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 №6
по дисциплине
Прикладная стеганография

Выполнил:

студент гр.МГ-411 Шевельков П.С.

ФИО студента

«15» мая 2025 г.

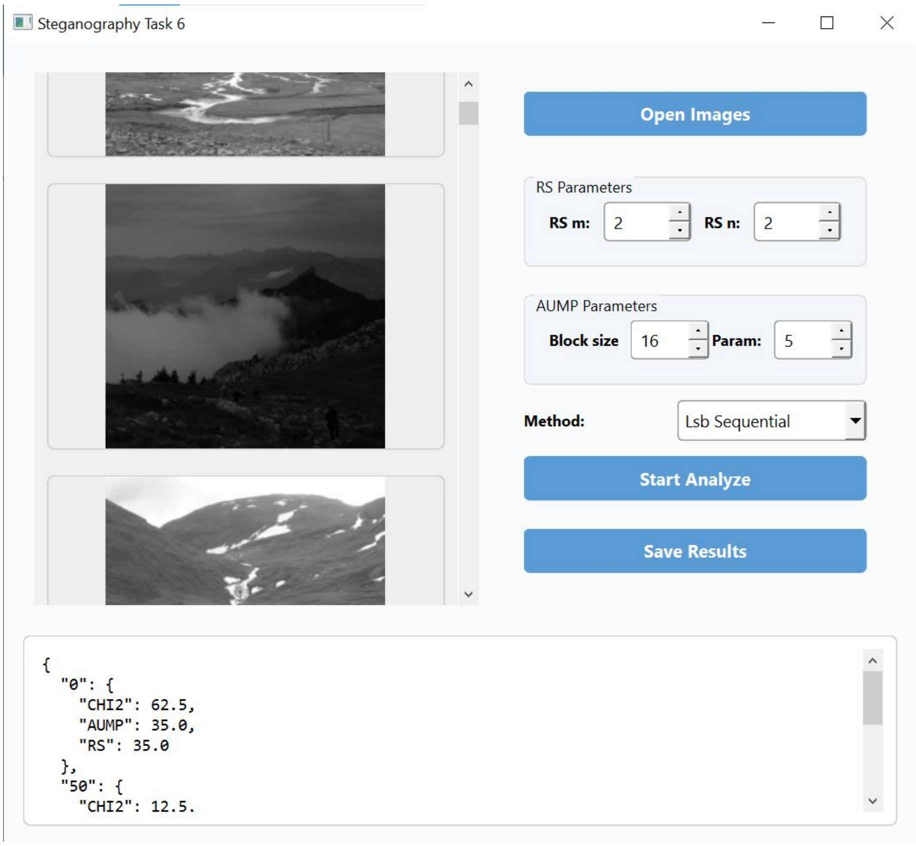
Новосибирск 2025 г.

Задание на лабораторную работу:

Написать программу, которая использует выходные текстовые файлы из программного средства стегоанализа, реализованного в задании №5 для подсчета ошибки 1 и 2 рода. Заполненные контейнеры взять из результатов работ №3-4.

Отчет по работе должен содержать описание формата вывода данных стегоанализа, подсчет ошибки 1 и 2 рода и таблицу сравнения методов стегоанализа. В таблице привести результаты стегоанализа при разном заполнении контейнеров, указав максимально возможную фактическую ёмкость контейнера и % заполнения стегоконтейнера. Например, 50% заполненный стегоконтейнер при последовательном заполнении; при рассеянном заполнении; также для 100%. Привести ссылку на исходники.

Результаты работы программы:



RS Parameters

RS m: 2 RS n: 2

AUMP Parameters

Block size 16 Param: 5

Method: Lsb Sequential

```
{
  "0": {
    "CHI2": 62.5,
    "AUMP": 35.0,
    "RS": 35.0
  },
  "50": {
    "CHI2": 12.5.
```

рисунок 1. Интерфейс программы.

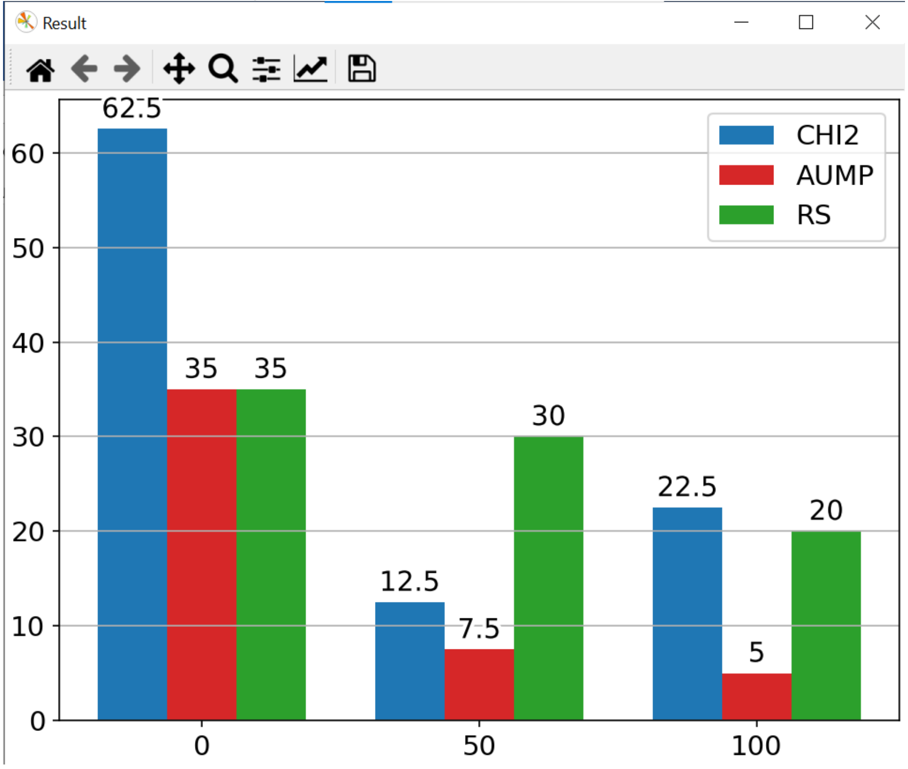


рисунок 2. Результат работы для выборки из 40 контейнеров.

Описание программы по анализу изображений с внедрёнными сообщениями

1. Загрузка набора изображений (в формате BMP).
2. Внедрение случайного битового сообщения в изображения с помощью одного из трёх методов LSB.
3. Проведение анализа изображений с различной степенью внедрения (0%, 50%, 100%).
4. Расчёт статистических метрик: Chi², AUMP, RS.
5. Отображение результатов в виде текстового отчёта и графика.
6. Возможность сохранить результаты в файл.

Используемые методы внедрения (LSB):

В проекте реализованы и сравниваются следующие методы внедрения сообщений:

1. LSB Pattern Matching (LSBSeq)

Последовательное внедрение битов в младшие биты пикселей.
Использует 3 младших бита на пиксель.
Простая и предсказуемая стратегия.

2. LSB Sequential (LSBMessage)

Вставка сообщений с учётом конкретной структуры изображения.
Предположительно, внедрение более устойчиво к обнаружению.

3. LSB Scaled

Масштабирует изображение перед внедрением.
Используется альтернативная стратегия внедрения и анализа.

Программа внедряет случайную битовую последовательность в каждое изображение в объёме:

- 0% (контрольная группа),
- 50%,
- 100% от максимально возможной вместимости.

Для каждой степени внедрения проводится анализ:

- Chi² — проверка распределения значений пикселей.
- AUMP — анализ шаблонов (параметризуемый).
- RS-анализ — метод, чувствительный к модификациям LSB.

Результаты нормализуются и отображаются в процентах.

Архитектура программы:

1. MainWindow (главное окно):

- Интерфейс на PyQt6.
- Содержит:

        - Виджет для отображения изображений.
        - Панель управления (выбор метода, параметры анализа).
        - Отображение результатов.
- Поддерживает асинхронную обработку изображений через поток AnalyseWorker.

2. **AnalyseWorker:**
   - Запускается в отдельном потоке.
   - Осуществляет вставку сообщений и анализ изображений параллельно (через multiprocessing.Pool).
   - Генерирует отчёт по каждому уровню внедрения (0%, 50%, 100%).

3. LSBSeq, LSBMessage, LSBScaledMessage:
   - Реализация различных LSB-методов внедрения.
   - Все используют библиотеку bitarray для работы с битовыми сообщениями.

4. control.*:
   - Модули анализа: ChiSquaredAnalysis, RSAnalysis, AUMPAnalysis.
   - Модуль PlotBuilder — построение графиков результатов анализа.

Выводы:

- Программа позволяет визуализировать и количественно оценить изменения в изображениях в результате внедрения скрытых сообщений.
- Используемые статистические методы позволяют определить степень внедрения, что может быть полезно для задач стеганализа.
- Сравнение методов показывает, какие из них более устойчивы к обнаружению.

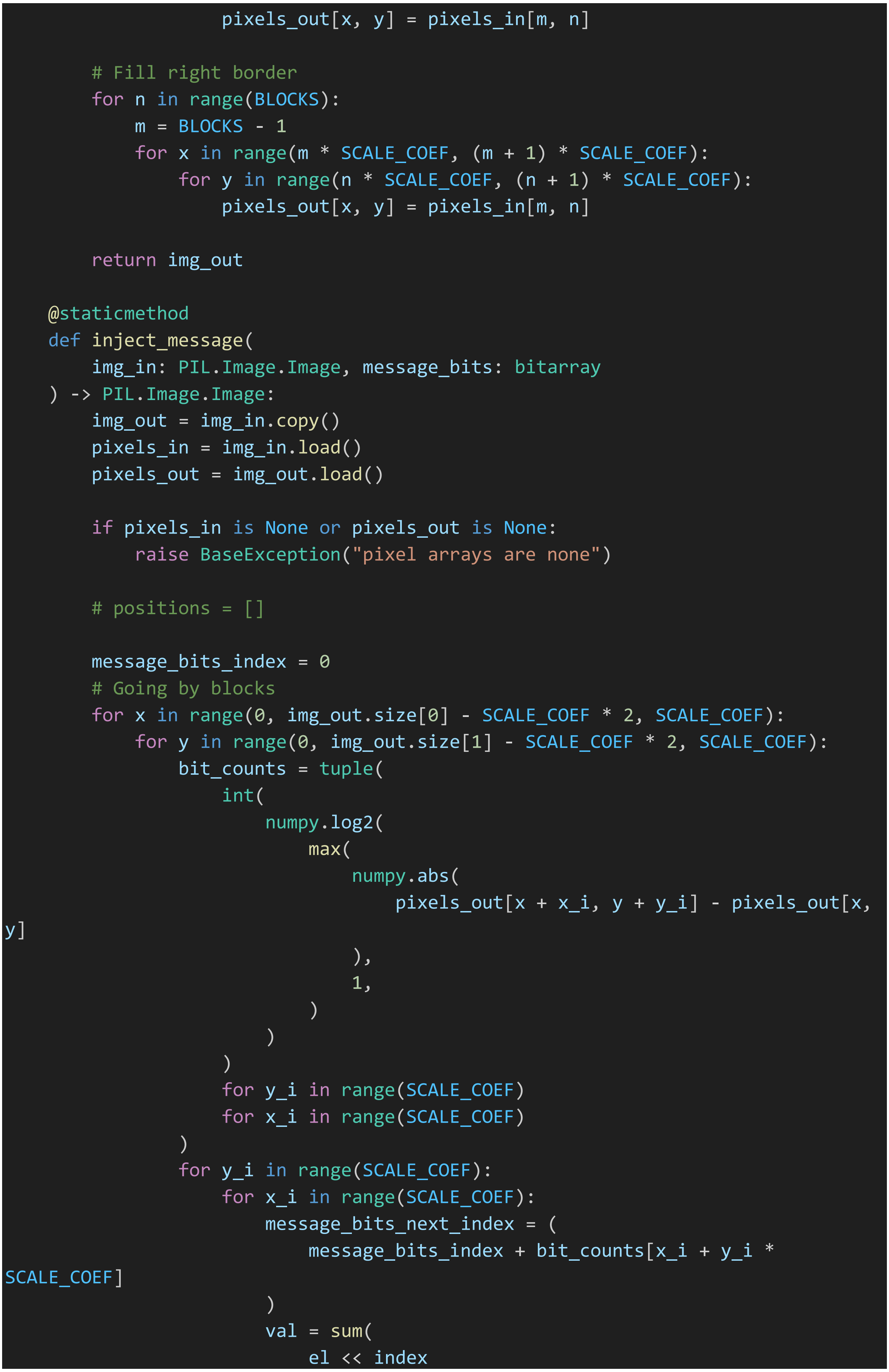
Ссылка на программу:

https://github.com/bothyD/steganograf

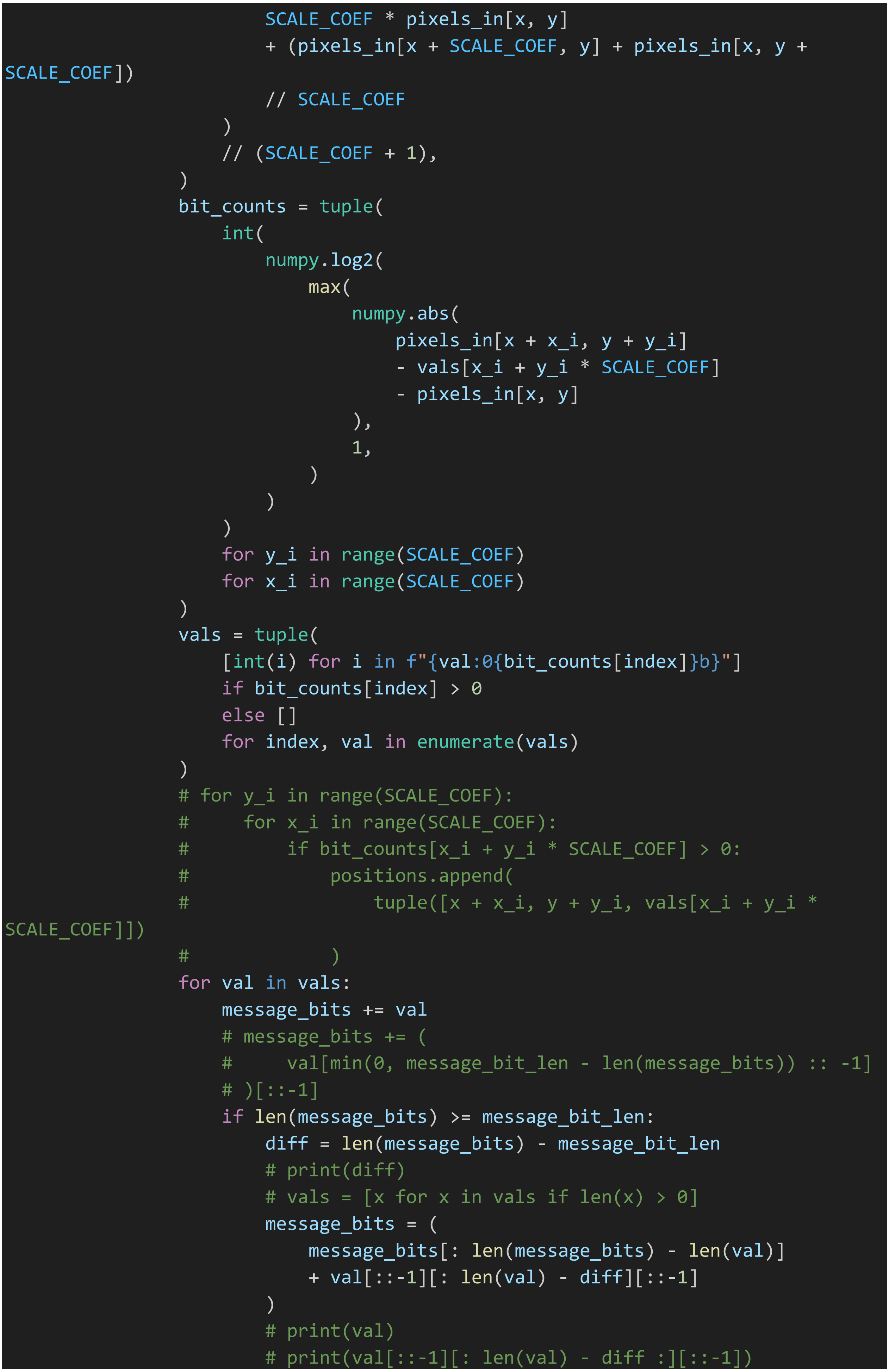
Листинг:

```
import PIL.Image
from bitarray import bitarray


class LSBMessage:
    @staticmethod
    def inject_message(
        img_in: PIL.Image.Image, message_bits: bitarray
    ) -> PIL.Image.Image:
        img = img_in.copy()
        pixels_in = img_in.load()
        pixels = img.load()
        if pixels_in is None or pixels is None:
            raise BaseException("pixels_in is None")

        msg_index = 0

        def index_in():
            return msg_index < len(message_bits) - 1

        def mi():
            try:
                return (message_bits[msg_index] << 1) | message_bits[msg_index +
1]
            except BaseException:
                return 0b100

        for x in range(img.size[0]):
            for y in range(img.size[1]):
                if not index_in():
                    break

                byte = pixels_in[x, y]
                clp = (byte & 0b11000000) >> 6
                cmp = (byte & 0b01100000) >> 5
                crp = (byte & 0b00110000) >> 4

                if mi() == clp:
                    byte |= 1 << 2
                    msg_index += 2
                else:
                    byte &= ~(1 << 2)
                if mi() == cmp:
                    byte |= 1 << 1
                    msg_index += 2
                else:
                    byte &= ~(1 << 1)
                if mi() == crp:
                    byte |= 1
                    msg_index += 2
                else:
                    byte &= ~(1)

                pixels[x, y] = byte
            else:
                continue
            break

        return img

    @staticmethod
    def extract_message(img_in: PIL.Image.Image, message_bit_len: int) ->
bitarray:
        pixels_in = img_in.load()
        msg_index = 0
        message_bits = []
        if pixels_in is None:
            raise BaseException("pixels_in is None")

        for x in range(img_in.size[0]):
            for y in range(img_in.size[1]):
                if msg_index >= message_bit_len:
                    break

                byte = pixels_in[x, y]

                # Извлекаем старшие биты (по 2 бита каждый)
                clp = (byte & 0b11000000) >> 6
                cmp = (byte & 0b01100000) >> 5
                crp = (byte & 0b00110000) >> 4

                # Извлекаем флаги (младшие 3 бита)
                clp_flag = (byte >> 2) & 1
                cmp_flag = (byte >> 1) & 1
                crp_flag = byte & 1

                # Если флаг установлен — значит соответствующий старший блок был
частью сообщения
                if clp_flag and msg_index + 2 <= message_bit_len:
                    message_bits.extend([(clp >> 1) & 1, clp & 1])
                    msg_index += 2
                if cmp_flag and msg_index + 2 <= message_bit_len:
                    message_bits.extend([(cmp >> 1) & 1, cmp & 1])
                    msg_index += 2
                if crp_flag and msg_index + 2 <= message_bit_len:
                    message_bits.extend([(crp >> 1) & 1, crp & 1])
                    msg_index += 2

                if msg_index >= message_bit_len:
                    break
            else:
                continue
            break

        return bitarray(message_bits)

    @staticmethod
    def get_max_capacity(img_in: PIL.Image.Image, message_bits: bitarray):
        img = img_in.copy()
        pixels_in = img_in.load()
        pixels = img.load()
        if pixels_in is None or pixels is None:
            raise BaseException("pixels_in is None")

        msg_index = 0

        def index_in():
            return msg_index < len(message_bits) - 1

        def mi():
            try:
                return (message_bits[msg_index] << 1) | message_bits[msg_index +
1]
            except BaseException:
                return 0b100

        for x in range(img.size[0]):
            for y in range(img.size[1]):
                if not index_in():
                    break

                byte = pixels_in[x, y]
                clp = (byte & 0b11000000) >> 6
                cmp = (byte & 0b01100000) >> 5
                crp = (byte & 0b00110000) >> 4

                if mi() == clp:
                    byte |= 1 << 2
                    msg_index += 2
                else:
                    byte &= ~(1 << 2)
                if mi() == cmp:
                    byte |= 1 << 1
                    msg_index += 2
                else:
                    byte &= ~(1 << 1)
                if mi() == crp:
                    byte |= 1
                    msg_index += 2
                else:
                    byte &= ~(1)
                pixels[x, y] = byte
            else:
                continue
            break

        return msg_index
```

```
import PIL.Image
from bitarray import bitarray
import numpy

SCALE_COEF = 2


class LSBScaledMessage:
    @staticmethod
    def scale_image(img_in: PIL.Image.Image) -> PIL.Image.Image:
        img_out = PIL.Image.new("L", (img_in.size[0] * 2, img_in.size[1] * 2))

        pixels_in = img_in.load()
        pixels_out = img_out.load()

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

        BLOCKS = img_in.size[0]

        for m in range(BLOCKS - 1):
            for n in range(BLOCKS - 1):
                x = m * SCALE_COEF
                y = n * SCALE_COEF
                pixels_out[x, y] = pixels_in[m, n]
                pixels_out[x + 1, y] = (
                    pixels_in[m, n] + pixels_in[m + 1, n]
                ) // SCALE_COEF
                pixels_out[x, y + 1] = (
                    pixels_in[m, n] + pixels_in[m, n + 1]
                ) // SCALE_COEF
                pixels_out[x + 1, y + 1] = (
                    SCALE_COEF * pixels_in[m, n]
                    + (pixels_in[m + 1, n] + pixels_in[m, n + 1]) // SCALE_COEF
                ) // (SCALE_COEF + 1)

        # Fill bottom border
        for m in range(BLOCKS):
            n = BLOCKS - 1
            for x in range(m * SCALE_COEF, (m + 1) * SCALE_COEF):
                for y in range(n * SCALE_COEF, (n + 1) * SCALE_COEF):
```



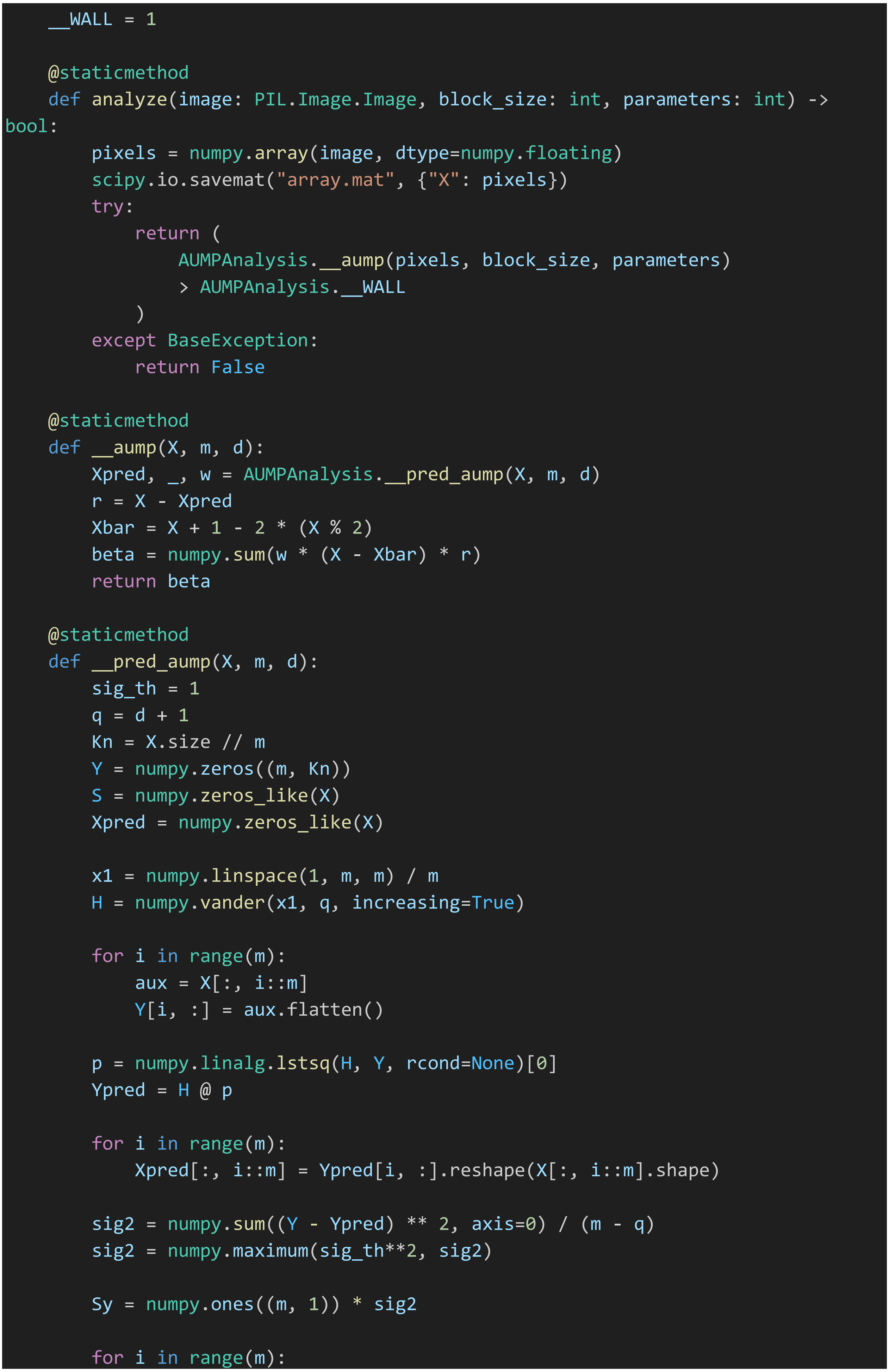
```
                    pixels_out[x, y] = pixels_in[m, n]

        # Fill right border
        for n in range(BLOCKS):
            m = BLOCKS - 1
            for x in range(m * SCALE_COEF, (m + 1) * SCALE_COEF):
                for y in range(n * SCALE_COEF, (n + 1) * SCALE_COEF):
                    pixels_out[x, y] = pixels_in[m, n]

        return img_out

    @staticmethod
    def inject_message(
        img_in: PIL.Image.Image, message_bits: bitarray
    ) -> PIL.Image.Image:
        img_out = img_in.copy()
        pixels_in = img_in.load()
        pixels_out = img_out.load()

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

        # positions = []

        message_bits_index = 0
        # Going by blocks
        for x in range(0, img_out.size[0] - SCALE_COEF * 2, SCALE_COEF):
            for y in range(0, img_out.size[1] - SCALE_COEF * 2, SCALE_COEF):
                bit_counts = tuple(
                    int(
                        numpy.log2(
                            max(
                                numpy.abs(
                                    pixels_out[x + x_i, y + y_i] - pixels_out[x,
y]
                                ),
                                1,
                            )
                        )
                    )
                    for y_i in range(SCALE_COEF)
                    for x_i in range(SCALE_COEF)
                )
                for y_i in range(SCALE_COEF):
                    for x_i in range(SCALE_COEF):
                        message_bits_next_index = (
                            message_bits_index + bit_counts[x_i + y_i *
SCALE_COEF]
                        )
                        val = sum(
                            el << index
                            for index, el in enumerate(
                                reversed(
                                    message_bits[
                                        message_bits_index:message_bits_next_inde
x
                                    ]
                                )
                            )
                        )
                        # if bit_counts[x_i + y_i * SCALE_COEF] > 0:
                        #     positions.append(tuple([x + x_i, y + y_i, val]))
                        pixels_out[x + x_i, y + y_i] += val

                        if message_bits_next_index >= len(message_bits):
                            break
                        message_bits_index = message_bits_next_index
                    # broke a leg, falling down the stairs
                    else:
                        continue
                    break
                else:
                    continue
                break
            else:
                continue
            break

        # print(positions)
        return img_out

    @staticmethod
    def extract_message(img_in: PIL.Image.Image, message_bit_len: int) ->
bitarray:
        pixels_in = img_in.load()
        if pixels_in is None:
            raise BaseException("pixels array is none")
        message_bits: list[int] = []
        # positions = []

        for x in range(0, img_in.size[0] - SCALE_COEF * 2, SCALE_COEF):
            for y in range(0, img_in.size[1] - SCALE_COEF * 2, SCALE_COEF):
                vals = (
                    0,
                    pixels_in[x + 1, y]
                    - (pixels_in[x, y] + pixels_in[x + SCALE_COEF, y]) //
SCALE_COEF,
                    pixels_in[x, y + 1]
                    - (pixels_in[x, y] + pixels_in[x, y + SCALE_COEF]) //
SCALE_COEF,
                    pixels_in[x + 1, y + 1]
                    - (
```



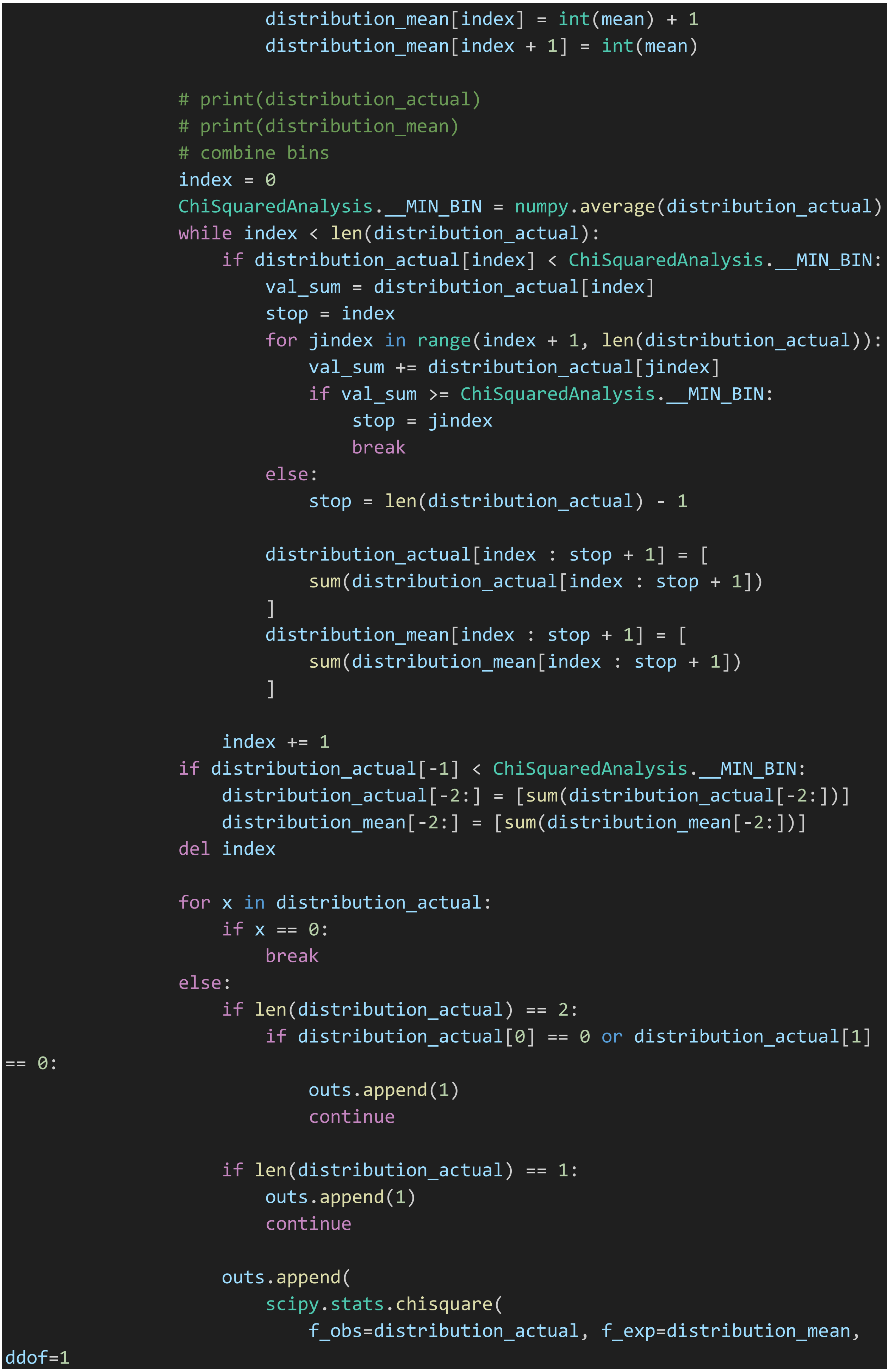
```
                        SCALE_COEF * pixels_in[x, y]
                        + (pixels_in[x + SCALE_COEF, y] + pixels_in[x, y +
SCALE_COEF])
                        // SCALE_COEF
                    )
                    // (SCALE_COEF + 1),
                )
                bit_counts = tuple(
                    int(
                        numpy.log2(
                            max(
                                numpy.abs(
                                    pixels_in[x + x_i, y + y_i]
                                    - vals[x_i + y_i * SCALE_COEF]
                                    - pixels_in[x, y]
                                ),
                                1,
                            )
                        )
                    )
                    for y_i in range(SCALE_COEF)
                    for x_i in range(SCALE_COEF)
                )
                vals = tuple(
                    [int(i) for i in f"{val:0{bit_counts[index]}b}"]
                    if bit_counts[index] > 0
                    else []
                    for index, val in enumerate(vals)
                )
                # for y_i in range(SCALE_COEF):
                #     for x_i in range(SCALE_COEF):
                #         if bit_counts[x_i + y_i * SCALE_COEF] > 0:
                #             positions.append(
                #                 tuple([x + x_i, y + y_i, vals[x_i + y_i *
SCALE_COEF]])
                #             )
                for val in vals:
                    message_bits += val
                    # message_bits += (
                    #     val[min(0, message_bit_len - len(message_bits)) :: -1]
                    # )[::-1]
                    if len(message_bits) >= message_bit_len:
                        diff = len(message_bits) - message_bit_len
                        # print(diff)
                        # vals = [x for x in vals if len(x) > 0]
                        message_bits = (
                            message_bits[: len(message_bits) - len(val)]
                            + val[::-1][: len(val) - diff][::-1]
                        )
                        # print(val)
                        # print(val[::-1][: len(val) - diff :][::-1])
                        break
                    else:
                        continue
                    break
                else:
                    continue
                break
            else:
                continue
            break
        # print(positions)

        return bitarray(message_bits)

    @staticmethod
    def get_max_capacity(img_in: PIL.Image.Image) -> int:
        pixels_in = img_in.load()
        if pixels_in is None:
            raise BaseException("pixels_in is none")
        capacity = 0
        for x in range(0, img_in.size[0] - SCALE_COEF, SCALE_COEF):
            for y in range(0, img_in.size[1] - SCALE_COEF, SCALE_COEF):
                bit_counts = tuple(
                    int(
                        numpy.log2(
                            max(
                                numpy.abs(
                                    pixels_in[x + x_i, y + y_i] - pixels_in[x, y]
                                ),
                                1,
                            )
                        )
                    )
                    for y_i in range(SCALE_COEF)
                    for x_i in range(SCALE_COEF)
                )
                capacity += sum(bit_counts)

        return capacity
```

```
import PIL.Image
from bitarray import bitarray


class LSBSeq:
    @staticmethod
    def inject_message(img_in: PIL.Image.Image, message: bitarray) ->
PIL.Image.Image:
        img = img_in.copy()
        pixels = img.load()
        last_x = None
        last_y = None
        msg_index = 0
        max_valid = len(message) - len(message) % 3
        if pixels is None:
            raise BaseException("pixels_in is None")
        for x in range(img.width):
            for y in range(img.height):
                if msg_index >= max_valid:
                    last_x = x
                    last_y = y
                    break
                byte = pixels[x, y]

                for index in range(3):
                    if message[msg_index + index]:
                        byte |= 1 << index
                    else:
                        byte &= ~(1 << index)

                pixels[x, y] = byte

                msg_index += 3
            else:
                continue
            break

        if last_x is not None and last_y is not None:
            last_bits_len = len(message) % 3
            byte = pixels[last_x, last_y]
            for index in range(last_bits_len):
                if message[-(last_bits_len - index)]:
                    byte |= 1 << index
                else:
                    byte &= ~(1 << index)
            pixels[last_x, last_y] = byte

        return img

    @staticmethod
    def get_max_capacity(img_in: PIL.Image.Image) -> int:
        return img_in.width * img_in.height // 3
```

```
import PIL.Image
import numpy
import scipy


class AUMPAnalysis:
```

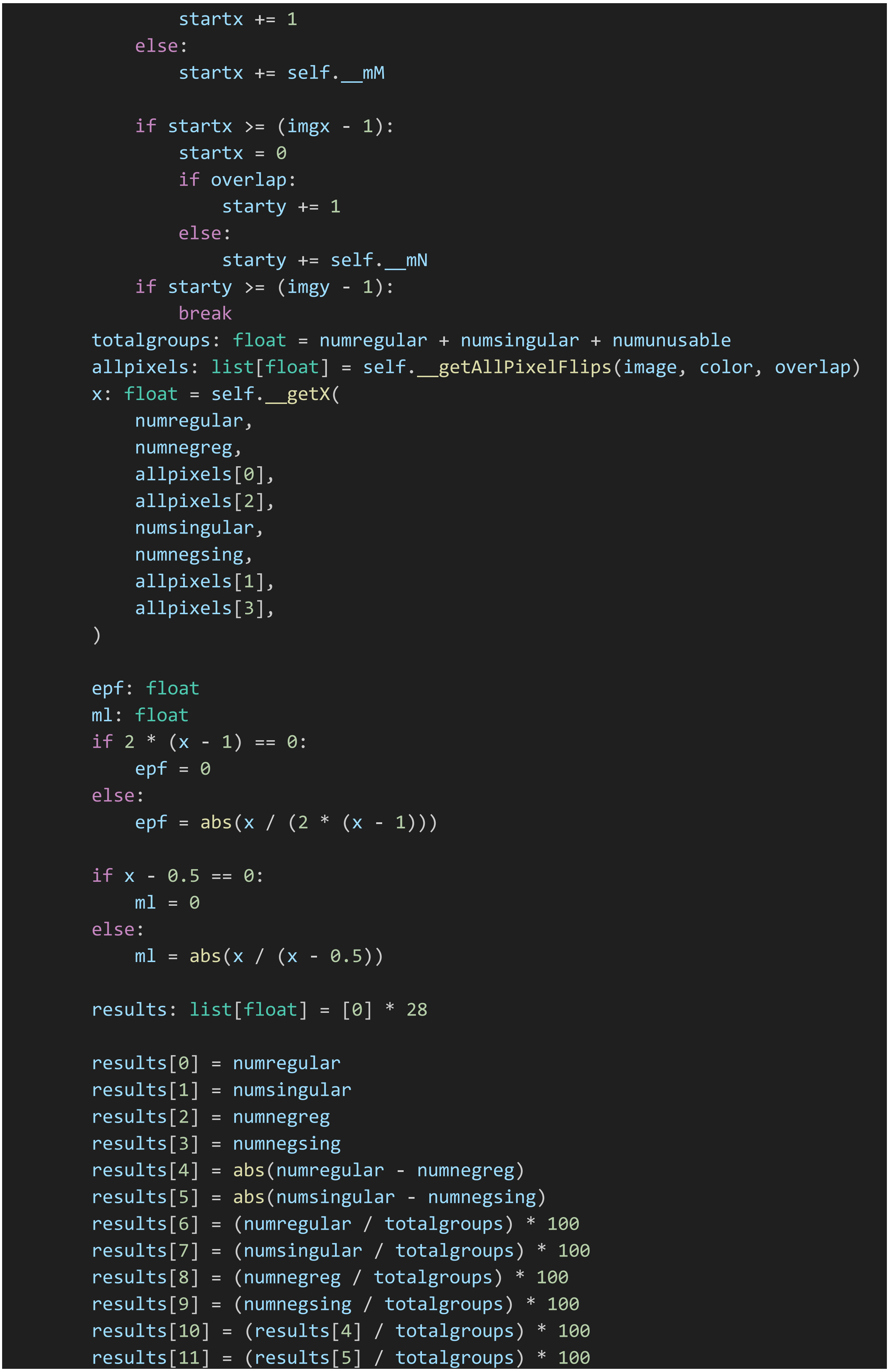


```python
    __WALL = 1

    @staticmethod
    def analyze(image: PIL.Image.Image, block_size: int, parameters: int) ->
bool:
        pixels = numpy.array(image, dtype=numpy.floating)
        scipy.io.savemat("array.mat", {"X": pixels})
        try:
            return (
                AUMPAnalysis.__aump(pixels, block_size, parameters)
                > AUMPAnalysis.__WALL
            )
        except BaseException:
            return False

    @staticmethod
    def __aump(X, m, d):
        Xpred, _, w = AUMPAnalysis.__pred_aump(X, m, d)
        r = X - Xpred
        Xbar = X + 1 - 2 * (X % 2)
        beta = numpy.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 = numpy.zeros((m, Kn))
        S = numpy.zeros_like(X)
        Xpred = numpy.zeros_like(X)

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

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

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

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

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

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

        return Xpred, S, w




import PIL.Image
import scipy.stats
import numpy


class ChiSquaredAnalysis:
    __MIN_BIN = 5
    __WALL = 0.5

    @staticmethod
    def analyze(image: PIL.Image.Image, block_size: int = 128) -> bool:
        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:
```

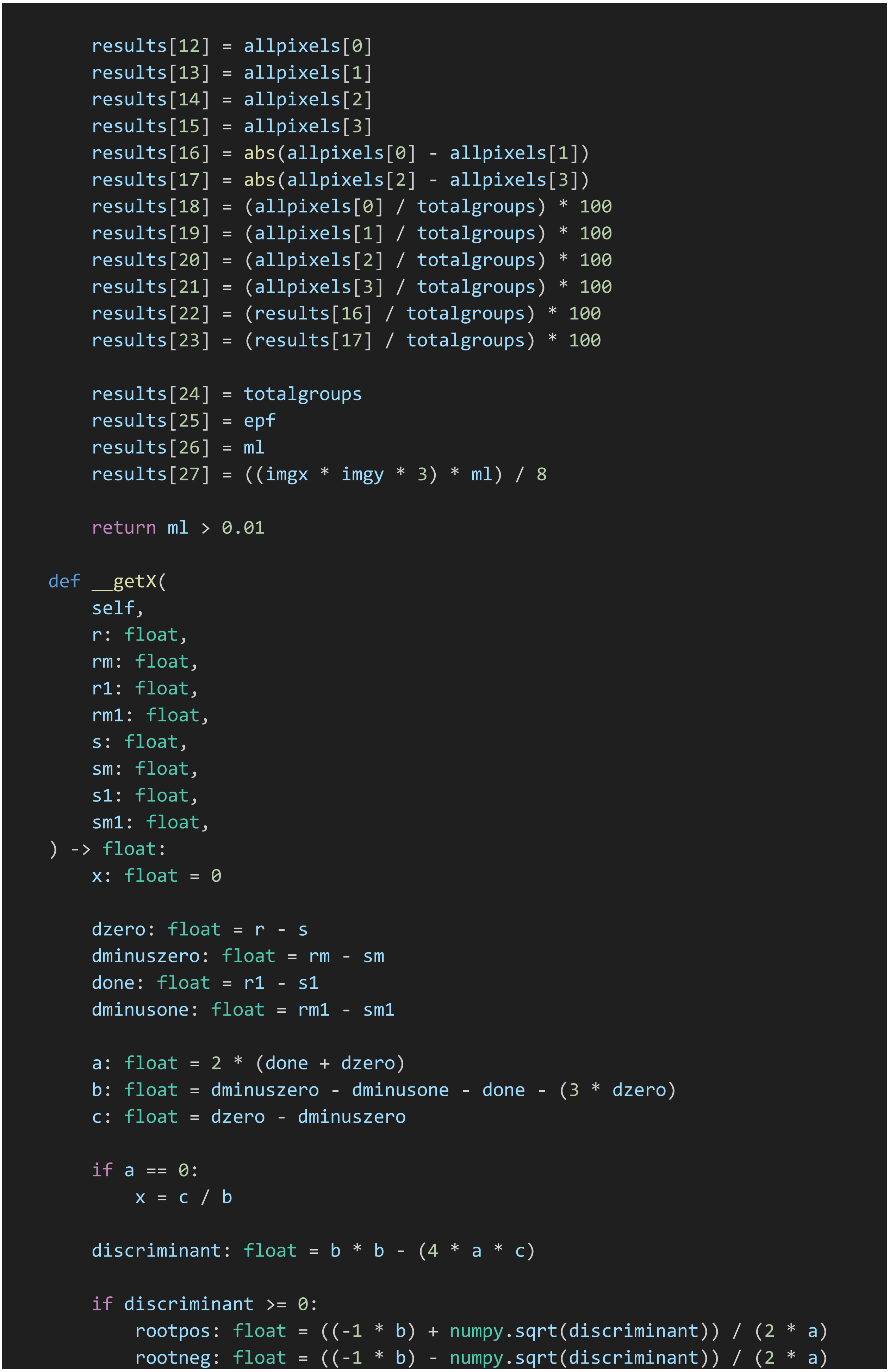


```
                        distribution_mean[index] = int(mean) + 1
                        distribution_mean[index + 1] = int(mean)

                # print(distribution_actual)
                # print(distribution_mean)
                # combine bins
                index = 0
                ChiSquaredAnalysis.__MIN_BIN = numpy.average(distribution_actual)
                while index < len(distribution_actual):
                    if distribution_actual[index] < ChiSquaredAnalysis.__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 >= ChiSquaredAnalysis.__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] < ChiSquaredAnalysis.__MIN_BIN:
                    distribution_actual[-2:] = [sum(distribution_actual[-2:])]
                    distribution_mean[-2:] = [sum(distribution_mean[-2:])]
                del index

                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
```

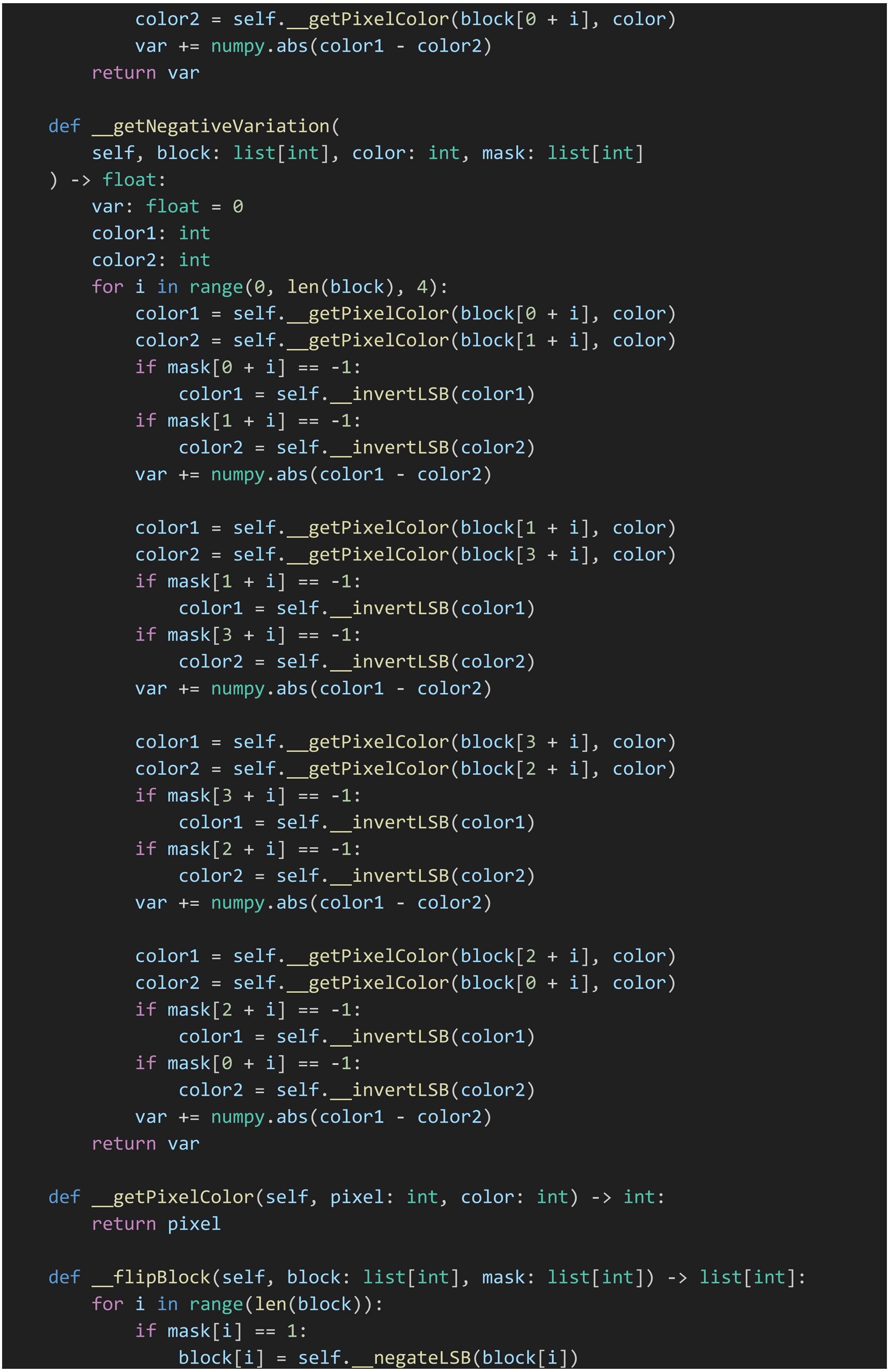
```
                        )[1]
                    )
        # print(outs)
        # print(numpy.average(outs))
        # outs = [x for x in outs if x > 0.000001]
        # return numpy.average(outs) > 0.5
        return numpy.average(outs) < ChiSquaredAnalysis.__WALL



import PIL.Image
import numpy


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: PIL.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
```

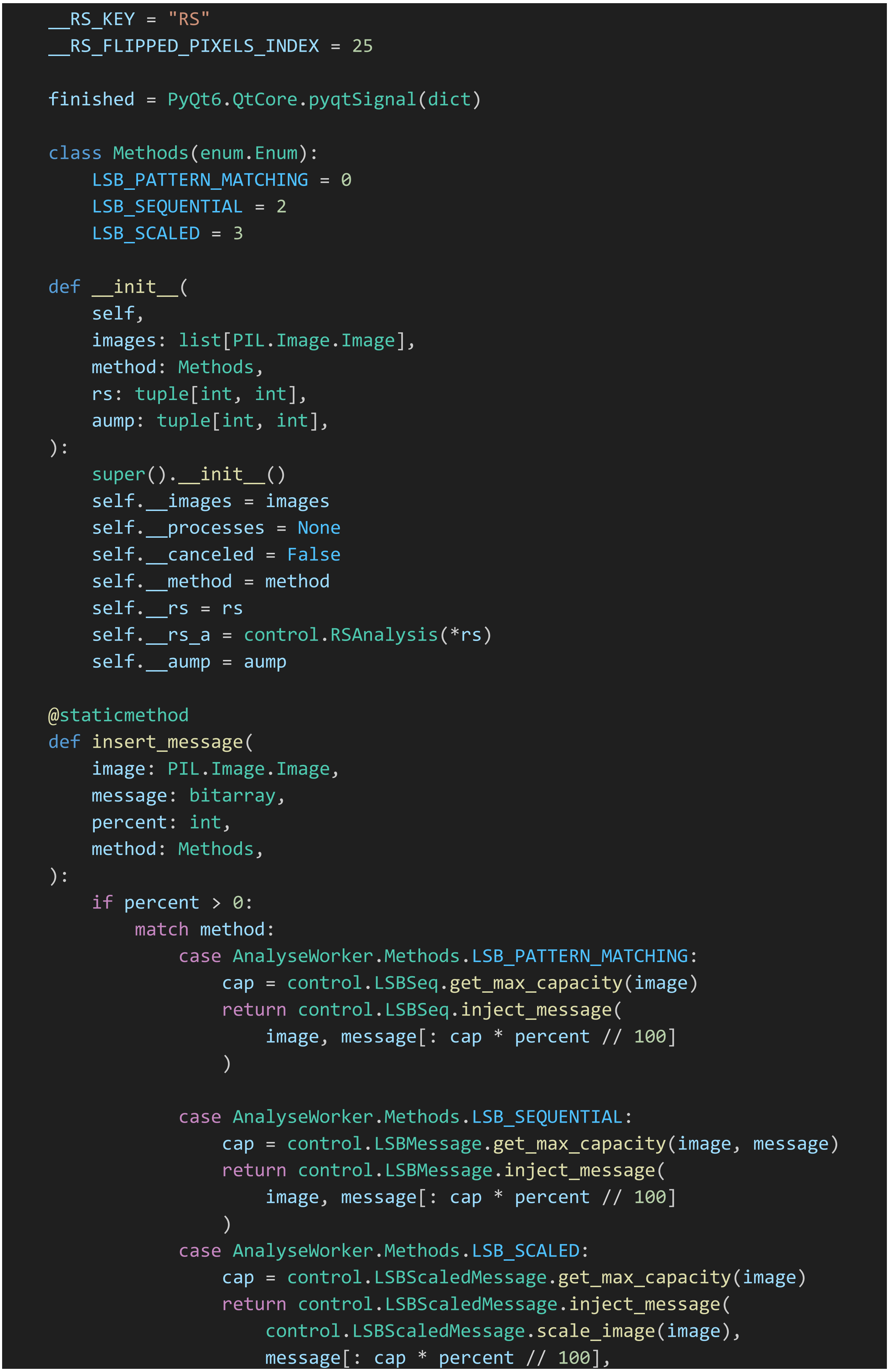


```
        results[12] = allpixels[0]
        results[13] = allpixels[1]
        results[14] = allpixels[2]
        results[15] = allpixels[3]
        results[16] = abs(allpixels[0] - allpixels[1])
        results[17] = abs(allpixels[2] - allpixels[3])
        results[18] = (allpixels[0] / totalgroups) * 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

        return ml > 0.01

    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) + numpy.sqrt(discriminant)) / (2 * a)
            rootneg: float = ((-1 * b) - numpy.sqrt(discriminant)) / (2 * a)
            if numpy.abs(rootpos) <= numpy.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: PIL.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 += numpy.abs(color1 - color2)
            color1 = self.__getPixelColor(block[3 + i], color)
            color2 = self.__getPixelColor(block[2 + i], color)
            var += numpy.abs(color1 - color2)
            color1 = self.__getPixelColor(block[1 + i], color)
            color2 = self.__getPixelColor(block[3 + i], color)
            var += numpy.abs(color1 - color2)
            color1 = self.__getPixelColor(block[2 + i], color)
```

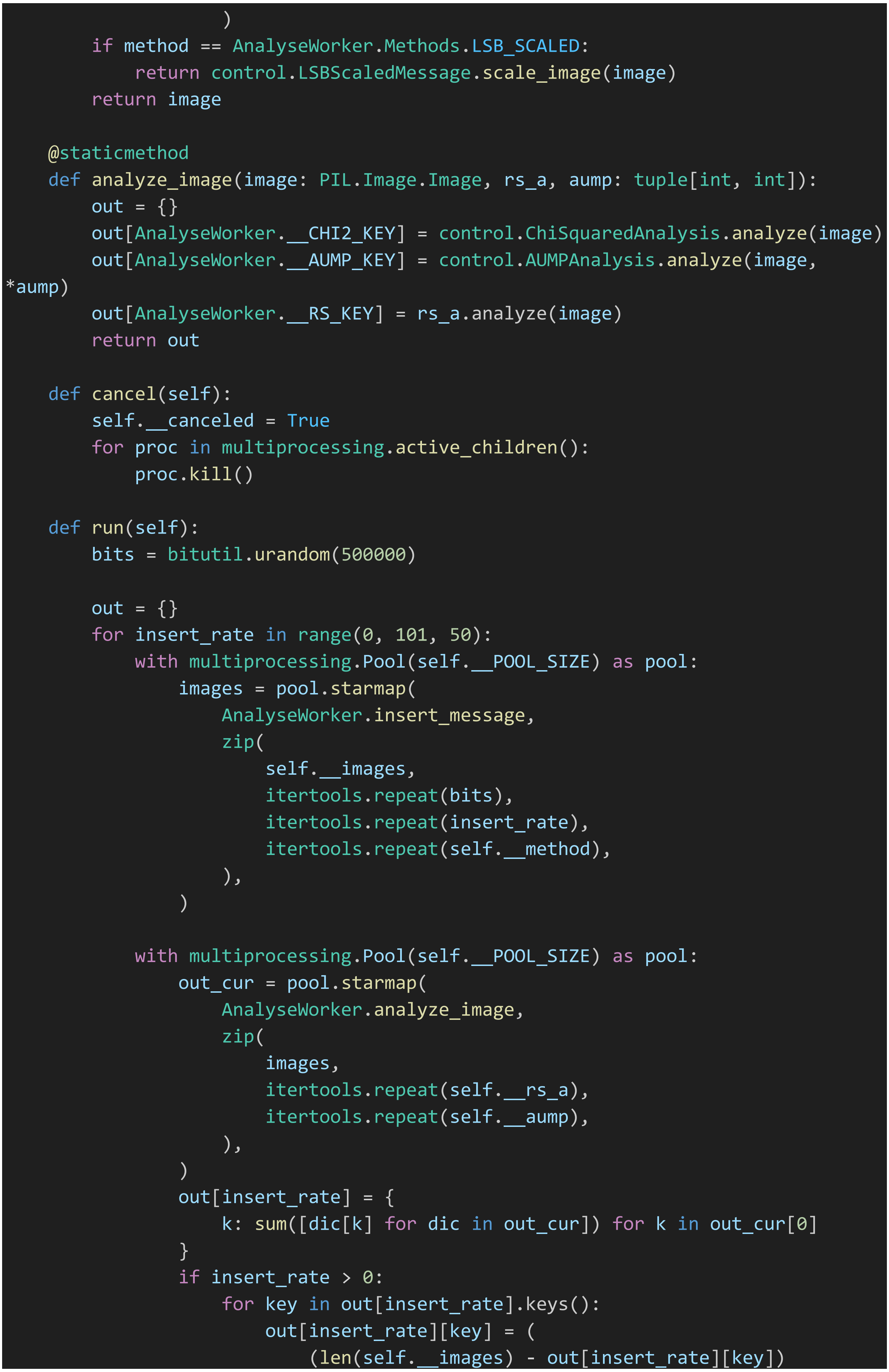


```
            color2 = self.__getPixelColor(block[0 + i], color)
            var += numpy.abs(color1 - color2)
        return var

    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 += numpy.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 += numpy.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 += numpy.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 += numpy.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]
```

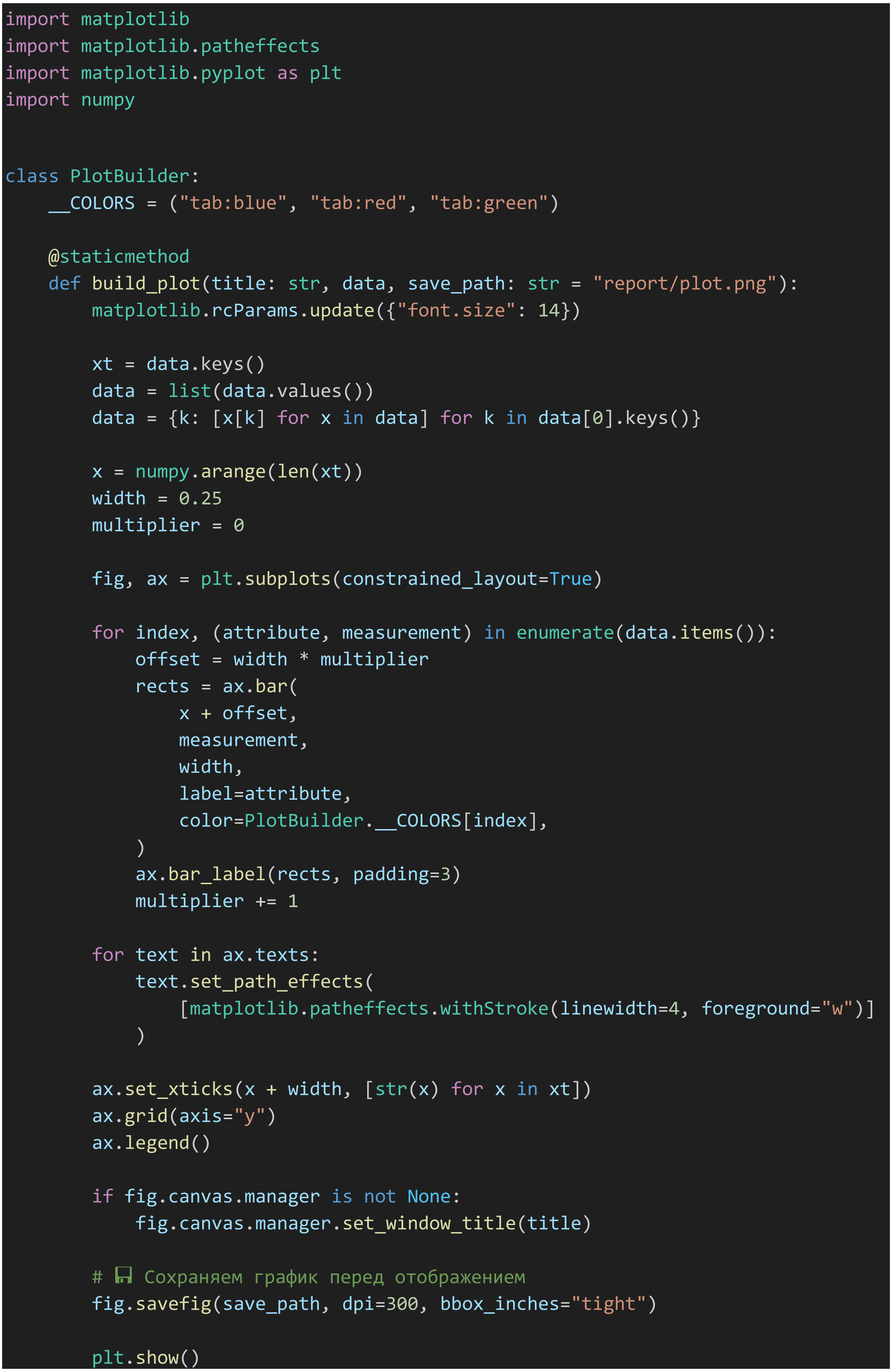
```
from ui.ImagesWidget import ImagesWidget
from ui.ButtonsWidget import ButtonsWidget

import control

import PIL.Image
import PIL.ImageQt

import PyQt6.QtWidgets
import PyQt6.QtGui
import PyQt6.QtCore

import multiprocessing
import pathlib
import json
import itertools
from bitarray import bitarray
import bitarray.util as bitutil

import enum


class AnalyseWorker(PyQt6.QtCore.QThread):
    __POOL_SIZE = 5
    __AUMP_KEY = "AUMP"
    __CHI2_KEY = "CHI2"
```



```
    __RS_KEY = "RS"
    __RS_FLIPPED_PIXELS_INDEX = 25

    finished = PyQt6.QtCore.pyqtSignal(dict)

    class Methods(enum.Enum):
        LSB_PATTERN_MATCHING = 0
        LSB_SEQUENTIAL = 2
        LSB_SCALED = 3

    def __init__(
        self,
        images: list[PIL.Image.Image],
        method: Methods,
        rs: tuple[int, int],
        aump: tuple[int, int],
    ):
        super().__init__()
        self.__images = images
        self.__processes = None
        self.__canceled = False
        self.__method = method
        self.__rs = rs
        self.__rs_a = control.RSAnalysis(*rs)
        self.__aump = aump

    @staticmethod
    def insert_message(
        image: PIL.Image.Image,
        message: bitarray,
        percent: int,
        method: Methods,
    ):
        if percent > 0:
            match method:
                case AnalyseWorker.Methods.LSB_PATTERN_MATCHING:
                    cap = control.LSBSeq.get_max_capacity(image)
                    return control.LSBSeq.inject_message(
                        image, message[: cap * percent // 100]
                    )

                case AnalyseWorker.Methods.LSB_SEQUENTIAL:
                    cap = control.LSBMessage.get_max_capacity(image, message)
                    return control.LSBMessage.inject_message(
                        image, message[: cap * percent // 100]
                    )
                case AnalyseWorker.Methods.LSB_SCALED:
                    cap = control.LSBScaledMessage.get_max_capacity(image)
                    return control.LSBScaledMessage.inject_message(
                        control.LSBScaledMessage.scale_image(image),
                        message[: cap * percent // 100],
```



```
                    )
        if method == AnalyseWorker.Methods.LSB_SCALED:
            return control.LSBScaledMessage.scale_image(image)
        return image

    @staticmethod
    def analyze_image(image: PIL.Image.Image, rs_a, aump: tuple[int, int]):
        out = {}
        out[AnalyseWorker.__CHI2_KEY] = control.ChiSquaredAnalysis.analyze(image)
        out[AnalyseWorker.__AUMP_KEY] = control.AUMPAnalysis.analyze(image,
*aump)
        out[AnalyseWorker.__RS_KEY] = rs_a.analyze(image)
        return out

    def cancel(self):
        self.__canceled = True
        for proc in multiprocessing.active_children():
            proc.kill()

    def run(self):
        bits = bitutil.urandom(500000)

        out = {}
        for insert_rate in range(0, 101, 50):
            with multiprocessing.Pool(self.__POOL_SIZE) as pool:
                images = pool.starmap(
                    AnalyseWorker.insert_message,
                    zip(
                        self.__images,
                        itertools.repeat(bits),
                        itertools.repeat(insert_rate),
                        itertools.repeat(self.__method),
                    ),
                )

            with multiprocessing.Pool(self.__POOL_SIZE) as pool:
                out_cur = pool.starmap(
                    AnalyseWorker.analyze_image,
                    zip(
                        images,
                        itertools.repeat(self.__rs_a),
                        itertools.repeat(self.__aump),
                    ),
                )
                out[insert_rate] = {
                    k: sum([dic[k] for dic in out_cur]) for k in out_cur[0]
                }
                if insert_rate > 0:
                    for key in out[insert_rate].keys():
                        out[insert_rate][key] = (
                            (len(self.__images) - out[insert_rate][key])
                            * 100
                            / len(self.__images)
                        )
                else:
                    for key in out[insert_rate].keys():
                        out[insert_rate][key] = (
                            out[insert_rate][key] * 100 / len(self.__images)
                        )

            if self.__canceled:
                self.finished.emit({})
                return
        # out = {k: [dic[k] for dic in out] for k in out[0]}
        # out = {
        #     key: {index * 25: out[key][index] for index in
range(len(out[key]))}
        #     for key in out.keys()
        # }
        self.finished.emit(out)


class MainWindow(PyQt6.QtWidgets.QMainWindow):
    __WINDOW_TITLE = "Steganography Task 6"

    def __init__(self):
        super().__init__()
        self.setWindowTitle(self.__WINDOW_TITLE)
        self.populate()
        self.connect_buttons()
        self.apply_styles()

    def apply_styles(self):
        self.setStyleSheet("""
            QMainWindow {
                background-color: #F9FBFD;
            }
            QTextBrowser {
                border: 1px solid #CCC;
                border-radius: 6px;
                background-color: #FFF;
                padding: 10px;
                font-family: Consolas, monospace;
                font-size: 13px;
            }
        """)

    def show_progress_indicator(self):
        self.__progress_indicator = PyQt6.QtWidgets.QProgressDialog(self)
        self.__progress_indicator.setWindowModality(
            PyQt6.QtCore.Qt.WindowModality.WindowModal
        )
        self.__progress_indicator.setRange(0, 0)
        self.__progress_indicator.setAttribute(
            PyQt6.QtCore.Qt.WidgetAttribute.WA_DeleteOnClose
        )
        self.__progress_indicator.setWindowTitle("Processing...")
        self.__progress_indicator.setLabelText("Analyzing images, please wait...")
        self.__progress_indicator.show()

    def hide_progress_indicator(self):
        self.__progress_indicator.close()

    def connect_buttons(self):
        self.__worker_thread = None

        def open_images_event():
            paths = PyQt6.QtWidgets.QFileDialog.getOpenFileNames(
                parent=self, caption="Select images to open", filter="BMP (*.bmp)"
            )[0]
            if paths:
                self.__images_before_widget.images = [PIL.Image.open(x) for x in paths]

        def analyze_images_event():
            def worker_finished(out: dict):
                self.hide_progress_indicator()
                self.__result_text_browser.setText(json.dumps(out, indent=2))
                control.PlotBuilder.build_plot("Result", out)

            if not self.__images_before_widget.images:
                PyQt6.QtWidgets.QMessageBox.warning(self, "Warning", "No images loaded!")
                return

            self.show_progress_indicator()
            self.__worker_thread = AnalyseWorker(
                self.__images_before_widget.images,
                self.__buttons_widget.method_dropdown.currentData(),
                (
                    self.__buttons_widget.rs_m_spinbox.value(),
                    self.__buttons_widget.rs_n_spinbox.value(),
                ),
                (
                    self.__buttons_widget.aump_block_size_spinbox.value(),
                    self.__buttons_widget.aump_parameter_spinbox.value(),
                ),
            )
            self.__worker_thread.finished.connect(worker_finished)
            self.__progress_indicator.canceled.connect(self.__worker_thread.quit)
            self.__worker_thread.start()
        def save_results_event():
            try:
                pathlib.Path("out.json").write_text(
                    self.__result_text_browser.toPlainText()
                )
                PyQt6.QtWidgets.QMessageBox.information(self, "Success", "Results saved to out.json")
            except Exception as e:
                PyQt6.QtWidgets.QMessageBox.critical(self, "Error", f"Failed to save results:\n{e}")

        self.__buttons_widget.open_images_button.clicked.connect(open_images_event)
        self.__buttons_widget.analyze_images_button.clicked.connect(analyze_images_event)
        self.__buttons_widget.save_resuts_button.clicked.connect(save_results_event)

    def populate(self):
        self.__central_widget = PyQt6.QtWidgets.QWidget(self)
        self.__main_layout = PyQt6.QtWidgets.QVBoxLayout(self.__central_widget)
        self.__main_layout.setContentsMargins(15, 15, 15, 15)
        self.__main_layout.setSpacing(15)
        self.__central_widget.setLayout(self.__main_layout)
        self.setCentralWidget(self.__central_widget)

        self.__upper_layout_widget = PyQt6.QtWidgets.QWidget(self)
        self.__upper_layout = PyQt6.QtWidgets.QHBoxLayout(self.__upper_layout_widget)
        self.__upper_layout.setSpacing(20)
        self.__upper_layout_widget.setLayout(self.__upper_layout)
        self.__main_layout.addWidget(self.__upper_layout_widget, stretch=1)

        self.__images_before_widget = ImagesWidget(self.__upper_layout_widget)
        self.__buttons_widget = ButtonsWidget(self.__upper_layout_widget)
        self.__upper_layout.addWidget(self.__images_before_widget, stretch=3)
        self.__upper_layout.addWidget(self.__buttons_widget, stretch=1)

        self.__result_text_browser = PyQt6.QtWidgets.QTextBrowser(self.__central_widget)
        self.__result_text_browser.setMinimumHeight(150)
        self.__main_layout.addWidget(self.__result_text_browser)

        # Заполнить комбобокс методами с удобочитаемыми названиями
        self.__buttons_widget.method_dropdown.clear()
        for method in AnalyseWorker.Methods:
            # Отображать название метода красиво
            self.__buttons_widget.method_dropdown.addItem(method.name.replace('_', ' ').title(), method)
```



```python
import matplotlib
import matplotlib.patheffects
import matplotlib.pyplot as plt
import numpy


class PlotBuilder:
    __COLORS = ("tab:blue", "tab:red", "tab:green")

    @staticmethod
    def build_plot(title: str, data, save_path: str = "report/plot.png"):
        matplotlib.rcParams.update({"font.size": 14})

        xt = data.keys()
        data = list(data.values())
        data = {k: [x[k] for x in data] for k in data[0].keys()}

        x = numpy.arange(len(xt))
        width = 0.25
        multiplier = 0

        fig, ax = plt.subplots(constrained_layout=True)

        for index, (attribute, measurement) in enumerate(data.items()):
            offset = width * multiplier
            rects = ax.bar(
                x + offset,
                measurement,
                width,
                label=attribute,
                color=PlotBuilder.__COLORS[index],
            )
            ax.bar_label(rects, padding=3)
            multiplier += 1

        for text in ax.texts:
            text.set_path_effects(
                [matplotlib.patheffects.withStroke(linewidth=4, foreground="w")]
            )

        ax.set_xticks(x + width, [str(x) for x in xt])
        ax.grid(axis="y")
        ax.legend()

        if fig.canvas.manager is not None:
            fig.canvas.manager.set_window_title(title)

        # 💾 Сохраняем график перед отображением
        fig.savefig(save_path, dpi=300, bbox_inches="tight")

        plt.show()
```