, Cr])

                if self.lossy:
                    # use irreversible component transform matrix to get back RGB values
                    rgb_array = np.matmul(self.i_component_transformation_matrix, yCbCr_array)
                else:
                    # use reversible component transform to get back RGB values
                    rgb_array = np.matmul(self.i_component_transformation_matrix, yCbCr_array)
                # save all three color dimensions to the given pixel
                tile.recovered_tile[j][i] = rgb_array
        # break

        # if self.debug:
        #     rgb_tile = cv2.cvtColor(tile.recovered_tile, cv2.COLOR_RGB2BGR)
        #     print "rgb_tile.shape: ", rgb_tile.shape
        #     cv2.imshow("tile.recovered_tile", rgb_tile)
        #     cv2.waitKey(0)

def dwt(self):
    """
    Run the 2-DWT (using Haar family) from the pywavelet library
    on every tile and save coefficient results in tile object
    """
    # loop through the tiles
    if self.lossy:
```

```
            self.wavelet = pywt.Wavelet('DB97', [self.dec_lo97, self.dec_hi97, self.rec_lo97, self.rec_hi97])
        else:
            self.wavelet = pywt.Wavelet('LG53', [self.dec_lo53, self.dec_hi53, self.rec_lo53, self.rec_hi53])

        for tile in self.tiles:
            # library function returns a tuple: (cA, (cH, cV, cD)), respectively LL, LH, HH, HL coefficients
            [cA3, (cH3, cV3, cD3), (cH2, cV2, cD2), (cH1, cV1, cD1)] = pywt.wavedec2(tile.y_tile, self.wavelet, level=3)
            tile.y_coeffs = [cA3, (cH3, cV3, cD3), (cH2, cV2, cD2), (cH1, cV1, cD1)]
            [cA3, (cH3, cV3, cD3), (cH2, cV2, cD2), (cH1, cV1, cD1)] = pywt.wavedec2(tile.Cb_tile, self.wavelet, level=3)
            tile.Cb_coeffs = [cA3, (cH3, cV3, cD3), (cH2, cV2, cD2), (cH1, cV1, cD1)]
            [cA3, (cH3, cV3, cD3), (cH2, cV2, cD2), (cH1, cV1, cD1)] = pywt.wavedec2(tile.Cr_tile, self.wavelet, level=3)
            tile.Cr_coeffs = [cA3, (cH3, cV3, cD3), (cH2, cV2, cD2), (cH1, cV1, cD1)]

        if self.debug:
            names = ['cH', 'cV', 'cD']
            tile = self.tiles[2]
            Image.fromarray(tile.y_tile).show()
            for i in range(4):
                if i == 0:
                    cv2.imshow("cA3", tile.y_coeffs[i])
                else:
                    for j in range(3):
                        cv2.imshow(names[j] + str(3-i+1), tile.y_coeffs[i][j])
            cv2.waitKey(0)

    def idwt(self):
        """
        Run the inverse DWT from the pywavelet library on every tile and save the recovered tiles in the tile object
        """
        # loop through tiles
        for tile in self.deTiles:
            tile.y_coeffs = pywt.waverec2(tile.y_Entropy, self.wavelet)
            tile.Cb_coeffs = pywt.waverec2(tile.Cb_Entropy, self.wavelet)
            tile.Cr_coeffs = pywt.waverec2(tile.Cr_Entropy, self.wavelet)

        if self.debug:
            tile = self.tiles[0]
            # print(np.mean(np.abs(tile.y_coeffs - tile.y_tile)))
            Image.fromarray(tile.y_coeffs).show()
            cv2.waitKey(0)

    def quantization_math(self, img):
        """
        Quantize img: for every coefficient in img,
        save the original sign and decrease number of
        decimals saved by flooring the absolute value
        of the coeffcient divided by the step size
        """
        # initialize array to hold quantized coefficients,
        # to be same size as img
        if('tuple' in str(type(img))):
            #imgCount=0
            quantization_img=[]
            for everyImg in img:
                #imgCount+=1
                quantization_img.append(self.quantization_math(everyImg))
            return(tuple(quantization_img))
        else:
            (h, w) = img.shape
            quantization_img = np.empty_like(img)
```

```
        # loop through every coefficient in img
        for i in range(0, w):
            for j in range(0, h):
                # save the sign
                if img[j][i] >= 0:
                    sign = 1
                else:
                    sign = -1
                # save quantized coeffcicient
                quantization_img[j][i] = sign * math.floor(abs(img[j][i]) / self.step)
        return quantization_img

def i_quantization_math(self, img):
    """
    Inverse quantization of img: un-quantize
    the quantized coefficients in img by
    multiplying the coeffs by the step size
    """
    if('tuple' in str(type(img))):
        #imgCount=0
        i_quantization_img=[]
        for everyImg in img:
            #imgCount+=1
            i_quantization_img.append(self.i_quantization_math(everyImg))
        return(tuple(i_quantization_img))
    else:
        # initialize array to hold un-quantized coefficients
        # to be same size as img
        (h, w) = img.shape
        i_quantization_img = np.empty_like(img)

        # loop through ever coefficient in img
        for i in range(0, w):
            for j in range(0, h):
                # save un-quantized coefficient
                i_quantization_img[j][i] = img[j][i] * self.step
        return i_quantization_img

def quantization_helper(self, img):
    """
    Quantize the 4 different data arrays representing
    the 4 different coefficient approximations/details
    """
    cA = self.quantization_math(img[0])
    cH = self.quantization_math(img[1])
    cV = self.quantization_math(img[2])
    cD = self.quantization_math(img[3])

    return cA, cH, cV, cD

def i_quantization_helper(self, img):
    """
    Un-quantize the 4 different data arrays representing
    the 4 different coefficient approximations/details
    """
    cA = self.i_quantization_math(img[0])
    cH = self.i_quantization_math(img[1])
    cV = self.i_quantization_math(img[2])
    cD = self.i_quantization_math(img[3])

    return cA, cH, cV, cD
```

```
def quantization(self):
    """
    Quantize the tiles, saving the quantized
    information to the tile object
    """
    for tile in self.tiles:
        # quantize the tile in all 3 colorspaces
        tile.y_coeffs = self.quantization_helper(tile.y_coeffs)
        tile.Cb_coeffs = self.quantization_helper(tile.Cb_coeffs)
        tile.Cr_coeffs = self.quantization_helper(tile.Cr_coeffs)

def i_quantization(self):
    """
    Un-quantize the tiles, saving the un-quantized
    information to the tile object
    """
    for tile in self.deTiles:
        tile.y_Entropy = self.i_quantization_helper(tile.y_Entropy)
        tile.Cb_Entropy = self.i_quantization_helper(tile.Cb_Entropy)
        tile.Cr_Entropy = self.i_quantization_helper(tile.Cr_Entropy)

def image_entropy(self):
    bitcode = []
    streamonly = []
    for oneTile in self.tiles:
        newBit, newStream = self.tile_entropy(oneTile)
        bitcode = np.hstack((bitcode, newBit))
        streamonly = np.hstack((streamonly, newStream))
    bitcode = [int(i) for i in bitcode]
    l = len(bitcode)
    with open('test.bin', 'wb') as f:
        f.write(struct.pack(str(l)+'i', *bitcode))
    streamonly = [int(i) for i in streamonly]
    l = len(streamonly)
    with open('streamonly.bin', 'wb') as f:
        f.write(struct.pack(str(l)+'i', *streamonly))

def tile_entropy(self, tile, h=64, w=64):
    tile_cA = tile.y_coeffs[0]
    # np.save("tile0.npy",(tile.y_coeffs,tile.Cb_coeffs,tile_cA))
    newBit, newStream = self.band_entropy(tile_cA, 'LL', h, w)
    bitcode = newBit
    streamOnly = newStream
    for i in range(1,4):
        temp_tile = tile.y_coeffs[i]
        newBit, newStream = self.band_entropy(temp_tile[0], 'LH', h, w)
        bitcode = np.hstack((bitcode, newBit))
        streamOnly = np.hstack((streamOnly, newStream))
        newBit, newStream = self.band_entropy(temp_tile[1], 'HL', h, w)
        bitcode = np.hstack((bitcode, newBit))
        streamOnly = np.hstack((streamOnly, newStream))
        newBit, newStream = self.band_entropy(temp_tile[2], 'HH', h, w)
        bitcode = np.hstack((bitcode, newBit))
        streamOnly = np.hstack((streamOnly, newStream))
    tile_cA = tile.Cb_coeffs[0]
    newBit, newStream = self.band_entropy(tile_cA, 'LL', h, w)
    bitcode = np.hstack((bitcode,newBit))
    streamOnly = np.hstack((streamOnly, newStream))
    for i in range(1,4):
        temp_tile = tile.Cb_coeffs[i]
```

```
            newBit, newStream = self.band_entropy(temp_tile[0], 'LH', h, w)
            bitcode = np.hstack((bitcode, newBit))
            streamOnly = np.hstack((streamOnly, newStream))
            newBit, newStream = self.band_entropy(temp_tile[1], 'HL', h, w)
            bitcode = np.hstack((bitcode, newBit))
            streamOnly = np.hstack((streamOnly, newStream))
            newBit, newStream = self.band_entropy(temp_tile[2], 'HH', h, w)
            bitcode = np.hstack((bitcode, newBit))
            streamOnly = np.hstack((streamOnly, newStream))
        tile_cA = tile.Cr_coeffs[0]
        newBit, newStream = self.band_entropy(tile_cA, 'LL', h, w)
        bitcode = np.hstack((bitcode,newBit))
        streamOnly = np.hstack((streamOnly, newStream))
        for i in range(1,4):
            temp_tile = tile.Cr_coeffs[i]
            newBit, newStream = self.band_entropy(temp_tile[0], 'LH', h, w)
            bitcode = np.hstack((bitcode, newBit))
            streamOnly = np.hstack((streamOnly, newStream))
            newBit, newStream = self.band_entropy(temp_tile[1], 'HL', h, w)
            bitcode = np.hstack((bitcode, newBit))
            streamOnly = np.hstack((streamOnly, newStream))
            newBit, newStream = self.band_entropy(temp_tile[2], 'HH', h, w)
            bitcode = np.hstack((bitcode, newBit))
            streamOnly = np.hstack((streamOnly, newStream))
        bitcode = np.hstack((bitcode, [2051]))
        return (bitcode, streamOnly)

    def band_entropy(self, tile, bandMark, h=64, w=64, num=8):
        # 码流:[h, w, CX1, 2048, stream1, 2048, ..., CXn, streamn, 2048, 2049,CXn+1, streamn+1, 2048, ...,2050]
        (h_cA, w_cA) = np.shape(tile)
        h_left_over = h_cA % h
        w_left_over = w_cA % w
        cA_extend = np.pad(tile, ((0,h-h_left_over), (0,w-w_left_over)), 'constant')
        bitcode = [h_cA, w_cA]
        streamOnly = []
        for i in range(0, h_cA, h):
            for j in range(0, w_cA, w):
                codeBlock = cA_extend[i:i + h, j:j + w]
                CX, D = self.codeBlockfun(codeBlock, bandMark, h, w, num)
                encoder = self.entropy_coding(CX, D)
                bitcode = np.hstack((bitcode, CX.flatten(), [2048], encoder.stream, [2048]))
                streamOnly = np.hstack((streamOnly, encoder.stream))
            bitcode = np.hstack((bitcode, [2049]))
        bitcode = np.hstack((bitcode, [2050]))
        return (bitcode, streamOnly)

    def image_deEntropy(self):
        # bitcode = np.load('jpeg2k.npy')
        bitcode = []
        with open('test.bin', 'rb') as f:
            while True:
                tmp = f.read(4)
                if not tmp:
                    break
                bitcode.append(*struct.unpack('i', tmp))
        while bitcode.__len__() != 0:
            _index = bitcode.index(2051)
            self.deTiles.append(self.tile_deEntropy(bitcode[0:_index+1]))
            if bitcode.__len__() > _index+1:
                bitcode = bitcode[_index+1:]
            else:
```

```
            bitcode = []

    def tile_deEntropy(self, codestream):
        temp = []
        tile = Tile(None)
        for i in range(0, 30):
            _index = codestream.index(2050)
            deStream = codestream[0:_index+1]
            temp.append(self.band_deEntropy(deStream))
            codestream = codestream[_index+1:]
        tile.y_Entropy = [temp[0],(temp[1],temp[2],temp[3]),(temp[4],temp[5],temp[6]),(temp[7], temp[8],temp[9])]
        tile.Cb_Entropy = [temp[10],(temp[11],temp[12],temp[13]),(temp[14],temp[15],temp[16]),(temp[17], temp[18],temp[19])]
        tile.Cr_Entropy = [temp[20],(temp[21],temp[22],temp[23]),(temp[24],temp[25],temp[26]),(temp[27], temp[28],temp[29])]
        return tile

    def band_deEntropy(self, codestream, h=64, w=64, num=8):
        h_cA = codestream[0]
        w_cA = codestream[1]
        codestream = codestream[2:]
        h_num = h_cA//h + 1
        w_num = w_cA//w + 1
        band_extend = np.zeros((h_num * h, w_num * w))
        for i in range(0, h_num):
            for j in range(0, w_num):
                _index = codestream.index(2048)
                deCX = codestream[0:_index]
                deCX = np.resize(deCX, (_index+1,1))
                codestream = codestream[_index+1:]
                _index = codestream.index(2048)
                deStream = codestream[0:_index]
                codestream = codestream[_index+1:]
                decodeD = self.entropy_decoding(deStream, deCX)
                band_extend[i*h:(i+1)*h,j*w:(j+1)*w] = self.decodeBlock(decodeD, deCX, h, w, num)
            if codestream[0] != 2049:
                print("Error!")
            codestream = codestream[1:]
        if codestream[0]!= 2050:
            print("Error!")
        return band_extend[0:h_cA, 0:w_cA]

    def codeBlockfun(self, codeBlock, bandMark, h=64, w=64, num=8):
        S1 = np.zeros((h, w))
        S2 = np.zeros((h, w))
        S3 = np.zeros((h, w))
        signs = (- np.sign(codeBlock) + 1) //2  # positive: 0, negative: 1
        unsigned = np.asarray(np.abs(codeBlock), dtype=np.uint8)
        bitPlane = np.unpackbits(unsigned).reshape((h, w, 8))# bitPlane[i][j][0] is the most important bit
        bitPlane = np.transpose(bitPlane,(2,0,1))
        # For Test
        """
        signs = np.zeros((8,8))
        bitPlane = np.zeros((2,8,8))
        bitPlane[0][1][1] = 1
        bitPlane[0][4][4] = 1
        bitPlane[1][0][2] = 1
        bitPlane[1][1] = np.array([0,1,0,0,1,1,0,0])
        bitPlane[1][2][2] = 1
        bitPlane[1][3][3] = 1
        bitPlane[1][4][5] = 1
        bitPlane[1][5] = np.array([0,0,0,0,1,1,0,1])
        bitPlane[1][6][6] = 1
```

```python
    """
    CX = np.zeros((100000, 1), dtype=np.uint8)
    D = np.zeros((100000, 1), dtype=np.uint8)
    pointer = 0
    for i in range(num):
        D, CX, S1, S3, pointer = self.SignifiancePropagationPass(D, CX, S1, S3, pointer, bitPlane[i], bandMark, signs, w, h)
        D, CX, S2, pointer = self.MagnitudeRefinementPass(D, CX, S1, S2, S3, pointer, bitPlane[i], w, h)
        D, CX, pointer, S1 = self.CLeanUpPass(D, CX, S1, S3, pointer, bitPlane[i], bandMark, signs, w, h)
        S3 = np.zeros((h, w))
    CX_final = CX[0:pointer]
    D_final = D[0:pointer]
    return CX_final, D_final

def put_byte(self, encoder):
    # 将T中的内容写入字节缓存
    if encoder.L >= 0:
        encoder.stream = np.append(encoder.stream, encoder.T)
    encoder.L = encoder.L + 1
    return encoder

def transfer_byte(self, encoder):
    CPartialMask = np.uint32(133693440)
    CPartialCmp = np.uint32(4161273855)
    CMsbsMask = np.uint32(267386880)
    CMsbsCmp = np.uint32(4027580415) # CMsbs的补码
    CCarryMask = np.uint32(2**27)
    if encoder.T == 255:
        # 不能将任何进位传给T
        encoder = self.put_byte(encoder)
        encoder.T = np.uint8((encoder.C & CMsbsMask)>>20)
        encoder.C = encoder.C & CMsbsCmp
        encoder.t = 7
    else:
        # 从C将任何进位传到T
        encoder.T = encoder.T + np.uint8((encoder.C & CCarryMask)>>27)
        encoder.C = encoder.C ^ CCarryMask
        encoder = self.put_byte(encoder)
        if encoder.T == 255:
            encoder.T = np.uint8((encoder.C & CMsbsMask)>>20)
            encoder.C = encoder.C & CMsbsCmp
            encoder.t = 7
        else:
            encoder.T = np.uint8((encoder.C & CPartialMask)>>19)
            encoder.C = encoder.C & CPartialCmp
            encoder.t = 8
    return encoder

def encode_end(self, encoder):
    nbits = 27-15-encoder.t
    encoder.C = encoder.C * np.uint32(2**encoder.t)
    while nbits > 0:
        encoder = self.transfer_byte(encoder)
        nbits = nbits - encoder.t
        encoder.C = encoder.C * np.uint32(2**encoder.t)
    encoder = self.transfer_byte(encoder)
    return encoder

def entropy_coding(self, CX, D):
    PETTable = np.load(r"PETTable.npy")
    CXTable = np.load(r"CX_Table.npy")
    encoder = Encoder()
```

```
    for i in range(D.__len__()):
        symbol = D[i][0]
        cxLabel = CX[i][0]
        expectedSymbol = CXTable[cxLabel][1]
        p = PETTable[CXTable[cxLabel][0]][3]
        encoder.A = encoder.A - p
        if encoder.A < p:
            # Conditional exchange of MPS and LPS
            expectedSymbol = 1-expectedSymbol
        if symbol == expectedSymbol:
            # assign MPS the upper sub-interval
            encoder.C = encoder.C + np.uint32(p)
        else:
            # assign LPS the lower sub-interval
            encoder.A = np.uint32(p)
        if encoder.A < 32768:
            if symbol == CXTable[cxLabel][1]:
                CXTable[cxLabel][0] = PETTable[CXTable[cxLabel][0]][0]
            else:
                CXTable[cxLabel][1] = CXTable[cxLabel][1]^PETTable[CXTable[cxLabel][0]][2]
                CXTable[cxLabel][0] = PETTable[CXTable[cxLabel][0]][1]
            while encoder.A < 32768:
                encoder.A = 2 * encoder.A
                encoder.C = 2 * encoder.C
                encoder.t = encoder.t-1
                if encoder.t == 0:
                    encoder = self.transfer_byte(encoder)
    encoder = self.encode_end(encoder)
    return encoder

def fill_lsb(self, encoder):
    encoder.t = 8
    if encoder.L==encoder.stream.__len__() or \
            (encoder.T == 255 and encoder.stream[encoder.L]>143):
        encoder.C = encoder.C + 255
    else:
        if encoder.T == 255:
            encoder.t = 7
        encoder.T = encoder.stream[encoder.L]
        encoder.L = encoder.L + 1
        encoder.C = encoder.C + np.uint32((encoder.T)<<(8-encoder.t))
    return encoder

def entropy_decoding(self, stream, CX):
    PETTable = np.load(r"PETTable.npy")
    CXTable = np.load(r"CX_Table.npy")
    encoder = Encoder()
    encoder.A = np.uint16(0)
    encoder.C = np.uint32(0)
    encoder.t = np.uint8(0)
    encoder.T = np.uint8(0)
    encoder.L = np.int32(0)
    encoder.stream = stream
    encoder = self.fill_lsb(encoder)
    encoder.C = encoder.C<<encoder.t
    encoder = self.fill_lsb(encoder)
    encoder.C = encoder.C << 7
    encoder.t = encoder.t - 7
    encoder.A = np.uint16(2**15)
    CActiveMask = np.uint32(16776960)
    CActiveCmp = np.uint32(4278190335)
```

```
    decodeD = []
    for i in range(CX.__len__()):
        cxLabel = CX[i][0]
        expectedSymbol = CXTable[cxLabel][1]
        p = PETTable[CXTable[cxLabel][0]][3]
        encoder.A = encoder.A - np.uint16(p)
        if encoder.A < np.uint16(p):
            expectedSymbol = 1-expectedSymbol
        if ((encoder.C & CActiveMask)>>8) < p:
            symbol = 1 - expectedSymbol
            encoder.A = np.uint16(p)
        else:
            symbol = expectedSymbol
            temp = ((encoder.C & CActiveMask)>>8) - np.uint32(p)
            encoder.C = encoder.C & CActiveCmp
            encoder.C = encoder.C + np.uint32((np.uint32(temp<<8)) & CActiveMask)
        if encoder.A < 2**15:
            if symbol == CXTable[cxLabel][1]:
                CXTable[cxLabel][0] = PETTable[CXTable[cxLabel][0]][0]
            else:
                CXTable[cxLabel][1] = CXTable[cxLabel][1]^PETTable[CXTable[cxLabel][0]][2]
                CXTable[cxLabel][0] = PETTable[CXTable[cxLabel][0]][1]
            while encoder.A < 2**15:
                if encoder.t == 0:
                    encoder = self.fill_lsb(encoder)
                encoder.A = 2 * encoder.A
                encoder.C = 2 * encoder.C
                encoder.t = encoder.t - 1
        decodeD.append([symbol])
    return decodeD

def RunLengthDecoding(self, CX, D):
    n = CX.__len__()
    wrong = 1
    if CX[0][0] == 17 and D[0][0] == 0 or CX[0][0] == 17 and CX[1][0] == 18 and CX[2][0] == 18 and D[0][0] == 1:
        wrong = 0
    if wrong == 0:
        if D[0][0] == 0:
            deLen = 4
            V = [0, 0, 0, 0]
        elif D[0][0] == 1 and D[1][0] == 0 and D[2][0] == 0:
            deLen = 1
            V = [1]
        elif D[0][0] == 1 and D[1][0] == 0 and D[2][0] == 1:
            deLen = 2
            V = [0,1]
        elif D[0][0] == 1 and D[1][0] == 1 and D[2][0] == 0:
            deLen = 3
            V = [0,0,1]
        elif D[0][0] == 1 and D[1][0] == 1 and D[2][0] == 1:
            deLen = 4
            V = [0,0,0,1]
        else:
            try:
                raise ValidationError('RunLengthDecoding: D not valid')
            except ValidationError as e:
                print(e.args)
                deLen = -1
                V = [-1]
    else:
        try:
```

```
            raise ValidationError('RunLengthDecoding: CX not valid')
        except ValidationError as e:
            print(e.args)
            deLen = -1
            V = [-1]
    return deLen, V

def SignDecoding(self, D, CX, neighbourS1):
    if neighbourS1.__len__() == 3 and neighbourS1[0].__len__() == 3:
        hstr = str(int(neighbourS1[1][0])) + str(int(neighbourS1[1][2]))
        vstr = str(int(neighbourS1[0][1])) + str(int(neighbourS1[2][1]))
        dict = {'00': 0, '1-1': 0, '-11': 0, '01': 1, '10': 1, '11': 1,
                '0-1': -1, '-10': -1, '-1-1': -1}
        h = dict[hstr]
        v = dict[vstr]
        hAndv = str(h) + str(v)
        hv2Sign = {'11': 0, '10': 0, '1-1': 0, '01': 0, '00': 0,
                   '0-1': 1, '-11': 1, '-10': 1, '-1-1': 1}
        hv2Context = {'11': 13, '10': 12, '1-1': 11, '01': 10, '00': 9,
                      '0-1': 10, '-11': 11, '-10': 12, '-1-1': 13}
        temp = hv2Sign[hAndv]
        deCX = hv2Context[hAndv]
        if deCX == CX:
            deSign = D[0]^temp
        else:
            try:
                raise ValidationError('SignDecoding: Context does not match. Error occurs.')
            except ValidationError as e:
                print(e.args)
                deSign = -1
    else:
        try:
            raise ValidationError('SignDecoding: Size of neighbourS1 not valid')
        except ValidationError as e:
            print(e.args)
            deSign = -1
    return deSign

def SignificancePassDecoding(self, V, D, CX, deS1, deS3, pointer, signs, w=64, h=64 ):
    S1extend = np.pad(deS1, ((1,1), (1,1)), 'constant')
    rounds = h // 4
    for i in range(rounds):
        for col in range(w):
            for ii in range(4):
                row = 4*i + ii
                temp = np.sum(S1extend[row:row+3,col:col+3])-S1extend[row+1][col+1]
                if deS1[row][col] != 0 or temp ==0:
                    continue
                V[row][col] = D[pointer][0]
                pointer = pointer + 1
                deS3[row][col] = 1
                if V[row][col] == 1:
                    signs[row][col] = self.SignDecoding(D[pointer], CX[pointer], S1extend[row:row+3,col:col+3])
                    pointer = pointer + 1
                    deS1[row][col]=1
                    S1extend = np.pad(deS1, ((1,1), (1,1)), 'constant')
    return V, signs, deS1, deS3, pointer

def MagnitudePassDecoding(self, V, D, deS1, deS2, deS3, pointer, w=64, h=64):
    rounds = h // 4
    for i in range(rounds):
```

```
        for col in range(w):
            for ii in range(4):
                row = 4*i + ii
                if deS1[row][col] != 1 or deS3[row][col] != 0:
                    continue
                V[row][col] = D[pointer][0]
                pointer = pointer + 1
                deS2[row][col] = 1
    return V, deS2, pointer

def CleanPassDecoding(self, V, D, CX, deS1, deS3, pointer, signs, w=64, h=64):
    S1extend = np.pad(deS1, ((1,1), (1,1)), 'constant')
    rounds = h // 4
    for i in range(rounds):
        for col in range(w):
            ii = 0
            row = 4*i
            tempSum = np.sum(S1extend[row:row+6,col:col+3]) + np.sum(deS3[row:row+4,col])
            # 整一列未被编码，都为非重要，且领域非重要
            if tempSum == 0:
                if CX.__len__() < pointer +3:
                    CXextend = np.pad(CX,(0,2), 'constant')
                    Dextend = np.pad(D, (0,2), 'constant')
                    tempCx = CXextend[pointer:pointer+3]
                    tempD = Dextend[pointer:pointer+3]
                else:
                    tempCx = CX[pointer:pointer+3]
                    tempD = D[pointer:pointer+3]
                ii, tempV = self.RunLengthDecoding(tempCx, tempD)
                if tempV == [0,0,0,0]:
                    V[row][col] = 0
                    V[row+1][col] = 0
                    V[row+2][col] = 0
                    V[row+3][col] = 0
                    pointer = pointer + 1
                else:
                    if tempV == [1]:
                        V[row][col] = 1
                        pointer = pointer + 3
                    elif tempV ==[0, 1]:
                        V[row][col] = 0
                        V[row+1][col] = 1
                        pointer = pointer + 3
                    elif tempV ==[0, 0, 1]:
                        V[row][col] = 0
                        V[row+1][col] = 0
                        V[row+2][col] = 1
                        pointer = pointer + 3
                    elif tempV ==[0, 0, 0, 1]:
                        V[row][col] = 0
                        V[row+1][col] = 0
                        V[row+2][col] = 0
                        V[row+3][col] = 1
                        pointer = pointer + 3
                    # sign coding
                    row = row + ii - 1
                    signs[row][col] = self.SignDecoding(D[pointer], CX[pointer], S1extend[row:row+3,col:col+3])
                    pointer = pointer + 1
                    deS1[row][col]=1
                    S1extend = np.pad(deS1, ((1,1), (1,1)), 'constant')
            while ii < 4:
```

```
            row = i*4 + ii
            ii = ii + 1
            if deS1[row][col] != 0 or deS3[row][col] != 0:
                continue
            V[row][col] = D[pointer][0]
            pointer = pointer + 1
            deS3[row][col] = 1
            if V[row][col] == 1:
                signs[row][col] = self.SignDecoding(D[pointer], CX[pointer], S1extend[row:row+3,col:col+3])
                pointer = pointer + 1
                deS1[row][col]=1
                S1extend = np.pad(deS1, ((1,1), (1,1)), 'constant')
    return V, deS1, deS3, signs, pointer

def decodeBlock(self, D, CX, h=64, w=64, num=8):
    deS1 = np.uint8(np.zeros((h, w)))
    deS2 = np.uint8(np.zeros((h, w)))
    deS3 = np.uint8(np.zeros((h, w)))
    signs = np.uint8(np.zeros((h,w)))
    V = np.uint8(np.zeros((num, h, w)))
    deCode = np.zeros((h,w))
    pointer = 0
    for i in range(num):
        V[i,:,:], signs, deS1, deS3, pointer = self.SignificancePassDecoding(V[i,:,:], D, CX, deS1, deS3, pointer, signs, w, h)
        V[i,:,:], deS2, pointer = self.MagnitudePassDecoding(V[i,:,:], D, deS1, deS2, deS3, pointer, w,h)
        V[i,:,:], deS1, deS3, signs, pointer = self.CleanPassDecoding(V[i,:,:], D, CX, deS1, deS3, pointer, signs, w,h)
        deS3 = np.zeros((h, w))
    V = np.transpose(V,(1,2,0))
    V = np.packbits(V).reshape((h, w))
    for i in range(h):
        for j in range(w):
            deCode[i][j] = (1-2*signs[i][j]) * V[i][j]
    return deCode

def bit_stream_formation(self, img):
    # idk if we need this or what it is
    pass

def forward(self):
    """
    Run the forward transformations to compress img
    """
    img = self.init_image(self.file_path)
    self.image_tiling(img)
    # self.dc_level_shift()
    self.component_transformation()
    self.dwt()
    if self.quant:
        self.quantization()
    self.image_entropy()

def backward(self):
    """
    Run the backwards transformations to get the image back
    from the compressed data
    """
    self.image_deEntropy()
    if self.quant:
        self.i_quantization()
    self.idwt()
    self.i_component_transformation()
```

```
    # self.idc_level_shift()
    self.image_splicing()

def run(self):
    """
    Run forward and backward transformations, saving
    compressed image data and reconstructing the image
    from the compressed data
    """
    self.forward()
    self.backward()

def MagnitudeRefinementCoding(self, neighbourS1, s2):
    # input neighbourS1: size 3*3, matrix of significance
    # input s2: whether it is the first time for Magnitude Refinement Coding
    # output: context
    if neighbourS1.__len__() == 3 and neighbourS1[0].__len__() == 3:
        temp = np.sum(neighbourS1)-neighbourS1[1][1]
        if s2 == 1:
            cx = 16
        elif s2 == 0 and temp >= 1:
            cx = 15
        else:
            cx = 14
    else:
        try:
            raise ValidationError('MagnitudeRefinementCoding: Size of neighbourS1 not valid')
        except ValidationError as e:
            print(e.args)
            cx = -1
    return cx

def SignCoding(self, neighbourS1, sign):
    # input neighbourS1: size 3*3, matrix of significance
    # input sign
    # output: signComp,(equal: 0, not equal: 1) context
    if neighbourS1.__len__() == 3 and neighbourS1[0].__len__() == 3:
        hstr = str(int(neighbourS1[1][0])) + str(int(neighbourS1[1][2]))
        vstr = str(int(neighbourS1[0][1])) + str(int(neighbourS1[2][1]))
        dict = {'00': 0, '1-1': 0, '-11': 0, '01': 1, '10': 1, '11': 1,
                '0-1': -1, '-10': -1, '-1-1': -1}
        h = dict[hstr]
        v = dict[vstr]
        hAndv = str(h) + str(v)
        hv2Sign = {'11': 0, '10': 0, '1-1': 0, '01': 0, '00': 0,
                   '0-1': 1, '-11': 1, '-10': 1, '-1-1': 1}
        hv2Context = {'11': 13, '10': 12, '1-1': 11, '01': 10, '00': 9,
                      '0-1': 10, '-11': 11, '-10': 12, '-1-1': 13}
        signPredict = hv2Sign[hAndv]
        context = hv2Context[hAndv]
        signComp = int(sign) ^ signPredict
    else:
        try:
            raise ValidationError('SignCoding: Size of neighbourS1 not valid')
        except ValidationError as e:
            print(e.args)
            signComp = -1
            context = -1
    return signComp, context

def ZeroCoding(self, neighbourS1, bandMark):
```

```
# input neighbourS1: size 3*3, matrix of significance
# input s2: whether it is the first time for Magnitude Refinement Coding
# output: context
if neighbourS1.__len__() == 3 and neighbourS1[0].__len__() == 3:
    h = neighbourS1[1][0] + neighbourS1[1][2]
    v = neighbourS1[0][1] + neighbourS1[2][1]
    d = neighbourS1[0][0] + neighbourS1[0][2] + neighbourS1[2][0] + neighbourS1[2][2]
    if bandMark == 'LL'or bandMark == 'LH':
        if h == 2:
            cx = 8
        elif h == 1 and v >= 1:
            cx = 7
        elif h == 1 and v == 0 and d >= 1:
            cx = 6
        elif h == 1 and v == 0 and d == 0:
            cx = 5
        elif h == 0 and v == 2:
            cx = 4
        elif h ==0 and v ==1:
            cx = 3
        elif h ==0 and v == 0 and d >=2:
            cx = 2
        elif h ==0 and v == 0 and d ==1:
            cx = 1
        else:
            cx = 0
    elif bandMark == 'HL':
        if v == 2:
            cx = 8
        elif v == 1 and h >= 1:
            cx = 7
        elif v == 1 and h == 0 and d >= 1:
            cx = 6
        elif v == 1 and h == 0 and d == 0:
            cx = 5
        elif v == 0 and h == 2:
            cx = 4
        elif v ==0 and h ==1:
            cx = 3
        elif v ==0 and h == 0 and d >=2:
            cx = 2
        elif v ==0 and h == 0 and d ==1:
            cx = 1
        else:
            cx = 0
    elif bandMark == 'HH':
        hPlusv = h + v
        if d >= 3:
            cx = 8
        elif d == 2 and hPlusv >= 1:
            cx = 7
        elif d == 2 and hPlusv == 0:
            cx = 6
        elif d == 1 and hPlusv >= 2:
            cx = 5
        elif d == 1 and hPlusv == 1:
            cx = 4
        elif d == 1 and hPlusv == 0:
            cx = 3
        elif d == 0 and hPlusv >= 2:
            cx = 2
```

```
        elif d == 0 and hPlusv == 1:
            cx = 1
        else:
            cx = 0
    else:
        try:
            raise ValidationError('ZeroCoding: bandMark not valid')
        except ValidationError as e:
            print(e.args)
            cx = -1
else:
    try:
        raise ValidationError('ZeroCoding: Size of neighbourS1 not valid')
    except ValidationError as e:
        print(e.args)
        cx = -1
return cx

def RunLengthCoding(self, listS1):
    # input listS1: size 1*4, list of significance
    # output n: number of elements encoded
    # output d: 0 means the RunLengthCoding does not end.
    # [1, x, x] means the RunLengthCoding ends and the position is indicated.
    # output cx: context
    if listS1.__len__() == 4:
        if listS1[0]==0 and listS1[1]==0 and listS1[2]==0 and listS1[3]==0:
            n = 4
            d = [0]
            cx = [17]
        elif listS1[0] == 1:
            n = 1
            d = [1, 0, 0]
            cx = [17, 18, 18]
        elif listS1[0] == 0 and listS1[1] == 1:
            n = 2
            d = [1, 0, 1]
            cx = [17, 18, 18]
        elif listS1[0] == 0 and listS1[1] == 0 and listS1[2] == 1:
            n = 3
            d = [1, 1, 0]
            cx = [17, 18, 18]
        elif listS1[0] == 0 and listS1[1] == 0 and listS1[2] == 0 and listS1[3] == 1:
            n = 4
            d = [1, 1, 1]
            cx = [17, 18, 18]
        else:
            try:
                raise ValidationError('RunLengthCoding: listS1 not valid')
            except ValidationError as e:
                print(e.args)
                n, d, cx = 0, -1, -1
    else:
        try:
            raise ValidationError('RunLengthCoding: length of listS1 not valid')
        except ValidationError as e:
            print(e.args)
            n, d, cx = 0, -1, -1
    return n, d, cx

def SignifiancePropagationPass(self, D, CX, S1, S3, pointer, plane, bandMark, signs, w=64, h=64):
    # input S1:  list of significance, size 64*64
```

```
    # input CX: the list of context
    # plane: the value of bits at this plane
    # bandMark: LL, HL, HH, or LH
    # pointer: the pointer of the CX
    # S3: denote that the element has been coded
    # output: D, CX, S1, S3, pointer
    S1extend = np.pad(S1, ((1,1), (1,1)), 'constant')
    rounds = h // 4
    for i in range(rounds):
        for col in range(w):
            for ii in range(4):
                row = 4*i + ii
                if S1[row][col] != 0:
                    continue # is significant
                temp = S1extend[row][col] + S1extend[row+1][col] + S1extend[row+2][col] + S1extend[row][col+1] + \
                    S1extend[row+2][col+1] + S1extend[row][col+2] + S1extend[row+1][col+2] + S1extend[row+2][col+2]
                if temp == 0:
                    continue # is insignificant
                tempCx = self.ZeroCoding(S1extend[row:row+3,col:col+3], bandMark)
                D[pointer][0] = plane[row][col]
                CX[pointer][0] = tempCx
                pointer = pointer + 1
                S3[row][col] = 1  # mark that plane[row][col] has been coded
                if plane[row][col] ==1:  # signcoding
                    signComp, tempCx = self.SignCoding(S1extend[row:row+3,col:col+3], signs[row][col])
                    D[pointer][0] = signComp
                    CX[pointer][0] = tempCx
                    pointer = pointer + 1
                    S1[row][col] = 1 # mark as significant
                    S1extend = np.pad(S1, ((1,1), (1,1)), 'constant')
    return D, CX, S1, S3, pointer

def MagnitudeRefinementPass(self, D, CX, S1, S2, S3, pointer, plane, w=64, h=64):
    S1extend = np.pad(S1, ((1,1), (1,1)), 'constant')
    rounds = h // 4
    for i in range(rounds):
        for col in range(w):
            for ii in range(4):
                row = 4*i + ii
                if S1[row][col] != 1 or S3[row][col] != 0:
                    continue
                tempCx = self.MagnitudeRefinementCoding(S1extend[row:row+3,col:col+3], S2[row][col])
                S2[row][col] = 1  # Mark that the element has been refined
                D[pointer][0] = plane[row][col]
                CX[pointer][0] = tempCx
                pointer = pointer + 1
    return D, CX, S2, pointer

def CLeanUpPass(self,D, CX, S1, S3, pointer, plane, bandMark, signs, w=64, h=64):
    S1extend = np.pad(S1, ((1,1), (1,1)), 'constant')
    rounds = h // 4
    for i in range(rounds):
        for col in range(w):
            ii = 0
            row = 4 * i
            tempSum = np.sum(S1extend[row:row+6,col:col+3]) + np.sum(S3[row:row+4,col])
            # 整一列未被编码，都为非重要，且领域非重要
            if tempSum == 0:
                ii, tempD, tempCx = self.RunLengthCoding(plane[row:row+4, col])
                if tempD.__len__() == 1:
                    D[pointer] = tempD
```

```
                CX[pointer] = tempCx
                pointer = pointer + 1
            else:
                D[pointer],D[pointer + 1], D[pointer+2] = tempD[0], tempD[1], tempD[2]
                CX[pointer],CX[pointer + 1], CX[pointer+2] = tempCx[0], tempCx[1], tempCx[2]
                pointer = pointer + 3
                # sign coding
                row = i*4 + ii - 1
                signComp, tempCx = self.SignCoding(S1extend[row:row+3,col:col+3], signs[row][col])
                D[pointer] = signComp
                CX[pointer] = tempCx
                pointer = pointer + 1
                S1[row][col] = 1
                S1extend = np.pad(S1, ((1,1), (1,1)), 'constant')
        while ii < 4:
            row = i*4 + ii
            ii = ii + 1
            if S1[row][col] != 0 or S3[row][col] != 0:
                continue
            tempCx = self.ZeroCoding(S1extend[row:row+3,col:col+3], bandMark)
            D[pointer] = plane[row][col]
            CX[pointer] = tempCx
            pointer = pointer + 1
            if plane[row][col] == 1:  # signcoding
                signComp, tempCx = self.SignCoding(S1extend[row:row+3,col:col+3], signs[row][col])
                D[pointer][0] = signComp
                CX[pointer][0] = tempCx
                pointer = pointer + 1
                S1[row][col] = 1 # mark as significant
                S1extend = np.pad(S1, ((1,1), (1,1)), 'constant')
    return D, CX, pointer, S1


class ValidationError(Exception):
    pass


def mq_table():
    CX_Table = [[4,0],[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], [0,0], [3,0],[46,0]]
    np.save(r"CX_Table", CX_Table)
    QeHex = ['5601','3401','1801','0AC1','0521','0221','5601','5401','4801','3801','3001','2401','1C01','1601',
        '5601','5401','5101','4801','3801','3401','3001','2801','2401','2201','1C01','1801','1601','1401',
        '1201','1101','0AC1','09C1','08A1','0521','0441','02A1','0221','0141','0111','0085','0049','0025',
        '0015','0009','0005','0001','5601']
    Qe = [int(x,16) for x in QeHex]
    NMPS = [1, 2, 3, 4, 5,38, 7, 8, 9,10,11,12,13,29,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,
         32,33,34,35,36,37,38,39,40,41,42,43,44,45,45,46]
    NLPS = [1, 6, 9,12,29,33, 6,14,14,14,17,18,20,21,14,14,15,16,17,18,19,19,20,21,22,23,24,25,26,27,28,29,30,31,
         32,33,34,35,36,37,38,39,40,41,42,43,46]
    swit = [0]*47
    swit[0] = 1
    swit[6] = 1
    swit[14] = 1
    PETTable = np.vstack((NMPS, NLPS, swit, Qe))
    PETTable = np.transpose(PETTable)
    np.save(r"PETTable", PETTable)


jpeg = JPEG2000(file_path='test.bmp', lossy=False, debug=False)
jpeg.run()
```