Федеральное государственное бюджетное образовательное учреждение высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»
(СибГУТИ)

Институт информатики и вычислительной техники

Кафедра прикладной математики и кибернетики

Лабораторная работа №5
по дисциплине
Прикладная стеганография

Выполнил:

студент гр.МГ-411 Шевельков П.С.
ФИО студента

«15» мая 2025 г.

Новосибирск 2025 г.

Задание на лабораторную работу:

1) Составить обзор статистических методов стегоанализа изображений: анализ статистики Хи-квадрат, RS-анализ, метод AUMP.

2) Реализовать программное средство для стегоанализа изображений, включающее в себя:

1. Визуальную атаку на стегоконтейнер, взятую из задания №1;

2. Анализ статистики Хи-квадрат по частям изображения;

3. RS-анализ, взятый из источника: https://github.com/b3dk7/StegExpose/blob/master/RSAnalysis.java

Программа выдает число - предполагаемый процент пикселей, содержащих скрытую информацию:

- *0%: встраивание не обнаружено*
- *1-5%: вероятно, нет скрытого сообщения*
- *5%+: вероятно, есть скрытая информация. Чем больше %, тем больше обнаружено.*

4. Метод AUMP, взятый из источника: http://dde.binghamton.edu/download/structural_lsb_detectors/

*Можно использовать следующие значения**:* **beta = aump(X, 16, 2);**
***m = 16**: Размер блока.* Чем больше **m**, тем более глобальными становятся предсказания значений пикселей, так как модель учитывает больше данных для построения аппроксимации. Маленькие значения **m** (например, 4 или 8) обеспечивают более локальную аппроксимацию, что может быть полезно для изображений с высокой текстурностью или резкими изменениями яркости.

***d = 2**: Степень полинома для учета небольших градиентов.* Чем выше значение **d**, тем более сложная модель используется для аппроксимации значений пикселей. Для маленьких блоков (`m ≤ 8`) рекомендуется использовать низкую степень (`d = 1`), чтобы избежать переобучения.

***sig_th = 1**: Порог дисперсии для обеспечения числовой стабильности (установлен по умолчанию)*

*Перед использованием в функции **aump**, изображение должно быть преобразовано в числовой формат в виде матрицы X (изображение в градациях серого).*

Загрузка изображения на матлаб:

X = imread('image.png'); % Загружаем изображение

Изображение должно быть представлено в градациях серого (grayscale):
X = rgb2gray(X); % Преобразуем в градации серого, если изображение цветное

Рекомендуемое **пороговое значение** для beta: 0.01

Пороговое значение можно подбирать самостоятельно, регулируя ошибки 1 и 2 рода для конкретного способа встраивания.

Значение β **не должно быть отрицательным** , если алгоритм реализован корректно. Если вы получаете отрицательное значение β, это указывает на возможную ошибку в реализации, некорректные входные данные или влияние шума/артефактов/сжатия в изображении.

5. Дополнительно можно реализовать стегоанализ на основе сжатия.

Необходимо, чтобы программа позволяла загружать как отдельное изображение, так и сразу несколько изображений, предоставив пользователю возможность выбрать расположение файлов.

Результаты стегоанализа должны отображаться в интерфейсе программного средства в понятном для пользователя виде, предполагая работу стороннего стегоаналитика. При анализе нескольких файлов сразу, результаты должны записываться в текстовый файл по выбранному пути сохранения.

Отчет по работе должен содержать результаты всех пунктов задания, включая описание кода программы и ссылку на него.

Результаты работы программы:

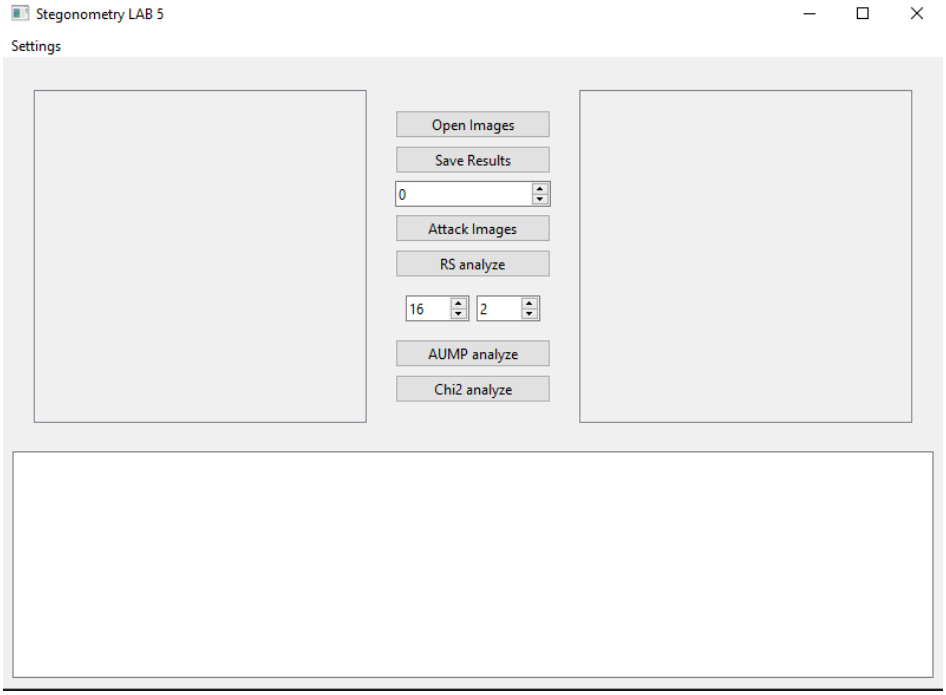

рисунок 1. Интерфейс программы.

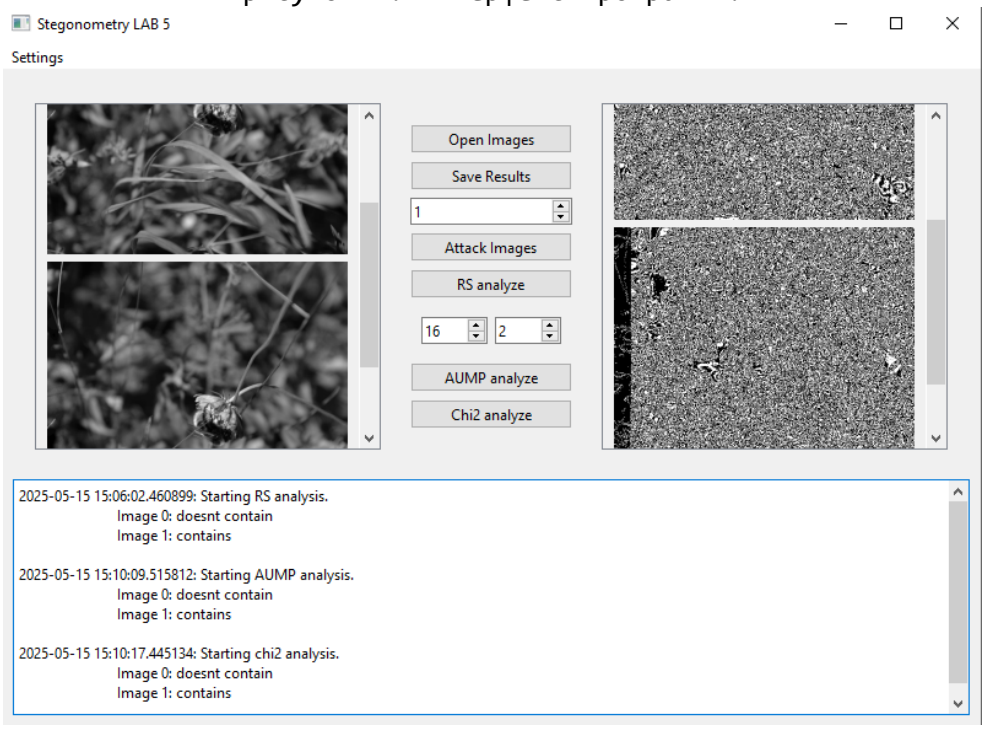

рисунок 2. Результат методов.

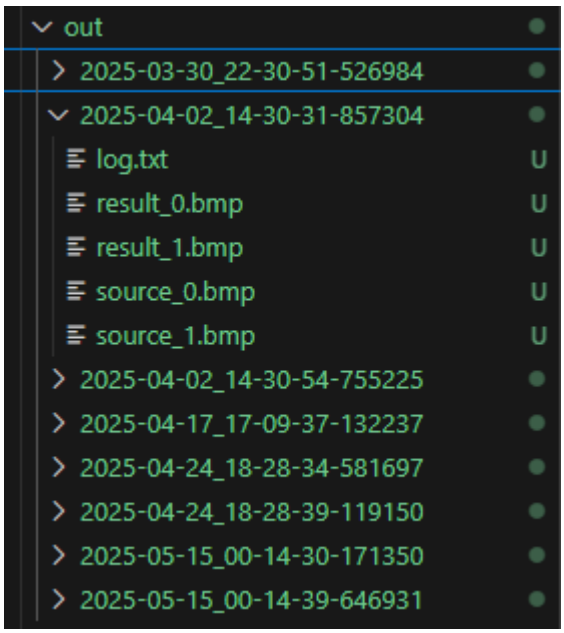

рисунок 3. Выходные файлы.

Описание методов анализа изображений для выявления скрытых данных

Методы анализа изображений, представленные в данной реализации, направлены на выявление наличия скрытых данных в изображениях, используя статистические и визуальные подходы. Эти методы входят в арсенал инструментов стегоанализа — дисциплины, исследующей возможность обнаружения скрытой информации, встроенной в мультимедийные файлы (например, изображения), с целью вскрытия или подтверждения факта стеганографического вмешательства.

Описание работы приложения:

Приложение "Стегоанализ изображений" предназначено для проведения анализа изображений с целью выявления следов скрытия информации. Пользователь может загрузить изображение, после чего применяются различные методы анализа, такие как:

- Визуализация битовых плоскостей (Bit Plane Slicing),
- Хи-квадрат анализ (Chi-square attack),
- RS-анализ (Regular-Singular Analysis).

Эти методы позволяют оценить вероятность наличия скрытых данных, в частности, встроенных с использованием методов LSB-замены или других стеганографических подходов.

Основные компоненты кода:

Класс Attacker:

Метод attack_image(img_in, bit_index): реализует визуальный метод анализа, отображая заданную битовую плоскость изображения. Например, если bit_index = 0, то отображается наименее значимый бит каждого пикселя. Это позволяет визуально выявить аномалии в распределении бит, характерные для стеганографического встраивания.

Класс Chi2:

Метод analyze(image, block_size): реализует хи-квадрат анализ, оценивая статистическое отклонение реального распределения значений пикселей от ожидаемого. Изображение разбивается на блоки, и в каждом блоке рассчитывается хи-квадрат статистика. Высокое значение статистики может указывать на наличие скрытой информации.

Метод возвращает логическое значение: True, если вероятность наличия скрытых данных значительна, и False в противном случае.

Класс RSAnalysis:

Метод analyze(image, color, overlap): реализует RS-анализ, основанный на анализе изменений вариаций блоков пикселей при применении масок. Метод оценивает количество регулярных и сингулярных блоков, а также рассчитывает значения, характеризующие скрытность данных.

Используется инвертирование и флип масок для оценки устойчивости изображения к модификациям, что позволяет выявить структуры, типичные

для стеганографического вмешательства.
Анализ проводится только для изображений в градациях серого.

Описание основных алгоритмов:

1. Bit Plane Slicing:
Позволяет визуализировать конкретный бит каждого пикселя изображения. Изменения в наименее значащих битах часто используются для сокрытия информации. Отображение этих битов помогает выявить подозрительные паттерны.

2. Хи-квадрат анализ:
Оценивает соответствие распределения значений пикселей ожидаемому распределению. При наличии скрытых данных, особенно при использовании LSB-замены, вероятностное распределение изменяется, что фиксируется хи-квадрат критерием.

3. RS-анализ:
Блоки изображения делятся на регулярные и сингулярные. Изменение количества таких блоков при применении флип-масок позволяет оценить вероятность наличия скрытых данных. Используется вычисление вариаций и сравнение изменений до и после модификации блоков.

Ссылка на программу:

https://github.com/bothyD/steganograf

Листинг:

```
import sys
from PIL import Image
import numpy as np
from scipy import stats as scipy_stats
import scipy.io


class Attacker:
    @staticmethod
    def attack_image(img_in: Image.Image, bit_index: int) -> Image.Image:
        img = Image.new("L", img_in.size)
        pixels_in = img_in.load()
        pixels = img.load()

        if pixels_in is None or pixels is None:
            raise BaseException("pixel arrays are none")

        for x in range(img.size[0]):
            for y in range(img.size[1]):
                pixels[x, y] = 255 * (pixels_in[x, y] & (1 << bit_index))
        return img


class Chi2:
    __MIN_BIN = 5

    @staticmethod
    def analyze(image: Image.Image, block_size: int = 64):
        pixels = image.load()
        if pixels is None:
            raise BaseException
        outs = []

        for x_max in range(0, image.width, block_size):
            for y_max in range(0, image.height, block_size):
                x_min = min(0, x_max - block_size)
                y_min = min(0, y_max - block_size)

                distribution_actual: list[int] = [0] * 8
                for x in range(x_min, x_max):
                    for y in range(y_min, y_max):
                        distribution_actual[pixels[x, y] & 0b111] += 1

                distribution_mean: list[int] = [0] * len(distribution_actual)
                for index in range(0, len(distribution_actual) - 1, 2):
                    mean = (
                        distribution_actual[index] + distribution_actual[index +
1]
                    ) / 2
                    if mean.is_integer():
```

```
                        distribution_mean[index] = distribution_mean[index + 1] =
int(
                            mean
                        )
                    elif distribution_actual[index] < distribution_actual[index +
1]:
                        distribution_mean[index] = int(mean)
                        distribution_mean[index + 1] = int(mean) + 1
                    else:
                        distribution_mean[index] = int(mean) + 1
                        distribution_mean[index + 1] = int(mean)

                # combine bins
                index = 0
                Chi2.__MIN_BIN = np.average(distribution_actual)
                while index < len(distribution_actual):
                    if distribution_actual[index] < Chi2.__MIN_BIN:
                        val_sum = distribution_actual[index]
                        stop = index
                        for jindex in range(index + 1, len(distribution_actual)):
                            val_sum += distribution_actual[jindex]
                            if val_sum >= Chi2.__MIN_BIN:
                                stop = jindex
                                break
                        else:
                            stop = len(distribution_actual) - 1

                        distribution_actual[index : stop + 1] = [
                            sum(distribution_actual[index : stop + 1])
                        ]
                        distribution_mean[index : stop + 1] = [
                            sum(distribution_mean[index : stop + 1])
                        ]

                    index += 1
                if distribution_actual[-1] < Chi2.__MIN_BIN:
                    distribution_actual[-2:] = [sum(distribution_actual[-2:])]
                    distribution_mean[-2:] = [sum(distribution_mean[-2:])]
                del index
                # print(distribution_actual)
                # print(distribution_mean)
                for x in distribution_actual:
                    if x == 0:
                        break
                else:
                    if len(distribution_actual) == 2:
                        if distribution_actual[0] == 0 or distribution_actual[1]
== 0:
                            outs.append(1)
                            continue
```

```
                    if len(distribution_actual) == 1:
                        outs.append(1)
                        continue

                    outs.append(
                        scipy_stats.chisquare(
                            f_obs=distribution_actual, f_exp=distribution_mean,
ddof=1
                        )[1]
                    )
        # outs = [x for x in outs if x > 0.000001]
        # print(outs)
        print(np.average(outs))
        return np.average(outs) < 0.5
        # return len(outs) > 2 * len([x for x in outs if x > 0.05])


class RSAnalysis:
    ANALYSIS_COLOR_GRAYSCALE = -1
    ANALYSIS_COLOR_RED = 0
    ANALYSIS_COLOR_GREEN = 1
    ANALYSIS_COLOR_BLUE = 2

    def __init__(self, m: int, n: int):
        self.__mMask = [[0] * (m * n), [0] * (m * n)]

        k: int = 0
        for i in range(n):
            for j in range(m):
                if ((j % 2) == 0 and (i % 2) == 0) or ((j % 2) == 1 and (i % 2)
== 1):
                    self.__mMask[0][k] = 1
                    self.__mMask[1][k] = 0
                else:
                    self.__mMask[0][k] = 0
                    self.__mMask[1][k] = 1
                k += 1

        self.__mM = m
        self.__mN = n

    # colorfull images are not supported currently
    # def analyze(self, image: Image.Image, color: int, overlap: bool) ->
list[float]:
    def analyze(
        self,
        image: Image.Image,
        color: int = ANALYSIS_COLOR_GRAYSCALE,
        overlap: bool = True,
    ) -> bool:
        imgx: int = image.width
```

```
        imgy: int = image.height

        startx: int = 0
        starty: int = 0
        block: list[int] = [0] * (self.__mM * self.__mN)

        numregular: float = 0
        numsingular: float = 0
        numnegreg: float = 0
        numnegsing: float = 0
        numunusable: float = 0
        numnegunusable: float = 0
        variationB: float
        variationP: float
        variationN: float

        pixels = image.load()
        if pixels is None:
            raise BaseException("pixels is none")

        while startx < imgx and starty < imgy:
            for m in range(2):
                k: int = 0
                for i in range(self.__mN):
                    for j in range(self.__mM):
                        block[k] = pixels[startx + j, starty + i]
                        k += 1

                variationB = self.__getVariation(block, color)

                block = self.__flipBlock(block, self.__mMask[m])
                variationP = self.__getVariation(block, color)
                block = self.__flipBlock(block, self.__mMask[m])

                self.__mMask[m] = self.__invertMask(self.__mMask[m])
                variationN = self.__getNegativeVariation(block, color,
self.__mMask[m])
                self.__mMask[m] = self.__invertMask(self.__mMask[m])

                if variationP > variationB:
                    numregular += 1
                if variationP < variationB:
                    numsingular += 1
                if variationP == variationB:
                    numunusable += 1

                if variationN > variationB:
                    numnegreg += 1
                if variationN < variationB:
                    numnegsing += 1
                if variationN == variationB:
```

```
                numnegunusable += 1

            if overlap:
                startx += 1
            else:
                startx += self.__mM

            if startx >= (imgx - 1):
                startx = 0
                if overlap:
                    starty += 1
                else:
                    starty += self.__mN
            if starty >= (imgy - 1):
                break
        totalgroups: float = numregular + numsingular + numunusable
        allpixels: list[float] = self.__getAllPixelFlips(image, color, overlap)
        x: float = self.__getX(
            numregular,
            numnegreg,
            allpixels[0],
            allpixels[2],
            numsingular,
            numnegsing,
            allpixels[1],
            allpixels[3],
        )

        epf: float
        ml: float
        if 2 * (x - 1) == 0:
            epf = 0
        else:
            epf = abs(x / (2 * (x - 1)))

        if x - 0.5 == 0:
            ml = 0
        else:
            ml = abs(x / (x - 0.5))

        results: list[float] = [0] * 28

        results[0] = numregular
        results[1] = numsingular
        results[2] = numnegreg
        results[3] = numnegsing
        results[4] = abs(numregular - numnegreg)
        results[5] = abs(numsingular - numnegsing)
        results[6] = (numregular / totalgroups) * 100
        results[7] = (numsingular / totalgroups) * 100
        results[8] = (numnegreg / totalgroups) * 100
```

```
        results[9] = (numnegsing / totalgroups) * 100
        results[10] = (results[4] / totalgroups) * 100
        results[11] = (results[5] / totalgroups) * 100

        results[12] = allpixels[0]
        results[13] = allpixels[1]
        results[14] = allpixels[2]
        results[15] = allpixels[3]
        results[16] = abs(allpixels[0] - allpixels[1])
        results[17] = abs(allpixels[2] - allpixels[3])
        results[18] = (allpixels[0] / totalgroups) * 100
        results[19] = (allpixels[1] / totalgroups) * 100
        results[20] = (allpixels[2] / totalgroups) * 100
        results[21] = (allpixels[3] / totalgroups) * 100
        results[22] = (results[16] / totalgroups) * 100
        results[23] = (results[17] / totalgroups) * 100

        results[24] = totalgroups
        results[25] = epf
        results[26] = ml
        results[27] = ((imgx * imgy * 3) * ml) / 8
        print(ml)

        return ml > 0.001# процент определения сообщения

    def __getX(
        self,
        r: float,
        rm: float,
        r1: float,
        rm1: float,
        s: float,
        sm: float,
        s1: float,
        sm1: float,
    ) -> float:
        x: float = 0

        dzero: float = r - s
        dminuszero: float = rm - sm
        done: float = r1 - s1
        dminusone: float = rm1 - sm1

        a: float = 2 * (done + dzero)
        b: float = dminuszero - dminusone - done - (3 * dzero)
        c: float = dzero - dminuszero

        if a == 0:
            x = c / b

        discriminant: float = b * b - (4 * a * c)
```

```
        if discriminant >= 0:
            rootpos: float = ((-1 * b) + np.sqrt(discriminant)) / (2 * a)
            rootneg: float = ((-1 * b) - np.sqrt(discriminant)) / (2 * a)

            if np.abs(rootpos) <= np.abs(rootneg):
                x = rootpos
            else:
                x = rootneg

        else:
            cr = (rm - r) / (r1 - r + rm - rm1)
            cs = (sm - s) / (s1 - s + sm - sm1)
            x = (cr + cs) / 2

        if x == 0:
            ar = ((rm1 - r1 + r - rm) + (rm - r) / x) / (x - 1)
            as_ = ((sm1 - s1 + s - sm) + (sm - s) / x) / (x - 1)
            if as_ > 0 or ar < 0:
                cr = (rm - r) / (r1 - r + rm - rm1)
                cs = (sm - s) / (s1 - s + sm - sm1)
                x = (cr + cs) / 2

        return x

    def __getAllPixelFlips(
        self, image: Image.Image, color: int, overlap: bool
    ) -> list[float]:
        allmask: list[int] = [1] * (self.__mM * self.__mN)

        imgx: int = image.width
        imgy: int = image.height

        startx: int = 0
        starty: int = 0
        block: list[int] = [0] * (self.__mM * self.__mN)

        numregular: float = 0
        numsingular: float = 0
        numnegreg: float = 0
        numnegsing: float = 0
        numunusable: float = 0
        numnegunusable: float = 0
        variationB: float
        variationP: float
        variationN: float

        pixels = image.load()
        if pixels is None:
            raise BaseException("pixels is none")
```

```
        while startx < imgx and starty < imgy:
            for m in range(2):
                k: int = 0
                for i in range(self.__mN):
                    for j in range(self.__mM):
                        block[k] = pixels[startx + j, starty + i]
                        k += 1

                block = self.__flipBlock(block, allmask)

                variationB = self.__getVariation(block, color)

                block = self.__flipBlock(block, self.__mMask[m])
                variationP = self.__getVariation(block, color)
                block = self.__flipBlock(block, self.__mMask[m])

                self.__mMask[m] = self.__invertMask(self.__mMask[m])
                variationN = self.__getNegativeVariation(block, color,
self.__mMask[m])
                self.__mMask[m] = self.__invertMask(self.__mMask[m])

                if variationP > variationB:
                    numregular += 1
                if variationP < variationB:
                    numsingular += 1
                if variationP == variationB:
                    numunusable += 1

                if variationN > variationB:
                    numnegreg += 1
                if variationN < variationB:
                    numnegsing += 1
                if variationN == variationB:
                    numnegunusable += 1

            if overlap:
                startx += 1
            else:
                startx += self.__mM

            if startx >= (imgx - 1):
                startx = 0
                if overlap:
                    starty += 1
                else:
                    starty += self.__mN
            if starty >= (imgy - 1):
                break

        results: list[float] = [0] * 4
```

```
        results[0] = numregular
        results[1] = numsingular
        results[2] = numnegreg
        results[3] = numnegsing

        return results

    @staticmethod
    def getResultNames() -> tuple[str, ...]:
        return (
            "Number of regular groups (positive)",
            "Number of singular groups (positive)",
            "Number of regular groups (negative)",
            "Number of singular groups (negative)",
            "Difference for regular groups",
            "Difference for singular groups",
            "Percentage of regular groups (positive)",
            "Percentage of singular groups (positive)",
            "Percentage of regular groups (negative)",
            "Percentage of singular groups (negative)",
            "Difference for regular groups %",
            "Difference for singular groups %",
            "Number of regular groups (positive for all flipped)",
            "Number of singular groups (positive for all flipped)",
            "Number of regular groups (negative for all flipped)",
            "Number of singular groups (negative for all flipped)",
            "Difference for regular groups (all flipped)",
            "Difference for singular groups (all flipped)",
            "Percentage of regular groups (positive for all flipped)",
            "Percentage of singular groups (positive for all flipped)",
            "Percentage of regular groups (negative for all flipped)",
            "Percentage of singular groups (negative for all flipped)",
            "Difference for regular groups (all flipped) %",
            "Difference for singular groups (all flipped) %",
            "Total number of groups",
            "Estimated percent of flipped pixels",
            "Estimated message length (in percent of pixels)(p)",
            "Estimated message length (in bytes)",
        )

    def __getVariation(self, block: list[int], color: int) -> float:
        var: float = 0
        color1: int
        color2: int
        for i in range(0, len(block), 4):
            color1 = self.__getPixelColor(block[0 + i], color)
            color2 = self.__getPixelColor(block[1 + i], color)
            var += np.abs(color1 - color2)
            color1 = self.__getPixelColor(block[3 + i], color)
            color2 = self.__getPixelColor(block[2 + i], color)
            var += np.abs(color1 - color2)
```

```
            color1 = self.__getPixelColor(block[1 + i], color)
            color2 = self.__getPixelColor(block[3 + i], color)
            var += np.abs(color1 - color2)
            color1 = self.__getPixelColor(block[2 + i], color)
            color2 = self.__getPixelColor(block[0 + i], color)
            var += np.abs(color1 - color2)
        return var

    def __getNegativeVariation(
        self, block: list[int], color: int, mask: list[int]
    ) -> float:
        var: float = 0
        color1: int
        color2: int
        for i in range(0, len(block), 4):
            color1 = self.__getPixelColor(block[0 + i], color)
            color2 = self.__getPixelColor(block[1 + i], color)
            if mask[0 + i] == -1:
                color1 = self.__invertLSB(color1)
            if mask[1 + i] == -1:
                color2 = self.__invertLSB(color2)
            var += np.abs(color1 - color2)

            color1 = self.__getPixelColor(block[1 + i], color)
            color2 = self.__getPixelColor(block[3 + i], color)
            if mask[1 + i] == -1:
                color1 = self.__invertLSB(color1)
            if mask[3 + i] == -1:
                color2 = self.__invertLSB(color2)
            var += np.abs(color1 - color2)

            color1 = self.__getPixelColor(block[3 + i], color)
            color2 = self.__getPixelColor(block[2 + i], color)
            if mask[3 + i] == -1:
                color1 = self.__invertLSB(color1)
            if mask[2 + i] == -1:
                color2 = self.__invertLSB(color2)
            var += np.abs(color1 - color2)

            color1 = self.__getPixelColor(block[2 + i], color)
            color2 = self.__getPixelColor(block[0 + i], color)
            if mask[2 + i] == -1:
                color1 = self.__invertLSB(color1)
            if mask[0 + i] == -1:
                color2 = self.__invertLSB(color2)
            var += np.abs(color1 - color2)
        return var

    def __getPixelColor(self, pixel: int, color: int) -> int:
        return pixel
```

```
    def __flipBlock(self, block: list[int], mask: list[int]) -> list[int]:
        for i in range(len(block)):
            if mask[i] == 1:
                block[i] = self.__negateLSB(block[i])
            elif mask[i] == -1:
                block[i] = self.__invertLSB(block[i])
        return block

    def __negateLSB(self, abyte: int) -> int:
        temp = abyte & 0xFE
        if temp == abyte:
            return abyte | 0x1
        else:
            return temp

    def __invertLSB(self, abyte: int) -> int:
        if abyte == 255:
            return 256
        if abyte == 256:
            return 255
        return self.__negateLSB(abyte + 1) - 1

    def __invertMask(self, mask: list[int]) -> list[int]:
        return [x * -1 for x in mask]


class AUMP:
    @staticmethod
    def analyze(image: Image.Image, block_size: int, parameters: int) -> bool:
        pixels = np.array(image, dtype=np.floating)
        scipy.io.savemat("array.mat", {"X": pixels})
        try:
            a = AUMP.__aump(pixels, block_size, parameters)
            print(a)
            return a > 0.01
        except BaseException:
            return True

    @staticmethod
    def __aump(X, m, d):
        Xpred, _, w = AUMP.__pred_aump(X, m, d)
        r = X - Xpred
        Xbar = X + 1 - 2 * (X % 2)
        beta = np.sum(w * (X - Xbar) * r)
        return beta

    @staticmethod
    def __pred_aump(X, m, d):
        sig_th = 1
        q = d + 1
        Kn = X.size // m
```

```
        Y = np.zeros((m, Kn))
        S = np.zeros_like(X)
        Xpred = np.zeros_like(X)

        x1 = np.linspace(1, m, m) / m
        H = np.vander(x1, q, increasing=True)

        for i in range(m):
            aux = X[:, i::m]
            Y[i, :] = aux.flatten()

        p = np.linalg.lstsq(H, Y, rcond=None)[0]
        Ypred = H @ p

        for i in range(m):
            Xpred[:, i::m] = Ypred[i, :].reshape(X[:, i::m].shape)

        sig2 = np.sum((Y - Ypred) ** 2, axis=0) / (m - q)
        sig2 = np.maximum(sig_th**2, sig2)

        Sy = np.ones((m, 1)) * sig2

        for i in range(m):
            S[:, i::m] = Sy[i, :].reshape(X[:, i::m].shape)

        s_n2 = Kn / np.sum(1.0 / sig2)
        w = np.sqrt(s_n2 / (Kn * (m - q))) / S

        return Xpred, S, w
```