Государственное образовательное учреждение высшего профессионального образования
Санкт-Петербургский национальный исследовательский университет
Информационных Технологий, Механики и Оптики
Факультет инфокоммуникационных технологий

ОТЧЕТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ № 3

По Мультимедиа Технонолиям
На тему: «JPEG сжатие»

Проверил(а): Хлопотов М.В.

_____

Дата: «___»_____201__г.

Оценка: _____

Работу выполнил(а):

Никончук А.П.
Студент(ка) группы К3242
Дневного отделения

_____

2019

Цель: Изучить алгоритмы сжатия на примере JPEG с точки зрения его реализации.
Задачи:

1. реализовать алгоритм сжатия изображения, взяв за основу JPEG
2. Оценить качество работы алгоритма по PSNR и энтропии

Выполнение работы:

Этап 1:
В изображениях RGB имеется существенная корреляция между цветными компонентами и с точки зрения сжатия этот формая является заведомо избыточным. Так как основная компонента в YUV сосредоточена в первом канале яркости, а в RGB яркость как суть картинки рассеяна по всем каналам. После перевода представления прореживаются менее значимые каналы U и V, что связано со склонностью чеовеческого глаза к восприятию яркости, а не цвета. Этот этап позволяет повысить эффективность сжатия.

Y = 0.299*R + 0.578*G + 0.114*B
Cb = 0.1678*R - 0.3313*G + 0.5*B
Cr = 0.5*R - 0.4187*G + 0.0813*B
Y нужно сохранить без изменений.

In [3]:
```python
# ЭТАП 1
# Преобразование цветового пространства

def yuvToRGB(y, u, v):
    rd = y[:,:] + 1.402*(v[:,:] - 128)
    gr = y[:,:] - 0.34414*(u[:,:] - 128) - 0.71414*(v[:,:] - 128)
    bl = y[:,:] + 1.772*(u[:,:] - 128)
    return rd, gr, bl
def make_u(rgb):
    return (-0.1687*rgb[:,:,0] - 0.3313*rgb[:,:,1] + 0.5*rgb[:,:,2] + 128)
def make_v(rgb):
    return (0.5*rgb[:,:,0] - 0.4187*rgb[:,:,1] - 0.0813*rgb[:,:,2] + 128)
def make_y(rgb):
    return (0.299*rgb[:,:,0] + 0.587*rgb[:,:,1] + 0.114*rgb[:,:,2])
def rgbToYUV(rgb):
    u = -0.1687*rgb[:,:,0] - 0.3313*rgb[:,:,1] + 0.5*rgb[:,:,2] + 128
    v = 0.5*rgb[:,:,0] - 0.4187*rgb[:,:,1] - 0.0813*rgb[:,:,2] + 128
    y = 0.299*rgb[:,:,0] + 0.587*rgb[:,:,1] + 0.114*rgb[:,:,2]
    return y, u, v
```

Рисунок 1. Вариант кода преобразований цветового пространства.

Этап 2: Ресемплинг / дискретизация
Субдискретизация компоненты цвета – дискретизация цветовых каналов. Возможет засчет чувствительности человеческого глаза к изменениям света яркостного канала и нечувствительности к цвету. Возможен один из трёх способов дискретизации:

- 4:4:4 – отсутствует субдискретизация;
- 4:2:2 – компоненты цветности меняются через одну по горизонтали;
- 4:2:0 – компоненты цветности меняются через одну строку по горизонтали, при этом по вертикали они меняются через строку.

Каждое значение указывает на часть остающуюся в блоках по 4 (первое число - частота дискретизации яркостного канала, выраженная коэффициентом базовой частоты, шириной макропикселя). То в первом случае остаются все 4 пикселя. Во втором прореживается каждые

вторая сточка и второй столбец. В третьем остаётся одно цветовое значение на 4 пикселя. Во втором и третьем вариантах соответственно избавляемся от 1/3 или 1/2 информации. Следующия рис. демонстрирует применение разных форматов.

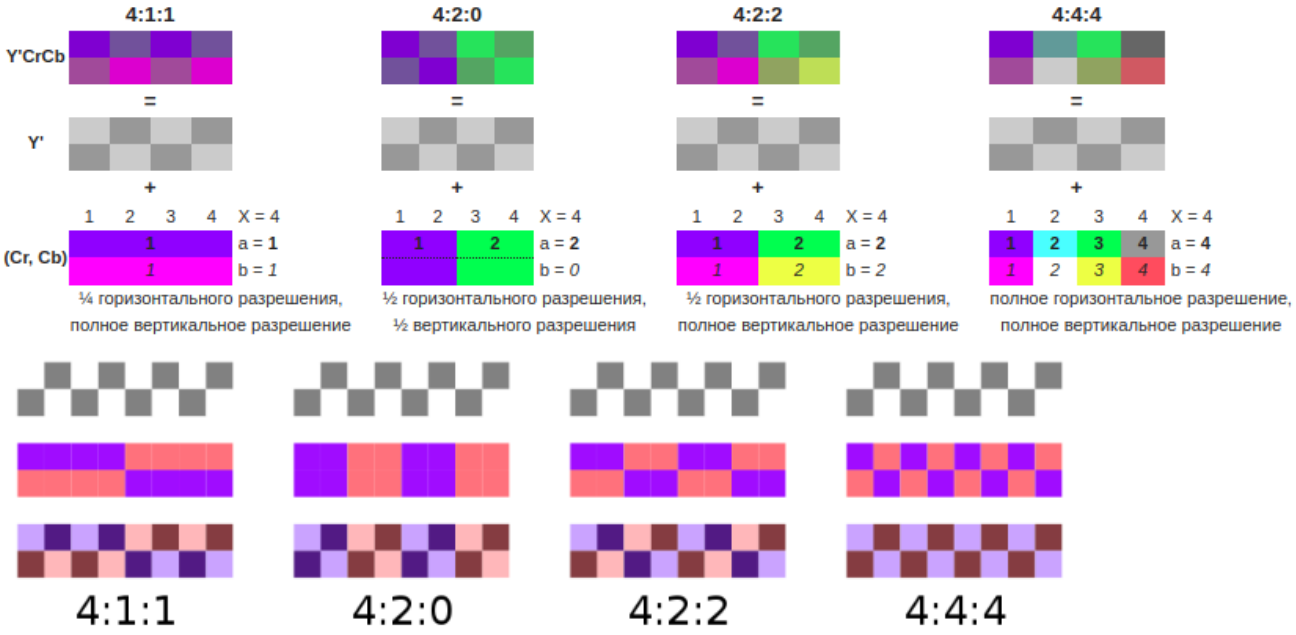


Рисунок 2. Форматы субдискретизации

In [4]:
```
# ЭТАП 2
# Субдискретизация компоненты цвета
# также называется ресамплингом

# квантование канала изображения на
# в адаптивном jpeg алгоритме предлагается выбор на данном этапе -
# какую часть оставлять 4:4:4, 4:2:2, 4:2:0/
def sampling(m):
    res = np.zeros((m.shape[0], m.shape[1]))
    res_plus1 = np.zeros((math.ceil(m.shape[0]/2)*2, math.ceil(m.shape[1]/2)*2))
    for i in range (m.shape[0]//2):
        for j in range (m.shape[1]//2):
            four = []
            four.append(m[i*2, j*2])
            if ((j*2)+1 < m.shape[1]):
                four.append(m[i*2, j*2+1])
            if ((i*2)+1 < m.shape[0]):
                four.append(m[i*2+1, j*2])
            if ((j*2)+1 < m.shape[1] and (i*2)+1 < m.shape[0]):
                four.append(m[i*2+1, j*2+1])
            summ = sum(four) / len(four)
            res_plus1[i*2:i*2+2, j*2:j*2+2] = summ
            """
            res[i*2, j*2] = summ
            if ((j*2)+1 < m.shape[1]):
                res[i*2, j*2+1] = summ
            if ((i*2)+1 < m.shape[0]):
                res[i*2+1, j*2] = summ
            if ((j*2)+1 < m.shape[1] and (i*2)+1 < m.shape[0]):
                res[i*2+1, j*2+1] = summ
            """
    res[::, ::] = res_plus1[0:res.shape[0], 0:res.shape[1]]
    return res
```

Рисунок 3. Вариант кода субдискретизации компонентов цвета

Этап 3: Разбиение на блоки (единицы данных)

Формируем рабочие блоки для дальнейших преобразований. Иногда этап сочетается с предыдущим, те изображени делится по компоненте y, а уветовые набираются через строчку и через столбец.

In [9]:
```
# ЭТАП 3
# деление на блоки с заданной размерностью
# вычисляется количество блоков общее x_block_count* y_block_count и
# затем обход картинки и сохранение блока в новом массиве - последовательное поблочное
# вытягивание. При неравном делении граничные блоки дополняются нулями - назначение
# переменной next_block. хитрость в создании дополнительных переменных - обрезка,
# например, значения 1504, если размер изображения был 1500 (блок 1696-1504 - берется
# значение 4x4, помещается в next_block, оставшаяся часть 0, тк  блок задается нулевым
#next_block = np.zeros)

def block_split(m, block_size):
    new_x_size, new_y_size  = \
        math.ceil(m.shape[0]/block_size)*block_size,\
        math.ceil(m.shape[1]/block_size)*block_size
    x_block_count, y_block_count = \
        int(new_x_size/block_size), int(new_y_size/block_size)
    m_blocks = []
    for i in range(x_block_count):
        for j in range(y_block_count):
            next_block = np.zeros((block_size, block_size))
            if i*block_size+block_size > len(m):
                x = len(m)
            else:
                x = i*block_size+block_size
            if j*block_size+block_size > len(m[0]):
                y = len(m[0])
            else:
                y = j*block_size+block_size
            next_block[0: x - i*block_size, 0: y - j*block_size] \
            = m[i*block_size: x, j*block_size: y]
            m_blocks.append(next_block)
    return m_blocks
```

Рисунок 4. Вариант кода разбиения на блоки.

Этап 4: Дискретное косинусное преобразование

(Discrete Cosine Transformation, DCT) позволяет понять насколько много деталей в каждом блоке. Оно применяется почти во всех видах компрессии, используемых в видеонаблюдении, за исключением вейвлет-сжатия. Во всех алгоритмах JPEG, MPEG и H.26x использована та или иная разновидность DCT. (Discrete Cosine Transformation, DCT). Оно применяется почти во всех видах компрессии, используемых в видеонаблюдении, за исключением вейвлет-сжатия. Во всех алгоритмах JPEG, MPEG и H.26x использована та или иная разновидность DCT. Можно сократить объем передаваемой информации, если допустить существование избыточности, определённым образом описав контуры объекта и указав средние значения яркости и цвета в пределах этого контура, то есть условно описать алгоритм (как у векторных изображений) а не множество точных пискелей. В цифровом видеосигнале может быть передан весь спектр пространственных частот, однако если провести частотный анализ изображения, то возможно оставить в сигнале лишь те частоты, что действительно в нём присутствуют. Катинка складывается из наложения всех уровней частот, домноженных на конкретный коэффициент, характерный уникальному сочетанию рисунка контуров.

	1	2	3	4	5	6	7	8
1	1603	203	11	45	-30	-14	-14	-7
2	108	-93	10	49	27	6	8	2
3	-42	-20	-6	16	17	9	3	2
4	56	69	7	-25	-10	-5	-2	-2
5	-33	-21	17	8	3	-4	-5	-3
6	-16	-14	8	2	-4	-2	1	1
7	0	-5	-6	-1	2	3	0	1
8	9	5	-6	-9	0	3	3	1

- НЧ компоненты;
- СЧ компоненты;
- ВЧ компоненты

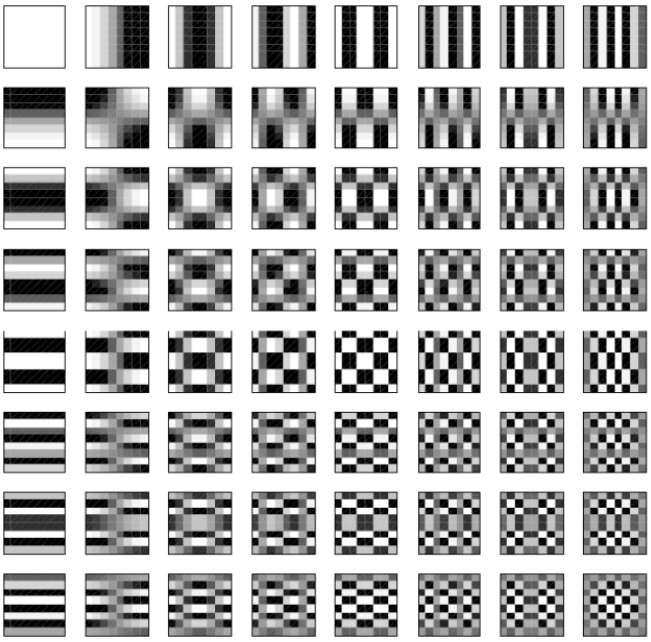


Рисунок 5. (левый) Отображение распределения низких средних и высоких частот дкп на матрице 8 на 8. (правый) Отображение частот пространственных волн.

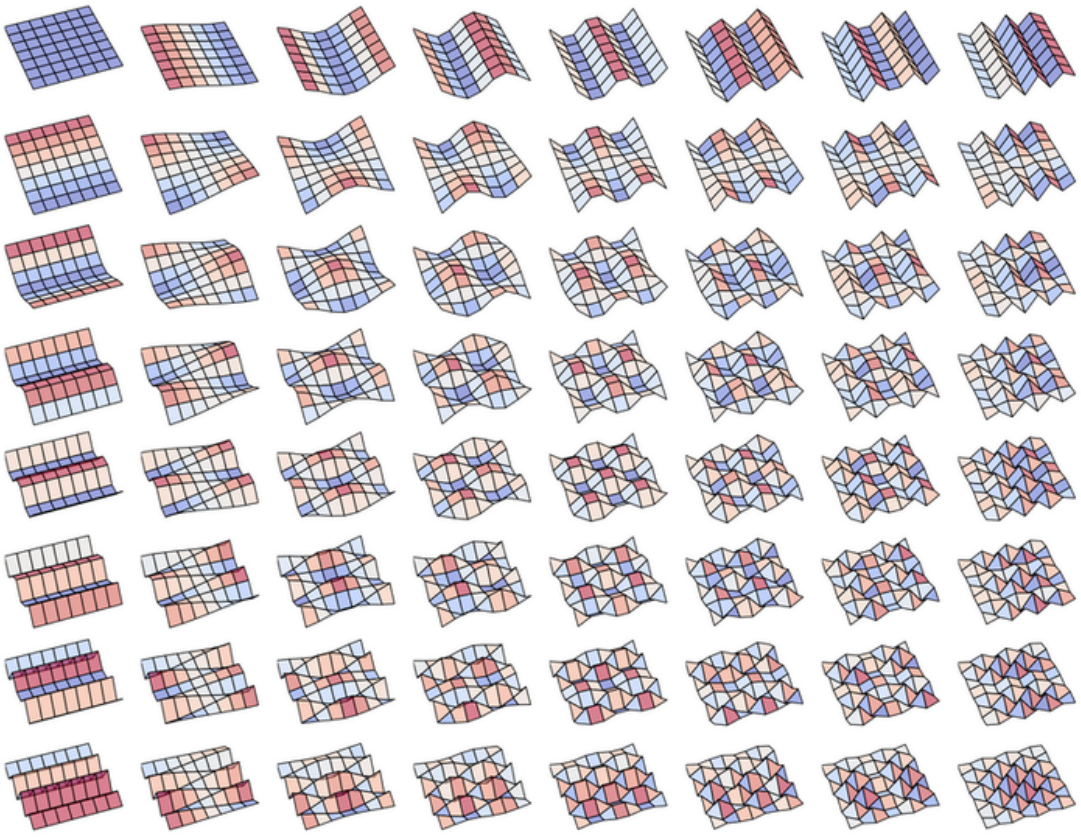


Рисунок 6. Трехмерные пространственные волны разной длинны.

Приведенное изображение показывает иллюстрирует работу преобразования.

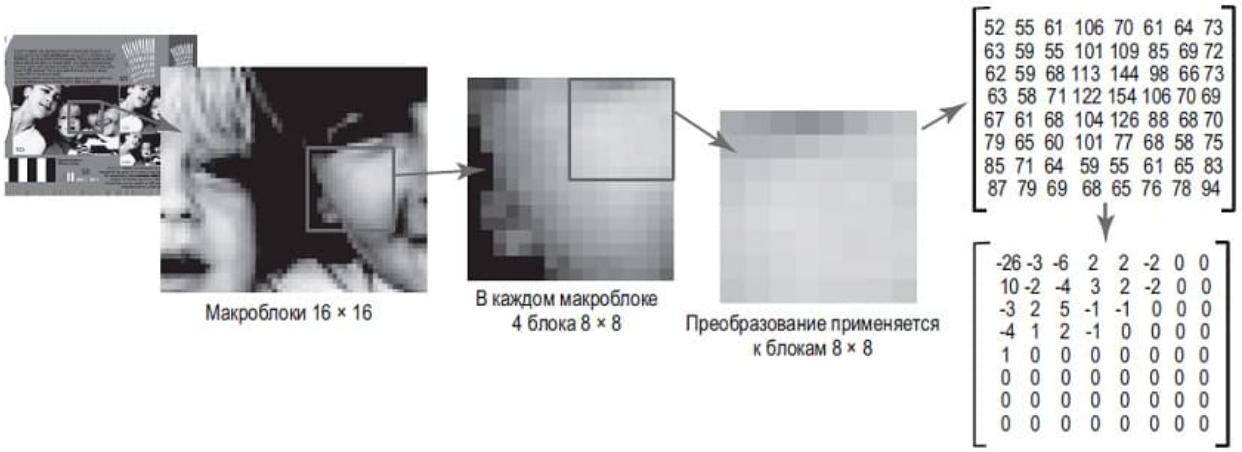


Рисунок 7. Пример выдиления блока и применения дкп

Изображение обрабатывается поблочно — как правило, квадратами 8 × 8 пикселов. В результате преобразования возникает матрица из 64 коэффициентов. Коэффициент — число, которое описывает содержание в блоке определённой пространственной частоты. На рисунке показаны матрицы значений уровня сигнала для каждого пиксела и коэффициентов пространственных спектральных составляющих DCT. Верхний слева коэффициент представляет собой усреднённую яркость блока — среднее арифметическое по всем пикселам — или постоянную составляющую. При перемещении вправо по пикселам коэффициенты отражают увеличение пространственной частоты по горизонтали, при перемещении вниз — по вертикали. Не отражающиеся частоты выражаются 0 коэффициентом.

При нулевом значении пересылка коэффициента не имеет смысла. При значении, близком к нулю, принятие коэффициента за ноль будет выражено в добавлении в сигнал той же самой пространственной частоты, но в противофазе. Решение об обнулении коэффициента зависит от того, насколько заметным должен быть этот нежелательный сигнал — а это регулируется заданием степени сжатия изображения. Если значение коэффициента слишком велико, чтобы им пренебречь, то компрессия всё равно может сделать своё дело, уменьшив количество данных для описания его значения. Визуально это выражается в добавлении к изображению небольшого «геометрического» шума. Артефактами компрессии, использующей дискретное косинусное преобразование, являются проявления пикселизации слишком «зажатых» изображений. Это происходит, как правило, на границах разбивки изображения на блоки 8 × 8 пикселов, а также на резких контрастных переходах.

В представленной работе используется быстрое преобразование Фурье dct пакета статистики быстрее всего работает в одном или двух измерениях. Теоретически существует 8 типов dct, в scipy преобразовании произвольной последовательности используется 3, один из которых обратный. По умолчанию используется второй. Указывается масштаирование, нормализация, которой по умолчанию нет, “ortho” - ортогональное (прямоугольное).

Этап 5: Квантование

Квантование – деление на ряд дискретных уровней и укладывание значений в этот диапазон квантов. Используется благодаря способности человека не замечать изменения в высокочастотных составляющих, поэтому коэффициенты, отвечающие за высокие частоты можно хранить с меньшей точностью. Для этого используется покомпонентное умножение (и

округление) матриц, полученных в результате ДКП, на матрицу квантования. На данном этапе тоже можно регулировать степень сжатия (чем ближе к нулю компоненты матрицы квантования, тем меньше будет диапазон итоговой матрицы).

In [13]:
```
# рекомендуемые QC - светимость и цветность
luminosity = [
    [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]
]
chromaticity = [
    [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]
]
```

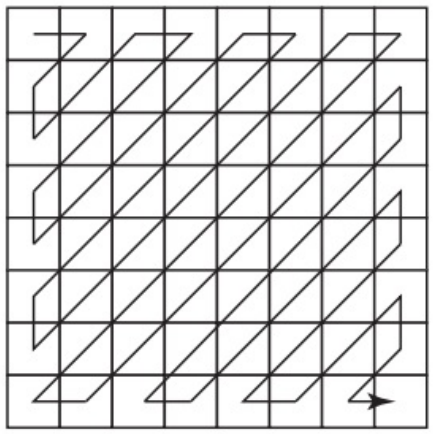
Рисунок 8. Рекоммендуемые коэффициенты квантования

In [72]:
```
# ЭТАП 5
# квантование - сокращение точности значений
# также часть дискретного косинусного преобразования засчет матрицы-делителя multiplier
# по теории также должен быть множитель coef, который уравнивает значения

def quantization(m_blocks, x_block_count, y_block_count, multiplier, coef):
    res = []
    for i in range(len(m_blocks)):
        res.append(np.round(m_blocks[i]/ multiplier*coef))
        # res.append((m_blocks[i]/ multiplier*coef).astype(np.int64))
    return res
```

Рисунок 9. Вариант кода квантования

Этап 6: Обход зигзагом



Зигзагообразное сканирование матрицы дискретного косинусного преобразования

Этап может быть упущен, однако для простоты дальнейшего кодирования рекомендуется вытягивать коэффициенты в линию. Последовательность охода зигзагом в заданном направлении, показанном на рисунке слева, обусловлено распределением частот по матрице.
При таком общохе по итогу, после применения также dct большая часть элементов в конце обрает линию из нулей, которую будет проще закодировать, используя RLE кодирование.

Рисунок 10. Последовательность обхода зигзагом

In [16]:
```python
# ЭТАП 6
# Обход зигзагом - вытягивание блоков для дальнейших преобразований

def zigzag(m):
    block_size = len(m)
    row, col = 0, 0
    flg = 1
    res = []
    res2 = []
    while col < block_size and row < block_size:
        if res == []:
            res.append([row, col])
        if res[-1]!=[row, col]:
            res.append([row, col])
        if row == 0 and flg == 1 and col!=block_size-1:
            col = col + 1
            flg = 0
        elif col == 0 and flg == 0 and row!=block_size-1:
            row = row + 1
            flg = 1
        elif row == block_size - 1 and flg == 0:
            col = col + 1
            flg = 1
        elif col == block_size - 1 and flg == 1:
            row = row + 1
            flg = 0

        if res[-1]!=[row, col]:
            res.append([row, col])
        if flg == 1 and col < block_size and row < block_size:
            row = row - 1
            col = col + 1
        elif flg == 0 and col < block_size and row < block_size:
            row = row + 1
            col = col - 1
    for i in res:
        res2.append(m[i[0]][i[1]])
    return res2
```

Рисунок 11. Вариант кода зигзагом

Этап 7: Шифрование

Часть1: Дельта-кодирование

Используется для сокращения чисел засчет предположения смежной схожести пикселов в блоке.

In [19]:
```python
# ЭТАП 7.1
# Дельта-кодирование - запись разницы пар чисел начиная с нуля. каждое число с 1 - разница
# с предыдущим.

def delta_code(a):
    ca = []
    ca = deepcopy(a)
    for i in range(len(ca)):
        for j in range(1, len(ca[i])):
            prev_el = a[i][j-1]
            ca[i][j] = ca[i][j] - prev_el
    return ca
```

Рисунок 12. Вариант кода дельта-кодирования

Часть2: RLE-кодирование

Применяется для сокращении ранее упомянутых идентичных последовательностей, например, большого количества нулей. Дельта-кодирование не влияет на RLE, теоритически их можно поменять местами.

In [21]:
```
# ЭТАП 7.2
# RLE-кодирование - сокращение количества идущих подряд чисел. Учитывая предшествующее
# дельта-кодирование к таким в изменённой последовательности относятся только нули


def rle_encode(a):
    rle = []
    counter = 0
    for i in range(len(a)):
        if a[i]!=0:
            if counter != 0:
                rle.append([counter, 0])
                counter = 0
            rle.append(a[i])
        else:
            counter += 1
    return rle
```

Рисунок 13. Вариант кода RLE-кодирования

Часть3: Кодирование кодом Хаффмана

In [32]:
```
# ЭТАП 7.3.1
# Часть кодирования кодом хаффмана, создающая дерево. В первую очередь составляется словарь
# частотности. По словарб частотности до опустошения списка из отсортированных ключей
# словаря объединением элементов и структур как элементов по очереди list_d.

def get_haffman_tree(array):
    freq = {0:0}
    for arr in array:
        for j in range(len(arr)):
            if isinstance(arr[j], int) or isinstance(arr[j], float):
                if arr[j] in freq:
                    freq[arr[j]] = freq.get(arr[j]) + 1
                else:
                    freq[arr[j]] = 1
            else:
                #print(freq.get(0))
                #print(arr[j])
                #print(arr[j][0])
                freq[0] = freq.get(0) + arr[j][0]

    # print(freq)
    list_d = list(freq.items())

    while len(list_d) > 1:
        list_d.sort(key=lambda i: i[1])
        list_d = [([list_d[0][0], list_d[1][0]], list_d[0][1]+ list_d[1][1])] + list_d[2::]
        #print(list_d, '\n')

    h_tree = list_d[0][0]

    return h_tree
```

Рисунок 14. Вариант кодирования дерева с использованием словаря частотности

In [23]:
```
# ЭТАП 7.3.2
# Рекурсивная функция записи по дереву кодов хаффмана в словарь

def get_haffman_codes(tree, code, codes):
    #print('get_haffman_codes', tree, code, codes)
    if len(tree)>0:
        if isinstance(tree[0], list):
            get_haffman_codes(tree[0], code + str(0), codes)
        elif isinstance(tree[0], int) or isinstance(tree[1], float):
            codes[tree[0]] = code + str(0)
    if len(tree)>1:
        if isinstance(tree[1], list):
            get_haffman_codes(tree[1], code + str(1), codes)
        elif isinstance(tree[1], int) or isinstance(tree[1], float):
            codes[tree[1]] = code + str(1)
    return codes
```

Рисунок 15. Вариант кодирования записи кода Хаффмана по дереву

In [48]:
```
# ЭТАП 7.3.3
# непосредственное кодирование кодом хаффмана по словарю

def haffman_code(h_codes, array):
    coparr = deepcopy(array)
    for i in range(len(array)):
        for j in range(len(array[i])):
            if isinstance(coparr[i][j], list):
                coparr[i][j][1] = h_codes.get(coparr[i][j][1])
            else:
                coparr[i][j] = h_codes.get(coparr[i][j])
    return coparr
```

Рисунок 16. Вариант шифрования кодом Хаффмана

Этап 8: JPEG сжатие изображения

In [73]:
```
# ЭТАП 8
# Кодирование JPEG последовательно по предшествующим этапам

def jpeg(img, coef):

    block_size = 8

    new_x_size, new_y_size  = \
        math.ceil(img.shape[0]/block_size)*block_size,\
        math.ceil(img.shape[1]/block_size)*block_size
    x_block_count, y_block_count = \
        int(new_x_size/block_size), int(new_y_size/block_size)

    y, u, v = make_y(img),  make_u(img),  make_v(img)
    print('YCbCr translated')
    sampled_u = sampling(u)
    sampled_V = sampling(v)
    print('sampling finished')

    splited_y = block_split(y, 8)
    splited_u = block_split(u, 8)
    splited_v = block_split(v, 8)
    print('spliting finished')

    # ЭТАП 4
    # использование встроенного дискретного косинусного преобразования dct
    y_lum = [dct(dct(block.T, norm='ortho').T, norm='ortho') for block in splited_y]
    u_chrom = [dct(dct(block.T, norm='ortho').T, norm='ortho') for block in splited_u]
    v_chrom = [dct(dct(block.T, norm='ortho').T, norm='ortho') for block in splited_v]
```

```
    print('discrete cosine transform imposed')

    y_lum = quantization(y_lum, x_block_count, y_block_count, luminosity, coef)
    u_chrom = quantization(u_chrom, x_block_count, y_block_count, chromaticity, coef)
    v_chrom = quantization(v_chrom, x_block_count, y_block_count, chromaticity, coef)
    print('channals quantizated')

    y_z = [zigzag(block) for block in y_lum]
    u_z = [zigzag(block) for block in u_chrom]
    v_z = [zigzag(block) for block in v_chrom]
    print('straighted to line')

    y_delta = delta_code(y_z)
    u_delta = delta_code(u_z)
    v_delta = delta_code(v_z)
    print('delta coding finished')

    y_delta = [rle_encode(array) for array in y_delta]
    u_delta = [rle_encode(array) for array in u_delta]
    v_delta = [rle_encode(array) for array in v_delta]
    print('rle coding finished')

    print('*')
    h_tree = get_haffman_tree(y_delta + u_delta + v_delta)
    # print(h_tree)
    h_codes = {}
    h_codes = get_haffman_codes(h_tree, '', h_codes)
    # print(h_codes)
    y_h = haffman_code(h_codes, y_delta)
    u_h = haffman_code(h_codes, u_delta)
    v_h = haffman_code(h_codes, v_delta)
    print('haffman coding finished')

    huffman_alph = {v:k for k, v in h_codes.items()}
    print(y_delta[0])
    print(y_h[0])
    #return 1
    return y_h, u_h, v_h, huffman_alph
```

Рисунок 17. Вариант кода JPEG сжатия

Обратное декодирование производит обратные функциям действия. Производилось частично параллельно написанию основного кода.

Выводы

Сжатии без потерь предполагает существование алгоритма, обратного алгоритму сжатия, позволяющего точно восстановить исходное изображение. Для алгоритмов сжатия с потерями обратного алгоритма не существует. Существует алгоритм, восстанавливающий изображение не обязательно точно совпадающее с исходным. Алгоритмы сжатия и восстановления подбираются так, чтобы добиться высокой степени сжатия и при этом сохранить визуальное качество изображения. Таким образом, сжатие в JPEG зачастую осуществляе ется за счет плавности изменения цветов в изображении и снижения качества изображения с минимальными потерями.

Приложение 1

1.1

Цветоразностный сигнал - цветовой видеосигнал, созданный путем вычитания яркостной и/или цветовой составляющей из одного из первичных цветовых сигналов (красного, зеленого, синего — RGB). В цветоразностном формате Betacam, например, яркость (Y) и цветоразностные компоненты (R-Y и B-Y) соотносятся ранее приведенными формулами.

Цветоразностный сигнал G-V не создается, так как он может быть реконструирован из остальных трех сигналов. Есть и другие соглашения о вычитании цветов: SMPTE, EBU-N1 0 и МП. Цветоразностные сигналы не следует называть компонентами видеосигнала. Этот термин зарезервирован для компонентов RGB. Нестрого термин «компонент видеосигнала» часто используется именно в смысле цветоразностных сигналов.

Формулы определяются с учетом матрицирования (- это операция передачи двух сигналов изображения с восстановлением третьего в приемнике путем вычитания переданных первичных сигналов из сигнала яркости. Операция возможна поскольку вычитание равнозначно сложению инвертированных, т. е. взятых в противоположной полярности сигналов. Во всех системах цветного телевидения принято передавать красный и синий) и в зависимости от спектральной чувствительности зрения настраиваются коэффициенты сигналов.

http://www.tvgarant.ru/glossary/cvetoraznostnyy-signal

http://principact.ru/content/view/92/29/

1.2

```
r = (0.587*im[:,:,1] + 0.114*im[:,:,2] + 1.402*(avg_v - 128)) / (1-0.299)
g = (0.299*im[:,:,0] + 0.114*im[:,:,2] - 0.34414*(avg_u - 128) - 0.71414*(avg_v - 128)) / (1-0.587)
b = (0.299*im[:,:,0] + 0.587*im[:,:,1] + 1.772*(avg_u - 128)) / (1-0.114)
```

Дополнительно

Принцип построения телевизионного сигнала

http://principact.ru/content/view/87/108/