Введение

В современном информационном мире одной из самых актуальных задач является сжатие изображений и видео. Один пример. Спутник, делающий снимки поверхности Земли в разных частотных диапазонах, передает информацию на Землю, которая затем помещается в хранилище изображений, где используются геоинформационными системами для самых разных целей. При этом, объем этой графической информации настолько велик, что при ее сжатии борются буквально за каждый байт, т.к. его хранение стоит денег. По некоторым оценкам дополнительное сжатие хотя бы на 5% дает выигрыш в миллионы долларов. Примерно та же ситуация сохраняется и при передаче изображений по каналам связи. Существующей пропускной способности не хватает, чтобы в полной мере удовлетворить потребности пользователей. Все выше сказанное и определяет актуальность задачи сжатия изображений.

Но чтобы говорить о сжатии информации необходимо сначала понять, за счет чего в принципе возможно сжатие конечных цифровых последовательностей? Чтобы показать принцип кодирования (сжатия) информации, который положен в основу всех современных кодеров сжатия без потерь, рассмотрим такой простой пример:

000011112222333555666666.

Теперь подряд идущие цифры заменим парой <значение, количество>, получим:

041424335465

и получаем сокращение исходной последовательности в 2 раза (24/12).

Если обобщить эту идею, то можно заключить, что кодер находит статистические закономерности в цифровой последовательности и за счет этого осуществляет сжатие (кодирование).

На сегодняшний день существует два метода сжатия без потерь – это статистические и словарные методы. Статистические методы играют на том, что если некоторый символ в последовательности встречается относительно часто, то ему ставится в соответствие короткий код, а если символ встречается реже – то ему присваивается более длинный код. В результате такого неравномерного кодирования, средняя длина последовательности становится короче. В другом, словарном методе, находятся похожие цепочки символов, которым присваиваются соответствующие более короткие коды. Эти методы независимы друг от друга и потому могут использоваться совместно для достижения более лучшего сжатия.

Рассмотрим более подробно, сначала, первый методы на примере кодирования по Хаффману.

Коды Хаффмана

Работа алгоритма начинается с составления списка символов (чисел) алфавита в порядке убывания их частоты (вероятности). Лучше всего продемонстрировать этот

алгоритм на простом примере. Пусть имеется пять символов: $a_1, a_2, ..., a_5$ с известными вероятностями: $p_1 = 0{,}4$, $p_2 = 0{,}2$, $p_3 = 0{,}2$, $p_4 = 0{,}1$ и $p_5 = 0{,}1$. Для построения кодов, выбираем сначала пару символов с наименьшими вероятностями – это символы a_4 и a_5. Наименее вероятному символу ставим в соответствие битовый ноль, а более вероятному – битовую единицу:

$$a_4 \to 1$$
$$a_5 \to 0$$

Символы a_4 и a_5 условно объединяются в единый символ a_{45} с вероятностью появления 0,2. Затем берется третий символ из упорядоченного списка – это символ a_3 с вероятностью 0,2. Поэтому код символа a_3 будет начинаться с битовой 1, а к кодам символов a_4 и a_5 дописывается битовый ноль:

$$a_3 \to 1$$
$$a_4 \to 10$$
$$a_5 \to 00$$

Далее, условно объединяем все три символа, в символ a_{345} с вероятностью появления 0,4 и рассматриваем его со следующим символом списка - a_2, вероятность которого 0,2. Так как вероятность появления символа a_2 меньше вероятности появления условного символа a_{345}, то код символа a_2 будет начинаться с нуля, а кодам символов a_3, a_4 и a_5 приписывается битовая единица:

$$a_2 \to 0$$
$$a_3 \to 11$$
$$a_4 \to 101$$
$$a_5 \to 001$$

Аналогично, получаем для последнего символа a_1:

$$a_1 \to 0$$
$$a_2 \to 01$$
$$a_3 \to 111$$
$$a_4 \to 1011$$
$$a_5 \to 0011$$

В итоге получаем коды Хаффмана, но записанные в обратном порядке, т.е. для кодирования символа a_1 имеем код 0, для a_2 - код 10, для a_3 - 111, a_4 - 1101 и a_5 - 1100. Заметим, что данные коды могут быть корректно декодированы, например, последовательность символов a_1, a_2, a_3, a_4, a_5 будет представлена последовательностью бит

01011111011100

В этой последовательности первым встречается битовый ноль, но с нуля начинается только один код – код символа a_1, поэтому он так и декодируется. Затем видим код, начинающийся с битовой единицы. Таких кодов несколько, поэтому необходимо прочитать следующий бит, который равен 0. Код 10 – единственный, соответствующий символу a_2. После этого видим код с тремя единицами подряд (111). Это говорит о том, что это символ a_3. Наконец последние два кода точно декодируют символы a_4 и a_5.

Построенные таким образом однозначные коды будут в среднем тратить 2,2 бита на символ:

$$0{,}4 \cdot 1 + 0{,}2 \cdot 2 + 0{,}2 \cdot 3 + 0{,}1 \cdot 4 + 0{,}1 \cdot 4 = 2{,}2 \text{ бит/символ}$$

Для сравнения, обычный двоичный код потребовал бы 3 бита для представления одного символа, т.е. в этом случае последовательность можно было бы сжать в 3/2,2=1,36 раза.

Однако чтобы произвести декодирование данных необходимо знать построенные коды. Самый лучший способ передать декодеру частоты появления символов в потоке, т.е. в данном случае дополнительно пять чисел. На основе этих частот декодер строит коды, а затем применяет их для декодирования последовательности. Поэтому, чем длиннее кодированная последовательность, тем меньше удельный вес служебной информации переданной декодеру.

И здесь возникает вопрос: а как можно численно определить степень статистической избыточности в заданной цифровой последовательности? Ведь тогда, зная эту характеристику мы могли бы определять максимально возможную степень сжатия этой последовательности и сравнивать конкретный алгоритм с этой нижней границей, т.е. мы могли бы объективно оценивать качество работы алгоритма сжатия. Оказывается, что такой величиной является энтропия, которая определяет минимальное число бит, необходимое для представления заданной последовательности чисел с последующей возможностью полного восстановления информации.

В 1948 г. сотрудник лаборатории Bell Labs Клод Шеннон показал, что минимальное число бит, которое необходимо затратить для представления одного символа той или иной информации можно найти с помощью формулы

$$H = -\sum_{i=1}^{N} p_i \log_2 p_i ,$$

где p_i - частота (вероятность) появления i-го числа в последовательности; N - число уникальных чисел в последовательности. Например, применяя данную формулу к нашей последовательности чисел, определяем, что число уникальных символов равно 5 – это a_1, a_2, a_3, a_4 и a_5. Частота появления цифр равна $p_1 = 0{,}4$, $p_2 = 0{,}2$, $p_3 = 0{,}2$, $p_4 = 0{,}1$ и $p_5 = 0{,}1$. В результате, минимальное число бит для представления одного символа в такой последовательности равно

$$H = -0{,}4 \log_2 0{,}4 - 2 \cdot 0{,}2 \log_2 0{,}2 - 2 \cdot 0{,}1 \log_2 0{,}1 \approx 2{,}1 \text{ бит/символ.}$$

Сравнивая полученный результат с ранее полученным для кодов Хаффмана (2,2 бит/символ), видим, что коды Хаффмана оказываются более близкими к нижней границе, чем равномерный код, которому необходимо 3 бита/символ.

Также можно посчитать и степень сжатия последовательности. Если равномерный код будет расходовать 3 бита на символ, то потенциальное сжатие будет равно

$$k = 3 / 2{,}1 = 1.4 \text{ раз},$$

а с помощью неравномерных кодов Хаффмана получим

$$k = 3 / 2{,}2 = 1{,}36 \text{ раз}.$$

Следовательно, представленный алгоритм сжатия не является лучшим. Кроме того, с помощью энтропии можно показать, что если последовательность содержит случайный набор чисел (не имеет закономерностей), то вероятности $p_i \to 1/N$, а возможность сжатия $k \to 1$, т.е. случайный набор данных сжать невозможно. Это положение в частности говорит о том, что если один раз к данным был применен хороший алгоритм сжатия, например, zip, то повторное сжатие этим или другим алгоритмом не приведет к лучшим результатам, скорее наоборот. Таким образом, на выходе любого хорошего алгоритма сжатия будет получаться почти случайный набор данных.

Адаптивные коды Хаффмана

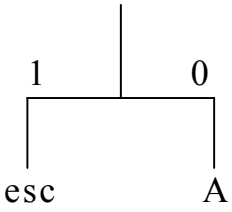
Работа рассмотренного выше алгоритма предполагает, что нам известны частоты появления символов. Для этого обычно просматривают кодируемую последовательность и подсчитывают число появлений того или иного символа, затем строят коды Хаффмана, а потом кодируют заданную последовательность. Такой подход имеет два недостатка: во-первых, приходится дважды просматривать кодируемую последовательность, что приводит к снижению скорости работы алгоритма, и, во-вторых, найденные частоты необходимо передавать декодеру для корректного декодирования данных. Поэтому на практике применяют другой метод формирования кодов переменной длины, который позволяет «на лету», по мере кодирования данных уточнять коды Хаффмана. В этом случае нет необходимости дважды просматривать последовательность и передавать коды приемной стороне, но за это приходится платить несколько худшим качеством сжатия. Такой алгоритм, например, лежит в основе программы compact операционной системы UNIX.

Основная идея адаптивного кодирования заключается в том, что компрессор и декомпрессор начинают работать с «пустого» дерева Хаффмана, а потом модифицируют его по мере чтения и обработки символов. Соответственно, и кодер и декодер должны модифицировать дерево одинаково, чтобы все время использовать один и тот же код, который может меняться по ходу процесса. Итак, в начале кодер строит пустое дерево Хаффмана, т.е. никакому символу коды еще не присвоены. Поэтому первый символ просто записывается в выходной поток в незакодированной форме, что обычно соответствует 8 битному коду ASCII. Затем, этот символ

помещается в дерево и ему присваивается код, например, 0. После этого кодируется следующий символ во входном потоке, и если этот символ встретился впервые, то он также записывается в выходной поток в виде 8 битного ASCII символа, и помещается в дерево Хаффмана, где ему присваивается определенный код в соответствии с его текущей частотой появления, равной 1. По мере того как поступают символы на вход кодера, происходит подсчет числа их появления и их количества и в соответствии с этой информацией выполняется перестройка дерева Хаффмана. Но здесь есть один нюанс: как отличить 8 битный ASCII символ от кода переменной длины в момент декодирования последовательности? Чтобы разрешить эту коллизию используют специальный esc (escape) символ, который показывает, что за ним следует незакодированный символ. Соответственно код самого esc символа должен находиться в дереве Хаффмана и будет меняться каждый раз по мере кодирования информации (перестройки дерева).

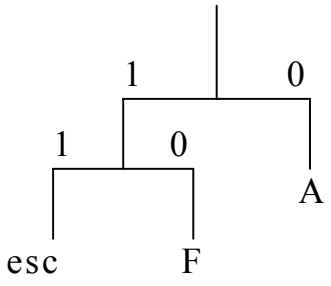
Рассмотрим пример реализации адаптивных кодов Хаффмана на последовательности символов AFFAADB. Так как вначале кодирования дерево Хаффмана пустое, то в выходной поток записывается 8-битовый код символа A (01000001), который затем добавляется в дерево Хаффмана (рис. 1, а). Следующий символ входного потока F также отсутствует в дереве Хаффмана, поэтому он записывается в выходной поток как код ASCII-символа, но перед ним ставится еще и esc-символ (рис. 1, б). Следующий, третий символ, снова F и для него имеется код в дереве Хаффмана, поэтому этот код записывается в выходной поток, счетчик для символа F увеличивается на единицу и становится равный двум и дерево Хаффмана принимает вид (рис. 1, в). Четвертый символ входного потока A также имеется в дереве Хаффмана, поэтому его код заносится в выходной поток, а счетчик принимает значение 2 (рис. 1, г). Аналогичные действия выполняются и при считывании следующего символа A (рис. 1, д).

Выходной поток: 01000001
Счетчик для A: 1



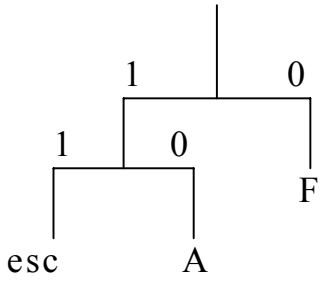
а)

Выходной поток: 01000001 1 01000110
Счетчик для A: 1
Счетчик для F: 1



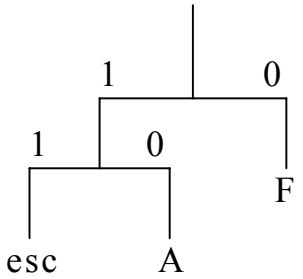
б)

Выходной поток: 01000001 1 01000110 10
Счетчик для A: 1
Счетчик для F: 2



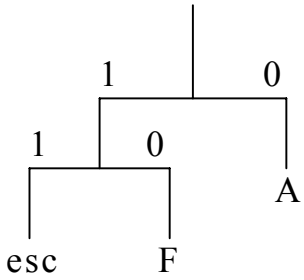
в)

Выходной поток: 01000001 1 01000110 10 10
Счетчик для A: 2
Счетчик для F: 2



г)

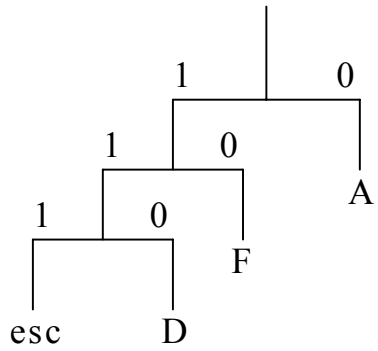
Выходной поток: 01000001 1 01000110 10 10 10
Счетчик для A: 3
Счетчик для F: 2



д)

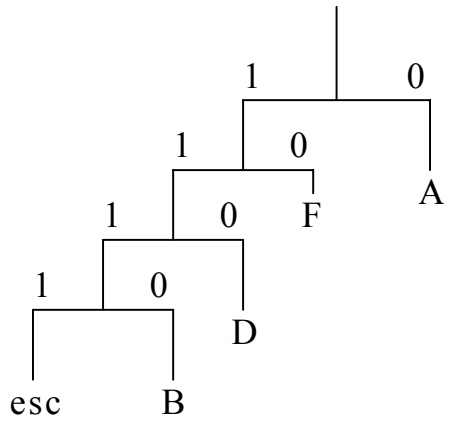
Следующий символ D является новым, поэтому он должен быть записан в выходной поток как ASCII-символ, перед которым следует поставить esc-символ (рис. 1, е). Здесь в дереве Хаффмана появляется третий уровень, т.к. сумма счетчиков для esc-символа и D равна 1 (счетчик esc всегда 0) и это меньше суммы значений счетчиков F и A. Наконец, последний символ B записывается в незакодированном виде в выходной поток и добавляется в дерево Хаффмана (рис. 1, ж).

Выходной поток: 01000001 1 01000110 10 10 10 11 01000100
Счетчик для A: 3
Счетчик для F: 2
Счетчик для D: 1



е)

Выходной поток: 01000001 1 01000110 10 10 10 11 01000100 111 01000010
Счетчик для A: 3
Счетчик для F: 2
Счетчик для D: 1
Счетчик для B: 1



ж)

В результате проведенного кодирования выходной поток составил 44 бит вместо 56, что соответствует сжатию в 1,27 раза. При кодировании более длинных последовательностей это значение будет увеличиваться, т.к. дерево Хаффмана будет содержать все больше и больше символов, которые будут представлены в выходном потоке как коды переменной длины, а не парами esc и ASCII символов.

Декодер при восстановлении исходной последовательности работает подобно кодеру, т.е. он также последовательно строит дерево Хаффмана по мере декодирования последовательности, что позволяет корректно определять исходные символы.

Адаптивное кодирование Хаффмана имеет ряд особенностей при своей работе. Первая связана с хранением значений счетчиков символов, при кодировании длинных последовательностей. Дело в том, что языки высокого уровня, обычно, оперируют с числами максимум в 32 бит, в которых можно записывать числа от 0 до 4294967296, т.е. можно работать с файлами размером 4Гб. Но иногда этих значений недостаточно и приходится вырабатывать дополнительные меры по предотвращению переполнения счетчиков символов. Вторая особенность связана с постоянным перестроением дерева Хаффмана, что влияет на скорость работы алгоритма. Вместе с тем можно заметить, что при считывании очередного символа из входного потока, дерево Хаффмана может остаться прежним и его перестраивать не имеет смысла. Это бывает, когда новое значение счетчика символа не влияет на его местоположении в дереве Хаффмана. Следовательно, при кодировании и декодировании необходимо определять: нужно или нет пересчитывать дерево Хаффмана и в зависимости от результата выполнять нужные действия. Вот две основные особенности работы адаптивного алгоритма кодирования Хаффмана, но, несмотря на указанные сложности, он часто применяется на практике, например, в известном протоколе V.32 передачи данных по модему со скоростью 14400 бод.

Арифметическое кодирование

Метод Хаффмана является простым, но эффективным только в том случае, когда вероятности появления символов равны числам $1/2^n$, где n - любое целое положительное число. Это связано с тем, что код Хаффмана присваивает каждому символу алфавита код с целым числом бит. Вместе с тем в теории информации известно, что, например, при вероятности появления символа равной 0,4, ему в идеале следует поставить код длиной $-\log_2 0{,}4 \approx 1{,}32$ бит. Понятно, что при построении кодов Хаффмана нельзя задать длину кода в 1,32 бита, а только лишь в 1 или 2 бита, что приведет в результате к ухудшению сжатия данных. Арифметическое кодирование решает эту проблему путем присвоения кода всему, обычно, большому передаваемому файлу вместо кодирования отдельных символов.

Идею арифметического кодирования лучше всего рассмотреть на простом примере. Предположим, что необходимо закодировать три символа входного потока, для определенности – это строка SWISS_MISS с заданными частотами появления символов: S – 0,5, W – 0,1, I – 0,2, M – 0,1 и _ - 0,1. В арифметическом кодере каждый символ представляется интервалом в диапазоне чисел [0, 1) в соответствии с частотой его появления. В данном примере, для символов нашего алфавита получим следующие наборы интервалов:

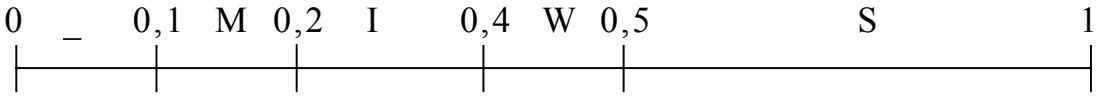


Рис. 2. Распределение интервалов представление символов

Процесс кодирования начинается со считывания первого символа входного потока и присвоения ему интервала из начального диапазона [0, 1). В данном случае для первого символа S получаем диапазон [0,5, 1). Затем, считывается второй символ – W, которому соответствует диапазон [0,4, 0,5). Но исходный диапазон [0, 1) уже сократился до [0,5, 1), поэтому символ W необходимо представить в этом новом диапазоне. Для этого достаточно вычислить новые нижнюю и верхнюю границы. Значение 0,4 будет соответствовать значению 0,7, а значение 0,5 – значению 0,75 (рис. 3).

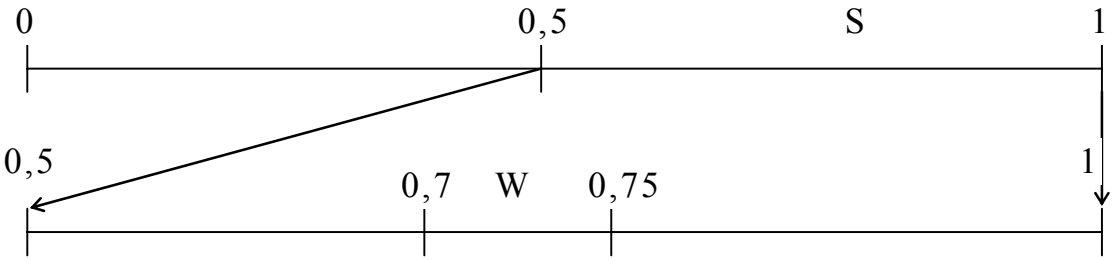


Рис. 3. Схема представления новых границ символа W

Данные границы можно вычислить по формулам:

NewHigh = OldLow + (OldHigh-OldLow)*HighRange(X),

NewLow = OldLow + (OldHigh-OldLow)*LowRange(X),

где OldLow – нижняя граница интервала, в котором представляется текущий символ; OldHigh – верхняя граница интервала; HighRange(X) – исходная верхняя граница кодируемого символа; LowRange(X) – исходная нижняя граница кодируемого символа. Применяя данные формулы к вычислению границ символа W, получаем:

OldLow = 0,5, OldHigh = 1,

HighRange(W) = 0,5, LowRange(W) = 0,4,

NewHigh = 0,5 + (1-0,5)*0,5 = 0,75,

NewLow = 0,5 + (1-0,5)*0,4 = 0,7.

Аналогичным образом выполняется кодирование символа I, для которого новые интервалы также можно вычислить по приведенной формуле:

OldLow = 0,7, OldHigh = 0,75,

HighRange(I) = 0,4, LowRange(I) = 0,2,

NewHigh = 0,7 + (0,75-0,7)*0,4 = 0,72,

NewLow = 0,7 + (0,75-0,7)*0,2 = 0,71.

Ниже, в табл. 1 представлены значения границ при кодировании строки SWISS_MISS.

Символ		Границы
S	L	0.0+(1.0-0.0)*0.5 = 0.5
	H	0.0+(1.0-0.0)*1.0 = 1.0
W	L	0.5+(1.0-0.5)*0.4=0.70

	H	0.5+(1.0-0.5)*0.5=0.75
I	L	0.7+(0.75-0.7)*0.2=0.71
	H	0.7+(0.75-0.7)*0.4=0.72
S	L	0.71+(0.72-0.71)*0.5=0.715
	H	0.71+(0.72-0.71)*1.0=0.72
S	L	0.715+(0.72-0.715)*0.5=0.7175
	H	0.715+(0.72-0.715)*1.0=0.72
_	L	0.7175+(0.72-0.7175)*0.0=0.7175
	H	0.7175+(0.72-0.7175)*0.1=0.71775
M	L	0.7175+(0.71775-0.7175)*0.1=0.717525
	H	0.7175+(0.71775-0.7175)*0.2=0.717550
I	L	0.717525+(0.717550-0.717525)*0.4=0.717530
	H	0.717525+(0.717550-0.717525)*0.5=0.717535
S	L	0.717530+(0.717535-0.717530)*0.5=0.7175325
	H	0.717530+(0.717535-0.717530)*1.0=0.717535
S	L	0.7175325+(0.717535-0.7175325)*0.5=0.71753375
	H	0.7175325+(0.717535-0.7175325)*1.0=0.717535

Конечный выходной код – это последнее значение переменной Low, равное 0.71753375, из которого следует взять лишь восемь цифр 71753375 для записи в файл.

Теперь рассмотрим возможность восстановления закодированной информации по восьми цифрам 71753375 и известным интервалам символов. Первая из восьми цифр – это 7, т.е. 0,7. Она принадлежит одному из заданных интервалов [0.5, 1), который соответствует символу S. Поэтому первый декодированный символ – это S. Теперь вернемся к рис. 3 и заметим, что второй символ был представлен в интервале символа S, т.е. [0.5, 1). Но для удобства декодирования его лучше представить в исходном интервале [0, 1). Для этого достаточно интервал [0.5, 1) увеличить до начального, т.е. умножить на два и границы сдвинуть на величину 0.5*2=1 (рис. 4).

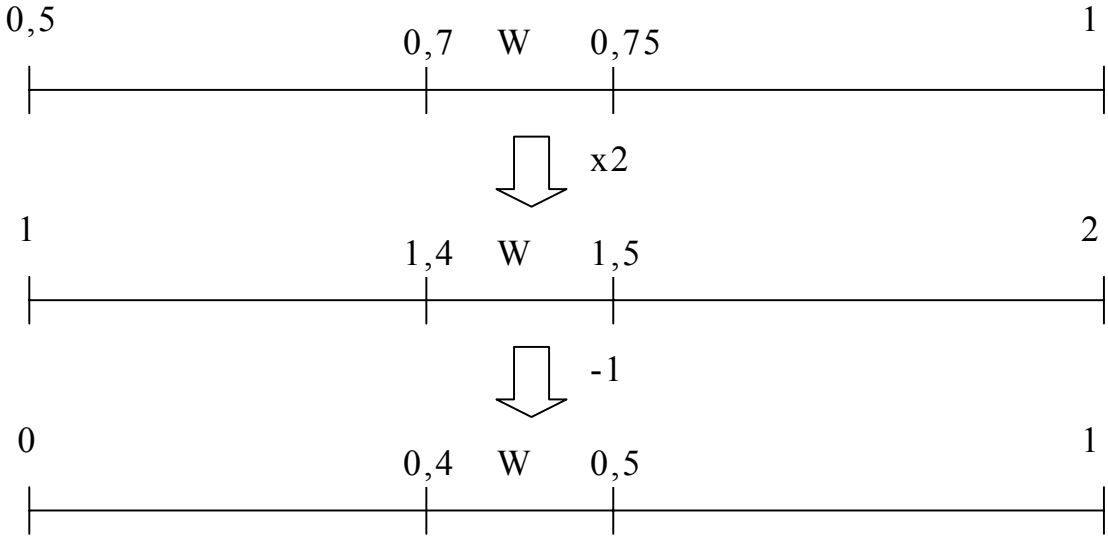


Рис. 4. Схема восстановления исходных интервалов символа W

Применяя данную схему к числу 0.71753375, получаем нижнюю границу следующего закодированного символа как будто он был начальным при кодировании:

$$0.71753375*2-1 = 0.4350675.$$

Полученное значение принадлежит диапазону [0.4, 0.5), который соответствует символу W. Затем, также полученное число 0.4350675 следует нормировать, что в общем случае выполняется по формуле:

$$Code = (Code-LowRange(X))/(HighRange(X)-LowRange(X)),$$

где Code – текущее значение кода. Например, пользуясь этой формулой применительно к коду 0.71753375, получаем значение

$$Code = (0.71753375-0.5)/(1-0.5)= 0.4350675,$$

которое в точности совпадает с предыдущей схемой вычисления. Аналогичным образом выполняется декодирование всех символов строки. В табл. 2 представлены коды, вычисляемые при декодировании символов. Здесь можно заметить, что процесс декодирования в данном случае можно остановить, если значение кода равно нулю.

Таблица 2. Вычисление кодов при декодировании

Символ	Code-Low	Область	
S	0.71753375 – 0.5	= 0.21753375	/ 0.5 = 0.4350675
W	0.4350675 – 0.4	= 0.0350675	/ 0.1 = 0.350675
I	0.350675 – 0.2	= 0.150675	/ 0.2 = 0.753375
S	0.753375 – 0.5	= 0.253375	/ 0.5 = 0.50675
S	0.50675 – 0.5	= 0.00675	/ 0.5 = 0.0135
_	0.0135 – 0	= 0.0135	/ 0.1 = 0.135
M	0.135 – 0.1	= 0.035	/ 0.1 = 0.35
I	0.35 – 0.2	= 0.15	/ 0.2 = 0.75
S	0.75 – 0.5	= 0.25	/ 0.5 = 0.5
S	0.5 – 0.5	= 0	/ 0.5 = 0

Однако это не всегда так. Бывают случаи, когда ноль содержит в себе код очередного символа, а не означает конец процедуры декодирования. Здесь возникает проблема завершения декодирования. Для этого используют специальный символ eof, говорящий о том, что он является последним и декодирование последовательности можно завершить. При этом частота этого символа очевидно должна быть маленькой по сравнению с частотой символов алфавита последовательности, например, [0,999999 1).

Описанный выше процесс кодирования невозможно реализовать на практике, т.к. в нем предполагается, что в переменных Low и High хранятся числа с неограниченной точностью. По существу, результат кодирования – это вещественное число с очень большой точностью. Например, файл объемом 1Мб будет сжиматься,

скажем, до 500 КБ, в котором будет записано одно число. Арифметические операции с такими числами реализовать сложно и долго. Поэтому любая практическая реализация арифметического кодера должна основываться на операциях с целыми числами, которые не должны быть слишком длинными. Рассмотрим такую реализацию, в которой переменные Low и High будут целыми числами длиной 16 или 32 бита. Эти переменные будут хранить верхние и нижние концы текущего подинтервала, но мы не будем им позволять неограниченно расти. Анализ табл. 1 показывает, что как только самые левые цифры переменных Low и High становятся одинаковыми, они уже не меняются в дальнейшем. Следовательно, эти цифры можно выдвинуть за скобки и работать с оставшейся дробной частью. После сдвига цифр мы будем справа дописывать 0 в переменную Low, а в переменную High – цифру 9. Для того, чтобы лучше понять весь процесс, можно представлять себе эти переменные как левый конец бесконечно длинного числа. Число Low имеет вид xxxx00… а число High = yyyy99…

Проблема состоит в том, что переменная High в начале должна равняться 1, однако мы интерпретируем Low и High как десятичные дроби меньшие 1. Решение заключается в присвоении переменной High значения 9999…, которое соответствует бесконечной дроби 0,9999…, равной 1.

Как это все работает? Закодируем этим способом ту же строку SWISS_MISS. Первая буква S определена диапазоном [0,5 1) и формально границы равны

L=0.0+(1.0-0.0)*0.5=0.5

H=0.0+(1.0-0.0)*1.0=1.0

но в нашем случае данные формулы следует преобразовать, чтобы переменные Low и High были целыми. Поэтому запишем их в таком виде:

Low=0+(10000-0)*0.5=5000

High=0+(10000-0)*1.0=10000

но по условию граница High является открытой, т.е. не включает число 10000, поэтому от конечного значения нужно отнять 1:

High=0+(10000-0)*1.0-1=9999

Таким образом, получили формулы для вычисления целочисленных границ Low и High, и процесс кодирования будет выглядеть так (табл. 3).

Табл. 3. Кодирование сообщения сдвигами

1		2			4	5
S	L=	0+(10000-0)*0.5	=	5000		5000
	H=	0+(10000-0)*1.0-1	=	9999		9999
W	L=	5000+(10000-5000)*0.4	=	7000	7	0000
	H=	5000+(10000-5000)*0.5	=	7499		4999
I	L=	0+(5000-0)*0.2	=	1000	1	0000
	H=	0+(5000-0)*0.4-1	=	1999		9999
S	L=	0+(10000-0)*0.5	=	5000		5000
	H=	0+(10000-0)*1.0-1	=	9999		9999
S	L=	5000+(10000-5000)*0.5	=	7500		7500
	H=	5000+(10000-5000)*1.0-1	=	9999		9999

_	L=	7500+(10000-7500)*0.0	=	7500	7	5000
	H=	7500+(10000-7500)*0.1-1	=	7749		7499
M	L=	5000+(7500-5000)*0.1	=	5250	5	2500
	H=	5000+(7500-5000)*0.2	=	5499		4999
I	L=	2500+(5000-2500)*0.2	=	3000	3	0000
	H=	2500+(5000-2500)*0.4-1	=	3499		4999
S	L=	0+(5000-0)*0.5	=	2500		2500
	H=	0+(5000-0)*1.0-1	=	4999		4999
S	L=	2500+(5000-2500)*0.5	=	3750	3750	3750
	H=	2500+(5000-2500)*1.0-1	=	4999		4999

На последнем шаге операции кодирования записываются все 4 цифры и полученная выходная последовательность имеет вид: 717533750.

Декодер работает в обратном порядке. В начале переменным Low и High присваиваются значения 0000 и 9999 соответственно, а переменной Code значение 7175. На основе этой информации требуется определить первый закодированный символ. Для этого число переменной Code нужно корректно представить в интервале от 0 до 1. В самом начале интервал такой и есть, поэтому частота символа, соответствующая значению Code будет равна

index = 7175/10000=0,7175.

Эта частота попадает в диапазон [0,5 1) и соответствует символу S. Теперь границы Low и High пересчитываются, так как это делалось в кодере и принимают значения 5000 и 9999 соответственно. Так как значащие цифры этих переменных отличаются, то переменная Code остается прежней и величина

index = (7175-5000)/(10000-5000)=0,4350.

Это значение попадает в диапазон [0,4 0,5) и соответствует символу W. После этого величины Low и High принимают значения 7000 и 7499 и после отбрасывания значащей цифры переходят в 0000 и 4999 соответственно, а переменная Code преобразуется в 1753. Таким образом раскодируется вся последовательность.

На практике обычно значение index принимает целочисленные значения, которые вычисляются по формуле

index = ((Code-Low)*10-1)/(High-Low+1)

и округляют до ближайшего целого.

Может показаться, что приведенный выше пример не производит никакого сжатия. Для того чтобы выяснить степень сжатия результаты кодирования нужно перевести в двоичную форму. Так как из конечного интервала

[0.71753375 0.717535)

можно выбрать любое число, выберем наименьшее для хранения – это 717534, которому соответствует битовое представление 10101111001011011110 и составляет 20 бит. И строка из 10 символов сжимается в 20 бит. Хорошее ли это сжатие? Для этого нужно найти энтропию кодируемой последовательности и она будет равна

$$H(X) = -0{,}5\log_2 0{,}5 - 0{,}2\log_2 0{,}2 - 0{,}1\log_2 0{,}1 - 0{,}1\log_2 0{,}1 - 0{,}1\log_2 0{,}1 \approx 1{,}96 \text{ бит/сим}$$

и составит величину

$$I = H(X) \cdot 10 \approx 19{,}6 \text{ бит},$$

что в целых значениях равно 20 бит. Следовательно, полученный код достиг минимально возможного значения и является оптимальным. В общем случае можно показать, что при достаточно большой последовательности арифметический кодер всегда приводит к оптимальным результатам сжатия, т.е. является наилучшим среди всех энтропийных кодеров.

Здесь можно добавить про адаптивное арифметическое кодирование

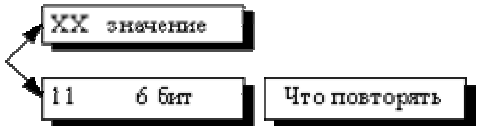
Словарные методы

Статистические методы компрессии используют статистическую модель данных, и качество сжатия информации напрямую зависит от того, насколько хороша была эта модель. Методы основанные на словарном подходе, не рассматривают статистические модели. Вместо этого они выбирают некоторые последовательности символов, сохраняют их в словаре, а все последовательности кодируются в виде меток (кодов, чисел), используя словарь.

Алгоритм RLE. Первый вариант алгоритма

Данный алгоритм необычайно прост в реализации. Групповое кодирование — от английского Run Length Encoding (RLE) — один из самых старых и самых простых алгоритмов архивации графики. Изображение в нем (как и в нескольких алгоритмах, описанных ниже) вытягивается в цепочку байт по строкам растра. Само сжатие в RLE происходит **за счет того, что в исходном изображении встречаются цепочки одинаковых байт**. Замена их на пары <счетчик повторений, значение> уменьшает избыточность данных.

В данном алгоритме признаком счетчика служат единицы в двух верхних битах считанного файла:



Соответственно оставшиеся 6 бит расходуются на счетчик, который может принимать значения от 1 до 64. Строку из 64 повторяющихся байтов мы превращаем в два байта, т.е. сожмем в 32 раза.

Алгоритм рассчитан на деловую графику — изображения с большими областями повторяющегося цвета. Ситуация, когда файл увеличивается, для этого простого алгоритма не так уж редка. Ее можно легко получить, применяя групповое кодирование к обработанным цветным фотографиям. Для того, чтобы увеличить

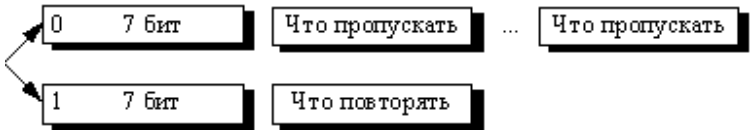
изображение в два раза, его надо применить к изображению, в котором значения всех пикселов больше двоичного 11000000 и подряд попарно не повторяются.

Данный алгоритм реализован в формате PCX.

Второй вариант алгоритма

Второй вариант этого алгоритма имеет больший максимальный коэффициент архивации и меньше увеличивает в размерах исходный файл.

Признаком повтора в данном алгоритме является единица в старшем разряде соответствующего байта:



Как можно легко подсчитать, в лучшем случае этот алгоритм сжимает файл в 64 раза (а не в 32 раза, как в предыдущем варианте), в худшем увеличивает на 1/128. Средние показатели степени компрессии данного алгоритма находятся на уровне показателей первого варианта.

Похожие схемы компрессии использованы в качестве одного из алгоритмов, поддерживаемых форматом TIFF, а также в формате TGA.

LZW

Название алгоритм получил по первым буквам фамилий его разработчиков — Lempel, Ziv и Welch. Сжатие в нем, в отличие от RLE, осуществляется уже за счет одинаковых цепочек байт.

Алгоритм LZW

Процесс сжатия выглядит достаточно просто. Мы считываем последовательно символы входного потока и проверяем, есть ли в созданной нами таблице строк такая строка. Если строка есть, то мы считываем следующий символ, а если строки нет, то мы заносим в поток код для предыдущей найденной строки, заносим строку в таблицу и начинаем поиск снова. LZW реализован в форматах GIF и TIFF.

LZW - это способ сжатия данных, который извлекает преимущества при повторяющихся цепочках данных. Поскольку растровые данные обычно содержат довольно много таких повторений, LZW является хорошим методом для их сжатия и раскрытия.

В данный момент давайте рассмотрим обычное кодирование и декодирование с помощью LZW-алгоритма. В GIF используется вариация этого алгоритма.

Первой вещью, которую мы делаем при LZW-сжатии является инициализация нашей цепочки символов. Чтобы сделать это, нам необходимо выбрать код размера (количество бит) и знать сколько возможных значений могут принимать наши символы. Давайте положим код размера равным 12 битам, что означает возможность запоминания 0FFF, или 4096, элементов в нашей таблице цепочек. Давайте также предположим, что мы имеем 32 возможных различных символа. (Это соответствует, например, картинке с 32 возможными цветами для каждого пиксела.) Чтобы инициализировать таблицу, мы установим соответствие кода #0 символу #0, кода #1 символу #1, и т.д., до кода #31 и символа #31. На самом деле мы указали, что каждый код от 0 до 31 является корневым. Больше в таблице не будет других кодов, обладающих этим свойством.

Процесс сжатия выглядит достаточно просто. Мы считываем последовательно символы входного потока и проверяем, есть ли в созданной нами таблице строк такая строка. Если строка есть, то мы считываем следующий символ, а если строки нет, то мы заносим в поток код для предыдущей найденной строки, заносим строку в таблицу и начинаем поиск снова.

Пусть мы сжимаем последовательность 45, 55, 55, 151, 55, 55, 55. Тогда, согласно изложенному выше алгоритму, мы поместим в выходной поток сначала код очистки <256>, потом добавим к изначально пустой строке “45” и проверим, есть ли строка “45” в таблице. Поскольку мы при инициализации занесли в таблицу все строки из одного символа, то строка “45” есть в таблице. Далее мы читаем следующий символ 55 из входного потока и проверяем, есть ли строка “45, 55” в таблице. Такой строки в таблице пока нет. Мы заносим в таблицу строку “45, 55” (с первым свободным кодом 258) и записываем в поток код <45>. Можно коротко представить архивацию так:

- “45” — есть в таблице;
- “45, 55” — нет. Добавляем в таблицу <258>“45, 55”. В поток: <45>;
- “55, 55” — нет. В таблицу: <259>“55, 55”. В поток: <55>;
- “55, 151” — нет. В таблицу: <260>“55, 151”. В поток: <55>;
- “151, 55” — нет. В таблицу: <261>“151, 55”. В поток: <151>;
- “55, 55” — есть в таблице;
- “55, 55, 55” — нет. В таблицу: “55, 55, 55” <262>. В поток: <259>;

Последовательность кодов для данного примера, попадающих в выходной поток: <256>, <45>, <55>, <55>, <151>, <259>.

Особенность LZW заключается в том, что для декомпрессии нам не надо сохранять таблицу строк в файл для распаковки. Алгоритм построен таким образом, что мы в состоянии восстановить таблицу строк, пользуясь только потоком кодов.

Мы знаем, что для каждого кода надо добавлять в таблицу строку, состоящую из уже присутствующей там строки и символа, с которого начинается следующая строка в потоке.

Следует отметить, что в целях экономии памяти ЭВМ таблицу цепочек можно эффективно представлять, учитывая следующую ее особенность: если, например, цепочка ASDFK входит в таблицу цепочек, то ASDF тоже входит в нее. Это обстоятельство наводит на мысль, что вместо запоминания в таблице всей цепочки, достаточно запомнить код для ASDF, за которым следует замыкающий символ K.

Для повышения степени сжатия изображений данным методом часто используется одна «хитрость» реализации этого алгоритма. Некоторые изображения, подвергаемые сжатию с помощью LZW, имеют часто встречающиеся цепочки одинаковых значений, например 12 12 12 … или 32 32 32 … и т. п. Их непосредственное сжатие будет генерировать выходной код 12 12 258 и т.д. Спрашивается, можно ли в этом частном случае повысить степень сжатия? Оказывается да, если мы оговорим следующие действия по кодированию и декодированию. Кодирование таких цепочек будем осуществлять следующим образом. Первый цвет 12 записывает с его же кодом из таблицы 12. Следующий цвет тоже 12, тогда добавляем в таблицу строку 12 12 с кодом 258 и в выходной поток сразу заносим этот код, т.е. 258. Смотрим входной поток дальше. Если на вход опять поступает цвет 12, он есть в таблице, смотрим следующий – 12, последовательность 12 12 тоже есть в таблице, смотрим дальше – 12, последовательности 12 12 12 присваиваем код 259 и заносим его в выходной поток. В результате на выходе получаем последовательность 12 258 259 …, которая гораздо короче прямого кодирования стандартным методом LZW.

Можно показать, что такая последовательность будет корректно восстановлена. Декодировщик сначала читает первый код – это 12, которому соответствует цвет 12. Затем читает код 258, но этого кода в его таблице нет. Но мы уже знаем, что такая ситуация возможна только в том случае, когда добавляемый символ равен первому символу только что считанной последовательности, т.е. 12. Поэтому он добавит в свою таблицу строку 12 12 с кодом 258, а в выходной поток поместит 12 12. И так может быть раскодирована вся цепочка кодов.

Мало того, описанное выше правило кодирования мы можем применять в общем случае не только к подряд идущим одинаковым цветам точек, но и к последовательностям, у которых очередной добавляемый символ равен первому символу цепочки.

LZW в GIF

Алгоритм LZW имеет несколько особенностей своей реализации в формате сжатия изображений gif. Первая особенность – это переменный размер кода таблицы цепочек, который не может превышать 12 бит, т.е. не превышать числа 4095. Вторая особенность состоит в использовании двух специальных кодов – это код обновления (реинициализации) таблицы цепочек <CC>, и код завершения потока символов <EOI>.

В самом начале своей работы алгоритм определяет количество цветов, используемых в изображении. В случае GIF их максимум может быть 256, т.к. любое изображение, даже с большим набором цветов преобразуется в 256 цветовое пространство. Минимум может быть 2 цвета. Если используется только два цвета, то начальный размер кодов в таблице равен 3 битам. Причем коду 0 ставится цвет 0, а коду 1 – цвет 1. Коды 4 и 5 соответствуют коду очистки таблицы <CC> и коду <EOI>. При большем количестве цветов размер кода таблицы равен числу бит N, приходящихся на один пиксел. При этом специальные коды равны 2^N и 2^N+1. Начальный размер кодов в таблице записывается в заголовок GIF файла.

Кодирование пикселей изображения начинается кодами размером $N+1$ бит. По мере накопления таблицы будут увеличиваться значения кодов и как только очередной код достигает значения $2^{N+1}-1$, то это значит, что значение N необходимо увеличить на 1, иначе значение кода превысит прежний размер в $N+1$ бит.

Разработчики формата GIF ограничили максимальный размер кодов в таблице 12 битами. Это значит, что когда код достигает значения $2^{12}-1=4095$, то размер N увеличивать уже нельзя. Но в то же время и размер кодов становится больше 12 бит. Как разрешить данную ситуацию? Простым решением является выполнить перегенерирование таблицы последовательностей, после чего она будет содержать только цепочки длины, равной 1, т.е. значения пикселей и плюс специальные коды. Таким образом, она перейдет в то же состояние, в котором была на момент начала кодирования изображения. А для того, чтобы декодировщик знал, что в определенный момент произошло перегенерирование таблицы, кодировщик в выходной поток записывает код управляющего символа <CC>.

Цветовые пространства

Известно, что цветовое изображение требует не менее трех чисел на один пиксел для точной передачи его цвета. Метод, выбранный для представления яркости и цвета, называется цветовым пространством.

Есть три наиболее популярные цветовые модели – это RGB (использующееся в компьютерной графике); YIQ, YUV или YCbCr (использующейся в видеосистемах); и CMYK (использующейся в цветовой печати). Все цветовые пространства могут быть получены из RGB пространства извлекаемое камерами и сканерами.

RGB

Данное цветовое пространство наиболее широко используется в компьютерной графике. Красный, зеленый и голубой главные компоненты цветов и представляют три размерности данного пространства (рис. 3). Указанная диагональ куба с равными значениями RGB указывает градации серого от черного до белого.

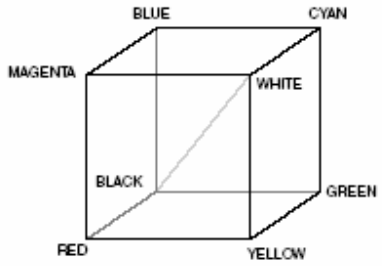


Рис. 3. Куб RGB цветов

Цветовые ЭЛТ и жидкокристаллические дисплеи отображают RGB изображения, отдельно освещая красные, зеленые и голубые компоненты каждого пиксела. Если смотреть на экран с расстояния обычного зрителя, то различные компоненты сливаются в единый «правильный цвет».

RGB пространство подходит для компьютерной графики, т.к. там для формирования цвета как раз и используются эти три компоненты. Однако RGB не очень эффективно, когда речь заходит о реальных изображениях. Дело в том, что для сохранения цвета изображений, необходимо знать и хранить все три компоненты RGB и потеря одной из них сильно исказит визуальное качество изображения. Также при обработке изображений в RGB пространстве не всегда удобно бывает изменить только яркость или контрастность отдельного пиксела, т.к. в этом случае необходимо будет прочитать все три значения компонент RGB, пересчитать их для желаемой яркости и записать обратно. По этим и другим причинам многие стандарты видео используют яркость и два цветоразностных сигнала как цветовую модель, отличную от RGB. Наиболее известными среди таких пространств являются YUV, YIQ и YCbCr. Несмотря на то, что все они связаны между собой, тем не менее имеются некоторые отличия.

YCbCr

Известно, что органы зрения человека менее чувствительны к цвету предметов, чем к их яркости. В цветовом пространстве RGB все три компонента считаются одинаково важными, и они обычно сохраняются с одинаковым разрешением. Однако можно отобразить цветовое изображение более эффективно, отделив светимость от цветовой информации и представив ее с большим разрешением, чем цвет. Поэтому

цветовое пространство YCbCr и его вариации является популярным методом эффективного представления цветных изображений.

Буква Y в таких цветовых пространствах обозначает компоненту светимость, которая вычисляется как взвешенное усреднение компонент R, G и B по следующей формуле:

$$Y = k_r R + k_g G + k_b B,$$

где k обозначает соответствующий весовой множитель. Остальные цветовые компоненты по существу определяются в виде разностей между светимостью Y и компонентами R, G и B:

$$Cb=B-Y,$$
$$Cr=R-Y,$$
$$Cg=G-Y.$$

При этом получаются четыре компоненты нового пространства вместо трех RGB. Однако число Cb+Cr+Cg является постоянным, поэтому только две из трех хроматических компонент необходимо хранить, а третью вычислять на основе них. Чаще всего в качестве две искомых цветовых компонент используют Cb и Cr. Преимущество пространства YCbCr по сравнению с RGB заключается в том, что Cb и Cr можно представлять с меньшим разрешением, чем Y, т.к. глаз человека менее чувствителен к цвету предметов, чем к их яркости. Это позволяет сократить объем информации, требуемый для представления хроматических компонент, без заметного ухудшения качества передачи цветовых оттенков изображения. Такой подход к преобразованию цветового пространства дает дополнительный эффект при сжатии цветных изображений. При этом алгоритмы сжатия сначала преобразуют исходное цветовое пространство из RGB в YCbCr, сжимают, а затем при восстановлении обратно преобразуют изображение в цветовое пространство RGB, т.к. оно используется в ЭВМ. При этом формулы для прямого и обратного преобразований выглядят следующим образом:

$$\left.\begin{aligned} Y &= k_r R + (1-k_b-k_r)G + k_b B, \\ Cb &= \frac{0{,}5}{1-k_b}(B-Y), \\ Cr &= \frac{0{,}5}{1-k_r}(R-Y). \end{aligned}\right\} \text{прямое преобразование}$$

$$\left.\begin{aligned} R &= Y + \frac{1-k_r}{0{,}5} Cr, \\ G &= Y - \frac{2k_b(1-k_b)}{1-k_b-k_r} C_b - \frac{2k_r(1-k_r)}{1-k_b-k_r} C_r, \\ B &= Y + \frac{1-k_b}{0{,}5} C_b. \end{aligned}\right\}$$ обратное преобразование

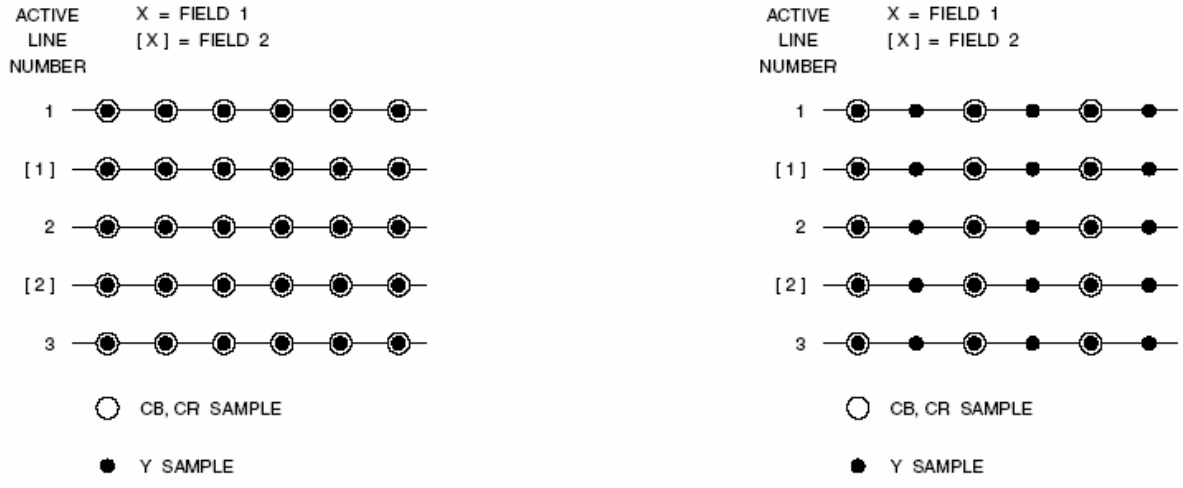
Отметим, что множитель kg получается из соотношения $k_g + k_r + k_b = 1$, а величина компоненты G получается вычитанием суммы Cb и Cr из Y.

Рекомендация ITU-T предлагает следующие коэффициенты: $k_b = 0{,}114$ и $k_r = 0{,}229$. Используя эти значения в данных уравнениях, получаем широко распространенные формулы:

$$Y = 0{,}299R + 0{,}587G + 0{,}114B,$$
$$Cb = 0{,}564(B - Y),$$
$$Cr = 0{,}713(R - Y);$$
$$R = Y + 1{,}402Cr,$$
$$G = Y - 0{,}344Cb - 0{,}714Cr,$$
$$B = Y + 1{,}772Cb.$$

Как отмечалось выше хроматические компоненты Cb и Cr могут быть представлены с меньшим разрешением, чем световая компонента Y. При этом на практике используют следующие форматы их взаимного представления.

Самый очевидный формат это так называемый формат 4:4:4, который означает полную точность в передаче хроматических компонент, т.е. на каждые 4 световые отсчеты Y передаются по 4 отсчета компонент Cb и Cr (рис. 4 а).

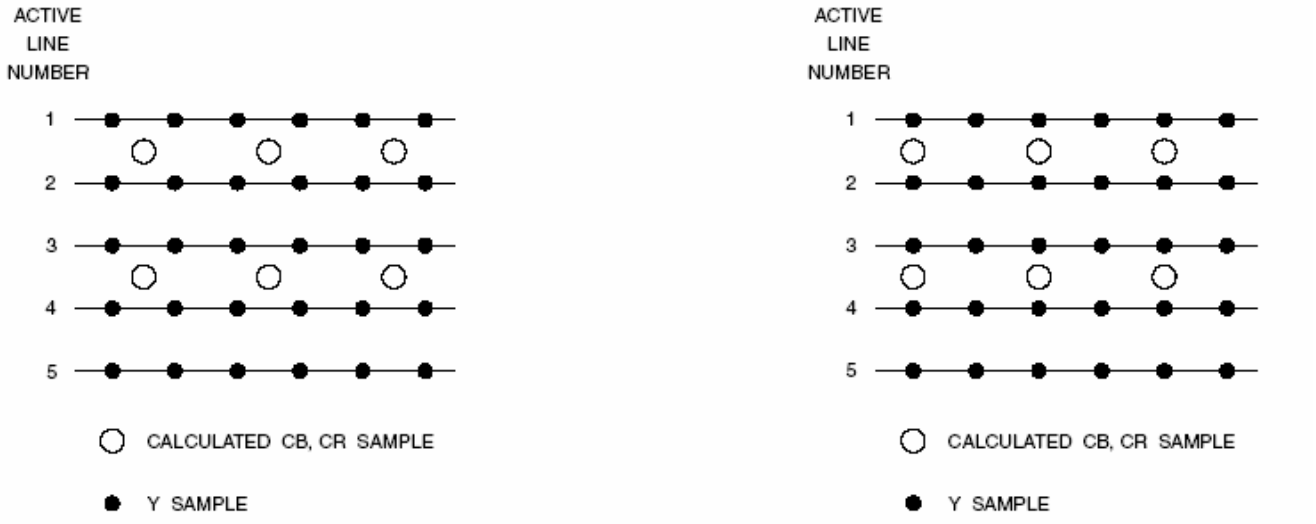


а) б)

Рис. 4. Расположение хроматических компонент

Другой формат 4:2:2 (YUY2) предполагает, что на каждые 4 отсчета компоненты Y приходится по два отсчета хроматических компонент, расположение которых представлено на рис. 4, б. Данный формат используется для высококачественного цветного видео и используется в стандартах MPEG-4 и H.264.

Наиболее популярный формат сэмплирования 4:2:0 (YV12) каждая компонента Cb и Cr имеет один отсчет на 4 отсчета Y (рис. 5 а, б). Причем отсчеты компонент Cb и Cr, как правило, вычисляются двумя способами. В первом случае выполняется интерполяция по 4 ближайшим отсчетам компонент Cb и Cr для формирования одного отсчета для них (рис. 5, а). Такой подход применяется в стандартах MPEG-1 и H.261, H.263. В другом случае выполняется интерполяция по двум вертикальным отсчетам (рис. 5, б) и применяется в стандарте MPEG-2.



а) б)

Рис. 5. Представление формата 4:2:0

Благодаря экономичному представлению цветных сцен, формат 4:2:0 широко используется во многих потребительских приложениях, таких как видеоконференции, цифровое телевидение, DVD. Поскольку хроматические компоненты отбираются в 4 раза реже компонент яркости, то пространство 4:2:0 YCbCr занимает в 2 раза меньше отсчетов по сравнению с форматом видео 4:4:4 RGB.

Сжатие изображений

Рассматриваемые ниже алгоритмы сжатия ориентированы на реальные изображения, получаемые, например, с помощью цифрового фотоаппарата или с помощью сканирования, либо системами захвата кадров с видео и т.п. В любом случае будем предполагать, что у нас имеется полноцветное изображение, представляющее собой двумерный массив, элементы которого содержат цвет соответствующей точки. При этом задачей кодирования является представить

исходное изображение как можно меньшим числом байт по сравнению с исходным размером.

Степень сжатия принято определять либо как коэффициент сжатия, равный

$$k = \frac{S_{код}}{S_{исх}},$$

где $S_{код}$ - размер закодированного (сжатого) изображения; $S_{исх}$ - размер исходного изображения, либо как фактор сжатия:

$$f = \frac{1}{k} = \frac{S_{исх}}{S_{код}}.$$

Таким образом, коэффициент k всегда меньше 1, а фактор сжатия больше 1. Обычно в литературе при сравнении алгоритмов сжатия используют фактор сжатия и говорят, например, что то или иное изображение сжато в 2 раза, и это означает, что фактор сжатия равен 2.

Все алгоритмы сжатия изображений можно условно разбить на два основных класса – это алгоритмы сжатия без потерь качества при восстановлении закодированного изображения, и алгоритмы с некоторой потерей качества восстановления. В первом случае гарантируется точное соответствие между исходным и восстановленным изображениями, но при этом достигается, как правило, невысокая степень сжатия, обычно 2-3. При сжатии с некоторой потерей качества, которая мало заметна для глаза, удается получить более высокую компрессию изображений в 10 и более раз. При этом выбор в пользу того или иного алгоритма сжатия следует делать в зависимости от конкретной прикладной задачи.

Когда речь заходит о сжатии с потерей качества, то возникает вопрос: каким образом следует оценивать качество восстановленного изображения? Универсального критерия оценки качества изображений не существует, поэтому для определения на сколько хорошо было восстановлено изображение пользуются наиболее подходящими критериями, из тех, что известны. Рассмотрим подробно наиболее распространенные критерии качества оценки изображений.

Критерий визуального восприятия

В случаях, когда преобразованное изображение предназначается для визуального восприятия, экспертам-наблюдателям предъявляется набор пар изображений (преобразованные и исходные), которые высказывают суждения на уровне: «искажения незаметны», «заметны, но не ухудшают», «ухудшают, но не мешают», «немного мешают», и т.п. Индивидуальные оценки обрабатываются и усредняются. Существуют специальные приемы, исключающие «привыкание» экспертов в процессе экспериментов, их пристрастия к конкретным сюжетам, и т.п.

Проведение подобной экспертизы – всегда сложная задача, и ее результаты весьма приблизительны. Кроме того, для специальных изображений (которые, например, получаются при дистанционном зондировании) эксперты должны быть

специалистами по решению соответствующих прикладных задач анализа видеоинформации.

Главный недостаток данного субъективного критерия – отсутствие количественных оценок. Он не позволяет решать задачи оптимизации систем обработки изображений в пространстве непрерывно меняющихся параметров. Поэтому желательно, чтобы критерий имел простую аналитическую форму и просто вычислялся по предъявляемым изображениям. Этому требованию удовлетворяет ряд критериев, рассматриваемых ниже.

Среднеквадратичный критерий

Пусть изображения $f(n_1,n_2)$ и $g(n_1,n_2)$ описываются моделями однородных случайных полей. Мерой соответствия преобразованного изображения начальному может служить среднее значение квадрата их разности:

$$\varepsilon_{кв}^2 = M\{(f-g)^2\}.$$

Известно, что для эргодических случайных процессов значение математического ожидания можно вычислить с использованием одной реализации согласно следующему выражению:

$$\varepsilon_{кв}^2 = \frac{1}{N_1 N_2}\sum_{n_1=0}^{N_1-1}\sum_{n_2=0}^{N_2-1}[f(n_1,n_2)-g(n_1,n_2)]^2.$$

Приведенное выражение позволяет вычислять среднеквадратическую ошибку для пар изображений, не обязательно описываемых стационарными полями. Однако в этом случае следует иметь в виду, что значение $\varepsilon_{кв}^2$ будет характеризовать «среднее» качество изображения в целом, а на различных его фрагментах ошибки могут сильно отличаться.

Достоинство среднеквадратичного критерия – его простота. При его использовании многие задачи анализа и оптимизации алгоритмов обработки изображений решаются относительно просто. Поэтому он часто применяется на практике.

При обработке изображений следует учитывать, что данный критерий плохо согласуется с критерием субъективного восприятия.

PSNR

Общепринятой величиной для оценки потерь при восстановлении изображений является метрика, называемая пиковое отношение сигнал/шум или по-английски PSNR. При этом, чем больше значение PSNR, тем меньше потерь при восстановлении и наоборот. Однако полученное значение не дает гарантию, что зрителю понравится восстановленный образ.

Данный критерий определяется выражением:

$$PSNR = 20\log_{10}\frac{\max_{ij}|x_{ij}|}{\sigma_\varepsilon},$$

где x_{ij} - значение яркости точки с координатами (i, j); σ_ε - среднеквадратическое отклонение между исходным и восстановленным сигналами изображений.

Число PSNR безразмерно, но из-за использования логарифма говорится, что PSNR измеряется в дБ. Использование логарифма сглаживает σ_ε, делает эту величину менее чувствительной. Например, деление σ_ε на 10 означает умножение PSNR на 2. Отметим, что PSNR не имеет абсолютного значения. Бессмысленно говорить, что если PSNR равно, скажем, 25, то это хорошо. Величина PSNR используется только для сравнения качества восстановления разных алгоритмов между собой. К примеру, комитет MPEG использует субъективный порог PSNR=0,5 дБ при включении кодовой оптимизации, поскольку считает, что улучшение на эту величину будет заметно глазу. Обычно PSNR варьируется в пределах от 20 до 40.

Критерий максимальной ошибки (равномерного приближения)

$$\varepsilon_{\max} = \max_{(n_1,n_2)}|f(n_1,n_2) - g(n_1,n_2)|.$$

Это строгий критерий. Он используется в тех случаях, когда выдвигается требование высокой точности представления изображения не в целом, а каждой его точки (отсчета). Это необходимо в ответственных случаях, при получении ценных, уникальных изображений.

Однако данный показатель имеет серьезный недостаток – сложность теоретической оценки и, соответственно, использования его в процедурах оптимизации.

ДИКМ и JPEG-LS

Данный метод кодирования основывается на предположении наличия корреляционной связи между соседними отсчетами изображения. В этом случае значение последующего отсчета x_i оценивается на основе предыдущих $x_{i-1}, x_{i-2}, \ldots$ Полученная оценка $\hat{x}_i$ используется для формирования разностного сигнала $\varepsilon_i = x_i - \hat{x}_i$, которая будет тем меньше, чем точнее построена текущая оценка. Можно доказать, что при наличии корреляционной зависимости между отсчетами сигнала, дисперсия ошибок оценивания $\bar{\varepsilon}$ будет меньше дисперсии исходного сигнала $\bar{x}$. Следовательно для представления $\bar{\varepsilon}$ можно использовать меньше уровней, чем для сигнала $\bar{x}$, т.е. каждый отсчет может быть представлен не 8 битами, а меньшим числом – обычно 4 битами.

В общем случае оценка элемента строится на основе линейной комбинации нескольких предыдущих отсчетов:

$$\hat{x}_i = \sum_{j=1}^{n} x_{i-j}\alpha_j = [x_{i-1}, x_{i-2}, ..., x_{i-n}] \cdot \begin{vmatrix} \alpha_1 \\ \alpha_2 \\ ... \\ \alpha_n \end{vmatrix} = X_i^T \overline{\alpha},$$

где $\overline{\alpha}$ - набор весовых коэффициентов, которые выбираются из условия минимума дисперсии ошибки $\varepsilon_i = x_i - \hat{x}_i$:

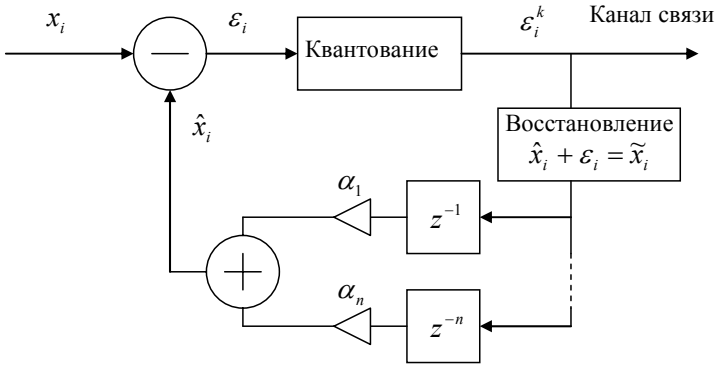
$$\sigma_{\varepsilon i}^2 = M\left\{\left(x_i - \hat{x}_i\right)^2\right\} \to \min.$$

Вектор весовых коэффициентов $\overline{\alpha}$ можно найти путем дифференцирования данного выражения и приравнивания результат нулю.

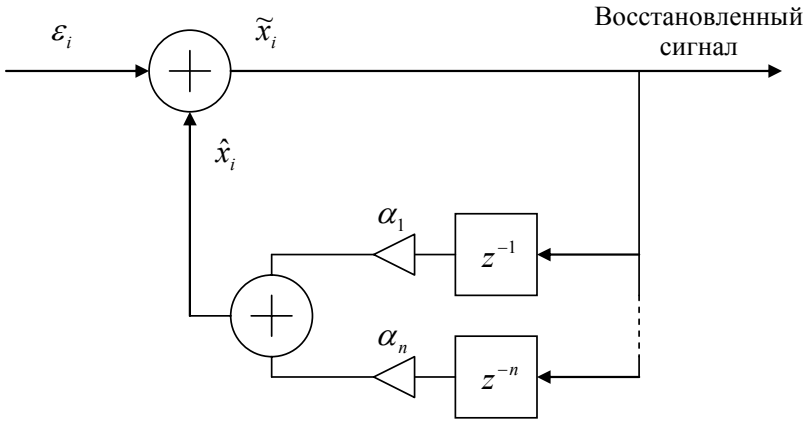
Полученная ошибка подвергается равномерному квантованию с шагом d согласно следующей формуле:

$$\varepsilon_i^k = \left\lceil \left(x_i - \hat{x}_i\right)/d \right\rceil \cdot d.$$

В результате возникает шум квантования $q_i = \varepsilon_i - \varepsilon_i^k$. Для того, чтобы величина шума не накапливалась при восстановлении сигнала, оценки элементов $\hat{x}_i$ строят на основе восстановленных значений. Таким образом получаем следующую схему алгоритма кодирования (рис. 4).



а)



б)

Рис. 4. Структурные схемы кодера и декодера ДИКМ

Алгоритм ДИКМ применяется в стандарте JPEG при сжатии изображений без потерь. В начале отсчеты исходного изображения заменяются на целочисленные разности между истинным значением яркости пиксела и его целочисленным (округленным) прогнозом. При этом прогноз текущего пиксела $\hat{x}_{ij}$ строится по трем ближайшим соседям (рис. 5). Сам прогноз строится по одному из выбранных вариантов (табл. 1).

Таблица 1. Коэффициенты алгоритма ДИКМ стандарта JPEG

Номер варианта	Коэффициенты
1	$a_1 = 1, a_2 = 0, a_3 = 0$
2	$a_1 = 0, a_2 = 1, a_3 = 0$
3	$a_1 = 0, a_2 = 0, a_3 = 1$
4	$a_1 = -1, a_2 = 1, a_3 = 1$
5	$a_1 = -0.5, a_2 = 0.5, a_3 = 1$
6	$a_1 = -0.5, a_2 = 1, a_3 = 0.5$
7	$a_1 = 0, a_2 = 0.5, a_3 = 0.5$

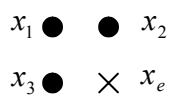


Рис. 5. Расположение наблюдений и оцениваемого элемента

Затем, полученные целочисленные разности сжимаются кодами Хаффмана и формируется выходной файл сжатого изображения.

Алгоритм ДИКМ строит прогноз на основе предыдущих значений отсчетов. Однако известно, что лучший прогноз можно построить, если использовать не только предыдущие, но и последующие отсчеты в изображении относительно оцениваемого. Алгоритм, использующий эту идею при построении оценок прогноза пикселов изображения носит название «иерархическая сеточная интерполяция».

Иерархическая сеточная интерполяция

Идея метода состоит в следующем. Временной сигнал, представленный на рис. 6, делится на две составляющие: $\lambda_i = x_{2i-1}$ - нечетные отсчеты; $\gamma_i = x_{2i}$ - четные отсчеты.

На основе нечетных элементов $\overline{\lambda_1}$ строятся оценки четных элементов $\overline{\gamma_1}$ с помощью линейной комбинации

$$\hat{x}_{2i} = \sum_{j=1}^{n} \lambda_{i-j} \alpha_j ,$$

где $\overline{\alpha}$ - вектор весовых коэффициентов. Затем, подобно алгоритму ДИКМ, вычисляются ошибки оценивания $\gamma_i = x_{2i} - \hat{x}_{2i}$, которые квантуются и передаются по

каналу связи. В свою очередь коэффициенты $\bar{\lambda}_1$, имеющие длину в два раза меньшую по сравнению с сигналом $\bar{x}$, подвергаются такому же преобразованию. В результате получается последовательность $\bar{\lambda}_2$ длиной в четыре раза меньше длины сигнала $\bar{x}$ и две последовательности $\bar{\gamma}_1$ и $\bar{\gamma}_2$. Суммарная длина последовательностей $\bar{\lambda}_2$, $\bar{\gamma}_2$ и $\bar{\gamma}_1$ равна длине сигнала $\bar{x}$. Данное преобразование можно повторять до тех пор, пока коэффициенты $\bar{\lambda}_n$ не будут представлять собой один элемент. При этом коэффициенты $\{\bar{\gamma}_1, \bar{\gamma}_2, ..., \bar{\gamma}_n\}$ будут описывать соответствующие ошибки оценивания, дисперсия которых, как правило, меньше дисперсий ошибок $\bar{\varepsilon}$ в алгоритме ДИКМ. Следовательно, коэффициенты $\{\bar{\gamma}_1, \bar{\gamma}_2, ..., \bar{\gamma}_n\}$ можно представить меньшим числом уровней квантования при сохранении хорошего качества восстановления.

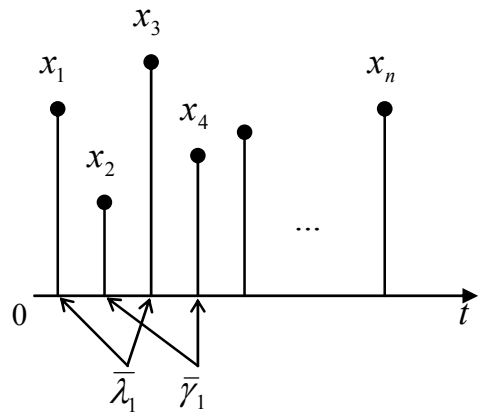


Рис. 6. Схема разделения временного сигнала

Восстановление осуществляется с уровня n с использованием последовательностей $\bar{\lambda}_n$ и $\bar{\gamma}^k{}_n$. В результате получается последовательность $\bar{\lambda}_{n-1}$, в которой нечетные элементы представляют собой отсчеты $\bar{\lambda}_n$, а четные определяются по формуле

$$\tilde{\lambda}_{2i}^{n-1} = \sum_{j=1}^{m} \lambda_{i-j}^{n} \alpha_j + \gamma_i^n = \hat{\lambda}_{2i}^{n-1} + \gamma_i^n.$$

Причем значения четных элементов будут отличаться от истинных на величину шума квантования $q_i = \gamma_i - \gamma_i^k$. На следующем шаге восстановления используются последовательности $\tilde{\bar{\lambda}}_{n-1}$ и $\bar{\gamma}^k{}_{n-1}$. В результате ошибки восстановления будут становиться все больше и больше, и вносить заметные искажения в восстановленный сигнал $\hat{\bar{x}}$. Для исправления этой ситуации, последовательности $\{\bar{\gamma}_1, \bar{\gamma}_2, ..., \bar{\gamma}_n\}$ лучше вычислять, начиная с уровня n, а затем вычислять величины $\bar{\gamma}_n$.

После квантования величин $\bar{\gamma}_n$ выполняется формирование последовательности $\tilde{\lambda}_{n-1}$, в которой нечетные элементы соответствуют элементам последовательности $\bar{\lambda}_n$, а четные определяются по формуле $\tilde{\lambda}_{2i}^{n-1} = \hat{\lambda}_{2i}^{n-1} + \gamma_i^n$.

На основе полученной новой последовательности $\tilde{\tilde{\lambda}}_{n-1}$ вычисляются коэффициенты $\bar{\gamma}_{n-1}$. В результате ошибка оценивания не будет нарастать и значения восстановленного сигнала $\hat{\bar{x}}$ будут отличаться от истинных $\bar{x}$ только на величину шума квантования q_i.

Интересной особенностью данного преобразования является то, что ошибки оценивания на первом, уровне описывают мелкие детали сигнала, и их можно положить равными нулю, а другие последовательности $\{\bar{\gamma}_2,...,\bar{\gamma}_n\}$ квантовать со все более точной шкалой на каждом последующем уровне. В результате можно добиться представление всех коэффициентов $\{\bar{\gamma}_1,\bar{\gamma}_2,...,\bar{\gamma}_n\}$, в среднем 0,5-1 бит на элемент при приемлемом качестве восстановления. Следовательно, для передачи таких данных потребуется канал с пропускной способностью $(0{,}5 \div 1) \cdot 8000 = 4 \div 8$ кбит/сек.

Разложение сигнала изображений по базисным векторам

Рассматриваемые выше методы преобразования отсчетов изображения оперировали яркостями в исходном пространстве, т.е. том, в котором они воспринимаются зрителем. Вместе с тем, любой коррелированный сигнал, к которым и относятся реальные изображения, можно эффективно, с точки зрения сжатия, представить в другом пространстве, так, что основная доля информации будет приходится на небольшое число коэффициентов разложения. Конкретизируем более подробно данную идею преобразования сигнала изображения.

Для того чтобы иметь возможность представить вектор в пространстве N измерений необходимо задать систему координат. Оси системы координат определяются векторами. В качестве простого примера рассмотрим ортогональную систему координат двух измерений, называемую декартовой. Базисные вектора описываются выражениями

$$\bar{i}_1 = \{1,0\},\ \bar{i}_2 = \{0,1\}.$$

Графически такую систему координат можно представить следующим образом:

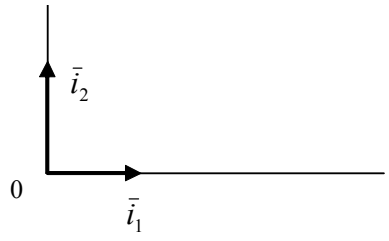


Рис. 1. Графическое представление базисных векторов

Увеличим размерность пространства на единицу и добавим к уже существующим базисным векторам $\bar{i}_1$ и $\bar{i}_2$ вектор $\bar{i}_3$. Соответствующая система координат примет вид:

$$\bar{i}_1 = \{1,0,0\}$$
$$\bar{i}_2 = \{0,1,0\}.$$
$$\bar{i}_3 = \{0,0,1\}$$

Из полученных выражений базисных векторов двух и трех измерений следует, что такое определении базиса представляет собой единичную матрицу, в которой строки или столбцы определяют координаты базисных векторов. Таким образом, для любого N - мерного пространства легко определить базис как единичную матрицу размерности $N \times N$ элементов.

Данное определение базиса в N - мерном пространстве не является единственным. В общем случае в качестве векторов $i_1, i_2, ..., i_N$ можно выбрать другой набор векторов $e_1, e_2, ..., e_N$. Причем их ориентация будет всегда определяться относительно базиса $i_1, i_2, ..., i_N$. При этом набор векторов $e_1, e_2, ..., e_N$ будет образовывать базис в N - мерном пространстве только тогда, когда проекции любого N-мерного вектора на данные вектора будут уникальны. Это условие обеспечивается в том случае, когда вектора $e_1, e_2, ..., e_N$ линейно независимы.

Определим математически данное условие. Пусть вектор $X = [x_1, x_2, ..., x_N]^T$ задан в базисе векторов $\bar{i}_1$, $\bar{i}_2$,…, $\bar{i}_N$. Тогда его координаты в новом базисе $\bar{e}_1, \bar{e}_2, ..., \bar{e}_N$ вычисляются как скалярные произведения вектора X с векторами $\bar{e}_i, i = \overline{1, N}$:

$$y_i = < X, \bar{e}_i > = \sum_{j=1}^{N} x_j e_{ij}\ ,\ i = \overline{1, N}\ ,$$

Или в матричной форме

$$Y = \begin{vmatrix} y_1 \\ y_2 \\ \dots \\ y_N \end{vmatrix} = AX = \begin{vmatrix} \bar{e}_1 \\ \bar{e}_2 \\ \dots \\ \bar{e}_N \end{vmatrix} \begin{vmatrix} x_1 \\ x_2 \\ \dots \\ x_N \end{vmatrix}.$$

Матрица A размерностью $N \times N$ элементов, составленная из базисных векторов $\bar{e}_i, i = \overline{1, N}$, называется матрицей преобразования.

Из условия единственности представления вектора X следует, что вычисленные координаты $y_i, i = \overline{1, N}$, должны уникальным образом описывать вектор X в базисе векторов $\{\bar{e}_i\}_{i=\overline{1,N}}$. Следовательно, существует обратное преобразование из Y в X, которое можно записать следующим образом:

$$A^{-1}Y = A^{-1}AX \Rightarrow X = A^{-1}Y\ .$$

Данное выражения справедливо, т.к. произведение матрицы на соответствующую обратную матрицу дает единичную, а произведение вектора X на единичную матрицу даст тот же вектор X. Таким образом, необходимым и достаточным условием того, что векторы $\bar{e}_i$ образуют базис в N - мерном

пространстве является существование обратной матрицы составленной из данных векторов.

Представленное разложение векторов можно обобщить на двумерный случай. Пусть задана матрица X размерностью $N \times N$ элементов. Ее проекции y_{ij} в базисе векторов $\bar{e}_i, i = \overline{1, N}$ вычисляются как

$$y_{ij} = \sum_{i=1}^{N} \sum_{j=1}^{N} a_{ijkl} x_{kl} .$$

Данное выражение показывает, что для преобразования двумерных сигналов, в общем случае, необходимо задавать четырехмерную матрицу преобразования A.

Однако в случае разделимого преобразования можно воспользоваться обычными двумерными матрицами, и которое в матричном виде записывается как

$$Y = AXA^T . \tag{2}$$

Обратное преобразование имеет вид

$$X = A^{-1} X (A^{-1})^T . \tag{3}$$

Из формул (2) и (3) видно, что основная доля вычислений приходится на нахождение обратной матрицы. Сократить объем вычислений можно, если воспользоваться ортогональными базисными векторами, которые удовлетворяют условию

$$\begin{cases} \sum_{i=1}^{N} \varphi_{mi} \varphi_{ni}^* = c, & \text{при } m = n, \\ \sum_{i=1}^{N} \varphi_{mi} \varphi_{ni}^* = 0, & \text{при } m \neq n, \end{cases} \tag{4}$$

где φ_i^* - базисные вектора, соответствующие обратной матрице A^{-1}. Из выражения (4) следует, что $A^{-1} = \frac{1}{c} A^T$. Тогда обратное преобразование (1) можно записать в виде

$$X = \frac{1}{c} A^T Y \frac{1}{c} A = \frac{1}{c^2} A^T Y A .$$

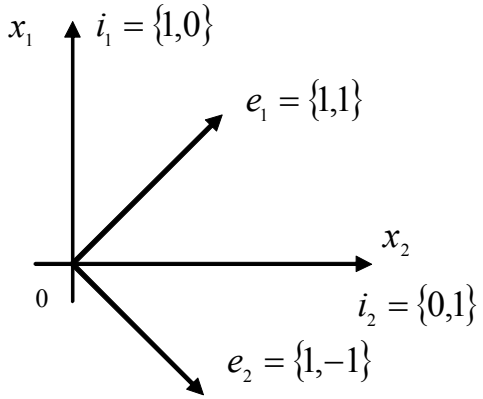
В этом случае матрица A называется матрицей ортогонального преобразования, а само преобразование унитарным.

Преобразование Адамара

Самым простым унитарным преобразованием является преобразование Адамара. Матрица H для случая двух случайных величин будет иметь вид:

$$H = \begin{vmatrix} 1 & 1 \\ 1 & -1 \end{vmatrix} .$$

Если из элементов матрицы H составить базисные вектора $e_1 = [h_{11}, h_{12}] = [1,1]$, $e_2 = [h_{21}, h_{22}] = [1,-1]$, то они будут характеризовать поворот ортогональной системы координат на 45° относительно единичного базиса.



Если случайные величины x_1 и x_2 имеют корреляционную зависимость $r = M\{x_1 x_2\} \neq 0$, то проекция вектора $X = [x_1, x_2]^T$ на базисный вектор e_1 будет, в среднем, больше, чем проекция этого же вектора на базисный вектор e_2. Благодаря этому информация будет сосредотачиваться в первом коэффициенте преобразования. Второй коэффициент служит для уточнения представления вектора X в новом базисе.

Рассмотрим подробнее процесс разложения вектора $X = [x_1, x_2]^T$ по базисным векторам Адамара. Проекция $y_1 = \langle X, \bar{e}_1 \rangle = x_1 + x_2$ представляет собой удвоенное среднее значение элементов вектора X, проекция $y_2 = \langle X, \bar{e}_2 \rangle = x_1 - x_2$ - удвоенную разность между средним значением и элементом x_1 (рис. 2).

Выполним восстановление вектора X по первому коэффициенту y_1 с помощью обратной матрицы $H^{-1} = \frac{1}{2} H^T = \frac{1}{2} \begin{vmatrix} 1 & 1 \\ 1 & -1 \end{vmatrix}$, получим:

$$\hat{X}_1 = \frac{1}{2} H^T \begin{vmatrix} y_1 \\ 0 \end{vmatrix} = \begin{vmatrix} y_1/2 \\ y_1/2 \end{vmatrix}. \qquad (5)$$

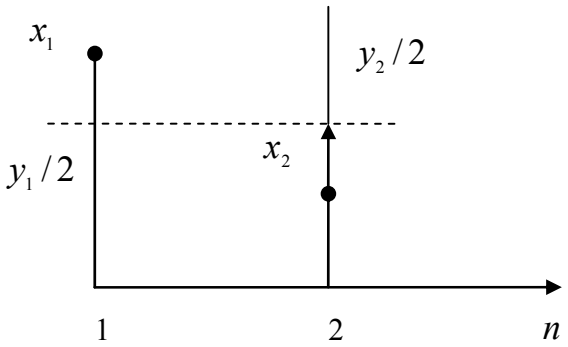


Рис. 2. Графическое представление проекций

Из выражения (5) видно, что восстановленный вектор $\hat{X}_1$ представляет собой средние значения элементов x_1 и x_2, что соответствует «грубому» приближению вектора X, без наличия мелких деталей.

Рассмотрим теперь восстановление того же вектора по коэффициенту y_2, получим:

$$\hat{X}_2 = [y_2/2, -y_2/2]^T. \quad (6)$$

Анализ выражения (6) показывает, что вектор $\hat{X}_2$ описывает только мелкие детали вектора X. При этом можно заметить, что сумма $\hat{X}_1 + \hat{X}_2$ даст исходный вектор X, что соответствует восстановлению по обоим коэффициентам разложения.

При увеличении размерности пространства большая часть информации будет сосредоточена в малом числе коэффициентов

Матрицу Адамара размерности 4x4 элементов легко построить из матрицы Адамара H размерностью 2x2 элемента:

$$H_4 = \begin{vmatrix} H & H \\ H & -H \end{vmatrix} = \begin{vmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{vmatrix},$$

Пользуясь данным соотношением можно построить матрицу Адамара любой размерности $N = 2^n$, где n - любое целое положительное число.

Анализ изображений выполняется на основе разделимого преобразования:

$$Y = HXH^T.$$

Так как матрица H представляет собой оператор ортогонального преобразования, то обратное преобразование из Y в X запишется в виде

$$X = \frac{1}{N^2} H^T Y H.$$

Восстановление изображения X по неполному числу коэффициентов разложения, также как и для случая двух случайных величин, будет приводить к похожим эффектам сглаживания и выделения мелких деталей. При этом возникают ошибки восстановления $\varepsilon_{ij} = x_{ij} - \hat{x}_{ij}$, которым можно поставить в соответствие некоторую функцию потерь $f(\bar{\varepsilon})$. Значение этой функции будет характеризовать качество восстановления. На практике часто используют квадратичную функцию потерь

$$f(\bar{\varepsilon}) = \sum_{i,j} \{\varepsilon_{ij}^2\} = \sum_{i,j} (x_{ij} - \hat{x}_{ij})^2. \quad (7)$$

Так как ε_{ij} носит случайный характер, то значение $f(\bar{\varepsilon})$ также является случайной величиной. При этом желательно, чтобы $f(\bar{\varepsilon})$, в среднем, была минимальна. Для этого перепишем выражение (7) в виде

$$\sigma_\varepsilon^2 = M\{f(\bar{\varepsilon})\} = \sum_{i,j} M\{\varepsilon_{ij}^2\} = \sum_{i,j} M\{(x_{ij} - \hat{x}_{ij})^2\}, \quad (8)$$

где M - знак математического ожидания. Найдем базисные вектора, минимизирующие (8).

Преобразование Карунена-Лоэва

В задачах сжатия изображений лучшим преобразованием является то, в котором на первый базисный вектор, в среднем, приходилась бы наибольшая информация об изображении, на второй максимум из оставшейся и т.д. Рассмотрим построение такого пространства со следующих позиций. Без потери общности предположим, что у нас имеется набор из N одномерных векторов S_i, $i = \overline{1,N}$. Для них требуется найти такой вектор φ_1, чтобы минимизировалась квадратическая разность

$$\varepsilon^2 = \sum_{i=1}^{N} (S_i - a_i\varphi_1)^2, \quad (1)$$

где a_i - некоторые пока неизвестные весовые коэффициенты. Очевидно, что φ_1 можно найти путем дифференцирования данного выражения по φ_1 и приравнивания результата нулю:

$$\frac{\partial\varepsilon^2}{\partial\varphi} = -2\sum_{i=1}^{N}(S_i - a_i\varphi_1)a_i^T = 0 \Rightarrow \varphi_1 = \frac{\sum_{i=1}^{N} S_i a_i^T}{\sum_{i=1}^{N} a_i^2}. \quad (2)$$

Но в этом выражении остаются неизвестными коэффициенты a_i. Однако они также должны быть выбраны из соображений минимума квадратической ошибки, следовательно, a_i находится как

$$\frac{\partial\varepsilon^2}{\partial a_i} = -2\varphi_1^T(S_i - a_i\varphi_1) = 0 \Rightarrow a_i = \frac{\varphi_1^T S}{\varphi_1^T\varphi_1}$$

и полагая $\varphi_1^T\varphi_1 = 1$, получим $a_i = \varphi_1^T S_i$, т.е. a_i есть не что иное как проекция сигнала S_i на вектор φ_1. Подставляя данное выражение в (2), можно записать следующее рекуррентное выражения для нахождения φ_1:

$$\varphi_1^n = \sum_{i=1}^{N} S_i S_i^T \varphi_1^{(n-1)} = \left(\sum_{i=1}^{N} S_i S_i^T\right)\varphi_1^{(n-1)}. \quad (3)$$

В последнем выражении опущен множитель $1/\sum_{i=1}^{N} a_i^2$, т.к. это число определяет лишь масштаб вектора φ_1, который нами был уже задан в виде соотношения $\varphi_1^T\varphi_1 = 1$, а значит $\sqrt{\|\varphi_1\|^2} = \|\varphi_1\| = 1$.

Выражение (3) описывает рекуррентное выражение для вычисления вектора φ_1, минимизирующего (1), которое сходится до установившегося значения уже на 20-30 итерации. В качестве начальных условий можно выбрать любой не нулевой вектор, например $\varphi_1 = [1,1,1,...,1]^T$. Таким образом, нашли первый базисный вектор, на который будет приходиться максимум информации о сигналах $\{S_i\}$ в виде проекций $\{a_i\}$. По условию второй базисный вектор должен содержать максимум оставшейся информации, т.е. той, которая не вошла в φ_1. Для нахождения φ_2 достаточно из векторов $\{S_i\}$ вычесть информацию, находящуюся в φ_1, получим:

$$S_i^{(2)} = S_i - a_i\varphi_1,\ a_i = S_i^T\varphi_1,\ i = \overline{1,N}.$$

Используя теперь уже векторы $\{S_i^{(2)}\}$, базисный вектор φ_2 будет вычисляться по аналогии с φ_1 по формуле

$$\varphi_2^n = \sum_{i=1}^{N} S_i^{(2)} S_i^{(2)T} \varphi_2^{(n-1)} = \left(\sum_{i=1}^{N} S_i^{(2)} S_i^{(2)T} \right) \varphi_2^{(n-1)}.$$

Следуя этому правилу, можем вычислить полный набор базисных векторов $\{\varphi_i\}$, которые наилучшим образом будут локализовывать информацию о сигналах $\{S_i\}$ в проекциях. Благодаря этому будет достигнут минимум ошибки восстановления (1) по неполному набору коэффициентов разложения.

Полученный алгоритм вычисления векторов $\{\varphi_i\}$ соответствует алгоритму вычисления собственных векторов матрицы $\left(\sum_{i=1}^{N} S_i^{(2)} S_i^{(2)T} \right)$. Следовательно, можно заключить, что наилучшая локализация сигналов достигается в пространстве собственных векторов, вычисленных на основе суммарной матрицы автокорреляции набора векторов $\{S_i\}$.

Следует также отметить, что при $N \to \infty$ матрица $\frac{1}{N}\left(\sum_{i=1}^{N} S_i^{(2)} S_i^{(2)T} \right)$ переходит в автоковариационную матрицу векторов $\{S_i\}$. Это значит, что если известна автокорреляционная матрица изображений, то оптимальное преобразование будет определяться собственными векторами этой матрицы.

Учитывая, что базисные векторы преобразования КЛ вычисляются исходя из статистических свойств кодируемых изображений, то в конкретном методе сжатия эти векторы следует записывать в сжатый файл для использования декодером. Кроме того, не существует быстрого алгоритма вычисления этих векторов. Все эти факты делают метод КЛ сугубо теоретическим без реальных приложений. В связи с этим возникает проблема поиска преобразования, которое обладало бы малой вычислительной сложностью и давало бы результаты преобразования близкие к преобразованию Карунена-Лоэва. Этим условиям удовлетворяет преобразование Фурье.

Преобразование Фурье

Удобной формой записи и реализации ФП является его комплексное представление, которое для одномерного случая имеет вид

$$X(k)=\sum_{k=0}^{N-1}x(n)e^{-j\frac{2\pi}{N}nk} \quad \text{- прямое ПФ,}$$

$$x(n)=\frac{1}{N}\sum_{k=0}^{N-1}X(k)e^{j\frac{2\pi}{N}nk} \quad \text{- обратное ПФ.}$$

Преобразование Фурье является разделимым, поэтому для анализа двумерных сигналов будут справедливы следующие соотношения:

$$X(k_1,k_2)=\sum_{n1=0}^{N_1-1}\sum_{n2=0}^{N_2-1}x(n_1,n_2)e^{-j\frac{2\pi}{N_1}k_1n_1-j\frac{2\pi}{N_2}k_2n_2} \quad \text{- прямое ПФ,}$$

$$x(n_1,n_2)=\frac{1}{N_1N_2}\sum_{k1=0}^{N_1-1}\sum_{k2=0}^{N_2-1}X(k_1,k_2)e^{j\frac{2\pi}{N_1}k_1n_1+j\frac{2\pi}{N_2}k_2n_2} \quad \text{- обратное ПФ.}$$

Результатом прямого ПФ является амплитудный спектр, который показывает амплитуду синусоиды частоты $\left(w_k=\frac{2\pi}{N_1}k_1, w_2=\frac{2\pi}{N_2}k_2\right)$. Можно заметить, что при $k_1=0$, $k_2=0$, комплексная синусоида превращается в плоскость, параллельную осям n_1 и n_2. Таким образом значение $X(0,0)$ - содержит среднее значение анализируемого сигнала. При увеличении k_1,k_2 происходит увеличении частоты базисных функций, которые выделяют детали изображения. Типичный амплитудный спектр изображения выглядит следующим образом.

Частным случаем ПФ является анализ симметричных сигналов относительно некоторой точки (i_1,i_2). В этом случае комплексная синусоида заменяется на функцию косинуса. Допустим, что симметричный массив образован путем зеркального отражения исходного массива относительно его краев согласно соотношению:

$$f_s(n_1,n_2)=\begin{cases} f(n_1,n_2) \text{ при } \mathrm{n}_1\ge 0, n_2\ge 0,\\ f(-1-j,k) \text{ при } \mathrm{n}_1<0, n_2\ge 0,\\ f(j,-1-k) \text{ при } \mathrm{n}_1\ge 0, n_2<0,\\ f(-1-j,-1-k) \text{ при } \mathrm{n}_1<0, n_2<0. \end{cases}$$

Построенный таким образом массив $f_s(n_1,n_2)$ симметричен относительно точки $n_1=-1/2, n_2=-1/2$. Преобразование Фурье для случая, когда начало координат находится в центре симметрии запишется как:

$$X_s(k_1,k_2)=\frac{1}{2N}\sum_{n1=-N}^{N-1}\sum_{n2=-N}^{N-1}f_s(n_1,n_2)\exp\left\{-\frac{2\pi i}{2N}\left[k_1\left(n_1+\frac{1}{2}\right)+k_2\left(n_2+\frac{1}{2}\right)\right]\right\}$$

где $k_1, k_2 = \overline{-N, N-1}$. Так как массив $f_s(n_1, n_2)$ симметричен и состоит из действительных чисел, соотношение переписать в виде

$$X(k_1, k_2) = \frac{2}{N} \sum_{n_1=0}^{N-1} \sum_{n_2=0}^{N-1} f(n_1, n_2) \cos\left[\frac{\pi}{N} k_1 \left(n_1 + \frac{1}{2}\right)\right] \cos\left[\frac{\pi}{N} k_2 \left(n_2 + \frac{1}{2}\right)\right].$$

Так как базисная функция при $k_1 = 0, k_2 = 0$ представляет собой постоянную составляющую и энергию в два раза большую по отношению ко всем остальным базисным функциям четного косинусного преобразования, то при выполнении обратного преобразования необходимо ввести нормирующий коэффициент $C(t)$, определяемый как

$$C(t) = \begin{cases} 1/2, t = 0 \\ 1, t > 0 \;\&\; t < 0 \end{cases}$$

Обратное преобразование имеет вид

$$f(n_1, n_2) = \frac{2}{N} \sum_{k_1=0}^{N-1} \sum_{k_2=0}^{N-1} C(k_1) C(k_2) F(k_1, k_2) \cos\left[\frac{\pi}{N} k_1 \left(n_1 + \frac{1}{2}\right)\right] \cos\left[\frac{\pi}{N} k_2 \left(n_2 + \frac{1}{2}\right)\right].$$

Четное ПФ имеет быстрый алгоритм вычисления. Благодаря этому широко используется на практике при анализе сигналов.

Принцип алгоритмов быстрых вычислений

В общем случае, чтобы выполнить унитарное преобразование матрицы изображения, содержащей $N \times N$ элементов, и получить матрицу из $N \times N$ спектральных коэффициентов, необходимо произвести примерно N^2 арифметических операций (умножений и сложений). Если размеры матрицы изображения велики, то число операций становится чрезвычайно большим. К счастью, для многих унитарных преобразований существуют эффективные алгоритмы вычислений, позволяющие ускорить выполнение преобразования.

Основной идеей этих быстрых алгоритмов является разделение всей задачи на ряд этапов, причем результаты, полученные на предыдущих этапах, многократно используются на последующих. В качестве примера рассмотрим процесс вычисления коэффициентов преобразования Адамара с неупорядоченной матрицей для последовательности из четырех элементов $f(j)$. При прямом способе вычисления находятся четыре величины по формулам:

$$F(0) = f(0) + f(1) + f(2) + f(3),$$
$$F(1) = f(0) - f(1) + f(2) - f(3),$$
$$F(2) = f(0) + f(1) - f(2) - f(3),$$
$$F(3) = f(0) - f(1) - f(2) + f(3).$$

Для этого необходимо выполнить $N(N-1)=12$ арифметических операций (сложений и вычитаний). Однако коэффициенты преобразования Адамара можно найти по-другому, разбив процесс вычисления на следующие этапы:

Первый этап

$$a(0) = f(0) + f(2),$$
$$a(1) = f(0) - f(2),$$
$$a(2) = f(1) + f(3),$$
$$a(3) = f(1) - f(3).$$

Второй этап

$$F(0) = a(0) + a(2),$$
$$F(1) = a(0) - a(2),$$
$$F(2) = a(1) + a(3),$$
$$F(3) = a(1) - a(3).$$

При этом для определения элементов матрицы H_4 требуется только $N\log_2 N = 8$ операций, т.е. экономится четыре операции.

Принцип описанный выше на примере преобразования Адамара, можно применить для быстрого вычисления многих других преобразований. Разработаны быстрые алгоритмы для преобразования Фурье, четного косинусного, Адамара. В общем случае для преобразования Карунена-Лоэва быстрого алгоритма не найдено, однако известны приближенные алгоритмы преобразования Карунена-Лоэва для марковских процессов.

Стандарт сжатия JPEG

Алгоритм разработан группой экспертов в области фотографии (Joint Photographic Expert Group) специально для сжатия 24-битных и полутоновых изображений в 1991 году. Этот алгоритм не очень хорошо сжимает двухуровневые изображении, но он прекрасно обрабатывает изображения с непрерывными тонами, в которых близкие пикселы обычно имеют схожие цвета. Обычно глаз не в состоянии заметить какой-либо разницы при сжатии этим методом в 10 или 20 раз.

Алгоритм основан на ДКП, применяемом к матрице непересекающихся блоков изображения, размером 8x8 пикселей. ДКП раскладывает эти блоки по амплитудам некоторых частот. В результате, получается матрица, в которой многие коэффициенты, как правило, близки к нулю, которые можно представить в грубой числовой форме, т.е. в квантованном виде без существенной потери в качестве восстановления.

Рассмотрим работу алгоритма подробнее. Предположим, что сжимается полноцветное 24-битное изображение. В этом случае получаем следующие этапы работы.

Шаг 1. Переводим изображение из пространства RGB в пространство YCbCr с помощью следующего выражения:

$$\begin{vmatrix} Y \\ Cb \\ Cr \end{vmatrix} = \begin{vmatrix} 0{,}299 & 0{,}587 & 0{,}114 \\ 0{,}5 & -0{,}4187 & -0{,}0813 \\ 0{,}1687 & -0{,}3313 & 0{,}5 \end{vmatrix} \begin{vmatrix} R \\ G \\ B \end{vmatrix} + \begin{vmatrix} 0 \\ 128 \\ 128 \end{vmatrix}.$$

Отметим сразу, что обратное преобразование легко получается путем умножения обратной матрицы на вектор $[Y, Cb, Cr]^T - [0{,}128{,}128]^T$, который по существу является пространством YUV:

$$\begin{vmatrix} R \\ G \\ B \end{vmatrix} = \begin{vmatrix} 1 & 0 & 1{,}402 \\ 1 & -0{,}34414 & -0{,}71414 \\ 1 & 1{,}772 & 0 \end{vmatrix} \left(\begin{vmatrix} Y \\ Cb \\ Cr \end{vmatrix} - \begin{vmatrix} 0 \\ 128 \\ 128 \end{vmatrix} \right).$$

Шаг 2. Разбиваем исходное изображение на матрицы 8x8. Формируем из каждой три рабочие матрицы ДКП – по 8 бит отдельно для каждой компоненты. При больших степенях сжатия блок 8x8 раскладывается на компоненты YCbCr в формате 4:2:0, т.е. компоненты для Cb и Cr берутся через точку по строкам и столбцам.

Шаг 3. Применение ДКП к блокам изображения 8x8 пикселей. Формально прямое ДКП для блока 8x8 можно записать в виде

$$Y(u,v) = \frac{1}{4} \sum_{i=0}^{7} \sum_{j=0}^{7} A(u) A(v) X(i,j) \cos\left(\frac{\pi(i+0{,}5)}{8} u \right) \cos\left(\frac{\pi(j+0{,}5)}{8} v \right),$$

где $A(x) = \begin{cases} \frac{1}{\sqrt{2}}, x = 0 \\ 1, x \neq 0 \end{cases}$. Так как ДКП является «сердцем» алгоритма JPEG, то желательно на практике вычислять его как можно быстрее. Простым подходом для ускорения вычислений является заблаговременное вычисление функций косинуса и сведения результатов вычисления в таблицу. Мало того, учитывая ортогональность функций косинусов с разными частотами, вышеприведенную формулу можно записать в виде

$$Y(u,v) = \underbrace{\frac{1}{4} \sum_{i=0}^{7} \underbrace{A(u) \cos\left(\frac{\pi(i+0{,}5)}{8} u \right)}_{C(i,u)}}_{C} X(i,j) \underbrace{\sum_{j=0}^{7} \underbrace{A(v) \cos\left(\frac{\pi(j+0{,}5)}{8} v \right)}_{C(j,v)}}_{C^T}.$$

Здесь C является матрицей, размером 8x8 элементов, описывающая 8-ми мерное пространство, для представления столбцов блока X в этом пространстве. Матрица C^T является транспонированной матрицей C и делает то же самое, но для строк блока X. В результате получается разделимое преобразование, которое в матричном виде записывается как

$$Y = CXC^T.$$

Здесь Y - результат ДКП, для вычисления которого требуется $2 \cdot 8 \cdot 64 = 2 \cdot 8^3 = 1024$ операций умножения и почти столько же сложений, что существенно меньше прямых вычислений по формуле выше. Например, для преобразования изображения размером 512x512 пикселей потребуется $64 \cdot 64 \cdot 1024 = 4096 \cdot 1024 = 4194304$ арифметических операций. Учитывая 3 яркостных компоненты, получаем значение 12 582 912 арифметических операций. Количество умножений и сложений можно еще больше сократить, если воспользоваться алгоритмом быстрого преобразования Фурье. В результате для преобразования одного блока 8x8 нужно будет сделать 54 умножений, 468 сложений и битовых сдвигов.

В результате ДКП получаем матрицу Y, в которой коэффициенты в левом верхнем углу соответствуют низкочастотной составляющей изображения, а в правом нижнем – высокочастотной.

Шаг 4. Квантование. На этом шаге происходит отбрасывание части информации. Здесь каждое число из матрицы Y делится на специальное число из «таблицы квантования», а результат округляется до ближайшего целого:

$$Y^q(u,v) = Round\left(\frac{Y(u,v)}{q(u,v)}\right).$$

Причем для каждой матрицы Y, Cb и Cr можно задавать свои таблицы квантования. Стандарт JPEG даже допускает использование собственных таблиц квантования, которые, однако, необходимо будет передавать декодеру вместе со сжатыми данными, что увеличит общий размер файла. Понятно, что пользователю сложно самостоятельно подобрать 64 коэффициента, поэтому стандарт JPEG использует два подхода для матриц квантования. Первый заключается в том, что в стандарт JPEG включены две рекомендуемые таблицы квантования: одна для яркости, вторая для цветности. Эти таблицы представлены ниже. Второй подход заключается в синтезе (вычислении на лету) таблицы квантовании, зависящей от одного параметра R, который задается пользователем. Сама таблица строится по формуле

$$Q(i,j) = 1 + (i+j)R.$$

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	58	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

светимость

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

цветность

Рекомендуемые таблицы квантования

На этапе квантования осуществляется управление степенью сжатия, и происходят самые большие потери. Понятно, что задавая таблицы квантования с большими коэффициентами, мы получим больше нулей и, следовательно, большую степень сжатия.

С квантованием связаны и специфические эффекты алгоритма. При больших значениях шага квантования потери могут быть настолько велики, что изображение распадется на квадраты однотонные 8x8. В свою очередь потери в высоких частотах могут проявиться в так называемом «эффекте Гиббса», когда вокруг контуров с резким переходом цвета образуется волнообразный «нимб».

Шаг 5. Переводим матрицу 8x8 в 64-элементный вектор при помощи «зигзаг»-сканирования (рис. 2).

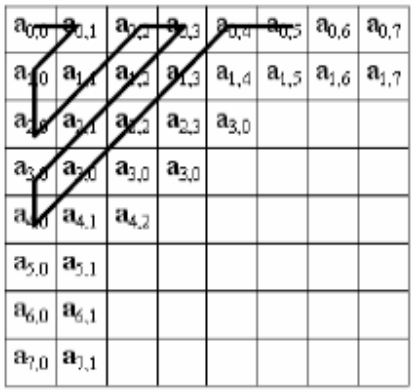


Рис. 2. «Зигзаг»-сканирование

В результате в начале вектора, как правило, будут записываться ненулевые коэффициенты, а в конце образовываться цепочки из нулей.

Шаг 6. Преобразовываем вектор с помощью модифицированного алгоритма RLE, на выходе которого получаем пары типа (пропустить, число), где «пропустить» является счетчиком пропускаемых нулей, а «число» - значение, которое необходимо поставить в следующую ячейку. Например, вектор 1118 3 0 0 0 -2 0 0 0 0 1 … будет свернут в пары (0, 1118) (0,3) (3,-2) (4,1) … .

Следует отметить, что первое число преобразованной компоненты Y, по существу, равно средней яркости блока 8x8 и носит название DC-коэффициента. Аналогично для всех блоков изображения. Это обстоятельство наводит на мысль, что коэффициенты DC можно эффективно сжать, если запоминать не их абсолютные значения, а относительные в виде разности между DC коэффициентом текущего блока и DC коэффициентом предыдущего блока, а первый коэффициент запомнить так, как он есть. При этом упорядочение коэффициентов DC можно сделать, например, так (рис. 3). Остальные коэффициенты, которые называются AC-коэффициентами сохраняются без изменений.

Шаг 7. Свертываем получившиеся пары с помощью неравномерных кодов Хаффмана с фиксированной таблицей. Причем для DC и AC коэффициентов используются разные коды, т.е. разные таблицы с кодами Хаффмана.

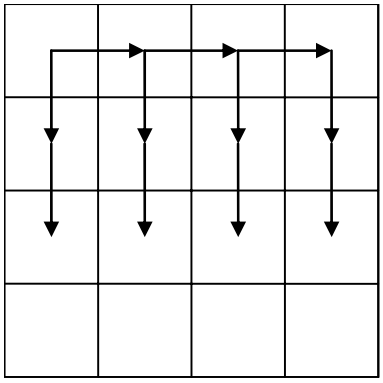


Рис. 3. Схема упорядочения DC коэффициентов

Рис. 4. Структурная схема алгоритма JPEG

Процесс восстановления изображения в этом алгоритме полностью симметричен. Метод позволяет сжимать изображения в 10-15 раз без заметных визуальных потерь.

При разработке данного стандарта руководствовались тем, что данный алгоритм должен был сжимать изображения довольно быстро – не более минуты на среднем изображении. Это в 1991 году! А его аппаратная реализация должна быть относительно простой и дешевой. При этом алгоритм должен был быть симметричным по времени работы. Выполнение последнего требования сделало возможным появление цифровых фотоаппаратов, снимающие 24 битные изображения. Если бы алгоритм был несимметричен, было бы неприятно долго ждать, пока аппарат «перезарядится» - сожмет изображение.

Хотя алгоритм JPEG и является стандартом ISO, формат его файлов не был зафиксирован. Пользуясь этим, производители создают свои, несовместимые между собой форматы, и, следовательно, могут изменить алгоритм. Так, внутренние таблицы алгоритма, рекомендованные ISO, заменяются ими на свои собственные. Встречаются также варианты JPEG для специфических приложений.

Квантование

В стандарте JPEG одним из этапов является квантование коэффициентов после ДКП и здесь следует сказать несколько слов о квантовании вообще. Цель

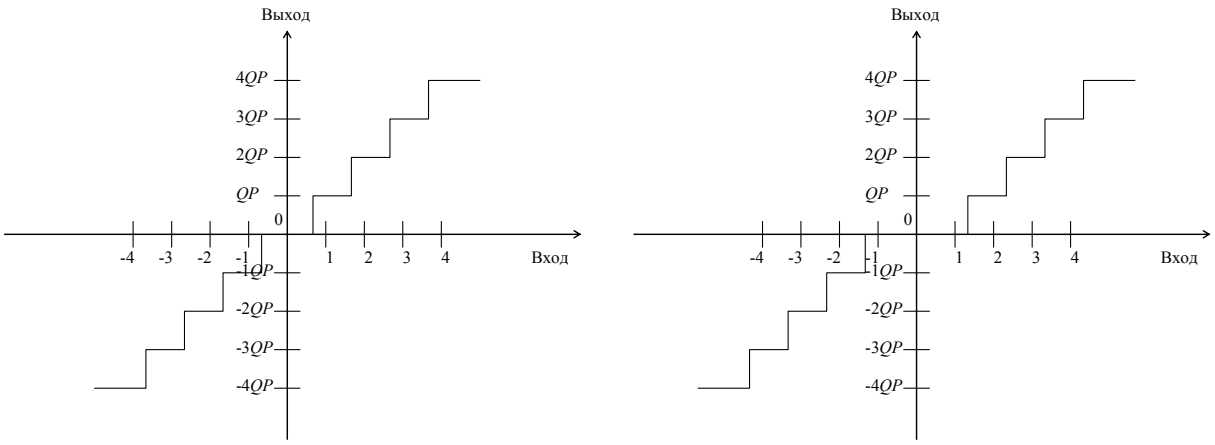
квантования состоит в отображении числового сигнала с областью значений X в квантованный сигнал области Y с уменьшенным числом значений. Это дает возможность представить квантованные величины с меньшим числом бит по сравнению с исходным. При этом известно два вида квантования: скалярное и векторное. Скалярный квантователь отображает одно входное число в одно квантованное значение на выходе, а векторный квантователь отображает группу чисел на входе (вектор) в группу квантованных величин.

Скалярное квантование

Простейшим примером скалярного квантования может служить метод округления дробного числа до ближайшего целого, т.е. отображения из R в Z. Это процесс с частичной потерей информации (он необратим). Более общий метод равномерного квантователя можно представить в виде формулы:

$$FQ = round\left(\frac{X}{QP}\right),\ Y = FQ \cdot QP,$$

где QP - это шаг квантования. То есть уровни квантованных выходов расположены на одинаковых расстояниях QP друг от друга (рис. 4, а). Однако более лучшие результаты, с точки зрения сокращения энтропии выходных данных при минимальных потерях, дает квантователь с мертвой зоной в нуле (рис. 4, б).



а) б)

Рис. 4

Векторное квантование

Векторный квантователь отображает множество входных данных в один элемент (кодовое слово), а декодер каждому кодовому слову сопоставляет некоторое приближение к исходному множеству данных (вектору). Множество векторов

хранится кодером и декодером в специальной кодовой книге. Типичная схема применения векторного квантования при сжатии изображений состоит в следующем:

1. Разделить исходное изображение на области (например, блоки пикселов размером MxN).
2. Выбрать в кодовой книге наиболее близкий вектор к текущей области.
3. Переслать декодеру индекс, идентифицирующий выбранный вектор.
4. На стороне декодера реконструировать область, используя выбранный вектор.

Ключевым моментом при разработке векторного квантователя является построение кодовой книги и алгоритма быстрого поиска в ней наилучшего вектора.

Основы вейвлет-преобразования

Известно, что произвольный сигнал $S(t)$, для которого выполняется условие $\int_{t_1}^{t_2}\left[S(t)\right]^2 dt < \infty$ может быть представлен ортогональной системой функций $\{\varphi_n(t)\}$:

$$S(t) = C_0\varphi_o(t) + ... + C_n\varphi_n(t) = \sum_{n=0}^{\infty} C_n\varphi_n(t), \qquad (18)$$

коэффициенты определяются из соотношения

$$C_n = \frac{1}{\left\|\varphi_n\right\|^2}\int_{t_1}^{t_2} S(t)\varphi_n(t)dt,$$

где $\left\|\varphi_n\right\|^2 = \int_{t_1}^{t_2}\varphi_n^2(t)dt$ - квадрат нормы или энергия базисной функции $\varphi_n(t)$. Ряд (18) называется обобщенным рядом Фурье. При этом произведения вида $C_n\varphi_n(t)$, входящие в ряд (18), представляют собой спектральную плотность сигнала $S(t)$, а коэффициенты C_n - спектр сигнала. Суть спектрального анализа сигнала $S(t)$ состоит в определении коэффициентов C_n. Зная эти коэффициенты возможен синтез (аппроксимация) сигналов при фиксированном числе N ряда:

$$\widetilde{S}(t) = C_0\varphi_0(t) + ... + C_N\varphi_N(t) = \sum_{n=0}^{N} C_n\varphi_n(t).$$

Обобщенный ряд Фурье при заданной системе базисных функций $\{\varphi_n(t)\}$ и числе слагаемых N он обеспечивает наилучший синтез по критерию минимума среднеквадратической ошибки ε, под которой понимается величина

$$\varepsilon = \int_{t_1}^{t_2}\left[S(t) - \widetilde{S}(t)\right]^2 dt.$$

Известные преобразования (Адамара, Карунена-Лоэва, Фурье) «плохо» представляют нестационарный сигнал в коэффициентах разложения. Покажем это на следующем примере. Пусть дана нестационарная функция

$$f(x) = \sin(x) + 0.1 \cdot sign(\sin(x - 0.005)) \qquad (19)$$

и ее преобразование Фурье (рис. 9).

Анализ рис. 9 показывает, что нестационарность временного сигнала $f(x)$ представляется большим числом высокочастотных коэффициентов отличных от нуля. При этом возникают следующие проблемы:

- сложно провести анализ временного сигнала по его Фурье образу;
- приемлемая аппроксимация временного сигнала возможна при учете большого числа высокочастотных коэффициентов;
- плохое визуальное качество реальных изображений восстановленных по низкочастотным коэффициентам; и т.п.

Существующие проблемы обусловили необходимость разработки математического аппарата преобразования нестационарных сигналов. Одним из возможных путей анализа таких сигналов стало вейвлет-преобразование (ВП).

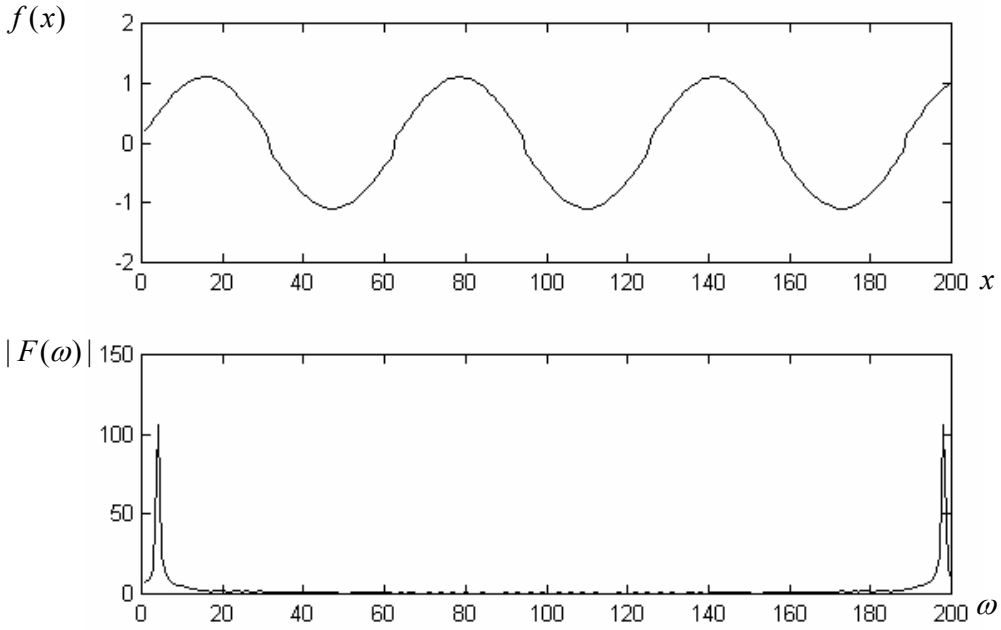


Рис. 9. Преобразование Фурье синусоидального сигнала с небольшими ступеньками при переходе через нуль

ВП одномерного сигнала – это его представление в виде обобщенного ряда Фурье или интеграла Фурье по системе базисных функций локализованных как в пространственной, так и в частотной областях. Примером такой базисной функции может служить вейвлет Хаара, который определяется выражением

$$\varphi(t) = \begin{cases} 1, \text{при } 0 \le t \le 1/2, \\ -1, \text{при } 1/2 \le t \le 1, \\ 0, \text{при } t < 0, t > 1. \end{cases} \qquad (20)$$

Графически вейвлет Хаара представляется следующим образом:

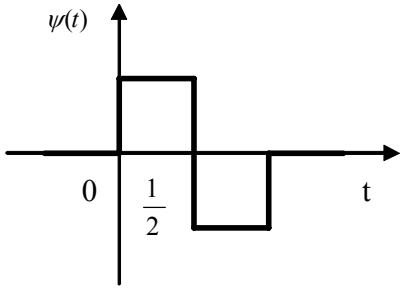


Рис. 10. Базисная функция вейвлета Хаара

Рассмотрим процесс разложения сигнала $S(t)$ в системе базисных функций Хаара. Первая базисная функция, в отличие от всех последующих, представляет собой прямую линию. В случае нормированного базиса $\{\varphi_n(t)\}$, свертка первой базисной функции с исходным сигналом будет определять его среднее значение. Пусть дан дискретный сигнал $S(t)$ длиной N отсчетов. Нормированная базисная функция на интервале $[0, N-1]$ описывается выражением $\varphi_0(n) = 1/N$. Тогда свертка данной функции с сигналом $S(t)$ приводит к выражению

$$C_0 = \sum_{n=0}^{N-1} S(n)\varphi_0(n) = \sum_{n=0}^{N-1} S(n) \cdot \frac{1}{N} = \frac{1}{N}\sum_{n=0}^{N-1} S(n).$$

Если выполнить синтез сигнала $S(t)$ по коэффициенту C_0 с помощью синтезирующей функции $\tilde{\varphi}_0(n) = 1$, получим постоянную составляющую, соответствующую среднему значению сигнала. Для того чтобы иметь возможность более детально описать сигнал, вычислим второй коэффициент с помощью базисной функции, представленной выражением (20):

$$C_1 = \sum_{n=0}^{N-1} S(n)\varphi_1(n) = \sum_{n=0}^{N/2-1} S(n) \cdot \frac{1}{N} + \sum_{n=N/2}^{N-1} S(n) \cdot \left(-\frac{1}{N}\right) = \frac{1}{N}\sum_{n=0}^{N/2-1} S(n) - \frac{1}{N}\sum_{n=N/2}^{N-1} S(n) =$$

$$= \frac{\frac{1}{N/2}\sum_{n=0}^{N/2-1} S(n) - \frac{1}{N/2}\sum_{n=N/2}^{N-1} S(n)}{2}.$$

Анализ данного выражения показывает, что коэффициент C_1 характеризует разности средних значений половинок сигнала $S(t)$. Если теперь выполнить синтез по двум коэффициентам с синтезирующей базисной функцией для второго коэффициента

$$\tilde{\varphi}_1(n) = \begin{cases} 1, 0 \le n < N/2, \\ -1, N/2 \le n \le N-1. \end{cases}$$

получим следующую аппроксимацию:

$$\widetilde{S}(n) = \begin{cases} C_0 + C_1, 0 \le n < N/2 \\ C_0 - C_1, N/2 \le n < N. \end{cases}$$

Дальнейшая операция анализа, т.е вычисления коэффициентов $C_2, C_3, ..., C_{N-1}$ и синтеза аналогична рассмотренной, с той разницей, что все действия повторяются для половинок сигнала, затем для четверти, и т.д. На самой последней итерации анализ осуществляется для пар случайных величин (рис 11).

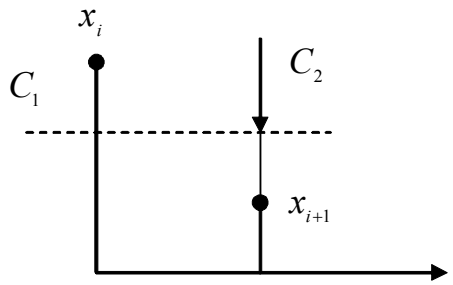


Рис. 11. Преобразование пар случайных величин

В результате исходный сигнал точно описывается коэффициентами вейвлет-преобразования Хаара. Вейвлет-коэффициенты сигнала (19) показаны на рис. 10.

Из приведенного рисунка видно, что нестационарности сигнала (резкие перепады) локализуются в малом числе вейвлет-коэффициентов. Это приводит к возможности лучшего восстановления нестационарного сигнала по неполным данным.

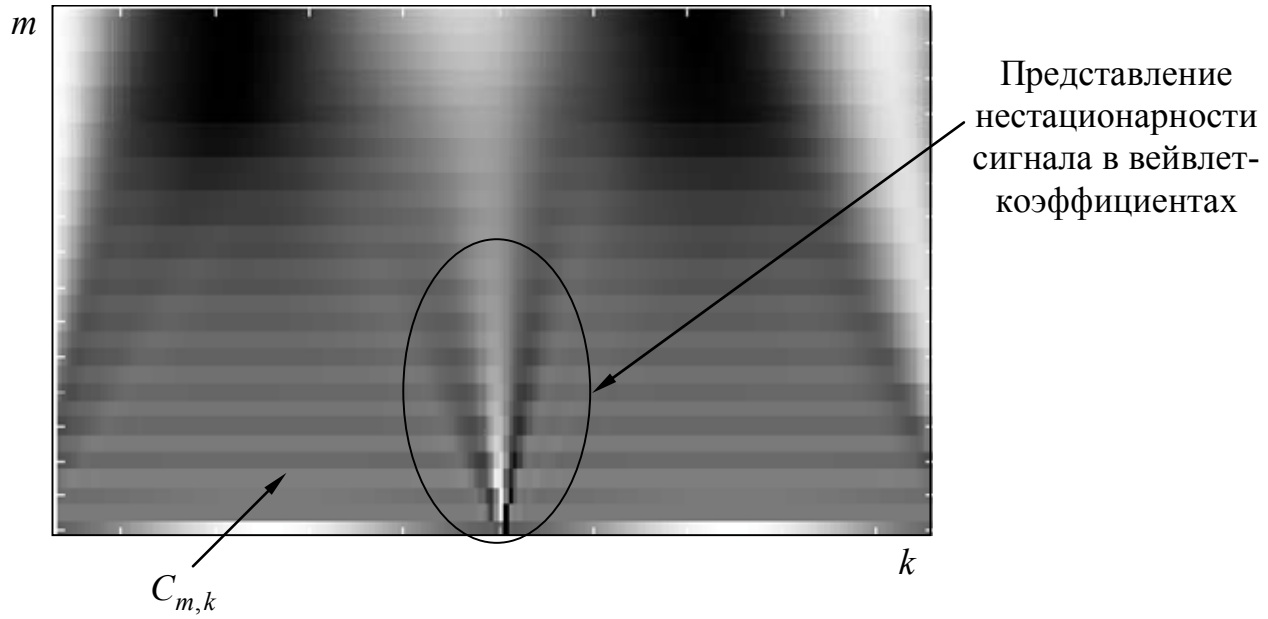


Рис. 12. Вейвлет-коэффициенты одного периода функции (19)

При вычислении вейвлет-коэффициентов $C_{m,k}$ базисные функции покрывали анализируемый сигнал следующим образом (рис. 12). Из рис. 12 видно, что система

базисных функций Хаара в дискретном пространстве должна задаваться двумя параметрами: сдвига и частоты (масштаба):

$$\varphi_{ab}(t) = \frac{1}{\sqrt{a}}\varphi\left(\frac{t-b}{a}\right),$$

где a - масштаб базисной функции; b - сдвиг. В дискретном случае параметр масштаба $a = 2^m$, где m - любое целое положительное число, параметр сдвига $b = k2^m$. Таким образом, все множество базисных функций можно записать как

$$\varphi_{mk}(n) = \frac{1}{\sqrt{2^m}}\varphi\left(2^{-m}n - k\right).$$

Прямое и обратное дискретные ВП вычисляются по формулам

$$C_{m,k} = \left\langle S(n)\varphi_{mk}(n)\right\rangle = \sum_{n=0}^{N-1} S(n)\varphi_{mk}(n),$$

$$S(n) = \sum_m \sum_k C_{m,k}\varphi_{mk}(n).$$

Следует отметить, что если число отсчетов $N = 2^n$, то максимальное значение m равно $n-1$. Наибольшее значение k для текущего m равно $2^{n-m}-1$.

Для непрерывных сигналов будут справедливы следующие интегральные выражения:

$$\widetilde{S}(t) = \frac{1}{C_\varphi}\int_a\int_b C(a,b)\varphi_{ab}(t)\frac{dadb}{a^2}. \qquad C(a,b) = \frac{1}{\sqrt{a}}\int_{t_1}^{t_2} S(t)\varphi_{ab}(t)dt = \frac{1}{\sqrt{a}}\int S(t)\varphi\left(\frac{t-b}{a}\right)dt,$$

Таким образом, задавая вейвлет-функции, можно выполнять разложение сигнала по вейвлет-базису непрерывных или дискретных сигналов.

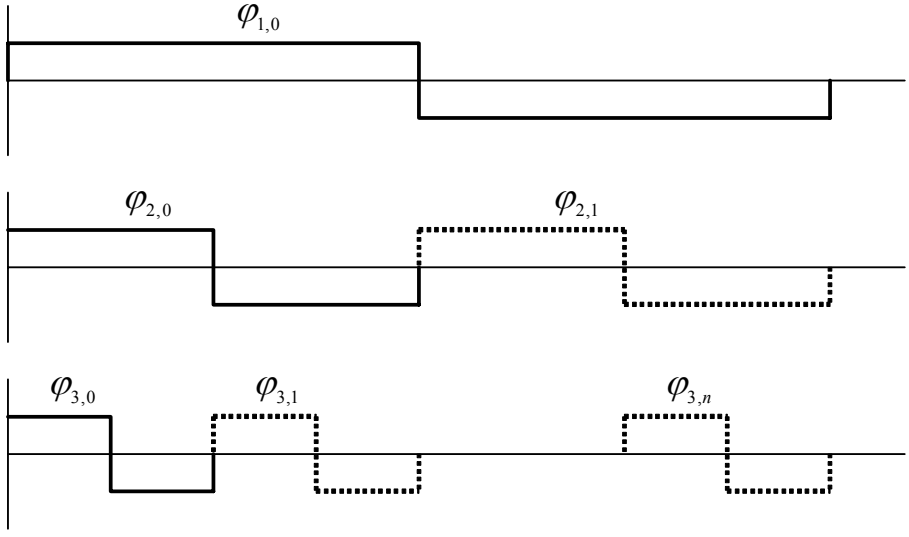


Рис. 13. Распределение базисных функций Хаара при анализе сигнала

Функция $\varphi(t)$ может образовывать вейвлет-базис, если она удовлетворяет следующим условиям:

1. Ограниченность нормы:

$$\|\varphi\|^2 = \int_{-\infty}^{\infty} |\varphi(t)|^2 \, dt < \infty .$$

2. Вейвлет-функция должна быть ограничена и по времени и по частоте:

Контрпример: дельта-функция и гармоническая функция не удовлетворяют данному условию. $|\varphi(t)| \le C(1+|t|)^{-1-e}$ **и** $|S_\varphi(w)| \le C(1+|w|)^{-1-e}$**, при** $e>0$**.**

3. Нулевое среднее:

Если обобщить данное условие, то можно получить формулу $\int_{-\infty}^{\infty} t^n \varphi(t) dt = 0$**,** которая определяет степень гладкости функции $\varphi(t)$. $\int_{-\infty}^{\infty} \varphi(t) dt = 0$**.**

Считается, что чем выше степень гладкости базисной функции, тем лучше ее аппроксимационные свойства.

В качестве примера приведем следующие известные вейвлет-функции:

$$\varphi(t) = -te^{-\frac{t^2}{2}}, \quad \varphi(t) = (1-t)e^{-\frac{t^2}{2}}.$$

Для ВП, также как и для ДПФ существует алгоритм быстрого преобразования. Рассмотрим снова ВП Хаара. Из рис. 13 видно, что функции с малым масштабным коэффициентом a используют те же отсчеты сигнала для вычисления коэффициентов, что и функции с большим масштабным коэффициентом. При этом операция суммирования одних и тех же отсчетов повторяется неоднократно. Следовательно, для уменьшения объема вычислений целесообразно вычислять ВП с самого малого масштабного коэффициента. В результате получаем вейвлет-коэффициенты, представляющие собой средние значения $C_{1,k} = (x_i + x_{i+1})/2$ и разности $C_{1,j} = (x_i - x_{i+1})/2$. Для коэффициентов $C_{1,k} = (x_i + x_{i+1})/2$ повторяем данную процедуру. При этом усреднение коэффициентов $(C_{1,k} + C_{1,k+1})/2$ будет соответствовать усреднению четырех отсчетов сигнала, но при этом расходуется одна операция умножения и одна операция сложения. Процесс разложения повторяется до тех пор, пока не будут вычислены все коэффициенты спектра $C_{m,k}$.

Запишем алгоритм быстрого вейвлет-преобразования Хаара в матричном виде. Пусть дан вектор $X = [x_1, x_2, .., x_8]^T$ размером 8 элементов. Матрица преобразования Хаара запишется в виде

$$A_8 = \begin{vmatrix} 0.5 & 0.5 & & & & & & \\ & & 0.5 & 0.5 & & & & \\ & & & & 0.5 & 0.5 & & \\ & & & & & & 0.5 & 0.5 \\ 0.5 & -0.5 & & & & & & \\ & & 0.5 & -0.5 & & & & \\ & & & & 0.5 & -0.5 & & \\ & & & & & & 0.5 & -0.5 \end{vmatrix},$$

В приведенных обозначениях один шаг быстрого ВП запишется как

$$\begin{vmatrix} c_{11} \\ c_{12} \\ c_{13} \\ c_{14} \\ d_{11} \\ d_{12} \\ d_{13} \\ d_{14} \end{vmatrix} = A_8 X = A_8 \begin{vmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{vmatrix}. \qquad (21)$$

Повторяем операцию преобразования для коэффициентов $C_1 = [c_{11}, c_{12}, c_{13,} c_{14}]^T$:

$$\begin{vmatrix} c_{21} \\ c_{22} \\ d_{21} \\ d_{22} \end{vmatrix} = A_4 C_1 = A_4 \begin{vmatrix} c_{11} \\ c_{12} \\ c_{13} \\ c_{14} \end{vmatrix}, \qquad (22)$$

для коэффициентов $C_2 = [c_{21}, c_{22}]^T$

$$\begin{vmatrix} c_{31} \\ d_{31} \end{vmatrix} = A_2 C_2 = A_2 \begin{vmatrix} c_{21} \\ c_{22} \end{vmatrix}. \qquad (23)$$

В результате получается набор вейвлет-коэффициентов $c_{31}, d_{31,} d_{21}, d_{22}, d_{11}, d_{12}, d_{13}, d_{14}$, по которым можно точно восстановить исходный сигнал. Для этого необходимо задать матрицу синтеза $\widetilde{A}$

$$\widetilde{A}_4 = \begin{vmatrix} 1 & & 1 & \\ 1 & & -1 & \\ & 1 & & 1 \\ & 1 & & -1 \end{vmatrix}, \; \widetilde{A}_2 = \begin{vmatrix} 1 & 1 \\ 1 & -1 \end{vmatrix}.$$

Восстановленный сигнал запишется в виде

$$\tilde{S} = \tilde{A}_8 \begin{vmatrix} \tilde{A}_4 & \begin{vmatrix} \tilde{A}_2 & \begin{vmatrix} c_{31} \\ d_{31} \end{vmatrix} \\ d_{21} & \\ d_{22} & \end{vmatrix} \\ d_{11} & \\ d_{12} & \\ d_{13} & \\ d_{14} & \end{vmatrix}. \qquad (24)$$

В общем случае вместо коэффициентов 0.5,–0.5 могут быть любые другие, которые описывают соответствующее ВП. При этом элементы в матрицах $A, \tilde{A}$ можно характеризовать как коэффициенты низко- и высокочастотных фильтров анализа и синтеза соответственно:

$$H_8 = \begin{vmatrix} h_0 & h_1 & h_2 & h_3 & & & & \\ & & h_0 & h_1 & h_2 & h_3 & & \\ & & & & h_0 & h_1 & h_2 & h_3 \\ h_2 & h_3 & & & & & h_0 & h_1 \end{vmatrix},$$

$$G_8 = \begin{vmatrix} g_0 & g_1 & g_2 & g_3 & & & & \\ & & g_0 & g_1 & g_2 & g_3 & & \\ & & & & g_0 & g_1 & g_2 & g_3 \\ g_2 & g_3 & & & & & g_0 & g_1 \end{vmatrix},$$

где H_8 - матрица выделения низкочастотных составляющих; G_8 - матрица выделения высокочастотных составляющих. При этом преобразование сигнала можно представить через свертку КИХ-фильтров (рис. 14).

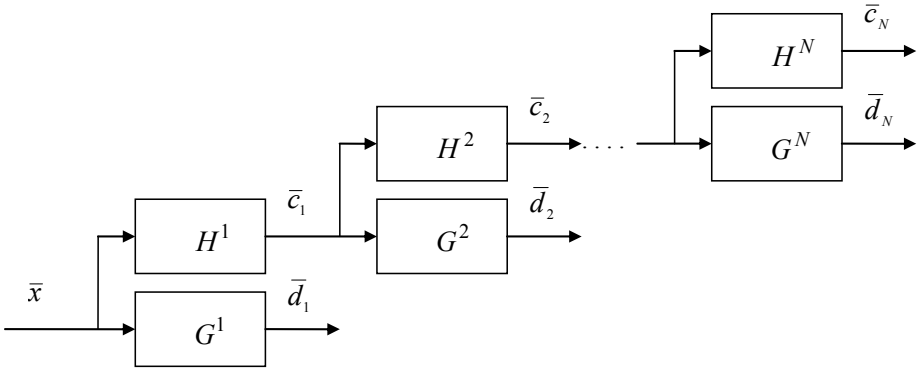


Рис. 14. Представление ВП через набор низко- и высокочастотных фильтров

Обобщение ВП на двумерный случай приводит к разделимому преобразованию:

$$C_{11} = H_8 X H_8^T; D_{11} = H_8 X G_8^T; D_{12} = G_8 X H_8^T; D_{13} = G_8 X G_8^T,$$

где C_{11} - низкочастотная составляющая; D_{11}, D_{12}, D_{13} - высокочастотные составляющие. Таким образом, при двумерном ВП изображение разбивается на четыре компоненты: три высокочастотные, представляющие мелкие детали, и одна низкочастотная, представляющая собой уменьшенную и сглаженную копию исходного изображения. В соответствии с алгоритмом БВП второй шаг ВП запишется в виде

$$C_{21} = H_4 C_{11} H_4^T; D_{21} = H_4 C_{11} G_4^T; D_{22} = G_4 C_{11} H_4^T; D_{23} = G_4 C_{11} G_4^T.$$

В результате получаем представление изображения на разных уровнях масштаба C_{11}, C_{21}, и т.д. Операции преобразования сигнала можно рекуррентно выполнять, пока низкочастотная составляющая не будет представлена одним отсчетом.

Расчет ортогональных вейвлет-фильтров

Рассмотрим матричный способ расчета вейвлет-фильтров для преобразования сигнала. Допустим, что низко- и высокочастотные коэффициенты фильтров ортогональны друг другу. В этом случае можно записать

$$g_i = (-1)^i h_{N-i}, \; i = \overline{1, N}. \qquad (25)$$

где $\overline{h}$ – коэффициенты низкочастотного фильтра; $\overline{g}$ – высокочастотного. Запишем матрицу преобразования A размером 8x8 элементов, при длине вейвлет-фильтров $N = 4$:

$$A = \begin{vmatrix} h_0 & h_1 & h_2 & h_3 & & & & \\ & & h_0 & h_1 & h_2 & h_3 & & \\ & & & & h_0 & h_1 & h_2 & h_3 \\ h_2 & h_3 & & & & & h_0 & h_1 \\ g_0 & g_1 & g_2 & g_3 & & & & \\ & & g_0 & g_1 & g_2 & g_3 & & \\ & & & & g_0 & g_1 & g_2 & g_3 \\ g_2 & g_3 & & & & & g_0 & g_1 \end{vmatrix}.$$

В соответствии с третьим условием базисных вейвлет-функций потребуем наличия 2-х нулевых моментов:

$$0^0 g_0 + 1^0 g_1 + 2^0 g_2 + 3^0 g_3 = 0,$$
$$0 g_0 + 1 g_1 + 2 g_2 + 3 g_3 = 0.$$

Для того чтобы преобразование было обратимым, необходимо выполнение условия $AA^T = I$, где I – единичная матрица. В результате получаем еще два уравнения:

$$h_0^2 + h_1^2 + h_2^2 + h_3^2 = 1,$$

$$h_0 h_2 + h_1 h_3 = 0.$$

Решение полученной системы уравнений даст коэффициенты низкочастотного фильтра:

$$h_0 = (1+\sqrt{3})/8, h_1 = (3+\sqrt{3})/8,$$

$$h_2 = (3-\sqrt{3})/8, h_3 = (1-\sqrt{3})/8.$$

Коэффициенты высокочастотного фильтра находятся из уравнения (25).

Вычисленные ортогональные вейвлеты получили название вейвлетов Добеши. Другим видом ВП является биортогональное преобразование. В этом случае базисные функции анализа не ортогональны друг другу. При этом удается получить вейвлет-фильтры с большим числом нулевых моментов.

Выполнение вейвлет-преобразования на основе лифтинговой схемы

Любое ВП можно эффективно выполнить с помощью лифтинговой схемы. Пусть задан вектор $X = [x_1, x_2, ..., x_N]^T$. Рассмотрим его разложение по базисным функциям Хаара. Первый высокочастотный коэффициент

$$\gamma_{-1,1} = 2d_{1,1} = x_2 - x_1, \tag{26}$$

низкочастотный

$$\lambda_{-1,1} = c_{1,1} = \frac{x_1 + x_2}{2} = x_1 + \frac{\gamma_{-1,1}}{2}. \tag{27}$$

Анализ выражений (26) и (27) показывает, что величину $\gamma_{-1,1}$ можно интерпретировать как ошибку оценивания элемента x_2. Величина $\lambda_{-1,1}$ представляет высокочастотные составляющие и вычисляется путем прибавления вейвлет-коэффициента $\gamma_{-1,1}/2$ к элементу x_1. Таким образом, вектор X можно разбить на оцениваемые элементы $x_2, x_4, ...$ и наблюдения $x_1, x_3, ...$ Перепишем выражения (26) и (27) в виде

$$\gamma_{-1,i} = x_{2i} - \frac{1}{2}\left(x_{2i-1} + x_{2i+1}\right), \tag{28}$$

$$\lambda_{-1,i} = x_{2i+1} + \frac{\gamma_{-1,i}}{2} = \frac{3}{4}x_{2i+1} + \frac{1}{2}x_{2i} - \frac{1}{4}x_{2i+1}.$$

При этом коэффициенты низкочастотного вейвлет-фильтра $h = [3/4, 1/2, -1/4]^T$. Лучших результатов сглаживания элементов $\overline{\lambda}$ можно получить если положить

$$\lambda_{-1,i} = x_{2i-1} + \frac{1}{4}\gamma_{-1,i-1} + \frac{1}{4}\gamma_{-1,i} = -\frac{1}{8}x_{2i-3} + \frac{1}{4}x_{2i-2} + \frac{3}{4}x_{2i-1} + \frac{1}{4}x_{2i} - \frac{1}{8}x_{2i+1}. \tag{29}$$

В этом случае коэффициенты низкочастотного фильтра $\bar{h}=[-1/8;1/4;3/4;1/4;-1/8]^T$ и высокочастотного $\bar{g}=[-1/2;1;-1/2]^T$. Если коэффициенты $\bar{h}$ и $\bar{g}$ записать в матрицу преобразования A, то можно вычислять данное ВП с помощью выражений (21)-(24). При этом матрица обратного преобразования $\tilde{A}=A^{-1}$.

Рассмотрим этап восстановления сигнала по вейвлет-коэффициентам $\bar{\gamma}_{-1}$ и низкочастотной составляющей $\bar{\lambda}_{-1}$. Также будем полагать, что $\bar{h}=[-1/8;1/4;3/4;1/4;-1/8]^T$ и $\bar{g}=[-1/2;1;-1/2]^T$. Из выражений (28) и (29) следует, что

$$x_{2i-1}=\lambda_{-1,i}-\frac{1}{4}\gamma_{-1,i-1}-\frac{1}{4}\gamma_{-1,i},$$

$$x_{2i}=\frac{1}{2}\left(x_{2i-1}+x_{2i+1}\right)+\gamma_{-1,i},$$

при $i=\overline{1,N/2}$. Объединение восстановленных четных и нечетных элементов даст исходный вектор X.

При анализе двумерных сигналов на основе лифтинговой схемы выполняется разделимое преобразование. Пусть имеется изображение X, размером $N\times N$ элементов. Требуется выполнить ВП с коэффициентами вейвлет-фильтров $\bar{h}=[-1/8;1/4;3/4;1/4;-1/8]^T$ и $\bar{g}=[-1/2;1;-1/2]^T$. Тогда для каждой строки $\bar{x}_i=[x_{i,1},x_{i,2},...,x_{i,N}]$ можно вычислить вейвлет-коэффициенты согласно выражениям (28) и (29):

$$\gamma_{-1,ij}=x_{i,2j}-\frac{1}{2}\left(x_{i,2j-1}+x_{i,2j+1}\right),$$

$$\lambda_{-1,ij}=x_{i,2j+1}+\frac{1}{4}\gamma_{-1,i,j-1}+\frac{1}{4}\gamma_{-1,i,j}, \qquad (30)$$

при $i=\overline{1,N}$, $j=\overline{1,N/2}$. Таким образом, получили две матрицы $\bar{\gamma}_{-1}$ и $\bar{\lambda}_{-1}$ размерностью $N\times 0.5N$. Так как преобразование является разделимым, то для вычисленных $\bar{\gamma}_{-1}$ и $\bar{\lambda}_{-1}$ выполняются аналогичные операции применительно к столбцам:

$$\gamma^{(1)}_{-1,ij}=\gamma_{-1,2i,j}-\frac{1}{2}\left(\gamma_{-1,2i-1,j}+\gamma_{-1,2i+1,j}\right),$$

$$\gamma^{(2)}_{-1,ij}=\gamma_{-1,2i+1,j}+\frac{1}{4}\gamma^{(1)}_{-1,i-1,j}+\frac{1}{4}\gamma^{(1)}_{-1,i,j},$$

$$\gamma^{(3)}_{-1,ij}=\lambda_{-1,2i,j}-\frac{1}{2}\left(\lambda_{-1,2i-1,j}+\lambda_{-1,2i+1,j}\right), \qquad (31)$$

$$\lambda^{(1)}_{-1,ij}=\lambda_{-1,2i+1,j}+\frac{1}{4}\gamma^{(3)}_{-1,i-1,j}+\frac{1}{4}\gamma^{(3)}_{-1,i,j},$$

при $i, j = \overline{1, N/2}$. В результате получаем четыре матрицы размером $0{,}5N \times 0{,}5N$. Здесь $\bar{\gamma}_{-1}^{(1)}$, $\bar{\gamma}_{-1}^{(2)}$, $\bar{\gamma}_{-1}^{(3)}$ представляют собой высокочастотные составляющие (детали изображения), а $\bar{\lambda}_{-1}^{(1)}$ - низкочастотную, представляющая уменьшенную в четыре раза и сглаженную копию исходного изображения. Выражения (30), (31) можно рекуррентно повторять для низкочастотных составляющих $\bar{\lambda}_{-1}^{(k)}$. На практике обычно выполняют 4-5 итераций.

Рассмотрим алгоритм обратного ВП. В соответствии с выражением (31) имеем:

$$\begin{aligned}
\lambda_{-1,2i+1,j} &= \lambda_{-1,ij}^{(1)} - \frac{1}{4}\gamma_{-1,i-1,j}^{(3)} - \frac{1}{4}\gamma_{-1,i,j}^{(3)}, \\
\lambda_{-1,2i,j} &= \gamma_{-1,ij}^{(3)} + \frac{1}{2}\left(\lambda_{-1,2i-1,j} + \lambda_{-1,2i+1,j}\right), \\
\gamma_{-1,2i+1,j} &= \gamma_{-1,ij}^{(2)} - \frac{1}{4}\gamma_{-1,i-1,j}^{(1)} - \frac{1}{4}\gamma_{-1,i,j}^{(1)}, \\
\gamma_{-1,2i,j} &= \gamma_{-1,ij}^{(1)} + \frac{1}{2}\left(\gamma_{-1,2i-1,j} + \gamma_{-1,2i+1,j}\right),
\end{aligned} \tag{32}$$

при $i, j = \overline{1, N/2}$. Объединение четных и нечетных строк даст матрицы $\bar{\lambda}_{-1}$ и $\bar{\gamma}_{-1}$ размером $N \times 0.5N$ элементов. Окончательно из выражения (30) имеем:

$$\begin{aligned}
x_{i,2j+1} &= \lambda_{-1,ij} - \frac{1}{4}\gamma_{-1,i,j-1} - \frac{1}{4}\gamma_{-1,i,j}, \\
x_{i,2j} &= \gamma_{-1,ij} + \frac{1}{2}\left(x_{i,2j-1} + x_{i,2j+1}\right),
\end{aligned} \tag{33}$$

при $i = \overline{1, N}$, $j = \overline{1, N/2}$. Выражения (32) и (33) определяют обратное двумерное ВП на основе лифтинговой схемы с $\bar{h} = [-1/8; 1/4; 3/4; 1/4; -1/8]^T$ и $\bar{g} = [-1/2; 1; -1/2]^T$. В общем случае для произвольных $\bar{h}$ и $\bar{g}$ прямое ВП на основе лифтинговой схемы можно записать в виде

$$\begin{cases}
\{\bar{\gamma}_j, \bar{\lambda}_j\} = S(X), \\
\bar{\gamma}_j = \bar{\gamma}_j - P(\bar{\lambda}_j), \\
\bar{\lambda}_j = \bar{\lambda}_j + U(\bar{\gamma}_j),
\end{cases} \tag{34}$$

где $S(\bullet)$ - оператор разбиения последовательности X на наблюдения $\bar{\lambda}_j$ и оцениваемые элементы $\bar{\gamma}_j$; $P(\bullet)$ - оператор оценивания элементов $\bar{\gamma}_j$ на основе наблюдений $\bar{\lambda}_j$; $U(\bullet)$ - оператор обновления. Схема обратного преобразования будет иметь вид

$$\begin{cases} \overline{\lambda}_j = \overline{\lambda}_j - U(\overline{\gamma}_j), \\ \overline{\gamma}_j = \overline{\gamma}_j + P(\overline{\lambda}_j), \\ X = \overline{\lambda}_j \cup \overline{\gamma}_j. \end{cases} \quad (35)$$

Выражения (34) и (35) описывают один шаг ВП на основе лифтинговой схемы.

Стандарт JPEG2000

Стандарт JPEG2000 разработан той же группой экспертов в области фотографии, что и JPEG. Формирование JPEG как международного стандарта было закончено в 1992 году. В 1997 стало ясно, что необходим новый, более гибкий и мощный стандарт, который и был доработан к зиме 2000 года. Основные отличия алгоритма JPEG2000 от JPEG заключаются в следующем:

1. Лучшее качество изображения при сильной степени сжатия.
2. Поддержка кодирования отдельных областей с лучшим качеством. Здесь предполагается, что имеется возможность человеку «на глаз» определить: какие области можно сжать с меньшим качеством, а для каких оставить прежнее. В результате при одинаковом субъективном качестве изображения могут быть достигнуты более высокие степени сжатия.
3. Алгоритм основан на вейвлет-преобразовании. Появилась возможность постепенного проявления изображения при его загрузке по сетям связи.
4. Бит-ориентированное арифметическое кодирование. Арифметическое кодирование предполагалось использовать еще в стандарте JPEG, но тогда оно было защищено патентами. Сейчас срок действия основного патента истек и стало возможным его применение в JPEG2000.
5. Поддержка сжатия без потерь. Благодаря этой опции появилась возможность применения JPEG2000 при сжатии медицинских изображений, полиграфии, для задач распознавания текста и т.п., критичных к потерям информации.
6. Поддержка сжатия однобитных (2-цветных) изображений.

Рассмотрим алгоритм JPEG2000 по шагам.

Шаг 1. В JPEG2000 выполняется центрирование яркости каждой компоненты RGB изображения перед преобразованием в цветовое пространство YUV. Это делается для выравнивания динамического диапазона, что приводит к увеличению степени сжатия. Формулу преобразования можно записать так:

$$x'(i,j) = x(i,j) - 2^{N-1},$$

где N - число бит/пиксел для кодируемого сигнала. Соответственно, обратное восстановление имеет вид

$$x(i,j) = x(i,j)' + 2^{N-1}.$$

Шаг 2. Переводим изображение из пространства RGB в пространство YUV. Этот перевод делается аналогично алгоритму JPEG в случае сжатия с потерями. При сжатии без потерь пространство переводится с помощью выражения:

$$\begin{vmatrix} Y \\ U \\ V \end{vmatrix} = \begin{vmatrix} \left\lfloor \frac{R+2G+B}{4} \right\rfloor \\ R-G \\ B-G \end{vmatrix},$$

обратное преобразование имеет вид

$$\begin{vmatrix} R \\ G \\ B \end{vmatrix} = \begin{vmatrix} U+G \\ Y-\left\lfloor \frac{U+V}{4} \right\rfloor \\ Y+G \end{vmatrix}.$$

Шаг 3. Дискретное ВП может быть двух видов – для сжатия с потерями и без. Его коэффициенты задаются таблицами, приведенными ниже.

Коэффициенты при упаковке		
i	Низкочастотные коэффициенты $h_L(i)$	Высокочастотные коэффициенты $h_H(i)$
0	1.1150870524569 94	0.6029490182363579
±1	0.5912717631142470	-0.2668641184428723
±2	-0.05754352622849957	-0.07822326652898785
±3	-0.09127176311424948	0.01686411844287495
±4	0	0.02674875741080976
Другие i	0	0

Коэффициенты при распаковке		
i	Низкочастотные коэффициенты $g_L(i)$	Высокочастотные коэффициенты $g_H(i)$
0	0.6029490182363579	1.1150870524569 94
±1	-0.2668641184428723	0.5912717631142470
±2	-0.07822326652898785	-0.05754352622849957
±3	0.01686411844287495	-0.09127176311424948
±4	0.02674875741080976	0
Другие i	0	0

Для сжатия без потерь коэффициенты задаются как:

	При упаковке		При распаковке	
i	Низкочастотные коэффициенты $h_L(i)$	Высокочастотные коэффициенты $h_H(i)$	Низкочастотные коэффициенты $g_L(i)$	Высокочастотные коэффициенты $g_H(i)$
0	6/8	1	1	6/8
±1	2/8	-1/2	1/2	-2/8
±2	-1/8	0	0	-1/8

В случае сжатия без потерь прямое ВП в одномерном случае будет иметь вид

$$y(2n+1) = -\frac{x(2n)}{2} + x(2n+1) - \frac{x(2n+2)}{2},$$

$$y(2n) = -\frac{y(2n-1)}{4} + x(2n) - \frac{y(2n+1)}{4},$$

что соответствует лифтинговой схеме для вычисления низкочастотной и высокочастотной составляющих. Так как преобразование должно быть без потерь, то сделаем так, чтобы выходные значения y были целочисленными. Это достигается следующими формулами:

$$y(2n+1) = x(2n+1) - \left\lfloor \frac{x(2n) + x(2n+2)}{2} \right\rfloor,$$

$$y(2n) = x(2n) + \left\lfloor \frac{y(2n-1) + y(2n+1) + 2}{4} \right\rfloor.$$

Рассмотрим на примере, как работает данное преобразование. Для того, чтобы преобразование можно было применять к крайним пикселам изображения, оно симметрично достраивается в обе стороны на несколько пикселов, как показано на рисунке ниже.

Симметричное расширение изображения (яркости АБ…Е) по строке вправо и влево

Пусть мы преобразуем строку из 10 пикселов. Расширим ее значения вправо и влево и применим ДВП:

n	-2	-1	0	1	2	3	4	5	6	7	8	9	10	11
x_{in}	3	2	1	2	3	7	10	15	12	9	10	5	10	9
y_{out}		0	1	0	3	1	11	4	13	-2	8	-5		

Получившаяся строка 1, 0, 1, 11, 4, 13, -2, 8, -5 и является цепочкой, однозначно задающей исходные данные.

Формулы для восстановления будут иметь вид

$$x(2n) = y(2n) - \left\lfloor \frac{y(2n-1) + y(2n+1) + 2}{4} \right\rfloor$$

$$x(2n+1) = y(2n+1) + \left\lfloor \frac{x(2n) + x(2n+2)}{2} \right\rfloor$$

Применение этих формул даст

n	-2	-1	0	1	2	3	4	5	6	7	8	9	10	11
y_{out}		0	1	0	3	1	11	4	13	-2	8	-5	8	-2
x_{out}			1	2	3	7	10	15	12	9	10	5	10	

В результате получили исходную цепочку чисел.

После ДВП вектор Y переупорядочивается так, чтобы элементы с четными индексами записывались в начало вектора, а с нечетными – в конец. В результате получится разделение на низко- и высокочастотную составляющую.

ДВП с последующим переупорядочиванием элементов в случае изображения применяется сначала к строкам, а затем к столбцам. В результате получается двумерное разделимое ВП. В результате изображение делится на 4 квадранта. После чего преобразование повторно применяется уже только к первому квадранту – 2-й этап декомпозиции.

Пример одного шага ВП

При сохранении результатов под данные 2 и 3 квадрантов выделяется на 1 бит больше, а под данные 4 квадранта – на 2 бита больше. То есть, если исходные данные были 8 битные, то на 2 и 3 квадранты нужно 9 бит, а на 4 – 10, независимо от уровня ДВП. При записи коэффициентов в файл можно использовать иерархическую структуру ДВП, помещая коэффициенты преобразований с большего уровня в начало файла. Это позволяет получить «изображение для предварительного просмотра»,

прочитав небольшой участок из начала файла, не распаковывая весь файл, как это приходилось делать в стандарте JPEG.

Шаг 4. Квантование. Коэффициенты квадрантов делятся на заранее заданное число. При увеличении этого числа снижается динамический диапазон коэффициентов, они становятся ближе к 0, и мы получаем большую степень сжатия. Варьируя эти числа для разных уровней преобразования, для разных цветовых компонент и разных квадрантов, можно получать разные степени сжатия при разных потерях. Формула прямого преобразования при квантовании имеет вид:

$$y^q(i,j) = \left\lfloor |y(i,j)| / \Delta \right\rfloor \mathrm{sgn}(y(i,j)),$$

где Δ - шаг квантования; $\lfloor \bullet \rfloor$ - знак округления до наименьшего целого. Обратное преобразование записывается как

$$\hat{y}(i,j) = (y^q(i,j) + r\,\mathrm{sgn}(y(i,j)))\Delta,$$

где r - некоторый коэффициент не определенный строго стандартом, но во многих практических случаях он устанавливается равный 0,5. В результате получается скалярное квантование с «мертвой зоной» в нуле.

Шаг 5. Для сжатия получающихся массивов данных в JPEG2000 используется вариант арифметического сжатия, называемый MQ-кодер.

На первом этапе квантованные в-к на каждом этапе декомпозиции разделяются в кодовые блоки. Кодовые блоки – это прямоугольники с произвольной шириной и длиной, которые должны удовлетворять двум условиям: 1 – длина и ширина должны принимать значения 2^n, n - целое положительное число; 2 – произведение ширины на длину не должно превышать значения 4096. Типичный размер кодовых блоков 64x64 отсчета. После разделения коэффициентов в кодовые блоки они независимо кодируются, используя битовый кодер. Для этих целей используется контекстно-зависимый адаптивный бинарный арифметический кодер, известный как MQ-кодер. После разложения каждого блока на битовые плоскости они кодируются независимо. При этом контекст определяется на основе ближайших отсчетов, как показано на рис. 9.

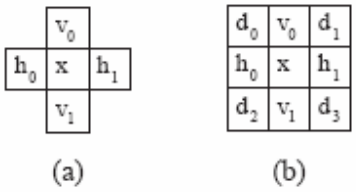


Fig. 9. Templates for context selection. The (a) 4-connected and (b) 8-connected neighbors.

В рамках стандарта JPEG2000 имеется возможность выбирать отдельные области для их лучшего представления, по сравнению с остальным изображением. В этом случае мы можем в разы увеличить степень сжатия за счет изменения качества разных участков изображений.

Проблемой этого подхода является то, что необходимо каким-то образом узнавать расположение наиболее важных для человека участков изображения.

Например, таким участком может быть лицо человека на фоне природы. Если при сжатии будет размыто дерево на фоне, то это не так критично как размытия лица.

Работы по автоматическому выделению таких областей активно ведутся. В частности, созданы алгоритмы автоматического выделения лиц на изображениях. Продолжаются исследования методов выделения наиболее значимых контуров и т.д. Однако очевидно, что универсальный алгоритм в ближайшее время создан не будет, поскольку для это требуется критерий визуального восприятия, которого на сегодняшний день отсутствует.

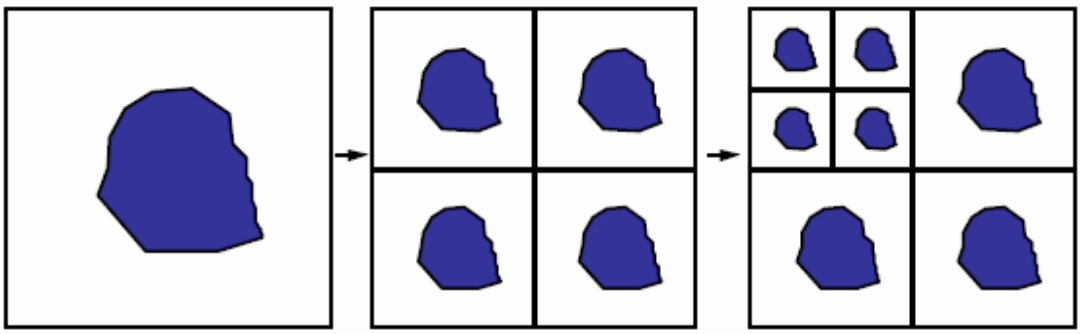
Вместо этого вполне реально применение автоматизированных алгоритмов сжатия, в которых значимые области определяются самим человеком. Данный подход уменьшает количество возможных областей применения такого алгоритма, но позволяет достигать больших степеней сжатия. Такой подход целесообразно применять, если:

1. Для приложения должна быть критична степень сжатия, причем настолько, что возможен индивидуальный подход к каждому изображению.
2. Изображение сжимается один раз, а разжимается множество раз.

В качестве примеров приложений, удовлетворяющим этим требованиям, можно привести практически все мультимедийные продукты на DVD. Также сюда относятся DVD энциклопедии и игры, где графика занимает до 70% всего объема. При этом технология производства дисков позволяет сжимать каждое изображение индивидуально, максимально повышая степень сжатия.

Интересным примером являются WWW-сервера. Для них тоже, как правило, выполняются оба изложенных выше условия. При этом совершенно не обязательно индивидуально подходить к каждому изображению поскольку по статистике 10% изображений будут запрашиваться 90% раз. То есть для крупных справочных или игровых серверов появляется возможность уменьшать время загрузки изображений и степень загруженности каналов связи адаптивно.

В JPEG2000 используется однобитное изображение-маска, задающее повышение качества в данной области изображения. Поскольку за качество областей у нас отвечают коэффициенты ДВП преобразования во 2, 3 и 4 квадрантах, то маска преобразуется т.о., чтобы указывать на все коэффициенты, соответствующие областям повышения качества:



Эти области обрабатываются далее другими алгоритмами (с меньшими потерями), что и позволяет достичь более высоких показателей сжатия.

Особенности сжатия в алгоритме SPIHT

Метод SPIHT был разработан для оптимальной прогрессирующей передачи изображений, а также для их сжатия. Самая важная особенность этого алгоритма заключается в том, что на любом этапе декодирования качество отображаемой в этот момент картинки является наилучшим для введенного объема информации. Также алгоритм SPIHT использует вложенное кодирование, т.е. если кодер делает два файла: один M бит, а другой поменьше m бит, то меньший файл совпадет с первыми m битами большого.

Основная цель прогрессирующего метода состоит в скорейшей передачи самой важной части информации об изображении. Эта информация дает самое большое сокращение расхождения исходного и восстановленного образов. Для количественного измерения этого расхождения в SPIHT используется выражение:

$$D = \frac{1}{N}\sum_i\sum_j \left(x_{ij} - \hat{x}_{ij}\right)^2 = \frac{1}{N}\sum_i\sum_j \left(c_{ij} - \hat{c}_{ij}\right)^2 .$$

Данное уравнение показывает, что мера уменьшается на $\left|c_{ij}\right|^2 / N$, когда кодер получает коэффициент преобразования c_{ij}. Отсюда становится ясно, что самые большие по модулю коэффициенты несут в себе самую большую информацию, сокращающую указанную меру. Поэтому прогрессирующее кодирование должно посылать эти коэффициенты в первую очередь. Как распознать такие коэффициенты в их битовом представлении? Очень просто. Так как при увеличении номера бита увеличивается и число, которое он описывает, то в таком алгоритме следует передавать или записывать сначала биты коэффициентов, у которых установлен самый старший бит, затем, брать более младшие биты и уточнять предыдущие коэффициенты плюс добавлять новые, у которых установлен этот более младший бит и т.д. Опишем этот процесс более детально.

Предположим, что вейвлет-коэффициенты $\{c_{ij}\}$ представляются 16 битами, причем, бит №15 будет содержать знак коэффициента, а остальные биты с 0 до 14 – его величину. На первом этапе выполняется их сортировка по убыванию, т.е. они выстраиваются в вектор. Теперь необходимо самую значимую информацию передать в первую очередь. Но это означает, что необходимо сначала взять коэффициенты, лежащие в диапазоне $2^{14} \le \left|c_{ij}\right| < 2^{15}$, что гарантирует, что 14-й бит числа установлен в 1, а знак может быть любым. Предположим, что таких коэффициентов два с координатами (2,3) и (3,4). Соответственно на выходе записываем значение 2, за которым следует пара координат (2,3) (3,4), а затем, идут 2 знаковых бита, т.е. записываются биты №15. Имея эту информацию, декодер сможет грубо восстановить

эти два коэффициента в виде двух чисел вида s100000000000000, где s – знак числа. Таким образом, самые значимые биты самых значимых коэффициентов передаются в первую очередь.

После этого, из сортированного массива выбираются коэффициенты, удовлетворяющие условию $2^{13} \le |c_{ij}| < 2^{14}$. Предположим, что таких коэффициента 4. Следовательно, число 4 передается на выход, за которым следуют координаты, например, (3,2) (4,4) (1,2) (3,1), знаки этих 4-х коэффициентов и 13-е биты предыдущих двух коэффициентов. Благодаря этой информации, декодер сможет уточнить первые два переданных коэффициента и узнать о следующих 4-х менее важных коэффициентах.

Данная процедура выполняется до тех пор, пока на выход не будут переданы все коэффициенты, описывающие конкретное изображение.

Главным недостатком этого алгоритма является большой объем информации координат передаваемых значимых коэффициентов $\{c_{ij}\}$. Для того чтобы оптимизировать данный процесс в алгоритме SPIHT поступают следующим образом. Разбивают все множество коэффициентов на подмножества $\{T_k\}$, например, так как показано на рис. 1.

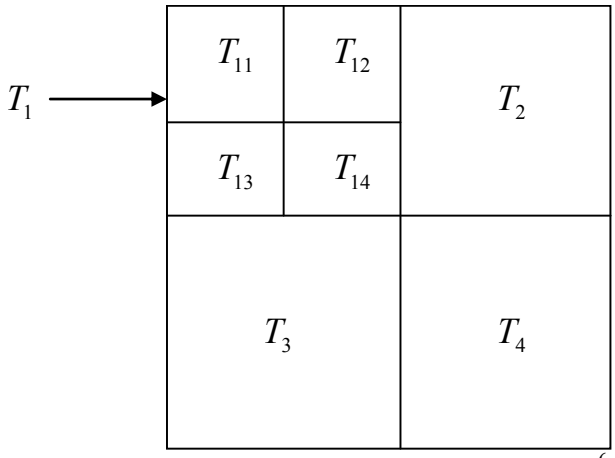


Рис. 1. Пример расположения множеств $\{T_k\}$

Затем, просматривают каждое множество T_1, T_2, T_3, T_4 и проверяют, имеются ли в них значимые коэффициенты $\{c_{ij}\}$, т.е. коэффициенты, удовлетворяющие условию $|c_{ij}| \ge 2^n$. Если найден в множестве T_k хотя бы один такой коэффициент, то такое множество также считается существенным. Предположим, что в нашем примере значимым множеством является только T_1. Тогда оно разбивается на подмножества $T_{11}, T_{12}, T_{13}, T_{14}$, для которых выполняется такая же проверка коэффициентов $\{c_{ij}\}$. И так далее пока не дойдем до множеств, состоящих из одного коэффициента. В результате декодеру передается битовая информация, указывающая, является ли конкретное множество T_k значимым или нет. Благодаря иерархической структуре

множеств $\{T_k\}$, общий объем информации о расположении значимых коэффициентов становится меньше, чем простая передача их координат.

Введение во фракталы

Рассмотрим способ построения и свойства известной фрактальной кривой Коха [3]. Для этого возьмем равносторонний треугольник, на каждой стороне которого достроим по треугольнику, сторона которого в три, а значит, площадь в девять раз меньше, чем у исходного (рис. 6.1). Повторим аналогичные действия с полученной кривой на втором шаге. И так далее. То, что получится после бесконечного количества таких шагов, называется кривой Коха. Определим периметр данной фигуры. Очевидно, что на втором шаге периметр фигуры увеличится в 4/3 раза. На третьем – еще в 4/3. Это произошло потому, что каждый отрезок заменялся ломаной, длина которой в 4/3 раза больше. А $(4/3)^n$ при n, стремящемся к бесконечности, также стремится к бесконечности. В то же время, если воспользоваться геометрической прогрессией, то можно убедиться, что площадь фигуры Коха конечна.

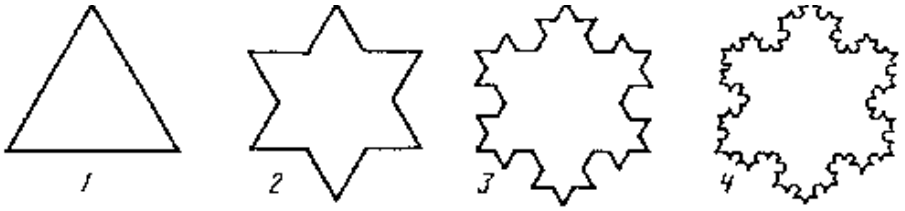


Рис. 6.1. Схема построения кривой Коха

Интересной особенностью данного построения является то, длина его периметра зависит от длины линейки, которой он измеряется. Действительно, каждый раз измеряя периметр, сложная линия кривой Коха заменяется ломаной, звенья которой не превышают длину линейки. Поэтому с уменьшением длины линейки будет увеличиваться и измеренный периметр.

Кривая Коха обладает еще одной интересной особенностью. Если взять одну из ее граней и увеличить, то увидим примерно следующее (рис. 6.2 а). При этом мелкие детали в крупном масштабе естественно будут теряться. Увеличим один «зубец» этой кривой до размеров исходной фигуры. В результате получим примерно такое же изображение (рис. 6.2 б). Если повторить этот эксперимент с рис. 6.2 б), то опять получим примерно такое же изображение (рис. 6.2 в). И так далее. Такое свойство фигуры выглядеть в любом сколь угодно малом масштабе примерно одинаково называется масштабной инвариантностью, а множества, которые им обладают, называются фракталами [3,4].

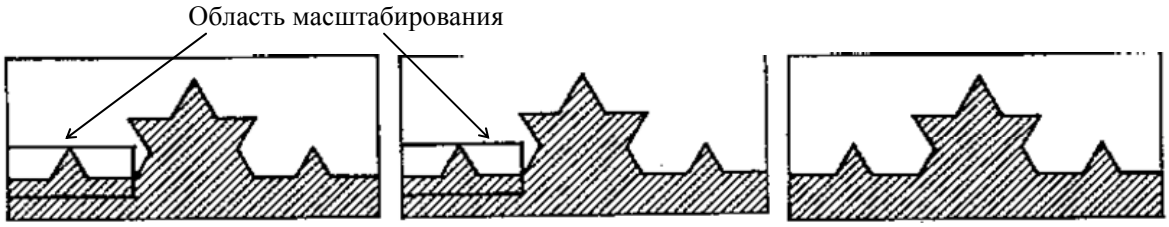


Рис. 6.2. Пример масштабной инвариантности:
а) исходное изображение;
б) увеличенная область масштабирования в три раза;
в) увеличенная область масштабирования в девять раз

Интересно также то, что кривая Коха принадлежит дробному пространству, а именно 1,26 размерности. Дробность размерности умозрительно можно представить в виде области определения кривой. Например, для линии область определения – принадлежит одномерному пространству, множество точек квадрата покрывают двумерное пространство, а область определения кривой Коха - полоса в двумерном пространстве, т.е. и не линия и не плоскость – дробная размерность.

Фигура Коха имеет непосредственное отношение к реальности. Например, английские военные топографы еще до войны заметили, что длина побережья Великобритании зависит от длины линейки, которой ее измеряют. Аналогичная зависимость определяет длину некоторых рек, побережье многих островов, путь, проходимый частицей при броуновском движении, и многое другое.

Другим известным примером фрактала является ковер Серпинского, придуманный польским математиком в 1915 г. Построение данной фигуры выполняется следующим образом. На первом этапе берется равносторонний треугольник вместе с областью, которую он охватывает (рис. 6.3, а). Затем из этого треугольника удаляется центральная треугольная область как показано на рис. 6.3, б. Повторяя эти действия многократно получаем изображение ковра Серпинского (рис. 6.3, с).

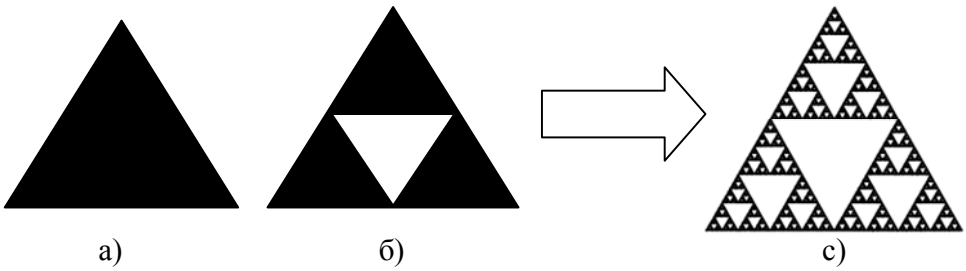


Рис. 6.3. Этапы построения ковра Серпинского

Интересной особенностью данного построения является то, что площадь отброшенных частей в точности равна площади исходного треугольника. Действительно, пусть изначально площадь треугольника была равна единице. На

первом шаге удаляется ¼ его площади, на втором $3 \cdot 1/4^2$ и т.д. В пределе, при числе шагов стремящихся к бесконечности, получается следующий сходящийся ряд:

$$\lim S = \lim \sum_{i=1}^{\infty} \frac{3^{i-1}}{4^i} = 1.$$

Следовательно, можно утверждать, что ковер Серпинского имеет нулевую площадь.

В общем случае существует большое многообразие фракталов, но все они обладают двумя свойствами: масштабная инвариантность (самоподобие) и дробность размерности.

Системы итерируемых функций

Мы обратимся теперь к одному из наиболее замечательных и глубоких достижений в построении фракталов — системам итерированных функций. Математические аспекты были разработаны Джоном Хатчинсоном [23], а сам метод стал широко известен благодаря Майклу Барнсли [4] и другим. Подход на основе систем итерированных функций предоставляет хорошую теоретическую базу для математического исследования многих классических фракталов, а также их обобщений. Разработанная теория непосредственно используется при переходе к исследованию хаоса, связанного с фракталами.

Следует иметь в виду с самого начала, что результат применения системы итерированных функций, называемый *аттрактором,* не всегда является фракталом. Это может быть любой компакт, включая интервал или квадрат. Тем не менее, изучение систем итерированных функций важно для фрактальной теории, так как с их помощью можно получить удивительное множество фракталов. Теория итерированных функций замечательна сама по себе и служит составной частью общей теории динамических систем, важного раздела математики.

Прежде чем углубиться в теорию систем итерированных функций, рассмотрим пример, а именно ковер Серпинского, который мы уже строили прежде. Для построения мы выбирали в качестве исходного множества треугольник и на каждом шаге выкидывали центральную треугольную часть (не включая границу) образующихся треугольников. Ниже мы рассмотрим два других метода: *детерминированный* и *рандомизированный*.

Построение ковра Серпинского в детерминированном алгоритме начинается с выбора компактного множества E_0, например квадрата (рис. 4.1). Затем задаются аффинные преобразования так чтобы исходное множество E_0 было преобразовано, как показано на втором шаге построения рис. 4.1. Это достигается путем выбора следующих трех преобразований:

$$T_1\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{bmatrix} 1/2 & 0 \\ 0 & 1/2 \end{bmatrix} \cdot \begin{vmatrix} x_1 \\ x_2 \end{vmatrix} + \begin{vmatrix} 0 \\ 0 \end{vmatrix},$$

$$T_2\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{bmatrix} 1/2 & 0 \\ 0 & 1/2 \end{bmatrix} \cdot \begin{vmatrix} x_1 \\ x_2 \end{vmatrix} + \begin{vmatrix} 1/2 \\ 0 \end{vmatrix},$$

$$T_3\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{bmatrix} 1/2 & 0 \\ 0 & 1/2 \end{bmatrix} \cdot \begin{vmatrix} x_1 \\ x_2 \end{vmatrix} + \begin{vmatrix} 1/4 \\ \sqrt{3}/4 \end{vmatrix}$$

и преобразование компактного множества E_0 записывается в виде

$$E_1 = T_1(E_0) \cup T_2(E_0) \cup T_3(E_0).$$

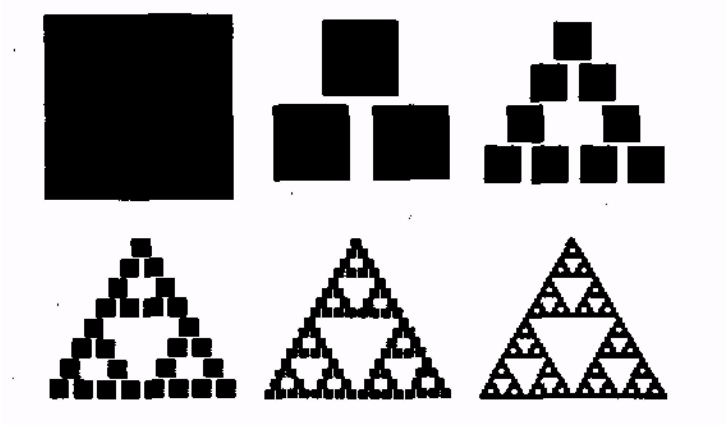


Рис. 4.1. Построение ковра Серпинского
с помощью детерминированной СИФ

После выполнения n таких преобразований, получаем изображение соответствующее ковру Серпинского:

$$E_n = T_1(E_{n-1}) \cup T_2(E_{n-1}) \cup T_3(E_{n-1}).$$

Заметим, что все множество возможных построений определяется набором аффинных преобразований $T_1, \ldots, T_n$.

Обычно коэффициенты аффинного преобразования записываются в матрицу C размером $n \times 6$ элементов:

$$C = \begin{bmatrix} a_1 & b_1 & c_1 & d_1 & e_1 & f_1 \\ a_2 & b_2 & c_2 & d_2 & e_2 & f_2 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ a_n & b_n & c_n & d_n & e_n & f_n \end{bmatrix},$$

которая полностью описывает фрактальное множество.

В рандомизированном алгоритме, который часто называют игрой в «Хаос», в качестве начального множества выбирают одну точку:

x_0 - начальная точка (с произвольными координатами)

$x_1 = T_1(x_0)$ или $T_2(x_0)$ или $T_3(x_0)$

$\vdots$

$x_n = T_1(x_{n-1})$ или $T_2(x_{n-1})$ или $T_3(x_{n-1})$

$\vdots$

На каждом шаге, вместо того чтобы применять сразу три преобразования $T_1(S), T_2(S), T_3(S)$, мы применяем только одно, выбранное случайным образом. Таким

образом, на каждом шаге мы получаем ровно одну точку. Оказывается, что после некоторого переходного процесса точки, сгенерированные в рандомизированном алгоритме, заполняют в точности ковер Серпинского.

Замечательным свойством алгоритмов, основанных на теории систем итерированных функций, является то, что их результат (аттрактор) совершенно не зависит от выбора начального множества E_0 или начальной точки x_0 В случае детерминированного алгоритма это означает, что в качестве E_0 можно взять любое компактное множество на плоскости: предельное множество по-прежнему будет совпадать с ковром Серпинского. В случае рандомизированного алгоритма, вне зависимости от выбора начальной точки x_0, после нескольких итераций точки начинают заполнять ковер Серпинского.

Для равномерного распределения точек по экрану в рандомизированном алгоритме аффинные преобразования следует выбирать с вероятностью

$$p_i = \det(A_i) / \sum_{j=1}^{n} \det(A_j),$$

где $A_i = \begin{bmatrix} a_i & b_i \\ c_i & d_i \end{bmatrix}$ - матрица соответствующего аффинного преобразования. Очевидно, что $p_1 + p_2 + ... + p_n = 1$.

В общем случае, для чтобы построить *систему итерированных функций* введем в рассмотрение совокупность сжимающих отображений:

T_1 - с коэффициентом сжатия $s_1 < 1$,

T_2 - с коэффициентом сжатия $s_2 < 1$,

$\vdots$

T_n - с коэффициентом сжатия $s_n < 1$.

Эти n отображений используются для построения одного сжимающего отображения T.

Таким образом, *системой итерированных функций* (СИФ) называют совокупность введенных выше отображений вместе с итерационной схемой:

E_0 - компактное множество (произвольное)

$E_1 = \mathrm{T}(E_0) = T_1(E_0) \bigcup ... \bigcup T_n(E_0)$,

$E_2 = \mathrm{T}(E_1)$,

$\vdots$

$E_n = \mathrm{T}(E_{n-1})$,

$\vdots$

Основная задача теории СИФ — выяснить, когда СИФ порождает предельное множество E_0:

$$E = \lim_{n \to \infty} E_n .$$

Если предел существует, то множество E называют аттрактором системы итерированных функций. Причем аттрактор часто (но не всегда!) оказывается

фрактальным множеством. Очевидно, для того чтобы обеспечить сходимость, требуется наложить определенные ограничения на введенные выше преобразования, к примеру запретить точкам уходить на бесконечность.

Основные математические идеи, необходимые для установления условий сходимости, уже были представлены. Если нам удастся показать, что T является сжимающим отображением на метрическом пространстве *(K.,H),* то мы сможем применить теорию сжимающих отображений. В этом случае аттрактор E есть не что иное, как неподвижная точка отображения T.

Сжатие изображений с использованием СИФ

Из примеров построения фракталов видно, что получаемые таким образом изображения могут быть очень похожими на реальные. Например, с помощью фрактальной геометрии можно получать реалистичные изображения листьев, трав, деревьев, гор, рек и других природных объектов. Причем каждый раз при генерации фрактальных множеств (изображений) используется только матрица коэффициентов C, содержащая коэффициенты сжимающих аффинных преобразований. Можно заметить, что объем данных, занимаемых элементами матрицы C, как правило, много меньше объема сгенерированного изображения, даже если его записать в известных форматах gif, jpeg, tif и др. Таким образом, некоторые реалистичные изображения можно описать коэффициентами аффинных преобразований и генерировать аттрактор с любым масштабом. При таком подходе иногда удается получать довольно большие коэффициенты сжатия от 1:1000 до 1:10000. Кроме того, масштаб сгенерированных изображений с помощью СИФ может быть любым при сохранении хорошего визуального качества.

При сжатии изображений используют трехмерные сжимающие преобразования вида

$$T\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & p \end{bmatrix} \begin{vmatrix} x \\ y \\ z \end{vmatrix} + \begin{vmatrix} e \\ f \\ q \end{vmatrix}.$$

Здесь третья координата z соответствует значению яркости отсчета изображения.

Допустим, что подбирается первое сжимающее преобразование, которое должно описывать область изображения, например, 8x8 пикселов (рис. 2), которая называется ранговой. Для этого перебираются на кодируемом изображении все возможные области большего размера, например 16x16 пикселов (доменные), и к ним применяется сжимающее преобразование T_1. Коэффициенты данного преобразования подбираются таким образом, чтобы сжатая область изображения соответствовала размеру 8x8 отсчетов и располагалась на месте ранговой области. Для наилучшего соответствия сжатой доменной области ранговой необходимо перебрать все возможные варианты и среди них выбрать тот, для которого мера d является минимальной.

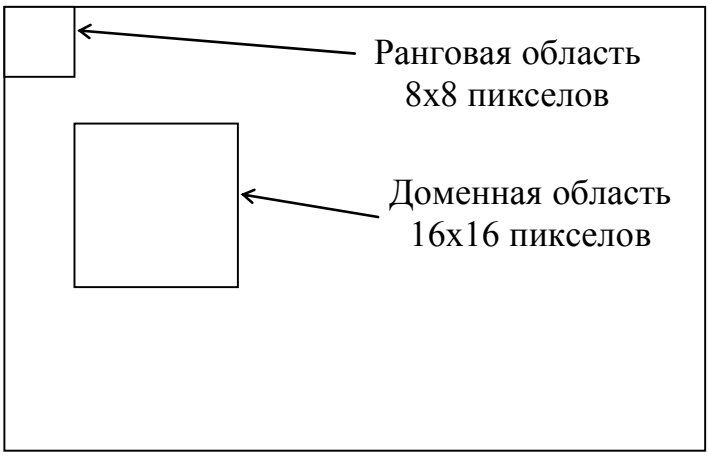


Рис. 2. Иллюстрация ранговой и доменной областей

Таким образом, целью сжатия является определение минимального числа сжимающих преобразований $T_i(\bar{x}), i = \overline{1,n}$, которые бы описывали изображение с наилучшим качеством d. Под качеством описания обычно понимают среднеквадратическую меру между исходным X и преобразованным $\hat{X}$ изображениями:

$$d = \sqrt{\frac{1}{N^2}\sum_{i=}^{N}\sum_{j=1}^{N}\left(x_{ij} - \hat{x}_{ij}\right)^2}\ .$$

На сегодняшний день подбор сжимающих преобразований является основной проблемой при фрактальном сжатии изображений, т.к. всевозможный перебор вариантов даже для небольших по размерам изображений составляет астрономическое число. Поэтому на практике ограничивают число возможных переборов, например следующим образом.

В ***учебном варианте алгоритма***, изложенном далее, сделаны следующие ограничения на области:

1. Все области *являются квадратами со сторонами, параллельными сторонам изображения.* Это ограничение достаточно жесткое. Фактически мы собираемся аппроксимировать все многообразие геометрических фигур лишь квадратами.
2. При переводе доменной области в ранговую уменьшение размеров производится *ровно в два раза*. Это существенно упрощает как компрессор, так и декомпрессор, т.к. задача масштабирования небольших областей является нетривиальной.
3. Все доменные блоки — квадраты и имеют *фиксированный размер*. Изображение равномерной сеткой разбивается на набор доменных блоков.
4. Доменные области берутся *“через точку” и по X, и по Y*, что сразу уменьшает перебор в 4 раза.

5. При переводе доменной области в ранговую поворот куба возможен *только на 0^0, 90^0, 180^0 или 270^0*. Также допускается зеркальное отражение. Общее число возможных преобразований (считая пустое) — 8.
6. Масштабирование (сжатие) по вертикали (яркости) осуществляется *в фиксированное число раз* — в 0,75.

Информацию о размере блоков можно хранить в заголовке файла. Таким образом, мы затратили менее 4 байт на одно аффинное преобразование. В зависимости от того, каков размер блока, можно высчитать, сколько блоков будет в изображении. Таким образом, мы можем получить оценку степени компрессии.

Например, для файла в градациях серого 256 цветов 512x512 пикселов при размере блока 8 пикселов аффинных преобразований будет 4096 (512/8x512/8). На каждое потребуется 3.5 байта. Следовательно, если исходный файл занимал 262144 (512x512) байт (без учета заголовка), то файл с коэффициентами будет занимать 14336 байт. Коэффициент архивации — 18 раз. При этом мы не учитываем, что файл с коэффициентами тоже может обладать избыточностью и архивироваться методом архивации без потерь, например LZW.

Отрицательные стороны предложенных ограничений:

1. Поскольку все области являются квадратами, невозможно воспользоваться подобием объектов, по форме далеких от квадратов (которые встречаются в реальных изображениях достаточно часто.)

2. Аналогично мы не сможем воспользоваться подобием объектов в изображении, коэффициент подобия между которыми сильно отличается от 2.

3. Алгоритм не сможет воспользоваться подобием объектов в изображении, угол между которыми не кратен 90^0.

Такова плата за **скорость компрессии** и за простоту упаковки коэффициентов в файл. Сам алгоритм упаковки сводится к перебору всех доменных блоков и подбору для каждого соответствующего ему рангового блока.

Как видно из приведенного алгоритма, для каждого рангового блока делаем его проверку со всеми возможными доменными блоками (в том числе с прошедшими преобразование симметрии), находим вариант с наименьшей мерой L_2 (наименьшим среднеквадратичным отклонением) и сохраняем коэффициенты этого преобразования в файл. Коэффициенты — это (1) координаты найденного блока, (2) число от 0 до 7, характеризующее преобразование симметрии (поворот, отражение блока), и (3) сдвиг по яркости для этой пары блоков. Сдвиг по яркости вычисляется как:

$$q = \left[\sum_{i=1}^{n} \sum_{j=1}^{n} d_{ij} - \sum_{i=1}^{n} \sum_{j=1}^{n} r_{ij} \right] / n^2 ,$$

где r_{ij} — значения пикселов рангового блока (R), а d_{ij} — значения пикселов доменного блока (D). При этом мера считается как:

$$d(R,D) = \sum_{i=1}^{n} \sum_{j=1}^{n} (0.75 r_{ij} + q - d_{ij})^2 .$$

Декомпрессия алгоритма фрактального сжатия чрезвычайно проста. Необходимо провести несколько итераций трехмерных аффинных преобразований, коэффициенты которых были получены на этапе компрессии.

В качестве начального может быть взято абсолютно любое изображение (например, черное), поскольку соответствующий математический аппарат гарантирует нам сходимость последовательности изображений, получаемых в ходе итераций СИФ, к неподвижному изображению (близкому к исходному). Обычно для этого достаточно 16 итераций.

Поскольку мы записывали коэффициенты для блоков *Rij* (которые, как мы оговорили, в нашем частном случае являются квадратами одинакового размера) *последовательно*, то получается, что мы последовательно заполняем изображение по квадратам сетки разбиения использованием аффинного преобразования.

Основы видеокодирования

Сжатие видео основано на двух важных принципах. Первый – это пространственная избыточность, присущая каждому кадру видеоряда. А второй принцип основан на том факте, что большую часть времени каждый кадр похож на своего предшественника. Это называется временной избыточностью. Таким образом, типичный метод сжатия видео начинается с кодирования первого кадра с помощью стандартного алгоритма сжатия изображения, а затем выполняет кодирование последующего кадра в виде разности между ним и предшествующим и полученное разностное изображение сжимает методом сжатия изображений.

Сжатие видео это почти всегда сжатие с потерей информации, иначе объем видео становится слишком большим для его хранения. Поэтому первый кадр кодируется как изображение алгоритмом сжатия с потерями, например, JPEG. Такой закодированный кадр называют I кадром. Следующий, второй кадр уже можно закодировать как разность между ним и первым. Такой кадр называют P кадр, и который сжимается, как правило, лучше, чем исходный кадр (не разностный). Это наводит на мысль, что все видео можно представить в виде следующих типов кадров: IPPPPPPPP…, что называется временной моделью. Однако на практике, при передаче видео по каналам связи или при ошибках в записи данных или по каким-либо другим причинам возможны потери информации в закодированном видеоряде. Значит, при указанной временной модели ошибка в каком-либо из кадров будет приводить к нарастанию ошибок в последующих кадрах и она будет уже сохраняться на протяжении всего видео. Чтобы устранить этот недостаток временную модель можно представить, например, в виде IPPPIPPPIPPP… В этом случае ошибка в каком-либо кадре устранится когда появится кадр I типа.

В первых видеокодеках, например MPEG-1, временная модель и выглядела примерно в таком виде. Но при развитии компьютерной техники появилась возможность создавать более сложные временные модели. Для них ввели новый тип кадра, в котором разности вычисляются не только на основе предыдущего, но и на

основе последующего кадров и такой тип кадра назвали B кадр. Очевидно, что компенсация B кадра будет, как правило, лучше по сравнению с компенсацией кадра P типа, а значит и будут достигаться более высокие степени сжатия видео. Однако чтобы осуществить компенсацию по предыдущему и последующем кадрам они должны быть известны как кодеру, так и декодеру. Это значит, что они должны быть сжаты кодером и переданы декодеру до кодирования кадров B типа. Одной из самой распространенной временной моделью такого типа является следующая последовательность кадров этих трех типов:

I B B B P B B B P B B B P B B B I…

Такая временная модель предполагает, что кодер сначала закодирует первый кадр, как кадр I типа. Затем, закодирует 5-й кадр, как кадр P типа. После этого выполнит кодирование B кадров 2,3 и 4 на основе кадров I и P. В результате на выходе кодера будет формироваться последовательность:

I P B B B P B B B P B B B I B B B…

Для корректного воспроизведения видеоряда декодеру будет необходимо в буфере сформировать как минимум 5 кадров до их отображения, т.к. они идут не по порядку. Это требует соответствующую скорость работы декодера.

Рассмотрим подробнее процесс компенсации кадров на примере кадров P типа. В самом простом случае можно взять просто разность между текущим и предыдущим кадрами. Но этот результат можно улучшить, если учесть, что при движении объектов на сцене некоторые группы пикселов на соседних кадрах могут быть немного сдвинуты относительно друг друга. Следовательно, если найти такую группу пикселей и знать искомое их смещение, то можно более точно осуществить прогноз следующего кадра и получить лучшее сжатие.

В принципе эта перемещающаяся часть может иметь любую форму. Однако на практике ограничиваются квадратами и прямоугольниками заранее выбранных размеров. При этом предыдущий кадр разбивают на непересекающиеся блоки, например, квадратов и для каждого квадрата на текущем кадре находят область наиболее близко совпадающую с соответствующим квадратом предыдущего блока в соответствии с выбранной мерой. После того как эта область найдена, кодер формирует смещение $(\Delta x, \Delta y)$ соответствующего блока, которое называют вектором движения. В результате для каждого кадра P типа декодеру передается скомпенсированный кадр и набор векторов движения блоков. При этом размер блоков обычно выбирают 8 или 16. В результате получается следующая схема видеокодера (рис. 1).

Нахождение векторов движения

Эта процедура обычно занимает много времени, поэтому ее следует тщательно оптимизировать. Обозначим через B текущий блок предыдущего кадра. Задача заключается в нахождении близкого к нему блока текущего кадра. Обычно этот поиск делается не по всему изображению, а по некоторой области вокруг B, которая

задается с помощью параметров dx, dy. Эти параметры задают максимальное расстояние по горизонтали и вертикали в пикселах. Если блок B – это квадрат со стороной в b пикселов, то область поиска состоит из (b+2dx)(b+2dy) пикселей (рис. 2), среди которых можно определить (2dx+1)(2dy+1) частично перекрывающихся квадратов длиной b пикселей.

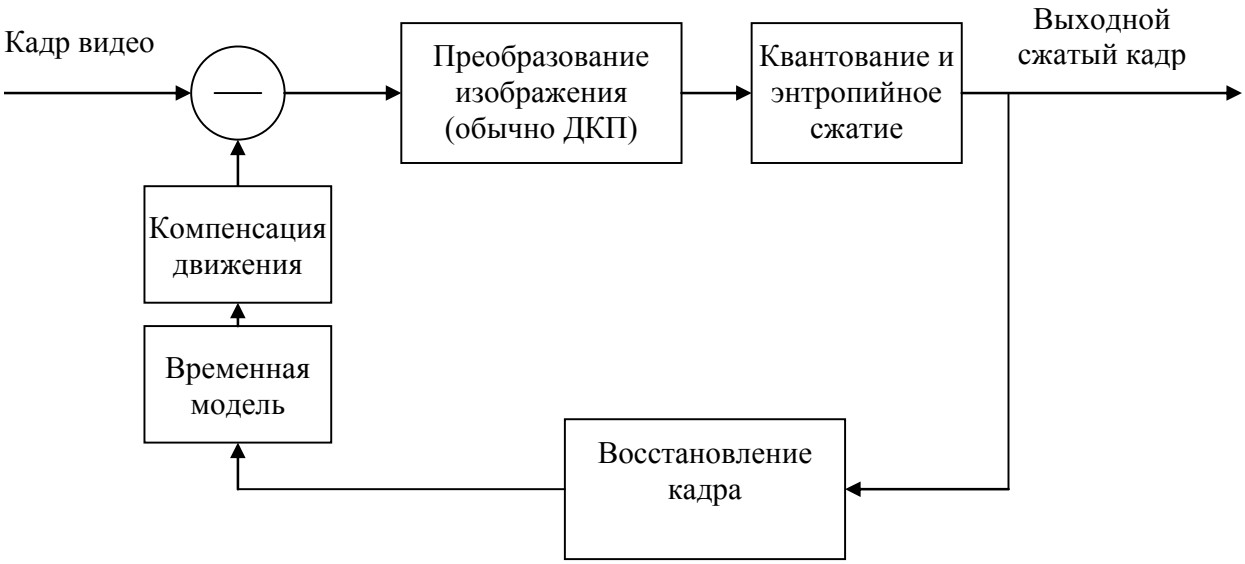


Рис. 1. Общая схема видеокодера

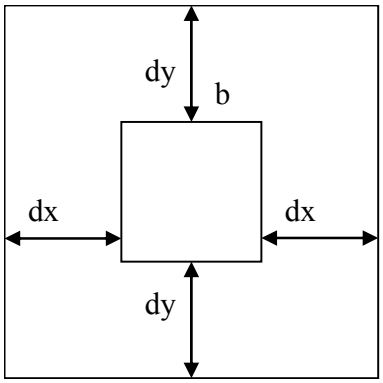


Рис. 2. Пример области поиска вектора движения

Для того, чтобы определять подобен или нет один блок на другой, необходимо определить меру сходства блоков. Причем эта процедура должна быть простой и быстрой, но, вместе с тем, надежной.

На практике обычно используют следующие меры. Средняя абсолютная разность, которая вычисляется по формуле

$$\frac{1}{b^2}\sum_{i=1}^{b}\sum_{j=1}^{b}\left|B_{ij}-C_{ij}\right|,$$

где B_{ij} - пиксел блока B; C_{ij} - пиксел блока-кандидата C. Для вычисления данной меры для одной позиции блоков необходимо произвести b^2 операций вычитания и взятия модуля числа. Эта мера соответственно вычисляется для каждого возможного положения (2dx+1)(2dy+1) блока-кандидата C до тех пор, пока она не будет превышать значение некоторого заданного порога C_k. Если для блока B не находится подходящего блока C, то блок B кодируется без компенсации движения.

Другой мерой расхождения может служить СКО, вычисляемой по формуле

$$\frac{1}{b^2}\sum_{i=1}^{b}\sum_{j=1}^{b}\left(B_{ij}-C_{ij}\right)^2 .$$

Кодирование векторов движения

Большая часть текущего кадра (возможно, его половина) может быть преобразована в векторы движения, поэтому кодирование этих векторов весьма актуально. Это кодирование должно быть без потерь. Известны два свойства векторов движения, которые позволяют сформулировать принципы их кодирования. Первое свойство заключается в том, что соседние блоки имеют близкие векторы движения. Второе свойство определяется тем, что векторы, как правило, направлены в одну, реже в две стороны; значит векторы распределены неравномерно.

На сегодняшний день не существует единого общего метода кодирования, который был бы идеальным для всех случаев. Обычно для этих целей применяется арифметическое кодирование, адаптивное кодирование Хаффмана в совокупности с алгоритмами прогнозирования «соседних» векторов движения.

Например, в MPEG-1 поступают таким образом. Строится прогноз вектора движения по его предшественникам, находящимся в той же строке и том же столбце текущего кадра. Затем, вычисляется разность между прогнозом и истинным значением вектора движения и эти разности кодируются методом Хаффмана.

Методы подоптимального поиска векторов движения

На практике важно, чтобы алгоритмы сжатия и восстановления видео работали быстро. Поэтому исследования в этой области направлены на уменьшение числа вычислений различных компонент кодеров, в том числе и для алгоритма поиска векторов движения.

В общем случае полный перебор всех возможных вариантов при поиске наилучшего вектора движения требует много времени, поэтому имеет смысл заняться поиском подоптимальных алгоритмов, которые осуществляют поиск не для всех вариантов, а для наиболее вероятных. Естественно, что такие методы не всегда находят самый похожий блок, но они существенно сокращают время поиска, а это часто важнее.

На сегодняшний день известно множество алгоритмов подоптимального поиска векторов движения. Рассмотрим наиболее известные среди них.

Сигнатурные методы: На первом шаге несколько лучших кандидатов определяются с помощью быстро вычисляемой меры, например, средней абсолютной разности, а потом, они уточняются с помощью более точной меры, например, величины взаимной корреляции.

Поиск с разбавленным расстоянием: Из опыта известно, что быстро перемещающиеся объекты выглядят смазанными при воспроизведении на экране, даже если имеют четкие очертания в каждом кадре. В результате можно построить такой алгоритм поиска векторов. Для близких блоков от блока B берутся все ближайшие блоки-кандидаты, а при увеличении расстояния, блоки берутся с большим шагом, а значит и их число будет меньше (рис. 3).

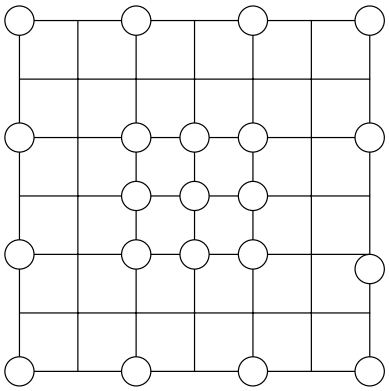


Рис. 3.

Локализованный поиск: Этот метод основан на гипотезе, что если хорошее совпадение найдено, что еще лучшее, скорее всего, находится где-то рядом. Тогда алгоритм поиска получается следующий. Сначала по разреженному расстоянию находится лучший блок, а затем, относительно него, с более точным расстоянием находится еще более подходящий.

Двумерный логарифмический поиск: Допустим имеется квадратная область (2d+1)(2d+1). В этой области берется пять точек с шагом $s = 2^{\lfloor \log_2 d \rfloor - 1}$ с координатами (a,b) (a-s,b) (a+s,b) (a,b-s) (a,b+s), т.е. в форме знака +. Затем выбирается наилучший блок из этих 5 и если лучший в точке (a,b), то s делится на 2 и поиск продолжается. В противном случае алгоритм перемещается в новую точку с тем же шагом. На последнем шаге работы этого алгоритма s=1 и тогда берутся все 9 точек и для них находится лучший блок.

Поиск за 3 шага: Данный алгоритм похож на предыдущий, но в отличие от него берет не 5, а 9 точек с определенным шагом s. Затем поиск переходит в новую точку (или остается в прежней, если этот блок лучший) и повторяет поиск при шаге s=s/2. И так пока s не дойдет до 1 – это будет последний шаг. Если изначально s=4, то алгоритм завершит свою работу за 3 шага.

Ортогональный поиск: Идея заключается в последовательном поиске лучшего блока сначала по горизонтали из 3-х возможных (центральный и два с боков). Затем

берется самый подходящий из них и после этого просматриваются блоки по вертикали (также 3: один выбранный и по одному сверху и снизу). Лучший блок становится центром для последующего поиска.

Поиск по одному: Этот алгоритм подобен ортогональному поиску с той лишь разницей, что сначала берутся все блоки по горизонтали (на той горизонтали где находится и начальный блок). Находится среди них самый подходящий и относительно него уже перебираются все блоки по соответствующей вертикали. Координаты лучшего найденного блока и возвращаются процедурой.

Перекрестный поиск: Сначала находятся лучшие блоки расположенные с разными шагами в форме знака x, а на последнем этапе, при s=1, форма поиска +.

Методы иерархического поиска: Иерархический поиск начинает поиск векторов движения с больших блоков. После этого большой блок разбивается на более мелкие, обычно на 4 или 9 и для этих меньших блоков векторы движения уточняются. Данный метод дает более лучшие результаты, чем рассмотренные выше, но обладает и большей вычислительной сложностью. На практике объем вычислений обычно сокращают следующими подходами:

1. Пока блок большой выбирать не лучшее приближение, а более-менее подходящее. Все равно этот вектор движения потом будет уточняться.
2. При исследовании больших блоков можно пропустить некоторые пикселы, например, оставить только четверть.

Стандарт сжатия MPEG-4 (простейший профиль)