Санкт-Петербургский государственный университет

Математико-механический факультет

кафедра системного программирования

Кудасов Федор Сергеевич

Вэйвлетное сжатие изображений без потерь

Дипломная работа

Допущен к защите

Зав. кафедрой:

д. ф.-м. н., профессор Терехов Андрей Николаевич

Научный руководитель:

зав. кафедрой параллельных алгоритмов,

д. ф.-м. н., профессор Демьянович Юрий Казимирович

Рецензент:

к. ф.-м. н., доцент Евдокимова Татьяна Олеговна

Санкт-Петербург

2010 г.

St. Petersburg State University

Faculty of Mathematics and Mechanics

Chair of Software Engineering

Kudasov Fedor Sergeevich

Wavelet lossless image compression

Graduate paper

admitted to proof

Head of the chair:

Dr. of Phys. and Math. Sci., professor Terehov Andrey Nikolaevich

Scientific advisor:

Head of the chair of parallel algorithms,

Dr. of Phys. and Math. Sci., professor Demjanovich Yury Kazimirovich

Reviewer:

Cand. of Phys. and Math. Sci., docent Evdokimova Tatjana Olegovna

St. Petersburg

2010 г.

Содержание

Введение **4**

1. Теоретическая часть **8**

1.1. Основные определения и понятия 8

1.2. Укрупнение сетки 12

1.3. Конечная сетка 17

2. Практическая реализация **21**

2.1. Кодирование сетки 21

2.2. Вэйвлетное сжатие 24

2.3. Программная реализация 28

2.4. Примеры кода 31

3. Заключение **51**

4. Дальнейшее развитие **53**

Список литературы **54**

Введение

Вэйвлеты — математические функции, использующиеся для анализа различных частотных компонент данных. Их особенностью является то, что целое семейство функций получается из одной сдвигом и растяжением (сжатием) по оси времени. Своим названием обязаны тем, что большая их часть похожа на разного вида волны.

Основой теории вэйвлетов является вэйвлетное преобразование [9]. Суть его заключается в следующем: на вход поступает набор коэффициентов разложения сигнала по базисным сплайнам, а на выходе мы получаем уже 2 набора. Первый из них также состоит из коэффициентов разложения основной части сигнала, но в менее точном виде (за счет уменьшения числа коэффициентов). Второй же набор представлляет собой поправку, используя которую можно привести первый набор обратно к исходному разложению сигнала.

Первые работы по теории вэйвлетов появились еще в начале XX века. Первое дискретное вэйвлет-преобразование было придумано венгерским математиком Альфредом Хааром в далеком 1910 году. Оно представляло собой простейшую группировку 2^n чисел по парам и преобразование в разности и суммы этих чисел [2]. Далее группировка чисел происходила рекурсивно уже с суммами. Однако, данное исследование не нашло применения в то время и было надолго забыто.

Интерес к теории вэйвлетов возрос в 80-х годах XX века. Гуппилауд, Гроссман и Морле в 1982 сформулировали то, что сейчас известно как непрерывное вэйвлет-преобразование. В 1988 бельгийский математик Ингрид Добеши вывела набор дискретных вэйвлет-преобразований,

который на данный момент является самым распространенным. За его основу берутся рекурентные соотношения для вычисления все более и более точных выборок неявно заданной функции материнского вэйвлета. А в 1989 году Малл публикует работу, в которой предлагает идею кратно-масштабного метода.

После этого результаты исследований начинают применяться во многих областях. В первую очередь — в обработке сигналов, где вэйвлетное преобразование начинает использоваться как альтернатива преобразованию Фурье. В некоторых случаях ДВП показывает значительно лучшие результаты, чем ДПФ. Хорошим примером является метод сжатия jpeg. Этот алгоритм сжатия дорабатывается, в нем заменяется дискретное косинусное преобразование на вэйвлетное, что, в свою очередь, позволяет избавиться от решетки 8 на 8 пикселей, характерной для jpeg. Новый формат получает название jpeg2000 и в дополнении к новым улучшениям не теряет возможности прогрессивного сжатия. Параллельно с этим в широко известном пакете компьютерной математики Mathcad появляются инструментальные средства по вэйвлетам. В настоящее время наблюдается повсеместное внедрение наработок теории вэйвелетов в областях, связанных с обработкой и анализом сигналов, будь то электрокардиограммы или анализ ДНК в медицине, или динамика солнечной активности в астрономии.

Вэйвлетное сжатие пусть не широко, но уже применяется в сжатии изображений (например, вышеупомянутый алгоритм jpeg2000). Также опубликованы (в основном в сети) статьи, которые описывают другие, но тоже сравнительно хорошие алгоритмы сжатия. Но эти алгоритмы сжимают изображения с потерями. Вэйвлетное преобразование позволяет

производить декомпозицию (процесс разложения исходного потока данных на два других — основной и вэйвлетный) и реконструкцию (процесс обратный декомпозиции) таким образом, что восстановленное изображение не будет ничем отличаться от исходного. И этой особенностью хочется воспользоваться. Конечно, очевидно, что восстановление происходит полностью лишь тогда, когда мы не теряем никаких данных; отсюда можно сделать вывод о том, что в этом случае мы и не получим сжатия. Однако, это не совсем верное утверждение. Как уже было сказано, теория вейвлетов в первую очередь широко применяется в области обработки сигнала и там зачастую удается сильно сжать сигнал практически без потерь за счет очень близкой аппроксимации базисными сплайнами с соответсвующими коэффициентами. Аналогично можно поступить и с изображениями. Стоит сразу уточнить, что хорошая аппроксимация будет достигаться только в областях, где яркость цвета слабо и монотонно меняется. Таким образом, хочется провести параллель между сигналом и информацией, кодирующей изображение, и так же хорошо эту информацию сжать за счет выбора областей сжатия. Выбор областей должен быть произведен таким образом, чтобы вэйвлетный поток либо оказался бы настолько малым, что мог бы быть безболезненно отброшен, и его отсутствие не повлияло бы на качество восстановленного изображения, либо поток должен быть хорошо сжимаемым, то есть, возможно, иметь величины, занимающие мало место в компьютерной памяти, либо представляться ограниченным набором этих величин для успешного его хэширования. Произвольный выбор областей дает нам еще одно преимущество вэйвлетного проебразования — возможность выбирать неравномерную сетку, на которой задан наш поток (закодированное яркостями пикселей изображение). Именно

неравномерная сетка позволяет сильно сжимать изображение там, где от пикселя к пикселю яркость меняется слабо, и слабо сжимать (для сохранения качества) изображение там, где идет быстрая смена яркостей от пикселя к пикселю.

Данная работа состоит из трех частей. В первой части содержатся теоретические выкладки, на основе которых происходит вэвлетное сжатие. В частности, вводятся используемые полиномиальные сплайны, биортогональные системы для определения коэффициентов, выводятся общие формулы декомпозиции и реконструкции, а также их частные случаи. Во второй части содержится формальное описание действий алгоритма по сжатию изображения. В третьей и четвертой частях подводятся итоги проделанной работы, представляются некоторые результаты, намечаются перспективы дальнейших исследований и обсуждается возможность модернизации данного подхода для улучшения результатов сжатия и для увеличения конкурентноспособности алгоритма.

1. Теоретическая часть

1.1. Основные определения и понятия

На интервале (α,β) вещественной оси рассмотрим сетку

$$X: \quad \ldots < x_{-1} < x_0 < x_1 < \ldots, \qquad \alpha \stackrel{\text{def}}{=} \lim_{j\to-\infty} x_j, \qquad \beta \stackrel{\text{def}}{=} \lim_{j\to+\infty} x_j.$$

Введем трехкомпонентную вектор-функцию (столбец) $\varphi : (\alpha,\beta) \mapsto \mathbb{R}^3$ класса $C^2(\alpha,\beta)$, вронскиан из компонент которой при $t \in (\alpha,\beta)$ равномерно отделен от нуля:

$$|\det(\varphi(t), \varphi\,'(t), \varphi\,''(t))| \geq c > 0. \tag{1.1}$$

Полагая $\varphi_k \stackrel{\text{def}}{=} \varphi(x_k)$, $\varphi'_k \stackrel{\text{def}}{=} \varphi'(x_k)$, $\varphi''_k \stackrel{\text{def}}{=} \varphi''(x_k)$, рассмотрим символический определитель

$$\widehat{a}(\varphi_{j+1}, \varphi'_{j+1}, \varphi_{j+2}, \varphi'_{j+2}) \stackrel{\text{def}}{=} \det \begin{pmatrix} \varphi_{j+1} & \varphi'_{j+1} \\ \det(\varphi_{j+2}, \varphi'_{j+2}, \varphi_{j+1}) & \det(\varphi_{j+2}, \varphi'_{j+2}, \varphi'_{j+1}) \end{pmatrix}.$$

Координатные $B_\varphi(X)$-сплайны второго порядка $\omega_j(t)$ определяются аппроксимационными соотношениями

$$a_{k-2}\omega_{k-2}(t) + a_{k-1}\omega_{k-1}(t) + a_k\omega_k(t) = \varphi(t) \quad \forall t \in (x_k, x_{k+1}), \quad \forall k \in \mathbb{Z}, \tag{1.2}$$

при условиях supp $\omega_j \subset [x_j, x_{j+3}]$, $a_j \stackrel{\text{def}}{=} \widehat{a}(\varphi_{j+1}, \varphi'_{j+1}, \varphi_{j+2}, \varphi'_{j+2})$. Для достаточно мелкой сетки X квадратные матрицы третьего порядка $A_j \stackrel{\text{def}}{=} (a_{j-2}, a_{j-1}, a_j)$ неособенные, и поэтому система (1.2) однозначно разрешима, а получаемое решение ω_j лежит в пространстве $C^1(\alpha,\beta)$.

Линейная оболочка $\mathbb{S}_{(\alpha,\beta)}(X,\varphi)$ функций ω_j

$$\mathbb{S}_{(\alpha,\beta)}(X,\varphi) \stackrel{\text{def}}{=} \left\{ u \mid u = \sum_j c_j\omega_j \quad \forall c_j \in \mathbb{R}^1,\ j \in \mathbb{Z} \right\}$$

представляет собой бесконечномерное пространство B_φ-сплайнов второго порядка на интервале (α, β); при этом $\mathbb{S}_{(\alpha,\beta)}(X,\varphi) \subset C^1(\alpha,\beta)$.

Если $\varphi(t) \overset{\text{def}}{=} (1, t, t^2)^T$, то $\mathbb{S}_{(\alpha,\beta)}(X,\varphi)$ оказывается пространством сплайнов второй степени с минимальным дефектом [3], [8]

$$\omega_j(t) = \frac{\det(a_{j-2}, a_{j-1}, \varphi(t))}{\det(a_{j-2}, a_{j-1}, a_j)} \text{ при } t \in [x_j, x_{j+1}), \tag{1.3}$$

$$\omega_j(t) = \frac{\det(a_{j-1}, \varphi(t), a_{j+1})}{\det(a_{j-2}, a_{j-1}, a_j)} \text{ при } t \in [x_{j+1}, x_{j+2}), \tag{1.4}$$

$$\omega_j(t) = \frac{\det(\varphi(t), a_{j+1}, a_{j+2})}{\det(a_{j-2}, a_{j-1}, a_j)} \text{ при } t \in [x_{j+2}, x_{j+3}]. \tag{1.5}$$

Нетрудно видеть, что для вектора

$$d_k = \left(\det \begin{pmatrix} [a_{k-2}]_1 & [a_{k-1}]_1 \\ [a_{k-2}]_2 & [a_{k-1}]_2 \end{pmatrix}, -\det \begin{pmatrix} [a_{k-2}]_0 & [a_{k-1}]_0 \\ [a_{k-2}]_2 & [a_{k-1}]_2 \end{pmatrix}, \det \begin{pmatrix} [a_{k-2}]_0 & [a_{k-1}]_0 \\ [a_{k-2}]_1 & [a_{k-1}]_1 \end{pmatrix} \right)^T$$

справедливы следующие соотношения:

$$d_j^T a_{j-3} \neq 0,\ d_j^T a_{j-2} = d_j^T a_{j-1} = 0,\ d_j^T a_j \neq 0, \tag{1.6}$$

(первое и последнее соотношения выполнены в силу неособенности всех A_j, а второе и третье очевидны); таким образом, цепочка векторов d_j локально ортогональна цепочке a_j.

Умножая (1.2) на d_k^T слева, ввиду (1.6), находим

$$\omega_k(t) = \frac{d_k^T \varphi(t)}{d_k^T a_k}.$$

При умножении (1.2) слева на ${d_{k-1}}^T$ имеем

$$\omega_{k-1}(t) = \frac{{d_{k-1}}^T\varphi(t)}{{d_{k-1}}^T a_{k-1}} - \frac{{d_{k-1}}^T a_k}{{d_{k-1}}^T a_{k-1}} \cdot \frac{{d_{k-1}}^T\varphi(t)}{{d_k}^T a_k}.$$

Наконец, умножим слева на d_{k-2} обе части соотношения (1.2); используя предыдущие формулы, получаем

$$\omega_{k-2}(t) = \frac{{d_{k-2}}^T\varphi(t)}{{d_{k-2}}^T a_{k-2}} - \frac{{d_{k-2}}^T a_{k-1}}{{d_{k-2}}^T a_{k-2}} \cdot \left(\frac{{d_{k-1}}^T\varphi(t)}{{d_{k-1}}^T a_{k-1}} - \frac{{d_{k-1}}^T a_k}{{d_{k-1}}^T a_{k-1}} \cdot \frac{{d_k}^T\varphi(t)}{{d_k}^T a_k}\right) -$$

$$-\frac{{d_{k-2}}^T a_{k-1}}{{d_{k-2}}^T a_{k-2}} \cdot \frac{{d_k}^T\varphi(t)}{{d_k}^T a_k}$$

Заметим, что отыскание функций ω_j можно проводить в последовательности ω_{k-2}, ω_{k-1}, ω_k; при этом получается другая форма представления этих функций. Умножение обеих частей соотношения (1.2) на ${d_{k+1}}^T$ дает упрощения для ω_{k-2}:

$$\omega_{k-2}(t) = \frac{d_{k+1}^T\varphi(t)}{d_{k+1}^T a_{k-2}}.$$

Для функций ω_j справедливы представления

$$\omega_j(t) = \frac{d_j^T\varphi(t)}{d_j^T a_j} \text{ при } t \in [x_j, x_{j+1}),$$

$$\omega_j(t) = \frac{{d_j}^T\varphi(t)}{{d_j}^T a_j} - \frac{{d_j}^T a_{j+1}}{{d_j}^T a_j} \cdot \frac{{d_{j+1}}^T\varphi(t)}{dj+1k^T a_{j+1}} \text{ при } t \in [x_{j+1}, x_{j+2}),$$

$$\omega_j(t) = \frac{d_{j+3}^T\varphi(t)}{d_{j+3}^T a_j} \text{ при } t \in [x_{j+2}, x_{j+3}].$$

Из бесконечной сетки X можно выделить конечную X_N:

$$X_N : x_0 < x_1 < \ldots < x_{N-1} < x_N;$$

а из полной бесконечной цепочки A — конечную цепочку векторов A_N :

$$A_N = \{a_{-2}, a_{-1}, ..., a_{N-1}\}.$$

Рассмотрим столбец b_s^T, определенный следующим отношением:

$$b_s^T x = \det(\varphi_s, {\varphi'}_s, x);$$

цепочка b_s^T локально ортогональна цепочке векторов a_j

$$b_j^T a_j \neq 0,\ b_j^T a_{j-2} = b_j^T a_{j-1} = 0\ \forall j \in \mathbb{Z}.$$

В случае $\varphi(t) = (1, t, t^2)^T$, получаем:

$$a_j = 2(x_{j+1} - x_{j+2}) \cdot \begin{pmatrix} 1 \\ \frac{x_{j+1}+x_{j+2}}{2} \\ x_{j+1}x_{j+2} \end{pmatrix},$$

$$b_s^T a_j = 2(x_{j+1} - x_s)(x_{j+1} - x_{j+2})(x_{j+2} - x_s),$$

$$b_s^T \varphi(t) = \det(\varphi_s, \varphi_s, \varphi(t)) = (t - x_s)^2,$$

$$\omega_j(t) = \frac{(t - x_j)^2}{2(x_{j+1} - x_j)(x_{j+1} - x_{j+2})(x_{j+2} - x_j)} \text{ при } t \in [x_j, x_{j+1}),$$

$$\omega_j(t) = \frac{t^2(x_j + x_{j+1} - x_{j+2} - x_{j+3}) + 2t(x_{j+2}x_{j+3} - x_j x_{j+1})}{2(x_{j+1} - x_{j+3})(x_{j+2} - x_j)(x_{j+2} - x_{j+1})(x_{j+3} - x_{j+1})} +$$

$$+ \frac{x_j x_{j+2}(x_{j+1} - x_{j+3}) + x_{j+1}x_{j+3}(x_j - x_{j+2})}{2(x_{j+1} - x_{j+3})(x_{j+2} - x_j)(x_{j+2} - x_{j+1})(x_{j+3} - x_{j+1})} \text{ при } t \in [x_{j+1}, x_{j+2}),$$

$$\omega_j(t) = \frac{(t - x_{j+3})^2}{(x_{j+1} - x_{j+3})(x_{j+1} - x_{j+2})(x_{j+2} - x_{j+3})} \text{ при } t \in [x_{j+2}, x_{j+3}],$$

$$\omega_j(t) = 0 \text{ при } t \notin [x_j, x_{j+3}].$$

1.2. Укрупнение сетки

Для фиксированного $k \in \mathbb{Z}$ положим

$$\widetilde{x}_j \overset{\text{def}}{=} x_j \text{ при } j \leq k, \text{ и } \widetilde{x}_j \overset{\text{def}}{=} x_{j+1} \text{ при } j \geq k+1, \tag{2.1}$$

и рассмотрим новую сетку $\widetilde{X}_k: \quad \ldots < \widetilde{x}_{-1} < \widetilde{x}_0 < \widetilde{x}_1 < \ldots$. Аналогично предыдущему определим функции $\widetilde{\omega}_j$ для сетки $\widetilde{X}_k$. Система функций $\{\widetilde{\omega}_j\}_{j\in\mathbb{Z}}$, несомненно зависит от k; однако, здесь и в дальнейшем для краткости зависимость рассматриваемых объектов от k отмечаем не во всех случаях.

Очевидно, что для $j \in \mathbb{Z}\backslash\{k-2, k-1, k\}$ и $t \in (\alpha, \beta)$ сплайны $\widetilde{\omega}_j$ совпадают с рассмотренными ранее сплайнами:

$$\widetilde{\omega}_j(t) \equiv \omega_j(t) \quad \forall j \leq k-3; \qquad \widetilde{\omega}_j(t) \equiv \omega_{j+1}(t) \quad \forall j \geq k+1.$$

Введем вектор-столбцы ω, $\widetilde{\omega}$ определенные формулами

$$\omega \overset{\text{def}}{=} (\ldots, \omega_{-1}, \omega_0, \omega_1, \ldots)^T, \quad \widetilde{\omega} \overset{\text{def}}{=} (\ldots, \widetilde{\omega}_{-1}, \widetilde{\omega}_0, \widetilde{\omega}_1, \ldots)^T,$$

и матрицу $\mathfrak{P}_k = (\mathfrak{p}_{ij})_{i,j\in\mathbb{Z}}$, где $\mathfrak{p}_{ij}$ находятся из соотношений $\widetilde{\omega}_i = \sum_j \mathfrak{p}_{ij}\omega_j$. Транспонирование $\mathfrak{P}_k$ дает матрицу $\mathfrak{P}_k^T$,

$$\mathfrak{P}_k^T \overset{\text{def}}{=} \begin{array}{c} \\ \\ {\scriptstyle k-5} \\ {\scriptstyle k-4} \\ {\scriptstyle k-3} \\ {\scriptstyle k-2} \\ {\scriptstyle k-1} \\ {\scriptstyle k} \\ {\scriptstyle k+1} \\ {\scriptstyle k+2} \\ {\scriptstyle k+3} \\ \\ \end{array} \begin{array}{c} \begin{array}{cccccccccccc} {\scriptstyle \ldots} & {\scriptstyle k-5} & {\scriptstyle k-4} & {\scriptstyle k-3} & {\scriptstyle k-2} & {\scriptstyle k-1} & {\scriptstyle k} & {\scriptstyle k+1} & {\scriptstyle k+2} & {\scriptstyle k+3} & {\scriptstyle k+4} & {\scriptstyle \ldots} \end{array} \\ \begin{pmatrix} \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ldots \\ \ldots & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ldots \\ \ldots & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ldots \\ \ldots & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \ldots \\ \ldots & 0 & 0 & 0 & \mathfrak{p}_{k-2,k-1} & \mathfrak{p}_{k-1,k-1} & 0 & 0 & 0 & 0 & 0 & \ldots \\ \ldots & 0 & 0 & 0 & 0 & \mathfrak{p}_{k-1,k} & \mathfrak{p}_{k,k} & 0 & 0 & 0 & 0 & \ldots \\ \ldots & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & \ldots \\ \ldots & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & \ldots \\ \ldots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \end{pmatrix} \end{array};$$

ее элементами служат числа

$$p_{k-2,k-1} \stackrel{\text{def}}{=} \frac{b_{k+2}^T a_{k-1}}{b_{k+2}^T a_{k-2}}, \quad p_{k-1,k-1} \stackrel{\text{def}}{=} \frac{b_{k-1}^T a_{k-1}}{b_{k-1}^T \widetilde{a}_{k-1}},$$

$$p_{k-1,k} \stackrel{\text{def}}{=} \frac{b_{k+3}^T a_k}{b_{k+3}^T \widetilde{a}_{k-1}}, \quad p_{k,k} \stackrel{\text{def}}{=} \frac{b_k^T a_k}{b_k^T a_{k+1}}. \tag{2.2}$$

В случае, когда $\phi(t) = (1, t, t^2)^T$, получаем

$$p_{k-2,k-1} = \frac{x_{k+1} - x_k}{x_k - x_{k-1}} \cdot \frac{x_{k+2} - x_{k+1}}{x_{k+2} - x_{k-1}},$$

$$p_{k-1,k-1} = \frac{x_{k+1} - x_k}{x_{k+2} - x_k} \cdot \frac{x_{k+3} - x_{k+1}}{x_{k+3} - x_k},$$

$$p_{k-1,k} = \frac{x_{k+2} - x_{k+1}}{x_{k+2} - x_k} \cdot \frac{x_{k+3} - x_{k+1}}{x_{k+3} - x_k},$$

$$p_{k,k} = \frac{x_{k+1} - x_k}{x_{k+3} - x_k} \cdot \frac{x_{k+2} - x_{k+1}}{x_{k+3} - x_{k+2}}.$$

Кратко преобразование можно записать так:

$$\widetilde{\omega}(t) = \mathfrak{P}_k \omega(t). \tag{2.3}$$

Введем (бесконечные) матрицы декомпозиции $\mathfrak{Q}_{k\langle\sigma\rangle}$ вида, где $\widetilde{g}_{i\langle\sigma\rangle}$ - биортогональная система [6]:

$$\mathfrak{Q}_{k\langle\sigma\rangle} \stackrel{\text{def}}{=} (\mathfrak{q}_{ij\langle\sigma\rangle}),\ \mathfrak{q}_{ij\langle\sigma\rangle} \stackrel{\text{def}}{=} \langle \widetilde{g}_{i\langle\sigma\rangle}, \omega_j \rangle, \quad i, j \in \mathbb{Z},$$

и прямоугольную матрицу $\mathfrak{Q}_{N,k}$ размером $(N+1) \times (N+2)$:

$$\mathfrak{Q}_{N,k} \stackrel{\text{def}}{=} (\mathfrak{q}_{ij}), \qquad \mathfrak{q}_{ij} \stackrel{\text{def}}{=} \langle \widetilde{g}_i, \omega_j \rangle, \quad i \in J_{N-2} \quad j \in J_{N-1},\ \text{где } J_k = \{-2, -1, 0, ..., k\}.$$

Справедливы следующие утверждения для коэффициентов матрицы декомпозиции:

Для чисел $\mathfrak{q}_{ij\langle\sigma\rangle} \stackrel{\text{def}}{=} \langle \widetilde{g}_{i\langle\sigma\rangle}, \omega_j \rangle$ *в случае* $\sigma = 0$, *выполнены соотношения*

$$\mathfrak{q}_{ij\langle 0\rangle} = \delta_{i,j} \quad \text{\textit{при}}\ \forall i \in \mathbb{Z},\ j \le k-3,$$

$$\mathfrak{q}_{ij\langle 0\rangle} = \delta_{i+1,j} \quad \text{\textit{при}}\ \forall i \in \mathbb{Z},\ j \ge k+2,$$

$$\mathfrak{q}_{i,k-2\langle 0\rangle} = 0 \quad \text{\textit{при}}\ (i \le k-3) \vee (i \ge k+1), \qquad \mathfrak{q}_{k-2,k-2\langle 0\rangle} = 1,$$

$$\mathfrak{q}_{k-1,k-2\langle 0\rangle} = \frac{b_{k-2}^T \widetilde{a}_{k-1}}{b_{k-2}^T a_{k-2}} - \frac{b_{k-2}^T a_{k-1}}{b_{k-2}^T a_{k-2}} \cdot \frac{b_{k-1}^T \widetilde{a}_{k-1}}{b_{k-1}^T a_{k-1}},$$

$$\mathfrak{q}_{k,k-2\langle 0\rangle} = \frac{b_{k+1}^T a_{k+1}}{b_{k+1}^T a_{k-2}},$$

$$\mathfrak{q}_{i,k-1\langle 0\rangle} = 0 \quad \text{\textit{при}}\ (i \le k-2) \vee (i \ge k+1),$$

$$\mathfrak{q}_{k-1,k-1\langle 0\rangle} = \frac{b_{k-1}^T \widetilde{a}_{k-1}}{b_{k-1}^T a_{k-1}},$$

$$\mathfrak{q}_{k,k-1\langle 0\rangle} = \frac{b_{k-1}^T a_{k+1}}{b_{k-1}^T a_{k-1}} - \frac{b_{k-1}^T a_k}{b_{k-1}^T a_{k-1}} \cdot \frac{b_k^T a_{k+1}}{b_k^T a_k},$$

$$\mathfrak{q}_{i,k\langle 0\rangle} = 0 \quad \text{\textit{при}}\ (i \le k-1) \vee (i \ge k+2), \qquad \mathfrak{q}_{k,k\langle 0\rangle} = \frac{b_k^T a_{k+1}}{b_k^T a_k},$$

$$\mathfrak{q}_{i,k+1\langle 0\rangle} = 0 \quad \forall i \in \mathbb{Z}, \qquad \mathfrak{q}_{k+1,k\langle 0\rangle} = 0.$$

В случае $\sigma = 1$;

$$\mathfrak{q}_{ij\langle 1\rangle} = \delta_{i,j} \quad \text{при}\ \ i \le k-2 \quad \forall j \in \mathbb{Z},$$

$$\mathfrak{q}_{ij\langle 1\rangle} = \delta_{i,j-1} \quad \text{при}\ \ i \ge k \quad \forall j \in \mathbb{Z},$$

$$\mathfrak{q}_{k-1,j\langle 1\rangle} = 0 \quad \text{при}\ (j \le k-1) \vee (k \le j),$$

$$\mathfrak{q}_{k-1,k-2\langle 1\rangle} = \frac{b_{k+1}^T \widetilde{a}_{k-1}}{b_{k+1}^T a_{k-2}}, \qquad \mathfrak{q}_{k-1,k-1\langle 1\rangle} = \frac{b_{k-1}^T \widetilde{a}_{k-1}}{b_{k-1}^T a_{k-1}}.$$

В случае $\sigma = 2$;

$$\mathfrak{q}_{ij\langle 2\rangle} = \delta_{i,j} \quad \text{при}\ \ i \le k-2 \quad \forall j \in \mathbb{Z},$$

$$\mathfrak{q}_{ij\langle 2\rangle} = \delta_{i,j-1} \quad \text{при} \;\; i \geq k \quad \forall j \in \mathbb{Z},$$

$$\mathfrak{q}_{k-1,j\langle 2\rangle} = 0 \quad \text{при} \;\; (j \leq k-1) \vee (k+2 \leq j),$$

$$\mathfrak{q}_{k-1,k\langle 2\rangle} = \frac{b_{k+3}^T \widetilde{a}_{k-1}}{b_{k+3}^T a_k}, \qquad \mathfrak{q}_{k-1,k+1\langle 2\rangle} = \frac{b_{k+1}^T \widetilde{a}_{k-1}}{b_{k+1}^T a_{k+1}}.$$

Матрицы $\mathfrak{Q}_{k\langle\sigma\rangle}$ являются левыми обратными для матрицы $\mathfrak{P}_k^T$, то есть

$$\mathfrak{Q}_{k\langle\sigma\rangle} \mathfrak{P}_k^T = I, \quad \sigma \in \{0,1,2\}.$$

Пространство, натянутое на линейную оболочку функций ω_j, является прямой суммой пространства, натянутого на линейную оболочку функций $\widetilde{\omega}_j$ и, так называемого, вэйвлетного пространства. Таким образом, функцию u можно разложить по базисным сплайнам следующим образом:

$$u = \sum_j c_j \omega_j,$$

$$u = \sum_j a_{i\langle\sigma\rangle} \widetilde{\omega}_i + \sum_j b_{j\langle\sigma\rangle} \omega_j,$$

где числа c_i, $a_{i\langle\sigma\rangle}$ и $b_{i\langle\sigma\rangle}$ — это коэффициенты соответствующих разложений [7].

Из предыдущих формул имеем $\sum_j c_j \omega_j = \sum_i a_{i\langle\sigma\rangle} \sum_j \mathfrak{p}_{i,j} \omega_j + \sum_j b_{j\langle\sigma\rangle} \omega_j$, откуда ввиду линейной независимости системы $\{\omega_j\}$ получаем

$$c_j = \sum_i a_{i\langle\sigma\rangle} \mathfrak{p}_{i,j} + b_{j\langle\sigma\rangle}. \tag{2.4}$$

Формулы (2.4) называются *формулами реконструкции*.

Перепишем (2.4) в виде $c_j = \sum_i \langle \widetilde{g}_{i\langle\sigma\rangle}, u \rangle \mathfrak{p}_{i,j} + b_{j\langle\sigma\rangle}$ и подставим сюда выражения для u: $c_j = \sum_i \sum_s c_s \mathfrak{q}_{is\langle\sigma\rangle} \mathfrak{p}_{i,j} + b_{j\langle\sigma\rangle}$, отсюда

$$b_{j\langle\sigma\rangle} = c_j - \sum_i \sum_s c_s \mathfrak{q}_{is\langle\sigma\rangle} \mathfrak{p}_{i,j},$$

$$a_{j\langle\sigma\rangle} = \sum_s c_s \langle \mathfrak{q}_{js\langle\sigma\rangle}, \omega_s \rangle. \tag{2.5}$$

Формулы (2.5) называются *формулами декомпозиции.*

Перепишем (2.4), (2.5) с учетом того, что у нас конечный интервал:

$$u = \sum_{j \in J_{N-1}} c_j \omega_j,$$

$$u = \sum_{i \in J_{N-2}} a_i \widetilde{\omega}_i + \sum_{j \in J_{N-1}} b_j \omega_j,$$

где

$$a_i \stackrel{\text{def}}{=} \langle \widetilde{g}_i, u \rangle, \quad b_j,\ c_j\ \in \mathbb{R}^1.$$

Далее имеем

$$\sum_{j \in J_{N-1}} c_j \omega_j = \sum_{i \in J_{N-2}} a_i \sum_{j \in J_{N-1}} \mathfrak{p}_{i,j} \omega_j + \sum_{j \in J_{N-1}} b_j \omega_j,$$

откуда ввиду линейной независимости системы $\{\omega_j\}_{j \in J_{N-1}}$ получаем

$$c_j = \sum_{i \in J_{N-2}} a_i \mathfrak{p}_{i,j} + b_j \quad \forall j \in J_{N-1} \Leftrightarrow$$

$$c_j = \sum_{i \in J_{N-2}} \langle \widetilde{g}_i, u \rangle \mathfrak{p}_{i,j} + b_j \qquad \forall j \in J_{N-1} \Leftrightarrow$$

$$c_j = \sum_{i \in J_{N-2}} \sum_{s \in J_{N-1}} c_s \mathfrak{q}_{i,s} \mathfrak{p}_{i,j} + b_j \qquad \forall j \in J_{N-1},$$

где $\mathfrak{q}_{i,s} \stackrel{\text{def}}{=} \langle \widetilde{g}_i, \omega_s \rangle$; отсюда

$$b_j = c_j - \sum_{i \in J_{N-2}} \sum_{s \in J_{N-1}} c_s \mathfrak{q}_{i,s} \mathfrak{p}_{i,j} \qquad \forall j \in J_{N-1},$$

$$a_i = \sum_{s \in J_{N-1}} c_s \mathfrak{q}_{i,s} \qquad \forall i \in J_{N-2}.$$

1.3. Конечная сетка

Преобразуем матрицы реконструкции из работы [5] для случая конечной сетки. Разным значениям σ соответствуют разные биортогональные системы, поэтому матрицы декомпозиции имеют различный вид. В частности, при $\sigma = 0$, получаем:

$$a_i = c_i \text{ при } i \leq k-2,$$

$$\begin{pmatrix} a_{k-2} \\ a_{k-1} \\ a_k \\ a_{k+1} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ \mathfrak{q}_{k-1,k-2} & \mathfrak{q}_{k-1,k-1} & 0 & 0 & 0 \\ \mathfrak{q}_{k,k-2} & \mathfrak{q}_{k,k-1} & \mathfrak{q}_{k,k} & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} c_{k-2} \\ c_{k-1} \\ c_k \\ c_{k+1} \\ c_{k+2} \end{pmatrix}, \tag{2.6}$$

$$a_i = c_{i+1} \text{ при } i \geq k+1,$$

$$b_{k+1} = c_{k+1} - (\mathfrak{q}_{k,k-2} \cdot c_{k-2} + \mathfrak{q}_{k,k-1} \cdot c_{k-1} + \mathfrak{q}_{k,k} \cdot c_k),$$

где коэффициенты $\mathfrak{q}_{i,j}$ имеют следующий вид

$$\mathfrak{q}_{k-1,k-2} = -\frac{x_{k+2} - x_k}{x_k - x_{k-1}} \cdot \frac{x_{k+2} - x_{k+1}}{x_{k+1} - x_{k-1}},$$

$$\mathfrak{q}_{k-1,k-1} = \frac{x_{k+2} - x_k}{x_{k+1} - x_k} \cdot \frac{x_{k+2} - x_{k-1}}{x_{k+1} - x_{k-1}},$$

$$\mathfrak{q}_{k,k-2} = \frac{x_{k+2} - x_{k+1}}{x_{k+1} - x_{k-1}} \cdot \frac{x_{k+3} - x_{k+2}}{x_k - x_{k-1}} \cdot \frac{x_{k+3} - x_{k+1}}{x_{k+1} - x_k},$$

$$\mathfrak{q}_{k,k-1} = -\frac{x_{k+2} - x_{k-1}}{x_{k+1} - x_k} \cdot \frac{x_{k+3} - x_{k+2}}{x_{k+1} - x_{k-1}} \cdot \frac{x_{k+3} - x_{k+1}}{x_{k+1} - x_k},$$

$$\mathfrak{q}_{k,k} = \frac{x_{k+3} - x_{k+2}}{x_{k+1} - x_k} \cdot \frac{x_{k+3} - x_k}{x_{k+2} - x_{k+1}}.$$

В случае $\sigma = 1$ получаем:

$$a_i = c_i \text{ при } i \leq k-2,$$

$$\begin{pmatrix} a_{k-2} \\ a_{k-1} \\ a_k \\ a_{k+1} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ \mathfrak{q}_{k-1,k-2} & \mathfrak{q}_{k-1,k-1} & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} c_{k-2} \\ c_{k-1} \\ c_k \\ c_{k+1} \\ c_{k+2} \end{pmatrix}, \qquad (2.7)$$

$$a_i = c_{i+1} \text{ при } i \geq k+1,$$

$$b_k = c_k - (c_{<1>} \cdot c_{k-2} + d_{<1>} \cdot c_{k-1} + e_{<1>} \cdot c_{k+1}),$$

$$c_{<1>} = \mathfrak{p}_{k-1,k} \cdot \mathfrak{q}_{k-1,k-2},\ d_{<1>} = \mathfrak{p}_{k-1,k} \cdot \mathfrak{q}_{k-1,k-1},\ e_{<1>} = \mathfrak{p}_{k,k},$$

где

$$\mathfrak{q}_{k-1,k-2} = \frac{x_{k+2} - x_{k+1}}{x_{k+1} - x_{k-1}} \cdot \frac{x_k - x_{k+2}}{x_k - x_{k-1}},$$

$$\mathfrak{q}_{k-1,k-1} = \frac{x_{k+2} - x_k}{x_{k+1} - x_k} \cdot \frac{x_{k+2} - x_{k-1}}{x_{k+1} - x_{k-1}},$$

а $\mathfrak{p}_{i,j}$ определяются формулами (2.2).

В случае $\sigma = 2$ находим:

$$a_i = c_i \text{ при } i \leq k-2,$$

$$\begin{pmatrix} a_{k-2} \\ a_{k-1} \\ a_k \\ a_{k+1} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathfrak{q}_{k-1,k} & \mathfrak{q}_{k-1,k+1} & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} c_{k-2} \\ c_{k-1} \\ c_k \\ c_{k+1} \\ c_{k+2} \end{pmatrix} \qquad (2.8)$$

$$a_i = c_{i+1} \text{ при } i \geq k+1,$$

$$b_{k-1} = c_{k+1} - (a_{<2>} \cdot c_{k-2} + b_{<2>} \cdot c_k + c_{<2>} \cdot c_{k+1}),$$

$$a_{<2>} = \mathfrak{p}_{k-2,k-1},$$

$$b_{<2>} = \mathfrak{p}_{k-1,k-2} \cdot \mathfrak{q}_{k-1,k},$$

$$c_{<2>} = \mathfrak{p}_{k-1,k-2} \cdot \mathfrak{q}_{k-1,k+1},$$

где коэффициенты $\mathfrak{q}_{i,j}$ вычисляются по формулам:

$$\mathfrak{q}_{k-1,k} = \frac{x_{k+3} - x_k}{x_{k+3} - x_{k+1}} \cdot \frac{x_{k+2} - x_k}{x_{k+2} - x_{k+1}},$$

$$\mathfrak{q}_{k-1,k+1} = \frac{x_{k+2} - x_k}{x_{k+2} - x_{k+3}} \cdot \frac{x_{k+1} - x_k}{x_{k+3} - x_{k+1}},$$

а $\mathfrak{p}_{i,j}$ определяются формулами (2.2).

И наконец, находим формулы реконструкции:

$$\begin{pmatrix} c_{k-2} \\ c_{k-1} \\ c_k \\ c_{k+1} \\ c_{k+2} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ \mathfrak{p}_{k-2,k-1} & \mathfrak{p}_{k-1,k-1} & 0 & 0 \\ 0 & \mathfrak{p}_{k-1,k} & \mathfrak{p}_{k,k} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} a_{k-2} \\ a_{k-1} \\ a_k \\ a_{k+1} \end{pmatrix} + \begin{pmatrix} 0 \\ b_{k-1} \\ b_k \\ b_{k+1} \\ 0 \end{pmatrix}.$$

При декомпозиции преобразуются не только коэффициенты, но и часть базисных сплайнов. Выпишем в явном виде формулы для их преобразования [4]:

$$\widetilde{\omega_{k-2}} = \omega_{k-2} + \frac{x_{k+1} - x_k}{x_k - x_{k-1}} \cdot \frac{x_{k+2} - x_{k+1}}{x_{k+2} - x_{k-1}} \cdot \omega_{k-1},$$

$$\widetilde{\omega_{k-1}} = \frac{x_{k+1} - x_k}{x_{k+2} - x_k} \cdot \frac{x_{k+3} - x_{k+1}}{x_{k+3} - x_k} \cdot \omega_{k-1} + \frac{x_{k+2} - x_{k+1}}{x_{k+2} - x_k} \cdot \frac{x_{k+3} - x_{k+1}}{x_{k+3} - x_k} \cdot \omega_k,$$

$$\widetilde{\omega_k} = \frac{x_{k+1} - x_k}{x_{k+3} - x_k} \cdot \frac{x_{k+2} - x_{k+1}}{x_{k+3} - x_{k+2}} \cdot \omega_k + \omega_{k+1}.$$

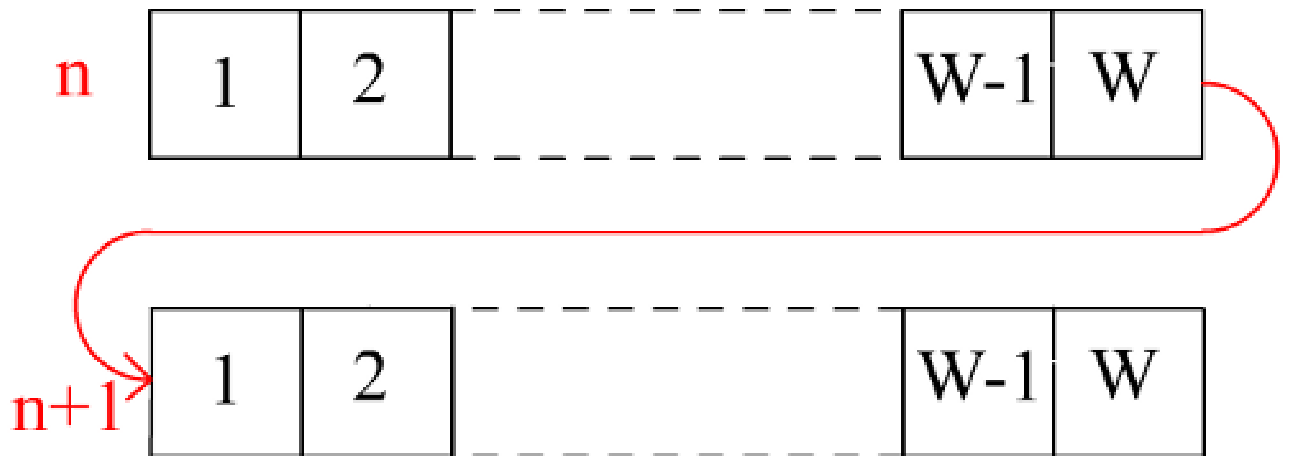
2. Практическая реализация

В этой части будет подробно описан процесс сжатия изображения. Попутно будет представлено несколько способов интерпретации изображения как одномерного потока данных (хотя, вообще говоря, изображение двумерно).

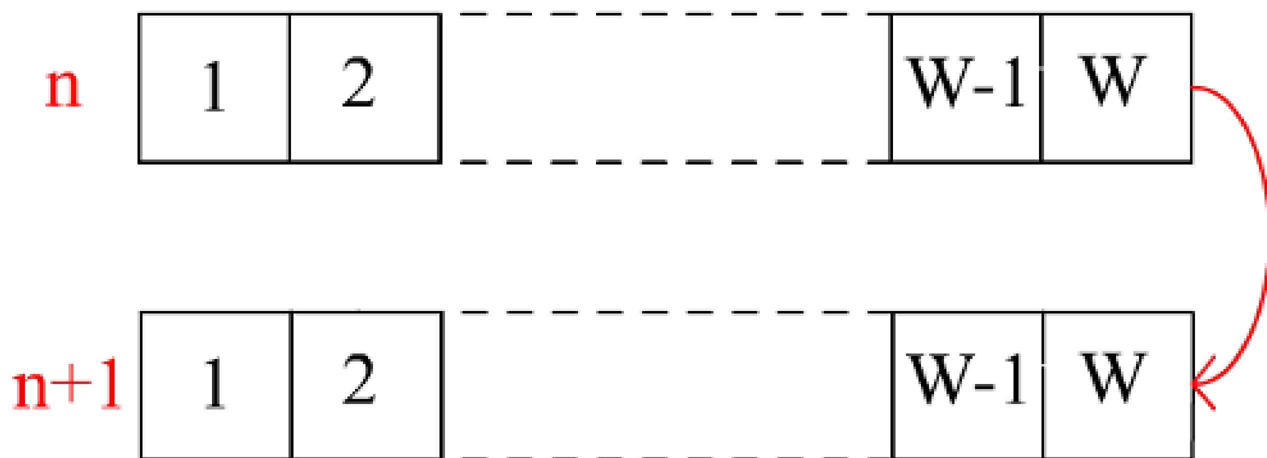
2.1. Кодирование сетки

На вход программе поступает изображение в формате BMP — самом простом формате без какого-либо сжатия. Обозначим за W — ширину изображения, а за H — высоту. Их мы записываем в любом случае, каждый в два байта, вне зависимости от дальнейшего сжатия. Сжатие картинки начинается с кодировки сетки. Вначале прямоугольное изображение преобразуется в сплошной поток данных (яркостей пикселей). Тут возможны некоторые варианты, т. к. то, как будет создан поток данных для сжатия, несомненно, будет сильно влиять на сами данные и, соответственно, на их способность к сжимаемости. Для сжатия цвет каждого пикселя изображения разбивается на красную, синюю и зеленую составляющую и получаются три потока, каждый из которых сжимается отдельно. В процессе разработки было рассмотрено два варианта получения потока, каждый из которых начинается с разбития изображения построчно, т. е. от каждого пикселя в строке берется определенная составляющая (для примера, возьмем красную) и записывается в поток последовательного пикселя. Стоит отметить, что построчная развертка выбрана по двум причинам: во-первых это самый распространенный вид развертки, а во-вторых ширина изображения чаще больше чем его длина, таким образом, мы можем сильнее приблизить изображение к гладкой функции. После построчного разложения в первом варианте

эти построчные потоки соединялись просто последовательно, т. е. красная составляющая первого пикселя $n+1$-ой строки идет в потоке непосредственно после красной составляющей последнего пикселя n-ной строки.



Во втором варианте построчные потоки укладывались „змейкой“, т. е. красная составляющая последнего пикселя в четной $n+1$-ой строке идет сразу после красной составляющей последнего пикселя в нечетной n-ной строке, а красная составляющая первого пикселя в нечетной $n+1$-ой строке идет сразу после красной составляющей первого пикселя n-ной строки, также стоит отметить, что потоки четных строк уложены в общий поток реверсивно.



Причина рассмотрения второго варианте в том, что ввиду некоторой относительной непрерывности изображения в месте сцепления построчных потоков будет наблюдаться не такой сильный разрыв (а, вероятно, его может и вообще не быть) как при первом варианте.

Следующей задачей становится кодирование сетки. В зависимости от вида и количества пикселей сетка кодируется по-разному. Вначале алгоритм проходит по последовательности байтов потока, кодирующих яркости пикселей изображения, и преобразует данные следующим образом: создается список M — изначально пустой. Проходим по байтам, производя следующие действия: если это первый байт последовательности (потока) или не первый, но отличающийся от предыдущего, то мы записываем в список M пару чисел вида ($<$значение этого байта$>$, 1), если анализируемый байт совпадает с предыдущим, то мы увеличиваем на единицу второй элемент пары последнего элемента массива M. Второй элемент пары, как нетрудно заметить, представляет из себя счетчик байтов идущих друг за другом.

Есть несколько видов кодирования сетки:

1. Каждому байту исходной сетки мы сопоставляем один бит равный единице, если этот байт отличается от предыдущего или стоит первым, и бит равный нулю иначе. А далее записываем только те байты, которым соответствует единица. Таким образом общее число байт закодированной сетки будет в данном случае равно $|M| + W \cdot H/8$.

2. Можно просто закодировать пары имеющиеся у нас в списке M. Размер данных будет равен $|M| \cdot (1+l)$, где l — число байт, отведенное на кодирование счетчика байтов идущих подряд — число от 1 до 4 (так как разумно предполагать, что в изображении не более 2^{32} пикселей).

Сравнивая количество байт, мы выбираем способ предоставляющий нам наилучшее сжатие. Если ни один из вышеперечисленных способов не дает результат, меньший размера сигнала исходного изображения, то записываем просто сам поток. В итоговой кодировке нужно будет выделить управляющий байт, в который мы запишем выбранный вид кодировки (например в первые три бита: 0 — сжатия нет (просто поток), 1 — сжатие первого типа, $1 + l$ — сжатие второго типа с определенным l).

2.2. Вэйвлетное сжатие

Когда сетка закодирована, приступаем ко второму этапу — множественному удалению узлов, которые можно восстановить с помощью сплайновой аппроксимации с соответствующими коэффициентами. Вызвана данный возможность тем, что константа, линейная функция (отрезок прямой) или квадратичная (часть параболы) аппроксимируются рассмотренными нами в теоретической части функциями настолько хорошо (из-за того, что они состоят из фрагментов кривых второго

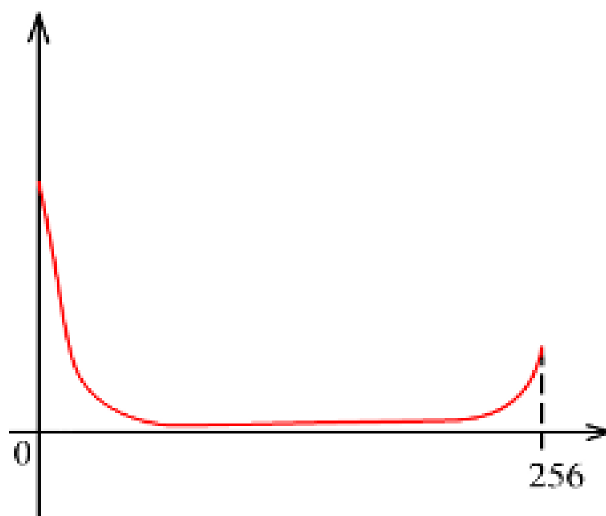
порядка), что некоторые промежуточные узлы можно выбросить и, соответственно, заменить сплайны, зависящие от выкинутых узлов. Последовательно проходя по закодированному в первой части потоку, мы проверяем, можно ли выбросить данный узел так, чтобы картинку можно все еще было восстановить без потерь. Если это возможно — мы удаляем данный узел, если же нет — оставляем. Здесь мы можем аналогично кодированию сетки по-разному закодировать выкинутые узлы, в зависимости от того, каким образом сжатие будет существенней. При плохом строении изображения (много разрывов, негладкий вид) данный этап может оставить закодированные узлы без изменений.

После вэйвлетной аппроксимации можно приступать к 3-ему этапу — сжатию с помощью вэйвлетного преобразования. Первое, что мы делаем — добавляем к сетке две точки вначале и две точки в конце — это позволит нам не потерять первую и последнюю исходные точки и связано с формулами для коэффициентов сплайн-вэйвлетного разложения [4]. Далее мы последовательно идем по сетке и можем применить одно из преобразований декомпозиции, описанных в теоретической части: (1.7), (1.8) или (1.9).

Здесь стоит обратить особое внимание на то, что, вообще говоря, данные преобразования оперируют с вещественными числами, которые имеют два серьезных недостатка при работе с вычислительной техникой: они занимают гораздо больше места, нежели целые, и они имеют ограниченную точность, то есть при операциях с ними накапливается ошибка. Но яркости цветов пикселей хранятся в целых числах, поэтому потенциально мы можем отказаться от вещественных чисел. Проблемой здесь является то, что после преобразований декомпозиции мы получим вещественные

числа, и этот факт лишит нас выгоды от наличия лишь целых чисел в яркостях пикселей. Поэтому было решено переписать вычисления в формулах (2.6), (2.7) или (2.8) в целых числах в поле вычетов по модулю простого числа. Огромным везением стало то, что существует простое число 257, элементами поля вычетов по модулю которого являются целые числа от 0 до 256 включительно. То есть, лишь одно число 256 не может быть помещено в один байт информации. Это все же могло бы являться значительным препятствием, но, как выяснилось, данное число совсем нечасто появляется среди результатов преобразований (массиве a), а те случаи, в которых оно встречается мы можем не использовать. Таким образом, мы получаем поле, по сути оперирующее с байтами.

Преобразования (2.6), (2.7) и (2.8) давали различные результаты по b, но в целом отличие было почти несущественным и преобладающего по количеству одинаковых b не было. Поэтому из трех видов преобразований декомпозиции было выбрано преобразование (2.4) с $\sigma = 0$. Внимание к преобладанию количества b здесь уделяется не просто так. Дело в том, что именно за счет b (вэйвлетного потока) мы и получаем сжатие, т. к. просто отбрасываем его и не храним, т. е. при каждом преобразовании от пяти коэффициентов c у нас остается четыре коэффициента a (и коэффициент b). В процессе разработки алгоритма возник вопрос, какие конкретно b мы будет отбрасывать. После проведенных многочисленных экспериментов был построен график характерный для подавляющего большинства изображений.



На данном графике по вертикали отложена вероятность получить тот или иной результат b. Хорошо видно, что чаще всего среди b появляется число 0, поэтому логично, что отбрасывать стоит именно те коэффициенты, у которых результат b после преобразования декомпозиции равен нулю (стоит не забывать, что нельзя использовать те, у которых один из коэффициентов a равен 256).

В итоге мы проходим по нашим коэффициентам и если какой-то из них попадает под условие $b = 0$ и $a_i \neq 256$, то мы применяем декомпозицию, таким образом уменьшая число хранимых коэффициентов. Тут также стоит заметить, что если таких коэффициентов мало по сравнению с общим их числом, то сжатие может не удасться. При тривиальном кодировании их местонахождения нужно как минимум, чтобы 1/8 коэффициентов была равна нулю.

В результате у нас есть три этапа сжатия, каждый из которых в той или иной степени, в зависимости от вида изображения сжимает поток. Совершенно не обязательно, что все этапы будут применены. Информация о том, произошло ли сжатие или нет, пишется в управляющий байт (записанный сразу после информации о размере изображения)

путем взведения или обнуления определенного бита. Данный процесс нужно повторить для потока каждого цвета и записать результирующую информацию по сжатию последовательно в файл после информации о размере и трех управляющих байтов по каждому цвету.

Об обратном преобразовании изображения (распаковке) будет сказано меньше, т. к. этот процесс представляет собой очевидную последовательность действий обратных к действиям при сжатии. Опишем чуть подробнее.

Мы считываем размер изображения, анализируем управляющий байт и смотрим, был ли осуществлен этап вэйвлетной декомпозиции. Если да, то применяем реконструкцию последовательно, начиная от последнего удаленного узла, подставляя в качестве b ноль. Далее, если был произведен второй этап, мы добавляем удаленные узлы, исходя из аппроксимации, на основе имеющихся записях об удаленных узлах и имеющихся коэффициентах. И, наконец, если имел место первый этап, мы восстанавливаем полностью исходных поток, а далее делим его таким образом, чтобы получилось исходное изображение (зная высоту и ширину изображения, это делается тривиально).

2.3. Программная реализация

Предложенный в предыдущей части алгоритм был реализован в виде программного обеспечения, написанного под операционные системы семейства Windows (на данный момент существует версия, скомпилированная под Windows XP и выше, но ничто не мешает с минимальными изменениями получить версии, например, для более старых систем) на языке $C++$. Программа может открывать изображения

в формате BMP, сохранять их в новый предложенный формат WFF (в сжатом виде), а также позволяет просматривать файлы формата WFF.

Структурно программа работает следующим образом. С помощью базовых функций интерфейсов программирования приложений операционных систем семейства Windows (WinAPI) приложение получает хэндл (handle) BMP изображения, аналогично из хэндла изображения получается информация о размере изображения, а так же поток данных, характеризующий цвет его пикселей. Далее данные о цвете раскладываются в 3 массива (для каждого из основных цветов) типа unsigned char.

Первым этапом каждый из этих массивов превращается в список структур, описанный в главе “Кодирование сетки”. Структура имеет примерно следующий вид:

```
struct
{
    long num;
    unsigned char val;
}
```

После чего данный список анализируется и выбирается оптимальный способ кодирования информации о сетке. Вторым этапом из этого списка составляется массив из переменных val. Этот массив поступает на вход процедуре, которая реализует вэйвлетную аппроксимацию и выдает информацию о возможном сжатии на основе аппроксимации. Процедура может иметь следующий вид:

```
void Approximation( unsigned char* inArr, long inArrSize,
 unsigned char* outArr, long* outArrSize, long* outArrNums );
```

Здесь inArr — входной массив байтов;

inArrSize — размер входного массива;

outArr — выходной массив байтов;

outArrSize — размер выходного массива;

outArrNums — номера элементов выходного массива во входном (часть удаляется, а часть остается, нужно знать на каких позициях находились оставшиеся).

Полученная информация анализируется аналогично первому этапу.

Третьим этапом идет уже само вэйвлетное кодирование оставшегося массива (outArr). Процедура, реализующая его, имеет синтаксис схожий с процедурой Approximation. На этом сжатие цветов заканчивается.

Завершением работы является запись в файл. Вначале в файл записывается ширина изображения (2 байта), затем высота (также 2 байта). Далее последовательно записывается информация о каждом цвете. Вначале пишется управляющий байт, где первые три бита отвечают за метод сжатия на первом этапе, вторые три бита за метод сжатия на втором этапе и седьмой бит показывает было ли сжатие на третьем этапе. От каждого этапа кодируется лишь вспомогательная информация — в основном это интерпретация того, что мы получаем в outArrNums (то есть то, как расположены убираемые байты) и лишь в конце записывается финальный массив типа unsigned char (байты), который представляет собой сжатую информацию о яркостях цветов.

2.4. Примеры кода

В этом разделе представлены некоторые выдержки из кода программы.

Следующий код относится к сплайнам и их преобразованиям.

```
double omega( double x1, double x2, double x3, double x4, double x )
{
if( x < x3 )
{
if( x < x2 )
{
if( x < x1 )
return 0;
else
return ( x - x1 )*( x - x1 )/(( x2 - x1 )*( x3 - x1 ));
}
else
{
return ( ( x1 + x2 - x3 - x4 )*x*x - 2*x*(x1*x2 - x4*x3)
 + x1*x2*x4 - x1*x3*x4 + x1*x2*x3 - x2*x3*x4 )/
 ((x3-x1)*(x3-x2)*(x4-x2));
}
}
else
{
if( x < x4 )
return ( x - x4 )*( x - x4 )/(( x4 - x3 )*( x4 - x2 ));
```

```
else
return 0;
}
}

class BaseFunction
{
public:
BaseFunction( int k = 0 )
{
_coeffs.push_back( 1 );
_first = k;
}
BaseFunction( const BaseFunction& bf )
{
*this = bf;
}
BaseFunction operator*( double c )
{
BaseFunction bfo;
bfo._coeffs.assign( _coeffs.begin(), _coeffs.end() );
for( unsigned i = 0; i < bfo._coeffs.size(); ++i )
bfo._coeffs[i] *= c;
bfo._first = _first;
return bfo;
}
```

```
BaseFunction operator+( BaseFunction bf )
{
BaseFunction bfo;
bfo._first = min( _first, bf._first );
bfo._coeffs.resize( max( bf._first+bf._coeffs.size(),
 _first+_coeffs.size() )-bfo._first, 0 );
for( unsigned i = 0; i < _coeffs.size(); ++i )
bfo._coeffs[i + _first - bfo._first] = _coeffs[i];
for( unsigned i = 0; i < bf._coeffs.size(); ++i )
bfo._coeffs[i + bf._first - bfo._first] += bf._coeffs[i];
return bfo;
}
void operator=( const BaseFunction& bf )
{
_first = bf._first;
_coeffs.assign( bf._coeffs.begin(), bf._coeffs.end() );
}
void Incr()
{
_first++;
}
public:
int _first;
std::vector< double > _coeffs;
};
```

```
void ConvertSpline( BaseFunction& w0, BaseFunction& w1,
 BaseFunction& w2, BaseFunction w_1, double* x)
{
double ak = (x[2]-x[1])*(x[3]-x[2])/((x[1]-x[0])*(x[3]-x[0]));
double bk = (x[2]-x[1])*(x[2]-x[0])/((x[3]-x[1])*(x[3]-x[0]));
double ck = (x[3]-x[2])*(x[4]-x[2])/((x[3]-x[1])*(x[4]-x[1]));
double dk = (x[2]-x[1])*(x[3]-x[2])/((x[4]-x[1])*(x[4]-x[3]));
w2 = w2 + w1 * ak;
w1 = w1 * bk + w0 * ck;
w0 = w0 * dk + w_1;
}
```

Далее следует код, отвечающий за декомпозицию и реконструкцию.

```
class ArrayDecomposition
{
public:
ArrayDecomposition( double* xs, double* cn, int _n ):c(NULL),x(NULL)
{
n = _n;
c = new double[n];
x = new double[n];
memcpy( c, cn, sizeof(double)*n );
memcpy( x, xs, sizeof(double)*n );
}
void Decompose0( double* a, double* b, int k )
```

```
{
double q1 = -(x[k+2]-x[k])/(x[k]-x[k-1])*
(x[k+2]-x[k+1])/(x[k+1]-x[k-1]);
double q2 = (x[k+2]-x[k])/(x[k+1]-x[k])*
(x[k+2]-x[k-1])/(x[k+1]-x[k-1]);
double q3 = (x[k+2]-x[k+1])/(x[k+1]-x[k-1])*
(x[k+3]-x[k+2])/(x[k]-x[k-1])*(x[k+3]-x[k+1])/(x[k+1]-x[k]);
double q4 = -(x[k+2]-x[k-1])/(x[k+1]-x[k])*
(x[k+3]-x[k+2])/(x[k+1]-x[k-1])*(x[k+3]-x[k+1])/(x[k+1]-x[k]);
double q5 = (x[k+3]-x[k+2])/(x[k+1]-x[k])*
(x[k+3]-x[k])/(x[k+2]-x[k+1]);
for( int i = 0; i < k-1; ++i )
a[i] = c[i];
a[k-1] = q1*c[k-2] + q2*c[k-1];
a[k] = q3*c[k-2] + q4*c[k-1] + q5*c[k];
for( int i = k+1; i < n-1; ++i )
a[i] = c[i+1];
*b = c[k+1] - q3*c[k-2] - q4*c[k-1] - q5*c[k];
}
void Decompose1( double* a, double* b, int k )
{
double q1 = (x[k+2]-x[k+1])/(x[k+1]-x[k-1])*
(x[k]-x[k+2])/(x[k]-x[k-1]);
double q2 = (x[k+2]-x[k])/(x[k+1]-x[k])*
(x[k+2]-x[k-1])/(x[k+1]-x[k-1]);
for( int i = 0; i < k-1; ++i )
```

```
a[i] = c[i];
a[k-1] = q1*c[k-2] + q2*c[k-1];
for( int i = k; i < n-1; ++i )
a[i] = c[i+1];
double p1 = (x[k+2]-x[k+1])/(x[k+2]-x[k])*
(x[k+3]-x[k+1])/(x[k+3]-x[k]);
double p2 = (x[k+1]-x[k])/(x[k+3]-x[k])*
(x[k+2]-x[k+1])/(x[k+3]-x[k+2]);
*b = c[k] - p1*q1*c[k-2] - p1*q2*c[k-1] - p2*c[k+1];
}
void Decompose2( double* a, double* b, int k )
{
//if( k < 2 )
double q1 = (x[k+3]-x[k])/(x[k+3]-x[k+1])*
(x[k+2]-x[k])/(x[k+2]-x[k+1]);
double q2 = (x[k+2]-x[k])/(x[k+2]-x[k+3])*
(x[k+1]-x[k])/(x[k+3]-x[k+1]);
for( int i = 0; i < k-1; ++i )
a[i] = c[i];
a[k-1] = q1*c[k] + q2*c[k+1];
for( int i = k; i < n-1; ++i )
a[i] = c[i+1];
double p1 = (x[k+1]-x[k])/(x[k]-x[k-1])*
(x[k+2]-x[k+1])/(x[k+2]-x[k-1]);
double p2 = (x[k+1]-x[k])/(x[k+2]-x[k])*
(x[k+3]-x[k+1])/(x[k+3]-x[k]);
```

```
*b = c[k-1] - p1*c[k-2] - p2*q1*c[k] - p2*q2*c[k+1];
}
~ArrayDecomposition()
{
if( c )
delete[] c;
if( x )
delete[] x;
}
private:
double* c;
double* x;
int n;
};

// (mod 257)
void IntDecomp0( unsigned char* c, long* a, long* b )
{
a[0] = c[0];
a[1] = (c[0]*256+c[1]*3)%257;
a[2] = (c[0]+254*c[1]+3*c[2])%257;
b[0] = (c[3]+256*c[0]+3*c[1]+254*c[2])%257;
}


class ArrayReconstruction
```

```
{
public:
ArrayReconstruction( double* xs, double* an, double* bn, int _n )
{
n = _n;
a = new double[n-1];
memcpy( a, an, sizeof(double)*(n-1) );
x = new double[n];
memcpy( x, xs, sizeof(double)*n );
b = *bn;
}
~ArrayReconstruction()
{
if( a )
delete[] a;
}
void Reconstruct( double* c, int k, int sigma )
{
double p1 = (x[k+1]-x[k])/(x[k]-x[k-1])*
(x[k+2]-x[k+1])/(x[k+2]-x[k-1]);
double p2 = (x[k+1]-x[k])/(x[k+2]-x[k])*
(x[k+3]-x[k+1])/(x[k+3]-x[k]);
double p3 = (x[k+2]-x[k+1])/(x[k+2]-x[k])*
(x[k+3]-x[k+1])/(x[k+3]-x[k]);
double p4 = (x[k+1]-x[k])/(x[k+3]-x[k])*
(x[k+2]-x[k+1])/(x[k+3]-x[k+2]);
```

```
for( int i = 0; i < k-1; ++i )
c[i] = a[i];
c[k-1] = a[k-2]*p1 + a[k-1]*p2;
c[k] = a[k-1]*p3 + a[k]*p4;
for( int i = k+1; i < n; ++i )
c[i] = a[i-1];
c[k+1-sigma] += b;
}
private:
double* a;
double b;
double* x;
int n;
};


// (mod 257)
void IntReconstr0( long* a, long* b, unsigned char* c )
{
c[0] = (unsigned char)a[0];
c[1] = (unsigned char)((86*a[0]+86*a[1])%257);
c[2] = (unsigned char)((86*a[1]+86*a[2])%257);
c[3] = (unsigned char)((a[2]+*b)%257);
}
```

И, наконец, код, сжимающий и реконструирующий потоки графической информации.

```
void Compress1Color( unsigned char* buff, long* sz, unsigned char* mode
{
long size1 = size / 8 + (size%8 > 0) + cntr+1;
long bts;
for( bts = 1; max > 1 << bts*8; bts++ );
long size2 = (bts+1)*(cntr+1);

unsigned char* fase2;
long ssize = 0;
if( size < size1 && size < size2 )
{
long tcntr = 0;
*mode = 0;
*sz = size;
for( int i = 0; i < cntr+1; ++i )
{
for( int j = 0; j < comp[i].num; ++j )
{
buff[tcntr++] = comp[i].c;
}
}
fase2 = buff;
ssize = size;
}
else
{
```

```
if( size1 < size2 )
{
long tcntr = 0;
*mode = 1;
*sz = size1;
for( int i = 0; i < cntr+1; ++i )
{
SetNBit( buff, tcntr );
tcntr += comp[i].num;
buff[ size1 - cntr - 1 + i ] = comp[i].c;
}
fase2 = &buff[ size1 - cntr - 1 ];
ssize = cntr+1;
}
else
{
*mode = (unsigned char)(1 + bts);
*sz = size2;
long* crt = (long*)buff;
for( int i = 0; i < cntr+1; ++i )
{
crt = (long*)&buff[ i*bts ];
*crt = comp[i].num;
}
for( int i = 0; i < cntr+1; ++i )
{
```

```
buff[ size2 - cntr - 1 + i ] = comp[i].c;
}
fase2 = &buff[ size2 - cntr - 1 ];
ssize = cntr+1;
}
}

unsigned char* buff2 = new unsigned char[ ssize ];
long* nums2 = new long[ ssize ];
long cntr2 = 1;
long max2 = 1;
buff2[0] = fase2[0]; nums2[0] = 1;
buff2[1] = fase2[1]; nums2[1] = 1;
long diff = buff2[1] - buff2[0];
for( unsigned j = 2; j < ssize; ++j )
{
if( fase2[j] == buff2[cntr2] + diff )
{
buff2[cntr2] = fase2[j];
nums2[cntr2-1]++;
if( nums2[cntr2-1] > max2 )
max2 = nums2[cntr2-1];
}
else
{
diff = fase2[j] - buff2[cntr2];
```

```
cntr2++;
buff2[cntr2] = fase2[j];
nums2[cntr2] = 1;
}
}
cntr2++;

long ssize1 = ssize / 8 + (ssize%8 > 0) + cntr2;
long bts2;
for( bts2 = 1; max2 > ( 1 << bts2*8 ); bts2++ );
long ssize2 = (bts2+1)*cntr2;
if( ssize < ssize1 && ssize < ssize2 )
{
return; // ничего не делаем если не меньше
}
else
{
if( ssize1 < ssize2)
{
long tcntr = 0;
*mode |= 0x08;
*sz = *sz - ssize + ssize1;
ZeroMemory( fase2, cntr2 );
for( int i = 0; i < cntr2; ++i )
{
SetNBit( fase2, tcntr );
```

```
tcntr += nums2[i];
fase2[ ssize1 - cntr2 + i ] = buff2[i];
}
}
else
{
*mode |= (unsigned char)((1 + bts)<<3);
*sz = *sz - ssize + ssize2;
long* crt = (long*)fase2;
for( int i = 0; i < cntr2; ++i )
{
crt = (long*)&fase2[ i*bts2 ];
*crt = nums2[i];
}
for( int i = 0; i < cntr2; ++i )
{
fase2[ ssize2 - cntr2 + i ] = buff2[i];
}
}
}
delete[] nums2;
delete[] buff2;
}

bool Decompress1Color( FILE* f, unsigned char* colbuff, long fs,
 long bonus, long w )
```

```
{
unsigned char mask = 0;
long rs; //real size ( rs <= fs )
fread( &mask, 1, 1, f );
bool mono = mask & 0x80;
long* nums = new long[ fs ];
switch( mask & 0x7 )
{
case 0:
{
for( long i = 0; i < fs; ++i )
nums[i] = 1;
rs = fs;
}
break;
case 1:
{
rs = 0;
long sz = fs/8 + (fs%8 > 0);
unsigned char* net = new unsigned char[ sz ];
fread( net, 1, sz, f );
for( long i = 0; i < fs; ++i )
{
if( GetNBit( net, i ) )
{
rs++;
```

```
nums[rs-1] = 1;
}
else
nums[rs-1]++;
}
delete[] net;
}
break;
case 2:
case 3:
case 4:
case 5:
{
rs = 0;
long bytesCnt = 0;
size_t byteNum = 1 << ( ( mask & 0x7 ) - 2 );
while( bytesCnt <= fs )
{
nums[ rs ] = 0;
fread( &nums[ rs ], byteNum, 1, f );
bytesCnt += nums[ rs ];
rs++;
}
}
break;
}
```

```
unsigned char* bytes4step1 = new unsigned char[ rs ];
switch( ( mask & 0x38 ) >> 3 )
{
case 0:
{
fread( bytes4step1, 1, rs, f );
}
break;
case 1:
{
long rs2 = 0;
//long cntr;
unsigned char* nums2 = new unsigned char[ rs ];
long sz = rs/8 + (rs%8 > 0);
unsigned char* net2 = new unsigned char[ sz ];
fread( net2, 1, sz, f );
for( long i = 0; i < rs; ++i )
{
if( GetNBit( net2, i ) )
{
rs2++;
nums2[rs2-1] = 1;
}
else
nums2[rs2-1]++;
```

```
}
delete[] net2;
unsigned char* bytes4step2 = new unsigned char[ rs2 ];
fread( bytes4step2, 1, rs2, f );
long cntr2 = 0;
for( long j = 0; j < rs2 - 1; ++j )
{
long diff = ( bytes4step2[ j+1 ] - bytes4step2[ j ] )
 / nums2[j];
for( long k = 0; k < nums2[j]; ++k )
bytes4step1[ cntr2++ ] =  bytes4step2[ j ] + diff*k;
}
delete[] bytes4step2;
delete[] nums2;
}
break;
case 2:
case 3:
case 4:
case 5:
{
long rs2 = 0;
long bytesCnt2 = 0;
size_t byteNum2 = 1 << ( ( ( mask & 0x38 ) >> 3 ) - 2 );
unsigned char* nums2 = new unsigned char[ rs ];
while( bytesCnt2 != rs )
```

```
{
nums2[ rs2 ] = 0;
fread( &nums2[ rs2 ], byteNum2, 1, f );
bytesCnt2 += nums2[ rs2 ];
rs2++;
}
unsigned char* bytes4step2 = new unsigned char[ rs2 ];
fread( bytes4step2, 1, rs2, f );
long cntr2 = 0;
for( long j = 0; j < rs2 - 1; ++j )
{
long diff = ( bytes4step2[ j+1 ] - bytes4step2[ j ] )
 / nums2[j];
for( long k = 0; k < nums2[j]; ++k )
bytes4step1[ cntr2++ ] =  bytes4step2[ j ] + diff*k;
}
delete[] bytes4step2;
delete[] nums2;
}
break;
}
long cntr = 0;
for( long i = 0; i < rs; ++i )
{
for( long j = 0; j < nums[ i ]; ++j )
{
```

```
long num = ( cntr / w ) % 2 ?
( ( cntr / w  + 1) * w - cntr % w ) : cntr;
colbuff[ 4*num + bonus ] = bytes4step1[ i ];
cntr++;
}
}
delete[] bytes4step1;
delete[] nums;
return mono;
}
```

3. Заключение

В процессе работы были получены формулы вэйвлетного разложения для конечной сетки, они были преобразованы с учетом особенностей вычислительной техники для работы в поле вычетов по модулю простого числа. Был разработан метод предварительного сжатия неравномерной сетки для потока, полученного из произвольного изображения.

На основе этих результатов было создано програмное обеспечение, имеющее возможность сжимать изображения типа BMP. Для испытания алгоритма были выбраны два типа изображений: реальные картинки — обои рабочего стола с изображением различных пейзажей, автомобилей, цветов и т. п. и искусственные изображения для проверки работы сжимающих фаз. Самое интересное, конечно же заключается в проверке на изображениях первого типа. В среднем лишь только в 5% случаев сжатие не дало ощутимого эффекта, в остальных же случаях изображение успешно сжималось, несмотря на то, что кое-где не было заметно предрасположенности изображения на сжатие (резкие перемены цвета, некая хаотичность структуры). При сжатии проводились регулярные сравнения результата с таким известным форматом, как PNG и TIFF, которые также сжимают изображения без потерь. В среднем данный алгоритм проигрывает PNG, но выигрывает у TIFF, хотя случаются и обратные случаи, конечно же все зависит от специфики конкретного изображения. Для более наглядного примера обратимся к таблице:

	Тип изображения			
Изображение	BMP	PNG	TIFF	Wavelet
Wallpaper	12.288.044	1.784.421	5.921.132	3.083.984
Spectrum	151.578	117.554	146.912	105.017
Auto	2.359.350	516.036	909.520	916.761
Landscape	2.359.350	1.835.615	2.577.234	2.295.157
Beach	3.932.214	1.518.191	2.518.844	1.937.886

Здесь представлены размеры в байтах изображений в нескольких известных форматах, а также в предложенном формате вэйвлетного сжатия. Видно, что на самых разных изображениях алгоритм держится достойно.

С искусственными изображениями все еще лучше. И, конечно же, алгоритм отлично сжимает изображения, где происходят плавные линейные переходы цвета без скачков, либо с редкими скачками.

В заключение нужно отметить, что время работы алгоритма, как и занимаемая память, линейно зависят от входных данных (размера изображения). Это легко видно из описания алгоритма.

4. Дальнейшее развитие

Очевидно, что для наилучшего сжатия нужно по-возможности использовать все особенности изображения как потока данных. В данной работе использовалась, если так можно выразиться, “непрерывность” изображения лишь по горизонтали: даже если изображение представляет собой повторяемую горизонтальную строчку, это почти не поможет нам в сжатии (разве что в крайних точках из-за составления потока в виде змейки).

Отсюда рождается идея использовать двумерные сплайны, т. е. мы повторяем наши теоретические выводы для двумерной сетки и выводим уже не кусочно-параболические сплайны, а в качестве базисных функций берем поверхности, представляющие из себя сегменты параболоида на конечном двумерном множестве (а большая часть по аналогии с одномерным случаем будет представлять собой плоскость с нулевой аппликатой).

Довольно интересной здесь является задача, связанная с построением сетки. Если в одномерном случае удаление узлов влияет лишь на соседние сплайны, то в двумерном, во-первых, влияние будет распространяться по двум перпендикулярным осям, а во-вторых, удаленный узел будет действовать на сетку по этим же направлениям. На данный момент нет строгого решения этой проблемы, и, несомненно, его нужно получать путем как математических выкладок, так и серий экспериментов на реальных изображениях, поэтому данное исследование представляет значительный интерес.

Список литературы

[1] **Демьянович Ю.К.** Биортогональная система для минимальных сплайнов и решения задач интерполяции // Докл. РАН. 2001. Т. 377, № 6. С. 739–742.

[2] **Демьянович Ю.К., Ходаковский В.А.** Введение в теорию вэйвлетов. Санкт-Петербург, 2007.

[3] **Демьянович Ю.К.** Всплески и минимальные сплайны (курс лекций). Санкт-Петербург, 2003.

[4] **Демьянович Ю.К., Косогоров О.М.** Калибровочные соотношения для неполиномиальных сплайнов // Проблемы математического анализа. 2009. Вып. 43. С. 51–68.

[5] **Они же.** О вычислении матриц декомпозиции в сплайн-вэйвлетном разложении // Сб. Методы вычислений. Вып. 23. С. 71–97.

[6] **Они же.** Сплайны и биортогональные системы // Зап. научн. сем. ПОМИ, 2009. 367, С. 9—26.

[7] **Они же.** Сплайн-вэйвлетные разложения на открытом и замкнутом интервалах // Проблемы математического анализа. 2009. Вып.43. С. 69–86.

[8] **Завьялов Ю.С., Квасов Б.И., Мирошниченко В.Л.** Методы сплайн-функций. М., 1980. 352 с.

[9] **Новиков И.Л., Стечкин С.Б.** Основы теории вейвлетов // Успехи мат.наук. 1998. Т. 53, Вып.6 (324). С. 54–128.