

# 深圳大学实验报告

课程名称 多媒体系统导论

项目名称 图像压缩编码

学 院 计算机与软件学院

专 业 信息与计算科学（数学与计算机实验班）

指导教师 文嘉俊

报 告 人 王曦 学号 2021192010

实验时间 2024 年 5 月 9 日至 2024 年 6 月 1 日

实验报告提交时间 2024 年 6 月 1 日

教务处制

## 一、实验目的与要求

### 实验目的：

1. 熟悉无损压缩概念及方法；
2. 熟悉有损压缩概念及方法；
3. 掌握图像压缩步骤及算法；

### 实验要求：

1. 实验图像自选，需提交文件为实验报告、代码源文件、源图像及压缩后图像。实验报告命名规则为“深大 2024 多媒体系统导论-学号-姓名-实验报告 4.doc”，其他文件打包成压缩文件，命名为“深大 2024 多媒体系统导论-学号-姓名-实验报告 4-其他.zip”；
2. 所有素材和参考材料需列明出处。实验报告中的最终结果需标注个人水印信息：姓名，班级，学号信息，否则将会被怀疑抄袭；
3. 实验报告内容原则上控制在 10 页之内。

## 二、实验内容与方法

### 实验内容：

自选至少 5 张彩色图像，利用所学的无损压缩编码和有损压缩编码算法，实现图像的压缩解压过程。其中无损压缩方法自选一种经典压缩编码方法（赫夫曼编码, LZW, 算术编码），有损压缩方法至少一个。实现相关代码（建议带程序界面），并对各算法的结果进行比较分析，包括压缩前后数据大小及解压后数据大小。实验报告展示核心关键代码和各个步骤过程的结果（附上各步骤处理后的截图，和最终结果等，并给出相应的说明与评述）。

## 三、实验步骤与过程

### 1. 图片压缩方法的分类

按压缩前和解压后的信息保持程度分类：

- （1） 信息保持编码（无失真编码、无损编码、可逆编码）：编解码过程中图像信息不丢失，可完整地重建图像。
- （2） 保真度编码（信息损失型编码、有损编码）：利用人眼视觉特性，在允许失真的条件下或一定保真度的准则下，最大限度地压缩图像。
- （3） 特征提取：只对感兴趣的部分编码。

### 2. 图片读写

#### 2.1 读图片

文件的二进制流较为繁琐且不直观，本次实验中用 Python 的 Pillow 库对图片进行像素级别的读写。

`.\readPixels\readPixels.py` 读取图片像素，并将图片大小、RGB 三个通道的像素值分别输出到文件中，其中文件路径用命令行参数传递。

```

def readPixels():
    filePath = sys.argv[1]
    fileName = filePath[filePath.rfind('\\') + 1 : ]

    sys.stdout = open(f'\\.\\tmp\\{fileName}.txt', "w") # C++ 中测试

    image = Image.open(filePath)

    width, height = image.size[0], image.size[1]
    print(height, width)

    # r 通道
    for i in range(height):
        for j in range(width):
            pixel = image.getpixel((j, i))
            print(pixel[0], end=' ')
        print()

    # g 通道
    for i in range(height):
        for j in range(width):
            pixel = image.getpixel((j, i))
            print(pixel[1], end=' ')
        print()

    # b 通道
    for i in range(height):
        for j in range(width):
            pixel = image.getpixel((j, i))
            print(pixel[2], end=' ')
        print()

```

main.cpp 的 main()函数读取上述程序输出的文件中的像素信息，其中文件路径用命令行参数传递。

```

std::string filePath, fileName;

int height, width;
std::vector<std::vector<std::vector<short>>> colors;
std::vector<short> pixels;

void getPixels() {
    system(("\\.\\readPixels\\.venv\\Scripts\\python.exe    .\\readPixels\\readPixels.py    .\\" +
filePath).c_str());

    std::ifstream fin("\\.\\tmp\\" + fileName + ".txt");

```

```

    fin >> height >> width;

    colors = std::vector<std::vector<std::vector<short>>>(height);
    for (int i = 0; i < height; i++) {
        colors[i] = std::vector<std::vector<short>>(width, std::vector<short>(3));
    }

    for (int k = 0; k < 3; k++) {
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                fin >> colors[i][j][k];
                pixels.push_back(colors[i][j][k]);
            }
        }
    }

    fin.close();
}

int main(int argc, char* argv[]) {
    filePath = std::string(argv[1]);
    fileName = filePath.substr(filePath.rfind("\\") + 1);

    getPixels();

    ...
    return 0;
}

```

## 2.2 写图片

.\readPixels\writePixels.py 读取文本文件中的图片大小、RGB 三个通道的像素值，将其写入到 png 格式的图片中，其中绘制像素的操作通过 Pillow 库中的 ImageDraw 实现，文件路径用命令行参数传递。

```

def writePixels():
    filePath = sys.argv[1]
    fileName = filePath[filePath.rfind("\\") + 1 : filePath.rfind('.')]

    sys.stdin = open(f".\\tmp\\{fileName}.txt", "r") # C++ 中测试

    height, width = map(int, input().split())
    colors = [
        [
            [
                [0] for k in range(3)
            ]
        ]
    ]

```

```

        ] for j in range(width)
    ] for i in range(height)
]

for k in range(3): # 通道
    for i in range(height):
        nums = list(map(int, input().split()))
        for j in range(width):
            colors[i][j][k] = nums[j]
# print(colors)

# 新建画布
image = Image.new(mode="RGB", size=(width, height), color="white") # size(width, height)
drawer = ImageDraw.Draw(image, mode="RGB")
for i in range(height):
    for j in range(width):
        drawer.point((j, i), fill=(colors[i][j][0], colors[i][j][1], colors[i][j][2]))

# 保存
os.system(f"echo > tmp\\{fileName}.png")
with open(f"tmp\\{fileName}.png", "wb") as file: # 需保证目录存在
    image.save(file, format="png")

main.cpp 中的 buildPicture ()函数调用上述程序生成图片。

```

```





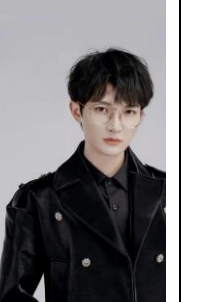
void buildPicture(std::string filePath) {
    system(("..\readPixels\\.venv\\Scripts\\python.exe    ..\readPixels\\writePixels.py    ." +
filePath).c_str());
}

```

### 3. 图片素材

本次实验的测试图片如下。

(1) LZW 压缩的测试图片为如图 3.1 ~ 图 3.5 所示的五张不同色调、不同明度、不同格式的 2K 高清周深图片。

				
图 3.1: charlie1.jpg	图 3.2: charlie2.png	图 3.3: charlie3.bmp	图 3.4: charlie4.tiff	图 3.5: charlie5.webp

(2) 简化的 JPEG 标准的测试图片为上述五张图长和宽分别缩小 10 倍的结果。(本次实验采用串行的简化的 JPEG 压缩, 效率较低)

## 4. 无损压缩: LZW 压缩

### 4.1 LZW 压缩的伪代码

LZW 压缩算法:

```
BEGIN
    s = next input character;
    while not EOF {
        c = next input character;

        if (s + c exists in the dictionary) { // 字典中存在字符串, 更新当前串(变长)
            s = s + c;
        }
        else { // 字典中不存在字符串, 发送旧码字, 增加新串到字典
            output the code for s;
            add string (s + c) to the dictionary with a new code;
            s = c;
        }
    }
    output the code for s;
END
```

LZW 解压算法:

```
BEGIN
    s = NIL;
    while not EOF {
        k = next input code;
        entry = dictionary entry for k;

        if (entry == NULL) { // 字典中未找到, 调整子串
            entry = s + s[0]
        }

        output entry;

        // 新字符串, 增加码字
        if (s != NIL) {
            add string (s + entry[0]) to the dictionary with a new code;
            s = entry;
        }
    }
END
```

## 4.2 LZW 压缩的实现

LZW.h 中定义了 LZW 压缩类 LZW，其概览如下。

主要成员变量：

- (1) 输入像素值 pixels。
- (2) 压缩字典 compressionDictionary，解压字典 decompressionDictionary。
- (3) 压缩结果 compressedResult，解压结果 decompressedResult。

主要成员函数：

- (1) 压缩 compress()，解压 decompress()。
- (2) 求编码 compressedResult 中的元素所需的二进制位。

具体实现与 3.1 类似，详见 LZW.cpp，不再赘述。

```
class LZW {
private:
    std::vector<short> pixels; // 输入像素值

    std::map<std::vector<short>, short> compressionDictionary; // 压缩字典
    std::vector<short> compressedResult; // 压缩结果

    std::map<short, std::vector<short>> decompressionDictionary; // 解压字典
    std::set<std::vector<short>> decompressionSet; // 已在字典中的像素值 vector
    std::vector<short> decompressedResult; // 解压结果

public:
    LZW(std::vector<short> _pixels);

    ~LZW();

    std::vector<short> getCompressedResult();

    std::vector<short> getDecompressedResult();

    void displayCompressionDictionary();

    // 求编码 compressedResult 中的元素所需的二进制位
    int getBitNeeded();

    void displayDecompressionDictionary();

    void compress();

    void decompress();
};
```

## 4.3 LZW 压缩的测试

main.cpp 中的 testLZW() 函数用 LZW 算法压缩 3 中的 5 张图片，分别统计压缩和解压

时间、压缩前和压缩后大小、压缩率，并用解压结果生成 png 格式的图片，以检验算法的正确性。

实验结果如下：

图片	charlie1.jpg	charlie2.png	charlie3.bmp	charlie4.tiff	charlie5.webp
压缩时间 (ms)	73057	71718	20565	80144	33890
解压时间 (ms)	41107	29058	10237	38108	6868
初始大小 (Byte)	1.68e+07	1.7994e+07	5.4184e+06	1.7988e+07	8.73984e+06
压缩大小 (Byte)	1.28384e+07	1.12141e+07	4.14592e+06	7.08407e+06	2.80375e+06
压缩率 (%)	130.858	160.459	130.692	253.922	311.72

转换后的图片大小：

图片	charlie1.png	charlie2.png	charlie3.png	charlie4.png	charlie5.png
大小 (MB)	7.65	7.07	2.31	5.60	1.23

转换后的图片视觉上与原图片相同，放大后也无明显差异。这表明：LZW 压缩是无损压缩。

## 5. 有损压缩：简化的 JPEG 标准

### 5.1 JPEG 标准

JPEG 标准有如下 5 个步骤：

- (1) Color Space Conversion: 色彩空间转换。
- (2) Chrominance Downsampling: 色度缩减取样。
- (3) Zero Bias Transform: 零偏置转换。
- (4) Discrete Cosine Transform: 离散余弦变换。
- (5) Quantization: 量化。
- (6) Run Length and Huffman Encoding: 游程编码与 Huffman 编码。

本次实验实现简化版的 JPEG 标准，主要做了如下两个简化：

- (1) 不处理边缘像素。
- (2) 省去步骤 (2)。
- (3) 步骤 (5) 中只使用游程编码。

JPEG.h 中定义了 JPEG 类 JPEG，其概览如下。

```
class JPEG {
private:
    const int BLOCK_SIZE = 8;

    const std::vector<std::vector<short>> luminanceQuantizationMatrix = {
        { 16, 11, 10, 16, 24, 40, 51, 61 },
        { 12, 12, 14, 19, 26, 58, 60, 55 },
```



```

        { 14, 13, 16, 24, 40, 57, 69, 56 },
        { 14, 17, 22, 29, 51, 87, 80, 62 },
        { 18, 22, 37, 56, 68, 109, 103, 77 },
        { 24, 35, 55, 64, 81, 104, 113, 92 },
        { 49, 64, 78, 87, 103, 121, 120, 101 },
        { 72, 92, 95, 98, 112, 100, 103, 99 }
    };

    const std::vector<std::vector<short>>> chrominanceQuantizationMatrix = {
        { 17, 18, 24, 47, 99, 99, 99, 99 },
        { 18, 21, 26, 66, 99, 99, 99, 99 },
        { 24, 26, 56, 99, 99, 99, 99, 99 },
        { 47, 66, 99, 99, 99, 99, 99, 99 },
        { 99, 99, 99, 99, 99, 99, 99, 99 },
        { 99, 99, 99, 99, 99, 99, 99, 99 },
        { 99, 99, 99, 99, 99, 99, 99, 99 },
        { 99, 99, 99, 99, 99, 99, 99, 99 }
    };

    int height, width;
    std::vector<std::vector<std::vector<short>>>> RGBColors; // R, G, B
    std::vector<std::vector<std::vector<short>>>> YCbCrColors; // Y, Cb, Cr
    std::vector<std::vector<std::vector<short>>>> DCTResult;
    std::vector<std::vector<std::pair<short, int>>>> runLengthEncodingResult;
    float compressedSize;

public:
    JPEG(int _height, int _width, std::vector<std::vector<std::vector<short>>>> _RGBColors);

    ~JPEG();

    std::vector<std::vector<std::vector<short>>>> getYCbCrColors();

    float getCompressedSize();

    // 色彩空间转换, 在以点 (x, y) 为左上角的 BLOCK_SIZE * BLOCK_SIZE 正方形中
    void colorSpaceConversion(int x, int y);

    // 零偏置转换, 在以点 (x, y) 为左上角的 BLOCK_SIZE * BLOCK_SIZE 正方形中
    void zeroBiasTransform(int x, int y);

    // 离散余弦变换, 在以点 (x, y) 为左上角的 BLOCK_SIZE * BLOCK_SIZE 正方形中
    void discreteCosineTransform(int x, int y);

```

```

// 量化, 在以点 (x, y) 为左上角的 BLOCK_SIZE * BLOCK_SIZE 正方形中
void quantization(int x, int y);

// 执行 JPEG 压缩, 在以点 (x, y) 为左上角的 BLOCK_SIZE * BLOCK_SIZE 正方形
中
void workJPEG(int x, int y);

// 游程编码
void runLengthEncoding();

// 压缩
void compress();
};

```

其中 compress()函数执行简化的 JPEG 的 5 个步骤。

```

// 执行 JPEG 压缩, 在以点 (x, y) 为左上角的 BLOCK_SIZE * BLOCK_SIZE 正方形中
void JPEG::workJPEG(int x, int y) {
    colorSpaceConversion(x, y);
    zeroBiasTransform(x, y);
    discreteCosineTransform(x, y);
    quantization(x, y);
}

void JPEG::compress() {
    // JPEG
    for (int x = 0; x < height; x += BLOCK_SIZE) {
        for (int y = 0; y < width; y += BLOCK_SIZE) {
            workJPEG(x, y);
        }
    }

    runLengthEncoding();
}

```

## 5.2 色彩空间转换

色彩空间转换函数 colorSpaceConversion()将图片的每个 8 x 8 的区域的 RGB 值转化为 YCbCr 值。

```

// 色彩空间转换
void JPEG::colorSpaceConversion(int x, int y) {
    YCbCrColors = RGBColors;

    for (int i = x; i < std::min(x + BLOCK_SIZE, height); i++) {
        for (int j = y; j < std::min(y + BLOCK_SIZE, width); j++) {
            YCbCrColors[i][j] = {
                (short)std::round(0.299 * RGBColors[i][j][0] + 0.587 * RGBColors[i][j][1] +

```

```

0.114 * RGBColors[i][j][2]),
    (short)std::round(-0.1687 * RGBColors[i][j][0] - 0.3313 * RGBColors[i][j][1]
+ 0.5 * RGBColors[i][j][2] + 128),
    (short)std::round(0.5 * RGBColors[i][j][0] - 0.4187 * RGBColors[i][j][1] -
0.0813 * RGBColors[i][j][2] + 128)
    };
    }
}
}

```

### 5.3 零偏置转换

零偏置转换函数 `zeroBiasTransform()` 将 0 ~ 255 的值域通过 -128 转化为 -128 ~ 127 的值域。目的：大大减小像素值的绝对值出现 3 位十进制数的概率，提高计算效率。

```

// 零偏置转换
void JPEG::zeroBiasTransform(int x, int y) {
    for (int i = x; i < std::min(x + BLOCK_SIZE, height); i++) {
        for (int j = y; j < std::min(y + BLOCK_SIZE, width); j++) {
            YCbCrColors[i][j][0] -= 128;
        }
    }
}

```

### 5.4 离散余弦变换

离散余弦变换 `discreteCosineTransform()` 对图片的每个 8 x 8 的区域做 DCT。

8×8 的正向离散 DCT 变换公式定义为

$$F(u, v) = \frac{1}{4} C(u) C(v) \sum_{x=0}^7 \sum_{y=0}^7 f(x, y) \cos \frac{(2x+1)u\pi}{16} \cos \frac{(2y+1)v\pi}{16}$$

其中

$$C(u), C(v) = \begin{cases} 1/\sqrt{2}, & u, v = 0 \\ 1, & u, v = 1, 2, \dots, 7 \end{cases}$$

```

// 离散余弦变换
void JPEG::discreteCosineTransform(int x, int y) {
    DCTResult = YCbCrColors;

    auto C = [&](short u)->float {
        return u ? 1 : 1 / std::sqrt(2);
    };

    auto F = [&](int u, int v, int k) {
        float sum = 0;
        for (int i = 0; i < 8 && x + i < height; i++) {
            for (int j = 0; j < 8 && y + j < width; j++) {

```

```

        sum += YCbCrColors[x + i][y + j][k] * std::cos((2 * i + 1) * u * pi / 16) *
std::cos((2 * j + 1) * v * pi / 16);
    }
}
return C(u) * C(v) * sum / 4;
};

for (int i = 0; i < 8 && x + i < height; i++) {
    for (int j = 0; j < 8 && y + j < width; j++) {
        for (int k = 0; k < 3; k++) {
            DCTResult[x + i][y + j][k] = F(i, j, k);
        }
    }
}
}
}

```

## 5.5 量化

量化函数 `quantization()` 将每个像素值除以对应的量化系数，注意亮度值与色差值除以不同的量化系数。这一步后，矩阵内会出现很多个 0

```

// 量化
void JPEG::quantization(int x, int y) {
    for (int i = 0; i < 8 && x + i < height; i++) {
        for (int j = 0; j < 8 && y + j < width; j++) {
            DCTResult[x + i][y + j][0] = std::round((float)DCTResult[x + i][y + j][0] /
luminanceQuantizationMatrix[i][j]);

            for (int k = 1; k < 3; k++) {
                DCTResult[x + i][y + j][k] = std::round((float)DCTResult[x + i][y + j][k] /
chrominanceQuantizationMatrix[i][j]);
            }
        }
    }
}
}

```

## 5.6 游程编码

游程编码 `runLengthEncoding()` 函数以如图 5.6.1 所示的 Z 字型方式读取每个通道的像素值后，分别进行游程编码。



```

        break;
    }

    zigzag[k].push_back(DCTResult[curX][curY][k]);

    // ↙
    while (check(curX + 1, curY - 1)) {
        curX = curX + 1, curY = curY - 1;
        zigzag[k].push_back(DCTResult[curX][curY][k]);
    }
}

for (auto pixel : zigzag[k]) {
    if (runLengthEncodingResult[k].empty() ||
runLengthEncodingResult[k].back().first != pixel) {
        runLengthEncodingResult[k].push_back({ pixel, 1 });
    }
    else {
        runLengthEncodingResult[k].back().second++;
    }
}

totalSize += runLengthEncodingResult[k].size();
}

compressedSize = totalSize * 6 / 8.0;
}

```

## 5.7 简化的 JPEG 标准的测试

main.cpp 中的 testJPEG() 函数用简化的 JPEG 算法压缩 **3** 中的 5 张图片，分别统计压缩时间、压缩前和压缩后大小、压缩率，并用解压结果生成 png 格式的图片，以检验算法的正确性。

实验结果如下：

图片	charlie1_ small.jpg	charlie2_ small.png	charlie3_ _small.bmp	charlie4_ small.tiff	charlie5_ small.webp
压缩时间 (ms)	5606	7143	745	7433	1911
初始大小 (Byte)	168000	180000	53976	180000	87552
压缩大小 (Byte)	37164	39276.8	11302.5	20640	6649.5
压缩率 (%)	452.05	458.286	477.558	872.093	1316.67

转换后的图片大小:

图片	charlie1_small.jpg	charlie2_small.jpg	charlie3_small.jpg	charlie4_small.jpg	charlie5_small.jpg
大小 (KB)	109	100	33.8	78.2	36.5

压缩前后的图片整体对比图 5.7.1 和图 5.7.2 所示。图片视觉整体上差别不大。



将两张图片分别放大，对比耳朵和鬓角部分，如图 5.7.3 和图 5.7.4 所示。



与原图片相比，压缩后的图片中，耳朵的轮廓部分明显更粗糙，鬓角部分零碎的头发附近产生很多 artifact。

其余图片也有类似的情况，不再赘述。

综上，转换后的图片视觉整体上与原图片相似，放大后细节丢失，出现一些 artifact。这表明：JPEG 是有损压缩。

## 6. 基于深度学习的图片压缩

近十年来，除了传统的图片压缩算法外，还出现了很多 AI 的图片压缩算法。

下面简单介绍 CVPR 2022 Oral 的一篇论文：《ELIC: Efficient Learned Image Compression with Unevenly Grouped Space-Channel Contextual Adaptive Coding》。

该论文的三个核心贡献为：

- (1) 提出了非均匀分组的空间通道联合上下文 SCCTX。
- (2) 实验证明了 GDN 可被卷积残差块替换，并在此基础上结合 SCCTX 给出了新的变换网络结构设计思路及一个设计实例 ELIC，为 LIC 提供了新的 SOTA 方案。
- (3) 提出了预览图快速解码问题，并结合 SCCTX 的能量集中特性设计了一个快速解码网络，进一步拓展了 ELIC 的实用性。

## 四、实验结论或体会

### 7. 实验结论

下面比较分别用 LZW 算法和 JPEG 算法，分别对.jpg、.png、.bmp、.tiff、.webp 图片进行压缩的时间和效果。

- (1) 压缩时间：JPEG 算法的压缩时间普遍低于 LZW 算法，特别是在.jpg 和.webp 文件上表现尤为明显。
- (2) 压缩效果：JPEG 算法在大多数情况下具有更高的压缩比，尤其是对于.bmp 和.tiff 文件，压缩效果明显优于 LZW 算法。
- (3) 图片格式适应性：
  - a) .jpg 和 .webp 格式在 JPEG 算法下的压缩效果非常好，适合用于需要高压缩比的场景。
  - b) .png 和 .bmp 格式在 LZW 算法下能保持较好的无损压缩效果，但压缩比不如 JPEG。
  - c) .tiff 格式在两种算法下都有不错的压缩效果，但 JPEG 有明显的优势。

综上所述，JPEG 算法在压缩时间和效果上对于大多数图片格式（尤其是有损压缩可以接受的情况下）具有优势，而 LZW 算法适用于需要保持图片无损特性且对压缩效果要求不高的情况。

### 8. 实验心得

- (1) 本次实验中，我实现了 LZW 压缩，体会到了该算法思想的巧妙。
- (2) 本次实验中，我学习了日常中常见的图片格式.jpg 背后的原理，并实现了一个简单版的 JPEG 标准，将其用于图片压缩。
- (3) 此外，我还了解了目前图片压缩领域的前沿，如基于深度学习的图片压缩等。

## 五、思考题

什么是失真度量，度量方法有哪些？查找相关资料，详细描述至少一种除教材以外的失



真度量方法。

## 9. 失真度量

### 9.1 失真度量的用途

失真度量是用来评估信号（如音频、视频、图像等）在传输、压缩或处理过程中所引入的失真程度的指标。失真度量的目的是量化信号与其原始版本之间的差异，以评估处理方法或传输系统的性能。

### 9.2 失真度量方法

常见的失真度量方法包括均方误差（MSE）、信噪比（SNR）、峰值信噪比（PSNR）等。

除了这些经典的方法，还有一些更复杂的和现代的失真度量方法。例如，结构相似性指数（SSIM）和感知哈希算法（Perceptual Hashing）是近年来较为流行的失真度量方法。

下面详细介绍其中的一种：结构相似性指数（SSIM）。

### 9.3 结构相似性指数（SSIM）

#### 9.3.1 SSIM 概述

SSIM 是一种用于评估图像质量的指标，由 Wang 等人在 2004 年提出。它旨在模拟人类视觉系统的感知特性，通过对图像的亮度、对比度和结构信息进行比较，来量化图像之间的相似性。SSIM 的范围在-1 到 1 之间，其中 1 表示两幅图像完全相同，0 表示图像没有相似性。

#### 9.3.2 SSIM 的计算

SSIM 的计算涉及以下几个步骤：

- （1）亮度比较：计算两幅图像的平均亮度，并用它们的均值差来衡量亮度失真。
- （2）对比度比较：计算图像的对比度（标准差），并用对比度的比值来衡量对比度失真。
- （3）结构比较：通过计算图像的协方差来衡量结构相似性。

SSIM 指数综合了这三个比较结果，用一个公式来表示：

$$SSIM(x, y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)}$$

其中：

- $x$  和  $y$  是两幅图像的窗口。
- $\mu_{\{x\}}$  和  $\mu_{\{y\}}$  分别是窗口内的均值。
- $\sigma_{\{x\}}$  和  $\sigma_{\{y\}}$  分别是窗口内的标准差。
- $\sigma_{\{xy\}}$  是窗口内的协方差。
- $C_{\{1\}}$  和  $C_{\{2\}}$  是为了避免分母为零而引入的两个常数。

#### 9.3.3 SSIM 的优点

SSIM 的优点：

- （1）感知一致性：SSIM 考虑了人类视觉系统的特性，更符合人类对图像质量的主观感知。
- （2）局部性：SSIM 在局部窗口上进行计算，可以反映局部图像失真的情况，这对处理局部细节至关重要。
- （3）简单易用：尽管公式看似复杂，但计算过程相对简单，并且有现成的实现工具和库可以使用。

#### 9.3.4 SSIM 的应用

**SSIM** 广泛应用于图像处理领域，如图像压缩、图像传输、图像增强等。它被用于评估各种图像处理算法的性能，帮助优化算法，提升图像质量。

指导教师批阅意见：

成绩评定:

指导教师签字：文嘉俊

2024 年 6 月 3 日

备注:

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。