САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Математико-механический факультет

Кафедра параллельных алгоритмов

Алгоритмы вейвлетной обработки

звукового потока

Дипломная работа студента 443 группы

Полякова Егора Андреевича

Допущена к защите.

Зав. кафедрой:

д. ф.-м. н., профессор Демьянович Ю.К.

Научный руководитель:

д. ф.-м. н., профессор Демьянович Ю.К.

Рецензент:

к. ф.-м. н., доцент Лебединский Д.М.

Санкт-Петербург

2017

ST. PETERSBURG STATE UNIVERSITY

Mathematics & Mechanics Faculty

Department of Parallel Algorithms

Algorithms of wavelet processing for sound flow

Polyakov Egor

Master’s Thesis

Admitted for defence.

Head of the chair:

Professor Yu.K. Demjanovich

Scientific supervisor:

Professor Yu.K. Demjanovich

Reviewer:

Docent D.M. Lebedinski

Saint-Petersburg

2017

**Оглавление**

[1. Введение 7](#_Toc483429044)

[2. Классический вариант вейвлетного разложения 8](#_Toc483429045)

[2.1. Вейвлетное разложение 8](#_Toc483429046)

[2.2. Программная реализация вейвлетного разложения для звукового потока 10](#_Toc483429047)

[3. Кусочно-линейная аппроксимация 12](#_Toc483429048)

[3.1. Кусочно-линейная аппроксимация с постоянным шагом 12](#_Toc483429049)

[3.2. Кусочно-линейная аппроксимация с варьирующимся шагом 14](#_Toc483429050)

[3.3. Программная реализация кусочно-линейной аппроксимации 16](#_Toc483429051)

[4. Неклассический вариант вейвлетного разложения 19](#_Toc483429052)

[4.1. Пространство кусочно-линейных сплайнов 19](#_Toc483429053)

[4.2. Укрупнение сетки и вложенность пространств 21](#_Toc483429054)

[4.3. Калибровочные соотношения 23](#_Toc483429055)

[4.4. Вейвлетное разложение. Формулы реконструкции 24](#_Toc483429056)

[4.5. Формулы декомпозиции 26](#_Toc483429057)

[5. Заключение 27](#_Toc483429058)

[6. Приложение 28](#_Toc483429059)

[7. Список литературы 34](#_Toc483429060)

1. Введение

В большинстве современных задач приходится иметь дело с большими числовыми потоками информации. В частности, это потоки данных размером порядка нескольких терабайт. Возникает проблема хранения и обработки такой информации. Особенно актуальна задача сжатия таких числовых потоков за счёт отбрасывания несущественной информации и последующего её восстановления.

Наиболее эффективные инструменты для решения подобных проблем – вейвлетные разложения потоков и использование сплайновых аппроксимаций. Они позволяют значительно сократить объём вычислений, при этом позволяя получать достаточно точные результаты. Реализация алгоритмов, основанных на сплайновых приближениях, во многих случаях может быть проще, чем в других алгоритмах.

*Цель работы* –применение основных положений теории вейвлетов, методов сжатия и восстановления потоков.

1. Классический вариант вейвлетного разложения

В этой главе даётся описание классического варианта вейвлетного разложения в простейшей ситуации и его программная реализация.

## Вейвлетное разложение

Основная идея вейвлетного преобразования – разложение исходного числового потока на основной поток, содержащий основную часть информации, и вейвлетный, в котором менее существенные детали. При этом подразумевается возможность восстановления исходного потока из получившихся при разложении. В качестве наглядного примера можно привести сжатие изображения: пусть изображение кодируется некоторым числовым потоком c0, c1, … , c2L-1

Сформируем два числовых потока:

,, где

Из потоков a и b легко восстановить исходный поток, при этом поток b содержит числа, как правило, на порядок меньшие чисел потока a. Это наглядный пример разложения, где один поток является основным, а другой – вейвлетным.

Для более подробного примера рассмотрим разложение кусочно-постоянной функции при помощи вейвлетов Хаара. Пусть задана функция

,

Где N=2k, а определяется как .

Заметим, что .

Обозначим .

Возьмем для наглядности N = 4. Тогда функцию можно представить в виде ,

где a, b – коэффициенты, полученные как в первом примере. Функции, равные разности характеристических функций в полученном представлении, называются вейвлетными базисами, а их линейные комбинации – пространством вейвлетов.

Заметим также, что эти функции получаются изменением аргумента функции .

Например, .

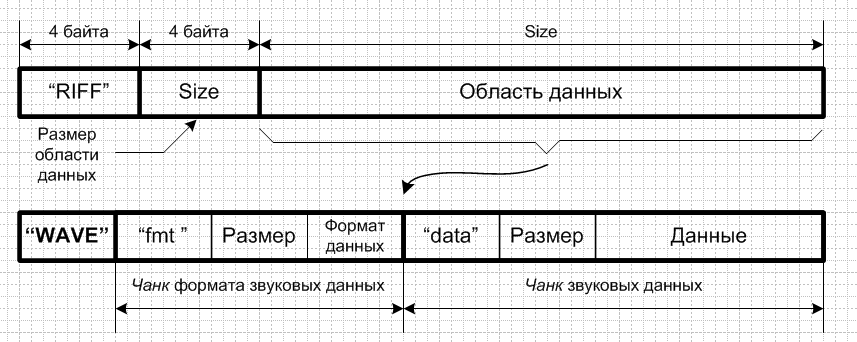
Функция w(t) называется вейвлетом Хаара.

Также можно вывести некоторые важные соотношения для функции из определения очевидно, что , представим в виде

Произведя замену, получаем тождество , называемое кратно-масштабным уравнением. Из этих соотношений следует, что , где – пространство кусочно-постоянных функций, каждая из которых постоянна на , где

## Программная реализация вейвлетного разложения для звукового потока

Применим вейвлетное разложение Хаара для звукового потока. Программная реализация алгоритма и визуализация графиков была написана на языке Delphi. В качестве потока информации использовались аудиофайлы в формате “\*.wav”. Структура wav-файла:



Данные в формате “\*.wav” хранятся в виде двухбайтного целого со знаком. Считав данные в этом формате из файла, был получен числовой поток. Затем было применено разложение Хаара, описанное ранее.

Рассмотрим пример звуковой дорожки, изображенной на картинке (Рисунок 1).

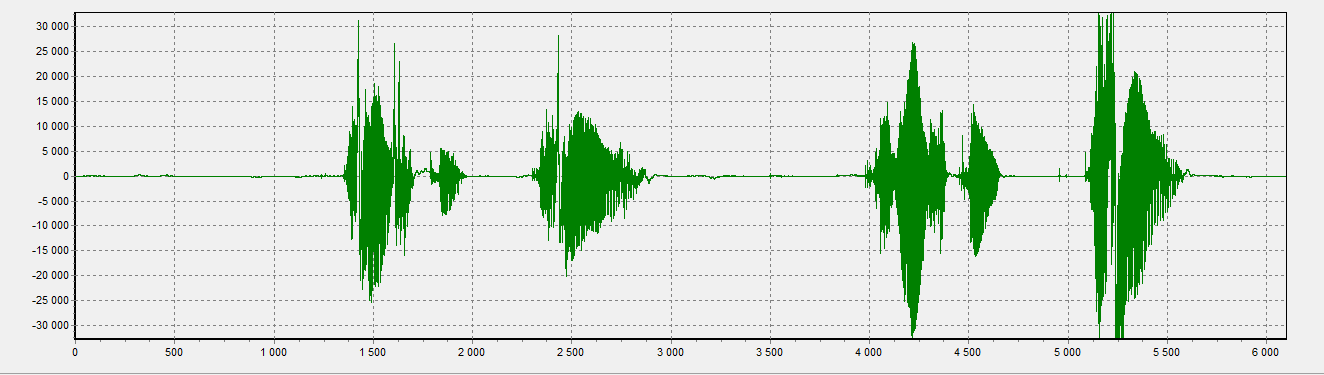


Рисунок 1.

Данный звуковой поток длится 6 секунд и имеет частоту дискретизации в 192000 Гц.

Разложим поток на полусуммы и полуразности. Получим два потока. Поток полусумм визуально не отличается от исходно, помимо того, что количество точек в нем сократилось в двое. А второй поток выглядит так:

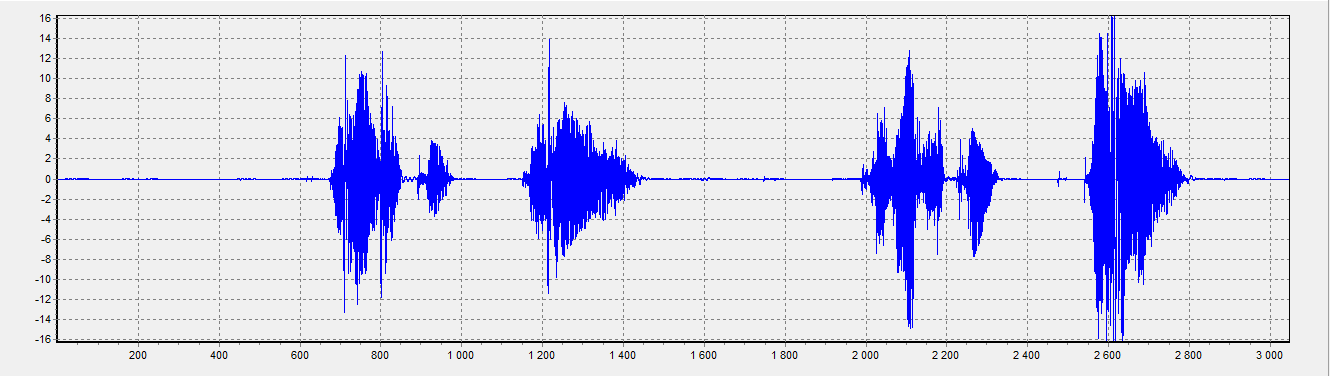


Рисунок 2.

Как видно на картинке (Рисунок 2), этот поток показывает все небольшие изменения. Амплитуда достигает лишь небольших значений. При сравнении с первым потоком, амплитуда колеблется в значениях в 2000 раз меньших.

Ниже (Приложение 1) приведена часть кода программы, использовавшей разложение, на которой были получены приведенные выше картинки.

1. Кусочно-линейная аппроксимация

В этой главе описываются методы кусочно-линейной аппроксимации и программная реализация одного из методов.

## Кусочно-линейная аппроксимация с постоянным шагом

Теперь используем другой подход для разложения числового потока. Рассмотрим наш дискретный сигнал как сетку с частотой, равной частоте дискретизации сигнала. Каждому i-ому шагу нашей сетки сопоставляется соответствующее значение сигнала. Поставим себе задачу – “укрупнять” сетку, пока сигнал новой сетки отличается от исходного не более чем на заданную изначально погрешность.

Введем некоторые обозначения:

t – момент времени; параметр, по которому строится сетка.

Пусть ,,…,– разбиение исходной сетки.

- исходный сигнал в момент времени .

- погрешность.

- шаг исходной сетки

То есть начальными условиями нам дается сигнал с шагом и некий Рассмотрим теперь сетку, состоящую только из нечетных элементов исходной сетки и её концов, т.е. построенную на ,,…,. Она будет соответствовать некоторому сигналу с шагом дискретизации в 2 раза меньше исходного.

Разницей сигналов в момент времени (где = , i = ) будем называть

Другими словами, разница сигналов описывается расстоянием от ломаной, соединяющей точки полученного разбиения, до точек исходного сигнала. Глобальной разницей или просто разницей сигналов будем называть

Ввиду начальных условий, требуется, чтобы разница между сигналами была меньше допустимой погрешности. Проверив это, будем повторять алгоритм “укрупнения” сетки.

На k-ом шаге разницей сигналов в момент времени t будем называть

где

.

– ломаная, построенная по значениям крупной сетки, доопределенная до исходной мелкой сетки.

Легко проверить, что определение разницы для k-ого шага при k=1 будет соответствовать определению разницы для первого шага, так что определение корректно. Определение глобальной разницы так же распространяется на любой, последующий после первого, шаг.

Будем проделывать описанные шаги, получая каждый раз сигнал, в котором в два раза меньше частота дискретизации. В тот момент, когда разница между сигналами превысит – мы получим кусочно-линейную аппроксимацию исходного сигнала.

Полученную аппроксимацию можно привести в исходный сигнал с помощью вейвлетного разложения и формулы реконструкции. Для этого надо рассмотреть оператор проектирования нашей мелкой сетки на полученную крупную сетку.

Тем самым для заданной погрешности и сигнала мы получаем основной результат сжатия и вейвлетный поток, из которых мы можем полностью восстановить исходный поток.

## Кусочно-линейная аппроксимация с варьирующимся шагом

В предыдущей главе мы рассматривали равномерную сетку с фиксированным шагом, которую “укрупняли”. На каждом шаге алгоритма проверяли всю сетку на условие погрешности и выбирали нечетные узлы для “укрупненной сетки”. Данный подход удобен в теоретическом плане, но на практике целесообразнее выбирать узлы в зависимости от того, как ведет себя поток рядом с этими узлами. Критерием выбора узлов будет являться разница значений <.

То есть в начале нам даётся равномерная сетка с шагом и – погрешностью. Далее мы, в отличие от предыдущей главы, не будем сразу рассматривать некую “укрупненную сетку”, а будем сами выбирать узлы для неё.

Рассмотрим узел со значением . Рассмотрим узел со значением . Предположим что сейчас эти 2 узла лежат в “укрупненной” сетке, тогда в нашей терминологии в узлах (s<i<k) можно определить ломаную :

=.

Для каждого из узлов мы должны проверить условие:

Если неравенство выполняется для всех узлов, то далее мы рассматриваем узлы и как узлы “укрупненной” сетки и повторяем действия. Если неравенство не выполнено для одного из узлов, то мы, теперь уже действительно, записываем узлы и в узлы “укрупненной” сетки и начинаем алгоритм с узлов и (снова рассматривая их как потенциальные узлы из “укрупненной” сетки).

В итоге получим неравномерную сетку для заданной – погрешности.

## Программная реализация кусочно-линейной аппроксимации

Была реализована кусочно-линейная аппроксимация, описанная в предыдущей главе, на кроссплатформенном языке Java. Программа может принять на вход аудиофайл и с некоторой, заданной в настройках, погрешностью разложить аудиофайл на 2 файла, которые приблизительно в 2 раза меньше исходного. Из первого файла программа может составить аудиофайл, похожий на исходный. Из двух полученных файлов программа может полностью восстановить исходный.

Была реализована кусочно-линейная аппроксимация, описанная в предыдущей главе. В качестве потока информации использовались аудиофайлы в формате “\*.wav”. Структура wav-файла была показана ранее (Раздел 2.2.). Программа поддерживает аудиофайлы как с одним каналом (моно), так и с двумя каналами (стерео).

Для обработки файла и получения набора байтов используется библиотека 'ava.io.ByteArrayInputStream’. Для обработки аудиофайла и получения информации о нём была использована библиотека ‘‘javax.sound.sampled.\*’. После обработки аудиофайла получаем массив данных, а также набор параметров, таких как: частота дискретизации, длина массива и т.д. (Приложение 2)

Был создан метод (doAproximate) (Приложение 3), который разбивал массив данных на два массива: аппроксимированные данные и погрешности. Забегая вперед скажу, что массив аппроксимированных данных меньше исходного массива и из него программа строит аудиофайл, который является сжатием исходного аудиофайла. В этом методе doAproximate мы берем узлы k и k+j исходного потока и проверяем погрешность для точек k+i, где 0<i<j. Если погрешность допустима для всех i, то повторяем процедуру для узлов k и k+j+1. Если недопустима, то мы записываем значение в узле k в первый массив (основной массив), значения в узлах k+i (0<i<j-1) во второй (дополнительный массив) массив и число (j-1) в третий массив (вспомогательный массив) и повторяем весь алгоритм для узлов k+j-1, k+j+1.

На рисунке 3 изображён алгоритм на примере 7 точек. Следуя алгоритму: изначально, мы берем узел x1 и узел x3. Затем мы проверяем разницу между значением в узле x2 и значением синего отрезка (1,3) в узле x2. Допустим разница меньше допустимой погрешности. Тогда мы берем узлы x1 и x4 и проверяем разницу между значением в узле x2 и значением красного отрезка (1,4) в узле x2. Предположим, что разница оказалось больше допустимой погрешности. Тогда мы записываем значение в узле x1 в основной массив, значение в узле x2 в дополнительный массив и единицу в вспомогательный массив. Далее начинаем снова наш алгоритм с узлов x3 и x5.

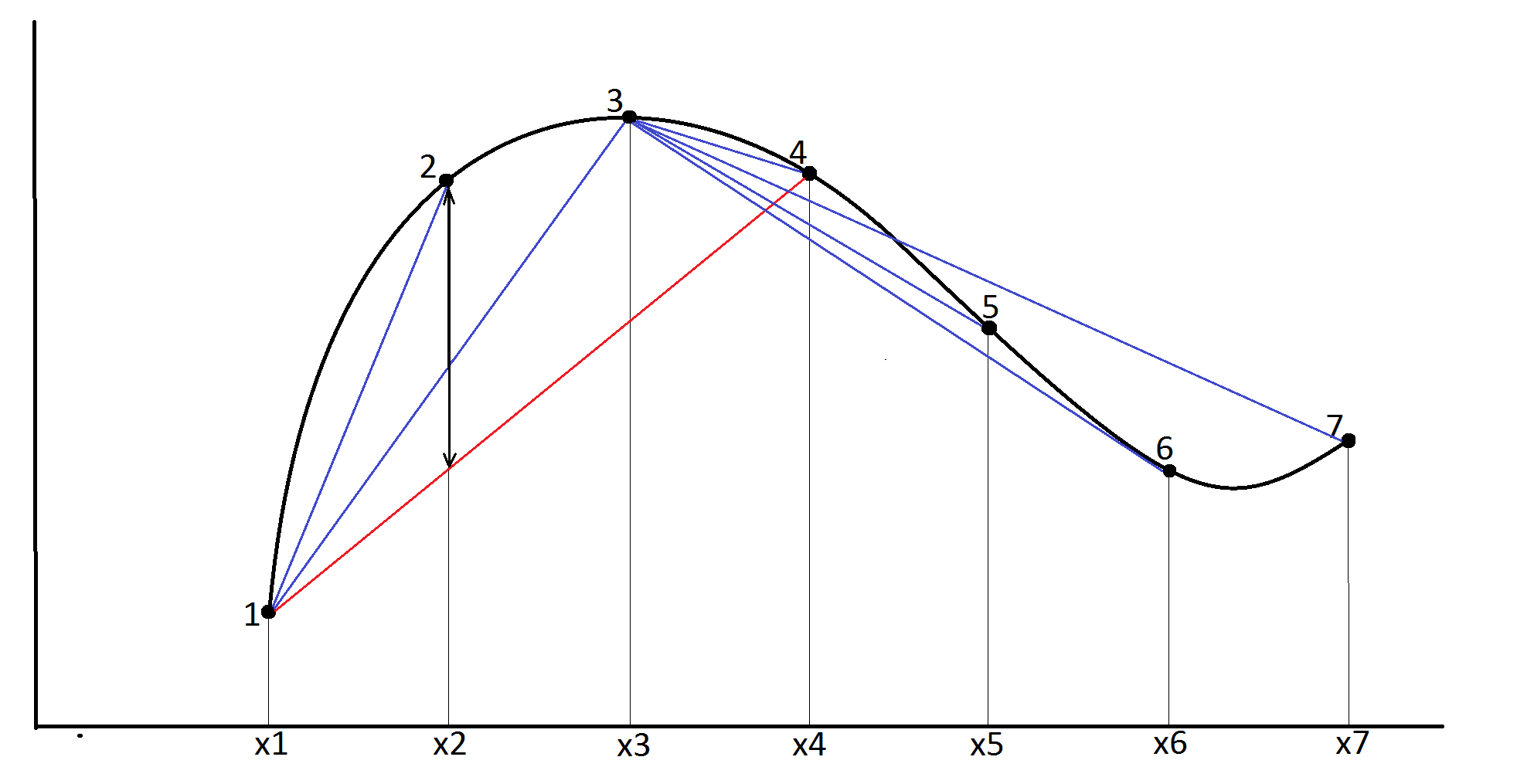


Рисунок 3.

На рисунке 4 изображены точки основного массива с достроенными точками в остальных узлах.

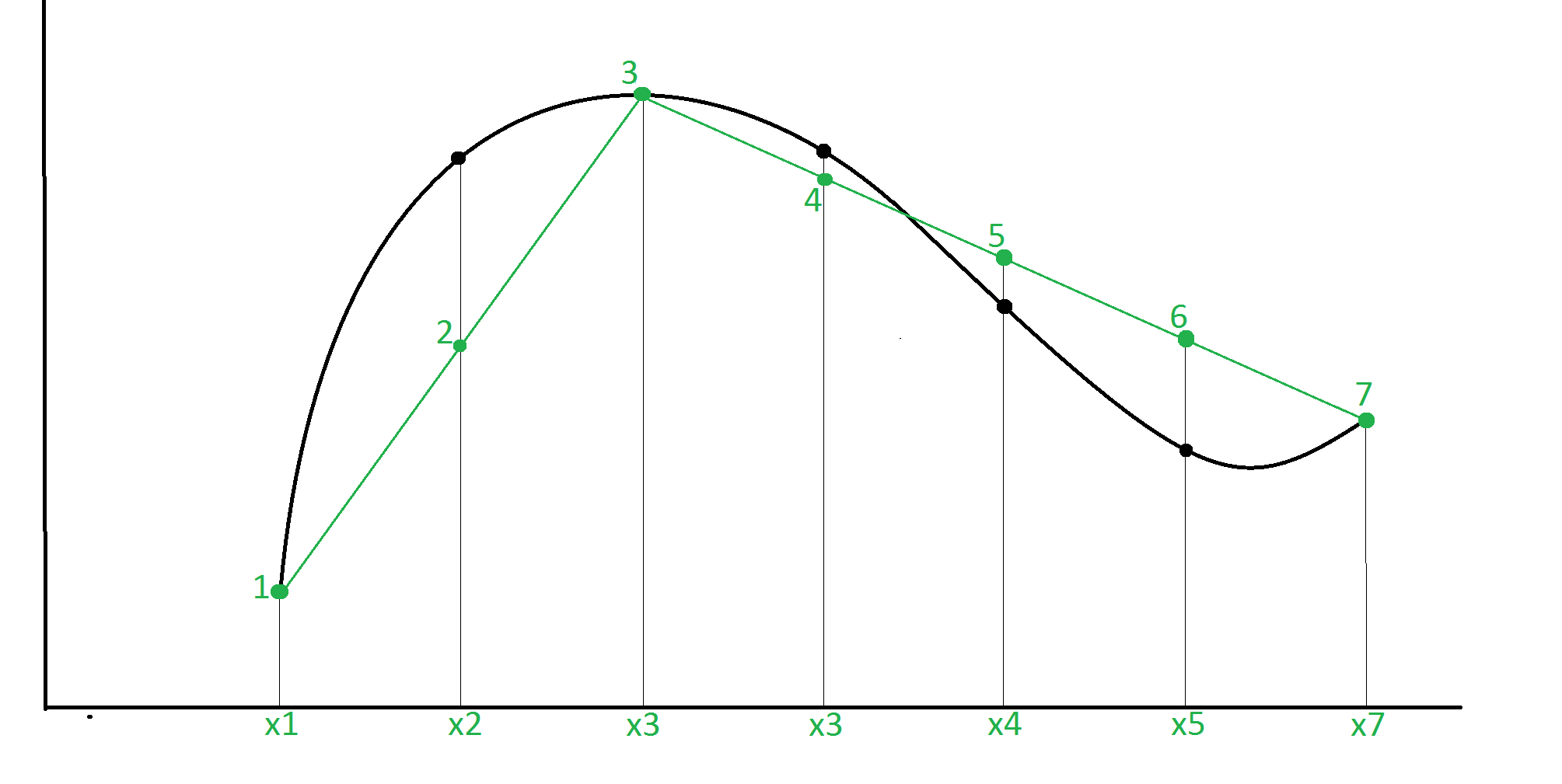


Рисунок 4.

То есть изначально в программу вводится аудиофайл формата “\*.wav” и погрешность. На выходе мы получаем основной массив, дополнительный массив и вспомогательный массив, хранящий количество ‘‘пропущенных’ узлов для каждого узла из основного массива. Из основного и вспомогательного массивов программа генерирует аудиофайл.

Вместе с этим, вспомогательный массив определяет “укрупненную” сетку, соответствующую заданной погрешности (узлы этой сетки соответствуют узлам основного массива). Это “укрупненная” сетка может быть использована для вейвлетного разложения, которое описано в следующих главах.

Также программа восстанавливает исходный аудиофайл из трёх вышеописанных массивов.

Стоит отметить, что главной целью данной программы является получение ‘‘оптимальной’’ “укрупненной сетки”. В данном контексте, оптимальность определяется тем, как звучит аудиофайл, построенный на данной “укрупненной” сетке. Регулируя погрешность, можно достичь нужное качество у аудиофайла. Целью программы не является достижение минимальных размеров у массивов, из которых генерируется аудиофайл. Программа записывает эти массивы в файлы лишь для того, чтобы можно было оценить насколько меньше узлов содержится в “укрупненной” сетке по сравнению с исходной сеткой.

1. Неклассический вариант вейвлетного разложения

## Пространство кусочно-линейных сплайнов

На интервале (α, β) вещественной оси рассмотрим сетку

X∶ . . . < x-2 < x-1 < x0 < x1 < x2 . . .

Со свойствами

Пусть имеется множество двумерных векторов , для которых справедливы соотношения

Множество А называется полной цепочкой векторов.

Пусть Для двухкомпонентной вектор-функции определим функции , из соотношений

**Определение.** Соотношения этого вида называются апроксимационными соотношениями.

При *t* Є (xj, xi+1) отсюда получаем

Так что ввиду определения А имеем

Фиксируя *j* Є и полагая *i=j* и *i=j*+1, получаем

Таким образом, функция *ωj* (t), определена на всем интервале (*α, β*), за исключением точек *xj*, *xj+1*, *xj+2*.

**Определение.** Функции ωj (t), полученные из апроксимационных соотношений, называются координатными сплайнами.

При имеет смысл линейная комбинация бесконечного числа этих функций

Ибо при каждом фиксированном имеется не более двух ненулевых слагаемых

Рассмотрим линейное пространство , определяемое соотношением

Здесь символ означает замыкание в поточечной топологии. Это пространство называется *пространством сплайнов первой степени*, а элементы этого пространства – *сплайнами первой степени*.

## Укрупнение сетки и вложенность пространств

Здесь рассмотрим вложенность пространства непрерывных сплайнов первой степени на “укрупненной” сетке в аналогичное пространство сплайнов на исходной сетке.

Для фиксированного положим

И рассмотрим новую сетку

Для из соотношений, аналогичных соотношениям

Получаем непрерывные сплайны первой степени ; они представляются теми же формулами, как и в предыдущем разделе, с заменой узлов сетки X на узлы сетки :

Полагая аналогично рассмотрим пространство

Очевидно, что для совпадают с рассмотренными ранее:

Кроме того,

Тогда вытекает равенство

Итого имеем

**Теорема.** Справедливо включение

## Калибровочные соотношения

**Теорема.** При справедливы следующие тождества

**Определение.** Представления координатных сплайнов, ассоциированных с крупной сеткой, в виде линейной комбинации координатных сплайнов, ассоциированных с мелкой сеткой, называются калибровочными соотношениями.

**Теорема.** Справедливы калибровочные соотношения

где

## Вейвлетное разложение. Формулы реконструкции

Введем функционалы , задавая их формулой

Рассмотрим оператор *P* проектирования пространства на подпространство , задаваемый формулой

И введем оператор *Q = I- P*, где *I* – тождественный оператор.

Пространством вейвлетов (всплесков) называется пространство

называемое сплайн-вейвлетным разложением пространства

Рассмотрим представление элемента

В соответствии с формулами калибровочных соотношений и сплайн-вейвлетного разложения имеем представление

С другой стороны, справедливо представление u через коэффициенты .

Приравнивание правых частей представлений с учетом линейной независимости координатных сплайнов приводит к соотношениям

Такие​ соотношения называются *формулами реконструкции*.

**Теорема.** Для сплайн-вейвлетного разложения пространства сплайнов первой степени формулы реконструкции имеют вид

## Формулы декомпозиции

Пусть теперь известны коэффициенты по элементам базиса , имеем

Последовательно имеем

Вводя обозначения , получаем

Данные формулы называются *формулами декомпозиции*.

**Теорема**. Формулы декомпозиции всплескового разложения имеют вид

1. Заключение

Рассмотрены основные положения теории вейвлетов, примеры некоторых способов сжатия и восстановления числовых потоков.

Рассмотрено использование кусочно-линейной аппроксимации. А также вейвлетное разложение в пространстве кусочно-линейных сплайнов.

Сделана программная реализация алгоритмов для звуковых потоков:

- Вейвлетное разложение Хаара

- Кусочно-линейная аппроксимация по заданной погрешности

1. Приложение

Ниже приведена часть кода, отвечающая за разложение потока.

**Приложение 1**:

for i:=0 to numSamples-n do

begin

x2:=x1;

for j:=0 to n do

x1:=x1+abs(DataArray[i+j]);

x1:=x1/n;

series[1].AddXY(i/SampleRate\*1000,x1,'',clBlack);

if (t=true) then

begin

series[2].AddXY(i/SampleRate\*1000/2,(x2+x1)/2,'',clBlue);

series[3].AddXY(i/SampleRate\*1000/2,(x2-x1)/2,'',clBlue);

t:=false;

end

else

t:=true;

end;

Ниже приведена часть описания класса WaveFile, который предназначен для обработки аудиофайлов.

**Приложение 2**:

*/\*\*  
 \* Описан аудиофайл (подразумевается аудиодорожка с её параметрами, а не сам файл).  
 \* Методы чтения аудиофайла из файла, запись и чтение сэмплов.  
 \*  
 \*/***public class** WaveFile {  
  
 **private int INT\_SIZE** = 4;  
 **public final int NOT\_SPECIFIED** = -1;  
 **public int sampleSize** = **NOT\_SPECIFIED**; *// размер сэмпла в байтах* **public long framesCount** = **NOT\_SPECIFIED**; *// количество кадров в файле* **public long dataLength** = -1; *// размер данных в байтах* **public byte**[] **data** = **null**; *// массив байт представляющий аудио-данные* **public** AudioInputStream **ais** = **null**; *//поток с аудио-данными* **public** AudioFormat **af** = **null**; *// информация о формате  
  
 /\*\*  
 \* Создает объект из указанного wave-файла  
 \*  
 \** ***@param file*** *- wave-файл  
 \** ***@throws*** *UnsupportedAudioFileException  
 \** ***@throws*** *IOException  
 \*/* **public** WaveFile(File file) **throws** UnsupportedAudioFileException, IOException {  
  
 **if** (!file.exists()) {  
 **throw new** FileNotFoundException(file.getAbsolutePath());  
 }  
  
 *// получаем поток с аудио-данными* **ais** = AudioSystem.*getAudioInputStream*(file);  
  
 *// получаем информацию о формате* **af** = **ais**.getFormat();  
  
 *// количество кадров в файле* **framesCount** = **ais**.getFrameLength();  
  
 *// размер сэмпла в байтах* **sampleSize** = **af**.getSampleSizeInBits() / 8;  
  
 *// размер данных в байтах* **dataLength** = **framesCount** \* **af**.getSampleSizeInBits() \* **af**.getChannels() / 8;  
  
 *// читаем в память все данные из файла разом* **data** = **new byte**[(**int**) **dataLength**];  
 **ais**.read(**data**);  
 }

*/\*\*  
 \* Сохраняет объект WaveFile в стандартный файл формата WAVE  
 \*  
 \** ***@param file*** *- файл, в который будем сохранять  
 \** ***@throws*** *IOException  
 \*/***public void** saveFile(File file) **throws** IOException {  
 AudioSystem.*write*(**new** AudioInputStream(**new** ByteArrayInputStream(**data**),  
 **af**, **framesCount**), AudioFileFormat.Type.***WAVE***, file);  
}  
  
*/\*\*  
 \* Возвращает значение сэмпла по порядковому номеру. Если данные  
 \* записаны в 2 канала, то учитывается, что сэмплы левого и  
 \* правого канала чередуются. Например, сэмпл под номером один это  
 \* первый сэмпл левого канала, сэмпл номер два это первый сэмпл правого  
 \* канала, сэмпл номер три это второй сэмпл левого канала и т.д..  
 \*  
 \** ***@param sampleNumber*** *- номер сэмпла, начиная с 0  
 \** ***@return*** *значение сэмпла  
 \*/***public int** getSampleInt(**int** sampleNumber) {  
  
 **int** sample = 0;  
  
 **if** (sampleNumber < 0 || sampleNumber >= **data**.**length** / **sampleSize**) {  
 **throw new** IllegalArgumentException(  
 **"sample number is can't be < 0 or >= data.length/"** + **sampleSize**);  
 }  
  
 *// массив байт для представления сэмпла* **byte**[] sampleBytes = **new byte**[**sampleSize**];  
  
 *// читаем из данных байты которые соответствуют  
 // указанному номеру сэмпла* **for** (**int** i = 0; i < **sampleSize**; i++) {  
 sample = sample \* 256 + **data**[sampleNumber \* **sampleSize** + i];  
 }  
  
 **return** sample;  
}  
  
*/\*\*  
 \* Устанавливает значение сэмпла  
 \*  
 \** ***@param sampleNumber*** *- номер сэмпла  
 \** ***@param sampleValue*** *- значение сэмпла  
 \*/***public void** setSampleInt(**int** sampleNumber, **int** sampleValue) {  
 *// последовательно переводим числа в байты  
 // и записываем в место, которое соответствует указанному  
 // номеру сэмпла* **for** (**int** i = 0; i < **sampleSize**; i++) {  
 **int** myMod = (**int**) Math.*pow*(256,(**sampleSize**-1 - i));  
 **data**[sampleNumber \* **sampleSize** + i] = (**byte**) (sampleValue / myMod);  
 sampleValue = sampleValue % myMod;  
 }  
}

Ниже приведено описание метода doAproximate и других вспомогательных методов.

**Приложение 3**:

*/\*\*  
 \* Реализует алгоритм аппроксимации данных.  
 \* Заполняются массивы mainData, secondData, helpRecoverData  
 \* Заполняются массивы data и aproximateData исходя их этих трёх массивов  
 \*  
 \** ***@param epsilon*** *- епсилон  
 \*/*

**public void** doAproximate(**int** epsilon) {  
 **int** firstPoint = 0; *// Левый узел* **int** secondPoint = 0; *// Правый узел  
  
 // повторяем, пока не дошли до конца потока* **while** (firstPoint < **data**.**length** - 1) {  
 **if** (secondPoint == **data**.**length**) **break**;  
 *// рассматриваем новую пару узлов* secondPoint = firstPoint + 1;  
 **while** (secondPoint < **data**.**length**) {  
 **int** i = firstPoint + 1;  
 *// проверяем погрешность для узлов, которые лежат между правым и левым узлами* **while** (i < secondPoint) {  
 *//вычисляем погрешность* **double** delta = *abs*((**data**[secondPoint] - **data**[firstPoint]) \* (i - firstPoint) / (secondPoint - firstPoint) - **data**[i]);  
 *// поверяем допустима ли погрешность* **if** (delta > epsilon)  
 **break**;  
 i++;  
 }  
 *// сюда программа придёт, когда закончится предыдущий while  
 // проверка на то, что while закончился, потому что все узлы прошли проверку на погрешность* **if** (i == secondPoint)  
 *// если все узлы прошли проверку, сдвигаем правый узел вправо на один* secondPoint++;  
 **else** {  
 *// если хотя бы один узел не прошел проверку, значит нынешний правый узел нам не подходит  
 // тогда будем считать что в укрупненную сетку пойдёт пара firstPoint и secondPoint - 1  
 // записываем левый узел в основной массив* **mainData**.add(**data**[firstPoint]);  
 *// записываем все узлы, лежащие между правым и (левым узлом - 1) в дополнительный массив* **for** (**int** j = firstPoint + 1; j < secondPoint - 1; j++) {  
 **secondData**.add(**data**[j]);  
 }  
 *// записываем количество узлов, которые мы записали в дополнительный массив, в вспомогательный массив* **helpRecoverData**.add( secondPoint - firstPoint - 2);  
  
 *// сдвигаем оба узла на место того правого узла, который нам подходит* secondPoint = secondPoint - 1;  
 firstPoint = secondPoint;  
 **break**;  
 }  
 }  
 }  
  
 *// Восстанавливаем массив данных из массивов mainData, secondData, helpRecoverData* doRestore();  
  
 *// Достраиваем узлы ломаной исходя из массивов mainData, helpRecoverData* doFinish();  
}  
  
*/\*\*  
 \* Восстанавливаем массив данных data из массивов mainData, secondData, helpRecoverData  
 \*  
 \*/***public void** doRestore(){  
 **int** n = -1; *// указатель на текущую позицию в массиве* **int** num; *// количество пропущенных узлов  
  
 // проходим по всем узлам из основного массива* **for** (**int** i = 0; i < **mainData**.size(); i++) {  
 *// сдвигаем указатель на один* n = n + 1;  
 *// записываем узел из основного массива* **data**[n] = (**int**) **mainData**.get(i);  
 *// считываем сколько узлов было пропущено между этим узлом из основного массива и следующим* num = (**int**) **helpRecoverData**.get(i);  
 *//считываем эти узлы из дополнительного массива* **for** (**int** j = n - i; j < n - i + num; j++) {  
 **data**[j + i + 1] = (**int**) **secondData**.get(j);  
 }  
 *// сдвигаем указатель на количество пропущенных узлов* n = n + num;  
 }  
}  
  
  
*/\*\*  
 \* Достраиваем узлы ломаной. Генерируем поток исходя из массивов mainData, helpRecoverData  
 \*  
 \*/***public void** doFinish(){  
 **int** n = 0; *// указатель на текущую позицию в массиве* **int** num; *// количество пропущенных узлов* **int** secondPoint = 0, firstPoint; *// левый и правый узлы  
  
 // проходим по всем узлам из основного массива* **for** (**int** i = 0; i < **mainData**.size() - 1; i++) {  
 *// считываем сколько узлов было пропущено между этим узлом из основного массива и следующим* num = (**int**) **helpRecoverData**.get(i);  
 *// считываем правый узел* secondPoint = (**int**) **mainData**.get(i+1);  
 *// считываем левый узел* firstPoint = (**int**) **mainData**.get(i);  
 *// записываем правый узел* **aproximateData**[n] = firstPoint;  
 *// достраиваем все узлы между правым и левым узлами и последовательно записываем их* **for** (**int** j = 0; j < num; j++) {  
 **aproximateData**[n + j + 1] = (secondPoint - firstPoint) \* (j+1) / num;  
 }  
 *// сдвигаем указатель на количество пропущенных узлов + 1* n = n + num + 1;  
 }  
 *// записываем последний узел* **aproximateData**[n] = secondPoint;  
}

1. Список литературы
2. Демьянович Ю.К., Ходаковский В.А. Введение в вейвлеты. 2007.
3. Демьянович Ю.К., Зимин А.В. О всплесковом разложении сплайнов эрмитова типа// Сб. Проблемы математического анализа, 2007. Т.35, С.33-45