ФЕДЕРАЛЬНОЕ АГЕНТСТВО

ПО ТЕХНИЧЕСКОМУ РЕГУЛИРОВАНИЮ И МЕТРОЛОГИИ

НАЦИОНАЛЬНЫЙ СТАНДАРТ РОССИЙСКОЙ ФЕДЕРАЦИИ

ГОСТ Р 56047— 2014

Системы охранные телевизионные

КОМПРЕССИЯ ОЦИФРОВАННЫХ АУДИОДАННЫХ

Классификация. Общие требования и методы оценки алгоритмов

Издание официальное

Москва
Стандартинформ
2015

gost.e-ecolog.ru

Предисловие

1 РАЗРАБОТАН Закрытым акционерным обществом «Нордавинд» и Всероссийским научно-исследовательским институтом стандартизации и сертификации в машиностроении (ВНИИНМАШ)

2 ВНЕСЕН Техническим комитетом по стандартизации ТК 234 «Системы тревожной сигнализации и противокриминальной защиты»

3 УТВЕРЖДЕН И ВВЕДЕН В ДЕЙСТВИЕ Приказом Федерального агентства по техническому регулированию и метрологии от 30 июня 2014 г. № 677-ст

4 ВВЕДЕН ВПЕРВЫЕ

Правила применения настоящего стандарта установлены в ГОСТ Р 1.0—2012 (раздел 8). Информация об изменениях к настоящему стандарту публикуется в годовом (по состоянию на 1 января текущего года) информационном указателе «Национальные стандарты», а официальный текст изменений и поправок — в ежемесячно издаваемом информационном указателе «Национальные стандарты». В случае пересмотра (замены) или отмены настоящего стандарта соответствующее уведомление будет опубликовано в ближайшем выпуске ежемесячного информационного указателя «Национальные стандарты». Соответствующая информация, уведомление и тексты размещаются также в информационной системе общего пользования — на официальном сайте Федерального агентства по техническому регулированию и метрологии (gost.ru)

© Стандартинформ, 2015

Настоящий стандарт не может быть полностью или частично вопроизведен, тиражирован или распространен в качестве официального издания без разрешения Федерального агентства по техническому регулированию и метрологии

Содержание

1 Область применения . . . 1
2 Нормативные ссылки . . . 1
3 Термины и определения . . . 1
4 Общие требования . . . 3
5 Классификация . . . 4
6 Методы оценки алгоритмов компрессии . . . 5
6.1 Общее описание методов оценки . . . 5
6.2 Метрика PEAQ . . . 7
6.3 Метрика PSNR . . . 7
6.4 Метрика «коэффициент различия форм сигналов» . . . 7
6.5 Метрика «коэффициент сжатия» . . . 8
7 Методы сравнения алгоритмов компрессии оцифрованных аудиоданных . . . 8
Приложение А (обязательное) Математическое описание алгоритмов расчета метрик оценки качества алгоритмов компрессии аудиоданных . . . 9
А.1 Алгоритм расчета метрики PEAQ . . . 9
А.2 Алгоритм расчета метрики PSNR . . . 23
А.3 Алгоритм расчета метрики «коэффициент различия форм сигналов» . . . 23
А.4 Алгоритм расчета коэффициента сжатия . . . 24
Приложение Б (рекомендуемое) Листинги программ расчета метрик качества аудиоданных . . . 25
Б.1 Листинг программы расчета метрики PEAQ на языке Matlab . . . 25
Б.2 Листинг программы расчета метрики PEAQ на языке С . . . 50
Б.3 Листинг программы расчета метрики PSNR на языке Matlab . . . 89
Б.4 Листинг программы расчета метрики PSNR на языке С . . . 89
Б.5 Листинг программы расчета метрики «Коэффициент различия форм сигналов» на языке Matlab . . . 89
Б.6 Листинг программы расчета метрики «Коэффициент различия форм сигналов» на языке С . . . 90

Введение

Активное применение в системах охранных телевизионных (СОТ) методов компрессии оцифрованных аудиоданных, заимствованных из мультимедийных применений телевидения, из-за низкого качества восстановленных после компрессии оцифрованных аудиоданных привело к невозможности осуществления следственных мероприятий, а также оперативных функций, с использованием отдельных СОТ.

Важной отличительной особенностью методов компрессии оцифрованных аудиоданных для СОТ является необходимость обеспечения высокого качества звука в восстановленных аудиоданных. Настоящий стандарт позволяет упорядочить существующие и разрабатываемые методы компрессии оцифрованных аудиоданных, предназначенные для применения в составе систем противокриминальной защиты.

В качестве критерия для классификации алгоритмов компрессии оцифрованных аудиоданных в настоящем стандарте установлены значения метрик качества, характеризующих степень отклонения восстановленных оцифрованных данных от соответствующих им исходных аудиоданных.

Методика классификации алгоритмов компрессии оцифрованных аудиоданных, приведенная в настоящем стандарте, основана на оценке качества восстановленных аудиоданных, с учетом психоакустических особенностей человеческого слуха. Этот подход к оценке качества восстановленных аудиоданных рекомендован Сектором радиосвязи Международного союза электросвязи (МСЭ-Р), членом которого является Российская Федерация.

НАЦИОНАЛЬНЫЙ СТАНДАРТ РОССИЙСКОЙ ФЕДЕРАЦИИ

Системы охранные телевизионные

КОМПРЕССИЯ ОЦИФРОВАННЫХ АУДИОДАННЫХ

Классификация. Общие требования и методы оценки алгоритмов

Video surveillance systems. Digital audio data compression.
Classification. General requirements and evaluation algorithm methods

Дата введения — 2015—09—01

1 Область применения

Настоящий стандарт распространяется на цифровые системы охранные телевизионные (далее — ЦСОТ) и устанавливает классификацию, общие требования и методы оценки алгоритмов компрессии оцифрованных аудиоданных.

Настоящий стандарт устанавливает методику сравнения различных алгоритмов компрессии и декомпрессии оцифрованных аудиоданных.

Настоящий стандарт применяют к алгоритмам компрессии и декомпрессии аудиоданных независимо от их реализации на аппаратном уровне.

Настоящий стандарт применяют совместно с ГОСТ Р 51558.

2 Нормативные ссылки

В настоящем стандарте использованы нормативные ссылки на следующие стандарты:

ГОСТ 15971 Системы обработки информации. Термины и определения

ГОСТ Р 51558 Средства и системы охранные телевизионные. Классификация. Общие технические требования. Методы испытаний

Примечание — При пользовании настоящим стандартом целесообразно проверить действие ссылочных стандартов в информационной системе общего пользования — на официальном сайте Федерального агентства по техническому регулированию и метрологии в сети Интернет или по ежегодному информационному указателю «Национальные стандарты», который опубликован по состоянию на 1 января текущего года, и по выпускам ежемесячного информационного указателя «Национальные стандарты» за текущий год. Если заменен ссылочный стандарт, на который дана недатированная ссылка, то рекомендуется использовать действующую версию этого стандарта с учетом всех внесенных в данную версию изменений. Если ссылочный стандарт отменен без замены, то положение, в котором дана ссылка на него, рекомендуется применять в части, не затрагивающей эту ссылку.

3 Термины и определения

В настоящем стандарте применены термины по ГОСТ 15971 и следующие термины с соответствующими определениями:

3.1 **алгоритм быстрого преобразования Фурье** (БПФ) (fast Fourier transform, FFT): Набор алгоритмов, реализация которых приводит к существенному уменьшению вычислительной сложности дискретного преобразования Фурье (ДПФ).

Примечание — Смысл быстрого преобразования Фурье состоит в том, чтобы разбить исходный N-отсчетный сигнал $x(n)$ на два более коротких сигнала, дискретные преобразования Фурье которых могут быть скомбинированы таким образом, чтобы получить дискретное преобразования Фурье исходного N-отсчетного сигнала.

Издание официальное

3.2 **алгоритм декомпрессии** (decompression algorithm): Точный набор инструкций и правил, описывающий последовательность действий, согласно которым сжатые аудиоданные преобразуются в восстановленные, реализуемый при помощи аудио декодера.

3.3 **алгоритм компрессии** (compression algorithm): Точный набор инструкций и правил, описывающий последовательность действий, согласно которым исходные аудиоданные преобразуются в сжатые, реализуемый при помощи аудио кодера.

3.4 **амплитудно-временная метрика** (time-amplitude metric): Метрика качества, основанная на сравнении оцифрованных и восстановленных аудиоданных по форме волны.

3.5 **аналого-цифровой преобразователь, АЦП** (analog-to-digital converter, ADC): Устройство, преобразующее входной аналоговый аудиосигнал в оцифрованные аудиоданные.

3.6 **аудио декодер** (audio decoder): Программные, аппаратные или аппаратно-программные средства, с помощью которых осуществляется декомпрессия сжатых аудиоданных.

3.7 **аудио кодер** (audio encoder): Программные, аппаратные или аппаратно-программные средства, с помощью которых осуществляется компрессия оцифрованных аудиоданных.

3.8 **аудиоданные** (audio data), **аудиосигнал** (audio signal), **моноканальный аудиосигнал** (mono channel audio): Аналоговый сигнал, несущий информацию об изменении во времени амплитуды звука.

3.9 **битрейт** (bitrate): Выраженная в битах оценка количества сжатых аудиоданных, определенная для некоторого временного интервала и отнесенная к длительности выбранного временного интервала в секундах.

3.10 **восстановленные аудиоданные** (recovered audio data): Данные, полученные из сжатых аудиоданных после их декомпрессии.

3.11 **декомпрессия сжатых аудиоданных** (decompression of compressed digitized audio data): Восстановление оцифрованных данных из сжатых аудиоданных.

3.12 **дискретное преобразование Фурье, ДПФ** (discrete Fourier transform, DFT): Преобразование, ставящее в соответствие N отсчетам дискретного сигнала N отсчетов дискретного спектра сигнала.

3.13 **дифференциация** (differentia): Выделение частного из общей совокупности по некоторым признакам.

3.14 **искаженный фрейм** (distorted frame): Фрейм, для которого максимальное отношение шума к порогу маскирования превышает 1,5 дБ.

3.15 **искусственная нейронная сеть** (artificial neural network, ANN): Модель биологической нейронной сети, которая представляет собой сеть элементов — искусственных нейронов — связанных между собой синаптическими соединениями.

Примечание — Нейронная сеть предназначена для обработки входной информации и в процессе изменения своего состояния во времени формирует совокупность выходных сигналов.

3.16 **качество восстановленных аудиоданных** (decoded audio data quality): Объективная оценка соответствия восстановленных аудиоданных исходным оцифрованным аудиоданным на основе рассчитанных метрик качества.

3.17 **кодек аудиоданных** (audio codec): Программный, аппаратный или аппаратно-программный модуль, способный выполнять как компрессию, так и декомпрессию аудиоданных.

3.18 **компрессия (сжатие) оцифрованных аудиоданных** (digitzed audio data compression): Обработка оцифрованных аудиоданных с целью уменьшения их объема.

3.19 **компрессия оцифрованных аудиоданных без потерь** (lossless digitized audio compression): Обработка оцифрованных аудиоданных, при которой не происходит потери информации, вследствие чего восстановленные (в результате выполнения декомпрессии) оцифрованные аудиоданные не отличаются от исходных оцифрованных аудиоданных.

3.20 **компрессия оцифрованных аудиоданных с потерями** (lossy compression of digitized audio data): Обработка оцифрованных аудиоданных, при которой происходит потеря информации, и вследствие этого восстановленные (в результате выполнения декомпрессии) оцифрованные аудиоданные отличаются от исходных оцифрованных аудиоданных.

3.21 **метод оценки алгоритма компрессии** (evaluation method of compression algorithm): Метод аналитического определения значений метрик качества на соответствие требованиям, предъявляемым к алгоритмам компрессии аудиоданных.

3.22 **метрика качества** (quality metric): Аналитически определяемые параметры, характеризующие степень отклонения восстановленных аудиоданных от исходных оцифрованных аудиоданных.

3.23 **многоканальный аудиосигнал** (multi-channel audio): Аудиосигнал, состоящий из объединения определенного количества аудиосигналов (каналов), которые несут информацию об одном и том же звуке.

Примечание — Предназначен для более качественной передачи звука с учетом пространственной ориентации.

3.24 **окно** (window): Весовая функция, которая используется для управления эффектами, обусловленными наличием боковых лепестков в спектральных оценках (растеканием спектра).

Примечание — Имеющуюся конечную запись данных или имеющуюся конечную корреляционную последовательность удобно рассматривать как некоторую часть соответствующей бесконечной последовательности, видимую через применяемое окно.

3.25 **оконное преобразование Ханна** (short-time Fourier transform with Hann window): Дискретное преобразование Фурье с весовой функцией — окном Ханна.

3.26 **оцифрованные аудиоданные** (digitized audio data): Данные, полученные путем аналого-цифрового преобразования аудиоданных, представляющие собой последовательность байтов в некотором формате (WAV или др.).

3.27 **передискретизация аудиосигнала** (resampling): Изменение частоты дискретизации аудиосигнала.

3.28 **пиковое отношение сигнал/шум** (peak-to-peak signal-to-noise ratio): Соотношение между максимумом возможного значения сигнала и мощностью шума.

3.29 **порог маскирования** (masking threshold): Пороговый уровень сигнала, не различаемого человеком из-за эффекта психоакустического маскирования.

3.30 **психоакустическая модель** (psychoacoustics model): Модель для сжатия аудиоданных с потерями, использующая особенности восприятия звука человеческим ухом.

3.31 **психоакустическое маскирование** (psychoacoustics masking): Подавление (сокрытие) при определенных условиях одного звука другим звуком из-за особенностей восприятия звука человеческим ухом.

3.32 **разрядность АЦП** (resolution of ADC): Количество бит, которым кодируется каждый отсчет сигнала в процессе АЦП.

3.33 **сжатые аудиоданные** (compressed audio data): Данные, полученные путем компрессии оцифрованных аудиоданных.

3.34 **спектр сигнала** (frequency spectrum): Результат разложения сигнала на простые синусоидальные функции (гармоники).

3.35 **спектрограмма** (spectrogram): Характеристика плотности мощности сигнала в частотно-временном пространстве.

3.36 **степень сжатия** (compressionratio): Коэффициент сокращения объема оцифрованных аудиоданных в результате компрессии.

3.37 **стереофонический двухканальный аудиосигнал** (stereophonic audiosignal), **стерео аудиосигнал** (stereo audio signal), **двухканальный аудиосигнал** (two-channel audio signal): Многоканальный аудиосигнал, состоящий из двух моноканальных аудиосигналов.

3.38 **формат оцифрованных аудиоданных** (digitized audio data format): Представление оцифрованных аудиоданных, обеспечивающее их обработку цифровыми вычислительными средствами.

3.39 **фрейм** (frame): Фрагмент звукового сигнала с заданным количеством значений (длиной фрейма).

3.40 **частота дискретизации** (sample rate): Частота взятия последовательных значений непрерывного во времени сигнала при его аналого-цифровом преобразовании в оцифрованные аудиоданные.

3.41 **частотно-временная метрика** (time-frequency metric): Метрика качества, основанная на сравнении спектрограмм оцифрованных и восстановленных аудиоданных.

3.42 **шум** (noise): Совокупность апериодических звуков различной интенсивности и частоты, не несущая полезную информацию.

4 Общие требования

4.1 Оценку качества восстановленных после сжатия аудиоданных определяют по качеству каждого отдельного звукового фрагмента восстановленных аудиоданных.

4.2 Размер звукового фрагмента определяется в секундах или количеством оцифрованных значений внутри фрагмента.

4.3 Качество звукового фрагмента восстановленных аудиоданных определяют по значениям метрик качества алгоритмов компрессии оцифрованных аудиоданных (далее — метрики качества), харак-

теризующих степень искажения восстановленных после сжатия аудиоданных в сравнении с исходными оцифрованными аудиоданными. Описание метрик приведено в разделе 6 настоящего стандарта, а порядок их расчета приведен в приложении А.

4.4 Алгоритмы компрессии оцифрованных аудиоданных относят к одному из трех классов, установленных в разделе 5 настоящего стандарта.

5 Классификация

5.1 Класс алгоритма компрессии оцифрованных данных определяют по рассчитанным для него значениям метрик качества. Для оценки качества восстановленных аудиоданных и классификации алгоритмов компрессии используют метрики качества, указанные в таблице 1.

Таблица 1 — Диапазоны значений метрик качества по классам алгоритмов компрессии оцифрованных аудиоданных

Метрика качества	Диапазон значений метрик качества по классам алгоритмов компрессии оцифрованных аудиоданных		
	Класс III	Класс II	Класс I
Пиковое отношение сигнал/шум (PSNR), дБ	Менее 30	[30; 40]	Свыше 40
Коэффициент различия форм сигналов	Более 10^{-4}	[10^{-5}; 10^{-4}]	Менее 10^{-5}
Объективная оценка аудиоданных с точки зрения восприятия (PEAQ)	[–3,98; –2,3)	[–2,3; –0,62]	(–0,62; 0,22]
Примечание — Метрики качества отражают изменения оцифрованных аудиоданных (после их обработки алгоритмами компрессии и декомпрессии), которые могут оказать критическое влияние на возможность использования восстановленных аудиоданных для установления наличия звуковых сигналов, дифференциации звуков и речи.			

5.2 В зависимости от значений метрик качества, вычисленных в ходе проведения их оценки, алгоритм компрессии оцифрованных аудиоданных относят к одному из классов:

- класс III — алгоритмы компрессии, обеспечивающие качество восстановленных аудиоданных, достаточное для установления наличия звуковых сигналов и не уступающее в этом качеству исходных аудиоданных, но создающее помехи при дифференциации звуков, понимании речи.

- класс II — алгоритмы компрессии, обеспечивающие качество восстановленных аудиоданных, достаточное для установления наличия звуковых сигналов, дифференциации звуков, речи и не уступающее в этом качеству исходных аудиоданных, но отличимое от качества исходных аудиоданных;

- класс I — полнофункциональные алгоритмы компрессии, обеспечивающие качество восстановленных аудиоданных, неотличимое от качества исходных аудиоданных.

5.3 Значения метрик качества определяют для каждого звукового фрагмента (длиной 5 с) оцифрованных аудиоданных, а в качестве результирующей оценки восстановленных аудиоданных выбирают наименьшее значение для метрик PSNR и PEAQ и наибольшее значение для коэффициента различия форм сигналов.

Для расчета метрик PSNR и коэффициента различия форм сигналов исходные и восстановленные цифровые аудиоданные должны быть представлены с частотой дискретизации 44100 Гц, 16 битами памяти на одно дискретное значение выборки и с одним звуковым каналом. Длина звукового фрагмента 5 с должна включать в себя 220500 оцифрованных значений.

Для расчета метрики PEAQ исходные и восстановленные цифровые аудиоданные должны быть представлены с частотой дискретизации 48000 Гц, 16 битами памяти на одно дискретное значение выборки и с одним или с двумя звуковыми каналами. Длина звукового фрагмента 5 с должна включать в себя 240000 оцифрованных значений для каждого канала.

Для сигналов с частотой, отличной от требуемой, необходимо предварительно выполнить передискретизацию аудиосигнала.

5.4 Алгоритмы компрессии следует различать по степени сжатия, выражаемой через коэффициент сжатия. Коэффициент сжатия определяют как отношение объема исходных несжатых данных к объему сжатых данных [порядок расчета данной метрики выполняют в соответствии с А.4 (приложение А)].

В зависимости от значения коэффициента сжатия алгоритмы компрессии аудиоданных подразделяют на:

- алгоритмы с высокой степенью сжатия — коэффициент сжатия более 42;
- алгоритмы со средней степенью сжатия — коэффициент сжатия от 15 до 42 включительно;
- алгоритмы с низкой степенью сжатия — коэффициент сжатия менее 15.

6 Методы оценки алгоритмов компрессии

6.1 Общее описание методов оценки

Общая схема работы ЦСОТ при использовании алгоритмов компрессии и декомпрессии представлена на рисунке 1.



Рисунок 1 — Общая схема работы ЦСОТ

Аналоговые аудиоданные подвергают аналогово-цифровому преобразованию, в результате которого получают оцифрованные аудиоданные с определенной частотой дискретизации и количеством битов на одно дискретное оцифрованное значение. На компьютере оцифрованные аудиоданные следует хранить в одном из форматов хранения оцифрованных аудиоданных.

Оцифрованные аудиоданные подвергают компрессии, в результате которой формируют сжатые аудиоданные.

Сжатые аудиоданные используют для хранения архива или для передачи по сети, после чего их подвергают декомпрессии. В результате декомпрессии сжатых аудиоданных получают восстановленные аудиоданные, которые используют для воспроизведения оператору и подают на вход программным модулям анализа аудиоданных.

В соответствии с представленной общей схемой работы ЦСОТ классификацию алгоритмов компрессии оцифрованных аудиоданных осуществляют путем оценки метрик качества восстановленных аудиоданных от исходных оцифрованных аудиоданных. В зависимости от особенностей технической реализации конкретной ЦСОТ существует два метода оценки: с разделением оцифрованных аудиоданных и с разделением аудиоданных.

Перед оценкой значений метрик качества оба аудиосигнала (исходный и восстановленный) должны быть преобразованы в сигналы с частотой дискретизации 44100 и 48000 Гц. Для указанных частот количество бит, приходящееся на одно дискретное оцифрованное значение, должно быть равным 16.

6.1.1 Метод оценки алгоритма компрессии на основе разделения оцифрованных аудиоданных

Для применения данного метода техническая реализация ЦСОТ должна позволять получить оцифрованные аудиоданные до их обработки алгоритмами компрессии и декомпрессии.



Рисунок 2 — Общая схема реализации метода оценки на основе разделения оцифрованных аудиоданных

Общая схема реализации метода оценки алгоритма на основе разделения оцифрованных аудиоданных представлена на рисунке 2.

Оценку алгоритма выполняют в последовательности:

- на вход испытуемой ЦСОТ подают последовательные аудиоданные;
- оцифрованные и восстановленные аудиоданные сохраняют на устройствах хранения;
- выполняют расчет значений метрик качества и осуществляют классификацию алгоритма компрессии по таблице 1. Описания метрик приведены в 6.2—6.5. Метрики должны быть рассчитаны в соответствии с приложением А.

6.1.2 Метод оценки алгоритма компрессии на основе разделения аудиоданных

Метод оценки алгоритма компрессии на основе разделения аудиоданных следует применять в случае, если техническая реализация ЦСОТ не позволяет применять метод оценки на основе разделения оцифрованных аудиоданных. Применение данного метода требует наличия дополнительной ЦСОТ в составе испытательного стенда, которая предназначена для сохранения оцифрованных аудиоданных.

Общая схема реализации метода оценки на основе разделения аудиоданных представлена на рисунке 3.

Оценку алгоритма компрессии выполняют в последовательности:

- на вход испытуемой ЦСОТ подают последовательные аудиоданные, которые автоматически дублируются на другую ЦСОТ посредством делителя аудиосигнала, являющегося элементом испытательного стенда;
- восстановленные аудиоданные сохраняют на устройствах хранения ЦСОТ;
- оцифрованные аудиоданные сохраняют на устройствах хранения с использованием возможностей второй ЦСОТ (из состава испытательного стенда);
- выполняют расчет значений метрик качества и осуществляют классификацию алгоритма компрессии по таблице 1. Описания метрик приведены в 6.2—6.5. Метрики должны быть рассчитаны в соответствии с приложением А.

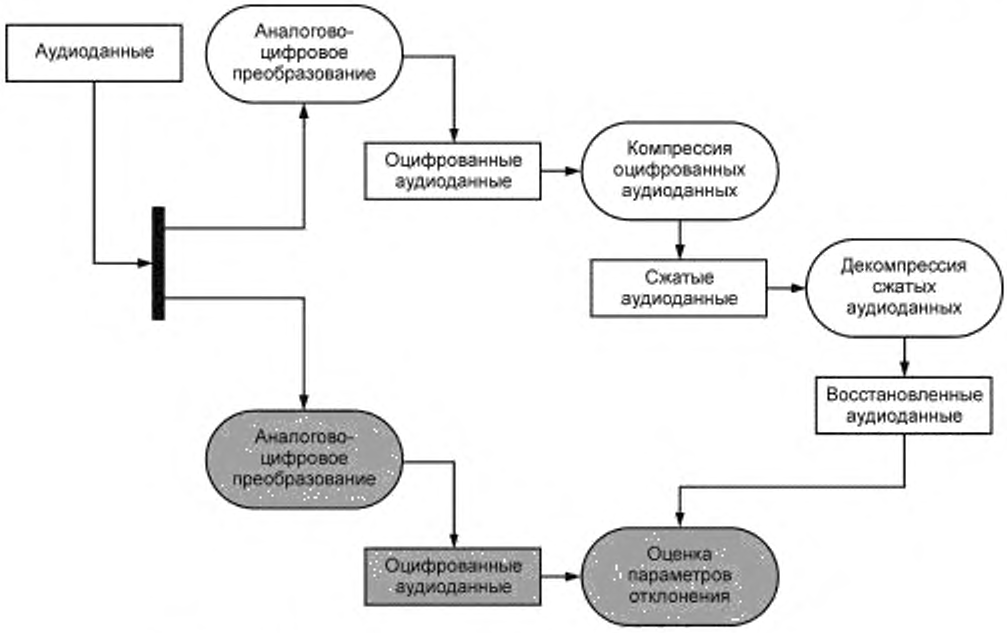

Рисунок 3 — Общая схема реализации метода оценки алгоритма компрессии на основе разделения аудиоданных

6.2 Метрика PEAQ

6.2.1 Метрика PEAQ предназначена для оценки качества обработанного сигнала относительно исходного с учетом слуховых особенностей человека (психоакустической модели). Метрика должна быть рассчитана в соответствии с А.1 (приложение А).

6.2.2 Для расчета метрики PEAQ к аудиосигналам предъявляют следующие требования:

- исходный и восстановленный аудиосигналы должны иметь частоту дискретизации равную 48000 Гц. Для сигналов с частотой отличной от указанной необходимо предварительно выполнить передискретизацию аудиосигнала;
- исходный и восстановленный аудиосигналы должны иметь одинаковую длину, т. е. состоять из одного и того же количества оцифрованных значений.

6.3 Метрика PSNR

6.3.1 Метрика PSNR выражает количественную характеристику отношения энергии шума (искажений), вносимого процессом кодирования, к максимально возможной энергии исходного сигнала. Значения метрики PSNR измеряют в децибелах. Метрика должна быть рассчитана в соответствии с А.2 (приложение А).

6.3.2 Для расчета метрики PSNR к аудиосигналам предъявляют следующие требования:

- исходный и восстановленный аудиосигналы должны иметь частоту дискретизации равную 44100 Гц. Для сигналов с частотой, отличной от указанной, необходимо предварительно выполнить передискретизацию аудиосигнала;
- если исходный и восстановленный аудиосигналы многоканальные, то используют только один канал исходного аудиосигнала и соответствующий ему один канал восстановленного аудиосигнала.

6.4 Метрика «коэффициент различия форм сигналов»

6.4.1 Разница соседних последовательных значений амплитуд исходного аудиосигнала, полученных в результате импульсно-кодовой модуляции, и разница соседних последовательных значений амплитуд восстановленного аудиосигнала определяют соответственно форму исходного аудиосигнала и форму восстановленного аудиосигнала в последовательные моменты времени. Конечное значение метрики «коэффициента различия форм сигналов» подсчитывают как суммарную среднеквадратичную

ошибку между формой исходного аудиосигнала и формой восстановленного аудиосигнала. Метрика должна быть рассчитана в соответствии с А.3 (приложение А).

6.4.2 Для расчета метрики «коэффициент различия форм сигналов» к аудиосигналам предъявляют следующие требования:

- исходный и восстановленный аудиосигналы должны иметь частоту дискретизации равную 44100 Гц. Для сигналов с частотой, отличной от указанной, необходимо предварительно выполнить передискретизацию аудиосигнала;
- если исходный и восстановленный аудиосигналы многоканальные, то для расчета метрики используют только один канал исходного аудиосигнала и соответствующий ему один канал восстановленного аудиосигнала.

6.5 Метрика «коэффицент сжатия»

6.5.1 Метрика «коэффициент сжатия»предназначена для характеристики качества алгоритма компрессии с точки зрения уменьшения объема занимаемой исходными аудиоданными памяти после их обработки алгоритмом сжатия. Метрика должна быть рассчитана в соответствии с А.4 (приложение А).

7 Методы сравнения алгоритмов компрессии оцифрованных аудиоданных

7.1 Два и более алгоритмов компрессии сравнимы друг с другом, если они принадлежат одному и тому же классу в соответствии с таблицей 1.

7.2 Из двух и более сравниваемых алгоритмов компрессии лучшим признают алгоритм, обеспечивающий лучшие значения хотя бы двух из трех метрик, приведенных в таблице 1. Лучшим значением метрики признают большее значение — для метрик PSNR и PEAQ и меньшее значение — для метрики «коэффициент различия форм сигналов». Если сравниваемые алгоритмы имеют одинаковые значения метрик качества, приведенных в таблице 1, то лучшим считают алгоритм с наибольшей степенью сжатия, определяемой коэффицентом сжатия по 6.5.

Приложение А
(обязательное)

Математическое описание алгоритмов расчета метрик оценки качества алгоритмов компрессии аудиоданных

А.1 Алгоритм расчета метрики PEAQ

А.1.1 Обозначения, используемые в алгоритме:

F_s = 48000 Гц — частота дискретизации сигналов;

N_F = 2048 — количество оцифрованных значений сигнала, определяющих длину звукового фрагмента (размер фрейма);

$x[n]$ — оцифрованные данные фрейма, где n — целое число, представляющее собой индекс конкретного значения амплитуды сигнала внутри звукового фрагмента (фрейма), $n \in [0, N_f - 1]$.

покадровый шаг вперед: $N_F/2$ = 1024, таким образом перекрытие фреймов составляет 50 %;

$F_{ss} = F_s/1024$ — частота выборки кадров с учетом покадрового шага;

N_c = 109 — количество частотных полос фильтрации.

А.1.2 Расчет метрики должен состоять из пяти этапов.

I — предварительная обработка сигналов;

II — обработка образов;

III — расчет выходных значений психоакустической модели;

IV — нормирование значений выходных переменных психоакустической модели;

V — оценка качества восстановленного сигнала с помощью искусственной нейронной сети.

А.1.2.1 Предварительная обработка сигналов

А.1.2.1.1 Применение оконного преобразования

Исходные оцифрованные данные разбивают на фреймы. Оцифрованные данные каждого фрейма подвергают масштабированному оконному преобразованию Ханна, h_w, по формуле

$$h_w[n] = \sqrt{8/3}\, h[n, N_F], \tag{А.1}$$

где $h[n, N_F]$ — функция, рассчитываемая по формуле

$$h[n, N_F] = \begin{cases} 0{,}5 \cdot \left(1 - \cos\left(\dfrac{2\pi n}{N_F - 1}\right)\right), & 0 < n < N_F - 1 \\ 0, & \text{в остальных случаях} \end{cases} \tag{А.2}$$

Переход в частотную область осуществляют путем применения дискретного преобразования Фурье (ДПФ) $X[k]$ по формуле

$$X[k] = \frac{1}{N_F} \sum_{n=0}^{N_F - 1} h_W[n]\, x[n]\, e^{-j2\pi nk / N_F}, \tag{А.3}$$

где $0 \le k \le N_F - 1$;

j — мнимая единица.

А.1.2.1.2 Модель наружного и среднего уха

Частотную характеристику наружного и среднего уха $W(f)$ вычисляют по формуле

$$W(f) = 10^{A_{dB}(f)/20}, \tag{А.4}$$

где f — частота, заданная в кГц, а $A_{dB}(f)$ вычисляют по формуле

$$A_{dB}(f) = -2{,}184 \left(\frac{f}{1000}\right)^{-0{,}8} + 6{,}5\, e^{-0{,}6\left(\frac{f}{1000} - 3{,}3\right)^2} - 0{,}001 \left(\frac{f}{1000}\right)^{3{,}6}. \tag{А.5}$$

По формуле (А.4) вектор весовых коэффициентов $W[k]$ вычисляют следующим образом:

$$W[k] = W\left(\frac{kF_s}{N_F}\right), \tag{А.6}$$

где $0 \le k \le \dfrac{N_F}{2}$.

Используя веса, рассчитанные по формуле (А.6), вычисляют взвешенную энергию ДПФ $|X_w[k]|^2$ по формуле:

$$|X_w[k]|^2 = G_L^2 W^2[k]|X[k]|^2, \quad (A.7)$$

где $0 \le k \le \frac{N_F}{2}$;

$G_L = 2{,}794 \cdot 10^{-3}$.

А.1.2.1.3 Разложение критической полосы слуха

Для преобразования частоты сигнала в высоту звука используют шкалу Барка. Расчет следует производить по формуле (А.8), для обратного преобразования — по формуле (А.9)

$$z = B(f) = 7 \operatorname{asinh}(f/650), \quad (A.8)$$

где z — высота звука, измеряемая в Барках.

$$f = B^{-1}(z) = 650\sinh(z/7). \quad (A.9)$$

Полосы частот определяют заданием нижней, центральной и верхней частот каждой полосы и их значение по шкале Барка определяют по формулам

$$z_l[i] = z_L + i\,\Delta z, \quad (A.10)$$

$$z_u[i] = \begin{cases} z_L + (i+1)\Delta z, & i + 1 \le \frac{z_U - z_L}{\Delta z}, \\ z_U & \text{в остальных случаях,} \end{cases} \quad (A.11)$$

$$z_c[i] = \frac{z_U[i] - z_l[i]}{2}, \quad (A.12)$$

где $\Delta z = 1/4$;

$z_L = B(f_L)$;

$z_U = B(f_U)$;

$f_L = 80$ Гц;

$F_U = 18$ кГц.

Обратное преобразование выполняют по формулам

$$f_l[i] = B^{-1}(z_l[i]), \quad (A.13)$$

$$f_c[i] = B^{-1}(z_c[i]), \quad (A.14)$$

$$f_u[i] = B^{-1}(z_u[i]) \quad (A.15)$$

где $i = 1, 2, \ldots, N_c = 109$.

Вклад энергии от k-ой основной частоты ДПФ $U[i, k]$ для i-й полосы частот вычисляют по формуле

$$U[i,k] = \frac{\max\left(0, \min\left(f_u[i], \frac{2k+1}{2}\frac{F_s}{N_F}\right) - \max\left(f_l[i], \frac{2k-1}{2}\frac{F_s}{N_F}\right)\right)}{\frac{F_s}{N_F}}. \quad (A.16)$$

Энергию i-й полосы частот $E_a[i]$ вычисляют по формуле

$$E_a[i] = U_l[i]\,|X_w[k_l[i]]|^2 + \sum_{k=k_l[i]+1}^{k_U[i]-1} |X_w[k]|^2 + U_u[i]\,|X_w[k_U[i]]|^2, \quad (A.17)$$

где $U_l[i] = U[i, k_l[i]]$;

$U_u[i] = U[i, k_U[i]]$.

Конечную формулу энергии i-й полосы частот $E_b[i]$ вычисляют по формуле

$$E_b[i] = \max(E_a[i], E_{min}), \quad (A.18)$$

где $E_{min} = 1 \cdot 10^{-12}$.

А.1.2.1.4 Внутренний шум уха

Для компенсации внутренних шумов в самом ухе, следует ввести надбавочное значение $E_{IN}[i]$ для энергии каждой полосы частот, $E[i]$:

$$E[i] = E_a[i] + E_{IN}[i], \quad (A.19)$$

где внутренний шум $E_{IN}[i]$ моделируют следующим образом.

$$\begin{cases} E_{INdB}(f_{kHz}) = 1{,}456\,(f/1000)^{-0{,}8}, \\ E_{IN}[i] = 10^{E_{INdB}[f_c[i]]/10}. \end{cases} \tag{A.20}$$

Энергии $E[i]$ — образы высоты.

А.1.2.1.5 Энергия распространения в пределах одного фрейма

Характеристику энергии распространения в шкале Барка $E_s[i]$ рассчитывают по формуле

$$E_s[i] = \frac{1}{B_s[i]}\left(E_{sL}[i] + E_{sU}[i]\right)^{2{,}5}, \tag{A.21}$$

где $B_s[i]$ — характеристика рассчитываемая по формуле

$$B_s[i] = \left(\sum_{l=0}^{N_c-1}\left(S(i, l, E[i])\right)^{2{,}5}\right), \tag{A.22}$$

где $E[i]$ — надбавочные значения энергий из (А.19),

$E[0]$ устанавливают равным 1,

$S(i, l, E)$ — характеристика рассчитываемая по формуле

$$S(i, l, E) = \begin{cases} \dfrac{1}{A(l,E)}\left(10^{2{,}7\Delta z}\right)^{i-l}, & i \le l, \\ \dfrac{1}{A(l,E)}\left[\left(10^{-2{,}4\Delta z}\right)\left(10^{-\frac{23\Delta z}{f_c[i]}}\right)\left(E^{0{,}2\Delta z}\right)\right]^{i-l}, & i \ge l \end{cases} = \begin{cases} \dfrac{1}{A(l,E)}a_L^{i-1}, & i \le l, \\ \dfrac{1}{A(l,E)}\left[a_U a_C[l]\, a_E(E)\right]^{i-l}, & i \le l, \end{cases} \tag{A.23}$$

где (для упрощения записи выражения) $a_L = 10^{2{,}7\Delta z}$,

$a_U = 10^{-2{,}4\Delta z}$,

$a_C(l) = 10^{-\frac{23\Delta z}{f_c[i]}}$;

$a_E = E^{0{,}2\Delta z}$.

Тогда $A^{-1}(l, E)$ преобразуется по формуле

$$A^{-1}(l, E) = \sum_{i=0}^{l} a_L^{i-1} + \sum_{i=l}^{N_c-1}\left(a_U\, a_C[l]\, a_E(E)\right)^{i-l} - 1 = \frac{1 - a_L^{-(l+1)}}{1 - a_L^{-1}} + \frac{1 - \left(a_U a_C[l]\, a_E(E)\right)^{N_c-1} - 1}{1 - a_U a_C[l]\, a_E(E)}. \tag{A.24}$$

Слагаемые $E_{sL}[i]$ и $E_{sU}[i]$ из формулы (А.21) вычисляют по формулам

$$E_{sL}[N_C - 1] = \left(\frac{E[N_c - 1]}{A(l, E[N_c - 1])}\right)^{0{,}4}, \tag{A.25}$$

где $i = N_c - 2, \ldots, 0$.

$$E_{sL}[i] = a_L^{-0{,}4}\, E_{sL}(i + 1) + \left(\frac{E[i]}{A(l, E[i])}\right)^{0{,}4}, \tag{A.26}$$

$$E_{sU}[i] = \left(\frac{E[i]}{A(l, E[i])}\right)^{0{,}4}\left[\left(a_U a_C[l]\, a_E(E[i])\right)^{0{,}4}\right]^{i-1}. \tag{A.27}$$

Энергии $E_s[i]$ в дальнейшем в тексте стандарта — образы нераспространенных возбуждений.

А.1.2.1.6 Фильтрация энергии

Фильтрацию энергии вычисляют по формулам

$$E_f[i, n] = \alpha[i]E_f[i, n-1] + (1 - \alpha[i])E_S[i, n], \tag{A.28}$$

$$\tilde{E}_s[i, n] = \max\left(E_f[i, n], E_s[i, n]\right) \tag{A.29}$$

где n — индекс фрейма (фреймы проиндексированы, начиная с $n = 0$);

$E_s[i, n]$ — энергия n-го фрейма, соответствующая формуле (А.21);

$\alpha[i]$ — постоянная времени для угасающей энергии. Начальное условие для фильтрации: $E_f[i, -1] = 0$.

$\tilde{E}_s[i, n]$ — конечные значения (образы возбуждений в дальнейшем).

А.1.2.1.7 Постоянные времени

Постоянную времени $\tau[i]$ для фильтрации i-й полосы вычисляют по формуле

$$\tau[i] = \tau_{min} + \frac{100}{f_c[i]}(\tau_{100} - \tau_{min}), \quad (A.30)$$

где $\tau_{100} = 0{,}03$ с;

$\tau_{min} = 0{,}008$ с.

Постоянную времени для угасающей энергии $\alpha[i]$ следует вычислять по формуле

$$\alpha[i] = \exp\left(-\frac{1}{F_{ss}\tau[i]}\right). \quad (A.31)$$

На рисунке А.1 приведена схема предварительных вычислений.

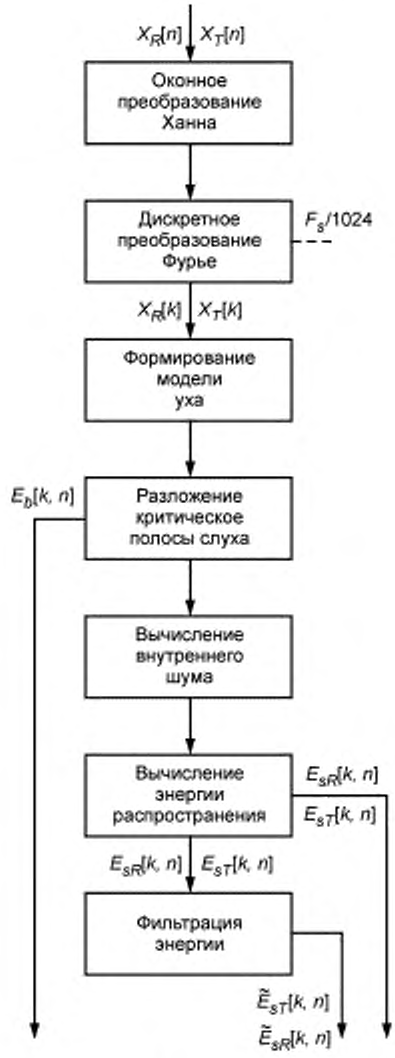

Рисунок А.1 — Схема предварительных вычислений

Индексами R и T обозначают исходный и восстановленный аудиосигналы соответственно. Индексом k обозначают индекс полосы частот (всего 109 полос частот), а индексом n — номер фрейма. Для рекуррентных формул на этом этапе и этапе III всегда выбирают нулевые начальные условия.

А.1.2.2 Обработка образов

А.1.2.2.1 Обработка образов возбуждений

Входными данными для этой стадии вычислений являются образы возбуждений $\tilde{E}_{sR}[k, n]$ и $\tilde{E}_{sT}[k, n]$, рассчитываемые по формулам (А.28) и (А.29) для исходного и восстановленного аудиосигналов соответственно.

Коррекция образов возбуждений

Сначала осуществляют фильтрацию для обоих аудиосигналов по формулам

$$P_R[k, n] = \alpha\,[k]P_R[k, n-1] + (1-\alpha\,[k])\tilde{E}_{sR}[k, n], \tag{А.32}$$

$$P_T[k, n] = \alpha\,[k]P_T[k, n-1] + (1-\alpha\,[k])\tilde{E}_{sT}[k, n], \tag{А.33}$$

где $\alpha\,[i\,]$ — постоянная времени, рассчитываемая по формулам (А.30) и (А.31), но при $\tau_{100} = 0{,}05c$, $\tau_{min} = 0{,}008c$.

Начальное условие для фильтрации выбирают равным 0.

Далее вычисляют коэффициент коррекции, $C_L[n]$:

$$C_L[n] = \left(\frac{\sum_{k=0}^{N_c-1} \sqrt{P_T[k,n]P_R[k,n]}}{\sum_{k=0}^{N_c-1} P_T[k,n]} \right)^2. \tag{А.34}$$

Образы возбуждений, $E_{LR}[k, n]$ и $E_{LT}[k, n]$, корректируют по формулам

$$E_{LR}[k, n] = \begin{cases} \dfrac{\tilde{E}_{sR}[k,n]}{C_L[n]}, & C_L[n] > 1, \\ \tilde{E}_{sR}[k,n], & C_L[n] \le 1. \end{cases} \tag{А.35}$$

$$E_{LT}[k, n] = \begin{cases} \tilde{E}_{sT}[k,n], & C_L[n] > 1, \\ \tilde{E}_{sT}[k,n]\,C_L[n], & C_L[n] \le 1. \end{cases} \tag{А.36}$$

Адаптация образов возбуждений

Используя те же постоянные времени и начальные условия, что и при коррекции образов возбуждений, выходные сигналы, рассчитанные по формулам (А.35) и (А.36), сглаживают в соответствии с формулами

$$R_n[k, n] = \alpha\,[k]R_n[k, n-1] + E_{LT}[k, n]E_{LR}[k, n], \tag{А.37}$$

$$R_d[k, n] = \alpha\,[k]R_d[k, n-1] + E_{LR}[k, n]E_{LR}[k, n]. \tag{А.38}$$

На основе соотношения между рассчитанными в формулах (А.37) и (А.38) значениями вычисляют пару вспомогательных сигналов:

$$R_R[k, n] = \begin{cases} 1, & R_n[k,n] \ge R_d[k,n], \\ \dfrac{R_n[k,n]}{R_d[k,n]}, & R_n[k,n] < R_d[k,n], \end{cases} \tag{А.39}$$

$$R_T[k, n] = \begin{cases} \dfrac{R_d[k,n]}{R_n[k,n]}, & R_n[k,n] \ge R_d[k,n], \\ 1, & R_n[k,n] < R_d[k,n]. \end{cases} \tag{А.40}$$

Если в формулах (А.39) и (А.40) числитель и знаменатель равны нулю, то необходимо выполнить действия:

$$R_T[k, n] = R_T[k-1, n] \text{ и } R_R[k, n] = R_R[k-1, n]. \tag{А.41}$$

Если $k = 0$, то $R_T[k, n] = R_R[k, n] = 1$. (А.42)

С целью формирования множителей для коррекции образов вспомогательные сигналы подвергают фильтрации с использованием тех же постоянных времени и начального условия, что и в формулах (А.32) и (А.33):

$$P_{CR}[k, n] = \alpha\,[k]P_{CR}[k, n-1] + (1-\alpha\,[k])R_{aR}[i, n], \tag{А.43}$$

$$P_{CT}[k, n] = \alpha\,[k]P_{CT}[k, n-1] + (1-\alpha\,[k])R_{aT}[i, n], \tag{А.44}$$

где

$$R_{sR}[k, n] = \frac{1}{M_1[k] + M_2[k] + 1} \sum_{i=k-M_1[k]}^{k+M_2[k]} R_R[i, n], \quad (A.45)$$

$$R_{sT}[k, n] = \frac{1}{M_1[k] + M_2[k] + 1} \sum_{i=k-M_1[k]}^{k+M_2[k]} R_T[i, n], \quad (A.46)$$

$$M_1[k] = \min(3, k), \; M_2[k] = \min(4, N_c - 1 - k). \quad (A.47)$$

Конечным результатом этой стадии обработки, на основе формул (А.43) и (А.44) получают спектрально адаптированные образы, $E_{PT}[k, n]$ и $T_{PR}[k, n]$, по формулам

$$E_{PT}[k, n] = E_{LT}[k, n] P_{CT}[k, n], \quad (A.48)$$

$$E_{PR}[k, n] = E_{LR}[k, n] P_{CR}[k, n]. \quad (A.49)$$

А.1.2.2.2 Обработка образов модуляции

Входными данными для этой стадии вычислений являются образы нераспространенных возбуждений $E_{sR}[k, n]$ и $E_{sT}[k, n]$, которые рассчитывают по формуле (А.21) для исходного и восстановленного аудиосигналов соответственно. Вычисляют меры модуляций огибающих спектра.

Предварительно вычисляют среднюю громкость, $\overline{E}_R[k, n]$ и $\overline{E}_T[k, n]$ по формулам

$$\overline{E}_R[k, n] = \alpha[k] \overline{E}_R[k, n-1] + (1 - \alpha[k])(E_{sR}[k, n])^{0,3}, \quad (A.50)$$

$$\overline{E}_T[k, n] = \alpha[k] \overline{E}_T[k, n-1] + (1 - \alpha[k])(E_{sT}[k, n])^{0,3}. \quad (A.51)$$

Далее необходимо вычислить разности $\overline{D}_R[k, n]$ и $\overline{D}_T[k, n]$:

$$\overline{D}_R[k, n] = \alpha[k] \overline{D}_R[k, n-1] + (1 - \alpha[k]) F_{ss} |(E_{sR}[k, n])^{0,3} - (E_{sR}[k, n-1])^{0,3}|, \quad (A.52)$$

$$\overline{D}_T[k, n] = \alpha[k] \overline{D}_T[k, n-1] + (1 - \alpha[k]) F_{ss} |(E_{sT}[k, n])^{0,3} - (E_{sT}[k, n-1])^{0,3}|. \quad (A.53)$$

Постоянные времени и начальные условия используют те же самые, что и в А.1.2.2.1.

Меры модуляции огибающих спектра вычисляют по формулам

$$M_R[k, n] = \frac{\overline{D}_R[k, n]}{1 + \overline{E}_R[k, n]/0,3}, \quad (A.54)$$

$$M_T[k, n] = \frac{\overline{D}_T[k, n]}{1 + \overline{E}_T[k, n]/0,3}. \quad (A.55)$$

А.1.2.2.3 Вычисление громкости

Образы громкости вычисляют в соответствии с формулами

$$N_R[k, n] = c \left(\frac{E_t[k]}{S[k] E_0} \right)^{0,23} \left[\left(1 - s[k] + \frac{s[k] + \widetilde{E}_{sR}[k, n]}{E_t[k]} \right)^{0,23} - 1 \right], \quad (A.56)$$

$$N_T[k, n] = c \left(\frac{E_t[k]}{s[k] E_0} \right)^{0,23} \left[\left(1 - s[k] + \frac{s[k] + \widetilde{E}_{sT}[k, n]}{E_t[k]} \right)^{0,23} - 1 \right], \quad (A.57)$$

где $c = 1,07664$.

$$s_{dB}(f) = -2 - 2,05 \operatorname{atan}\left(\frac{f}{4000} \right) - 0,75 \operatorname{atan}\left(\left(\frac{f}{1600} \right)^2 \right), \quad (A.58)$$

$$S[k] = 10^{s_{dB}(f_c[k]/1000)^{0,6}}, \quad (A.59)$$

$$E_{tdB}(f) = 3,64 \cdot (f/1000)^{-0,8}, \quad (A.60)$$

$$E_t[k] = 10^{E_{tdB}(f_c[k])/10}. \quad (A.61)$$

Общие громкости для обоих сигналов вычисляют по формулам

$$N_{totR}[n] = (24/N_c) \sum_{k=0}^{N_c-1} \max(N_R[k, n], 0), \quad (A.62)$$

$$N_{totT}[n] = (24/N_c) \sum_{k=0}^{N_c-1} \max(N_T[k, n], 0). \quad (A.63)$$

А.1.2.3 Расчет выходных значений психоакустической модели

А.1.2.3.1 Общее описание процесса расчета параметров

Выходные характеристики из А.1.2.1 используют для вычисления выходных характеристик А.1.2.2 в соответствии со схемой, приведенной на рисунке А.2.

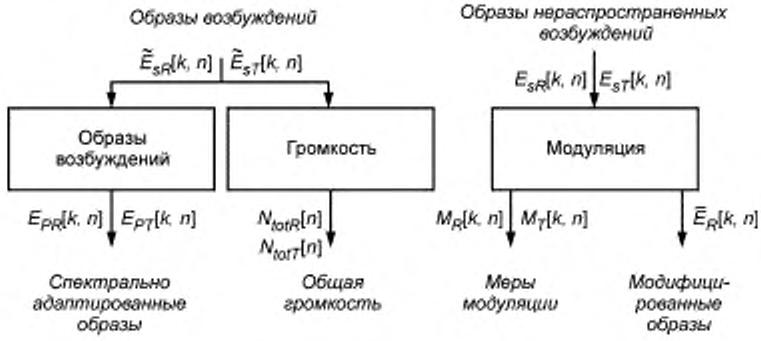

Рисунок А.2 — Схема обработки образов

Значения из А.1.2.2 используют для вычисления выходных значений переменных психоакустической модели (см. таблицу А.1 и рисунок А.3).

Рассчитывают значения 11 переменных психоакустической модели. Наименование и краткое описание переменных психоакустической модели приведены в таблице А.1.

Таблица А.1 — Выходные переменные психоакустической модели

Наименования выходной переменной модели	Описание
BandwidthRefB	Ширина полосы исходного аудиосигнала
BandwidthTestB	Ширина полосы восстановленного аудиосигнала
Total NMRB	Отношение уровня шума к порогу маскирования
WinModDiff1B	Оконная разница модуляций
ADBB	Среднее поблочное искажение
EHSB	Гармоническая структура ошибки
AvgModDiff1B	Средняя разница модуляции 1
AvgModDiff2B	Средняя разница модуляции 2
RmsNoiseLoudB	Громкость шума
MFPDB	Максимальная вероятность обнаружения искажения
RelDistFramesB	Относительное искажение фреймов

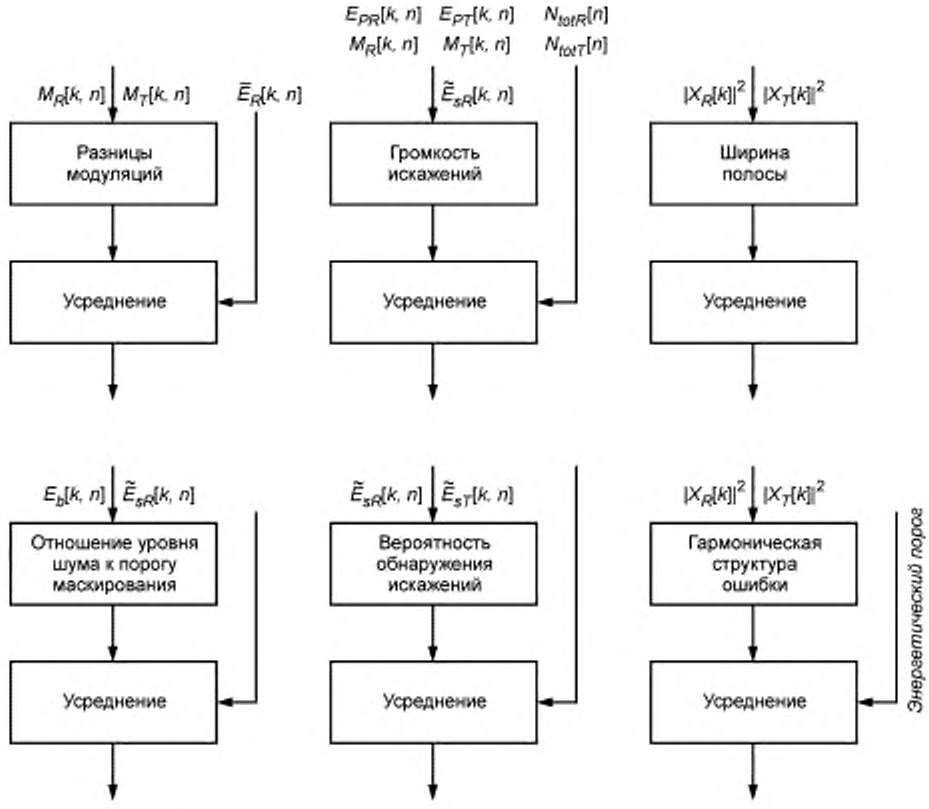

Рисунок А.3 — Схема вычисления значений выходных переменных психоакустической модели

Для двухканальных аудиосигналов значения переменных для каждого канала следует рассчитывать отдельно, а затем усреднять. Значения всех переменных (кроме значений переменных ADBB и MFPDB) для каждого канала сигнала рассчитывают независимо от второго канала.

Все значения выходных переменных модели получают путем усреднения по всем фреймам функций времени и частоты, введенных на предыдущем шаге (в результате имеем скалярное значение).

Значения, которые будут усреднены, должны лежать в границах, определяемых следующим условием: начало или конецданных, которые будут подвержены усреднению, определяют как первую позицию с начала или с конца последовательности значений амплитуд аудиосигнала, для которой сумма пяти последовательных абсолютных значений амплитуд превышает 200 в любом из аудио каналов. Фреймы, которые лежат вне этих границ, следует игнорировать при усреднении. Значение порога 200 используют в случае, если амплитуды входных аудиосигналов нормализованы в диапазоне от минус 32768 до плюс 32767. В противном случае значение порога A_{th} вычисляют следующим образом.

$$A_{th} = 200 \frac{A_{max}}{32768}, \tag{A.64}$$

где A_{max} — максимальное значение амплитуды аудиосигнала.

В дальнейшем индекс фрейма n начинается с нуля для первого фрейма, удовлетворяющего условиям проверки границ с порогом A_{th}, и отсчитывает число фреймов N вплоть до последнего фрейма, удовлетворяющего вышеприведенному условию.

А.1.2.3.2 Оконная разница модуляций 1 (WinModDiff1B)

Вычисляют мгновенную разницу модуляций, $M_{diff1B}[k, n]$, по формуле

$$M_{diff1_B}[k, n] = \frac{|M_T[k,n] - M_R[k,n]|}{1 + M_R[k,n]}. \tag{A.65}$$

Значение мгновенной разницы модуляций усредняют по всем полосам частот N_c в соответствии со следующей формулой

$$\tilde{M}_{\text{diff1}_B}[n] = \frac{100}{N_c} \sum_{k=0}^{N_c-1} M_{\text{diff1}_B}[k,n]. \tag{A.66}$$

Конечное значение выходной переменной получают усреднением формулы (А.66) со скользящим окном $L = 4$ (85 мс, так как каждый шаг равен 1024 оцифрованных значений):

$$M_{Wdiff1_B} = \sqrt{\frac{1}{N-L+1} \sum_{n=L-1}^{N-1} \left\{ \frac{1}{L} \sum_{l=0}^{L-1} \sqrt{\tilde{M}_{\text{diff1}_B}[n-l]} \right\}^4}. \tag{A.67}$$

При этом применяют усреднение с задержкой — первые 0,5 с сигнала не участвуют в вычислениях. Количество пропускаемых фреймов составляет:

$$N_{dw} = \lceil 0{,}5 F_{sa} \rceil. \tag{A.68}$$

В формуле (А.68) операция $\lceil\ \rceil$ означает отбрасывание дробной части.

В формуле (А.67) индекс фреймов включает только фреймы, которые идут после задержки величиной 0,5 с.

А.1.2.3.3 Средняя разница модуляций 1 (WinModDiff1B)

Значение данной выходной переменной психоакустической модели, M_{Adiff1_B}, вычисляют по формуле

$$M_{Adiff1_B} = \frac{\sum_{n=0}^{N-1} W_{1_B}[n] M_{\text{diff1}_B}[n]}{\sum_{n=0}^{N-1} W_{1_B}[n]}, \tag{A.69}$$

где

$$W_{1_B}[n] = \sum_{k=0}^{N_c-1} \frac{\overline{E}_R[k,n]}{E_R[k,n] + 100(E_W[k])^{0,3}}. \tag{A.70}$$

Для вычисления этого значения также применяют усреднение с задержкой.

А.1.2.3.4 Средняя разница модуляций 2 (WinModDiff2B)

Сначала вычисляют значение мгновенной разницы модуляций по формуле

$$M_{Wdiff2_B}[k,n] = \begin{cases} \dfrac{M_T[k,n] - M_R[k,n]}{0{,}01 + M_R[k,n]}, & M_T[k,n] \geq M_R[k,n], \\ 0{,}1 \dfrac{M_R[k,n] - M_T[k,n]}{0{,}01 + M_R[k,n]}, & M_T[k,n] < M_R[k,n]. \end{cases} \tag{A.71}$$

Затем вычисляют усредненное по полосам частот значение разности модуляций.

$$\tilde{M}_{\text{diff2}_B}[n] = \frac{100}{N_c} \sum_{k=0}^{N_c-1} M_{\text{diff2}_B}[k,n]. \tag{A.72}$$

Конечное значение переменной психоакустической модели M_{Adiff2_B} вычисляют по формулам (А.73) и (А.74)

$$M_{Adiff2_B} = \frac{\sum_{n=0}^{N-1} W_{2_B}[n] \tilde{M}_{\text{diff2}_B}[n]}{\sum_{n=0}^{N-1} W_{2_B}[n]}, \tag{A.73}$$

где

$$W_{2_B}[n] = \sum_{k=0}^{N_c-1} \frac{\overline{E}_R[k,n]}{\overline{E}_R[k,n] + 100(E_W[k])^{0,3}}. \tag{A.74}$$

При вычислении этого значения также применяют усреднение с задержкой.

А.1.2.3.5 Громкость шума (RmsNoiseLoudB)

Значение мгновенной громкости шума рассчитывают по формуле

$$\tilde{N}_L[n] = (24/N_c)\sum_{k=0}^{N_c-1} N_L[k, n], \tag{A.75}$$

где $N_L[k, n] = \left(\frac{E_t[k]}{s_T[k,n]E_0}\right)^{0,23}\left[\left(1+\frac{\max(s_T[k,n]E_{PT}[k,n]-s_R[k,n]E_{PR}[k,n],0)}{E_t[k]+\beta[k,n]s_R[k,n]E_{PR}[k,n]}\right)^{0,23}-1\right]$, (A.76)

где $E_0 = 1$.

$$S_R[k, n] = T_0 M_R[k, n] + S_0; \tag{A.77}$$

$$S_T[k, n] = T_0 M_T[k, n] + S_0; \tag{A.78}$$

$$\beta[k, n] = \exp\left(-\alpha\frac{E_{PT}[k,n]-E_{PR}[k,n]}{E_{PR}[k,n]}\right), \tag{A.79}$$

где $\alpha = 1,5$;

$T_0 = 0,15$;

$S_0 = 0,5$.

Если мгновенная громкость менее 0, ее устанавливают равной 0:

$$\tilde{N}_L[n] = \begin{cases}\tilde{N}_L[n] & \tilde{N}_L[n] \geq 0,\\ 0 & \tilde{N}_L[n] < 0.\end{cases} \tag{A.80}$$

Значение конечной выходной переменной психоакустической модели N_{LrmsB} находят усреднением мгновенной громкости по формуле (А.81):

$$N_{LrmsB} = \sqrt{\frac{1}{N}\sum_{n=0}^{N-1}(\tilde{N}_L[n])^2}. \tag{A.81}$$

Для вычисления этого значения применяют усреднение с задержкой. Совместно с усреднением с задержкой используют порог громкости для нахождения значения мгновенной громкости шума, начиная с которого выполняют процесс усреднения. Таким образом, усреднение начинают с первого значения, определяемого условием превышения порога громкости, но не позднее 0,5 с от начала сигнала (в соответствии с усреднением с задержкой).

Условие превышения порога громкости

Значения мгновенной громкости шума в начале обоих сигналов (исходного и восстановленного) игнорируют до тех пор, пока не пройдет 50 мс после того, как значение общей громкости превысит в обоих каналах одного из сигналов значение порога, равное 0,1.

Условие превышения порога имеет вид:

$$\begin{aligned}&(N_{totT}[n] \geq L_t) \wedge (N_{totR}[n] \geq L_t), && \text{для моносигнала}\\ &[(N_{totT1}[n] \geq L_t) \wedge (N_{totR1}[n] \geq L_t)] \vee [(N_{totT2}[n] \geq L_t) \wedge (N_{totR2}[n] \geq L_t)], && \text{стереосигнала,}\end{aligned} \tag{A.82}$$

где $L_t = 0,1$.

Количество пропускаемых после превышения порога фреймов N_{off} рассчитывают по формуле (А.83)

$$N_{off} = 0,05F_{ss}. \tag{A.83}$$

А.1.2.3.6 Ширина полос исходного и восстановленного аудиосигналов (BandwidthRefB и BandwidthTestB)

Операции вычислений ширины полос исходного и восстановленного аудиосигналов описывают в терминах операций на выходных значениях ДПФ, выраженных в децибелах. Прежде всего для каждого фрейма выполняют следующие операции:

- для восстановленного сигнала: находят самую большую компоненту после частоты 21600 Гц. Это значение называют уровнем порога;
- для исходного сигнала: выполняя поиск вниз, начиная с частоты 21600 Гц, находят первое значение, которое превышает значение уровня порога на 10 дБ. Соответствующую этому значению частоту называют шириной полосы для исходного сигнала;
- для восстановленного сигнала: выполняя поиск вниз, начиная со значения ширины полосы исходного сигнала, находят первое значение, которое превышает значение уровня порога на 5 дБ. Обозначают соответствующую этому значению частоту как ширину полосы для восстановленного сигнала.

Если найденные частоты для исходного сигнала не превышают 8100 Гц, то ширину полосы для этого фрейма игнорируют.

Значения ширин полос для всех фреймов называют основными частотами ДПФ.

Основную частоту ДПФ для n-го фрейма обозначают как $K_R[n]$для исходного сигнала и как $K_T[n]$— для восстановленного сигнала. Вычисление конечных значений переменных психоакустической модели, значений ширин полос исходного и восстановленного сигналов необходимо выполнять по следующим формулам соответственно:

$$W_R = \frac{1}{N}\sum_{n=0}^{N-1} K_R[n], \tag{A.84}$$

$$W_T = \frac{1}{N}\sum_{n=0}^{N-1} K_T[n], \tag{A.85}$$

где суммирование осуществляют только для тех фреймов, в которых основная частота ДПФ превышает 8100 Гц.

А.1.2.3.7 Отношение уровня шума к порогу маскирования (TotalNMRB)

Порог маскирования $M[k, n]$ вычисляют по формуле (А.86)

$$M[k, n] = \frac{\tilde{E}_{xR}[k,n]}{10^{m_{dB}[k]/10}} = g_m[k]\tilde{E}_{xR}[k, n], \tag{A.86}$$

где

$$m_{dB}[k] = \begin{cases} 3, & k \le 12/\Delta z, \\ 0{,}25k\Delta z, & k > 12/\Delta z. \end{cases} \tag{A.87}$$

Уровень шума, $X^2_{wN}[k]$, вычисляют по формуле (А.88)

$$X^2_{wN}[k] = |X_w[k]|^2 - 2\sqrt{|X_{wT}[k]|^2 |X_{wR}[k]|^2} + |X_{wR}[k]|^2, \tag{A.88}$$

где k — индекс основной частоты ДПФ.

Отношение уровня шума к порогу маскирования в k-й полосе частот выражают формулой (А.89)

$$R_{NM}[k, n] = \frac{E_{bN}[k,n]}{M[k,n]} = \frac{E_{bN}[k,n]}{g_m[k]\tilde{E}_{xR}[k,n]}. \tag{A.89}$$

Конечное отношение уровня шума к порогу маскирования R_{NMtot}, дБ, вычисляют по формуле (А.90)

$$R_{NMtot} = 10\log_{10}\left(\frac{1}{N}\sum_{n=0}^{N-1}\frac{1}{N_C}\sum_{k=0}^{N_C-1} R_{NM}[k,n]\right). \tag{A.90}$$

А.1.2.3.8 Относительное искажение фреймов (RelDistFramesB)

Максимальное отношение шума к порогу маскирования фрейма $R_{N\max}[n]$ вычисляют по формуле (А.91):

$$R_{N\max}[n] = \max_{0\le k\le N_C-1}(R_{NM}[k,n]). \tag{A.91}$$

Искаженным считают тот фрейм, в котором максимальное отношение шума к порогу маскирования более 1,5 дБ.

Конечное значение выходной переменной психоакустической модели представляет собой отношение количества искаженных фреймов к общему количеству фреймов.

А.1.2.3.9 Максимальная вероятность обнаружения искажения (MFPDB)

Сначала вычисляют асимметричное возбуждение:

$$L[k, n] = \begin{cases} 0{,}3\tilde{E}_{xRdB}[k,n] + 0{,}7\tilde{E}_{xTdB}[k,n], & \tilde{E}_{xR}[k,n] > \tilde{E}_{xT}[k,n], \\ \tilde{E}_{xTdB}[k,n], & \tilde{E}_{xR}[k,n] \le \tilde{E}_{xT}[k,n], \end{cases} \tag{A.92}$$

где $\tilde{E}_{xRdB}[k,n] = 10\log_{10}(\tilde{E}_{xR}[k,n])$,
$\tilde{E}_{xTdB}[k,n] = 10\log_{10}(\tilde{E}_{xT}[k,n])$. (A.93)

Далее вычисляют шаг для обнаружения искажения, $s[k, n]$, по формуле

$$S[k, n] = \begin{cases} c_0 + c_1L[k,n] + c_2L^2[k,n] + c_3L^3[k,n] + c_4L^4[k,n] + d_1\left(\frac{d_2}{L[k,n]}\right)^{\gamma}, & L[k, n] > 0, \\ 1\cdot 10^{30}, & L[k,n] \le 0, \end{cases} \tag{A.94}$$

где $c_0 = -0{,}198719$; $c_1 = 0{,}0550197$; $c_2 = -0{,}00102438$;
$c_3 = 5{,}05622\cdot 10^{-6}$; $c_4 = 9{,}01033\cdot 10^{-11}$; $d_1 = 5{,}95072$. (A.95)
$d_2 = 6{,}39468$; $\gamma = 1{,}71332$;

Вероятность обнаружения вычисляют по формуле

$$P_c[k, n] = 1 - (0{,}5)^{\left|\frac{\tilde{E}_{sRdB}[k,n] - \tilde{E}_{sTdB}[k,n]}{s[k,n]}\right|^b}, \quad (A.96)$$

где показатель степени b вычисляют по формуле

$$b = \begin{cases} 4 & \tilde{E}_{sR}[k,n] > \tilde{E}_{sT}[k,n], \\ 6 & \tilde{E}_{sR}[k,n] \le \tilde{E}_{sT}[k,n]. \end{cases} \quad (A.97)$$

Затем вычисляют количество шагов сверх порога вероятности обнаружения, $q_c[k, n]$, по формуле

$$q_c[k, n] = \frac{\left|\left[\tilde{E}_{sRdB}[k,n] - \tilde{E}_{sTdB}[k,n]\right]\right|}{s[k,n]}. \quad (A.98)$$

Характеристики по формулам (А.96) и (А.98) вычисляют для каждого канала сигнала. Для каждой частоты и времени полную вероятность обнаружения и полное число шагов сверх порога выбирают как большее значение из всех каналов:

$$P_b[n] = 1 - \prod_{k=0}^{N_c-1} (1 - \max(p_1[k, n], p_2[k, n])), \quad (A.99)$$

$$Q_b[n] = \sum_{k=0}^{N_c-1} \max(q_1[k, n], q_2[k, n]), \quad (A.100)$$

где индексы 1 и 2 — номера каналов.

Для одноканальных сигналов характеристики вычисляют по формулам (А.101) и (А.102):

$$P_b[n] = 1 - \prod_{k=0}^{N_c-1} (1 - p_c[k, n]), \quad (A.101)$$

$$Q_b[n] = \sum_{k=0}^{N_c-1} q_c[k, n]. \quad (A.102)$$

Выполняют следующие вычисления:

$$\tilde{P}_b[n] = c_0\tilde{P}_b[n-1] + (1 - c_0)P_b[n], \quad (A.103)$$

где $c_0 = 0{,}9$ и начальное условие — нулевое.

Максимальную вероятность обнаружения искажения $\tilde{P}_M[n]$ вычисляют по рекуррентной формуле

$$\tilde{P}_M[n] = \max(P_M[n-1], \tilde{P}_b[n]). \quad (A.104)$$

Конечное значение выходной переменной психоакустической модели $MFPD_B$ рассчитывают по формуле (А.105)

$$MFPD_B = \tilde{P}_M[N-1]. \quad (A.105)$$

А.1.2.3.10 Среднее поблочное искажение (ADBB)

Сначала вычисляют сумму общего числа шагов сверх порога обнаружения Q_s по формуле

$$Q_s = \sum_{n=0}^{N-1} Q_b[n]. \quad (A.106)$$

Причем суммирование ведут для всех значений, для которых $P_b[n] > 0{,}5$.

Конечная характеристика ADB_B имеет вид:

$$ADB_B = \begin{cases} 0, & N = 0, \\ \log_{10}\left(\frac{Q_s}{N}\right), & N > 0,\ Q_s > 0, \\ -0{,}5 & N > 0,\ Q_s = 0. \end{cases} \quad (A.107)$$

А.1.2.3.11 Гармоническая структура ошибки (EHSB)

Выходы ДПФ для исходного и восстановленного сигналов обозначают как $X_R[k]$ и $X_T[k]$ соответственно.

Вычисляют характеристику $D[k]$:

$$D[k] = \log\left(|W[k]X_T[k]|^2\right) - \log\left(|W[k]X_R[k]|^2\right) = \log\left(\frac{|X_T[k]|^2}{|X_R[k]|^2}\right), \quad 0 \le K \le N_F/2. \tag{A.108}$$

Формируют вектор длины M из значений $D[k]$:

$$D_t = (D[l], \ldots, D[l + M - 1])^T, \tag{A.109}$$

где $M = L_{max} = 2^{\lceil \log_2(N_F \cdot 8000/F_s) \rceil - 5} = 256$.

Нормализованную автокорреляцию вычисляют по формуле (А.110)

$$C(l, t) = \frac{D_t D_{(t+l)}}{\sqrt{|D_t|^2 |D_{t+l}|^2}}, \tag{A.110}$$

где $l \in [0; L_{max}]$.

При $C[l] = C(l, 0)$ необходимо вычислить:

$$|D_l|^2 = \begin{cases} |D_0|^2, & l = 0, \\ |D_{l-1}|^2 + D[l + M - 1]^2 - D[l - 1]^2 & 1 \le l \le L_{max}. \end{cases} \tag{A.111}$$

При вычислении по формуле (А.110) в случае, если сигналы равны, необходимо установить нормализованную автокорреляцию равной единице, чтобы избежать деления на ноль.

Вводят оконную функцию:

$$H[l] = \begin{cases} 0{,}5\sqrt{8/3}\left(1 - \cos\left(\frac{2\pi l}{L_{max} - 1}\right)\right), & 0 < l < L_{max} - 1, \\ 0, & \text{в остальных случаях.} \end{cases} \tag{A.112}$$

К нормализованной автокорреляции применяют оконное преобразование по формуле (А.113)

$$C_H[m] = H[m](C[m + 1] - \overline{C}), \qquad 0 \le m \le L_{max} - 1, \tag{A.113}$$

$$\text{где } \overline{C} = \frac{1}{L_{max}} \sum_{l=1}^{L_{max}} C[l], \tag{A.114}$$

Спектр мощности $S[k]$ вычисляют по формуле (А.115)

$$S[k] = \left| \frac{1}{L_{max}} \sum_{l=0}^{L_{max}-1} C_H[l] e^{-j2\pi kl/L_{max}} \right|^2 \tag{A.115}$$

Поиск максимального пика спектра мощности начинают с $k = 1$ и заканчивают при $S[k] > S[k - 1]$ или $k > L_{max}/2$. Найденное значение максимального пика обозначают как $E_{H\,max}[n]$ Тогда конечное значение выходной переменной психоакустической модели рассчитывают по формуле (А.116)

$$E_{H_0} = (1000/N) \sum_{n=0}^{N-1} E_{H\,max}(n). \tag{A.116}$$

При вычислении этого значения исключают фреймы с низкой энергией. Для определения фреймов с низкой энергией вводят пороговое значение

$$A^2_{thr} = 8000(A_{max}/32768)^2, \tag{A.117}$$

где $A_{max} = 32768$ для амплитуд, хранимых в виде 16 битного целого числа.

Энергию фрейма A^2 оценивают по формуле (А.118):

$$A^2 = \sum_{n=N_F/2}^{N_F-1} x^2[n]. \tag{A.118}$$

При вычислении гармонической структуры ошибки фрейм игнорируют, если:

$$\begin{aligned} &(A^2_T < A^2_{thr}) \wedge (A^2_R < A^2_{thr}), && \text{для моносигнала,} \\ &(A^2_{T1} < A^2_{thr}) \wedge (A^2_{R1} < A^2_{thr}) \wedge (A^2_{T2} < A^2_{thr}) \wedge (A^2_{R2} < A^2_{thr}), && \text{для стереосигнала.} \end{aligned} \tag{A.119}$$

А.1.2.4 Нормирование значений выходных переменных психоакустической модели

Нормирование полученных на предыдущем шаге значений выходных переменных психоакустической модели выполняют по формуле (А.120)

$$M_i'[i] = \frac{M_i[i] - a_{min}[i]}{a_{max}[i] - a_{min}[i]} \quad (A.120)$$

где $M_i[i]$ — значение i-й выходной переменной психоакустической модели, а значения $a_{min}[i]$ и $a_{max}[i]$ приведены в таблице А.2.

Таблица А.2 — Константы для нормирования значений выходных переменных психоакустической модели

Индекс переменой, i	Наименование переменной	Минимальное значение $a_{min}[i]$	Максимальное значение $a_{max}[i]$
0	BandwidthRefB	393,916656	921,0
1	BandwidthTestB	361,965332	881,131226
2	TotalNMRB	−24,045116	16,212030
3	WinModDiff1B	1,110661	107,137772
4	ADBB	−0,206623	2,886017
5	EHSB	0,074318	13,933351
6	AvgModDiff1B	1,113683	63,257874
7	AvgModDiff2B	0,950345	1145,018555
8	RmsNoiseLoudB	0,029985	14,819740
9	MFPDB	0,000101	1,0
10	RelDistFramesB	0,0	1,0

А.1.2.5 Оценка качества восстановленного сигнала с помощью искусственной нейронной сети

На вход нейронной сети подают значения 11 выходных переменных психоакустической модели, рассчитанных в А.1.2.1—А.1.2.4. Нейронная сеть имеет 11 нейронов во входном слое, один скрытый слой с тремя нейронами и один нейрон в выходном слое. Выход нейронной сети — конечное значение метрики PEAQ рассчитывают по формуле (А.121)

$$PEAQ = b_{min} + (b_{max} + b_{min}) sig(D_j). \quad (A.121)$$

где $b_{min} = -3,98$ и $b_{max} = 0,22$, а функция $sig(x)$ — асимметричная сигмоида.

$$sig(x) = \frac{1}{1 + e^{-x}}. \quad (A.122)$$

Значение D_j вычисляют следующим образом:

$$D_j = w_{yb} + \sum_{j=0}^{J-1} \left[w_y[j] sig\left(w_{xb}[j] + \sum_{i=0}^{I-1} w_x[i, j] M_i'[i] \right) \right], \quad (A.123)$$

где $M_i'[i]$ — нормализованное значение i-й выходной переменной, I — количество выходных переменных (равное 11), J — количество нейронов в скрытом слое (равное трем), $[w_x[i, j], (w_{xb}[j], w_y[j], w_{yb}$ — значения весов и смещений нейронной сети, приведенные в таблицах А.3—А.5.

Таблица А.3 — Веса нейронной сети

Индекс i	Вес $w_x[i, 0]$	Вес $w_x[i, 1]$	Вес $w_x[i, 2]$
0	−0,502657	0,436333	1,219602
1	4,307481	3,246017	1,123743
2	4,984241	2,211189	−0,192096
3	0,0511056	−1,762424	4,331315
4	2,321580	1,789971	−0,754560
5	−5,303901	−3,452257	−10,814982

Окончание таблицы А.3

Индекс i	Вес $w_x[i, 0]$	Вес $w_x[i, 1]$	Вес $w_x[i, 2]$
6	2,730991	–6,111805	1,519223
7	0,624950	–1,331523	–5,955151
8	3,10288	0,871260	–5,922878
9	–1,051468	–0,939882	–0,142913
10	–1,804679	–0,503610	0,620456

Таблица А.4 — Смещения нейронной сети

Bias w_{yb}	Bias $w_{xb}[0]$	Bias $w_{xb}[1]$	Bias $w_{xb}[2]$
–0,307594	–2,518254	0,654841	–2,207228

Таблица А.5 — Веса и смещения нейронной сети

Индекс j	Вес $w_y[j]$
0	–3,817048
1	4,017138
2	4,629582

Это значение метрики (PEAQ) представляет собой вещественное число, принадлежащее отрезку [–3,98; 0,22].

А.2 Алгоритм расчета метрики PSNR

Пиковое отношение сигнал/шум между исходным аудиосигналом X_R и восстановленным X_T рассчитывают по формулам:

$$\mathrm{PSNR} = 10\lg \frac{\max X_R^2}{S}, \tag{А.124}$$

$$S = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(d_i - \overline{d})^2}, \tag{А.125}$$

где разности значений сигналов d_i и их математическое ожидание $\overline{d}$ рассчитывают по формулам:

$$d_i = X_{R_i} - X_{T_i}, \quad \overline{d} = \frac{1}{n}\sum_{i=1}^{n} d_i, \tag{А.126}$$

где X_{R_i} и X_{T_i} — i-е оцифрованные значения исходного и восстановленного аудиосигналов соответственно, $i = 1, 2, \ldots, n$,

$\max X_R$ — максимальное значение среди оцифрованных значений исходного аудиосигнала.

А.3 Алгоритм расчета метрики «коэффициент различия форм сигналов»

Пусть X_R — исходный моноканальный аудиосигнал (либо один канал исходного многоканального аудиосигнала), а X_T — восстановленный моноканальный аудиосигнал (либо один канал восстановленного многоканального аудиосигнала). Оба сигнала состоят из одинакового количества значений N.

Массивы значений амплитуд сигналов X_R и X_T представляют в виде относительного изменения значений амплитуд сигнала:

$$dX_R[i] = X_R[i] - X_R[i-1], \; i = \overline{2,N}, \tag{А.127}$$

$$dX_T[i] = X_T[i] - X_T[i-1], \; i = \overline{2,N}. \tag{А.128}$$

Значение метрики «коэффициент различия форм сигналов» K вычисляют как среднеквадратическое отклонение массивов значений амплитуд dX_R и dX_T

$$K = \frac{\sum_{i=1}^{N} (dX_R - dX_T)^2}{N}. \quad (A.129)$$

А.4 Алгоритм расчета коэффициента сжатия

Пусть S_0 — объем памяти, который занимают исходные аудиоданные, а S_c — объем памяти, который занимают сжатые данные, тогда коэффицент сжатия k рассчитывают по формуле

$$k = \frac{S_0}{S_c}. \quad (A.130)$$

Приложение Б
(рекомендуемое)

Листинги программ расчета метрик качества аудиоданных

Б.1 Листинг программы расчета метрики PEAQ на языке Matlab

```
function ODG = PQevalAudio (Fref, Ftest, StartS, EndS)
% Оценка качества аудиоданных с точки зрения восприятия (Perceptual evaluation of audio quality)

% - StartS – индекс значения, соответствующего началу сигнала.
% - EndS – индекс значения, соответствующего концу сигнала.

% глобальные переменные
global MOVC PQopt

% параметры
NF = 2048;
Nadv = NF / 2;
Version = 'Basic';

% настройки
PQopt.ClipMOV = 0;
PQopt.PCinit = 0;
PQopt.PDfactor = 1;
PQopt.Ni = 1;
PQopt.DelayOverlap = 1;
PQopt.DataBounds = 1;
PQopt.EndMin = NF / 2;

addpath ('CB', 'MOV', 'Misc', 'Patt');

if (nargin < 3)
   StartS = [0, 0];
end
if (nargin < 4)
   EndS = [ ];
end

% вычислить количество значений и каналов для каждого входного файла
WAV(1) = PQwavFilePar (Fref);
WAV(2) = PQwavFilePar (Ftest);

% согласовать размеры файлов
PQ_CheckWAV (WAV);
if (WAV(1).Nframe ~= WAV(2).Nframe)
   disp ('>>> Number of samples differ: using the minimum');
end

% границы данных
Nchan = WAV(1).Nchan;
[StartS, Fstart, Fend] = PQ_Bounds (WAV, Nchan, StartS, EndS, PQopt);

% фреймов PEAQ
Np = Fend - Fstart + 1;
if (PQopt.Ni < 0)
   PQopt.Ni = ceil (Np / abs(PQopt.Ni));
end

% инициализация структуры MOV
MOVC = PQ_InitMOVC (Nchan, Np);

Nc = PQCB (Version);
for (j = 0:Nchan-1)
   Fmem(j+1) = PQinitFMem (Nc, PQopt.PCinit);
end

is = 0;
for (i = -Fstart:Np-1)
```

```
    % считать фрейм данных
    xR = PQgetData (WAV(1), StartS(1) + is, NF);    % Reference file
    xT = PQgetData (WAV(2), StartS(2) + is, NF);    % Test file
    is = is + Nadv;

    % обработка фрейма
    for (j = 0:Nchan-1)
      [MOVI(j+1), Fmem(j+1)] = PQeval (xR(j+1,:), xT(j+1,:), Fmem(j+1));
    end

    if (i >= 0)
      % вывести MOV в новую структуру
      PQframeMOV (i, MOVI);   % выходные значения теперь в глобальной переменной MOVC

      % вывод значений
      if (PQopt.Ni ~= 0 & mod (i, PQopt.Ni) == 0)
%         PQprtMOVCi (Nchan, i, MOVC);
      end
    end
end

% усреднение по времени значений MOV
if (PQopt.DelayOverlap)
  Nwup = Fstart;
else
  Nwup = 0;
end
MOVB = PQavgMOVB (MOVC, Nchan, Nwup);

% запуск нейронной сети
ODG = PQnNet (MOVB);

% совокупный вывод значений
% PQprtMOV (MOVB, ODG);

%----------
function PQ_CheckWAV (WAV)
% проверка файлов

Fs = 48000;

if (WAV(1).Nchan ~= WAV(2).Nchan)
  error ('>>> Number of channels differ');
end
if (WAV(1).Nchan > 2)
  error ('>>> Too many input channels');
end
if (WAV(1).Nframe ~= WAV(2).Nframe)
  disp ('>>> Number of samples differ');
end
if (WAV(1).Fs ~= WAV(2).Fs)
  error ('>>> Sampling frequencies differ');
end
if (WAV(1).Fs ~= Fs)
  error ('>>> Invalid Sampling frequency: only 48 kHz supported');
end

%----------
function [StartS, Fstart, Fend] = PQ_Bounds (WAV, Nchan, StartS, EndS, PQopt)

PQ_NF = 2048;
PQ_NADV = (PQ_NF / 2);

if (isempty (StartS))
  StartS(1) = 0;
  StartS(2) = 0;
elseif (length (StartS) == 1)
  StartS(2) = StartS(1);
end
```

```
Ns = WAV(1).Nframe;
% границы данных
if (PQopt.DataBounds)
   Lim = PQdataBoundary (WAV(1), Nchan, StartS(1), Ns);
   fprintf ('PEAQ Data Boundaries: %ld (%.3f s) - %ld (%.3f s)\n', ...
         Lim(1), Lim(1)/WAV(1).Fs, Lim(2), Lim(2)/WAV(1).Fs);
else
   Lim = [Starts(1), StartS(1) + Ns - 1];
end

% номер первого фрейма
Fstart = floor ((Lim(1) - StartS(1)) / PQ_NADV);

% номер последнего фрейма
Fend = floor ((Lim(2) - StartS(1) + 1 - PQopt.EndMin) / PQ_NADV);

%---------
function MOVC = PQ_InitMOVC (Nchan, Np)
MOVC.MDiff.Mt1B = zeros (Nchan, Np);
MOVC.MDiff.Mt2B = zeros (Nchan, Np);
MOVC.MDiff.Wt   = zeros (Nchan, Np);
MOVC.NLoud.NL   = zeros (Nchan, Np);
MOVC.Loud.NRef  = zeros (Nchan, Np);
MOVC.Loud.NTest = zeros (Nchan, Np);
MOVC.BW.BWRef  = zeros (Nchan, Np);
MOVC.BW.BWTest = zeros (Nchan, Np);
MOVC.NMR.NMRavg = zeros (Nchan, Np);
MOVC.NMR.NMRmax = zeros (Nchan, Np);
MOVC.PD.Pc = zeros (1, Np);
MOVC.PD.Qc = zeros (1, Np);
MOVC.EHS.EHS = zeros (Nchan, Np);
```

```
function ODG = PQnNetB (MOV)
% нейронная сеть для получения конечного значения метрики

persistent amin amax wx wxb wy wyb bmin bmax I J CLIPMOV
global PQopt

if (isempty (amin))
   I = length (MOV);
   if (I == 11)
      [amin, amax, wx, wxb, wy, wyb, bmin, bmax] = NNetPar ('Basic');
   else
      [amin, amax, wx, wxb, wy, wyb, bmin, bmax] = NNetPar ('Advanced');
   end
   [I, J] = size (wx);
end

sigmoid = inline ('1 / (1 + exp(-x))');

% Scale the MOV's
Nclip = 0;
MOVx = zeros (1, I);
for (i = 0:I-1)
   MOVx(i+1) = (MOV(i+1) - amin(i+1)) / (amax(i+1) - amin(i+1));
   if (~ isempty (PQopt) & PQopt.ClipMOV ~= 0)
      if (MOVx(i+1) < 0)
         MOVx(i+1) = 0;
         Nclip = Nclip + 1;
      elseif (MOVx(i+1) > 1)
         MOVx(i+1) = 1;
         Nclip = Nclip + 1;
      end
   end
end
if (Nclip > 0)
   fprintf ('>>> %d MOVs clipped\n', Nclip);
end
```

```
% нейронная сеть
DI = wyb;
for (j = 0:J-1)
   arg = wxb(j+1);
   for (i = 0:I-1)
      arg = arg + wx(i+1,j+1) * MOVx(i+1);
   end
   DI = DI + wy(j+1) * sigmoid (arg);
end

ODG = bmin + (bmax - bmin) * sigmoid (DI);

function [amin, amax, wx, wxb, wy, wyb, bmin, bmax] = NNetPar (Version)

if (strcmp (Version, 'Basic'))
   amin = ...
      [393.916656,  361.965332,  -24.045116,    1.110661,  -0.206623, ...
        0.074318,    1.113683,    0.950345,    0.029985,   0.000101, ...
        0];
   amax = ...
      [921,        881.131226,   16.212030,  107.137772,   2.886017, ...
       13.933351,   63.257874, 1145.018555,   14.819740,   1,        ...
        1];
   wx = ...
      [ [ -0.502657,  0.436333,   1.219602 ];
        [  4.307481,  3.246017,   1.123743 ];
        [  4.984241, -2.211189,  -0.192096 ];
        [  0.051056, -1.762424,   4.331315 ];
        [  2.321580,  1.789971,  -0.754560 ];
        [ -5.303901, -3.452257, -10.814982 ];
        [  2.730991, -6.111805,   1.519223 ];
        [  0.624950, -1.331523,  -5.955151 ];
        [  3.102889,  0.871260,  -5.922878 ];
        [ -1.051468, -0.939882,  -0.142913 ];
        [ -1.804679, -0.503610,  -0.620456 ] ];
   wxb = ...
      [ -2.518254,  0.654841, -2.207228 ];
   wy = ...
      [ -3.817048,  4.107138,  4.629582 ];
   wyb = -0.307594;
   bmin = -3.98;
   bmax = 0.22;
else
   amin = ...
      [  13.298751,  0.041073,  -25.018791,  0.061560,   0.024523 ];
   amax = ...
      [ 2166.5,     13.24326,    13.46708,  10.226771,  14.224874 ];
   wx = ...
      [ [  21.211773, -39.913052, -1.382553, -14.545348,  -0.320899 ];
        [  -8.981803,  19.956049,  0.935389,  -1.686586,  -3.238586 ];
        [   1.633830,  -2.877505, -7.442935,   5.606502,  -1.783120 ];
        [   6.103821,  19.587435, -0.240284,   1.088213,  -0.511314 ];
        [  11.556344,   3.892028,  9.720441,  -3.287205, -11.031250 ] ];
   wxb = ...
      [   1.330890,   2.686103,    2.096598, -1.327851,   3.087055 ];
   wy = ...
      [ -4.696996,   -3.289959,    7.004782,  6.651897,   4.009144 ];
   wyb = -1.360308;
   bmin = -3.98;
   bmax = 0.22;
end
```

```
function [Nc, fc, fl, fu, dz] = PQCB (Version)
% параметры критической полосы пропускания

B = inline ('7 * asinh (f / 650)');
Bl = inline ('650 * sinh (z / 7)');

fL = 80;
fU = 18000;
if (strcmp (Version, 'Basic'))
   dz = 1/4;
elseif (strcmp (Version, 'Advanced'))
   dz = 1/2;
else
   error ('PQCB: Invalid version');
end

zL = B(fL);
zU = B(fU);
Nc = ceil((zU - zL) / dz);
zl = zL + (0:Nc-1) * dz;
zu = min (zL + (1:Nc) * dz, zU);
zc = 0.5 * (zl + zu);

fl = Bl (zl);
fc = Bl (zc);
fu = Bl (zu);

if (strcmp (Version, 'Basic'))
   fl = [   80.000,   103.445,   127.023,   150.762,   174.694, ...
           198.849,   223.257,   247.950,   272.959,   298.317, ...
           324.055,   350.207,   376.805,   403.884,   431.478, ...
           459.622,   488.353,   517.707,   547.721,   578.434, ...
           609.885,   642.114,   675.161,   709.071,   743.884, ...
           779.647,   816.404,   854.203,   893.091,   933.119, ...
           974.336,  1016.797,  1060.555,  1105.666,  1152.187, ...
          1200.178,  1249.700,  1300.816,  1353.592,  1408.094, ...
          1464.392,  1522.559,  1582.668,  1644.795,  1709.021, ...
          1775.427,  1844.098,  1915.121,  1988.587,  2064.590, ...
          2143.227,  2224.597,  2308.806,  2395.959,  2486.169, ...
          2579.551,  2676.223,  2776.309,  2879.937,  2987.238, ...
          3098.350,  3213.415,  3332.579,  3455.993,  3583.817, ...
          3716.212,  3853.817,  3995.399,  4142.547,  4294.979, ...
          4452.890,  4616.482,  4785.962,  4961.548,  5143.463, ...
          5331.939,  5527.217,  5729.545,  5939.183,  6156.396, ...
          6381.463,  6614.671,  6856.316,  7106.708,  7366.166, ...
          7635.020,  7913.614,  8202.302,  8501.454,  8811.450, ...
          9132.688,  9465.574,  9810.536, 10168.013, 10538.460, ...
         10922.351, 11320.175, 11732.438, 12159.670, 12602.412, ...
         13061.229, 13536.710, 14029.458, 14540.103, 15069.295, ...
         15617.710, 16186.049, 16775.035, 17385.420 ];
   fc = [   91.708,   115.216,   138.870,   162.702,   186.742, ...
           211.019,   235.566,   260.413,   285.593,   311.136, ...
           337.077,   363.448,   390.282,   417.614,   445.479, ...
           473.912,   502.950,   532.629,   562.988,   594.065, ...
           625.899,   658.533,   692.006,   726.362,   761.644, ...
           797.898,   835.170,   873.508,   912.959,   953.576, ...
           995.408,  1038.511,  1082.938,  1128.746,  1175.995, ...
          1224.744,  1275.055,  1326.992,  1380.623,  1436.014, ...
          1493.237,  1552.366,  1613.474,  1676.641,  1741.946, ...
          1809.474,  1879.310,  1951.543,  2026.266,  2103.573, ...
          2183.564,  2266.340,  2352.008,  2440.675,  2532.456, ...
          2627.468,  2725.832,  2827.672,  2933.120,  3042.309, ...
          3155.379,  3272.475,  3393.745,  3519.344,  3649.432, ...
          3784.176,  3923.748,  4068.324,  4218.090,  4373.237, ...
          4533.963,  4700.473,  4872.978,  5051.700,  5236.866, ...
          5428.712,  5627.484,  5833.434,  6046.825,  6267.931, ...
          6497.031,  6734.420,  6980.399,  7235.284,  7499.397, ...
          7773.077,  8056.673,  8350.547,  8655.072,  8970.639, ...
          9297.648,  9636.520,  9987.683, 10351.586, 10728.695, ...
```

```
        11119.490, 11524.470, 11944.149, 12379.066, 12829.775, ...
        13294.850, 13780.887, 14282.503, 14802.338, 15341.057, ...
        15899.345, 16477.914, 17077.504, 17690.045 ];
    fu = [ 103.445,   127.023,   150.762,   174.694,   198.849, ...
           223.257,   247.950,   272.959,   298.317,   324.055, ...
           350.207,   376.805,   403.884,   431.478,   459.622, ...
           488.353,   517.707,   547.721,   578.434,   609.885, ...
           642.114,   675.161,   709.071,   743.884,   779.647, ...
           816.404,   854.203,   893.091,   933.113,   974.336, ...
          1016.797,  1060.555,  1105.666,  1152.187,  1200.178, ...
          1249.700,  1300.816,  1353.592,  1408.094,  1464.392, ...
          1522.559,  1582.668,  1644.795,  1709.021,  1775.427, ...
          1844.098,  1915.121,  1988.587,  2064.590,  2143.227, ...
          2224.597,  2308.806,  2395.959,  2486.169,  2579.551, ...
          2676.223,  2776.309,  2879.937,  2987.238,  3098.350, ...
          3213.415,  3332.579,  3455.993,  3583.817,  3716.212, ...
          3853.348,  3995.399,  4142.547,  4294.979,  4452.890, ...
          4643.482,  4785.962,  4961.548,  5143.463,  5331.939, ...
          5527.217,  5729.545,  5939.183,  6156.396,  6381.463, ...
          6614.671,  6856.316,  7106.708,  7366.166,  7635.020, ...
          7913.614,  8202.302,  8501.454,  8811.450,  9132.688, ...
          9465.574,  9810.536, 10168.013, 10538.460, 10922.351, ...
         11320.175, 11732.438, 12159.670, 12602.412, 13061.229, ...
         13536.710, 14029.458, 14540.103, 15069.295, 15617.710, ...
         16186.049, 16775.035, 17385.420, 18000.000 ];
end
```

```
function Es = PQspreadCB (E, Ver)
% распространение возбуждений
% E и Es - энергии

persistent Bs Version

if (~ strcmp (Ver, Version))
   Version = Ver;
   Nc = length (E);
   Bs = PQ_SpreadCB (ones(1,Nc), ones(1,Nc), Version);
end

Es = PQ_SpreadCB (E, Bs, Version);

%------------------------
function Es = PQ_SpreadCB (E, Bs, Ver);

persistent Nc dz fc aL aUC Version

e = 0.4;

if (~ strcmp (Ver, Version))
   Version  = Ver;
   [Nc, fc, fl, fu, dz] = PQCB (Version);
end

% выделение памяти
aUCEe = zeros (1, Nc);
Ene = zeros (1, Nc);
Es = zeros (1, Nc);

% вычисление термов, зависящих от энергии
aL = 10^(-2.7 * dz);
for (m = 0:Nc-1)
   aUC = 10^((-2.4 - 23 / fc(m+1)) * dz);
   aUCE = aUC * E(m+1)^(0.2 * dz);
   glL = (1 - aL^(m+1)) / (1 - aL);
   glU = (1 - aUCE^(Nc-m)) / (1 - aUCE);
   En = E(m+1) / (glL + glU - 1);
   aUCEe(m+1) = aUCE^e;
   Ene(m+1) = En^e;
end
```

```
% распространение вниз
Es(Nc-1+1) = Ene(Nc-1+1);
aLe = aL^e;
for (m = Nc-2:-1:0)
   Es(m+1) = aLe * Es(m+1+1) + Ene(m+1);
end

% распространение вверх i > m
for (m = 0:Nc-2)
   r = Ene(m+1);
   a = aUCEe(m+1);
   for (i = m+1:Nc-1)
      r = r * a;
      Es(i+1) = Es(i+1) + r;
   end
end

for (i = 0:Nc-1)
   Es(i+1) = (Es(i+1))^(1/e) / Bs(i+1);
end
```

```
function Eb = PQgroupCB (X2, Ver)
% группировка вектора энергии ДПФ и критической полосы пропускания
% X2 – вектор значений, возведенных в степень 2
% Eb – вектор возбуждений

persistent Nc kl ku Ul Uu Version

Emin = 1e-12;

if (~ strcmp (Ver, Version))
   Version = Ver;

   NF = 2048;
   Fs = 48000;
   [Nc, kl, ku, Ul, Uu] = PQ_CBMapping (NF, Fs, Version);
end

% выделение памяти
Eb = zeros (1, Nc);

% вычисление возбуждений в каждой полосе
for (i = 0:Nc-1)
   Ea = Ul(i+1) * X2(kl(i+1)+1);
   for (k = (kl(i+1)+1):(ku(i+1)-1))
      Ea = Ea + X2(k+1);
   end
   Ea = Ea + Uu(i+1) * X2(ku(i+1)+1);
   Eb(i+1) = max(Ea, Emin);
end

%----------------------------------------
function [Nc, kl, ku, Ul, Uu] = PQ_CBMapping (NF, Fs, Version)

[Nc, fc, fl, fu] = PQCB (Version);

df = Fs / NF;
for (i = 0:Nc-1)
   fli = fl(i+1);
   fui = fu(i+1);
   for (k = 0:NF/2)
      if ((k+0.5)*df > fli)
         kl(i+1) = k;
         Ul(i+1) = (min(fui, (k+0.5)*df) ...
                  - max(fli, (k-0.5)*df)) / df;
         break;
      end
   end
```

```
    for (k = NF/2:-1:0)
      if ((k-0.5)*df < fui)
        ku(i+1) = k;
        if (kl(i+1) == ku(i+1))
          Uu(i+1) = 0;
        else
          Uu(i+1) = (min(fui, (k+0.5)*df) ...
                - max(fli, (k-0.5)*df)) / df;
        end
        break;
      end
    end
  end
```

```
function [MOVI, Fmem] = PQeval (xR, xT, Fmem)
% PEAQ – обработка единичного фрейма

NF = 2048;
Version = 'Basic';

% оконное ДПФ
X2(1,:) = PQDFTFrame (xR);
X2(2,:) = PQDFTFrame (xT);

[EbN, Es] = PQ_excitCB (X2);

[Ehs(1,:), Fmem.TDS.Ef(1,:)] = PQ_timeSpread (Es(1,:), Fmem.TDS.Ef(1,:));
[Ehs(2,:), Fmem.TDS.Ef(2,:)] = PQ_timeSpread (Es(2,:), Fmem.TDS.Ef(2,:));

% адаптация паттернов возбуждения
[EP, Fmem.Adap] = PQadapt (Ehs, Fmem.Adap, Version, 'FFT');

% паттерны модуляции
[M, ERavg, Fmem.Env] = PQmodPatt (Es, Fmem.Env);

% громкость
MOVI.Loud.NRef  = PQloud (Ehs(1,:), Version, 'FFT');
MOVI.Loud.NTest = PQloud (Ehs(2,:), Version, 'FFT');

% разница модуляций
MOVI.MDiff = PQmovModDiffB (M, ERavg);

% громкость шума
MOVI.NLoud.NL = PQmovNLoudB (M, EP);

% полоса пропускания
MOVI.BW = PQmovBW (X2);

% отношений шума к маскированию
MOVI.NMR = PQmovNMRB (EbN, Ehs(1,:));

% вероятность обнаружения
MOVI.PD = PQmovPD (Ehs);

% ошибка гармонической структуры
MOVI.EHS.EHS = PQmovEHS (xR, xT, X2);

%--------------------
function [EbN, Es] = PQ_excitCB (X2)

persistent W2 EIN

NF = 2048;
Version = 'Basic';
if (isempty (W2))
  Fs = 48000;
  f = linspace (0, Fs/2, NF/2+1);
  W2 = PQWOME (f);
  [Nc, fc] = PQCB (Version);
  EIN = PQIntNoise (fc);
end
```

```
% выделение памяти
XwN2 = zeros (1, NF/2+1);

% фильтрация на основе модели внешнего и среднего уха
Xw2(1,:) = W2 .* X2(1,1:NF/2+1);
Xw2(2,:) = W2 .* X2(2,1:NF/2+1);
for (k = 0:NF/2)
   XwN2(k+1) = (Xw2(1,k+1) - 2 * sqrt (Xw2(1,k+1) * Xw2(2,k+1)) ...
         + Xw2(2,k+1));
end

Eb(1,:) = PQgroupCB (Xw2(1,:), Version);
Eb(2,:) = PQgroupCB (Xw2(2,:), Version);
EbN     = PQgroupCB (XwN2, Version);

E(1,:) = Eb(1,:) + EIN;
E(2,:) = Eb(2,:) + EIN;

Es(1,:) = PQspreadCB (E(1,:), Version);
Es(2,:) = PQspreadCB (E(2,:), Version);

%-------------------
function [Ehs, Ef] = PQ_timeSpread (Es, Ef)

persistent Nc a b

if (isempty (Nc))
   [Nc, fc] = PQCB ('Basic');
   Fs = 48000;
   NF = 2048;
   Nadv = NF / 2;
   Fss = Fs / Nadv;
   t100 = 0.030;
   tmin = 0.008;
   [a, b] = PQtConst (t100, tmin, fc, Fss);
end

Ehs = zeros (1, Nc);

for (m = 0:Nc-1)
   Ef(m+1) = a(m+1) * Ef(m+1) + b(m+1) * Es(m+1);
   Ehs(m+1) = max(Ef(m+1), Es(m+1));
end
```

```
function X2 = PQDFTFrame (x)

persistent hw

NF = 2048;

if (isempty (hw))
   Amax = 32768;
   fc = 1019.5;
   Fs = 48000;
   Lp = 92;
   GL = PQ_GL (NF, Amax, fc/Fs, Lp);
   hw = GL * PQHannWin (NF);
end

xw = hw .* x;
X = PQRFFT (xw, NF, 1);
X2 = PQRFFTMSq (X, NF);

%-----------------------------------------
function GL = PQ_GL (NF, Amax, fcN, Lp)

W = NF - 1;
gp = PQ_gp (fcN, NF, W);
GL = 10^(Lp / 20) / (gp * Amax/4 * W);
```

```
%----------
function gp = PQ_gp (fcN, NF, W)

df = 1 / NF;
k = floor (fcN / df);
dfN = min ((k+1) * df - fcN, fcN - k * df);

dfW = dfN * W;
gp = sin(pi * dfW) / (pi * dfW * (1 - dfW^2));
```

```
function Lim = PQdataBoundary (WAV, Nchan, StartS, Ns)

PQ_L = 5;
Amax = 32768;
NBUFF = 2048;
PQ_ATHR = 200 * (Amax / 32768);
Lim(1) = -1;
is = StartS;
EndS = StartS + Ns - 1;
while (is <= EndS)
   Nf = min (EndS - is + 1, NBUFF);
   x = PQgetData (WAV, is, Nf);
   for (k = 0:Nchan-1)
      Lim(1) = max (Lim(1), PQ_DataStart (x(k+1,:), Nf, PQ_L, PQ_ATHR));
   end
   if (Lim(1) >= 0)
      Lim(1) = Lim(1) + is;
      break
   end
   is = is + NBUFF - (PQ_L-1);
end
Lim(2) = -1;
is = StartS;
while (is <= EndS)
   Nf = min (EndS - is + 1, NBUFF);
   ie = is + Nf - 1;
   js = EndS - (ie - StartS + 1) + 1;
   x = PQgetData (WAV, js, Nf);
   for (k = 0:Nchan-1)
      Lim(2) = max (Lim(2), PQ_DataEnd (x(k+1,:), Nf, PQ_L, PQ_ATHR));
   end
   if (Lim(2) >= 0)
      Lim(2) = Lim(2) + js;
      break
   end
   is = is + NBUFF - (PQ_L-1);
end
if (~ ((Lim(1) >= 0 & Lim(2) >= 0) | (Lim(1) < 0 & Lim(2) < 0)))
   error ('>>> PQdataBoundary: limits have difference signs');
end
if (~(Lim(1) <= Lim(2)))
   error ('>>> PQdataBoundary: Lim(1) > Lim(2)');
end

if (Lim(1) < 0)
   Lim(1) = 0;
   Lim(2) = 0;
end

%----------
function ib = PQ_DataStart (x, N, L, Thr)

ib = -1;
s = 0;
```

```
M = min (N, L);
for (i = 0:M-1)
   s = s + abs (x(i+1));
end
if (s > Thr)
   ib = 0;
   return
end

for (i = 1:N-L)
   s = s + (abs (x(i+L-1+1)) - abs (x(i-1+1)));
   if (s > Thr)
      ib = i;
      return
   end
end

%---------
function ie = PQ_DataEnd (x, N, L, Thr)

ie = -1;
s = 0;
M = min (N, L);
for (i = N-M:N-1)
   s = s + abs (x(i+1));
end
if (s > Thr)
   ie = N-1;
   return
end

for (i = N-2:-1:L-1)
   s = s + (abs (x(i-L+1+1)) - abs (x(i+1+1)));       if (s > Thr)
      ie = i;
      return
   end
end
```

```
function x = PQgetData (WAV, i, N)

persistent Buff

iB = WAV.iB + 1;
if (N == 0)
   Buff(iB).N = 20 * 1024;     % Fixed size
   Buff(iB).x = PQ_ReadWAV (WAV, i, Buff(iB).N);
   Buff(iB).i = i;
end

if (N > Buff(iB).N)
   error ('>>> PQgetData: Request exceeds buffer size');
end
is = i - Buff(iB).i;
if (is < 0 | is + N - 1 > Buff(iB).N - 1)
   Buff(iB).x = PQ_ReadWAV (WAV, i, Buff(iB).N);
   Buff(iB).i = i;
end
Nchan = WAV.Nchan;
is = i - Buff(iB).i;
x = Buff(iB).x(1:Nchan,is+1:is+N-1+1);

%------
function x = PQ_ReadWAV (WAV, i, N)

Amax = 32768;
Nchan = WAV.Nchan;
```

```
x = zeros (Nchan, N);

Nz = 0;
if (i < 0)
   Nz = min (-i, N);
   i = i + Nz;
end

Ns = min (N - Nz, WAV.Nframe - i);
if (i >= 0 & Ns > 0)
   x(1:Nchan,Nz+1:Nz+Ns-1+1) = Amax * (wavread (WAV.Fname, [i+1 i+Ns-1+1]))';
end
```

```
function hw = PQHannWin (NF)

hw = zeros (1, NF);

for (n = 0:NF-1)
   hw(n+1) = 0.5 * (1 - cos(2 * pi * n / (NF-1)));
end
```

```
function Fmem = PQinitFMem (Nc, PCinit)

Fmem.TDS.Ef(1:2,1:Nc) = 0;
Fmem.Adap.P(1:2,1:Nc) = 0;
Fmem.Adap.Rn(1:Nc) = 0;
Fmem.Adap.Rd(1:Nc) = 0;
Fmem.Adap.PC(1:2,1:Nc) = PCinit;
Fmem.Env.Ese(1:2,1:Nc) = 0;
Fmem.Env.DE(1:2,1:Nc) = 0;
Fmem.Env.Eavg(1:2,1:Nc) = 0;
```

```
function EIN = PQIntNoise (f)

N = length (f);
for (m = 0:N-1)
   INdB = 1.456 * (f(m+1) / 1000)^(-0.8);
   EIN(m+1) = 10^(INdB / 10);
End
```

```
function X = PQRFFT (x, N, ifn)

if (ifn > 0)
   X = fft (x, N);
   XR = real(X(0+1:N/2+1));
   XI = imag(X(1+1:N/2-1+1));
   X = [XR XI];
else
   xR = [x(0+1:N/2+1)];
   xI = [0 x(N/2+1+1:N-1+1) 0];
   x = complex ([xR xR(N/2-1+1:-1:1+1)], [xI -xI(N/2-1+1:-1:1+1)]);
   X = real (ifft (x, N));
end
```

```
function X2 = PQRFFTMSq (X, N)

X2 = zeros (1, N/2+1);

X2(0+1) = X(0+1)^2;
for (k = 1:N/2-1)
   X2(k+1) = X(k+1)^2 + X(N/2+k+1)^2;
end
X2(N/2+1) = X(N/2+1)^2;
```

```
function [a, b] = PQtConst (t100, tmin, f , Fs)
N = length (f);
for (m = 0:N-1)
   t = tmin + (100 / f(m+1)) * (t100 - tmin);
   a(m+1) = exp (-1 / (Fs * t));
   b(m+1) = (1 - a(m+1));
end
```

```
function W2 = PQWOME (f)
N = length (f);
for (k = 0:N-1)
   fkHz = f(k+1) / 1000;
   AdB = -2.184 * fkHz^(-0.8) + 6.5 * exp(-0.6 * (fkHz - 3.3)^2) ...
        - 0.001 * fkHz^(3.6);
   W2(k+1) = 10^(AdB / 10);
end
```

```
function WAV = PQwavFilePar (File)
persistent iB
if (isempty (iB))
   iB = 0;
else
   iB = mod (iB + 1, 2);
end
[size WAV.Fs Nbit] = wavread (File, 'size');
WAV.Fname = File;
WAV.Nframe = size(1);
WAV.Nchan = size(2);
WAV.iB = iB;
PQgetData (WAV, 0, 0);
```

```
function MOV = PQavgMOVB (MOVC, Nchan, Nwup)

Fs = 48000;
NF = 2048;
Nadv = NF / 2;
Fss = Fs / Nadv;
tdel = 0.5;
tex = 0.050;

 [MOV(0+1), MOV(1+1)] = PQ_avgBW (MOVC.BW);

% Total NMRB, RelDistFramesB
[MOV(2+1), MOV(10+1)] = PQ_avgNMRB (MOVC.NMR);

% WinModDiff1B, AvgModDiff1B, AvgModDiff2B
N500ms = ceil (tdel * Fss);
Ndel = max (0, N500ms - Nwup);
[MOV(3+1), MOV(6+1), MOV(7+1)] = PQ_avgModDiffB (Ndel, MOVC.MDiff);

% RmsNoiseLoudB
N50ms = ceil (tex * Fss);
Nloud = PQloudTest (MOVC.Loud);
Ndel = max (Nloud + N50ms, Ndel);
MOV(8+1) = PQ_avgNLoudB (Ndel, MOVC.NLoud);

% ADBB, MFPDB
[MOV(4+1), MOV(9+1)] = PQ_avgPD (MOVC.PD);

% EHSB
MOV(5+1) = PQ_avgEHS (MOVC.EHS);

%-----------------------------------------
function EHSB = PQ_avgEHS (EHS)
```

```
[Nchan, Np] = size (EHS.EHS);

s = 0;
for (j = 0:Nchan-1)
   s = s + PQ_LinPosAvg (EHS.EHS(j+1,:));
end
EHSB = 1000 * s / Nchan;

%--------------------------------------------
function [ADBB, MFPDB] = PQ_avgPD (PD)

global PQopt

c0 = 0.9;
if (isempty (PQopt))
   c1 = 1;
else
   c1 = PQopt.PDfactor;
end

N = length (PD.Pc);
Phc = 0;
Pcmax = 0;
Qsum = 0;
nd = 0;
for (i = 0:N-1)
   Phc = c0 * Phc + (1 - c0) * PD.Pc(i+1);
   Pcmax = max (Pcmax * c1, Phc);

   if (PD.Pc(i+1) > 0.5)
      nd = nd + 1;
      Qsum = Qsum + PD.Qc(i+1);
   end
end

if (nd == 0)
   ADBB = 0;
elseif (Qsum > 0)
   ADBB = log10 (Qsum / nd);
else
   ADBB = -0.5;
end

MFPDB = Pcmax;

%--------------------------------------------
function [TotalNMRB, RelDistFramesB] = PQ_avgNMRB (NMR)

[Nchan, Np] = size (NMR.NMRavg);
Thr = 10^(1.5 / 10);

s = 0;
for (j = 0:Nchan-1)
   s = s + 10 * log10 (PQ_LinAvg (NMR.NMRavg(j+1,:)));
end
TotalNMRB = s / Nchan;

s = 0;
for (j = 0:Nchan-1)
   s = s + PQ_FractThr (Thr, NMR.NMRmax(j+1,:));
end
RelDistFramesB = s / Nchan;

%--------------------------------------------
function [BandwidthRefB, BandwidthTestB] = PQ_avgBW (BW)

[Nchan, Np] = size (BW.BWRef);

sR = 0;
sT = 0;
```

```
for (j = 0:Nchan-1)
   sR = sR + PQ_LinPosAvg (BW.BWRef(j+1,:));
   sT = sT + PQ_LinPosAvg (BW.BWTest(j+1,:));
end
BandwidthRefB  = sR / Nchan;
BandwidthTestB = sT / Nchan;

%-----------------------------------------
function [WinModDiff1B, AvgModDiff1B, AvgModDiff2B] = PQ_avgModDiffB (Ndel, MDiff)

NF = 2048;
Nadv = NF / 2;
Fs = 48000;

Fss = Fs / Nadv;
tavg = 0.1;

[Nchan, Np] = size (MDiff.Mt1B);
L = floor (tavg * Fss);
s = 0;
for (j = 0:Nchan-1)
   s = s + PQ_WinAvg (L, MDiff.Mt1B(j+1,Ndel+1:Np-1+1));
end
WinModDiff1B = s / Nchan;
s = 0;
for (j = 0:Nchan-1)
   s = s + PQ_WtAvg (MDiff.Mt1B(j+1,Ndel+1:Np-1+1), MDiff.Wt(j+1,Ndel+1:Np-1+1));
end
AvgModDiff1B = s / Nchan;
s = 0;
for (j = 0:Nchan-1)
   s = s + PQ_WtAvg (MDiff.Mt2B(j+1,Ndel+1:Np-1+1), MDiff.Wt(j+1,Ndel+1:Np-1+1));
end
AvgModDiff2B = s / Nchan;

%-----------------------------------------
function RmsNoiseLoudB = PQ_avgNLoudB (Ndel, NLoud)

[Nchan, Np] = size (NLoud.NL);

s = 0;
for (j = 0:Nchan-1)
   s = s + PQ_RMSAvg (NLoud.NL(j+1,Ndel+1:Np-1+1));
end
RmsNoiseLoudB = s / Nchan;

%---------------------------------------
function s = PQ_LinPosAvg (x)

N = length(x);

Nv = 0;
s = 0;
for (i = 0:N-1)
   if (x(i+1) >= 0)
      s = s + x(i+1);
      Nv = Nv + 1;
   end
end

if (Nv > 0)
   s = s / Nv;
end

%---------
function Fd = PQ_FractThr (Thr, x)

N = length (x);
```

```
Nv = 0;
for (i = 0:N-1)
   if (x(i+1) > Thr)
      Nv = Nv + 1;
   end
end

if (N > 0)
   Fd = Nv / N;
else
   Fd = 0;
end

%-----------
function s = PQ_WinAvg (L, x)

N = length (x);

s = 0;
for (i = L-1:N-1)
   t = 0;
   for (m = 0:L-1)
      t = t + sqrt (x(i-m+1));
   end
   s = s + (t / L)^4;
end

if (N >= L)
   s = sqrt (s / (N - L + 1));
end

%----------
function s = PQ_WtAvg (x, W)

N = length (x);

s = 0;
sW = 0;
for (i = 0:N-1)
   s = s + W(i+1) * x(i+1);
   sW = sW + W(i+1);
end

if (N > 0)
   s = s / sW;
end

%----------
function LinAvg = PQ_LinAvg (x)

N = length (x);
s = 0;
for (i = 0:N-1)
   s = s + x(i+1);
end

LinAvg = s / N;

%----------
function RMSAvg = PQ_RMSAvg (x)

N = length (x);
s = 0;
for (i = 0:N-1)
   s = s + x(i+1)^2;
end

if (N > 0)
   RMSAvg = sqrt(s / N);
else
   RMSAvg = 0;
end
```

```
function PQframeMOV (i, MOVI)

global MOVC

[Nchan,Nc] = size (MOVC.MDiff.Mt1B);

for (j = 1:Nchan)

   % Modulation differences
   MOVC.MDiff.Mt1B(j,i+1) = MOVI(j).MDiff.Mt1B;
   MOVC.MDiff.Mt2B(j,i+1) = MOVI(j).MDiff.Mt2B;
   MOVC.MDiff.Wt(j,i+1)   = MOVI(j).MDiff.Wt;

   % Noise loudness
   MOVC.NLoud.NL(j,i+1) = MOVI(j).NLoud.NL;

   % Total loudness
   MOVC.Loud.NRef(j,i+1)  = MOVI(j).Loud.NRef;
   MOVC.Loud.NTest(j,i+1) = MOVI(j).Loud.NTest;

   % Bandwidth
   MOVC.BW.BWRef(j,i+1) = MOVI(j).BW.BWRef;
   MOVC.BW.BWTest(j,i+1) = MOVI(j).BW.BWTest;

   % Noise-to-mask ratio
   MOVC.NMR.NMRavg(j,i+1) = MOVI(j).NMR.NMRavg;
   MOVC.NMR.NMRmax(j,i+1) = MOVI(j).NMR.NMRmax;

   % Error harmonic structure
   MOVC.EHS.EHS(j,i+1) = MOVI(j).EHS.EHS;
end

% Probability of detection (collapse frequency bands)
[MOVC.PD.Pc(i+1), MOVC.PD.Qc(i+1)] = PQ_ChanPD (MOVI);

%-----------------------------------------
function [Pc, Qc] = PQ_ChanPD (MOVI)

Nc = length (MOVI(1).PD.p);
Nchan = length (MOVI);

Pr = 1;
Qc = 0;
if (Nchan > 1)
   for (m = 0:Nc-1)
      pbin = max (MOVI(1).PD.p(m+1), MOVI(2).PD.p(m+1));
      qbin = max (MOVI(1).PD.q(m+1), MOVI(2).PD.q(m+1));
      Pr = Pr * (1 - pbin);
      Qc = Qc + qbin;
   end
else
   for (m = 0:Nc-1)
      Pr = Pr * (1 - MOVI.PD.p(m+1));
      Qc = Qc + MOVI.PD.q(m+1);
   end
end

Pc = 1 - Pr;
```

```
function Ndel = PQloudTest (Loud)

 [Nchan, Np] = size (Loud.NRef);

Thr = 0.1;
Ndel = Np;
for (j = 0:Nchan-1)
   Ndel = min (Ndel, PQ_LThresh (Thr, Loud.NRef(j+1,:), Loud.NTest(j+1,:)));
end
```

```
%----------
function it = PQ_LThresh (Thr, NRef, NTest)

Np = length (NRef);

it = Np;
for (i = 0:Np-1)
   if (NRef(i+1) > Thr & NTest(i+1) > Thr)
      it = i;
      break;
   end
end
```

```
function BW = PQmovBW (X2)

persistent kx kl FR FT N

if (isempty (kx))
   NF = 2048;
   Fs = 48000;
   fx = 21586;
   kx = round (fx / Fs * NF);    % 921
   fl = 8109;
   kl = round (fl / Fs * NF);    % 346
   FRdB = 10;
   FR = 10^(FRdB / 10);
   FTdB = 5;
   FT = 10^(FTdB / 10);
   N = NF / 2;     % Limit from pseudo-code
end

Xth = X2(2,kx+1);
for (k = kx+1:N-1)
   Xth = max (Xth, X2(2,k+1));
end

BW.BWRef = -1;
XthR = FR * Xth;
for (k = kx-1:-1:kl+1)
   if (X2(1,k+1) >= XthR)
      BW.BWRef = k + 1;
      break;
   end
end

BW.BWTest = -1;
XthT = FT * Xth;
for (k = BW.BWRef-1:-1:0)
   if (X2(2,k+1) >= XthT)
      BW.BWTest = k + 1;
      break;
   end
end
```

```
function MDiff = PQmovModDiffB (M, ERavg)

persistent Nc Ete

if (isempty (Nc))
   e = 0.3;
   [Nc, fc] = PQCB ('Basic');
   Et = PQIntNoise (fc);
   for (m = 0:Nc-1)
      Ete(m+1) = Et(m+1)^e;
   end
end
```

```
negWt2B = 0.1;
offset1B = 1.0;
offset2B = 0.01;
levWt = 100;

s1B = 0;
s2B = 0;
Wt = 0;
for (m = 0:Nc-1)
   if (M(1,m+1) > M(2,m+1))
      num1B = M(1,m+1) - M(2,m+1);
      num2B = negWt2B * num1B;
   else
      num1B = M(2,m+1) - M(1,m+1);
      num2B = num1B;
   end
   MD1B = num1B / (offset1B + M(1,m+1));
   MD2B = num2B / (offset2B + M(1,m+1));
   s1B = s1B + MD1B;
   s2B = s2B + MD2B;
   Wt = Wt + ERavg(m+1) / (ERavg(m+1) + levWt * Ete(m+1));
end

MDiff.Mt1B = (100 / Nc) * s1B;
MDiff.Mt2B = (100 / Nc) * s2B;
MDiff.Wt = Wt;
```

```
function NL = PQmovNLoudB (M, EP)

persistent Nc Et

if (isempty (Nc))
   [Nc, fc] = PQCB ('Basic');
   Et = PQIntNoise (fc);
end

alpha = 1.5;
TF0 = 0.15;
S0 = 0.5;
NLmin = 0;
e = 0.23;

s = 0;
for (m = 0:Nc-1)
   sref  = TF0 * M(1,m+1) + S0;
   stest = TF0 * M(2,m+1) + S0;
   beta = exp (-alpha * (EP(2,m+1) - EP(1,m+1)) / EP(1,m+1));
   a = max (stest * EP(2,m+1) - sref * EP(1,m+1), 0);
   b = Et(m+1) + sref * EP(1,m+1) * beta;
   s = s + (Et(m+1) / stest)^e * ((1 + a / b)^e - 1);
end

NL = (24 / Nc) * s;
if (NL < NLmin)
   NL = 0;
end
```

```
function NMR = PQmovNMRB (EbN, Ehs)

persistent Nc gm

if (isempty (Nc))
   [Nc, fc, fl, fu, dz] = PQCB ('Basic');
   gm = PQ_MaskOffset (dz, Nc);
end
```

```
NMR.NMRmax = 0;
s = 0;
for (m = 0:Nc-1)
   NMRm = EbN(m+1) / (gm(m+1) * Ehs(m+1));
   s = s + NMRm;
   if (NMRm > NMR.NMRmax)
      NMR.NMRmax = NMRm;
   end
end
NMR.NMRavg = s / Nc;
%--------------------------------------------
function gm = PQ_MaskOffset (dz, Nc)

for (m = 0:Nc-1)
   if (m <= 12 / dz)
      mdB = 3;
   else
      mdB = 0.25 * m * dz;
   end
   gm(m+1) = 10^(-mdB / 10);
end
```

```
function PD = PQmovPD (Ehs)

Nc = length (Ehs);

PD.p = zeros (1, Nc);
PD.q = zeros (1, Nc);

persistent c g d1 d2 bP bM

if (isempty (c))
   c = [-0.198719 0.0550197 -0.00102438 5.05622e-6 9.01033e-11];
   d1 = 5.95072;
   d2 = 6.39468;
   g = 1.71332;
   bP = 4;
   bM = 6;
end

for (m = 0:Nc-1)
   EdBR = 10 * log10 (Ehs(1,m+1));
   EdBT = 10 * log10 (Ehs(2,m+1));
   edB = EdBR - EdBT;
   if (edB > 0)
      L = 0.3 * EdBR + 0.7 * EdBT;
      b = bP;
   else
      L = EdBT;
      b = bM;
   end
   if (L > 0)
      s = d1 * (d2 / L)^g ...
         + c(1) + L * (c(2) + L * (c(3) + L * (c(4) + L * c(5))));
   else
      s = 1e30;
   end
   PD.p(m+1) = 1 - 0.5^((edB / s)^b);
   PD.q(m+1) = abs (fix(edB)) / s;
end
```

```
function PQprtMOV (MOV, ODG)

N = length (MOV);
PQ_NMOV_B = 11;
PQ_NMOV_A = 5;
```

```
fprintf ('Model Output Variables:\n');
if (N == PQ_NMOV_B)
   fprintf ('  BandwidthRefB: %g\n', MOV(1));
   fprintf ('  BandwidthTestB: %g\n', MOV(2));
   fprintf ('     Total NMRB: %g\n', MOV(3));
   fprintf ('   WinModDiff1B: %g\n', MOV(4));
   fprintf ('           ADBB: %g\n', MOV(5));
   fprintf ('           EHSB: %g\n', MOV(6));
   fprintf ('   AvgModDiff1B: %g\n', MOV(7));
   fprintf ('   AvgModDiff2B: %g\n', MOV(8));
   fprintf ('   RmsNoiseLoudB: %g\n', MOV(9));
   fprintf ('          MFPDB: %g\n', MOV(10));
   fprintf ('  RelDistFramesB: %g\n', MOV(11));
elseif (N == NMOV_A)
   fprintf ('       RmsModDiffA: %g\n', MOV(1));
   fprintf ('  RmsNoiseLoudAsymA: %g\n', MOV(2));
   fprintf ('    Segmental NMRB: %g\n', MOV(3));
   fprintf ('              EHSB: %g\n', MOV(4));
   fprintf ('       AvgLinDistA: %g\n', MOV(5));
else
   error ('Invalid number of MOVs');
end

fprintf ('Objective Difference Grade: %.3f\n', ODG);
return;
```

```
function PQprtMOVCi (Nchan, i, MOVC)

fprintf ('Frame: %d\n', i);

if (Nchan == 1)
   fprintf ('  Ntot   : %g %g\n', ...
      MOVC.Loud.NRef(1,i+1), MOVC.Loud.NTest(1,i+1));
   fprintf ('  ModDiff: %g %g %g\n', ...
      MOVC.MDiff.Mt1B(1,i+1), MOVC.MDiff.Mt2B(1,i+1), MOVC.MDiff.Wt(1,i+1));
   fprintf ('  NL     : %g\n', MOVC.NLoud.NL(1,i+1));
   fprintf ('  BW     : %g %g\n', ...
      MOVC.BW.BWRef(1,i+1), MOVC.BW.BWTest(1,i+1));
   fprintf ('  NMR    : %g %g\n', ...
      MOVC.NMR.NMRavg(1,i+1), MOVC.NMR.NMRmax(1,i+1));
   fprintf ('  PD     : %g %g\n', MOVC.PD.Pc(i+1), MOVC.PD.Qc(i+1));
   fprintf ('  EHS    : %g\n', 1000 * MOVC.EHS.EHS(1,i+1));
else
   fprintf ('  Ntot   : %g %g // %g %g\n', ...
      MOVC.Loud.NRef(1,i+1), MOVC.Loud.NTest(1,i+1), ...
      MOVC.Loud.NRef(2,i+1), MOVC.Loud.NTest(2,i+1));
   fprintf ('  ModDiff: %g %g %g // %g %g %g\n', ...
      MOVC.MDiff.Mt1B(1,i+1), MOVC.MDiff.Mt2B(1,i+1), MOVC.MDiff.Wt(1,i+1), ...
      MOVC.MDiff.Mt1B(2,i+1), MOVC.MDiff.Mt2B(2,i+1), MOVC.MDiff.Wt(2,i+1));
   fprintf ('  NL     : %g // %g\n', ...
      MOVC.NLoud.NL(1,i+1), ...
      MOVC.NLoud.NL(2,i+1));
   fprintf ('  BW     : %g %g // %g %g\n', ...
      MOVC.BW.BWRef(1,i+1), MOVC.BW.BWTest(1,i+1), ...
      MOVC.BW.BWRef(2,i+1), MOVC.BW.BWTest(2,i+1));
   fprintf ('  NMR    : %g %g // %g %g\n', ...
      MOVC.NMR.NMRavg(1,i+1), MOVC.NMR.NMRmax(1,i+1), ...
      MOVC.NMR.NMRavg(2,i+1), MOVC.NMR.NMRmax(2,i+1));
   fprintf ('  PD     : %g %g\n', MOVC.PD.Pc(i+1), MOVC.PD.Qc(i+1));
   fprintf ('  EHS    : %g // %g\n', ...
      1000 * MOVC.EHS.EHS(1,i+1), ...
      1000 * MOVC.EHS.EHS(2,i+1));
end
```

```
function EHS = PQmovEHS (xR, xT, X2)

persistent NF Nadv NL M Hw

if (isempty (NL))
   NF = 2048;
   Nadv = NF / 2;
   Fs = 48000;
   Fmax = 9000;
   NL = 2^(PQ_log2(NF * Fmax / Fs));
   M = NL;
   Hw = (1 / M) * sqrt(8 / 3) * PQHannWin (M);
end

EnThr = 8000;
kmax = NL + M - 1;

EnRef  = xR(Nadv+1:NF-1+1) * xR(Nadv+1:NF-1+1)';
EnTest = xT(Nadv+1:NF-1+1) * xT(Nadv+1:NF-1+1)';
if (EnRef < EnThr & EnTest < EnThr)
   EHS = -1;
   return;
end

D = zeros (1, kmax);
for (k = 0:kmax-1)
   D(k+1) = log (X2(2,k+1) / X2(1,k+1));
end

C = PQ_Corr (D, NL, M);

Cn = PQ_NCorr (C, D, NL, M);
Cnm = (1 / NL) * sum (Cn(1:NL));
Cw = Hw .* (Cn - Cnm);

% DFT
cp = PQRFFT (Cw, NL, 1);
c2 = PQRFFTMSq (cp, NL);
EHS = PQ_FindPeak (c2, NL/2+1);

%---------------------------------------
function log2 = PQ_log2 (a)

log2 = 0;
m = 1;
while (m < a)
   log2 = log2 + 1;
   m = 2 * m;
end
log2 = log2 - 1;

%----------
function C = PQ_Corr (D, NL, M)

NFFT = 2 * NL;
D0 = [D(1:M) zeros(1,NFFT-M)];
D1 = [D(1:M+NL-1) zeros(1,NFFT-(M+NL-1))];

d0 = PQRFFT (D0, NFFT, 1);
d1 = PQRFFT (D1, NFFT, 1);

dx(0+1) = d0(0+1) * d1(0+1);
for (n = 1:NFFT/2-1)
   m = NFFT/2 + n;
   dx(n+1) = d0(n+1) * d1(n+1) + d0(m+1) * d1(m+1);
   dx(m+1) = d0(n+1) * d1(m+1) - d0(m+1) * d1(n+1);
end
dx(NFFT/2+1) = d0(NFFT/2+1) * d1(NFFT/2+1);
```

```
% Inverse DFT
Cx = PQRFFT (dx, NFFT, -1);
C = Cx(1:NL);

%----------
function Cn = PQ_NCorr (C, D, NL, M)

Cn = zeros (1, NL);

s0 = C(0+1);
sj = s0;
Cn(0+1) = 1;
for (i = 1:NL-1)
   sj = sj + (D(i+M-1+1)^2 - D(i-1+1)^2);
   d = s0 * sj;
   if (d <= 0)
      Cn(i+1) = 1;
   else
      Cn(i+1) = C(i+1) / sqrt (d);
   end
end

%----------
function EHS = PQ_FindPeak (c2, N)

cprev = c2(0+1);
cmax = 0;
for (n = 1:N-1)
   if (c2(n+1) > cprev)    % Rising from a valley
      if (c2(n+1) > cmax)
         cmax = c2(n+1);
      end
   end
end
EHS = cmax;
```

```
function [EP, Fmem] = PQadapt (Ehs, Fmem, Ver, Mod)

persistent a b Nc M1 M2 Version Model

if (~strcmp (Ver, Version) | ~strcmp (Mod, Model))
   Version = Ver;
   Model = Mod;
   if (strcmp (Model, 'FFT'))
      [Nc, fc] = PQCB (Version);
      NF = 2048;
      Nadv = NF / 2;
   else
      [Nc, fc] = PQFB;
      Nadv = 192;
   end
   Version = Ver;
   Model = Mod;
   Fs = 48000;
   Fss = Fs / Nadv;
   t100 = 0.050;
   tmin = 0.008;
   [a b] = PQtConst (t100, tmin, fc, Fss);
   [M1, M2] = PQ_M1M2 (Version, Model);
end

EP = zeros (2, Nc);
R = zeros (2, Nc);
```

```
sn = 0;
sd = 0;
for (m = 0:Nc-1)
   Fmem.P(1,m+1) = a(m+1) * Fmem.P(1,m+1) + b(m+1) * Ehs(1,m+1);
   Fmem.P(2,m+1) = a(m+1) * Fmem.P(2,m+1) + b(m+1) * Ehs(2,m+1);
   sn = sn + sqrt (Fmem.P(2,m+1) * Fmem.P(1,m+1));
   sd = sd + Fmem.P(2,m+1);
end

CL = (sn / sd)^2;

for (m = 0:Nc-1)

   if (CL > 1)
      EP(1,m+1) = Ehs(1,m+1) / CL;
      EP(2,m+1) = Ehs(2,m+1);
   else
      EP(1,m+1) = Ehs(1,m+1);
      EP(2,m+1) = Ehs(2,m+1) * CL;
   end

   Fmem.Rn(m+1) = a(m+1) * Fmem.Rn(m+1) + EP(2,m+1) * EP(1,m+1);
   Fmem.Rd(m+1) = a(m+1) * Fmem.Rd(m+1) + EP(1,m+1) * EP(1,m+1);
   if (Fmem.Rd(m+1) <= 0 | Fmem.Rn(m+1) <= 0)
      error ('>>> PQadap: Rd or Rn is zero');
   end
   if (Fmem.Rn(m+1) >= Fmem.Rd(m+1))
      R(1,m+1) = 1;
      R(2,m+1) = Fmem.Rd(m+1) / Fmem.Rn(m+1);
   else
      R(1,m+1) = Fmem.Rn(m+1) / Fmem.Rd(m+1);
      R(2,m+1) = 1;
   end
end

for (m = 0:Nc-1)
   iL = max (m - M1, 0);
   iU = min (m + M2, Nc-1);
   s1 = 0;
   s2 = 0;
   for (i = iL:iU)
      s1 = s1 + R(1,i+1);
      s2 = s2 + R(2,i+1);
   end
   Fmem.PC(1,m+1) = a(m+1) * Fmem.PC(1,m+1) + b(m+1) * s1 / (iU-iL+1);
   Fmem.PC(2,m+1) = a(m+1) * Fmem.PC(2,m+1) + b(m+1) * s2 / (iU-iL+1);

   EP(1,m+1) = EP(1,m+1) * Fmem.PC(1,m+1);
   EP(2,m+1) = EP(2,m+1) * Fmem.PC(2,m+1);
end

%---------------------------------------
function [M1, M2] = PQ_M1M2 (Version, Model)

if (strcmp (Version, 'Basic'))
   M1 = 3;
   M2 = 4;
elseif (strcmp (Version, 'Advanced'))
   if (strcmp (Model, 'FFT'))
      M1 = 1;
      M2 = 2;
   else
      M1 = 1;
      M2 = 1;
   end
end
```

```
function Ntot = PQloud (Ehs, Ver, Mod)

e = 0.23;

persistent Nc s Et Ets Version Model

if (~strcmp (Ver, Version) | ~strcmp (Mod, Model))
   Version = Ver;
   Model = Mod;
   if (strcmp (Model, 'FFT'))
      [Nc, fc] = PQCB ('Version);
      c = 1.07664;
   else
      [Nc, fc] = PQFB;
      c = 1.26539;
   end
   E0 = 1e4;
   Et = PQ_enThresh (fc);
   s = PQ_exIndex (fc);
   for (m = 0:Nc-1)
      Ets(m+1) = c * (Et(m+1) / (s(m+1) * E0))^e;
   end
end

sN = 0;
for (m = 0:Nc-1)
   Nm = Ets(m+1) * ((1 - s(m+1) + s(m+1) * Ehs(m+1) / Et(m+1))^e - 1);
   sN = sN + max(Nm, 0);
end
Ntot = (24 / Nc) * sN;

%====================
function s = PQ_exIndex (f)

N = length (f);
for (m = 0:N-1)
   sdB = -2 - 2.05 * atan(f(m+1) / 4000) - 0.75 * atan((f(m+1) / 1600)^2);
   s(m+1) = 10^(sdB / 10);
end

%--------------------
function Et = PQ_enThresh (f)

N = length (f);
for (m = 0:N-1)
   EtdB = 3.64 * (f(m+1) / 1000)^(-0.8);
   Et(m+1) = 10^(EtdB / 10);
end
```

```
function [M, ERavg, Fmem] = PQmodPatt (Es, Fmem)

persistent Nc a b Fss

if (isempty (Nc))
   Fs = 48000;
   NF = 2048;
   Fss = Fs / (NF/2);
   [Nc, fc] = PQCB ('Basic');
   t100 = 0.050;
   t0 = 0.008;
   [a, b] = PQtConst (t100, t0, fc, Fss);
end

M = zeros (2, Nc);
```

```
e = 0.3;
for (i = 1:2)
   for (m = 0:Nc-1)
      Ee = Es(i,m+1)^e;
      Fmem.DE(i,m+1) = a(m+1) * Fmem.DE(i,m+1) ...
                  + b(m+1) * Fss * abs (Ee - Fmem.Ese(i,m+1));
      Fmem.Eavg(i,m+1) = a(m+1) * Fmem.Eavg(i,m+1) + b(m+1) * Ee;
      Fmem.Ese(i,m+1) = Ee;

      M(i,m+1) = Fmem.DE(i,m+1) / (1 + Fmem.Eavg(i,m+1)/0.3);
   end
end

ERavg = Fmem.Eavg(1,:);
```

Б.2 Листинг программы расчета метрики PEAQ на языке С

Файл: common.h

```
#define DEBUG
#define HANN 2048
#define BARK 109

#define DOUBLE

#if defined(DOUBLE)
   #define module(x) fabs((double) x)
   #define p(x,y) pow((double)x, (double)y)
#elif defined(LDOUBLE)
   #define module(x) fabsl((long double) x)
   #define p(x,y) powl((long double)x, (long double)y)
#endif

/********************** detprob **********************/
#define C1 1.0
/************************ end ************************/

/********************* harmstruct *********************/
#define AVGHANN
#define SKIPFRAME
#define GETMAX

#define Fup 18000.0
#define Flow 80.0
#define PATCH 1
/****************************** end ************************/

/***************************** peaqb **********************/
#define LOGVARIABLE

#ifdef LOGVARIABLE
#define LOGALLFRAMES
#endif
/****************************** end ************************/

struct processing {
        double fftref[HANN/2];
        double ffttest[HANN/2];
        double ffteref[HANN/2];
        double fftetest[HANN/2];
        double fnoise[HANN/2];
        double pptest[BARK];
        double ppref[BARK];
        double ppnoise[BARK];
        double E2test[BARK];
        double E2ref[BARK];
        double Etest[BARK];
        double Eref[BARK];
```

```
        double Mref[BARK];
        double Modtest[BARK];
        double Modref[BARK];
};
```

Файл: peaqb.h

```
#define LOGRESULT "analized"

#ifdef DEBUG
#define LOGFILE "debugged.txt"
#endif

#define OPT_REF      0x01
#define OPT_TEST       0x02

#define THRESHOLDDELAY 0.050
#define AVERAGINGDEALAY 0.5
```

```
#define B(f) 7 * asinh((double)f /650)
#define Bl(z) 650 * sinh((double)z /7)

/* Function prototypes */
void fatalerr(char *,...);
void usage(char *);
void logvariable(const char *, double *, int);
void ProcessFrame(signed int *, signed int *, int, signed int *,
            signed int *, int, int, int, int);
/* Prototypes end */
```

Файл: peaqb.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include <getopt.h>
#include <assert.h>
#include <math.h>
#include <fftw.h>
#include <common.h>
#include <wavedump.h>
#include <getframe.h>
#include <bandwidth.h>
#include <levpatadapt.h>
#include <moddiff.h>
#include <modulation.h>
#include <loudness.h>
#include <neural.h>
#include <nmr.h>
#include <detprob.h>
#include <energyth.h>
#include <harmstruct.h>
#include <boundary.h>
#include <critbandgroup.h>
#include <earmodelfft.h>
#include <noiseloudness.h>
#include <reldistframes.h>
#include <spreading.h>
#include <timespreading.h>
#include <threshold.h>
#include <peaqb.h>

extern int errno;
char *fileref, *filetest;
```

```
double hannwindow[HANN];
double Etesttmpch1[BARK], Etesttmpch2[BARK], Ereftmpch1[BARK],
    Ereftmpch2[BARK], Cffttmpch1[HANN/2], Cffttmpch2[HANN/2];
int delaytime1, delaytime2;
int count = 0;
int harmsamples = 1;
fftw_plan plan, plan2;
```

```
/* Bark Tables */
double *fL, *fC, *fU;
int bark;

struct levpatadaptin levinch1, levinch2;
struct modulationin modintestch1, modintestch2, modinrefch1, modinrefch2;
struct moddiffin moddiffinch1, moddiffinch2;
struct bandwidthout bandwidthch1, bandwidthch2;
struct outframes processed;

int
main(int argc, char *argv[])
{

  signed int ch1ref[HANN];
  signed int ch2ref[HANN];
  signed int ch1test[HANN];
  signed int ch2test[HANN];

  int opt_line = 0;
  int rateref, numchref, bitsampleref, lpref;
  int ratetest, numchtest, bitsampletest, lptest;
  int boundflag, totalframes = 0;
  FILE *fpref, *fptest;

  struct boundaryflag boundbe = {0, 0};
  struct out oveRet;

  /* Parse command line */
  if (argc < 3)
      usage(argv[0]);

  {
  int c = 0;

  while ((c = getopt(argc, argv, "r:t:h")) != EOF)
      switch (c) {
          case 'h':
              usage(argv[0]);
              break;
          case 'r':
              opt_line |= OPT_REF;
              fileref = optarg;
              break;
          case 't':
              opt_line |= OPT_TEST;
              filetest = optarg;
              break;
      }
  }

  /* Input control */
  if (!(opt_line & OPT_REF) || !*fileref)
      fatalerr ("err: -r/--reference <arg> required");

  if (!(opt_line & OPT_TEST) || !*filetest)
      fatalerr ("err: -t/--test <arg> required");
```

```
lpref = LevelPression(fileref);
lptest = LevelPression(filetest);

/* Init routines */
// make Hann Window  (2.1.3)
{
int k;

for(k=0;k<HANN;k++)
    hannwindow[k] = 0.5*sqrt((double)8/3)*
                (1 - cos((double)2*M_PI*k/(HANN -1)));
}
// make Bark tables  (2.1.5)
{
int k;
double zL, zU;
double *zl, *zc, *zu;
zL = B(Flow);
zU = B(Fup);

bark = ceil((zU - zL) / dz);
fL = (double *)malloc(bark * sizeof(double));
fC = (double *)malloc(bark * sizeof(double));
fU = (double *)malloc(bark * sizeof(double));
zl = (double *)malloc(bark * sizeof(double));
zc = (double *)malloc(bark * sizeof(double));
zu = (double *)malloc(bark * sizeof(double));
assert(fL != NULL && fC != NULL && fU != NULL && zl != NULL
    && zc != NULL && zu != NULL);

for(k=0;k<bark;k++) {
    zl[k] = zL + k*dz;
    zu[k] = zL + (k+1)*dz;
    zc[k] = 0.5 * (zl[k] + zu[k]);
}

zu[bark-1] = zU;
zc[bark -1] = 0.5 * (zl[bark-1] + zu[bark-1]);

for(k=0;k<bark;k++) {
    fL[k] = Bl(zl[k]);
    fU[k] = Bl(zu[k]);
    fC[k] = Bl(zc[k]);
}

free(zl);
free(zu);
free(zc);
}

// Initialize temp var
memset(&levinch1, 0x00, sizeof(struct levpatadaptin));
memset(&levinch2, 0x00, sizeof(struct levpatadaptin));

memset(&modintestch1, 0x00, sizeof(struct modulationin));
```

```
memset(&modintestch2, 0x00, sizeof(struct modulationin));
memset(&modinrefch1, 0x00, sizeof(struct modulationin));
memset(&modinrefch2, 0x00, sizeof(struct modulationin));

memset(Etesttmpch1, 0x00, BARK * sizeof(double));
memset(Etesttmpch2, 0x00, BARK * sizeof(double));
memset(Ereftmpch1, 0x00, BARK * sizeof(double));
memset(Ereftmpch2, 0x00, BARK * sizeof(double));
memset(Cfttmpch1, 0x00, (HANN/2) * sizeof(double));
memset(Cfttmpch2, 0x00, (HANN/2) * sizeof(double));
```

```
memset(&moddiffinch1, 0x00, sizeof(struct moddiffin));
memset(&moddiffinch2, 0x00, sizeof(struct moddiffin));

memset(&bandwidthch1, 0x00, sizeof(struct bandwidthout));

memset(&bandwidthch2, 0x00, sizeof(struct bandwidthout));

// ref file
if ((fpref = fopen(fileref,"r")) == NULL)
    fatalerr("err: %s", strerror(errno));
if ((rateref = SampleRate(fpref)) == -1)
    fatalerr("err: error in WaveHeader");
if ((numchref = NumOfChan(fpref)) == -1)
    fatalerr("err: error in WaveHeader");
if ((bitsampleref = BitForSample(fpref)) == -1)
    fatalerr("err: error in WaveHeader");
if(FindData(fpref) == -1)
    fatalerr("err: can't find Data Field");

// test file
if ((fptest = fopen(filetest,"r")) == NULL)
    fatalerr("err: %s", strerror(errno));
if ((ratetest = SampleRate(fptest)) == -1)
    fatalerr("err: error in WaveHeader");
if ((numchtest = NumOfChan(fptest)) == -1)
    fatalerr("err: error in WaveHeader");
if ((bitsampletest = BitForSample(fptest)) == -1)
    fatalerr("err: error in WaveHeader");
if(FindData(fptest) == -1)
    fatalerr("err: can't find Data Field");

fprintf(stdout,"\n PEAQb Algorithm. Author Giuseppe Gottardi 'oveRet'"
            " <gost.e-ecolog.ru                    );

fprintf(stdout,"\nRef File %s"
            "\n  - Sample Rate: %d"
            "\n  - Number Of Channel: %d"
            "\n  - Bits for Sample: %d"
              "\n  - Level Playback: %d\n\n", fileref, rateref,
              numchref, bitsampleref, lpref);

fprintf(stdout,"\nTest File %s"
            "\n  - Sample Rate: %d"
            "\n  - Number Of Channel: %d"
            "\n  - Bits for Sample: %d"
              "\n  - Level Playback: %d\n\n", filetest, ratetest,
              numchtest, bitsampletest, lptest);
```

```
// Processing
if(ratetest != rateref)
    fatalerr("err: Can't process Wave Files with different Sample Rate");
if(numchref != numchtest)
    fatalerr("err: Can't process Mono Wave with Stereo Wave");

// Find delaytime1 for Loudness Threshold
delaytime1 = ceilf((float)THRESHOLDDELAY*ratetest*2/HANN);
// Find delaytime2 for Delayed Averaging
delaytime2 = ceilf((float)AVERAGINGDEALAY*ratetest*2/HANN);

// make fft plan
plan = fftw_create_plan(HANN, FFTW_FORWARD, FFTW_MEASURE);

while(harmsamples < (Fup/ratetest)*(HANN/2.0)/2.0)
    harmsamples *= 2;
plan2 = fftw_create_plan(harmsamples, FFTW_FORWARD, FFTW_MEASURE);
```

```
if(numchref == 1) {
    if (fseek(fpref, (HANN/2)*bitsampleref/8, SEEK_CUR) == -1)
       fatalerr("err: %s", strerror(errno));
    if (fseek(fptest, (HANN/2)*bitsampletest/8, SEEK_CUR) == -1)
       fatalerr("err: %s", strerror(errno));

    #ifdef DATABOUND_BE
    #undef DATABOUND_ONE
    {
    int i = 0, flag = 0, f1, f2;
    long dataref, datatest, br1, br2;

    dataref = ftell(fpref);
    datatest = ftell(fptest);

    while(1) {
       br1 = ftell(fpref);
       br2 = ftell(fptest);
       f1 = GetMonoFrame(fpref, (signed int *)ch1ref,
                         bitsampleref/8, HANN);
       f2 = GetMonoFrame(fptest, (signed int *)ch1test,
                          bitsampletest/8, HANN);
       if(f1 && f2) {
       totalframes++;
            if(boundary(ch1ref, ch1test, NULL, NULL, HANN) && !flag) {
               boundbe.begin = totalframes;
               flag = 1;
            }
       }
       else {
          fseek(fptest, br1, SEEK_SET);
           fseek(fpref, br2, SEEK_SET);
           break;
       }
    }

    fseek(fptest, -(HANN/2)*bitsampletest/8, SEEK_CUR);
```

```
    fseek(fpref, -(HANN/2)*bitsampleref/8, SEEK_CUR);
    while(i<totalframes) {
       GetMonoFrame(fpref, (signed int *)ch1ref, bitsampleref/8, HANN);
       GetMonoFrame(fptest, (signed int *)ch1test, bitsampletest/8, HANN);
       fseek(fptest, -2*(HANN/2)*bitsampletest/8, SEEK_CUR);
       fseek(fpref, -2*(HANN/2)*bitsampleref/8, SEEK_CUR);
       i++;
       if(boundary(ch1ref, ch1test, NULL, NULL, HANN)) {
            boundbe.end = totalframes-i;
            break;
       }
    }

    fseek(fptest, datatest, SEEK_SET);
    fseek(fpref, dataref, SEEK_SET);
    }
    #endif

    while (GetMonoFrame(fpref, (signed int *)ch1ref,
                        bitsampleref/8, HANN)
    && GetMonoFrame(fptest, (signed int *)ch1test,
                    bitsampletest/8, HANN)) {

       count++;
       #ifdef DATABOUND_BE
       if(count >= boundbe.begin && count <= boundbe.end)
            boundflag = 1;
```

```
        else
            boundflag = 0;
        #else
        boundflag = boundary(ch1ref, ch1test, NULL, NULL, HANN);
          #ifdef DATABOUND_ONE
           {
           static int flag1 = 0, flag2 = 0;
           if(boundflag && !flag1)
                flag1 = 1;
            if(!boundflag && flag1)
                flag2 = 1;
            if(flag2)
                boundflag = 0;
            }
          #endif
        #endif

        ProcessFrame((signed int *)ch1ref,
                  (signed int *)NULL, lpref,
                     (signed int *)ch1test,
                     (signed int *)NULL,
                     lptest, rateref, boundflag, HANN);

        oveRet = neural(processed);

        fprintf(stdout,"\nframe: %d"
                        #ifdef DATABOUND_BE
                        "/%d"
```

```
                        "\ndata boundary: %d -> %d"
                        #endif
                        "\nBandwidthRefb: %g"
                        "\nBandwidthTestb: %g"
                        "\nTotalNMRb %g"
                        "\nWinModDiff1b: %g"
                        "\nADBb: %g"
                        "\nEHSb: %g"
                        "\nAvgModDiff1b: %g"
                        "\nAvgModDiff2b %g"
                        "\nRmsNoiseLoudb: %g"
                        "\nMFPDb: %g"
                        "\nRelDistFramesb: %g"
                        "\nDI: %g"
                        "\nODG: %g\n",
                        count,
                        #ifdef DATABOUND_BE
                        totalframes, boundbe.begin, boundbe.end,
                        #endif
                        processed.BandwidthRefb,
                        processed.BandwidthTestb, processed.TotalNMRb,
                        processed.WinModDiff1b, processed.ADBb,
                        processed.EHSb, processed.AvgModDiff1b,
                        processed.AvgModDiff2b,
                        processed.RmsNoiseLoudb, processed.MFPDb,
                        processed.RelDistFramesb,
                        oveRet.DI, oveRet.ODG);
}

{
FILE *res;

res = fopen(LOGRESULT,"a+");
fprintf(res,"\nFile: %s\n"
            "\nframe: %d"
            "\nBandwidthRefb: %g"
            "\nBandwidthTestb: %g"
```

```
                "\nTotalNMRb %g"
                "\nWinModDiff1b: %g"
                "\nADBb: %g"
                "\nEHSb: %g"
                "\nAvgModDiff1b: %g"
                "\nAvgModDiff2b %g"
                "\nRmsNoiseLoudb: %g"
                "\nMFPDb: %g"
                "\nRelDistFramesb: %g"
                "\nDI: %g"
                "\nODG: %g\n",
                filetest, count, processed.BandwidthRefb,
                processed.BandwidthTestb, processed.TotalNMRb,
                processed.WinModDiff1b, processed.ADBb,
                processed.EHSb, processed.AvgModDiff1b,
                processed.AvgModDiff2b,
```

```
                processed.RmsNoiseLoudb, processed.MFPDb,
                processed.RelDistFramesb,
                oveRet.DI, oveRet.ODG);

        fclose(res);
        }
    }
    if(numchref == 2) {
        if (fseek(fpref, HANN*bitsampleref/8, SEEK_CUR) == -1)
            fatalerr("err: %s", strerror(errno));
        if (fseek(fptest, HANN*bitsampletest/8, SEEK_CUR) == -1)
            fatalerr("err: %s", strerror(errno));

        #ifdef DATABOUND_BE
        #undef DATABOUND_ONE
        {
        int i = 0, flag = 0, f1, f2;
        long dataref, datatest, br1, br2;

        dataref = ftell(fpref);
        datatest = ftell(fptest);

        while(1) {
            br1 = ftell(fpref);
            br2 = ftell(fptest);
            f1 = GetStereoFrame(fpref, (signed int *)ch1ref,
                        (signed int *)ch2ref, bitsampleref/8, HANN);
            f2 = GetStereoFrame(fptest, (signed int *)ch1test,
                          (signed int *)ch2test, bitsampletest/8, HANN);
            if(f1 && f2) {
            totalframes++;
                if(boundary(ch1ref, ch1test, ch2ref, ch2test, HANN) && !flag) {
                    boundbe.begin = totalframes;
                    flag = 1;
                }
            }
            else {
               fseek(fptest, br1, SEEK_SET);
                fseek(fpref, br2, SEEK_SET);
                break;
            }
        }
        fseek(fptest, -HANN*bitsampletest/8, SEEK_CUR);
        fseek(fpref, -HANN*bitsampleref/8, SEEK_CUR);
        while(i<totalframes) {
            GetStereoFrame(fpref, (signed int *)ch1ref,
                          (signed int *)ch2ref, bitsampleref/8, HANN);
```

```
    GetStereoFrame(fptest, (signed int *)ch1test,
                        (signed int *)ch2test, bitsampletest/8, HANN);
    fseek(fptest, -2*HANN*bitsampletest/8, SEEK_CUR);
    fseek(fpref, -2*HANN*bitsampleref/8, SEEK_CUR);
    i++;
    if(boundary(ch1ref, ch1test, ch2ref, ch2test, HANN)) {
        boundbe.end = totalframes-i+1;
```

```
        break;
    }
}
fseek(fptest, datatest, SEEK_SET);
fseek(fpref, dataref, SEEK_SET);
}
#endif
while (GetStereoFrame(fpref, (signed int *)ch1ref,
                    (signed int *)ch2ref, bitsampleref/8, HANN)
&& GetStereoFrame(fptest, (signed int *)ch1test,
                     (signed int *)ch2test, bitsampletest/8, HANN)) {
    count++;
    #ifdef DATABOUND_BE
    if(count >= boundbe.begin && count <= boundbe.end)
        boundflag = 1;
    else
        boundflag = 0;
    #else
    boundflag = boundary(ch1ref, ch1test, ch2ref, ch2test, HANN);
      #ifdef DATABOUND_ONE
      {
      static int flag1 = 0, flag2 = 0;
      if(boundflag && !flag1)
          flag1 = 1;
       if(!boundflag && flag1)
          flag2 = 1;
       if(flag2)
          boundflag = 0;
       }
      #endif
    #endif
    ProcessFrame((signed int *)ch1ref,
              (signed int *)ch2ref, lpref,
                (signed int *)ch1test,
                  (signed int *)ch2test,
                  lptest, rateref, boundflag, HANN);
    oveRet = neural(processed);
    fprintf(stdout,"\nframe: %d"
                    #ifdef DATABOUND_BE
                    "/%d"
                    "\ndata boundary: %d -> %d"
                    #endif
                    "\nBandwidthRefb: %g"
                    "\nBandwidthTestb: %g"
                    "\nTotalNMRb %g"
                    "\nWinModDiff1b: %g"
                    "\nADBb: %g"
```

```
                    "\nEHSb: %g"
                    "\nAvgModDiff1b: %g"
                    "\nAvgModDiff2b %g"
                    "\nRmsNoiseLoudb: %g"
                    "\nMFPDb: %g"
                    "\nRelDistFramesb: %g"
                    "\nDI: %g"
                    "\nODG: %g\n",
                    count,
                    #ifdef DATABOUND_BE
                    totalframes, boundbe.begin, boundbe.end,
                    #endif
                    processed.BandwidthRefb,
                    processed.BandwidthTestb, processed.TotalNMRb,
                    processed.WinModDiff1b, processed.ADBb,
                    processed.EHSb, processed.AvgModDiff1b,
                    processed.AvgModDiff2b,
                    processed.RmsNoiseLoudb, processed.MFPDb,
                    processed.RelDistFramesb,
                    oveRet.DI, oveRet.ODG);
        }
        {
        FILE *res;
        res = fopen(LOGRESULT,"a+");
        fprintf(res,"\nFile: %s\n"
                    "\nframe: %d"
                    "\nBandwidthRefb: %g"
                    "\nBandwidthTestb: %g"
                    "\nTotalNMRb %g"
                    "\nWinModDiff1b: %g"
                    "\nADBb: %g"
                    "\nEHSb: %g"
                    "\nAvgModDiff1b: %g"
                    "\nAvgModDiff2b %g"
                    "\nRmsNoiseLoudb: %g"
                    "\nMFPDb: %g"
                    "\nRelDistFramesb: %g"
                    "\nDI: %g"
                    "\nODG: %g\n",
                    filetest, count, processed.BandwidthRefb,
                    processed.BandwidthTestb, processed.TotalNMRb,
                    processed.WinModDiff1b, processed.ADBb,
                    processed.EHSb, processed.AvgModDiff1b,
                    processed.AvgModDiff2b,
                    processed.RmsNoiseLoudb, processed.MFPDb,
                    processed.RelDistFramesb,
                    oveRet.DI, oveRet.ODG);
        fclose(res);
        }
    }
```

```
    fftw_destroy_plan(plan);
    fclose(fpref);
    fclose(fptest);
    return 0;
}
void
ProcessFrame(signed int *ch1ref, signed int *ch2ref, int lpref,
             signed int *ch1test, signed int *ch2test, int lptest,
             int rate, int boundflag, int hann)
{
```

```
int k;
static int ch = 1;
double Ntotaltest, Ntotalref;
struct levpatadaptout lev;
struct moddiffout mod;
struct processing processch1, processch2;
earmodelfft(ch1ref, lpref, hann, processch1.ffteref,
        processch1.fftref);
earmodelfft(ch1test, lptest, hann, processch1.fftetest,
            processch1.ffttest);

critbandgroup(processch1.ffteref, rate, hann, processch1.ppref);
AddIntNoise(processch1.ppref);

critbandgroup(processch1.fftetest, rate, hann, processch1.pptest);
AddIntNoise(processch1.pptest);

for(k=0;k<hann/2;k++)
    processch1.fnoise[k] = module(processch1.ffteref[k])
                         - module(processch1.fftetest[k]);

critbandgroup(processch1.fnoise, rate, hann, processch1.ppnoise);

spreading(processch1.pptest, processch1.E2test);
spreading(processch1.ppref, processch1.E2ref);
timespreading(processch1.E2test, Etesttmpch1, rate, processch1.Etest);
timespreading(processch1.E2ref, Ereftmpch1, rate, processch1.Eref);

threshold(processch1.Eref, processch1.Mref);

modulation(processch1.E2test, rate, &modintestch1, processch1.Modtest);
modulation(processch1.E2ref, rate, &modinrefch1, processch1.Modref);
// Data boundary
if(boundflag) {
    static int countboundary = 1;
    static double RelDistFramesb = 0, nmrtmp = 0;

    bandwidth(processch1.ffttest, processch1.fftref, hann,
    &bandwidthch1);
    processed.BandwidthRefb = bandwidthch1.BandwidthRefb;
    processed.BandwidthTestb = bandwidthch1.BandwidthTestb;

    processed.TotalNMRb = nmr(processch1.ppnoise, processch1.Mref,
                          &nmrtmp, countboundary);
    processed.RelDistFramesb = reldistframes(processch1.ppnoise,
                                             processch1.Mref,
                                             &RelDistFramesb,
                                             countboundary);
    countboundary++;

    // Data boundary + Energy threshold
    if(energyth(ch1test, ch1ref, hann)) {
       static int countenergy = 1;
       static double EHStmp = 0;

       processed.EHSb = harmstruct(processch1.ffttest,
                                   processch1.fftref,
                                   &EHStmp, rate, Cfftmpch1,
                                   harmsamples, &countenergy);
       countenergy++;
    }
}
// Delayed Averaging
if(count > delaytime2) {
    static double nltmp = 0;
    static int noise = 0, internal_count = 0, loudcounter = 0;
```

```
    mod = moddiff(processch1.Modtest, processch1.Modref,
            (double *)&(modinrefch1.Etildetmp));
    processed.WinModDiff1b = ModDiff1(mod, &moddiffinch1,
                                count - delaytime2);
    processed.AvgModDiff1b = ModDiff2(mod, &moddiffinch1);
    processed.AvgModDiff2b = ModDiff3(mod, &moddiffinch1);

    Ntotaltest = loudness(processch1.Etest);
    Ntotalref = loudness(processch1.Eref);

    if(Ntotaltest > 0.1 || Ntotalref > 0.1) {
       noise = 1;
       #if defined(LOUDMODO2)
       internal_count = 0;
       #endif
    }

    // Delayed Averaging + loudness threshold
    if(noise && internal_count <= delaytime1) {
       // skip 0.05 sec (about 3 frames)
       internal_count++;
       loudcounter++;
    }
    else {
       lev = levpatadapt(processch1.Etest, processch1.Eref, rate,
                &levinch1, hann);
       processed.RmsNoiseLoudb = noiseloudness(processch1.Modtest,
                                    processch1.Modref,
                                    lev, &nltmp,
                                    count - delaytime2
                                    - loudcounter);
    }
}
```

```
/*{
extern double Cfft[];
extern int maxk;
FILE *fp;

logvariable("Cfftsx.txt", Cfft, 128);
fp = fopen("Cfftsxmaxpos.txt", "a+");
fprintf(fp,"%d\n",maxk);
fclose(fp);
}*/

if(*ch2ref && *ch2test) {
    ch = 2;

    earmodelfft(ch2ref, lpref, hann, processch2.ffteref,
            processch2.fftref);
    earmodelfft(ch2test, lptest, hann, processch2.fftetest,
            processch2.ffttest);

    critbandgroup(processch2.ffteref, rate, hann, processch2.ppref);
    AddIntNoise(processch2.ppref);

    critbandgroup(processch2.fftetest, rate, hann, processch2.pptest);
    AddIntNoise(processch2.pptest);

    for(k=0;k<hann/2;k++)
       processch2.fnoise[k] = module(processch2.ffteref[k])
                                - module(processch2.fftetest[k]);

    critbandgroup(processch2.fnoise, rate, hann, processch2.ppnoise);

    spreading(processch2.pptest, processch2.E2test);
    spreading(processch2.ppref, processch2.E2ref);
```

```
timespreading(processch2.E2test, Etesttmpch2, rate,
            processch2.Etest);
timespreading(processch2.E2ref, Ereftmpch2, rate,
            processch2.Eref);

threshold(processch2.Eref, processch2.Mref);

modulation(processch2.E2test, rate, &modintestch2,
          processch2.Modtest);
modulation(processch2.E2ref, rate, &modinrefch2,
          processch2.Modref);

// Data boundary
if(boundflag) {
   static int countboundary = 1;
   static double RelDistFramesb = 0, nmrtmp = 0;

   bandwidth(processch2.ffttest, processch2.fftref, hann,
&bandwidthch2);
   processed.BandwidthRefb += bandwidthch2.BandwidthRefb;
   processed.BandwidthTestb += bandwidthch2.BandwidthTestb;
   processed.BandwidthRefb /= 2.0;
   processed.BandwidthTestb /= 2.0;

   processed.TotalNMRb += nmr(processch2.ppnoise,
                              processch2.Mref, &nmrtmp,
```

```
                              countboundary);
   processed.RelDistFramesb += reldistframes(processch2.ppnoise,
                                             processch2.Mref,
                                          &RelDistFramesb,
                                             countboundary);
   processed.TotalNMRb /= 2.0;
   processed.RelDistFramesb /= 2.0;
   countboundary++;

   // Data boundary + Energy threshold
   if(energyth(ch2test, ch2ref, hann)) {
      static int countenergy = 1;
      static double EHStmp = 0;

      processed.EHSb += harmstruct(processch2.ffttest,
                                   processch2.fftref,
                                &EHStmp, rate, Cfftmpch2,
                                   harmsamples, &countenergy);
      processed.EHSb /= 2.0;
      countenergy++;
   }
}

// Delayed Averaging
if(count > delaytime2) {
   static double nltmp = 0;
   static int noise = 0, internal_count = 0, loudcounter = 0;

   mod = moddiff(processch2.Modtest, processch2.Modref,
            (double *)&(modinrefch2.Etildetmp));
   processed.WinModDiff1b += ModDiff1(mod, &moddiffinch2,
                                      count - delaytime2);
   processed.AvgModDiff1b += ModDiff2(mod, &moddiffinch2);
   processed.AvgModDiff2b += ModDiff3(mod, &moddiffinch2);
   processed.WinModDiff1b /= 2.0;
   processed.AvgModDiff1b /= 2.0;
   processed.AvgModDiff2b /= 2.0;

   Ntotaltest = loudness(processch2.Etest);
   Ntotalref = loudness(processch2.Eref);
```

```
        if(Ntotaltest > 0.1 || Ntotalref > 0.1) {
            noise = 1;
            #if defined(LOUDMODO2)
            internal_count = 0;
            #endif
        }

        // Delayed Averaging + loudness threshold
        if(noise && internal_count <= delaytime1) {
            // skip 0.05 sec (about 3 frames)
            internal_count++;
            loudcounter++;
        }
```

```
        else {
            lev = levpatadapt(processch2.Etest, processch2.Eref, rate,
                    &levinch2, hann);
            processed.RmsNoiseLoudb += noiseloudness(processch2.Modtest,
                                                processch2.Modref,
                                                lev, &nltmp,
                                                count - delaytime2
                                                - loudcounter);
            processed.RmsNoiseLoudb /= 2.0;
        }
    }
}

{
static int ndistorcedtmp = 0;
static double Ptildetmp = 0, PMtmp = 0, Qsum = 0;

if(ch == 2)
    processed.ADBb = detprob(processch1.Etest, processch2.Etest,
                        processch1.Eref, processch2.Eref,
                        &Ptildetmp, &PMtmp, &Qsum,
                        &ndistorcedtmp, hann);
else
    processed.ADBb = detprob(processch1.Etest, NULL,
                        processch1.Eref, NULL,
                        &Ptildetmp, &PMtmp, &Qsum,
                        &ndistorcedtmp, hann);
processed.MFPDb = PMtmp;
}
/*
#ifdef LOGVARIABLE
logvariable("fftetestsx.txt", processch1.fftetest, hann/2);
logvariable("ffterefsx.txt", processch1.ffteref, hann/2);
logvariable("ffttestsx.txt", processch1.ffttest, hann/2);
logvariable("fftrefsx.txt", processch1.fftref, hann/2);
logvariable("Etestsx.txt", processch1.Etest, bark);
logvariable("Erefsx.txt", processch1.Eref, bark);
logvariable("E2testsx.txt", processch1.E2test, bark);
logvariable("E2refsx.txt", processch1.E2ref, bark);
logvariable("pptestsx.txt", processch1.pptest, bark);
logvariable("pprefsx.txt", processch1.ppref, bark);
logvariable("ppnoisesx.txt", processch1.ppnoise, bark);
logvariable("Mrefsx.txt", processch1.Mref, bark);
logvariable("Modtestsx.txt", processch1.Modtest, bark);
logvariable("Modrefsx.txt", processch1.Modref, bark);

logvariable("fftetestdx.txt", processch2.fftetest, hann/2);
logvariable("ffterefdx.txt", processch2.ffteref, hann/2);
logvariable("ffttestdx.txt", processch2.ffttest, hann/2);
```

```
    logvariable("fftrefdx.txt", processch2.fftref, hann/2);
    logvariable("Etestdx.txt", processch2.Etest, bark);
    logvariable("Erefdx.txt", processch2.Eref, bark);
    logvariable("E2testdx.txt", processch2.E2test, bark);
    logvariable("E2refdx.txt", processch2.E2ref, bark);
```

```
    logvariable("pptestdx.txt", processch2.pptest, bark);
    logvariable("pprefdx.txt", processch2.ppref, bark);
    logvariable("ppnoisedx.txt", processch2.ppnoise, bark);
    logvariable("Mrefdx.txt", processch2.Mref, bark);
    logvariable("Modtestdx.txt", processch2.Modtest, bark);
    logvariable("Modrefdx.txt", processch2.Modref, bark);
    #endif
    */

    /*{
    extern double Cfft[];
    extern int maxk;
    FILE *fp;

    logvariable("Cfftdx.txt", Cfft, 128);
    fp = fopen("Cfftdxmaxpos.txt", "a+");
    fprintf(fp,"%d\n",maxk);
    fclose(fp);
    }*/
    return;
}

void
fatalerr(char * pattern,...) /* Error handling routine */
{
    va_list ap;

    va_start(ap, pattern);

    fprintf(stderr,"PEAQ-");
    vfprintf(stderr,pattern,ap);
    fprintf(stderr," (exit forced).\n");

    va_end(ap);

    exit(-1);
}

#ifdef DEBUG
void
debug(char * pattern,...) /* Debug handling routine */
{
    FILE *log;
    va_list ap;

    va_start(ap, pattern);

    log = fopen(LOGFILE,"a+");
    vfprintf(log,pattern,ap);

    va_end(ap);
    fclose(log);

    return;
}
#endif

#ifdef LOGVARIABLE
void
```

```
logvariable(const char *filename, double *var, int len)
{
```

```
    FILE *fp;
    int k;

    #ifdef LOGALLFRAMES
    fp = fopen(filename,"a+");
    #else
    fp = fopen(filename,"w");
    #endif

    for(k=0;k<len;k++)
    fprintf(fp,"%g\n",var[k]);

    fclose(fp);
    return;
}
#endif

void
usage(char * name) /* Print usage */
{
    fprintf(stderr, "PEAQ Algorithm. Giuseppe Gottardi 'oveRet'"
                    "<gottardi@ailinux.org>\n\n");

    fprintf(stderr, "usage: %s <option>\n", name);
    fprintf(stderr, "     -r  reffile[:lp] (lp default = 92)\n"
                    "     -t  testfile[:lp] (lp default = 92)\n"
                    "     -h  print this help\n");

    exit (0);
}
```

Файл: bandwidth.h

```
#define ZEROTHRESHOLD 921
#define BwMAX 346

struct bandwidthout {
    double sumBandwidthRefb;
    int countref;
    double sumBandwidthTestb;
    int counttest;
    double BandwidthRefb;
    double BandwidthTestb;
};

/* Function prototypes */
int bandwidth(double *, double *, int, struct bandwidthout *);
/* Prototypes end */
```

Файл: bandwidth.c

```
#include <stdlib.h>
#include <math.h>
#include <common.h>
#include <bandwidth.h>

int
bandwidth(double *ffttest, double *fftref, int hann,
          struct bandwidthout *out)
{
    int k, BwRef = 0, BwTest = 0;
    double Flevtest, Flevref;
    double ZeroThreshold;

    ZeroThreshold = 20.0 * log10((double)ffttest[ZEROTHRESHOLD]);

    for(k=ZEROTHRESHOLD;k<hann/2;k++) {
        Flevtest = 20.0 * log10((double)ffttest[k]);
```

```
            if(Flevtest > ZeroThreshold)
                ZeroThreshold = Flevtest;
        }
        for(k=ZEROTHRESHOLD-1;k>=0;k--) {
            Flevref = 20.0 * log10((double)fftref[k]);
            if(Flevref >= 10.0+ZeroThreshold) {
                BwRef = k + 1;
                break;
            }
        }
        for(k=BwRef-1;k>=0;k--) {
            Flevtest = 20.0 * log10((double)fftest[k]);
            if(Flevtest >= 5.0+ZeroThreshold) {
                BwTest = k + 1;
                break;
            }
        }
        if(BwRef > BwMAX) {
            out->sumBandwidthRefb += (double)BwRef;
            out->countref++;
        }
        if(BwTest > BwMAX) {
            out->sumBandwidthTestb += (double)BwTest;
            out->counttest++;
        }
        if(out->countref == 0)
            out->BandwidthRefb = 0;
        else
            out->BandwidthRefb = out->sumBandwidthRefb/(double)out->countref;
        if(out->counttest == 0)
            out->BandwidthTestb = 0;
        else
            out->BandwidthTestb = out->sumBandwidthTestb/(double)out->counttest;
        return 0;
    }
```

Файл: boundary.h

```
#define BOUNDWIN 5
#define BOUNDLIMIT 200

struct boundaryflag {
    int begin;
    int end;
};

/* Function prototypes */
int boundary(signed int *, signed int *, signed int *, signed int *, int);
/* Prototypes end */
```

Файл: boundary.c

```
#include <stdlib.h>
#include <math.h>
#include <common.h>
#include <boundary.h>

int
boundary(signed int *ch1ref, signed int *ch1test, signed int *ch2ref,
        signed int *ch2test, int hann)
{
```

```
    int k, i, sum;
    int ch1t = 0, ch1r = 0, ch2t = 0, ch2r = 0;

    for(k=0;k<hann-BOUNDWIN+1;k++) {
        if(!ch1t) {
            sum = 0;
            for(i=0;i<BOUNDWIN;i++)
                sum += abs(ch1test[k+i]);
            if(sum > BOUNDLIMIT)
                ch1t = 1;
        }

        if(!ch1r) {
            sum = 0;
            for(i=0;i<BOUNDWIN;i++)
                sum += abs(ch1ref[k+i]);
            if(sum > BOUNDLIMIT)
                ch1r = 1;
         }

        if(ch1t || ch1r) // || or &&
            return 1;
    }
    if(ch2test == NULL && ch2ref == NULL)
        return 0;

    for(k=0;k<hann-BOUNDWIN+1;k++) {
        if(!ch2t) {
            sum = 0;
            for(i=0;i<BOUNDWIN;i++)
                sum += abs(ch2test[k+i]);
            if(sum > BOUNDLIMIT)
                ch2t = 1;
        }

        if(!ch2r) {
            sum = 0;
            for(i=0;i<BOUNDWIN;i++)
        sum += abs(ch2ref[k+i]);
            if(sum > BOUNDLIMIT)
        ch2r = 1;
        }

        if((ch1t || ch2t) || (ch1r || ch2r)) // || or &&
            return 1;
    }

    return 0;
}
```

Файл: critbandgroup.h

```
/* Function prototypes */
int critbandgroup(double *, int, int, double *);
int AddIntNoise(double *);
/* Prototypes end */
```

Файл: critbandgroup.c

```
#include <stdlib.h>
#include <math.h>
#include <common.h>
#include <critbandgroup.h>

extern double *fL, *fC, *fU;
extern int bark;
```

```
int
critbandgroup(double *ffte, int rate, int hann, double *pe)
{
    double fres;
    int i, k;

    fres = (double)rate/hann;

    for(i=0;i<bark;i++) {
        pe[i] = 0;
        for(k=0;k<hann/2;k++) {
            if(((double)(k-0.5)*fres) >= fL[i]
            && ((double)(k+0.5)*fres <= fU[i]))
                pe[i] += p(ffte[k], 2.0);
            else
                if(((double)(k-0.5)*fres) < fL[i]
                && ((double)(k+0.5)*fres > fU[i]))
                    pe[i] += p(ffte[k], 2.0)*(fU[i]-fL[i])/fres;
                else
                if(((double)(k-0.5)*fres) < fL[i]
                && ((double)(k+0.5)*fres > fL[i]))
                    pe[i] += p(ffte[k], 2.0)*(double)((k+0.5)
                        *fres-fL[i])/fres;
                else
                if(((double)(k-0.5)*fres) < fU[i]
                && ((double)(k+0.5)*fres > fU[i]))
                    pe[i] += p(ffte[k], 2.0)*(fU[i]-(double)(k-0.5)
                        *fres)/fres;
        }

        if(pe[i] < p(10.0, -12.0))
            pe[i] = p(10.0, -12.0);
    }

    return 0;
}

int
AddIntNoise(double *pe)
{
    int k;
    double Pthres;

    for(k=0;k<bark;k++) {
        Pthres = p(10.0, 0.4*0.364*p(fC[k]/1000.0, -0.8));
        pe[k] += Pthres;
    }

    return 0;
}
```

Файл: detprob.h

```
/* Function prototypes */
double detprob(double *, double *, double *, double *,
            double *, double *, int *, int);
/* Prototypes end */
```

Файл: detprob.c

```
#include <stdlib.h>
#include <math.h>
#include <common.h>
#include <detprob.h>

extern int bark;
```

```
double
detprob(double *Etestch1, double *Etestch2, double *Erefch1,
        double *Erefch2, double *Ptildetmp, double *PMtmp,
        double *Qsum, int *ndistorcedtmp, int hann)
{
   int k;
   double Etilderefch1, Etilderefch2, Etildetestch1, Etildetestch2;
   double L, s, e, b, a, pch1, pch2, pbin,
          qch1, qch2, qbin, P, c0, c1, ADBb;
   double prod = 1.0, Q = 0.0;

   for(k=0;k<bark;k++) {
      Etildetestch1 = 10.0*log10((double)Etestch1[k]);
      Etilderefch1 = 10.0*log10((double)Erefch1[k]);

      if(Etilderefch1 > Etildetestch1)
         L = 0.3*Etilderefch1;
      else
         L = 0.3*Etildetestch1;

      L += 0.7*Etildetestch1;

      if(L > 0)
         s = 5.95072*p(6.39468/L, 1.71332)+9.01033*p(10.0, -11.0)
            *p(L, 4.0)+5.05622*p(10.0, -6.0)*p(L, 3.0)-0.00102438
             *p(L, 2.0)+0.0550197*L-0.198719;
      else
         s = p(10.0, 30.0);

      e = Etilderefch1 - Etildetestch1;

      if(Etilderefch1 > Etildetestch1)
         b = 4.0;
      else
         b = 6.0;

      a = (double)p(10.0, log10((double)log10((double)2.0))/b)/s;
      pch1 = 1.0 - p(10.0, -p(a*e, b));
      qch1 = abs((int)e)/s;  // don't touch this

      pbin = pch1;
      qbin = qch1;

      if(Etestch2 != NULL && Erefch2 != NULL) {
         Etildetestch2 = 10.0*log10((double)Etestch2[k]);
         Etilderefch2 = 10.0*log10((double)Erefch2[k]);

         if(Etilderefch2 > Etildetestch2)
            L = 0.3*Etilderefch2;
         else
            L = 0.3*Etildetestch2;

         L += 0.7*Etildetestch2;

         if(L > 0)
            s = 5.95072*p(6.39468/L, 1.71332)+9.01033*p(10.0, -11.0)
               *p(L, 4.0)+5.05622*p(10.0, -6.0)*p(L, 3.0)
               -0.00102438*p(L, 2.0)+0.0550197*L-0.198719;
         else
            s = 1.0*p(10.0, 30.0);

         e = Etilderefch2 - Etildetestch2;

         if(e > 0)
            b = 4.0;
         else
            b = 6.0;

         a = (double)p(10.0, log10((double)log10((double)2.0))/b)/s;
```

```
            pch2 = 1.0 - p(10.0, -p(a*e, b));
            qch2 = abs((int)e)/s;  // don't touch this

            if(pch2 > pch1)
                pbin = pch2;
            if(qch2 > qch1)
                qbin = qch2;
        }
        prod *= (1.0 - pbin);
        Q += qbin;
    }

    P = 1.0 - prod;
    if(P > 0.5) {
        *Qsum += Q;
        (*ndistorcedtmp)++;
    }

    if(*ndistorcedtmp == 0)
        ADBb = 0;
    else
    if(*Qsum > 0)
        ADBb = log10((double)*Qsum / (*ndistorcedtmp));
    else
        ADBb = -0.5;

    c0 = p(0.9, hann/(2.0*1024.0));
    #if !defined(C1)
    c1 = p(0.99, hann/(2.0*1024.0));
    #else
    c1 = C1;
    #endif
    *Ptildetmp = (1.0 - c0)*P + (*Ptildetmp)*c0;
    if(*Ptildetmp > (*PMtmp)*c1)
        *PMtmp = *Ptildetmp;
    else
        *PMtmp = (*PMtmp)*c1;

    return ADBb;
}
```

Файл: earmodelfft.h

```
#define NORM 11361.301063573899
#define FREQADAP 23.4375 // for 48 kHz

/* Function prototypes */
int earmodelfft(signed int *, int, int, double *, double *);
/* Prototypes end */
```

Файл: earmodelfft.c

```
#include <stdlib.h>
#include <math.h>
#include <fftw.h>
#include <common.h>
#include <earmodelfft.h>

extern double hannwindow[ ];
extern fftw_plan plan;

int
earmodelfft(signed int *ch, int lp, int hann, double *ffte, double *absfft)
{
    int k;
    double w, fac;
```

```
    fftw_complex in[HANN], out[HANN];

    fac = p(10.0, lp/20.0)/NORM;

    for(k=0;k<hann;k++) {
        in[k].re = hannwindow[k] * (double)ch[k];
        in[k].im = 0;
    }

    fftw_one(plan, in, out);

    for(k=0;k<hann/2;k++) {
        out[k].re *= (double)(fac/hann);
        out[k].im *= (double)(fac/hann);
        absfft[k] = sqrt((double)(p(out[k].re, 2.0) + p(out[k].im, 2.0)));

        w = -0.6*3.64*p(k * FREQADAP/1000.0, -0.8) +
            6.5*exp((double)-0.6*p(k * FREQADAP/1000.0 - 3.3, 2.0)) -
            0.001*p(k * FREQADAP/1000.0, 3.6);

        ffte[k] = absfft[k]*p(10.0, w/20.0);

    }

    return 0;
}
```

Файл: energyth.h

```
#define ENERGYLIMIT 8000

/* Function prototypes */
int energyth(signed int *, signed int *, int);
/* Prototypes end */
```

Файл: energyth.c

```
#include <stdlib.h>
#include <math.h>
#include <common.h>
#include <energyth.h>

int
energyth(signed int *test, signed int *ref, int hann)
{
    int k;
    double sum;

    sum = 0;
    for(k=0;k<hann/2;k++) {
    sum += p(test[hann/2 + k], 2.0);
    if(sum > ENERGYLIMIT)
        return 1;
    }

    sum = 0;
    for(k=0;k<hann/2;k++) {
    sum += p(ref[hann/2 + k], 2.0);
    if(sum > ENERGYLIMIT)
        return 1;
    }

    return 0;
}
```

Файл: getframe.h

```
#define LP 92

/* Function prototypes */
signed int GetFrameValue(FILE *, int);
```

```
int GetMonoFrame(FILE *, signed int *, int , int );
int GetStereoFrame(FILE *, signed int *, signed int *, int , int);
int LevelPression(char *);
int ReadInt(FILE *, int);
void fatalerr(char *,...);
/* Prototypes end */
```

Файл: getframe.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <getframe.h>

extern int errno;

signed int
GetFrameValue(FILE *fp, int bytes)
{
   int intvalue;

   if (bytes <= 0)
        return 0;
   intvalue = ReadInt(fp, bytes);
   switch(bytes) {
        case 3:
          if (intvalue & 0x00800000)
               intvalue |= 0xff000000;
          break;
        case 2:
          if (intvalue & 0x00008000)
               intvalue |= 0xffff0000;
          break;
        case 1:
          if (intvalue & 0x00000080)
               intvalue |= 0xffffff00;
          break;
   }

   return (signed int) intvalue;
}

int
GetMonoFrame(FILE *fp, signed int *vect, int bytes, int hann)
{
   int i = 0;

   if (fp == NULL)
        return 0;

   if (fseek(fp, (-hann/2)*bytes, SEEK_CUR) == -1)
        fatalerr("err: %s", strerror(errno));

   while(!feof(fp) && i < hann) {
        vect[i] = GetFrameValue(fp, bytes);
        i++;
   }

   if(i < hann) {
        bzero(vect, hann*4);
        fseek(fp, -i*bytes, SEEK_END);
        return 0;
   }

   /* Number of samples wrote */
   return i ;
}
```

```
int
GetStereoFrame(FILE *fp, signed int *sx, signed int *dx, int bytes,
            int hann)
{
   int i = 0, k = 0, count = 0;

   if (fp == NULL)
      return 0;

   if (fseek(fp, -hann*bytes, SEEK_CUR) == -1)
      fatalerr("err: %s", strerror(errno));

   while(!feof(fp) && count < hann*2) {
      if(!k) {
         sx[i] = GetFrameValue(fp, bytes);
         k = 1;
         i--;
         count++;
      }
      else {
         dx[i] = GetFrameValue(fp, bytes);
         k = 0;
         count++;
      }
      i++;
   }
   if (count < hann*2) {
      bzero(sx, hann*4);
      bzero(dx, hann*4);
      fseek(fp, -count*bytes, SEEK_END);
      return 0;
   }

   /* Number of samples wrote */
   return count ;
}

int
LevelPression(char *f)
{
   int lp;

   while(*f != ':' && *f)
      f++;

   if(*f == ':') {
      lp = atoi(f + 1);
      *f = '\0';
      return lp;
   }
   else
   return LP;
}
```

Файл: harmstruct.h

```
/* Function prototypes */
double harmstruct(double *, double *, double *, int, double *, int, int *);
void debug(char *,...);
/* Prototypes end */
```

Файл: harmstruct.c

```
#include <stdlib.h>
#include <math.h>
#include <string.h>
```

```
#include <fftw.h>
#include <common.h>
#include <harmstruct.h>

extern char *filetest;
extern int count;
extern fftw_plan plan2;

double Cfft[HANN/2];
int maxk;

double
harmstruct(double *ffttest, double *fftref, double *EHStmp, int rate,
        double *Cffttmp, int p, int *n)
{
   int k, i;
   double F0[HANN/2], C[HANN/2], hannwin[HANN/2];
   double num, denoma, denomb, Csum = 0;
   fftw_complex in[HANN/2], out[HANN/2];
   double max;

   bzero(Cfft, 8 * HANN/2);

   for(k=0;k<p*2-1;k++) {
      if(!fftref[k] || !ffttest[k]) {  // skip log(0)
         #if defined(SKIPFRAME) && !defined(ZERO)
         (*n)--;
         return 0;
         #elif defined(ZERO)
         if(!fftref[k]) {
            fftref[k] = ZERO;
            #ifdef DEBUG
            debug("Warning [%s:%d] in Harmstruct.c: "
               "fftref[%d] is set around zero\n",
               filetest, count, k, fftref[k]);
            #endif
         }
         if(!ffttest[k]) {
            ffttest[k] = ZERO;
            #ifdef DEBUG
            debug("Warning [%s:%d] in Harmstruct.c: "
               "ffttest[%d] is set around zero\n",
               filetest, count, k, ffttest[k]);
            #endif
         }
         F0[k] = log10(p(fftref[k], 2.0)) - log10(p(ffttest[k], 2.0));
         #else
         if(!fftref[k] && !ffttest[k])
            F0[k] = 0;
         else {
            #ifdef DEBUG
            debug("Error [%s:%d] in Harmstruct.c: "
               "log(fftref[%d] = %g  log(ffttest[%d] = %g\n",
               filetest, count, k, fftref[k], k, ffttest[k]);
            #endif
            F0[k] = 0;

         }
         #endif
      }
      else
         F0[k] = log10(p(fftref[k], 2.0)) - log10(p(ffttest[k], 2.0));
   }

   for(i=0;i<p;i++) {
      num = 0;
```

```
    denoma = 0;
    denomb = 0;
    for(k=0;k<p;k++) {
        num += F0[k] * F0[i+k];
        denoma += p(F0[k], 2.0);
        denomb += p(F0[i+k], 2.0);
    }
    hannwin[i] = 0.5*sqrt((double)8.0/3.0)*(1.0 - cos((double)2.0
                *M_PI*i/(p-1.0)));
    C[i] = num / (sqrt((double)denoma) * sqrt((double)denomb));
    #if !defined(AVGHANN)
    C[i] *= hannwin[i];
    #endif
    Csum += C[i];
}
for(i=0;i<p;i++) {
    C[i] -= (double)Csum/p;
    #if defined(AVGHANN)
    C[i] *= hannwin[i];
    #endif
    in[i].im = 0;
    in[i].re = C[i];
}
fftw_one(plan2, in, out);
for(k=0;k<p/2;k++) {
    out[k].re *= (double)(1.0/p);
    out[k].im *= (double)(1.0/p);
    Cfft[k] = p(out[k].re, 2.0) + p(out[k].im, 2.0);
}
#ifdef EHSMODO2
for(k=0;k<p/2;k++) {
   Cffttmp[k] += Cfft[k];
    Cfft[k] = Cffttmp[k]/(*n);
}
#endif
#if !defined(GETMAX)
i = 0+PATCH;
while(1) {
    if(Cfft[i] >= Cfft[i+1]) {
        while(i < p/2-1 && Cfft[i] >= Cfft[i+1])
            i++;
        if(i < p/2-1)
            break;
        else {
            (*n)--;
            return 0;
        }
    }
    else {
        while(i < p/2-1 && Cfft[i] <= Cfft[i+1])
            i++;
        while(i < p/2-1 && Cfft[i] >= Cfft[i+1])
            i++;
        if(i < p/2-1)
            break;
        else {
            (*n)--;
            return 0;
```

```
            }
        }
    }
    #else
    i = 0;
    #endif

    max = 0;
    for(k=i+1;k<p/2;k++)
        if(Cfft[k] > max) {
            max = Cfft[k];
            maxk = k;
        }

    #ifdef EHSMODO2
    return max*1000.0;
    #endif

    (*EHStmp) += max;
    return ((*EHStmp)*1000.0/(*n));
}
```

Файл: levpatadapt.h

```
#define T100 0.05
#define Tmin 0.008
#define M 8
// if M is odd
/*
#define M1 (M-1)/2
#define M2 M1
*/
// if M is even
#define M1 M/2 - 1
#define M2 M/2

struct levpatadaptout {
    double Epref[BARK];
    double Eptest[BARK];
};

struct levpatadaptin {
    double Ptest[BARK];
    double Pref[BARK];
    double PattCorrTest[BARK];
    double PattCorrRef[BARK];
    double Rnum[BARK];
    double Rdenom[BARK];
};

/* Function prototypes */
struct levpatadaptout levpatadapt(double *, double *, int,
                                  struct levpatadaptin *, int);
/* Prototypes end */
```

Файл: levpatadapt.c

```
#include <stdlib.h>
#include <math.h>
#include <common.h>
#include <levpatadapt.h>

extern int bark;
extern double *fC;

struct levpatadaptout
```

```
levpatadapt(double *Etest, double *Eref, int rate,
        struct levpatadaptin *tmp, int hann)
{
   int k, i, m1, m2;
   double T, levcorr, numlevcorr = 0, denomlevcorr = 0, R;
   double pattcoeffref, pattcoefftest;
   double Elref[BARK], Eltest[BARK], Rtest[BARK], Rref[BARK], a[BARK];
   struct levpatadaptout out;

   for(k=0;k<bark;k++) {
      T = (double)Tmin + (100.0/fC[k])*(T100 - Tmin);
      a[k] = exp((double)-hann/(2.0*rate * T));
      tmp->Ptest[k] = tmp->Ptest[k]*a[k] + (1.0-a[k])*Etest[k];
      tmp->Pref[k] = tmp->Pref[k]*a[k] + (1.0-a[k])*Eref[k];
      numlevcorr += sqrt(tmp->Ptest[k]*tmp->Pref[k]);
      denomlevcorr += tmp->Ptest[k];
   }

   levcorr = p(numlevcorr/denomlevcorr, 2.0);

   for(k=0;k<bark;k++) {
      if(levcorr > 1.0) {
         Elref[k] = (double)Eref[k]/levcorr;
         Eltest[k] = Etest[k];
      }
      else {
         Eltest[k] = (double)Etest[k]*levcorr;
         Elref[k] = Eref[k];
      }

      // Autocorrelation
      tmp->Rnum[k] *= a[k];
      tmp->Rdenom[k] *= a[k];
      tmp->Rnum[k]  += Elref[k]*Eltest[k];
      tmp->Rdenom[k] += Elref[k]*Elref[k];

      if(tmp->Rdenom[k] == 0 && tmp->Rnum[k] != 0) {
         Rtest[k] = 0;
         Rref[k] = 1.0;
      }
      else
      if(tmp->Rdenom[k] == 0 && tmp->Rnum[k] == 0) {
         //copy from frequency band below
         if(k) {
            Rtest[k] = Rtest[k-1];
            Rref[k] = Rref[k-1];
         }
         //if don't exist
         else {
            Rtest[k] = 1.0;
            Rref[k] = 1.0;
         }
      }
      else {
         R = tmp->Rnum[k] / tmp->Rdenom[k];
         if(R >= 1.0) {
            Rtest[k] = 1.0/R;
            Rref[k] = 1.0;
         }
         else {
            Rtest[k] = 1.0;
            Rref[k] = R;
         }
      }
   }
```

```
    for(k=0;k<bark;k++) {
        m1 = M1;
        m2 = M2;
        pattcoefftest = 0;
        pattcoeffref = 0;

        if(m1 > k)
            m1 = k;
        if(m2 > bark -k -1)
            m2 = bark -k -1;

        for(i = -m1;i <= m2;i++) {
            pattcoefftest += Rtest[k+i];
            pattcoeffref += Rref[k+i];
        }

        tmp->PattCorrTest[k] = a[k]*tmp->PattCorrTest[k] +
                               pattcoefftest*(1.0-a[k])/(m1+m2+1);
        tmp->PattCorrRef[k] = a[k]*tmp->PattCorrRef[k] +
                              pattcoeffref*(1.0-a[k])/(m1+m2+1);

        out.Epref[k] = Elref[k] * tmp->PattCorrRef[k];
        out.Eptest[k] = Eltest[k] * tmp->PattCorrTest[k];
    }

    return out;

}
```

Файл: loudness.h

```
#define CONST 1.07664

/* Function prototypes */
double loudness(double *);
/* Prototypes end */
```

Файл: loudness.c

```
#include <stdlib.h>
#include <math.h>
#include <common.h>
#include <loudness.h>

extern double *fC;
extern int bark;

double
loudness(double *E)
{
    int k;
    double Ntot = 0;
    double s, N, Ethres;

    for(k=0;k<bark;k++) {
        s = p(10.0, (-2.0-2.05*atan((double)fC[k]/4000.0) - 0.75
            *atan((double)p(fC[k]/1600.0, 2.0)))/10.0);
        Ethres = p(10.0, 0.364*p(fC[k]/1000.0, -0.8));
        N = (double)CONST*p(Ethres/(s*10000.0), 0.23)
            *(p(1.0-s+s*E[k]/Ethres, 0.23) - 1.0);
        if(N > 0)
            Ntot += N;
    }

    Ntot *= (double)24.0/bark;

    return Ntot;
}
```

Файл: moddiff.h

```
#define L 4

struct moddiffin {
   double win;
   int Lcount;
   double modtmp;
   double mod[L];
   double num2;
   double denom2;
   double num3;
   double denom3;
};

struct moddiffout {
   double ModDiff1;
   double ModDiff2;
   double TempWt;
};

/* Function prototypes */
struct moddiffout moddiff(double *, double *, double *);
double ModDiff1(struct moddiffout, struct moddiffin *, int);
double ModDiff2(struct moddiffout, struct moddiffin *);
double ModDiff3(struct moddiffout, struct moddiffin *);
/* Prototypes end */
```

Файл: moddiff.c

```
#include <stdlib.h>
#include <assert.h>
#include <math.h>
#include <common.h>
#include <moddiff.h>

extern double *fC;
extern int bark;

struct moddiffout
moddiff(double *Modtest, double *Modref, double *Etilderef)
{
   int k;
   struct moddiffout out;
   double Pthres;

   out.ModDiff1 = 0;
   out.ModDiff2 = 0;
   out.TempWt = 0;

   for(k=0;k<bark;k++) {
      // WinModDiff1B && AvgModDiff1B
      out.ModDiff1 += module(Modtest[k] - Modref[k])/(1.0 + Modref[k]);
      // AvgModDiff2B
      if(Modtest[k] > Modref[k])
         out.ModDiff2 += module(Modtest[k] - Modref[k])
                  /(0.01 + Modref[k]);
      else
         out.ModDiff2 += 0.1*module(Modtest[k] - Modref[k])
                  /(0.01 + Modref[k]);

      Pthres = p(10.0, 0.4*0.364*p(fC[k]/1000.0, -0.8));
      out.TempWt += Etilderef[k]/(Etilderef[k] + p(Pthres, 0.3)*100.0);
   }

   out.ModDiff1 *= (double)100.0/bark;
   out.ModDiff2 *= (double)100.0/bark;
```

```
    return out;
}

double
ModDiff1(struct moddiffout in, struct moddiffin *intmp, int n)
{
    int i;

    intmp->mod[intmp->Lcount] = in.ModDiff1;
    intmp->Lcount++;
    if(intmp->Lcount == L)
    intmp->Lcount = 0;

    if(n < L)
    return 0;

    intmp->modtmp = 0;
    for(i=0;i<L;i++)
    intmp->modtmp += sqrt((double)intmp->mod[i]);

    intmp->modtmp /= (double)L;

    intmp->win += p(intmp->modtmp, 4.0);

    return sqrt((double)intmp->win/(double)(n-L+1.0));
}

double
ModDiff2(struct moddiffout in, struct moddiffin *intmp)
{
    intmp->num2 += in.ModDiff1 * in.TempWt;
    intmp->denom2 += in.TempWt;

    return (intmp->num2/intmp->denom2);
}

double
ModDiff3(struct moddiffout in, struct moddiffin *intmp)
{
    intmp->num3 += in.ModDiff2 * in.TempWt;
    intmp->denom3 += in.TempWt;

    return (intmp->num3/intmp->denom3);
}
```

Файл: modulation.h

```
#define T100 0.05
#define Tmin 0.008

struct modulationin {
    double Edertmp[BARK];
    double E2tmp[BARK];
    double Etildetmp[BARK];
};

/* Function prototypes */
int modulation(double *, int, struct modulationin *, double *);
/* Prototypes end */
```

Файл: modulation.c

```
#include <stdlib.h>
#include <assert.h>
#include <math.h>
#include <common.h>
#include <modulation.h>

extern int bark;
```

```
extern double *fC;

int
modulation(double *E2, int rate, struct modulationin *in, double *Mod)
{
   int k;
   double T, a;

   for(k=0;k<bark;k++) {
      T = (double)Tmin + (double)(100/fC[k])*(double)(T100- Tmin);
      a = exp((double)-HANN/(2*rate * T));
      in->Edertmp[k] = in->Edertmp[k]*a+(1-a)*(double)(rate/(HANN/2))
                  *module(p(E2[k], 0.3) - p(in->E2tmp[k], 0.3));
      in->E2tmp[k] = E2[k];
      in->Etildetmp[k] = a*in->Etildetmp[k]+(1-a)*p(E2[k], 0.3);
      Mod[k] = in->Edertmp[k]/(1 + (in->Etildetmp[k]/0.3));
   }

   return 0;
}
```

Файл: neural.h

```
#define sig(x) (double)(1.0/(1.0 + exp((double)-(x))))

#define I 11
#define J 3

struct outframes {
   double WinModDiff1b;
   double AvgModDiff1b;
   double AvgModDiff2b;
   double RmsNoiseLoudb;
   double BandwidthRefb;
   double BandwidthTestb;
   double TotalNMRb;
   double RelDistFramesb;
   double ADBb;
   double MFPDb;
   double EHSb;
};

struct out {
   double ODG;
   double DI;
};

/* Function prototypes */
struct out neural(struct outframes processed);
/* Prototypes end */
```

Файл: neural.c

```
#include <stdlib.h>
#include <math.h>
#include <common.h>
#include <neural.h>

double amin[11] = {393.916656, 361.965332, -24.045116, 1.110661, -0.206623,
                   0.074318, 1.113683, 0.950345, 0.029985, 0.000101, 0.0};
double amax[11] = {921.0, 881.131226, 16.212030, 107.137772, 2.886017,
                   13.933351, 63.257874, 1145.018555, 14.819740, 1.0, 1.0};
double wx[12][3] = {{-0.502657, 0.436333, 1.219602},
                    {4.307481, 3.246017, 1.123743},
                    {4.984241, -2.211189, -0.192096},
                    {0.051056, -1.762424, 4.331315},
```

```
                {2.321580, 1.789971, -0.754560},
                {-5.303901, -3.452257, -10.814982},
                {2.730991, -6.111805, 1.519223},
                {0.624950, -1.331523, -5.955151},
                {3.102889, 0.871260, -5.922878},
                {-1.051468, -0.939882, -0.142913},
                {-1.804679, -0.503610, -0.620456},
                {-2.518254, 0.654841, -2.207228}};
double wy[4] = {-3.817048, 4.107138, 4.629582, -0.307594};
double bmin = -3.98;
double bmax = 0.22;

struct out
neural(struct outframes processed)
{
   int j, i;
   struct out oveRet;
   double sum1, sum2;
   double x[11];

   x[0] = processed.BandwidthRefb;
   x[1] = processed.BandwidthTestb;
   x[2] = processed.TotalNMRb;
   x[3] = processed.WinModDiff1b;
   x[4] = processed.ADBb;
   x[5] = processed.EHSb;
   x[6] = processed.AvgModDiff1b;
   x[7] = processed.AvgModDiff2b;
   x[8] = processed.RmsNoiseLoudb;
   x[9] = processed.MFPDb;
   x[10] = processed.RelDistFramesb;

      // gcodcla.wav
      /*x[0] = 834.117;
      x[1] = 647.095;
      x[2] = -14.6048;
      x[3] = 6.89483;
      x[4] = 0.432969;
      x[5] = 0.503605;
      x[6] = 7.14863;
      x[7] = 24.9353;
      x[8] = 0.124738;
      x[9] = 0.968876;
      x[10] = 0.0485208;  // сосми tutto ok*/

      // ccodsax.wav
      /*x[0] = 853.375;
      x[1] = 645.444;
      x[2] = -7.94882;
      x[3] = 11.4108;
      x[4] = 1.41971;
      x[5] = 0.491164;
      x[6] = 12.6383;
      x[7] = 44.7187;
      x[8] = 0.21807;
      x[9] = 1.15;//0.675505;
      x[10] = 0.556215;*/

   sum2 = 0;
   for(j=0;j<J;j++) {
      sum1 = 0;
      for(i=0;i<I;i++)
         sum1 += wx[i][j]*((x[i] - amin[i])/(amax[i] - amin[i]));
         sum2 += wy[j]*sig(wx[I][j] + sum1);
   }
```

```
    oveRet.DI = wy[J] + sum2;
    oveRet.ODG = bmin + (bmax - bmin)*sig(oveRet.DI);

    return oveRet;
}
```

Файл: nmr.h

```
/* Function prototypes */
double nmr(double *, double *, double *, int);
/* Prototypes end */
```

Файл: nmr.c

```
#include <stdlib.h>
#include <math.h>
#include <common.h>
#include <nmr.h>

extern int bark;

double
nmr(double *Pnoise, double *M, double *nmrtmp, int n)
{
    int k;
    double sum = 0;

    for(k=0;k<bark;k++)
        sum += Pnoise[k]/M[k];

    sum *= (double)1.0/bark;
    *nmrtmp += sum;

    return (10.0*log10((*nmrtmp)/n));
}
```

Файл: noiseloudness.h

```
#define THRESFAC0 0.15
#define S0 0.5
#define E0 1.0
#define ALPHA 1.5

/* Function prototypes */
double noiseloudness(double *, double *, struct levpatadaptout,
                     double *, int);
/* Prototypes end */
```

Файл: noiseloudness.c

```
#include <stdlib.h>
#include <math.h>
#include <common.h>
#include <levpatadapt.h>
#include <noiseloudness.h>

extern int bark;
extern double *fC;

double
noiseloudness(double *Modtest, double *Modref, struct levpatadaptout lev,
              double *nltmp, int n)
{
    int k;
    double Pthres, stest, sref, beta, num, denom;
    double nl = 0;
```

```
    for(k=0;k<bark;k++) {
        Pthres = p(10.0, 0.4*0.364*p(fC[k]/1000.0, -0.8));
        stest = (double)THRESFAC0*Modtest[k] + S0;
        sref = (double)THRESFAC0*Modref[k] + S0;
        if(lev.Eptest[k] == 0 && lev.Epref[k] == 0)
            beta = 1.0;
        else
        if(lev.Epref[k] == 0)
            beta = 0;
        else
            beta = exp((double)-ALPHA*(lev.Eptest[k] - lev.Epref[k])
                /lev.Epref[k]);

                num = stest*lev.Eptest[k] - sref*lev.Epref[k];
        denom = Pthres + sref*lev.Epref[k]*beta;
        if(num < 0)
            num = 0;

        nl += p(Pthres/(E0*stest), 0.23)*(p(1.0 + num/denom, 0.23) - 1.0);
    }
    nl *= (double)24.0/bark;
    if(nl < 0)
    nl = 0;

    *nltmp += p(nl, 2.0);

    return sqrt((double)*nltmp/n);
}
```

Файл: reldistframes.h

```
/* Function prototypes */
double reldistframes(double *, double *, double *, int);
/* Prototypes end */
```

Файл: reldistframes.c

```
#include <stdlib.h>
#include <math.h>
#include <common.h>
#include <reldistframes.h>

extern int bark;

double
reldistframes(double *Pnoise, double *M, double *reldisttmp, int n)
{
    int k;

    for(k=0;k<bark;k++) {
        if(10.0*log10((double)Pnoise[k]/M[k]) >= 1.5) {
            *reldisttmp = *reldisttmp  + 1;
            break;
        }
    }

    return ((double)*reldisttmp/n);
}
```

Файл: spreading.h

```
#ifdef ADVANCED
  #define dz 0.5
#else
  #define dz 0.25
#endif
```

```
/* Function prototypes */
int spreading(double *, double *);
/* Prototypes end */
```

Файл: spreading.c

```
#include <stdlib.h>
#include <math.h>
#include <common.h>
#include <spreading.h>

extern double *fC;
extern int bark;

int
spreading(double *pp, double *e2)
{
   int k, j, u;
   double L;
   double denom, sum1, sum2, Eline, Su, Sl = 27.0;

   for(k=0;k<bark;k++) {
      sum1 = 0;
      sum2 = 0;
      // Eline
      for(j=0;j<bark;j++) {
         L = 10.0*log10((double)pp[j]);
         Su = -24.0-230.0/fC[j]+0.2*L;
         Eline = p(10.0, L/10.0);
         if(k < j)
            Eline *= p(10.0, -dz*(j-k)*Sl/10.0);
         else
            Eline *= p(10.0, dz*(k-j)*Su/10.0);

               denom = 0;
         for(u=0;u<j;u++)
            denom += p(10.0, -dz*(j-u)*Sl/10.0);
         for(u=j;u<bark;u++)
            denom += p(10.0, dz*(u-j)*Su/10.0);

         Eline /= denom;
         sum1 += p(Eline, 0.4);
      }

      // Eline (tilde)
      for(j=0;j<bark;j++) {
         Su = -24.0-230.0/fC[j]; // L = 0
         if(k < j)
            Eline = p(10.0, -dz*(j-k)*Sl/10.0);
         else
            Eline = p(10.0, dz*(k-j)*Su/10.0);

         denom = 0;
         for(u=0;u<j;u++)
            denom += p(10.0, -dz*(j-u)*Sl/10.0);
         for(u=j;u<bark;u++)
            denom += p(10.0, dz*(u-j)*(-24.0-230.0/fC[j])/10.0);

         Eline /= denom;
         sum2 += p(Eline, 0.4);
      }

      sum2 = p(sum2, 1.0/0.4);
      sum1 = p(sum1, 1.0/0.4);

      e2[k] = sum1/sum2;
   }

   return 0;
}
```

Файл: threshold.h

```
#ifdef ADVANCED
  #define dz 0.5
#else
  #define dz 0.25
#endif

/* Function prototypes */
int threshold(double *, double *);
/* Prototypes end */
```

Файл: threshold.c

```
#include <stdlib.h>
#include <math.h>
#include <common.h>
#include <threshold.h>

extern int bark;

int
threshold(double *E, double *M)
{
    int k;
    double m;

    for(k=0;k<bark;k++) {
        if((m = k * dz) <= 12)
            m = 3.0;
        else
            m *= 0.25;

        M[k] = E[k]/p(10, m/10);
    }

    return 0;
}
```

Файл: timespreading.h

```
/* Function prototypes */
int timespreading(double *, double *, int, double *);
/* Prototypes end */
```

Файл: timespreading.c

```
#include <stdlib.h>
#include <math.h>
#include <common.h>
#include <timespreading.h>

#define T100 0.03
#define Tmin 0.008

extern int bark;
extern double *fC;

int
timespreading(double *E2, double *Etmp, int rate, double *E)
{
    int k;
    double a, T;

    for(k=0;k<bark;k++) {
        T = (double)Tmin + (double)(100.0/fC[k])*(double)(T100- Tmin);
        a = exp((double)-HANN/(double)(T*2.0*rate));
        Etmp[k] = Etmp[k]*a + (1.0-a)*E2[k];
```

```
        if(Etmp[k] >= E2[k])
            E[k] = Etmp[k];
        else
            E[k] = E2[k];
    }
    return 0;
}
```

Файл: wavedump.h

```
/* Function prototypes */
int HeaderDump(FILE *, const char *);
int ReadInt(FILE *, int);
int SampleRate(FILE *);
int BitForSample(FILE *);
int NumOfChan(FILE *);
int FindData(FILE *);
void fatalerr(char *,...);
/* Prototypes end */
```

Файл: wavedump.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <wavedump.h>

extern int errno;

int
HeaderDump(FILE *fp, const char *string)
{
    char buff[7];
    int k, offset = 0;

    while (!feof(fp)) {
        if(!fread(buff, 7, 1, fp))
            fatalerr("err: error in WaveHeader");
        offset += 7;
        for(k=0;k<4;k++)
            if (!strncmp((char *)(buff + k), string, 4)) {
                fseek(fp, k-3, SEEK_CUR);
                offset += k-3;
                return offset;
            }
        fseek(fp, -3, SEEK_CUR);
        offset += -3;
    }

    return -1;
}

int
SampleRate(FILE *fp)
{
    int rate, offset;

    if ((offset = HeaderDump(fp,"fmt ")) == -1)
        return -1;
    if (fseek(fp, 8, SEEK_CUR) == -1)
        fatalerr("err: %s", strerror(errno));

    rate = ReadInt(fp,4);
    fseek(fp, -8 - offset, SEEK_CUR);

    return rate;
}
```

```
int
BitForSample(FILE *fp)
{
    int bit, offset;

    if ((offset = HeaderDump(fp,"fmt ")) == -1)
        return -1;
    if (fseek(fp, 18, SEEK_CUR) == -1)
        fatalerr("err: %s", strerror(errno));

    bit = ReadInt(fp,2);
    fseek(fp, -18 - offset, SEEK_CUR);

    return bit;
}

int
NumOfChan(FILE *fp)
{
    int chan, offset;

    if ((offset = HeaderDump(fp,"fmt ")) == -1)
        return -1;
    if (fseek(fp, 6, SEEK_CUR) == -1)
        fatalerr("err: %s", strerror(errno));

    chan = ReadInt(fp,2);
    fseek(fp, -6 - offset, SEEK_CUR);

    return chan;
}

int
FindData(FILE *fp)
{
    int offset;

    if (fp == NULL)
        return -1;

    if ((offset = HeaderDump(fp,"data")) == -1)
        return -1;
    if (fseek(fp, 4 + offset, SEEK_CUR) == -1)
        fatalerr("err: %s", strerror(errno));

    return 1;
}

#if defined(LITTLE) && !defined(BIG)
int
ReadInt(FILE *fp, int size)
{
    int I;
    unsigned char c;

    if (size <= 0)
        return 0;

    c = fgetc(fp);
    I = ((int) c) & 255;
    I |= (ReadInt(fp, size-1) << 8);

    return I;
}
#elif defined(BIG) && !defined(LITTLE)
int
ReadInt(FILE *fp, int size)
{
    int I;
```

```
    if (size <= 0)
        return 0;

    I = (ReadInt(fp, size-1) << 8);
    I |= ((int) fgetc(fp)) & 255;

return I;
}
#endif
```

Б.3 Листинг программы расчета метрики PSNR на языке Matlab

```
function PSNR = calcPSNR(X,Y)
% чем больше значение PNSR, тем более похожи сигналы
    m = max(X);
    d = var(X-Y);
    R = m/sqrt(d);
    PSNR = 20* log10(R);
end
```

Б.4 Листинг программы расчета метрики PSNR на языке C

```
#include <math.h>

double var(double* arr, int size)
{
    if(!arr || !size)
        return 0.0;

    double v = 0.0;
    double avg = 0.0;

    for(int i = 0; i < size; ++i)
        avg+= arr[i];

    avg/= size;

    for(int i = 0; i < size; ++i)
    {
        v+= (arr[i] - avg) * (arr[i] - avg);
    }

    return v / (size - 1);
}

double calcPSNR(double* X, int sizeX, double* Y, int sizeY )
{
    if(!X || !sizeX || !Y || !sizeY || (sizeX != sizeY))
        return 0.0;

    double D[sizeX];
    double maxX = X[0];
    for(int i = 0; i < sizeX; ++i)
    {
        if(X[i] > maxX)
            maxX = X[i];

        D[i] = X[i] - Y[i];
    }

    double v = var(D, sizeX);

    if(v == 0)
        return 0.0;

    return 20 * log10(maxX / sqrt(v));

}
```

Б.5 Листинг программы расчета метрики «Коэффициент различия форм сигналов» на языке Matlab

```
function val = calcMeasureBasedOnSingalsForms(X,Y)
% чем меньше результирующее значение, тем более похожи сигналы
```

```
    X= X(:,1);
    Y= Y(:,1);
    dX=X(2:end) - X(1:end-1);
    dY=Y(2:end) - Y(1:end-1);

val = var(dX-dY);

end
```

Б.6 Листинг программы расчета метрики «Коэффициент различия форм сигналов» на языке *C*

```
#include <math.h>

double var(double* arr, int size)
{
    if(!arr || !size)
        return 0.0;

    double v = 0.0;
    double avg = 0.0;

    for(int i = 0; i < size; ++i)
        avg+= arr[i];

    avg/= size;

    for(int i = 0; i < size; ++i)
    {
        v+= (arr[i] - avg) * (arr[i] - avg);
    }

    return v / (size - 1);
}

double calcMeasureBasedOnSingalsForms(double* X, int sizeX, double* Y, int sizeY )
{
    if(!X || !sizeX || !Y || !sizeY || (sizeX != sizeY))
        return 0.0;

    double dX[sizeX - 1];
    double dY[sizeX - 1];
    double D[sizeX - 1];

    for(int i = 1; i < sizeX; ++i)
    {
        dX[i - 1] = X[i] - X[i - 1];
        dY[i - 1] = Y[i] - Y[i - 1];
        D[i - 1]  = dX[i - 1] - dY[i - 1];
    }

    return var(D, sizeX - 1);
}
```

УДК 621.398:006.354 ОКС 13.320 П77 ОКП 43 7200

Ключевые слова: системы охранные телевизионные, аудиоданные, аудио-компрессия, аудиокодер, аудиодекодер, кодек аудиоданных

Редактор *Н.А. Аргунова*
Технический редактор *В.Н. Прусакова*
Корректор *В.И. Варенцова*
Компьютерная верстка *И.А. Налейкиной, Л.А. Круговой*

Сдано в набор 19.03.2015. Подписано в печать 13.10.2015. Формат 60×84⅛. Гарнитура Ариал.
Усл. печ. л. 10.70. Уч.-изд. л. 7.50. Тираж 32 экз. Зак. 3277.

Издано и отпечатано во ФГУП «СТАНДАРТИНФОРМ», 123995 Москва, Гранатный пер., 4.
gost.e-ecolog.ru gost.e-ecolog.ru

gost.e-ecolog.ru