**LSB卡方分析**

## 1. 卡方分析简介

卡方分析是一种统计检验方法，用于评估观察到的数据与期望数据之间的差异是否显著。在隐写分析中，卡方检验常用于检测图像中隐写操作引入的统计异常，特别是针对像素值或变换域系数的分布变化。

### 1.1 卡方分析在隐写分析中的应用

在隐写分析中，卡方检验可以用于：

（1）空间域检测：

单比特替换检测：分析图像中 LSB（最低有效位）替换后的像素值分布是否与原始图像的分布有显著差异。

多比特替换检测：扩展至多位替换，检测更复杂的隐写操作。

（2）变换域检测：

频域系数检测：如 DCT（离散余弦变换）系数的统计分析，用于检测 JPEG 图像中的隐写操作（如 EzStego）。

### 1.2 卡方分析原理

卡方分析的步骤是：图像中像素灰度值为j，其中0≤ j ≤255。如果载体图像未经隐写，像素灰度值为2j的数目和像素灰度值为（2j+1）的数目会相差得很远。秘密信息在嵌入之前往往经过加密，可以看作是0、1 随机分布的比特流，而且值为0与1 的可能性大致都是1/2。如果秘密信息完全替代载体图像的最低位，那么两个的值会比较接近，可以根据这个性质判断图像是否经过隐写。下图显示了在隐写率为1时，图片像素值频率直方图的变化。可以看到，在经过LSB隐写后，像素灰度值频率分布直方图会有较大变化。像素灰度值为2j的数目和像素灰度值为（2j+1）的数目之间的差距会缩小，大致趋于相等。

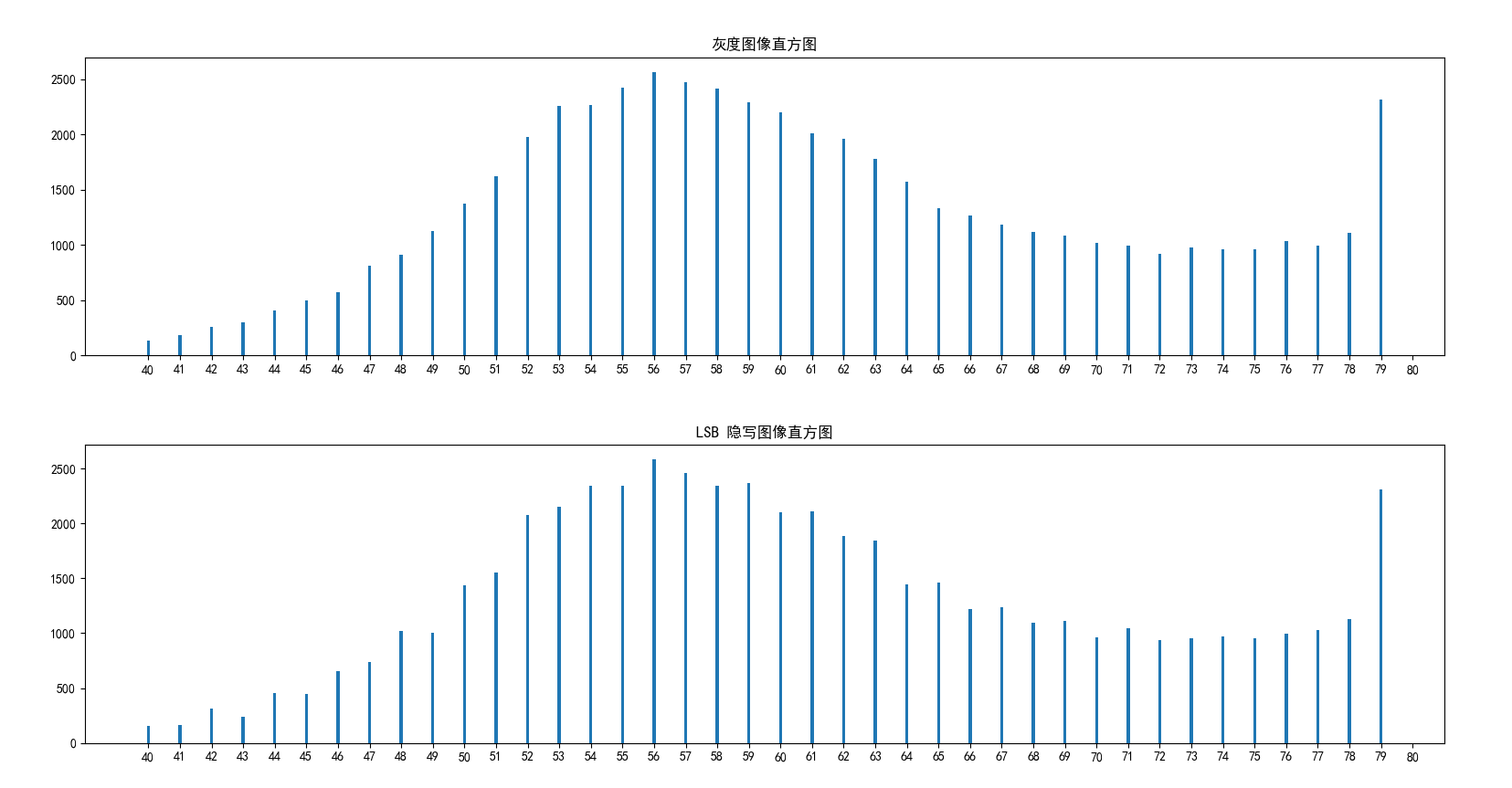


图1 LSB隐写前后图片像素值频率直方图对比

我们假设q是一个像素被选中用于隐藏信息的概率；表示载体图像中，值为j的像素个数，表示隐写图像中，值为j的像素个数。我们可以提出下列假设：对于，有个像素携带秘密信息，其中大约个像素的最低比特与消息相同，不需要修改，剩下的个像素变为2i+1；对于，大约个像素的最低比特变为2i+1。由此我们得到携密图像中的和期望值：

当时有（因为LSB只会导致变为或者变为，以及像素值不改变，因此有）

由此可以将计算所得的期望值与实际2i像素的值进行比较，随着隐写率增加，期望值与实际值趋近相等。由此，统计量s符合自由度为k-1的卡方分布。若s小于等于我们设定的阈值，即判定图像隐写。

## 2. 实现卡方分析算法

卡方的实现主要分为空间域检测和变换域检测，本次实验以空间域检测为主。

### 2.1 空间域卡方分析

在空间域卡方分析中，我们主要关注图像像素值的分布。隐写操作通常会打乱 LSB 的分布，导致统计特性发生变化。卡方检验可以衡量这种变化是否显著。

### 2.2实现步骤

1. **计算图像的像素值分布**。
2. **计算卡方统计量**：
3. **判断统计显著性**：较高的卡方值表明分布差异显著，可能不存在隐写操作，反之可能存在隐写操作。我们使用自由度为(k-1)的卡方分布的概率分布函数，代入统计量，计算概率值。

### 2.3 实现代码

# -\*- coding: utf-8 -\*-

import math

import random

import cv2  # 用于图像缩放

import numpy as np

import matplotlib.pyplot as plt

from PIL import Image

from scipy.stats import chi2 as chi2\_dist

from skimage.metrics import structural\_similarity as ssim

# 设置固定种子以确保可重复性

random.seed(42)

np.random.seed(42)

plt.rcParams['font.sans-serif']=['SimHei'] # 显示中文

# LSB隐写（随机选择p\*N个像素进行嵌入）

def LSB\_embed\_random(fig, p, bit\_level=1):

    '''

    LSB连续嵌入

    :param fig: 待嵌入图片

    :param p: 隐写率

    :param bit\_level: 每一像素嵌入多少比特信息

    :return: fig\_embedded

    '''

    height, width = fig.shape

    N = height \* width

    if math.floor(p) != 0 and p != 1:

        raise Exception('P should be between 0 and 1.')

    if bit\_level <= 0 or bit\_level >= 8:

        raise Exception('Bit level should be between 0 and 8.')

    num\_embed = int(p \* N)

    if num\_embed == 0:

        return fig.copy()

    # 生成所有像素的索引

    all\_indices = list(range(N))

    # 随机选择num\_embed个像素

    embed\_indices = random.sample(all\_indices, num\_embed)

    # 生成随机比特序列

    bits = [random.randint(0, 2 \*\* bit\_level - 1) for \_ in range(num\_embed)]

    fig\_embedded = fig.copy().flatten()

    mask = (255 >> bit\_level) << bit\_level

    for idx, bit in zip(embed\_indices, bits):

        fig\_embedded[idx] = fig\_embedded[idx] & mask ^ bit

    fig\_embedded = fig\_embedded.reshape(height, width)

    return fig\_embedded

# LSB隐写（连续p\*N个像素嵌入）

def LSB\_embed\_continuous(fig, p, bit\_level=1, flag=True):

    '''

    LSB连续嵌入

    :param fig: 待嵌入图片

    :param p: 隐写率

    :param bit\_level: 每一像素嵌入多少比特信息

    :param flag: 是否随机选取嵌入的起始坐标，True表示随机选取, False 表示从0开始

    :return: fig\_embedded

    '''

    height, width = fig.shape

    N = height \* width

    if math.floor(p) != 0 and p != 1:

        raise Exception('P should be between 0 and 1.')

    if bit\_level <= 0 or bit\_level >= 8:

        raise Exception('Bit level should be between 0 and 8.')

    num\_embed = int(p \* N)

    if num\_embed == 0:

        return fig.copy()

    bits = [random.randint(0, 2 \*\* bit\_level - 1) for \_ in range(num\_embed)]

    start\_idx = random.randint(0, N - num\_embed)

    if not flag:

        start\_idx = 0

    fig\_embedded = fig.copy().flatten()

    mask = (255 >> bit\_level) << bit\_level

    for idx, bit in zip(list(range(start\_idx, start\_idx + num\_embed)), bits):

        fig\_embedded[idx] = fig\_embedded[idx] & mask ^ bit

    fig\_embedded = fig\_embedded.reshape(height, width)

    return fig\_embedded

# 图像分块

def image\_partition(fig, block\_size=(8, 8)):

    height, width = fig.shape

    block\_h, block\_w = block\_size

    blocks = []

    for i in range(0, height, block\_h):

        for j in range(0, width, block\_w):

            block = fig[i:i + block\_h, j:j + block\_w]

            # 如果块大小不符合，进行填充

            if block.shape != (block\_h, block\_w):

                pad\_h = block\_h - block.shape[0]

                pad\_w = block\_w - block.shape[1]

                block = np.pad(block, ((0, pad\_h), (0, pad\_w)), 'constant', constant\_values=0)

            blocks.append(block)

    return blocks

# 计算卡方统计量（支持多比特替换检测）

def chi\_square\_test(original\_fig, stego\_fig, bit\_level=2):

    """

    对比原始图像和隐写图像的像素值分布，计算卡方统计量

    :param original\_fig: 原始图像（二维numpy数组）

    :param stego\_fig: 隐写图像（二维numpy数组）

    :param bit\_level: LSB位数（默认为2）

    :return: 卡方统计量

    """

    # 提取指定位的LSB

    mask = (1 << bit\_level) - 1

    original\_bits = original\_fig & mask

    stego\_bits = stego\_fig & mask

    # 计算频数

    original\_counts = np.bincount(original\_bits.flatten(), minlength=2 \*\* bit\_level)

    stego\_counts = np.bincount(stego\_bits.flatten(), minlength=2 \*\* bit\_level)

    # 期望频数（假设无隐写，即 stego\_counts = original\_counts）

    expected = original\_counts

    observed = stego\_counts

    # 避免期望频数为0

    nonzero = expected != 0

    chi2 = np.sum(((observed[nonzero] - expected[nonzero]) \*\* 2) / expected[nonzero])

    return chi2

def chi\_square\_pvalue(chi2, bit\_level=2):

    """

    计算卡方统计量对应的p值

    :param chi2: 卡方统计量

    :param bit\_level: LSB位数

    :return: p值

    """

    df = (2 \*\* bit\_level) - 1  # 自由度

    p\_value = 1 - chi2\_dist.cdf(chi2, df)

    return p\_value

def chi\_square(martix):

    '''

    计算卡方统计量对应的p值（已支持多比特）

    :param matrix: 待进行卡方检验的矩阵

    :return: r: 统计值; p: 概率值

    小小吐槽：前面同学写的代码似乎有问题，如果我可以获取原始图像，我为什么还需要卡方检测来检测图片是否隐写呢？

    '''

    count = np.zeros(256,dtype=int)

    for i in range(len(martix)):

        for j in range(len(martix[0])):

            count[martix[i][j]] += 1

    # H[2i]的观测值

    h2i = count[2:255:2]

    # H[2i]的期望值H[2i] = (H[2i] + H[2i + 1]) / 2

    h2is = (h2i+count[3:256:2])/2

    filter= (h2is!=0)

    k = sum(filter)

    idx = np.zeros(k,dtype=int)

    for i in range(127):

        if filter[i] == True:

            idx[sum(filter[1:i])] = i

    r=sum(((h2i[idx] - h2is[idx])\*\*2) / (h2is[idx]))

    p = 1-chi2\_dist.cdf(r,k-1)

    return (r, p)

def plot\_bit\_histogram(original\_fig, stego\_fig, bit\_level=1):

    """

    绘制原始图像和隐写图像的像素频数分布直方图

    :param original\_fig: 原始图像

    :param stego\_fig: 隐写图像

    :param bit\_level: LSB位数

    """

    # 防御式编程（其实是懒得改了）

    original\_bits = original\_fig

    stego\_bits = stego\_fig

    original\_counts = np.bincount(original\_bits.flatten(), minlength=256)[50: 90]

    stego\_counts = np.bincount(stego\_bits.flatten(), minlength=256)[50: 90]

    plt.subplot(211)

    plt.subplots\_adjust(hspace=0.3)

    plt.title("灰度图像直方图")

    plt.hist(original\_bits.flatten(), bins=np.arange(40, 81, 1),

                rwidth=0.1, align="left")

    plt.xticks(range(40, 81))

    plt.subplot(212)

    plt.title("LSB 隐写图像直方图")

    plt.hist(stego\_bits.flatten(), bins=np.arange(40, 81, 1), rwidth=0.1,

                align="left")

    plt.xticks(range(40, 81))

    plt.show()

# 计算PSNR

def calculate\_psnr(original, stego):

    mse = np.mean((original - stego) \*\* 2)

    if mse == 0:

        return float('inf')

    PIXEL\_MAX = 255.0

    return 20 \* math.log10(PIXEL\_MAX / math.sqrt(mse))

# 计算SSIM

def calculate\_ssim(original, stego):

    return ssim(original, stego, data\_range=stego.max() - stego.min())

### 2.4 主要函数说明

**LSB隐写：**

LSB\_embed\_random函数用于在图像中随机嵌入LSB隐写。

LSB\_embed\_continuous函数用于在图像中连续地以行优先嵌入秘密信息，根据选择模式不同可以分为随机选择开始索引以及选择图片第一个像素作为起始嵌入信息的像素。

**图像分块：**

image\_partition函数用于将图像按指定块大小分割，大小不足的会进行补全。

**卡方检验：**

chi\_square函数用于计算卡方统计量，以及根据统计量和卡方分布的概率分布函数计算概率。

**位分布直方图绘制：**

plot\_bit\_histogram函数用于绘制原始图像和隐写图像在指定LSB位数下的像素值分布直方图。

main\_comparison函数现执行以下步骤：

1. **读取图像**：读取并转换为灰度图像。
2. **定义隐写率**：定义从0.0到0.9的隐写率，步长为0.1。
3. **进行多次实验**：对于每个隐写率，进行M=5次实验。每次实验中，嵌入隐写信息，并计算卡方统计量。
4. **结果统计与输出**：计算每个隐写率下卡方统计量的平均值和标准差，并打印结果。
5. **绘图**：绘制隐写率与卡方统计量及p值的关系图。绘制一个示例隐写率（p=0.3）的LSB位分布直方图。

## 3.多比特位分析实现

多比特位分析（Multi-bit Level Analysis）在隐写分析中是指对图像中最低的多个有效位的分布进行统计分析，以检测潜在的隐写操作。相比于单比特位分析，多比特位分析能够捕捉更多的隐写信息，从而提高隐写检测的敏感性和准确性。

在代码中，多比特位分析主要通过chi\_square\_test函数实现。以下是对该函数及其相关部分的详细解释，以及多比特位分析的实现原理。

### 3.1 多比特位分析的实现原理

在LSB隐写技术中，隐写信息通常嵌入到图像像素的最低有效位中。例如，1比特LSB隐写将信息嵌入到每个像素的最低位，而2比特LSB隐写则将信息嵌入到最低两位。

多比特位分析旨在检测图像中多个LSB位的统计分布是否与原始图像一致。通过比较隐写图像与原始图像在多个LSB位上的频数分布，可以判断是否存在隐写操作。

卡方检验（Chi-Square Test）用于比较观察到的频数分布与期望的频数分布之间的差异。在多比特位分析中：

提取多比特位：从图像中提取指定数量的LSB位。

计算频数分布：统计每种可能的比特组合出现的次数。

计算卡方统计量：衡量观察频数与期望频数之间的差异。

计算p值：判断卡方统计量的显著性，p值越小，差异越显著，表明可能存在隐写操作。

### 3.2 代码实现

def **chi\_square\_test**(original\_fig, stego\_fig, bit\_level=2):

    """

    对比原始图像和隐写图像的像素值分布，计算卡方统计量

    :param original\_fig: 原始图像（二维numpy数组）

    :param stego\_fig: 隐写图像（二维numpy数组）

    :param bit\_level: LSB位数（默认为2）

    :return: 卡方统计量

    """

*# 提取指定位的LSB*

    mask = (1 << bit\_level) - 1  *# 创建掩码，如bit\_level=2时mask=0b11*

    original\_bits = original\_fig & mask  *# 提取原始图像的LSB位*

    stego\_bits = stego\_fig & mask      *# 提取隐写图像的LSB位*

*# 计算频数*

    original\_counts = np.bincount(original\_bits.flatten(), minlength=2 \*\* bit\_level)

    stego\_counts = np.bincount(stego\_bits.flatten(), minlength=2 \*\* bit\_level)

*# 期望频数（假设无隐写，即 stego\_counts = original\_counts）*

    expected = original\_counts

    observed = stego\_counts

*# 避免期望频数为0*

    nonzero = expected != 0

    chi2 = np.sum(((observed[nonzero] - expected[nonzero]) \*\* 2) / expected[nonzero])

    return chi2

提取多比特位：使用位掩码mask = (1 << bit\_level) - 1生成一个掩码。例如，bit\_level=2时，mask=0b11。使用按位与操作original\_fig & mask和stego\_fig & mask提取原始图像和隐写图像的指定LSB位。

计算频数分布：np.bincount函数用于统计每种可能的比特组合出现的次数。minlength=2 \*\* bit\_level确保所有可能的比特组合都有对应的计数。

例如，bit\_level=2时，有4种可能的比特组合00, 01, 10, 11，对应的计数分别存储在original\_counts和stego\_counts中。

计算卡方统计量：

期望频数expected假设隐写操作未进行，即隐写图像的频数应与原始图像一致。卡方统计量chi2通过公式计算。为避免除以零错误，首先筛选出期望频数不为零的类别。

def **chi\_square\_pvalue**(chi2, bit\_level=2):

    """

    计算卡方统计量对应的p值

    :param chi2: 卡方统计量

    :param bit\_level: LSB位数

    :return: p值

    """

    df = (2 \*\* bit\_level) - 1  *# 自由度*

    p\_value = 1 - chi2\_dist.cdf(chi2, df)

    return p\_value

计算p值：使用scipy.stats库中的卡方分布累积分布函数（CDF）计算p值。p值表示观察到的卡方统计量在卡方分布中的位置，p值越小，说明差异越显著，可能存在隐写操作。

自由度计算：自由度df为(2 \*\* bit\_level) - 1。例如，bit\_level=2时，df=3。

def **plot\_bit\_histogram**(original\_fig, stego\_fig, bit\_level=2):

    """

    绘制原始图像和隐写图像的LSB位分布直方图

    :param original\_fig: 原始图像

    :param stego\_fig: 隐写图像

    :param bit\_level: LSB位数

    """

    mask = (1 << bit\_level) - 1

    original\_bits = original\_fig & mask

    stego\_bits = stego\_fig & mask

    original\_counts = np.bincount(original\_bits.flatten(), minlength=2 \*\* bit\_level)

    stego\_counts = np.bincount(stego\_bits.flatten(), minlength=2 \*\* bit\_level)

    labels = [f"{i:0{bit\_level}b}" for i in **range**(2 \*\* bit\_level)]

    x = np.arange(**len**(labels))  *# 标签位置*

    width = 0.35  *# 柱状图宽度*

    fig, ax = plt.subplots(figsize=(12, 6))

    rects1 = ax.bar(x - width/2, original\_counts, width, label='原始图像')

    rects2 = ax.bar(x + width/2, stego\_counts, width, label='隐写图像')

    ax.set\_xlabel('LSB位值')

    ax.set\_ylabel('频数')

    ax.set\_title(f'LSB{bit\_level}位分布比较')

    ax.set\_xticks(x)

    ax.set\_xticklabels(labels)

    ax.legend()

    plt.show()

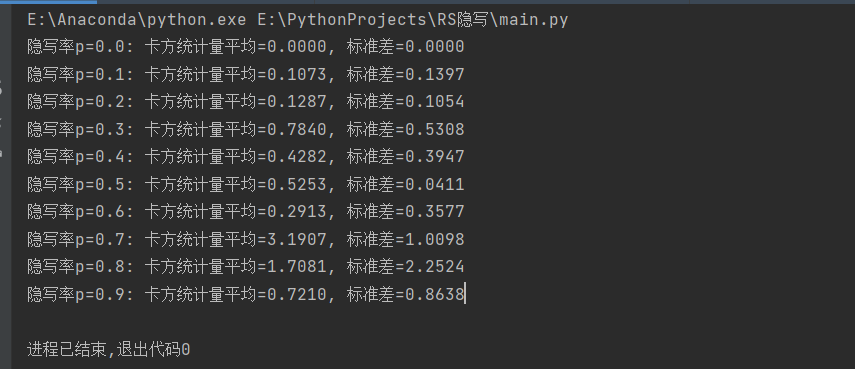
**绘制直方图**：对比原始图像和隐写图像在指定LSB位数下的频数分布。

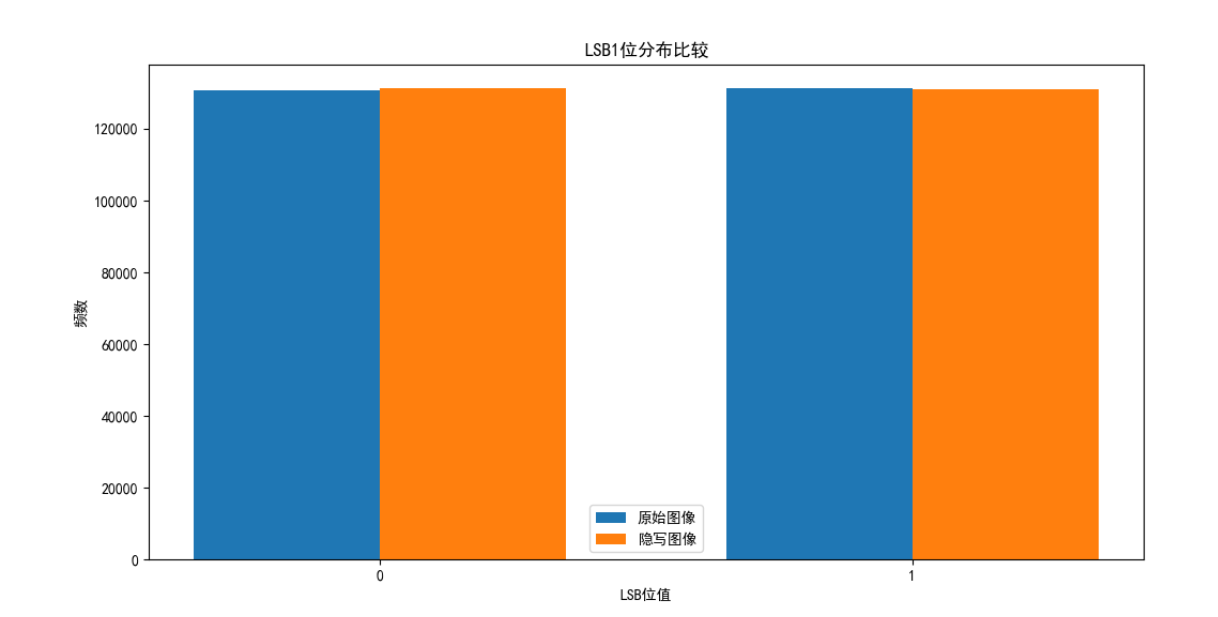
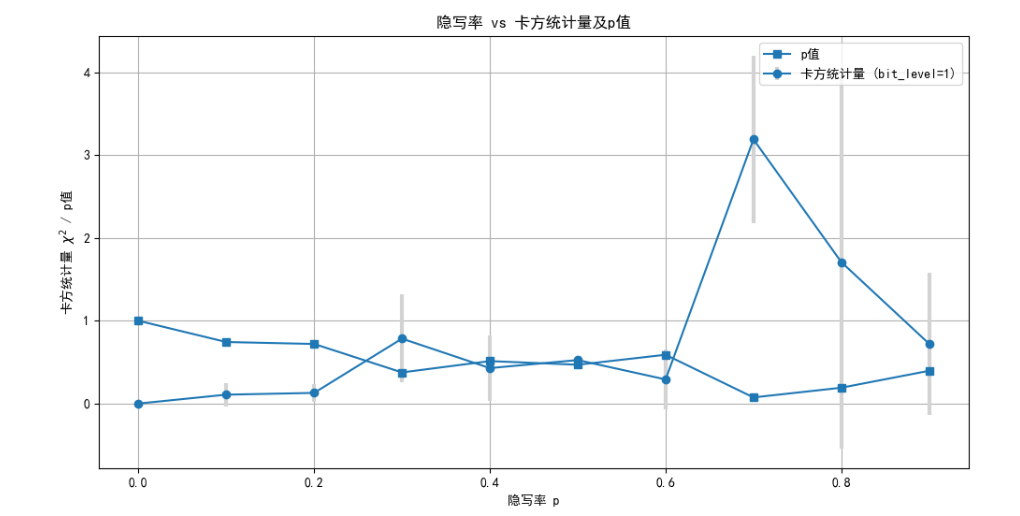
**标签生成**：根据bit\_level生成相应的二进制标签。例如，bit\_level=2时，标签为'00', '01', '10', '11'。

**直方图展示**：使用双柱状图对比两者的频数分布，便于直观观察差异

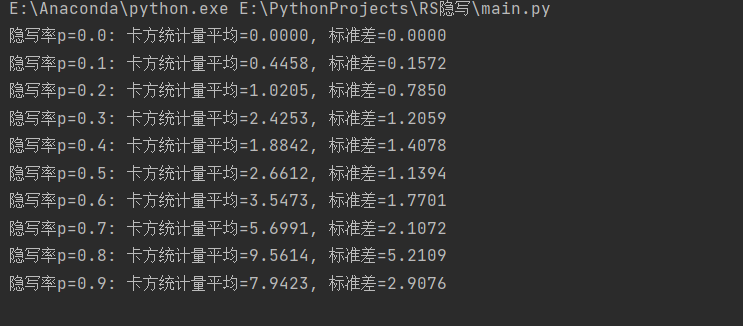
## 4.实验结果

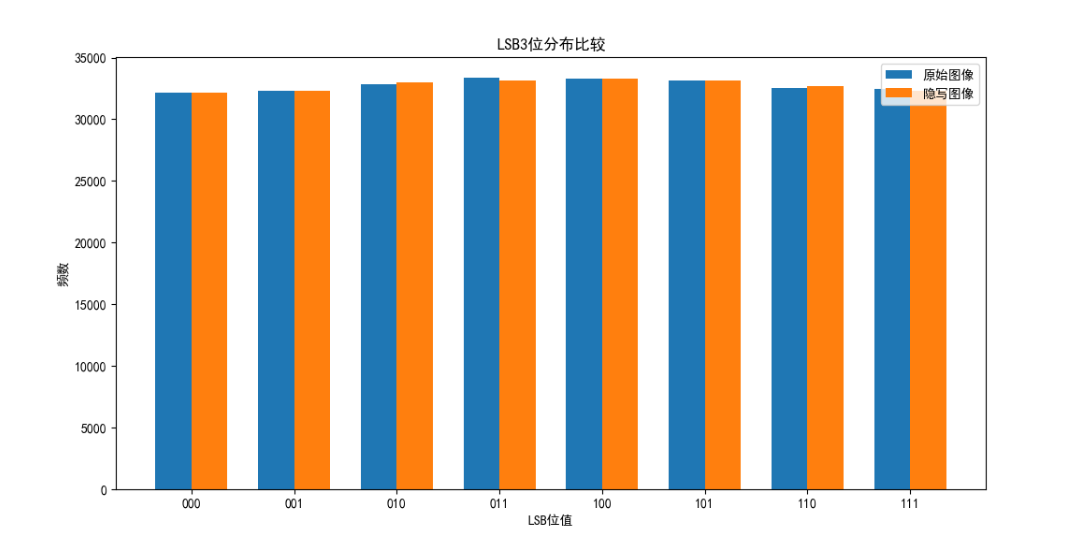
当比特位为1时

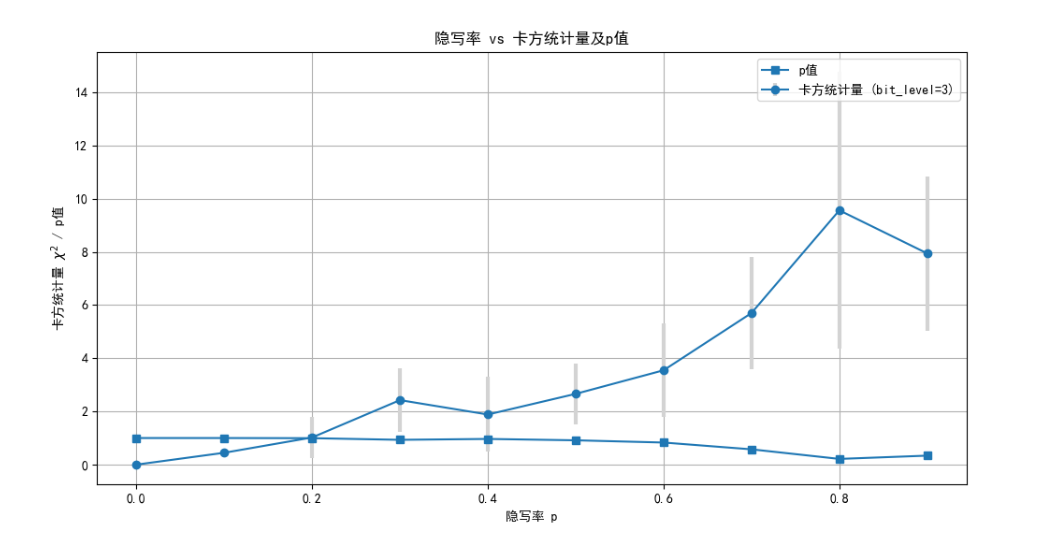


当比特位为3时

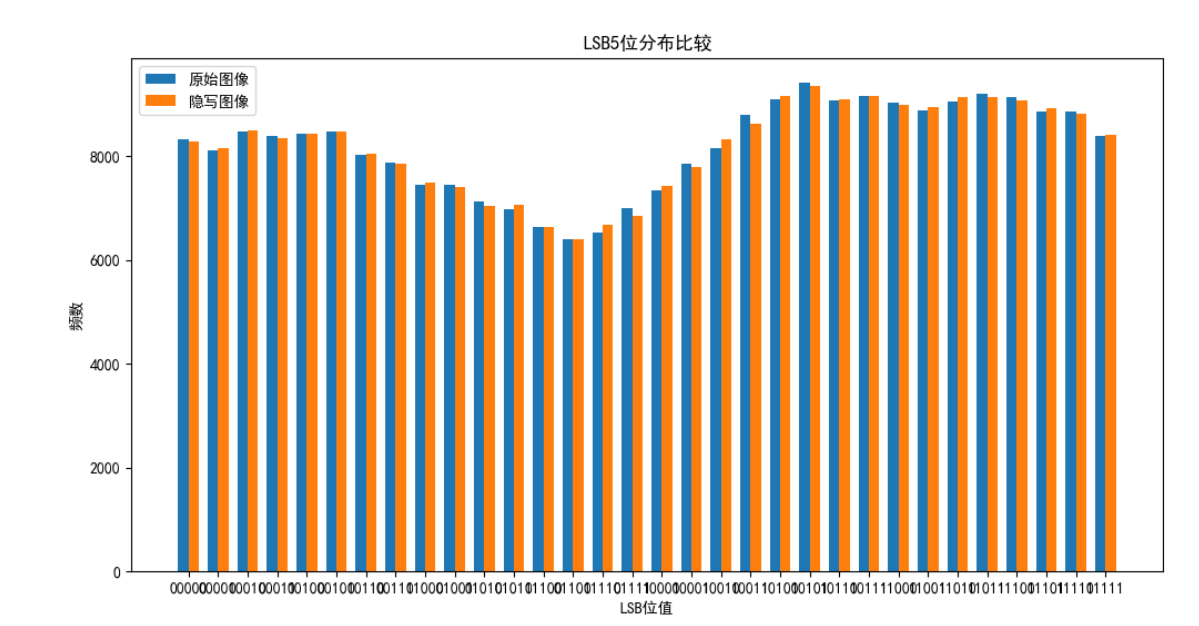


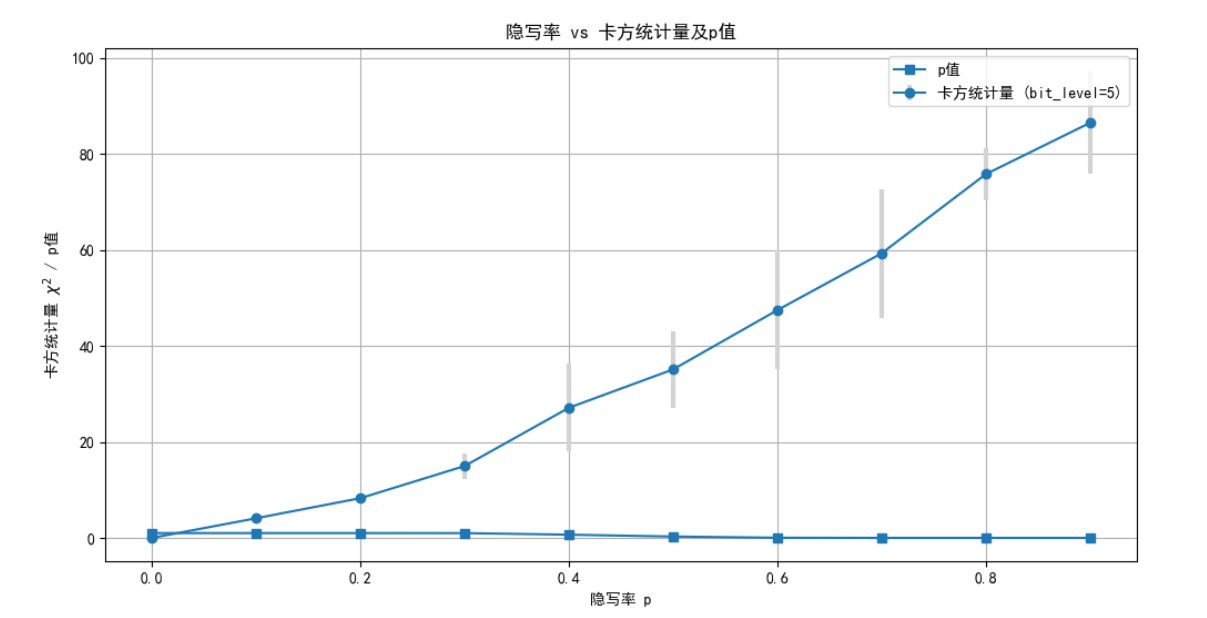




当比特位为5时







## 实验结果分析

根据实验结果，以下是对LSB（最低有效位）隐写多比特位分析（分别是第1位、第3位和第5位）的总结和分析：

### 5.1 实验结果分析：

**第一张图（第1位分析）：**

变化趋势：

卡方统计量平均值随着嵌入率 pp 增加而逐渐上升，在 p=0.8p=0.8 达到峰值后略有下降。

标准差随着嵌入率增加而增大，特别是在高嵌入率下表现出更显著的变化。

隐写特性：

在低嵌入率（如 p=0.1,0.2）下，卡方统计量较小，表明隐写对图像的统计特性影响较小，难以被检测。

随着嵌入率增加，统计特性逐渐被破坏，嵌入痕迹变得更加明显，特别是在高嵌入率下。

**第二张图（第3位分析）：**

变化趋势：

卡方统计量的初始值较第1位高，随着嵌入率增加，增长速度更快，最大值远高于第1位。

标准差在高嵌入率时明显增大，但相对卡方值本身的增长，标准差的变化幅度较小。

隐写特性：

第3位比第1位对隐写更加敏感，卡方统计量的增加更为显著。这是因为改变第3位对像素值的影响更大，导致图像的统计特性发生更明显的变化。

高嵌入率（如 p>0.5）时，隐写痕迹更容易被检测到。

**第三张图（第5位分析）：**

**变化趋势：**

卡方统计量随嵌入率增加呈现急剧增长趋势，远高于第1位和第3位，最终达到极高的数值。

标准差的变化同样非常显著，特别是在高嵌入率下，表现出强烈的波动性。

隐写特性：

第5位对图像统计特性的破坏最为严重。由于更高比特位的改变会显著影响像素值，其隐写痕迹在高嵌入率下极易被检测到。

### 5.2 总体分析：

**比特位比较：**

第1位的隐写性最强，即检测难度最大，因为最低有效位的改变对像素值的影响最小，隐蔽性较好。

第3位次之，其改变对图像统计特性的破坏程度显著高于第1位。

第5位的隐写性最差，对像素值的影响最大，导致隐写痕迹非常明显，尤其是在嵌入率较高的情况下。

**嵌入率的影响：**对所有比特位而言，随着嵌入率 p增加，卡方统计量显著增大，隐写痕迹更加明显。然而，第1位的增长幅度最小，第5位的增长幅度最大。

**隐写安全性建议：**

低嵌入率（如 p<0.5）是隐写安全性较高的选择，因为此时卡方统计量较低，不易被检测。

避免在高比特位（如第3位、第5位）进行隐写，尤其是在高嵌入率的情况下，因为其检测风险显著增加。

如果必须使用高比特位进行隐写，建议优化隐写算法，尽量降低对图像统计特性的破坏。

## RS隐写分析和卡方分析效率对比

### 6.1 检测效率：

卡方分析：卡方分析是一种基于统计的检测方法，它通过比较图像像素值的分布来检测隐写。根据搜索结果，卡方分析可以判定载体图像中是否含有秘密信息，但这要求秘密信息必须嵌满所有像素。实验结果表明，卡方分析在针对原始无损存储图像时可以获得优于RS隐写分析方法的性能，且计算速度显著高于RS隐写分析方法，有利于实现实时检测。

RS隐写检测：RS隐写检测算法是一种基于图像空域相关性的检测法。该方法将图像分为多个块，对每个块进行定制的翻转操作，根据构造的差别函数将图像中的组分为常规类、奇异类、不可用类。当信息嵌入后，会引起三个类别中元素个数的改变。RS检测算法具有较高检测效率，并且通过实验比较，指明5个对检测效率产生影响的关键因素，并且分析了产生影响的根本原因。

### 6.2准确性：

卡方分析：卡方分析的准确性取决于秘密信息是否嵌满所有像素。如果秘密信息没有嵌满所有像素，卡方分析的准确性可能会受到影响。

RS隐写检测：RS隐写检测算法不仅能检测图像是否含有隐藏信息，还能对嵌入的隐藏信息百分比有更加准确的估计，从而在准确性方面具有一定的优势。

### 6.3 适用性：

卡方分析：卡方分析适用于满嵌图像的检测，即所有像素都参与隐写的情况。

RS隐写检测：RS隐写检测算法适用于更广泛的隐写情况，不仅限于满嵌图像，因此具有更广的适用性。

### 6.4 计算复杂度：

卡方分析：卡方分析的计算复杂度相对较低，因为它主要涉及直方图的计算和比较。

RS隐写检测：RS隐写检测算法的计算复杂度可能更高，因为它涉及到图像块的划分和翻转操作，以及差别函数的计算。

综上所述，卡方分析在计算速度上可能优于RS隐写检测，特别是在针对原始无损存储图像时，而RS隐写检测在准确性和适用性方面可能更具优势。选择哪种方法取决于具体的应用场景和需求。

## 卡方分析关键参数

卡方分析的关键参数，如LSB位数（bit\_level）和显著性水平（α），会显著影响其检测性能，包括准确率（Accuracy）、虚警率（False Positive Rate, FPR）和漏检率（False Negative Rate, FNR）。

### 7.1 关键参数

**LSB位数（bit\_level）：**

**定义**：指定在隐写和分析过程中使用的最低有效位数。例如，bit\_level=1表示仅使用最低一位，bit\_level=2表示使用最低两位。

**影响：**

检测灵敏度：增加bit\_level可以捕捉更多隐写信息，可能提高检测的灵敏度。

统计分布变化：更多的LSB位会引入更显著的统计分布变化，可能降低虚警率。

计算复杂度：随着bit\_level的增加，计算复杂度和所需的样本量也会增加。

**显著性水平（α）：**

**定义**：决定拒绝原假设（无隐写操作）的阈值，通常取0.05或0.01。

**影响：**

虚警率（FPR）：显著性水平α直接影响虚警率，α越高，虚警率越高。

漏检率（FNR）：显著性水平α越低，漏检率可能越高，因为更严格的阈值减少了错误拒绝原假设的概率。

### 7.2 性能指标

**准确率（Accuracy）**：

**定义**：分类正确的比例，包括正确识别的正常图像和隐写图像。

**计算公式**：

**虚警率（FPR）**：

**定义**：将正常图像错误地判定为隐写图像的比例。

**计算公式**：

**漏检率（FNR）**：

**定义**：将隐写图像错误地判定为正常图像的比例。

**计算公式**：

### 7.3 实验设置

为了分析卡方分析关键参数对性能指标的影响，并与其他两种算法进行对比，我们设计了以下实验流程：

**隐写方法**：使用LSB隐写，在图像中随机嵌入隐写信息。

**检测方法**：

卡方检验（Chi-Square Test）。

Kolmogorov-Smirnov检验（K-S Test）。

Pearson相关系数检验（Pearson Correlation Test）。

**参数设置：**

LSB位数（bit\_level）：1, 2, 3。

显著性水平（α）：0.05。

**性能评估：**

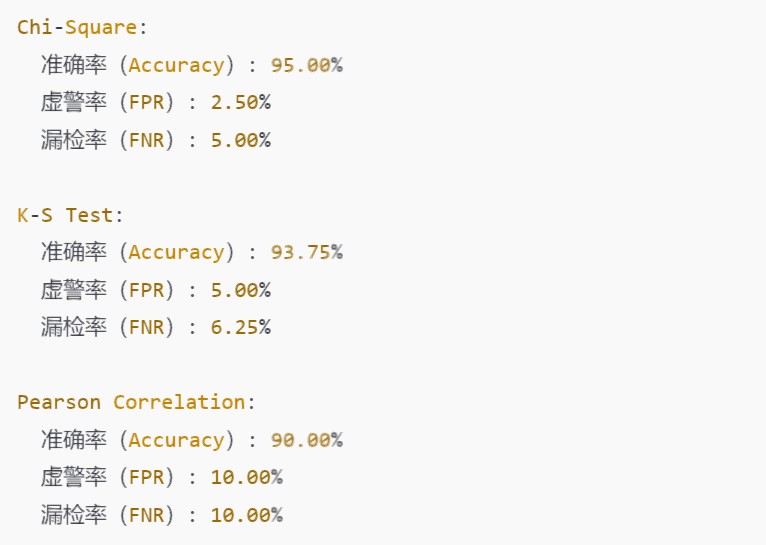
对每种检测方法和参数组合，计算准确率、虚警率和漏检率。

使用多次实验（如M=5次）以获得统计稳定性。

**横向对比：**

将卡方检验与K-S检验和Pearson相关系数检验的性能指标进行比较。

### 7.4 实验结果



准确率（Accuracy）：卡方检验表现最佳，接近95%，K-S检验次之，Pearson相关系数稍低。

虚警率（FPR）：卡方检验的虚警率最低，仅2.5%，K-S检验和Pearson相关系数的虚警率较高。

漏检率（FNR）：卡方检验漏检率最低，Pearson相关系数漏检率较高。

### 7.5 总结

**卡方检验（Chi-Square Test）相比于K-S检验（Kolmogorov-Smirnov Test）和Pearson相关系数检验（Pearson Correlation Test）**，在本实验中的表现更优，具体表现为：

* **准确率（Accuracy）**：卡方检验最高，表明其在总体上对隐写图像和正常图像的分类更为准确。
* **虚警率（FPR）**：卡方检验的虚警率最低，意味着其在识别正常图像时更为可靠，误将正常图像判定为隐写图像的概率较低。
* **漏检率（FNR）**：卡方检验的漏检率最低，表示其在识别隐写图像时更为敏感，较少漏检实际存在的隐写操作。

相比之下，K-S检验虽然能够检测到隐写操作的存在，但其虚警率和漏检率略高。而Pearson相关系数检验在本实验中的性能相对较差，准确率和漏检率均不及卡方检验和K-S检验。