Койнов Стас

Быстрое преобразование Фурье
(реализация на языке Free Pascal)

Энгельс 16.08.2011

От автора

Хочу выразить свою благодарность всему коллективу сайта http://www.freepascal.ru/ всем, кто помогал мне разбираться в различного рода вопросах: «спасибо вам ребята!». Особенно хочется выделить Сергея Горелкина, который выступал в роли моего заочного учителя, за что ему: «Большое спасибо!».

Если вы заметили ошибку, неточность в тексте или исходниках напишите мне, пожалуйста на E-Mail: MaxizarSoft@yandex.ru

Лицензия

Соглашения принятые в работе:

[ ] - в квадратных скобках мы будем указывать номер(а), которым соответствует номер(а) работ из списка дополнительной литературы.

Нумерация таблиц, рисунков и листингов будет начинаться сначала в каждом пункте работы. Пример: **Рис. 2.4.5.1** - рисунок с номером 1, в пункте 2.4.5, данный подход позволяет достаточно быстро вносить изменения, однако в ущерб удобству редактирования документа, нумерация формул сквозная. Это более удобно при чтении работы, особенно если вы решите провести вывод формул самостоятельно.

Содержание

Введение

Одним из мощных инструментом обработки данных, непрерывных функций f(x) или некоторого набора дискретных данных $f(x_i)$, является спектральный анализ, имеющий в своей основе различные интегральные преобразования. Спектральный анализ используется в различных целях, таких как: подавление шума, фильтрация, построения амплитудно частотной характеристики (АЧХ), так и для решения других проблем обработки данных.

Спектром функции f(x) или совокупности данных $f(x_i)$ - называют некоторую функцию другой координаты (или координат) F(w), полученную в соответствии с определенным алгоритмом. Примерами спектров является преобразование Фурье и вейвлет-преобразование.

Преобразование Фурье имеет огромное значение для различных математических приложений и широко применяется в науке и технике. Очень часто приходится иметь дело с дискретным набором данных $f(x_i)$, вследствие чего был разработан так называемый спектральный анализ при помощи дискретного преобразования Фурье (ДПФ). Но в большинстве случаев скорость расчета ДПФ, оставляет желать лучшего, и для него был разработан очень эффективный алгоритм, называемый БПФ (быстрое преобразование Фурье). На самом деле алгоритмов БПФ достаточно много, мы в данной работе рассмотрим лишь один из них, а именно БПФ по основанию два с прореживанием по времени.

Основными задачами данной работы являются:

1. Рассмотреть такие понятия как:
   - Преобразования Фурье
   - Дискретное преобразование Фурье
   - Быстрое преобразование Фурье
2. Реализовать на языке Free Pascal алгоритм ДПФ.
3. Рассмотреть алгоритм БПФ по основанию 2 с прореживанием по времени (вывод БПФ из ДПФ).
4. Реализовать на языке Free Pascal алгоритм БПФ по основанию 2 с прореживанием по времени.
5. Провести оптимизацию при помощи встроенного ассемблера языка Free Pascal.
6. Разработать программу для построения спектра звуковых файлов в формате WAV PCM Mono 16 bit per sample.

1 Преобразование Фурье

В 1807 французский математик и физик Жан Батист Жозеф Фурье представил во Французский Институт (Institut de France) доклад о синусоидальном представлении температурных распределений. Доклад содержал спорное утверждение о том, что любой непрерывный периодический сигнал может быть представлен суммой выбранных должным образом сигналов синусоидальной формы. Среди членов комитета, занимавшихся обзором публикаций, были два известных математика – Жозеф Луи Лагранж и Пьер Симон де Лаплас. Лагранж категорически возразил против публикации на основании того, что подход Фурье неприменим к разрывным функциям, таким как сигналы прямоугольной формы. Работа Фурье была отклонена, прежде всего из-за возражения Лагранжа, и была издана после смерти Лагранжа, приблизительно пятнадцатью годами позже.

Преобразование Фурье — это интегральное преобразование, которое раскладывает исходную функцию на базисные функции, в качестве которых выступают синусоидальные функции, то есть представляет исходную функцию в виде интеграла синусоид различной частоты, амплитуды и фазы.

Общий вид преобразования Фурье для непрерывной функции **f(t)**, имеет вид (1).

$$\varphi(s)=\int_{-\infty}^{+\infty} f(t)\cdot \mathrm{e}^{-i\cdot s\cdot t}\cdot dt \qquad \textbf{1.}$$

Где функция **φ(s)** и есть преобразование Фурье, для функции **f(t).** Обратное преобразование Фурье (ОПФ) имеет вид (2).

$$f(t)=\frac{1}{2\cdot\pi}\int_{-\infty}^{+\infty} \varphi(s)\cdot \mathrm{e}^{i\cdot s\cdot t}\cdot ds \qquad \textbf{2.}$$

Недостаток преобразования Фурье в том, что он «работает» только с непрерывной функцией, что не всегда возможно и целесообразно. В большинстве случаев, мы имеем дело лишь с дискретной функцией (данными) или вообще ограниченным набором данных, для которых мы не можем применить преобразование Фурье, для этого мы должны провести интерполяцию данных. Однако для дискретного набора данных, было придумано так называемое *дискретное преобразование Фурье* (ДПФ).

2 Дискретное преобразование Фурье

Дискретное преобразование Фурье — это одно из преобразований Фурье, широко применяемых в алгоритмах цифровой обработки сигналов (его модификации применяются в сжатии звука в MP3, сжатии изображений в JPEG и др.), а также в других областях, связанных с анализом частот в дискретном (к примеру, оцифрованном аналоговом) сигнале. Дискретное преобразование Фурье требует в качестве входа дискретную функцию. Такие функции часто создаются путём дискретизации (выборки значений из непрерывных функций). Дискретные преобразования Фурье помогают решать частные дифференциальные уравнения и выполнять такие операции, как свёртки. Дискретные преобразования Фурье также активно используются в статистике, при анализе временных рядов. Преобразования бывают одномерные, двумерные и даже трёхмерные [8].

Прежде чем рассмотреть ДПФ, нам необходимо понять смысл дискретизации аналогового сигнала.

2.1 Дискретизации аналогового сигнала

Дискретизация — преобразование непрерывной функции в дискретную.

Давайте рассмотрим звуковой сигнал синусоидального вида. Пусть частота данного сигнала V = 5 000 Гц. Чтобы представить этот сигнал на графике, введем функцию f(t), которая будет нашим аналоговым сигналом (звуком), данная функция будет иметь вид (3)

$$f(t)=\sin\left(\frac{2\cdot\pi\cdot t}{T}\right) \quad \textbf{3.}$$

Где T=1/V — период колебаний, V — частота сигнала.
Мы выберем интервал времени t=0..0.0008 сек., что позволит увидеть 4-е полных колебания *(Рис. 2.1.1).*

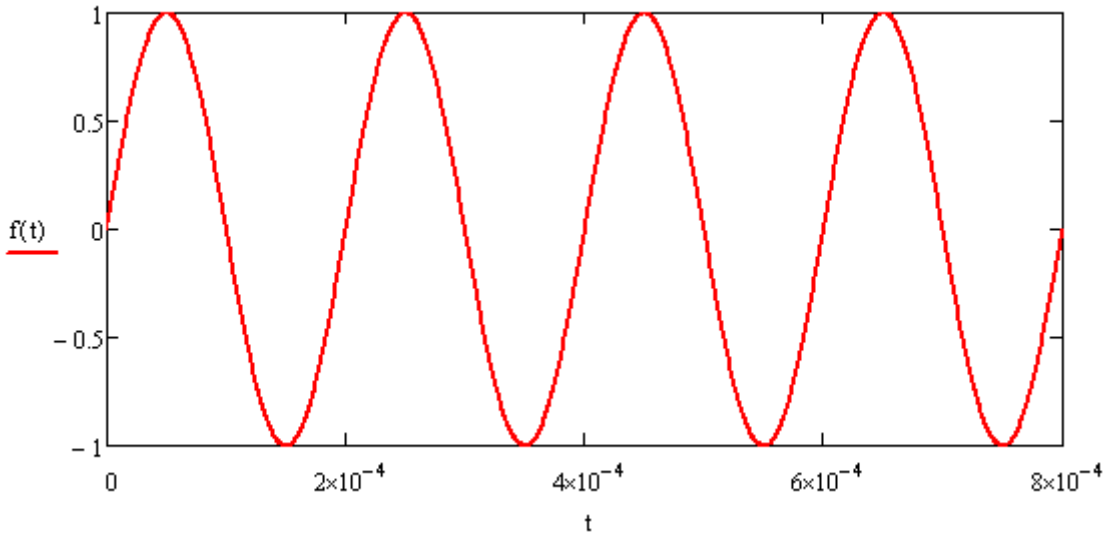


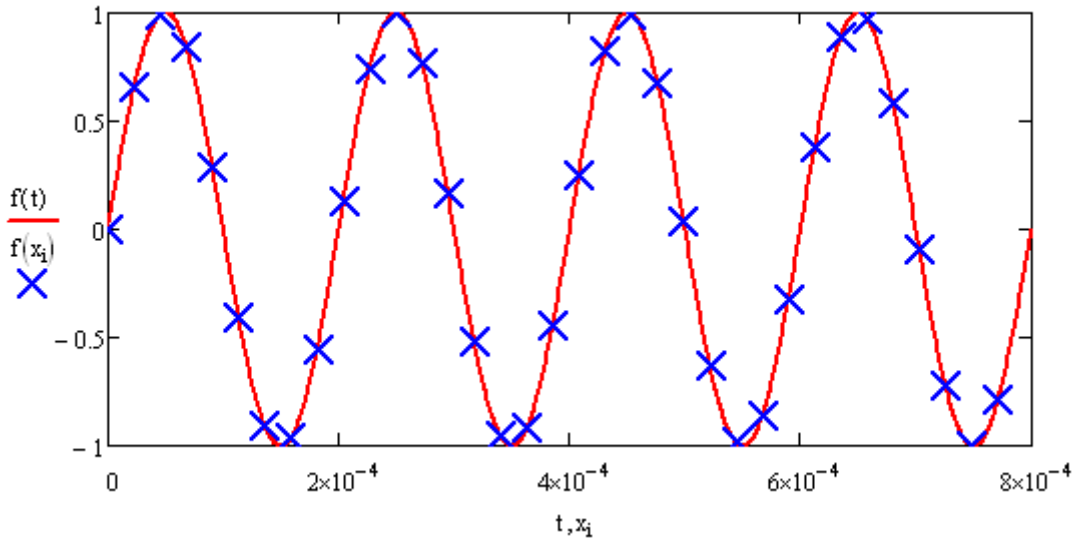
Рис. 2.1.1. Графическое представление функции f(t)

Теперь проведем дискретизацию данного сигнала, чтобы это сделать необходимо выбрать, так называемую, частоту дискретизации **Vd.**

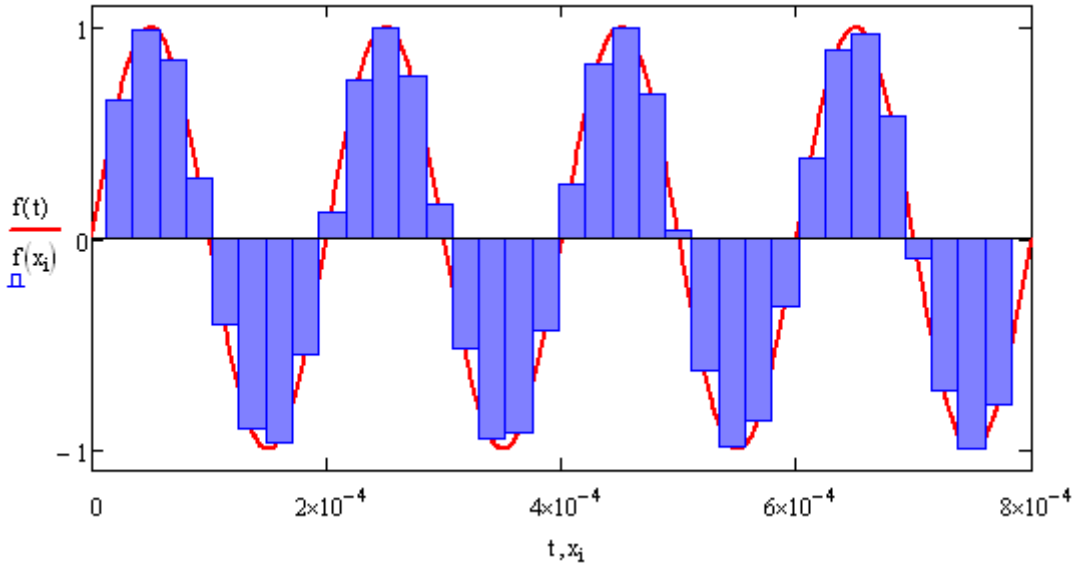
Частота дискретизации (или **частота сэмплирования**) — частота взятия отсчетов (сохранения) непрерывного во времени сигнала при его дискретизации (в частности, аналого-цифровым преобразователем). Измеряется в герцах.

Мы ничего придумывать не будем, а возьмем стандартную частоту дискретизации принятую при записи музыки в компакт дисках, а именно Vd = 44100Гц.

Чтобы понять, что мы будем делать, необходимо рассчитать период дискретизации Td. Td = 1/Vd. - это будет время, через которое мы будем брать отсчеты функции f(t). То есть мы будем превращать непрерывную функцию f(t) в дискретную $f(x_i)$, где x_i = Td*i., i=0..Max_i *(Рис. 2.1.2)*



а



б

Рис. 2.1.2. Графическое представление дискретизации непрерывного сигнала f(t).

На *Рис. 2.1.2а* достаточно хорошо видно, какие значение мы берем при дискретизации, тогда как на *Рис. 2.1.2б* видно с каким шагом мы берем отсчеты.

Переменная **i** может принимать значение от 0 (нуля) до Max_i, каким образом определяется величина Max_i?. Ведь сигнал наш имеет определенное время звучания, а так как мы знаем частоту дискретизации Vd, мы знаем сколько отсчетов мы должны взять в одной секунде звука, их ровно Vd, то есть в нашем случае 44100 отсчетов. Если же наш звук длится 5 секунд, то Max_i = Vd*5. Или в общем виде при длительности звука выраженного в секундах и равного TimeSound, **Max_i = Vd* TimeSound**. Это необходимо очень четко себе представлять!

Мы рассмотрели почти все аспекты дискретизации, которые нам необходимы для понимания ДПФ, кроме одной сущности, как частота дискретизации Vd. Мы просто сказали, что при записи компакт дисков Vd = 44100 Гц. Но почему это так, мы

умолчали.

Для того чтобы понять какую частоту дискретизации выбрать, нужно обратится к **теореме Найквиста**, либо **теореме Котельникова**, которая гласит, что если аналоговый сигнал имеет ограниченный спектр (есть предельная частота), то он может быть восстановлен однозначно и без потерь по своим дискретным отсчётам, взятым с частотой ***строго большей*** *удвоенной максимальной частоты спектра*.

После того как дискретизация проведена, мы получаем набор данных (дискретную функцию) с которым как раз и работает ДПФ.

2.2 Свойства ДПФ

Если считать, что сигнал **f(n)** у нас уже представлен в дискретном виде. И при этом сигнал имеет N дискретных значений и **f(n) -** это n-ое значение, т.е. n=0..N-1. Тогда для k-го значения данного сигнала, дискретное преобразование Фурье, будет иметь вид (4).

$$F(k)=\sum_{n=0}^{N-1}\left(f(n)\cdot\exp(-i\frac{2\cdot\pi}{N}\cdot n\cdot k)\right),\qquad k=0..\mathrm{N}-1; \tag{4.}$$

А обратное дискретное преобразование Фурье (ОДПФ) будет иметь вид (5).

$$f(n)=\frac{1}{N}\cdot\sum_{k=0}^{N-1}\left(F(k)\cdot\exp(i\frac{2\cdot\pi}{N}\cdot n\cdot k)\right),\qquad n=0..\mathrm{N}-1; \tag{5.}$$

если ввести обозначение:

$$W_N^{n\cdot k}=\exp(-i\frac{2\cdot\pi}{N}\cdot n\cdot k) \tag{6.}$$

тогда выражение для ДПФ и ОДПФ примут вид (7), (8) соответственно.

$$F(k)=\sum_{n=0}^{N-1}\left(f(n)\cdot W_N^{n\cdot k}\right),\qquad k=0..\mathrm{N}-1; \tag{7.}$$

$$f(n)=\frac{1}{N}\cdot\sum_{k=0}^{N-1}\left(F(k)\cdot W_N^{-n\cdot k}\right),\qquad n=0..\mathrm{N}-1\ ; \tag{8.}$$

Это конечные вид функций, именно с ними мы будем проводить ту или иную работу. Теперь давайте разберем, непосредственно сам процесс преобразования. Возьмем наш звуковой сигнал, который прошел дискретизацию, чтобы рассчитать для него ДПФ, нам необходимо будет выбрать, для какой длинны выборки N мы будем это делать.

В литературе применяются различные названия *длины выборки*, такие как *длина блока* или просто *блок* (*окно*), ну или просто *размер ДПФ*.

Вообще говоря, перевод сигнала в частотное представление возможен только блоками (окнами). Мы делим исходный дискретный сигнал на блоки и проведя ДПФ говорим: «в этом блоке имеются такие-то частоты с такими-то амплитудами». Вернее, так: «его можно получить, сложив такие-то частоты с такими-то амплитудами при помощи ОДПФ».

Длина выборки ДПФ N - говорит нам, сколько частот мы сможем рассчитать для блока. А именно число частот равняется **N/2**. То есть если мы выберем **N = 256**, то из всех частот, которые присутствуют в данном блоке сигнала, мы сможем рассчитать только **128**. Но какие это частоты? **Это один из самых главных вопросов всей работы.**

Рассчитывая ДПФ для определенной *выборки N* (*окна длинной N* или просто *блока N*), мы рассчитываем область частот, от «нулевой» до максимальной **Vmax** с равным шагом Δ**V**. Шаг определяется как:

$$\Delta V=\frac{Vd}{N} \tag{9.}$$

Следовательно, чем больше **N**, тем меньше шаг определяемых частот, то есть спектр получается **более узкополосным**. То есть **N** определяет так называемое *разрешение по частоте*.

Значения **Vmax**, тесно связаны с частотой дискретизации данных, а именно:

$$Vmax=\frac{Vd}{2} \tag{10.}$$

Давайте разберемся на «живом» примере, пусть у нас имеется компакт диск (CD) со звуковой дорожкой. Стандартные CD записывают звук с частотой

дискретизации Vd=44100 Гц., следовательно, максимальная частота, которая может быть в звуковой дорожке **Vmax** = 22 050Гц (вспоминаем теорему Найквиста). Если в звуковой дорожке имелись более высокие частоты, то они не записываются на CD при дискретизации, вернее так, звуковая карта не сможет восстановить данные частоты при интерполяции дискретных данных в своем ЦАП. Максимальная частота, которую звуковая карта сможет воссоздать это как раз та самая Vmax и то при условии, что у нас хорошая звуковая карта.

Скажем моя встроенная звуковая карта «плывет» на частотах 17.5КГц., тогда как «хорошая» Asus Xonar D1 должна «тянуть» такие частоты, но это при наличии качественной акустики, которой у меня к сожалению нет :(.

Почему CD используют частоту 44100? на этот вопрос вам поможет ответить Интернет, считается, что этого достаточно для идеальной передачи звука. Меломаны с этим не согласны....

Пусть в нашем случае (**N = 4096, Vd=44100**) **Vmax** = 22 050Гц. Значит шаг расчета частот Δ**V =** 10.767Гц.

Проведя наш расчет, для N=4096, мы рассчитаем N/2=2048 частот, а именно:
V1=ΔV*1 = 10.767Гц; ……. V2048=ΔV*2048=Vmax= 22 050Гц.

Некоторые авторы утверждают, что мы рассчитываем с частоты равной нулю и далее с шагом Δ**V,** на самом деле минимальная частота которую мы можем рассчитать, равна как раз Δ**V** и далее с шагом Δ**V** вплоть до **Vmax**. Но при этом не нужно думать, что частоты не попавшие в расчетные, мы не сможем увидеть, сможем, но не очень хорошо, они будут «размазываться» между соседними расчетными частотами, эта проблема «точности» ПФ и как следствие ДПФ и БПФ рассматривается в работе [1].

В Таблице 2.2.1 приведена зависимость Δ**V** и количества рассчитанных частот от длины окна N.

N	128	256	512	1024	2048	4096	8192
Количество рассчитанных частот	64	128	256	512	1024	2048	4096
Δ**V (Гц)**	344,53	172,27	86,13	43,07	21,53	10,77	5,38

Таблица 2.2.1 Значение количества рассчитанных частот и шага по частоте, при различных значениях N, для звукового сигнала с частотой дискретизации Vd=44100Гц.

Еще разок про окна ДПФ\БПФ. Пусть мы имеем звук длинной 1 сек. Этот звук уже представлен в дискретном виде и пусть частота дискретизации равна **Vd=44100.** Это значит, что мы имеем ровно 44100 значений амплитуды нашего звука (ведь у нас звук длится всего 1 сек). Мы можем посчитать спектр только для окна, окно — это данные длинной N. По этому число N мы можем взять любым вплоть до 44100. Вопрос, каким же взять размер окна? Это зависит от того, что мы хотим получить динамику изменения частоты либо точность в определении частоты.

Если необходима информация о наличие или отсутствии какой-либо частоты, то чем больше размер блока, тем выше разрешение по частоте см. таблицу 2.2.1

Если хотим получить динамику: то есть мы хотим увидеть сколько раз за секунду в нашем звуке появляется какая-либо частота Vx. То мы должны взять N – как можно меньшим, но при этом понимать есть ли хоть одна из рассчитанных частот Vi (i=0..N-1) близкая к Vx, если есть, то все хорошо, если нет, то размер блока нужно увеличивать, чтоб приблизится к Vx (увеличиваем разрешение по частоте). Но при увеличении размера блока мы уменьшаем разрешение по времени, ведь блок стал длиннее по времени. Скажем пусть N=4096, тогда мы получим 10 целых блоков (44100/4096 = 10.7, то есть каждый блок это 0.1 секунда) рассчитав спектры, мы можем сказать, сколько раз из 10 появляется частота Vx. А что если частота появляется скажем 20 раз? А уменьшить блок мы не можем, так как потеряем разрешение по частоте (мы не будем видеть нужную частоту Vx)... Вот это как раз тот случай, когда говорят: «приплыли». Для решения этой проблемы используют перекрытия блоков [1]. *Рис. 2.2.1*

Рис. 2.2.1 Различная конфигурация расположения блоков, при расчете ДПФ. (для большей наглядности перекрытые блоки смещены по вертикали)

У внимательного читателя на данный момент должен был возникнуть вопрос: «почему в формуле 7, мы говорим, что *k =0..N-1,* а полезных частот мы получим только N/2 ?». Если мы рассчитаем ДПФ для всех k, мы получим массив из N элементов, который будет иметь нужные нам значения (амплитуды) в первой половине [0..N/2-1] массива. Вторая половина [N/2..N-1], является зеркальным отражением первой. Это получается за счет математической симметрии ДПФ, более подробно с этим можно ознакомиться в работах [1,6,8]. Из-за чего рассчитывать вторую половину нет необходимости.

Теперь обратим свое внимание на то, что мы рассчитали! Вот мы имеем наши дискретные данные, рассчитали мы скажем **F(1)** то есть частоту **V1= 10.767Гц.**

Но чему равно **F(1)** да и вообще любая **F(n)?** Вообще-то ДПФ мы рассматриваем в комплексном виде и все **F(n)** — это комплексные числа. Как известно комплексное число состоит из реальной (**Real**) и мнимой (**Imaginary**) части. Следовательно, F(n) будет иметь вид (13).

$$F(n) = Real + i \cdot Imaginary, \quad где \ \ i - мнимая \ еденица \qquad \textbf{13.}$$

Так вот если мы рассчитаем модуль этого комплексного числа, мы получим амплитуду нашей частоты **V1**. То есть мы сможем сказать, что в исходном сигнале частота **V1** имела определенную амплитуду = **|F(1)|**, Где модуль комплексного числа имеет вид (14).

$$|F(n)| = \sqrt{F(n).Real^2 + F(n).Imaginary^2} \qquad \textbf{14.}$$

В большинстве случаев, авторы опускают такой факт, что даже если мы и рассчитаем модуль этого комплексного числа, это не будет амплитуда нашего сигнала, потому что она будет превосходить ее по величине в несколько раз. Например, наш сигнал имеет частоту **V1** и амплитуда этой гармоники пускай будет равна 10. То проведя расчет ДПФ мы удивимся узнав, что амплитуда равна 20 000!. Чтобы привести данную амплитуду к реальной цифре, проводят так называемую нормализацию.

Нормализация заключается в том, что мы каждую амплитуду полученную при помощи ДПФ, должны умножить\разделить на некое число. На какое число, это зависит как от самого сигнала, так и от поставленной задачи. Вот скажем пакет программ MathCAD делит амплитуду на: $\sqrt{N}$ другие делят непосредственно на **N**. Мы с вами будет делить на **N/2**, почему именно на эту величину, не буду скрывать, не помню где я это прочитал, но это позволит почти с точностью 85% получать достоверную амплитуду, первоначального сигнала, при условии, что он имел *одну гармонику* и незначительные шумы.

Я не буду переписывать (копировать) большую часть работы [1], но ее обязательно нужно прочитать от начала до конца!. При этом прочитать, ее нужно до того, как вы приступите к дальнейшему чтению этой работы. Это с учетом описанного выше, позволит вам практически полностью понять возможности ДПФ. А так же сможет приблизить нас к осознанию таких деталей, которые просто необходимо понимать при программировании ДПФ.

Будем считать, что мы стали иметь представление о том, что дает нам ДПФ, какие частоты и амплитуды этих частот мы можем получить. По этому в следующей главе мы перейдем к непосредственному рассмотрению необходимых модулей и типов данных, которые нам понадобятся при программировании ДПФ и БПФ.

3 Реализация алгоритма ДПФ на языке Free Pascal

Для того, чтобы приступить к непосредственному написанию ДПФ на языке Free Pascal, нам необходимо остановится на дополнительных модулях и типах данных, которыми мы будем пользоваться при программировании ДПФ.

3.1 Модуль для работы с комплексными числами на языке Free Pascal

Рассматривать мы будет комплексное ДПФ. По этому нам просто необходимо иметь тип комплексного числа. В свое время я подобный тип описывал в Delphi, при помощи расширенной записи (Extended Record), потому что язык Delphi не может перегружать операторы для простой записи :(. А для расширенной может, но при переходе на язык Free Pascal, я был удивлен, что там нету расширенных записей и подумал, что все - далеко мне не уехать. Но оказалось, все наоборот: Free Pascal, позволяет перегружать операторы для обыкновенной записи, именно так как я этого хотел в Delphi. Поэтому я переписал модуль с Delphi на Free Pascal, и немного расширил его. Я не буду приводить весь листинг модуля, а приведу лишь объявление типа *Листинг 3.1.1.*

Листинг 3.1.1

```
unit complex;

{$H+}
{$mode objfpc}
{$define ComplexIsSingle}


interface

uses
   SysUtils,Math;


  {$ifdef ComplexIsSingle}
    const
      MinComplex  =  1.5e-45;
      MaxComplex  =  3.4e+38;

    Type
      PComplex =^TComplex;
      TComplex = record
         Re,
         Im:Single;
    end;// TComplex = record
```

```
  {$else}
    const
      MinComplex  =  5.0e-324;
      MaxComplex  =  1.7e+308;

    Type
      PComplex =^TComplex;
      TComplex = record
         Re,
         Im:Double;
    end;// TComplex = record
  {$endif}
```

Как можно было заметить, благодаря нашей директиве компилятора `{$define ComplexIsSingle}` мы можем задать два типа таких чисел. Первый, когда директива активна, мы получим комплексное число, у которого мнимая и реальная часть есть числа типа: *Single (4 байта)*, тогда как при не активности директивы (директива за-комментирована) мнимая и реальная часть числа будут иметь тип: *Double (8 байт).* Более детально ознакомится с данным модулем и типом данных, можно в соответствующем исходнике. С расположением исходных файлов и правилами формирования имен каталогов в прилагаемом к работе архиве, можно ознакомится в *приложении 1.*

Так же нам будет необходим массив комплексных чисел, данный тип мы обозначим как *TcmxArray листинг 3.1.2.*

Листинг 3.1.2

```
type
  TCmxArray    = array of TComplex;
```

3.2 Модуль для подсчета времени выполнения кода

Нам будет необходимо каким-то образом подсчитывать сколько выполняется тот или иной участок кода, или сколько по времени выполнялась подпрограмма. Это позволит нам судить, о том на сколько быстро\медленно проходят вычисления или на сколько был успешен рефакторинг или оптимизация кода. Для этого мы будем использовать модуль *PerformanceTime.*

Для измерения времени выполнения участка кода, используются различные подходы, в этом модуле, я постарался реализовать возможность подсчета времени как в системе Windows, так и в системе Linux. (хотя в Linux, я не тестировал после последней модификации). В Windows XP SP3, модуль меня полностью устраивает и он со своей задачей справляется полностью. Так же стоит оговориться, что измерения при помощи данного модуля относительны, то есть нельзя говорить, что на одном ПК код 1 выполнялся X секунда а код 2 на другом ПК X2 секунд. Нельзя сказать что код 2 лучше\хуже чем код 1, так как измерения проходили на разных ПК.

Мы можем утверждать, что код 1 лучше\хуже чем код 2, проведя измерения в рамках одного ПК. Если мы провели такое измерения в рамках одно ПК и получили, что код 1 быстрее кода 2, то мы вправе утверждать, что код 1 будет быстрее кода 2 на другом ПК, с аналогичным процессором. Модуль не очень большой, поэтому приведем его полный исходник, (*Листинг 3.2.1*).

Листинг 3.2.1

```
unit PerformanceTime;
{  ================================================================================
 Модуль PerformanceTime содержит описание класса  TPerformanceTime, который
 позволяет измерить время выполнения куска кода. Необходимо инициализировать
 переменную типа TPerformanceTime, выполнить метод Start. проделать работу
(код)
 Выполнить метод Stop, после чего в св-ве Delay будет время выполнения кода
 в секундах.
 Пример:
     T:=TPerformanceTime.Create;
     T.Start;
     Sleep(1000);
     T.Stop;
     Caption:=FloatToStr(T.Delay);//покажет время равное 1 секунде +/-
погрешность

 Примечание: Позволяет измерять время выполнения кода. Если код "быстрый" можно
 использовать for I:=1 to N do (Код), после чего полученное время разделить
 на N, При этом чем выше N тем меньше будет дисперсия.
 Чем выше частота процессора, то по идее точность должна быть выше, по крайней
 мере в Windows.


 Среда разработки: Lazarus v0.9.29 beta и выше
 Компилятор:       FPC v 2.4.1 и выше
 Автор: Maxizar
 Дата создания: 03.03.2010
 Дата редактирования: 12.05.2011
 }
{$mode objfpc}
{$H+}

interface

uses
  Classes, SysUtils,
  {$IFDEF windows}
  Windows;
  {$ENDIF}
  {$IFDEF UNIX}
  Unix, BaseUnix;
  {$ENDIF}


  Type
    TPerformanceTime=class
      private
        FDelay    :Real;   //измеренное время в секундах
        StartTime :Real;   //Время начала теста в секундах

      public
```

```
        constructor Create;

        property  Delay:Real read FDelay;
        procedure Start;
        procedure Stop;
    end;

  Function  GetTimeInSec:Real;   //вернет время в секундах, с начало работы ОС

implementation


function GetTimeInSec: Real;
var
  {$IFDEF windows}
  StartCount, Freq: Int64;
  {$ENDIF}

   {$IFDEF UNIX}
  TimeLinux:timeval;
  {$ENDIF}
begin
  {$IFDEF windows}
   if QueryPerformanceCounter(StartCount) then //возвращает текущее значение
счетчика
    begin
      QueryPerformanceFrequency(Freq);    //Кол-во тиков в секунду
      Result:=StartCount/Freq;            //Результат в секундах
    end
  else
    Result:=GetTickCount/1000;            //1000, т.к  GetTickCount вернет
милиСекунды
  {$ENDIF}

  {$IFDEF UNIX}
   fpGetTimeOfDay(@TimeLinux,nil);
   Result:=TimeLinux.tv_sec + TimeLinux.tv_usec/1000000;
  {$ENDIF}
end;

{ TPerformanceTime }
//---------------------------------------------------------------------//
constructor TPerformanceTime.Create;
var TempValue:Real;
begin
  TempValue :=GetTimeInSec; //Первый раз холостой, чтобы подгрузить нужные
системные dll
  TempValue :=GetTimeInSec; //Ну на всякий случай :)
end;
//---------------------------------------------------------------------//
procedure TPerformanceTime.Start;
begin
   StartTime:=GetTimeInSec;
end;
//---------------------------------------------------------------------//
procedure TPerformanceTime.Stop;
begin
   FDelay:=GetTimeInSec-StartTime;
end;

end.
```

3.3 Программа тестирования

Для того чтобы продемонстрировать работу нашей реализации ДПФ, нам понадобится программа, показывающая результаты расчета. В ней нет ничего сложного, и она будет представлять собой форму, с расположенным на ней компонентом Image для отображения спектра, двух кнопок и memo для отображения дополнительной информации, скриншот данной программы можно увидеть на *Рис. 3.3.1.*

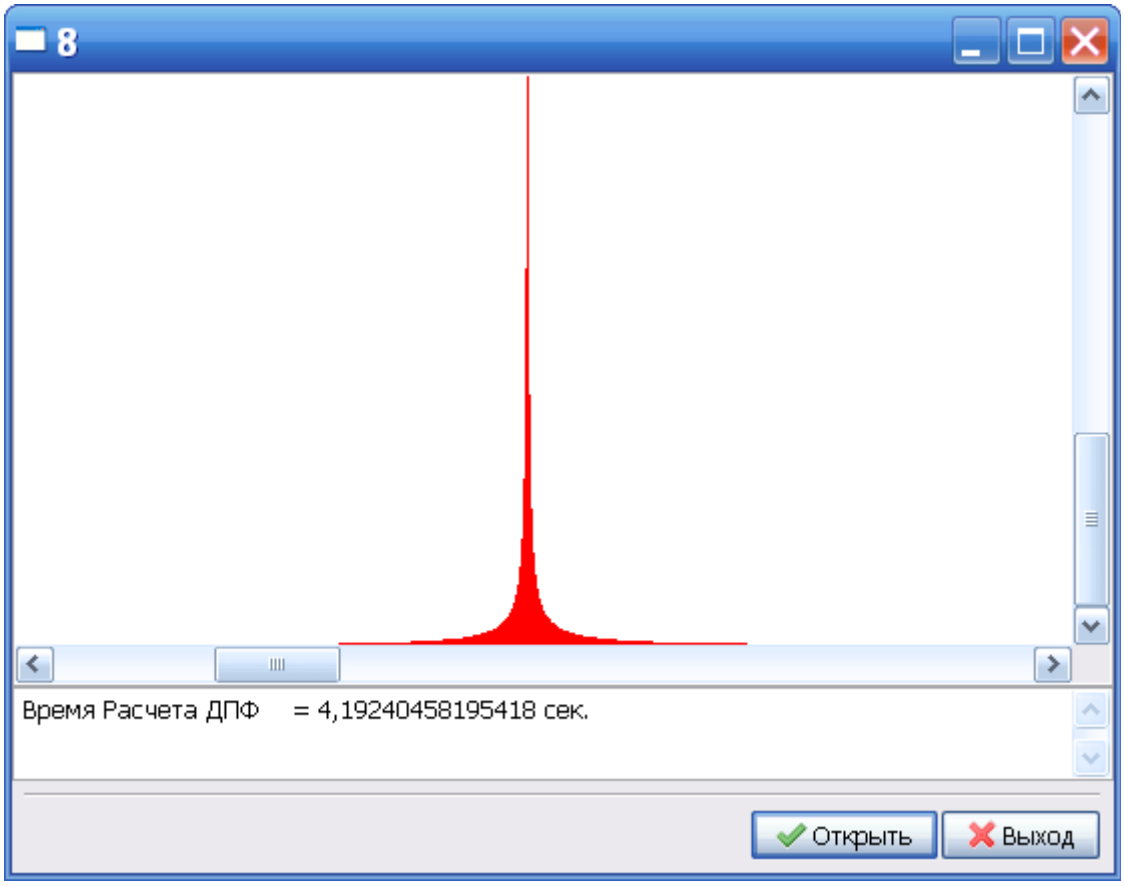


Рис. 3.3.1 Скриншот программы для тестирования нашей реализаций ДПФ

Как можно было заметить, на скриншоте изображена программа в рабочем состоянии с рассчитанным спектром. В поле Memo, мы вывели дополнительную информацию, в данном случае время расчета ДПФ.

Для того чтобы проводить то или иное тестирование, необходимо иметь исходные эталонные данные. В прилагаемом к работе архиве, есть несколько текстовых файлов, которые представляют собой список сэмплов звукового файла определенной частоты и амплитуды. К примеру, файл *4096Sampls_10KHz_32000.txt* есть текстовый список сэмплов звука, частотой 10КГЦ, амплитудой 32000, количество сэмплов 4096. Это позволит провести расчет ДПФ для длины выборки N =4096 или меньше.

Загружать данные мы будем в массив *Sampl,* массив будет иметь тип: *TcmxArray*. Подпрограмма загрузки данных будет иметь имя: LoadFileToArray и представлена в *листинге 3.3.1*.

Листинг 3.3.1

```
Const
  FCount     = 4096;
  FCount_1   = FCount-1;
  FCountDiv2 = FCount div 2;
  MaxValue   = 32000;

var
  Form1       : TForm1;
  List        : TStringList;
  NameFile    : String;
  Sampl       : TCmxArray;



implementation

{$R *.lfm}

{ TForm1 }


function TForm1.LoadFileToArray: Boolean;
var I:Integer;
begin
  Result :=False;

  if (List = Nil) or (List.Count < FCount) then Exit;

  try
    For I:=0 to FCount_1 do
      Sampl[I]:=StrToFloat(List[I]);

  except
    exit;
  end;

  Result:=True;
end;
```

Как Вы заметили, в модуле тестового приложения имеется ряд констант, я думаю их назначение понятно, ну или станет понятно в процессе дальнейшего обсуждения. Что касается процедуры загрузки данных, видно, что элементам массива *Sampl*, присваивается число с плавающей точкой, это стало возможным благодаря перегрузке операторов присваивания в модуле *complex*.

Процедура, которая будет выводить рассчитанный спектр на компонент Image, будет иметь имя Draw. Если считать, что мы уже имеем массив данных, рассчитанный при помощи ДПФ в виде массива *Sampl,* то процедура Draw будет иметь следующий вид (листинг 3.3.2).

Листинг 3.3.2

```
procedure TForm1.Draw;
var I:Integer;
    my:Real;
begin
  Image1.Picture:=Nil;
  Image1.Canvas.Brush.Color:=clWhite;
  Image1.Canvas.FillRect(Rect(0,0,Image1.Width,Image1.Height));


  my:=Image1.Height/MaxValue; //масштабный коэффициент

  Image1.Canvas.Pen.Color:=clRed;

  for I:=0 to FCount_1 do
    begin
      Image1.Canvas.MoveTo(I,Image1.Height);
      Image1.Canvas.LineTo(I,Image1.Height-Round(Sampl[I].Re*my));
    end;

  //Проведем линию середины
  Image1.Canvas.Pen.Color:=clBlack;
  Image1.Canvas.MoveTo(FCountDiv2,Image1.Height);
  Image1.Canvas.LineTo(FCountDiv2,0);

end;
```

Процедура достаточно проста, она выводит одномерный массив Sampl, на компонент Image. Стоит оговориться, что ширина Image, (Image1.Width) равна N (длине окна ДПФ). Это сделано специально, чтобы продемонстрировать, что рассчитав ДПФ для всего массива (окна) мы получим спектр сигнала, как в первой, так и во второй половине массива. Так же для большей визуализации этой особенности ДПФ, мы проводим центральную линию в Image. Так же стало понятно, зачем мы ввели значение максимальной амплитуды MaxValue, просто она (амплитуда) достаточно большая и изображать спектр в натуральную величину не имеет смысла. Для этого мы вводим масштабный коэффициент my, который будет масштабировать спектр в компоненте Image. В дальнейшем мы введем такой коэффициент и для масштабирования по оси x.

Далее рассмотрим процедуру *Calculate* (листинг 3.3.3), которая непосредственно вызывает процедуру расчета ДПФ (DFT) и измеряет время расчета спектра, для этого используется переменная Time, имеющая тип TperformanceTime из модуля performancetime, рассмотренного выше.

Листинг 3.3.3

```
procedure TForm1.Calcculate;
var
  Time:TPerformanceTime;
begin
  Time:=TPerformanceTime.Create;
  Time.Start;

  DFT(Sampl);

  Time.Stop;
  Memo1.Lines.Add('Время Расчета ДПФ    = '+FloatToStr(Time.Delay)+' сек.');
```

```
  Time.Free;
end;
```

Работа с классом TperformanceTime тривиальна, и мы не будет подробно на этом останавливаться.

3.4 Реализация ДПФ на языке Free Pascal

Ну вот мы наконец дошли до самой реализации ДПФ, в данном случае процедура расчета ДПФ имеет имя DFT. Процедура принимает массив исходных данных по ссылке и рассчитывает спектр при помощи ДПФ, после чего помещает рассчитанные данные в исходный массив. Мы не будем изобретать «крутой» велосипед, а попробуем решить задачу «в лоб» запрограммировав функцию (7) так как она есть (листинг 3.4.1).

Листинг 3.4.1

```
procedure TForm1.DFT(var D: TCmxArray);
var
    I,J,Len,Len_1,LenDiv2:Integer;
    TempAr:TCmxArray;
    wn:TComplex;
begin

  Len         := Length(D);
  Len_1       := Len-1;
  LenDiv2     := Len div 2;

  SetLength(TempAr,Len);

  For I:=0 to Len_1 do
    begin
      TempAr[I]:=0;

      For J:=0 to Len_1 do
        begin
          wn.Re := 0;
          wn.Im := -2*Pi*I*J/Len;
          wn    := exp(wn);

          TempAr[I]:=TempAr[I]+D[J]*wn;
        end;
    end;

 For I:=0 to Len_1 do
     D[I] := abs(TempAr[I])/LenDiv2;


  TempAr:=Nil;

end;
```

Видно, что в процедуре изначально готовится дополнительный массив, который будет содержать рассчитанный спектр. Он необходим, потому что мы не можем провести расчет спектра на месте (в исходном массиве). Далее идет два цикла For, первый это цикл расчета I-ой частоты, второй For, это сумма из формулы (7). Во втором цикле мы видим, exp-множители и непосредственно суммирование. Вы заметили, что мы считаем *exp* для комплексного числа и проводим математические операции (+,-,/, *) благодаря перегрузке операторов для данного типа (более подробно см модуль *Complex*). Это позволяет сократить код, и увеличить его прозрачность практически в разы. После выхода из циклов мы проводим расчет амплитуды и ее нормализацию (делим на N/2). После выхода из процедуры DFT, исходный массив данных D, будет представлять из себя спектр сигнала, который мы и выводим на компонент Image. Исходники данного проекта для Lazarus 0.9.30 и выше располагаются в директории *DFT/DFT_NO*

Вообще, строго говоря, нельзя трактовать результат полученный при помощи ДПФ как спектр сигнала. Этот спектр - просто данные, применив к которым обратное преобразование, можно получить исходную функцию, **вот именно по этому и применяются различного рода нормализации**.

Вы можете провести эксперименты, изменив значение константы Fcount которая и есть размер ДПФ, и посмотреть, что при этом изменится.

Мы умолчали, один важный факт, а именно то, что число N (размер ДПФ) должно быть степенью двойки. На данный момент, это не имеет никакого смысла, но в дальнейшем на этом будет строиться весь алгоритм, когда мы перейдем к изучению БПФ.

При этом при работе с файлом *4096Sampls_10KHz_32000.txt* не очень отчетливо видно, что первая половина массива является зеркальным отражением второй. Чтобы это увидеть, необходимо открыть файла *4096Sampls_20KHz_15000.txt,* который представляет собой сэмплы звука частотой 20КГц. И амплитудой 15000, рассчитав ДПФ, для этих данных, Вы должны были получить следующее (Рис. 3.4.1).

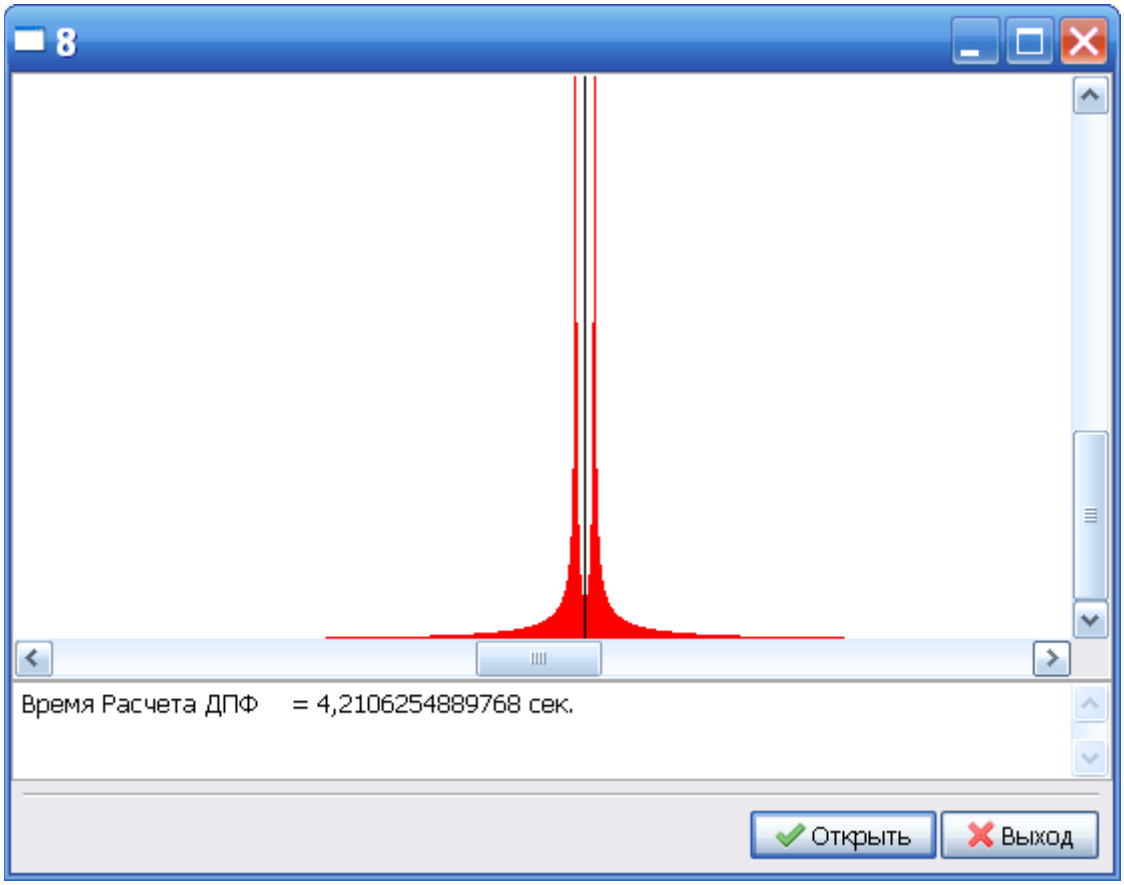


Рис. 3.4.1 Скриншот программы для тестирования нашей реализаций ДПФ, при расчете данных из файла 4096Sampls_20KHz_15000.txt

Рис. 3.4.1 очень хорошо иллюстрирует зеркальную симметричность рассчитанного спектра. Также можно заметить, что амплитуды уменьшились (при условии, что константа MaxValue=32000), так как амплитуда теперь составляет всего 15000.

Но как в первом, так и во втором случае, ДПФ нам рассчитало частоты с заниженной амплитудой. Отчетливо видно, что амплитуды меньше на 5-10%, это связано с погрешностью как самого ДПФ, так и погрешностью работы ПК с переменными вещественного типа. Видно, что на самом деле мы рассчитали не одну частоту, а целый ряд частот. Метод ДПФ смог выделить основную гармонику, но при этом он «напридумывал» еще несколько. Это поведение ДПФ рассматривается в работах [1,2,3,4]. В работах [3,4] рассматривались так называемые оконные функции сглаживания, которые могут уменьшить данный эффект.

3.5 Применение сглаживающих оконных функций

Для иллюстрации влияния оконных функций, в директории *DFT/DFT_NO_Hamming_window*, располагается проект аналогичный выше рассмотренному, но с применением оконной функции, на примере окна Хемминга (Hamming window). Результат расчета спектра для файла *4096Sampls_20KHz_15000.txt,* можно видеть на *Рис. 3.5.1*

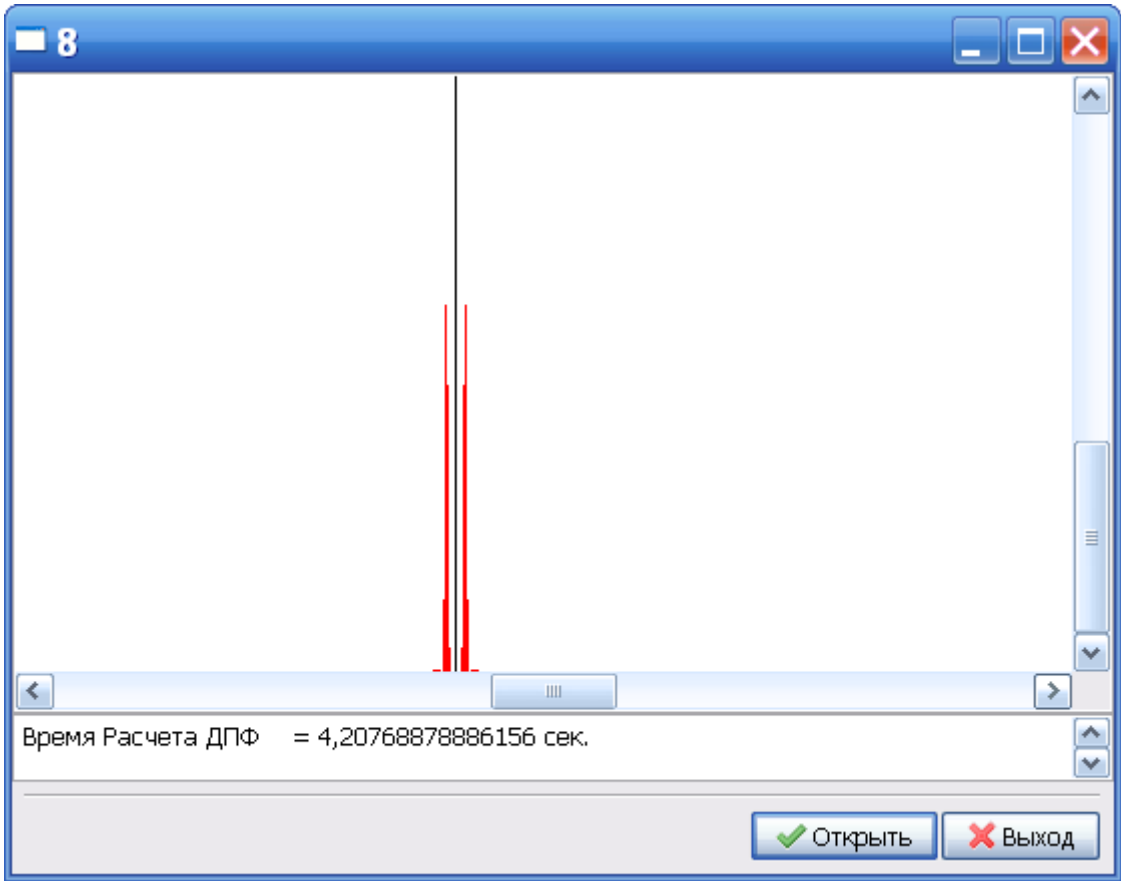


Рис. 3.5.1 Скриншот программы для тестирования нашей реализаций ДПФ, при расчете данных из файла 4096Sampls_20KHz_15000.txt, с использованием оконной функции Хемминга.

Как можно было заметить, оконная функция действительно уменьшила количество частот, которых просто нет в первоначальном сигнале. Почему оконные функции способствуют такому результату, я не берусь объяснять.

3.6 Оптимизация алгоритма расчета ДПФ

Мы уже ни раз говорили и даже провели расчеты некоторых спектров. И видели, что для ДПФ нет необходимости считать все N точек. Ведь нужный нам спектр мы можем получить, рассчитав всего N/2 точек. По этому в директории *DFT/DFT_O1* располагается проект, в котором мы учли данное обстоятельство. Согласно этому наша процедура расчета ДПФ примет следующий вид (Листинг 3.6.1).

Листинг 3.6.1

```
procedure TForm1.DFT(var D: TCmxArray);
var
    I,J,Len,Len_1,LenDiv2,LenDiv2_1:Integer;
    TempAr:TCmxArray;
    wn:TComplex;
begin

  Len         := Length(D);
  Len_1       := Len-1;
  LenDiv2     := Len div 2;
  LenDiv2_1   := LenDiv2 -1;

  SetLength(TempAr,LenDiv2);

  For I:=0 to LenDiv2_1 do
    begin
      TempAr[I]:=0;

      For J:=0 to Len_1 do
        begin
          wn.Re := 0;
          wn.Im := -2*Pi*I*J/Len;
          wn    := exp(wn);

          TempAr[I]:=TempAr[I]+D[J]*wn;
        end;
    end;

 For I:=0 to LenDiv2_1 do
     D[I] := abs(TempAr[I])/LenDiv2;


  TempAr:=Nil;

end;
```

По идее, так как мы уменьшили число расчетов вдвое, мы вправе ожидать прирост производительности так же в два раза. На *Рис. 3.6.1*. виден расчет данных (*4096Sampls_10KHz_32000.txt)*, для N = 4096, видно, что по сравнению со старой функцией DFT (Рис. 3.3.1), которая «пробегала» все N, мы получили увеличение скорости практически в два раза.

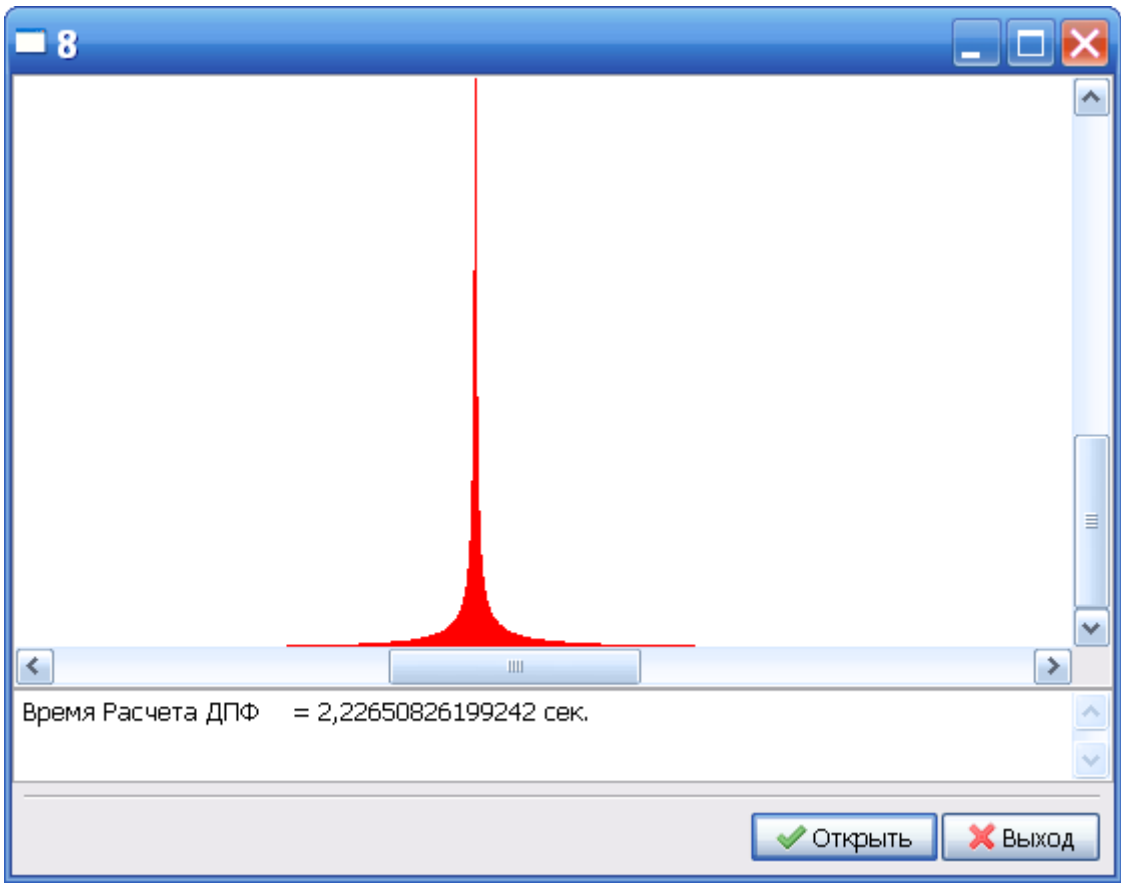


Рис. 3.6.1 Скриншот программы для тестирования нашей реализаций ДПФ, при расчете данных из файла 4096Sampls_10KHz_32000.txt, при расчете только N/2 точек

На данном этапе, я хотел бы вас попросить, более детально ознакомится с тестирующей программой, попробовать поэкспериментировать с различными значениями длины ДПФ N. Провести расчеты спектра используя различные оконные функции, список последних можно найти к примеру в работе [10]. Также посмотрите, что будет, если проводить нормализацию при другом коэффициенте, отличном от N/2 или не проводить ее вовсе. Сама процедура расчета ДПФ занимает достаточно мало места и я не думаю, что в ее понимании возникнут трудности. Благодаря перегрузке основных операторов и функций, мы позволили себе обращаться с комплексными числами как с вещественными, что в свою очередь позволило нам заострять внимание только на самом алгоритме и не обращать внимание на то как нужно число сложить или умножить, про расчет exp я вообще молчу. Некоторые авторы при программировании ДПФ или БПФ, не создают отдельный тип комплексного числа и имеют дело с двумя массивами один для *реальной*, другой для *мнимой* части комплексного числа. Из-за чего им приходится описывать любую операцию самим, по этому процедура ДПФ у них может занимать более 100 строк. Мы этого недостатка лишены, благодаря модулю Complex. По этому я убедительно прошу вас более детально изучить модуль Complex.

В остальной части работы, мы практически не будем останавливаться на тестирующей программе, потому что последняя не будет претерпевать каких-либо значительных изменений. Всё наше внимание мы будем уделять непосредственно

программированию алгоритма БПФ и его оптимизации.

Как Вы уже заметили, расчет спектра для данных длинной N=4096, в последнем варианте у нас занял около 2-х секунд. Вы спросите много это или мало, я отвечу, что для моего процессора Pentium D 640 3GHz это слишком много. Посчитайте сами, что звук на компакт дисках имеет частоту дискретизации Vd=44100, то есть в одной секунде звука у нас ровно 44100 сэмплов (отсчетов). В среднем музыкальная композиция занимает 5 минут. Так как мы рассчитали данные длинной 4096 за 2.2 секунды, то 1 секунду звука мы будем рассчитывать приблизительно Vd/N * 2.2 = 23.7 секунд. А всю музыкальную композицию 5*60*23.7= 7 110 секунд или 118 минут. Это просто неприемлемо. Вот чтобы рассчитывать спектр звука в реальном времени, мы должны уменьшить время расчета приблизительно в 24 раза. Некоторые скажут, что если бы мы написали алгоритм полностью на языке ассемблера, то все было бы хорошо. Считается, что если подпрограмма достаточно хорошо и правильно (оптимизировано) написана на языке высокого уровня, в данном случае на Free Pascal. То переписав подпрограмму на язык ассемблера (при помощи встроенного ассемблера), это может дать прирост в производительности в 2-20 раз. Что даже в самом лучшем случае будет не достаточно для работы в реальном времени. Для этих целей был придуман, так называемый алгоритм *быстрого преобразования Фурье* (БПФ), который уменьшает время расчета ДПФ в 100 и более раз. На этом мы закончим рассмотрение ДПФ и в следующей главе перейдем к изучению БПФ.

Для примера приведем сравнительную диаграмму времени расчета одного и того же спектра (рис. 3.6.2). В директории Andere/ALGLIB_FFT находится проект для расчета спектра при помощи библиотеки ALGLIB ver. 2.6.0.

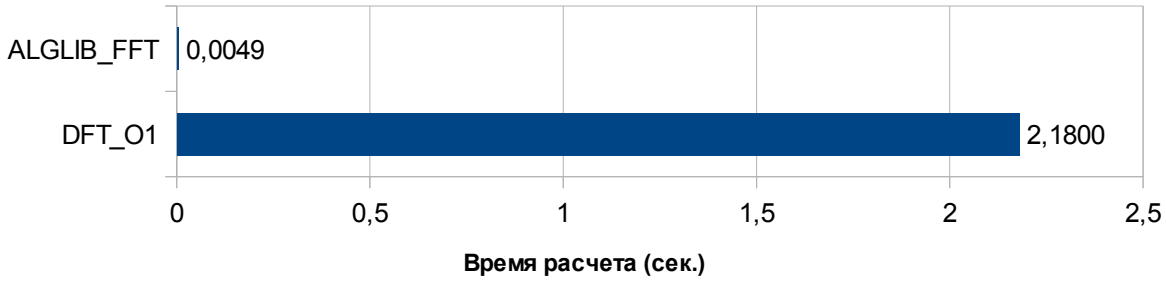


Рис. 3.6.2 Временная диаграмма расчета спектра. DFT_O1 — наша реализация ДПФ и ALGLIB_FFT — расчет при помощи библиотеки ALGLIB ver. 2.6.0 методом БПФ.

Как можно было заметить из рис. 3.6.2, наш метод расчета при помощи ДПФ проигрывает библиотеке ALGLIB ver. 2.6.0 более чем в 400 раз!!!. Можно сказать, нам есть к чему стремится. Но как бы мы не старались переписать ДПФ, мы не получим такой прирост скорости, даже при использовании языка ассемблера, для получения таких показателей нам необходимо пересмотреть подход к расчету формулы 7. Именно так и поступили авторы БПФ, они смогли переписать формулу 7 так, что ее стало легче посчитать. В следующей главе мы с вами перепишем формулу 7 одним из способов, который приведет нас к так называемому методу БПФ с прореживанием по времени.

4 Быстрое преобразование Фурье

Изобретение *Быстрого преобразования Фурье* приписывается Кули (Coolet) и Таки (Tukey) — 1965 г. На самом деле БПФ неоднократно изобреталось до этого, но важность его в полной мере не осознавалась до появления современных компьютеров. Некоторые исследователи приписывают открытие БПФ Рунге (Runge) и Кёнигу (Konig) в 1924 г. Наконец, открытие этого метода приписывается ещё Гауссу (Gauss) в 1805 г.
В данной работе мы с вами рассмотрим одну из возможных реализаций алгоритма быстрого преобразования Фурье (БПФ) по степени 2 с прореживанием по времени, на языке Free Pascal и дальнейшей оптимизацией при помощи встроенного ассемблера.

4.1 Вывод БПФ по основанию 2 с прореживанием по времени

На самом деле БПФ есть тоже самое, что и ДПФ, но его переписали таким образом, что его стало легче посчитать. Чтобы это продемонстрировать мы возьмем нашу формулу для расчета ДПФ, а именно формулу (7). И проведем аналогичные рассуждения и выкладки, что позволит нам получить долгожданную формулу БПФ. Для удобства я приведу формулу (7) еще раз, чтобы не мучить колесико вашей мыши. При этом сохраним нумерацию формул, и так: ДПФ будет иметь вид (15) она же формула (7).

$$F(k)=\sum_{n=0}^{N-1}\left(f(n)\cdot W_N^{n\cdot k}\right),\qquad k=0..\mathrm{N}-1; \tag{15.}$$

БПФ по основанию 2 с прореживанием по времени заключается в разбиении исходной последовательности отсчетов $f(n),\ n=0..\mathrm{N}-1$ на две последовательности длинной $N/2,\quad f0(n)\ u\ f1(n),\quad n=0..\mathrm{N}/2-1$, таких что: $f0(n)=f(2\cdot n)$, а $f1(n)=f(2\cdot n+1),\quad n=0..\mathrm{N}/2-1$
Другими словами, последовательность $f0(n)$ содержит отсчеты исходной последовательности $f(n)$ с четными индексами, а $f1(n)$ - с нечетными.

С учетом, этого формула (15) будет иметь вид (16):

$$F(k)=\sum_{n=0}^{N/2-1}\left(f(2\cdot n)\cdot W_N^{2\cdot n\cdot k}\right)+\sum_{n=0}^{N/2-1}\left(f(2\cdot n+1)\cdot W_N^{(2\cdot n+1)\cdot k}\right)=\ldots$$
$$\ldots=\sum_{n=0}^{N/2-1}\left(f(2\cdot n)\cdot W_N^{2\cdot n\cdot k}\right)+W_N^k\cdot\sum_{n=0}^{N/2-1}\left(f(2\cdot n+1)\cdot W_N^{2\cdot n\cdot k}\right),\quad k=0..\mathrm{N}-1. \qquad \textbf{16.}$$

Если рассмотреть только первую половину спектра $F(k),\ k=0..\mathrm{N}/2-1$, а также учесть что:

$$W_N^{2\cdot n\cdot k}=\exp\left(-i\frac{2\cdot\pi}{N}\cdot 2\cdot n\cdot k\right)=W_{N/2}^{n\cdot k}, \qquad \textbf{17.}$$

тогда (16) можно записать так:

$$F(k)=\sum_{n=0}^{N/2-1}\left(f(2\cdot n)\cdot W_{N/2}^{n\cdot k}\right)+W_N^k\cdot\sum_{n=0}^{N/2-1}\left(f(2\cdot n+1)\cdot W_{N/2}^{n\cdot k}\right)=\ldots \qquad \textbf{18.}$$
$$\ldots=F0(k)+W_N^k\cdot F1(k),\quad k=0..\mathrm{N}/2-1.$$

Где $F0(k)\ \ u\ \ F1(k),\ k=0..\mathrm{N}/2-1$ есть ни что иное как ДПФ для «четной» $f0(n)$ и «нечетной» $f1(n),\quad n=0..\mathrm{N}/2-1$ последовательности исходного сигнала.

$$F0(k)=\sum_{n=0}^{N/2-1}\left(f0(n)\cdot W_{N/2}^{n\cdot k}\right),\quad k=0..\mathrm{N}/2-1,$$
$$F1(k)=\sum_{n=0}^{N/2-1}\left(f1(n)\cdot W_{N/2}^{n\cdot k}\right),\quad k=0..\mathrm{N}/2-1. \qquad \textbf{19.}$$

Теперь мы с вами рассмотрим вторую половину спектра:
$F(k+N/2),\ k=0..\mathrm{N}/2-1$

$$F(k+N/2)=\sum_{n=0}^{N/2-1}\left(f(2\cdot n)\cdot W_{N/2}^{n\cdot(k+N/2)}\right)+\ldots$$
$$\ldots+W_N^{(k+N/2)}\cdot\sum_{n=0}^{N/2-1}\left(f(2\cdot n+1)\cdot W_{N/2}^{n\cdot(k+N/2)}\right),\qquad k=0..\mathrm{N}/2-1. \quad \textbf{20.}$$

Рассмотрим более детально множители:

$$W_{N/2}^{n\cdot(k+N/2)}=\ W_{N/2}^{n\cdot N/2}\cdot W_{N/2}^{n\cdot k}=\ W_{N/2}^{n\cdot k} \quad \textbf{21.}$$

Так как:

$$W_{N/2}^{n\cdot N/2}=\ \exp\left(-i\frac{2\cdot\pi}{N/2}\cdot n\cdot N/2\right)=\exp(-i\cdot 2\cdot\pi\cdot n)=\ 1 \quad \textbf{22.}$$

Причем, это справедливо для любого целого **n**, потому что согласно формуле Эйлера:

$$\exp(i\cdot x)=\cos(x)+i\cdot\sin(x) \quad \textbf{23.}$$

а как известно:

$$\cos(2\cdot\pi\cdot n)=1,\qquad a\qquad \sin(2\cdot\pi\cdot n)=0 \quad \textbf{23.}$$

для любого целого **n**.

Рассмотрим еще один коэффициент в (20):

$$W_N^{(k+N/2)}=\ W_N^{N/2}\cdot W_N^k=\exp(-i\frac{2\cdot\pi}{N}\frac{N}{2})\cdot W_N^k=-W_N^k \quad \textbf{24.}$$

для понимания этого необходимо также обратится к формулам Эйлера.

Теперь, согласно (21) и (24) выражение (20) будет иметь вид (25)

$$F(k+N/2)=\sum_{n=0}^{N/2-1}\left(f(2\cdot n)\cdot W_{N/2}^{n\cdot k}\right)-\ldots$$
$$\ldots-W_N^k\cdot\sum_{n=0}^{N/2-1}\left(f(2\cdot n+1)\cdot W_{N/2}^{n\cdot k}\right)=\ldots \qquad \textbf{25.}$$
$$\ldots=F0(k)-W_N^k\cdot F1(k),\qquad k=0..\mathrm{N}/2-1.$$

Таким образом окончательно можно записать:

$$F(k)=F0(k)+W_N^k\cdot F1(k),\qquad k=0..\mathrm{N}/2-1, \qquad \textbf{26.}$$
$$F(k+N/2)=F0(k)-W_N^k\cdot F1(k),\qquad k=0..\mathrm{N}/2-1.$$

Вот и все, мы получили формулу БПФ. Вывод, занял всего-то 2 страницы, и вся «фишка» заключалась в знании умножения exp и формул Эйлера. Но эти сокращения и упрощения, которые мы провели, позволяют вновь переписанному ДПФ обрабатывать данные в разы быстрее.

4.2 Как работает БПФ

Достаточно сложно объяснить каким образом по формулам (26) считать спектр сигнала, для этого мы изобразим это графически на примере данных, длинной N=8. В начале мы имеем исходные данные (дискретный сигнал), на выходе же мы получим спектр сигнала (Рис. 4.2.1)

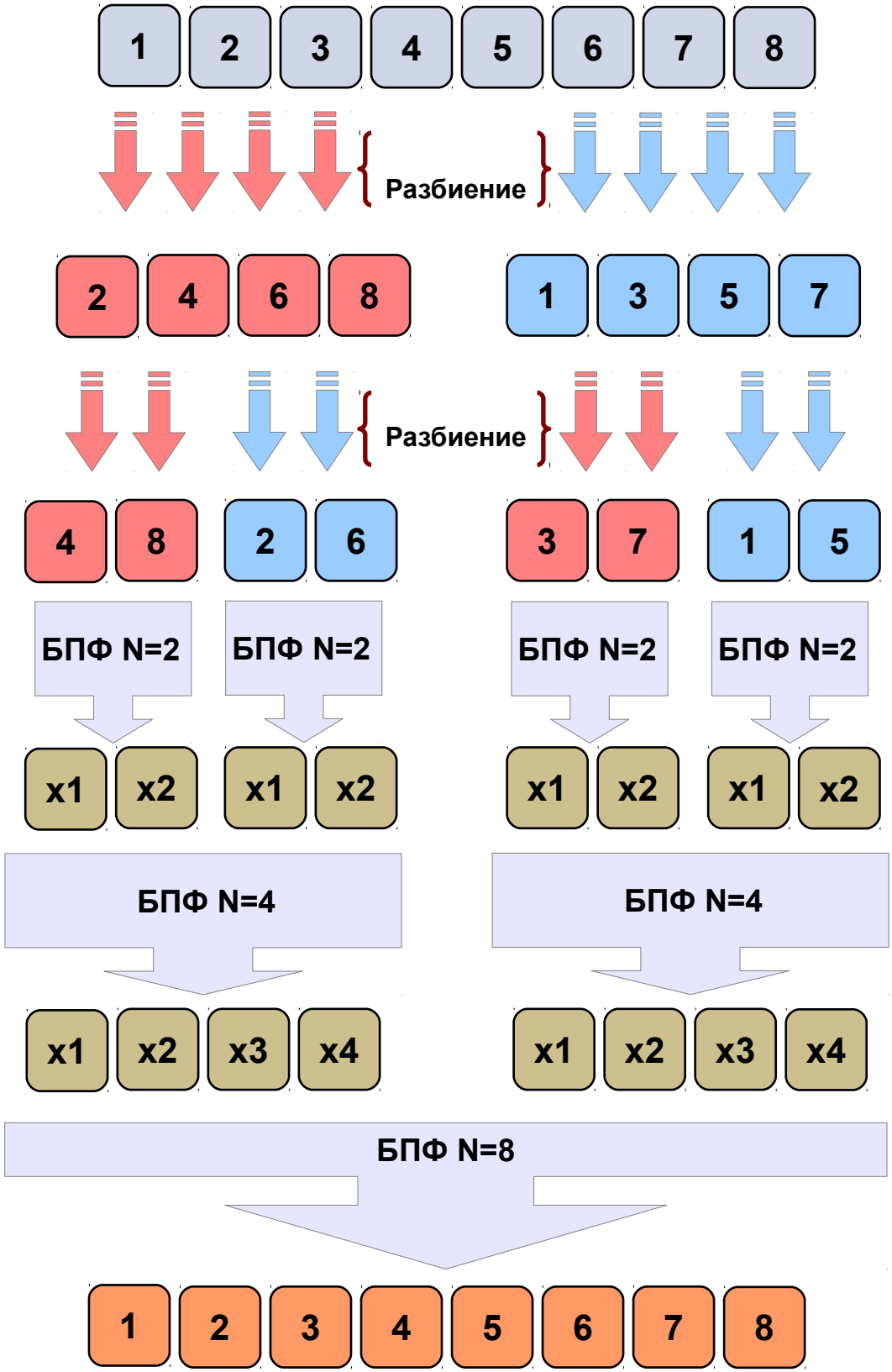


Рис. 4.2.1. Графическое представление алгоритма БПФ на примере данных длиной N=8.

Для большей иллюстрации, мы обозначили красным цветом — четные сэмплы, синим — нечетные. Видно, что после первого разбиения, мы слева получаем все четные сэмплы, а с права нечетные относительно первоначального расположения сэмплов. При втором разбиении, мы как бы начинаем нумерацию с начала, по этому цвета имеют все то же значение, но для того чтобы увидеть куда переместились первоначальные сэмплы, мы оставили нумерацию сквозной. Дойдя до блоков длиной N=2, мы можем для них подсчитать БПФ по формуле (26). После чего мы по этой же формуле считаем БПФ для N=4 и т.д. до тех пор пока не дойдем до крайнего N, в данном случае до 8. Аналогичным образом проходит работа и с другими значениями N. Мы начинаем разбивать последовательность на половинки, разделяя четные\нечетные и дойдя до N=2 начинаем расчет БПФ. Почему именно до N=2, да потому что не имеет ни какого практического смысла считать БПФ для N=1, так как БПФ для одного числа есть само число. И еще потому что для этого случая коэффициент $W_N^k = 1$ так как в данном случае N=2, а k может принимать значение только 1 (см формулу (26)), ведь длина блока равна всего двум (N=2). По этому в БПФ для данных длиной N=2 и умножения как такового не требуется. Так же, дойдя до расчета БПФ определенной длины, для нас перестают играть роль сами данные (в смысле нам не важно рассчитываем мы первый блок [f(4), f(8)] или третий [f(3), f(7)]) формулы одинаковы!. По этому дойдя до расчета БПФ, мы обозначаем данные как некий набор данных ***x***, над которыми и проводим работу.

Ну давайте уже более детально, так сказать на цифрах. Вот дошли мы до N=2, и пускай мы рассматриваем первый блок, то есть [f(4), f(8)], тогда x1, x2 будут подсчитаны следующим образом:

$$\begin{aligned} x1 &= f(4) + f(8), \\ x2 &= f(4) - f(8) \end{aligned} \qquad \textbf{27.}$$

Мы помним, что коэффициент $W_N^k = 1$, аналогично мы будет считать и третий блок [f(3), f(7)]:

$$\begin{aligned} x1 &= f(3) + f(7), \\ x2 &= f(3) - f(7) \end{aligned} \qquad \textbf{28.}$$

Ну или в общем виде для блока длинной N=2 [d1, d2]:

$$\begin{aligned} x1 &= d1 + d2, \\ x2 &= d1 - d2 \end{aligned} \qquad \textbf{29.}$$

Для блоков длинной больше двух, уже будет необходим коэффициент W_N^k , давайте рассмотрим как бы мы считали блок длинной N=8. Пусть наш блок представлен вот таким набором данных: [d1, d2, d3, d4, d5, d6, d7, d8], тогда значения БПФ будут иметь вид (30).

$$
\begin{aligned}
x1 &= d1 + W_8^1 \cdot d5\,, \\
x5 &= d1 - W_8^1 \cdot d5\,, \\
x2 &= d2 + W_8^2 \cdot d6\,, \\
x6 &= d2 - W_8^2 \cdot d6\,, \\
x3 &= d3 + W_8^3 \cdot d7\,, \\
x7 &= d3 - W_8^3 \cdot d7\,, \\
x4 &= d4 + W_8^4 \cdot d8\,, \\
x8 &= d4 - W_8^4 \cdot d8.
\end{aligned}
\qquad \textbf{30.}
$$

Благодаря (30), виден общий принцип, а именно, мы сшиваем (объединяем) первую половину данных (тут располагаются четные элементы, ведь до этого мы их как раз и перемешивали) со второй половиной (нечетные).

На входе мы имеем блок длиной N, после чего начинаем разбивать его на четную\нечетную половинки, далее каждую из этих половинок мы так же разбиваем на четную и нечетную половинку и так до тех пор пока не дойдем до половинок длинной 2 (вот почему нам важно чтобы N было степенью двойки). Так же вы должны понимать что половинок длиной 2 ровно N/2. См таблицу 4.2.1. Когда мы дошли до половинок, состоящих только из двух сэмплов, мы начинаем расчет БПФ для каждой такой половинки. Когда мы рассчитали БПФ для каждой половинки состоящей из 2-х сэмплов, мы начинаем считать БПФ для половинок состоящих из 4-х и т.д, пока не дойдем до половинки, длина которой равна длине блока N.

Длина половинки:	**Длина блока N**											
	2	4	8	16	32	64	128	256	512	1024	2048	4096
	Количество половинок											
2	1	2	4	8	16	32	64	128	256	512	1024	2048
4		1	2	4	8	16	32	64	128	256	512	1024
8			1	2	4	8	16	32	64	128	256	512
16				1	2	4	8	16	32	64	128	256
32					1	2	4	8	16	32	64	128
64						1	2	4	8	16	32	64
128							1	2	4	8	16	32
256								1	2	4	8	16
512									1	2	4	8
1024										1	2	4
2048											1	2
4096												1

Таблица 4.2.1 Количество половинок определенной длины, для различной длины блока БПФ.

4.3 Вычислительная эффективность алгоритма БПФ по основанию 2 с прореживанием по времени

Ну а теперь, попытаемся разобраться, почему же БПФ быстрее ДПФ. Некоторые авторы употребляют в своих объяснениях выражение типа: «сложность ДПФ больше чем сложность БПФ...» и приводят аналитическое выражение, и это и хорошо и плохо. Они можно сказать ближе к математикам, и они не различают умножение и сложение и говорят, что нужно выполнить столько-то операций. Так-то оно так, но ведь даже в «хороших» книгах иногда пишут, что операция сложения может занять 1 или даже 0.5 такта процессора, тогда как умножение как минимум 5 или больше (тут я сказал навскидку). Поэтому то, что у математиков операция это еще ничего не значит для нас как программистов. При этом для нас это имеет более важный момент, ведь мы работаем с комплексными числами, для сложения которых необходимо уже две «чистых» операции сложения, а для умножения комплексного числа и того больше.

Давайте условимся, что в дальнейшем мы не будем различать операцию сложения и вычитания, ведь со стороны 1-1 это ничто иное, что 1+(-1). В современных процессорах, эти операции выполняются достаточно быстро и практически за одинаковое время или вообще за одно и тоже количество тактов. Это просто избавит от нагромождений при дальнейшем рассмотрении.

Так что давайте рассмотрим, сколько же нам необходимо комплексных операций сложения и умножения для ДПФ и аналогичного БПФ. Так же для простоты предположим, что все поворотные коэффициенты (W_N^k) рассчитаны заранее.

Введем такое обозначение как X(+), что будет значить X операций сложения, X(*) X – операций умножения.

И так: для N=8, ДПФ затратит 4*(8(+) и 8(*)), 4 так как мы уже обсуждали, что для ДПФ достаточно рассчитать только первую половину данных. N/2 =4.

Итого для ДПФ при N=8: 32(+) и 32(*).

БПФ: тут для облегчения необходимо обратится к (рис. 4.2.1) так как мы уже показали, что для нижней итерации БПФ (когда N=2) нам вообще не нужно умножение, все можно решить одним сложением. И для БПФ при N=2 необходимо всего 2(+) так как блоков по два у нас 4, то на нижней итерации необходимо 8(+).

При сшивании блока длиной N=4, необходимо 4(+) и 2(*) и блоков по 4 у нас два, итого: 8(+) и 4(*).

При сшивании блоков длиной N=8, необходимо 8(+) и 4(*). И количество блоков по 8 у нас только один, ведь сэмплов то всего 8 (N=8).

Давайте для наглядности сведем все в одну таблицу:

Длина блока N	Операций сложения	Операций умножения
2	8	0
4	8	4
8	8	4

Итого для БПФ при N=8: 24(+) и 8(*).

При этом, чем больше N, тем больше выигрыш дает БПФ. Самое главное, что уменьшилось число операций умножения, а это не может не радовать.

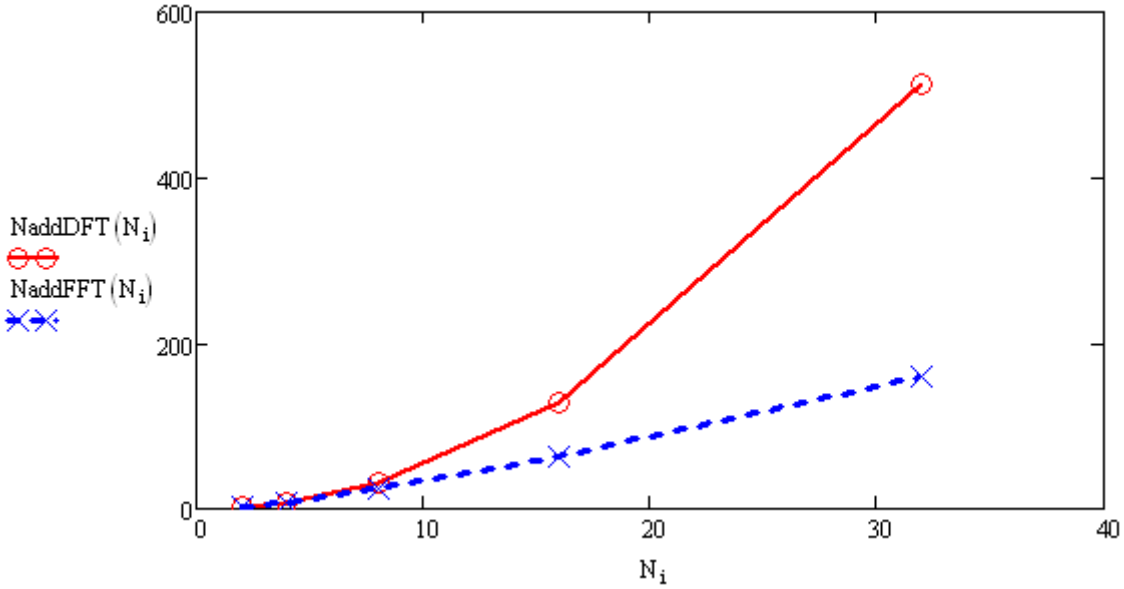
Ну и в общем виде, для ДПФ при длине блока N, число операций сложения Nadd равно числу операций умножения Nmul и выражается как:

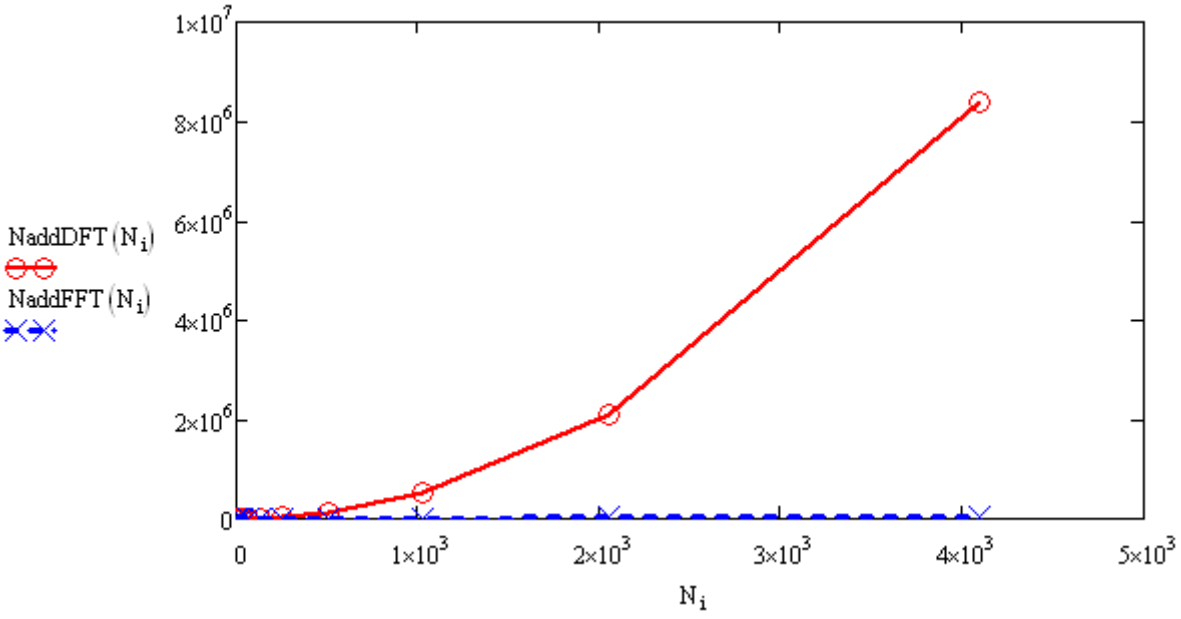
$$Nadd = Nmul = \frac{N}{2} \cdot N \qquad \textbf{31.}$$

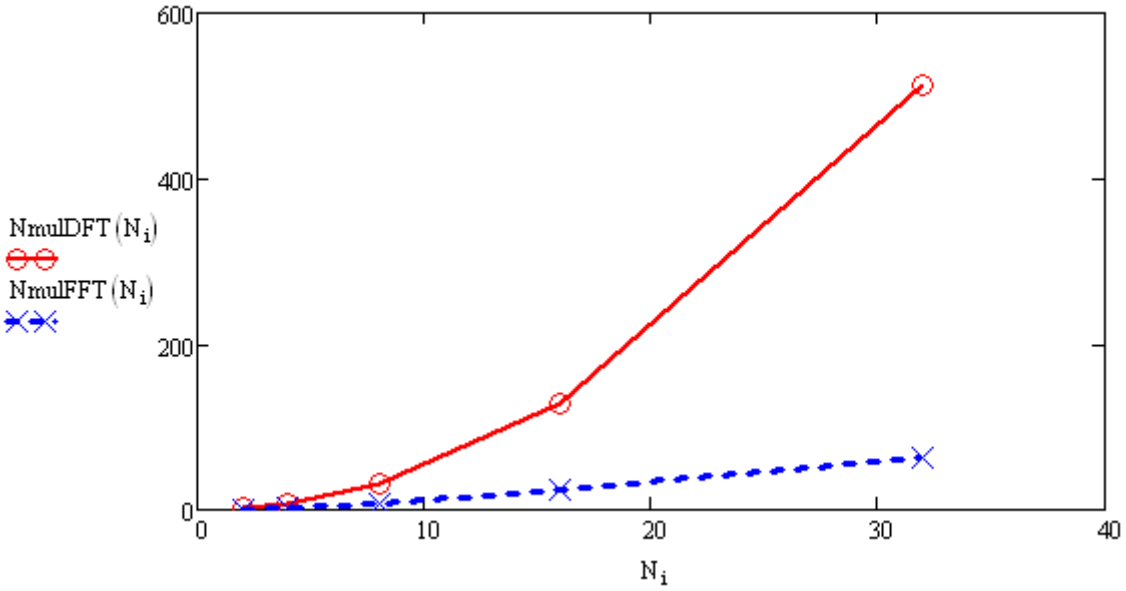
Тогда как для БПФ соответствующие значения имеют вид (32)

$$Nadd = N \cdot \log_2(N),$$
$$Nmul = \frac{N}{2} \cdot \left(\log_2(N) - 1\right) \qquad \textbf{32.}$$

Давайте отобразим эти зависимости графически, чтобы уж закончить с рассмотрением вопроса: почему БПФ быстрее ДПФ.




NaddDFT(N_i)
NaddFFT(N_i)
1×10^7
8×10^6
6×10^6
4×10^6
2×10^6
0
0
1×10^3
2×10^3
3×10^3
4×10^3
5×10^3
N_i


NmulDFT(N_i)
NmulFFT(N_i)
600
400
200
0
0
10
20
30
40
N_i

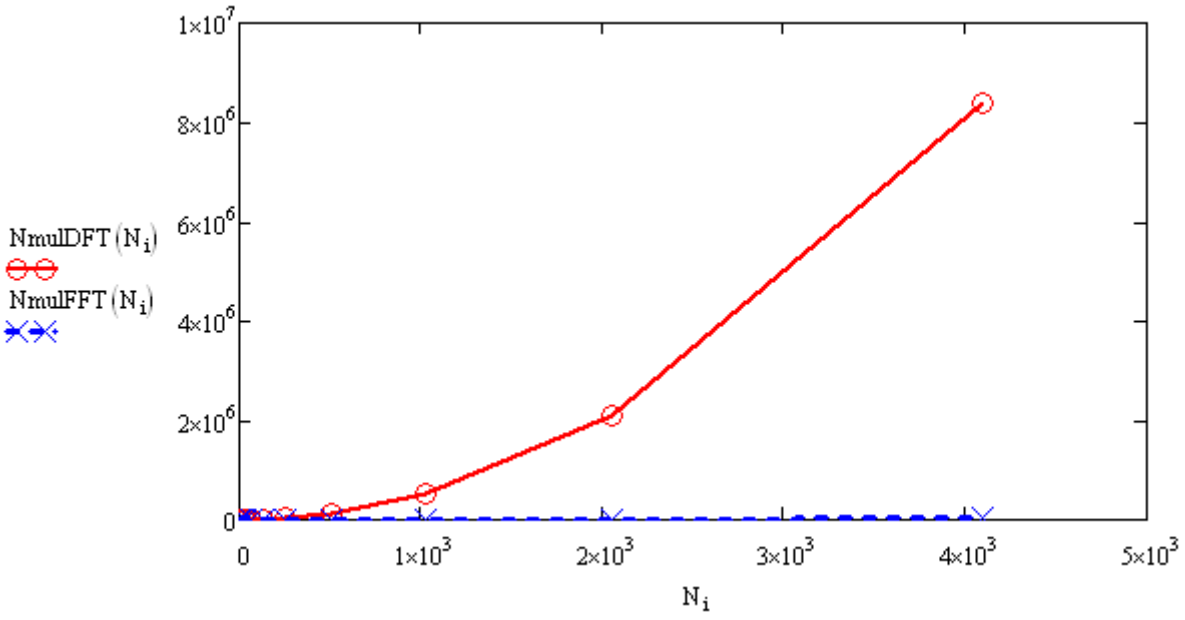


Рис. 4.3.1 Графическое представление количества операций сложения (add) и операций умножения (mul) при различной длине данных N, для ДПФ (DFT) и БПФ (FFT).

Из Рис 4.3.1 очень хорошо видно, что чем больше число N, тем более колоссальной становится разность по количеству операций между БПФ и ДПФ. Вот почему БПФ называют б*ыстрым*.

Ну и если у вас возник вопрос: «почему же при сшивании блоков длинной N нам требуется всего N/2 умножения, а не N?”, то считайте это домашним заданием :).

5 Реализация алгоритма БПФ по основанию 2 с прореживанием по времени на языке Free Pascal

5.1 Рекурсивный метод

Большинство авторов, рассматривают именно рекурсивный метод расчета БПФ, это связано с тем, что БПФ с прореживанием по времени (рис. 4.2.1) достаточно легко запрограммировать именно рекурсивным методом. Мы не будем отступать от этого правила. Но мы поступим немного иначе в вопросе программирования. К примеру мы не будем рассчитывать exp множители внутри функции FFT, это просто ни к чему. Мы ведь с вами понимаем, что выбрав длину окна БПФ (N), мы будем рассчитывать какое-то количество данных, длина которых превосходит N. Значит рассчитывать каждый раз exp множители не нужно... Для этого мы введем дополнительную подпрограмму, которая будет возвращать нам массив множителей, который мы по ссылке будем передавать непосредственно в подпрограмму расчета БПФ. Ведь согласитесь лучше передать один лишний параметр (в данном случае ссылку на массив) чем каждый раз считать эти множители внутри функции. Ведь посчитайте сами, сколько операций необходимо для расчета поворотных множителей скажем для БПФ при N=4096.

По этому мы выносим это в отдельную функцию **GetFFTExpTable**, которая вернет нам массив множителей W_N^k *(листинг 5.1.1)*

Листинг 5.1.1

```
{-------------------------------------------------------------------------------
function GetFFTExpTable(CountFFTPoints:Integer;
                        InverseFFT:Boolean=False): TCmxArray;

Вернет таблицу Exp множителей для FFT анализа. (Комплексных)
CountFFTPoints - число точек для FFT Анализа.
InverseFFT     - Если True, то множители нужны для обратного БПФ,
                 иначе для прямого.


Пример:
CountFFTPoints = 32
CountBlock     = 5+1, а именно [0][1][2..3][4..7][8..15][16..31]
Len            = 32

После чего заполняем каждый блок множителями для БПФ, начиная с блока [1]
Result (Массив) будет выглядеть так:
[0][1][2..3][4..7][8..15][16..31]
 |  |    |      |      |       |
 |  |    |      |      |       |--- для сшивания блоков длинной = 16
 |  |    |      |      |
 |  |    |      |      |---------- для сшивания блоков длинной = 8
 |  |    |      |
 |  |    |      |----------------- для сшивания блоков длинной = 4
 |  |    |
 |  |    |----------------------- для сшивания блоков длинной = 2
 |  |
 |  |-------------------------- для сшивания блоков длинной = 1
 |
 |------------------------------ не используется, необходим для выравнивания

 а именно, если знаем, что нужно сшить блоки длинной 16 (блоков всегда два),
 то и exp множители лежат в массиве начиная с номера = 16, и так как блок имеет
 длину 16, множителей тоже 16 !. прям полная идилия :)}
function GetFFTExpTable(CountFFTPoints:Integer; InverseFFT:Boolean=False):
TCmxArray;
var I,StartIndex,N,LenB:Integer;
    w,wn:TComplex;
    Mnogitel:Double;
begin

  Mnogitel := -2*Pi;               //Прямое  БПФ

  if InverseFFT then
     Mnogitel:= 2*Pi;              //Обратное  БПФ


  SetLength(Result,CountFFTPoints);

  LenB:=1;

  while LenB <CountFFTPoints do //Пробегаем каждый блок.
    begin
      N     := LenB;       //поучаем число элементов в блоке
      LenB  := LenB shl 1;
```

```
      //Готовим первый EXP множитель(корень) для БПФ.
      wn.Re := 0;
      wn.Im := Mnogitel/LenB;
      wn    := exp(wn);
      w     := 1;

      StartIndex:=N;    //Стартовый индекс в массиве.
      Dec(N);           //Т.к индексация с нуля
      For I:=0 to N do  //заполняем блок exp множителями для БПФ, в массиве
        begin
          Result[StartIndex+I]:=w;
          w:=w*wn;
        end;
    end;
end;
```

Для того, чтобы иметь возможность использовать технику, при которой можно рассчитывать БПФ на месте (в переданном массиве) необходимо массив данных расположить согласно нижней итерацией (то есть чтобы входные данные были расположены в последовательности, которая получается после всех перемешиваний (разделения на четные\нечетные)).
При этом мы понимаем, чтобы не рассчитывать эти индексы каждый раз, мы будем иметь массив, в котором будут расположены индексы согласно нижней итерации. Для этого вводим функцию **GetArrayIndex**, которая вернет нам этот самый массив индексов. (листинг 5.1.2)

Листинг 5.1.2

```
{-----------------------------------------------------------------------------
function GetArrayIndex(CountPoint: Integer;MinIteration: Integer): TIndexArray;
Вернет массив длиной CountPoint из чисел(индексов) от 0 до CountPoint-1.
Этот массив будет содержать в себе индексы, в той последовательности, которая
необходима при самой нижней итерации при работе с БПФ. Индексы для того, чтобы
правильно расположить дискретные данные для БПФ.

Проводить перестановку будем согласно значению MinIteration..
Этот массив позволит перемешивать массив дискретных данных один раз,
а не при каждом вызове fft функции..
См более детально в доках по БПФ прореживание по времени}
function GetArrayIndex(CountPoint: Integer; MinIteration: Integer):
TIndexArray;
Var I,LenBlock,HalfBlock,HalfBlock_1,
    StartIndex,EndIndex,ChIndex,NChIndex :Integer;
    TempArray:TIndexArray;
begin

  EndIndex:=CountPoint-1;

  SetLength(Result,CountPoint);
    For I:=0 to EndIndex do           //Располагаем индексы по порядку
      Result[I]:=I;


  LenBlock   :=CountPoint;
  HalfBlock  :=LenBlock shr 1;
```

```
  HalfBlock_1:=HalfBlock -1;


  while LenBlock > MinIteration do    //переставляем индексы в блоках длинной
LenBlock
    begin
      StartIndex:=0;                  //начинаем с крайнего левого блока
      TempArray :=Copy(Result,0,CountPoint);

      repeat //переставляем индексы в конкретном(начало блока =StartIndex)
             блоке длинной  LenBlock
        ChIndex :=StartIndex;
        NChIndex:=ChIndex+1;
        EndIndex:=StartIndex+HalfBlock_1;

        //Выделяем четные и нечетные индексы и помещаем обратно в исходный
        //массив
        for I:=StartIndex to EndIndex do
          begin
            Result[I]           :=TempArray[ChIndex];
            Result[I+HalfBlock]:=TempArray[NChIndex];

            ChIndex :=ChIndex +2;
            NChIndex:=NChIndex+2;
          end;

        StartIndex:=StartIndex+LenBlock;   //переходим к другому блоку
      Until StartIndex >= CountPoint;      //если дошли до длины массива,
                                           //значит блоков больше нет.

  //уменьшаем длину блок, пока не дойдем до минимального размера (MinIteration)
      LenBlock   :=LenBlock shr 1;
      HalfBlock  :=LenBlock shr 1;
      HalfBlock_1:=HalfBlock -  1;
    end;


  TempArray:=Nil;
end;
```

Мы уже рассматривали с Вами вопрос нормализации данных полученных при помощи ДПФ и БПФ. Теперь мы введем две функции нормализации **NormalizeFFTSpectr**, одна из которых будет являться «стандартной» в нашем понимании (нормализация на N/2), а другая универсальная с возможностью задать любой множитель нормализации (листинг 5.1.3)

Листинг 5.1.3

```
{-------------------------------------------------------------------------
procedure  NormalizeFFTSpectr(var FFTArray: TCmxArray);
Процедура стандартной нормализации (выделение амплитуды)
яв-ся интерфейсом для вызова функции
NormalizeFFTSpectr(var FFTArray: TCmxArray; Mnogitel: Double);

Нормализация имеет вид:  FFTArray[I].Re:=abs(FFTArray[I])/(N/2);
что соответствует, выделению амплитуды сигнала...

Например в MathCad идет нормализация на 1/sqrt(N),
```

```
что не яв-ся амплитудой сигнала.
Где N = Length(FFTArray)}
procedure NormalizeFFTSpectr(var FFTArray: TCmxArray);
begin
   NormalizeFFTSpectr(FFTArray, 2/Length(FFTArray));
end;

{-------------------------------------------------------------------------
procedure NormalizeFFTSpectr(var FFTArray: TCmxArray; Mnogitel: Double);

Процедура Нормализации вектора данных, полученных при помощи БПФ,
после обработки массив будет иметь только реальные значения в первой половине
, вторая половина массива не обрабатывается она яв-ся зеркальным
отражением первой половины массива FFTArray.

Нормализация имеет вид:  FFTArray[I].Re:=abs(FFTArray[I])*Mnogitel;
что дает возможность менять множитель и проводить нужную нормализацию}
procedure NormalizeFFTSpectr(var FFTArray: TCmxArray; Mnogitel: Double);
var I,Len:Integer;
begin
    Len:= Length(FFTArray);
    Len:= Len shr 1;        //Len := Len/2;
    Dec(Len);               //т.к индексация с нуля

    For I:=0 to Len do
      FFTArray[I].Re:=abs(FFTArray[I])*Mnogitel;

end;
```

Ну вот мы и дошли до функции БПФ (листинг 5.1.4). Функция **FFT** является интерфейсом для вызова низкоуровневой подпрограммы **_fft**, которая и рассчитывает БПФ.

Листинг 5.1.4

```
{-------------------------------------------------------------------------
procedure _fft(var D: TCmxArray; StartIndex, ALen: Integer;
               const TableExp:TCmxArray);

Рекурсивный метод БПФ (прореживание по времени), над данными D,
которые должны быть расположены согласно нижней итерации БПФ,
для этого есть проц GetArrayIndex, которая вернет массив, в котором расположены
индексы 0..Length(D), в той последовательности которая нужна.
В нашем случае в процедуру  GetArrayIndex нужно передать MinIteration = 2

StartIndex - должен при первом входе равняться 0 (нулю)
ALen       - Должна быть равна Length(D)
TableExp   - массив Exp множителей для БПФ, созданный GetFFTExpTable

После отработки, массив D, будет содержать БПФ входного массива D
Т.е преобразование проходит на месте, без создания копий}
procedure _fft(var D: TCmxArray; StartIndex, ALen: Integer;
               const TableExp:TCmxArray);
var
    I,NChIndex,TableExpIndex:Integer;
    TempBn,D0,D1:TComplex;
begin

  if ALen=2 then     //Достигли минимальной итерации
```

```
    begin
      D0              :=D[StartIndex];
      D1              :=D[StartIndex+1];
      D[StartIndex]   :=D0+D1;
      D[StartIndex+1] :=D0-D1;
      exit;
    end;

  ALen    := ALen shr 1;
  NChIndex:= StartIndex+ALen;

  _fft(D,StartIndex,ALen,TableExp); //Рекурсия БПФ, для первой половины данных
  _fft(D,NChIndex,ALen,TableExp);   //Рекурсия БПФ, для второй половины данных


  TableExpIndex:=ALen;
  Dec(ALen);
  for I:=0 to ALen do//Далее преобразование Бабочки БПФ (сшиваем две половинки)
    begin
      TempBn        :=D[NChIndex]*TableExp[TableExpIndex+I];
      D0            :=D[StartIndex];
      D[StartIndex] :=D0+TempBn;
      D[NChIndex]   :=D0-TempBn;

      Inc(NChIndex);
      Inc(StartIndex);
    end;
end;

{-------------------------------------------------------------------------
procedure FFT(var D: TCmxArray; const TableExp: TCmxArray);

Процедура служит роль некого интерфейса, для вызова более
низкоуровневой процедуры _fft, которая и будет производить
расчет БПФ, над данным D.

Входные параметры: Смотри процедуру _fft.}
procedure FFT(var D: TCmxArray; const TableExp: TCmxArray);
begin
   _fft(D,0,Length(D),TableExp);
end;
```

Я думаю, достаточно все прозрачно. Большое количество комментариев позволяет мне опустить объяснения того, что мы делали в той или иной строчке кода.

Общие замечания:
передав данные в функцию расчета БПФ, мы проверяем дошли ли мы до нижней итерации (в данном случае до длины блока равной 2), если да то рассчитываем тривиальный случай и выходим. Если длина блока превосходит 2, мы отдельно «рекурсивно» вызываем для каждой половинки блока функцию **_fft**. После того как мы дошли таким образом до минимальной итерации для каждого блока, мы начинаем раскручивать рекурсию вверх. На этом этапе мы сшиваем две половинки в один блок согласно формулам 26. Чтобы более четко себе представить эти манипуляции обратитесь к *рис. 4.2.1.*

Исходники описанного выше проекта находятся в папке:
Test_fft\FFT\FreePascal\NO_U\

На рис. 5.1.1 изображен процесс тестирования.

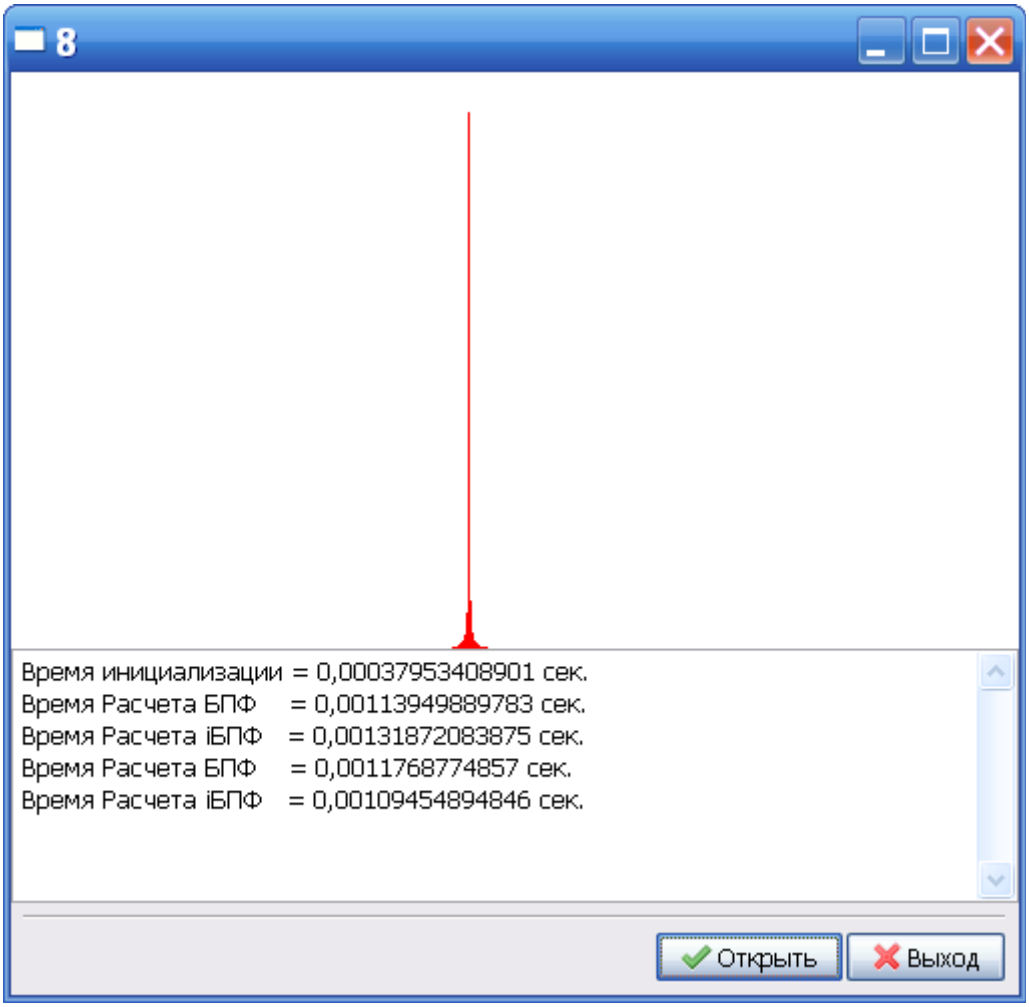


Рис. 5.1.1 Скриншот программы для тестирования нашей реализаций БПФ, при расчете данных из файла 4096Sampls_10KHz_32000.txt.

Обратите внимание на время расчета БПФ, при этом время инициализации - есть время расчета массива exp множителей и массива индексов. Мы не будет более детально рассматривать каждую строчку кода, большое количество комментариев делают код прозрачным, что касается самого алгоритма, то если он вам не достаточно понятен, то вы можете обратиться к дополнительной литературе.

5.1.1 Оптимизация рекурсивного метода на языке Free Pascal №1

Проведем маленькую оптимизацию подпрограммы _fft. (листинг 5.1.1.1)

Листинг 5.1.1.1.

```
procedure _fft(var D: TCmxArray; StartIndex, ALen: Integer;
               const TableExp:TCmxArray);
var
    I,NChIndex,TableExpIndex:Integer;
    TempBn,D1:TComplex;
begin

  if ALen=2 then      //Достигли минимальной итерации
    begin
      NChIndex           :=StartIndex+1;
      D1                 :=D[NChIndex];
      D[NChIndex].Re     :=D[StartIndex].Re-D1.Re;
      D[StartIndex].Re   :=D[StartIndex].Re+D1.Re;
      exit;
    end;

  ALen    := ALen shr 1;
  NChIndex:= StartIndex+ALen;

  _fft(D,StartIndex,ALen,TableExp); //Рекурсия БПФ, для первой половины данных
  _fft(D,NChIndex,ALen,TableExp);   //Рекурсия БПФ, для второй половины данных


TableExpIndex:=ALen;
  Dec(ALen);
  for I:=0 to ALen do  //Далее преобразование Бабочки БПФ (сшиваем две
половинки)
    begin
      TempBn         :=D[NChIndex]*TableExp[TableExpIndex+I];
      D[NChIndex]    :=D[StartIndex]-TempBn;
      D[StartIndex]  :=D[StartIndex]+TempBn;

      Inc(NChIndex);
      Inc(StartIndex);
    end;
end;
```

Как видно, мы избавились от переменной D0, и оптимизировали процесс присвоения.

5.2 Итеративный метод

Рекурсивный метод, хорош тем, что он более понятен и логичен. Но все таки это рекурсия, а как известно рекурсивные методы могут привести к переполнению стека, или как быть, если мы не имеем того размера стека, который нам необходим скажем для расчета БПФ при N=4096 или еще большем значении N (скажем для анализа СВЧ)? Для этого мы перепишем наш алгоритм БПФ с использованием только одних циклов.

5.2.1 Итеративный метод, вариант №1

Так исторически сложилось, что я одновременно разрабатывал итеративный и рекурсивный метод, из-за чего некоторые оптимизации одинаковы и там и там. Из-за чего мы рассмотрим с вами уже оптимизированный вариант (оптимизация O1) (листинг 5.2.1.1)

Листинг 5.2.1.1

```
procedure FFT(var D: TCmxArray; const TableExp: TCmxArray);
var
    I,J,NChIndex,TableExpIndex,Len,StartIndex,LenBlock,HalfBlock:Integer;
    TempBn:TComplex;
begin

  Len           := Length(D);
  LenBlock      := 2;
  HalfBlock     := 1;

  While LenBlock <= Len do //Пробегаем блоки от 2 до  Len
    begin
      I:=0;
      NChIndex  := HalfBlock;
      StartIndex:= 0;

      TableExpIndex := HalfBlock;
      Dec(HalfBlock);

      while I<Len do  //Пробегаем блоки
        begin

          for J:=0 to HalfBlock do //Работаем в конкретном блоке
            begin
              //(Бабочка БПФ)
              TempBn         :=D[NChIndex]*TableExp[TableExpIndex+J];
              D[NChIndex]    :=D[StartIndex]-TempBn;
              D[StartIndex]  :=D[StartIndex]+TempBn;


              Inc(NChIndex);
              Inc(StartIndex);
            end;

          I          := I + LenBlock;
          NChIndex   := I + HalfBlock+1;
```

```
          StartIndex:= I;
        end;

      HalfBlock := LenBlock;
      LenBlock  := LenBlock shl 1;
    end;
end;
```

Исходники проекта находятся в папке: FFT\FreePascal\O1_U\Iterative_fft_FP\

5.2.2 Итеративный метод, вариант №2

Мною было реализовано два варианта итеративного расчета БПФ, в листинге 5.2.2.1, представлен второй вариант (см директорию FFT\FreePascal\O1_U\Iterative_fft_FP_2).
Разница между первым и вторым вариантом, заключается во внутренних циклах, а именно: в первом варианте мы пробегаем блоки от первого до конечного и в цикле for работаем с каким-то конкретным блоком, тогда как вариант два, работает немного по-другому. Во втором варианте мы не пробегаем как бы каждый блок по отдельности, а пробегаем сначала первый элемент **каждого** блока, потом второй элемент **каждого** блока и т. д. После того как дошли до последнего элемента блока, увеличиваем размер блока, пока не достигнем N.

Листинг 5.2.2.1

```
procedure FFT(var D: TCmxArray; const TableExp: TCmxArray);
var
    I,J,NChIndex,TableExpIndex,Len,StartIndex,LenBlock,HalfBlock:Integer;
    TempBn:TComplex;
begin

  Len           := Length(D);
  LenBlock      := 2;
  HalfBlock     := 1;


  while LenBlock <= Len do  //Пробегаем блоки, начинаем с блока длинной =2
    begin

      TableExpIndex:= HalfBlock;
      J:=0;
      while J < HalfBlock do
        begin

          I:=0;
          while I<Len do    //Бабочка БПФ
            begin
              StartIndex    := I+J;
              NChIndex      := StartIndex+HalfBlock;

              TempBn        := D[NChIndex]*TableExp[TableExpIndex];
              D[NChIndex]   := D[StartIndex]-TempBn;
```

```
              D[StartIndex]:= D[StartIndex]+TempBn;


              I             := I+LenBlock;
            end;

          Inc(TableExpIndex);
          Inc(J);
        end;

      HalfBlock := LenBlock;
      LenBlock  := LenBlock shl 1;
    end;
end;
```

В директории FFT\FreePascal\NO_U\ находится не оптимизированная версия, если интересно можете сравнить оба варианта.

6 Оптимизация при помощи встроенного ассемблера языка Free Pascal

Изначально я хотел достаточно подробно описать каждый шаг оптимизации, но передумал, я достаточно много пишу комментариев по коду (иногда это раздражает некоторых). Но это позволит вам достаточно легко понять весь мой код. Ведь даже мне это помогает вспомнить, что я написал :)

Некоторые важные моменты:

1. В одной из тем форума [11] обсуждалось отличие динамических массивов Delphi и Free Pascal. На самом деле отличие сводится к тому, что Free Pascal по смещению -4 относительно первого элемента хранит значение High, тогда как Delphi хранит значение Length массива.
2. Во Free Pascal я так и не смог понять каким образом выравнивать элементы динамического массива по адресам кратным 16. Для статических массивов выравнивание на 16 вроде бы стоит по умолчанию. Эта проблема обсуждались в этих темах форума [12, 13]. Данное выравнивание необходимо для более быстрых команд SSEx, требующих как раз такое выравнивание адресов. В теме [12] я отписался, что во время запуска программы под IDE в регистрах мы наблюдаем адреса кратные 16, однако если запустить программу вне IDE при расчете БПФ происходит вылет, из-за чего я создал тему [13]. Складывается такое впечатление, что при работе под IDE с отладчиком все хорошо вне IDE нет, и именно с командами типа MOVaPD.
3. Если вам реально интересны мои попытки оптимизации при помощи встроенного ассемблера, то вы можете проследить шаги оптимизаций согласно правил именования директорий (см приложение 1)
4. Можно провести еще ряд оптимизаций, таких как избавится от 4-го параметра, воспользовавшись тем, что массив это указатель на первый элемент... тем самым перейти полностью на указатели и забыть про параметр StartIndex.
5. Я не успел рассмотреть в качестве нижней итерации длину блока равной 4, что может принести навскидку минимум 5-7% производительности.
6. При расчете нижних блоков, я не рассчитываю мнимую часть, это связано с тем, что наш метод предназначен для действительных данных, а так как мы с вами помним, что для нижней итерации множитель $W_N^k = 1$ то мнимых частей на данном этапе просто нет.

7 Программа расчета спектра Wav файла

Достаточно трудно придумать программу, в которой бы использовалось БПФ и это использование или точнее результат этой программы был интересен практически каждому. Одним был бы интересен спектр скажем СВЧ, другим спектр цветов в картинке. При этом для каждого такого примера нужны будут данные для получения которых нужно затратить некоторое время. Поэтому было принято решение использовать нашу реализацию БПФ для расчета спектра музыкального файла. Ведь все мы пользуемся торрентами и замечали, что для подтверждения подлинности CD диска приводятся те самые спектры сигнала. Которые могут сказать о присутствии тех или иных частот в сигнале. Такие же спектры приводятся чтобы доказать, что Flac был получен при сжатии с диска, а не пережат с mp3 (так называемый фарш).

При этом мы упростим себе задачу, рассмотрев только wav файлы с одной звуковой дорожкой (mono файлы). Все «уважающие» себя звуковые редакторы имеют возможность построения спектрограмм. Давайте рассчитаем спектр для mono wav файла при помощи программы Audacity рис. 7.1

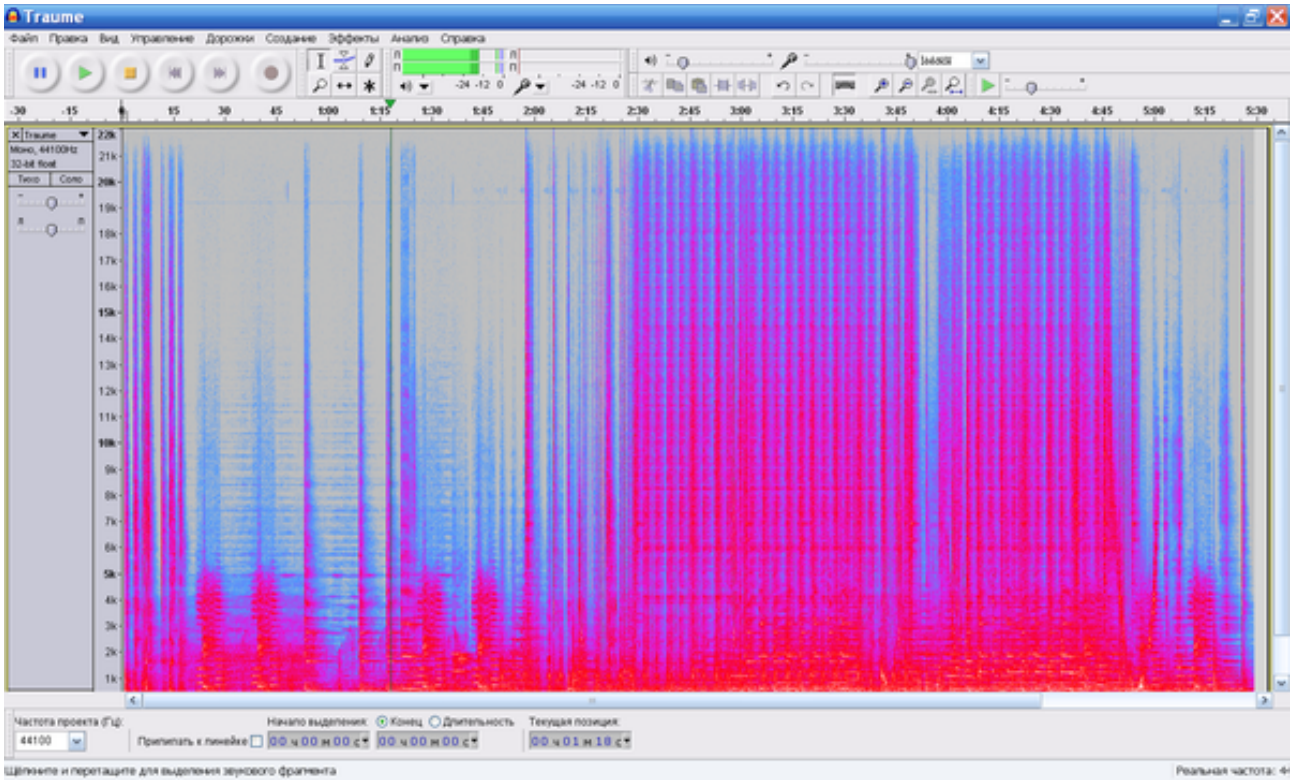


Рис. 7.1 Спектр звукового файла (Traume.wav), рассчитанный в программе Audacity.

Как можно заметить из рис. 7.1 композиция не имеет срезов по частоте, что свидетельствует о том, что это «хороший» рип с диска. При этом видно, что редактор

Audacity даже чуть-чуть «напридумывал» файл имеет частоту дискретизации 44100Гц, то есть максимальная частота не может превышаться 21050 Гц.

В противовес этому спектру приведем спектр wav файла полученного при распаковки из mp3 рис. 7.2.

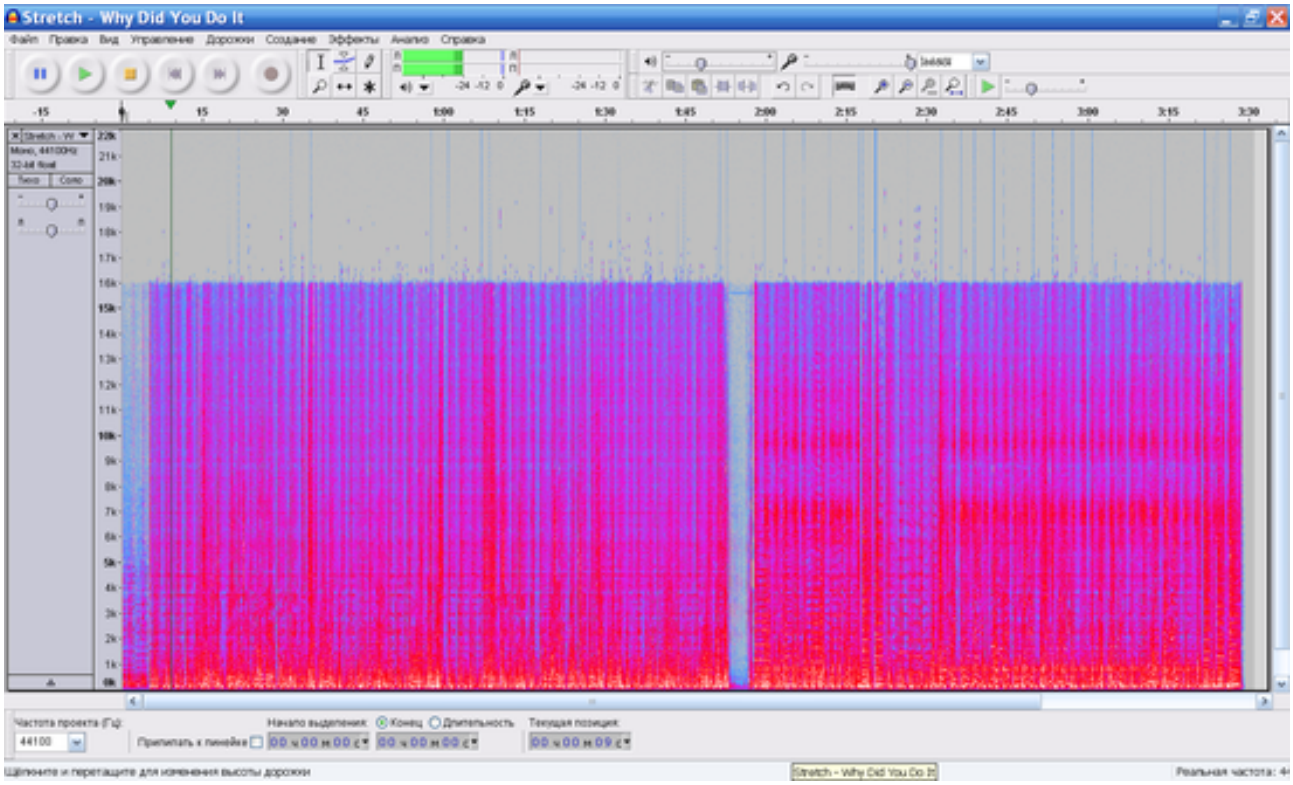


Рис. 7.2 Спектр звукового файла (Stretch - Why Did You Do It.wav), рассчитанный в программе Audacity.

Рис. 7.2 показывает типичный спектр звукового файла, который был сжат при помощи mp3. Отчетливо виден грубый срез по частоте 16КГц. Теперь вы понимаете, почему купив более качественную аудиосистему, вы начинаете слышать недостатки mp3 формата.

Так же заметно, что спектр построен из прямоугольников, в данном случае высота прямоугольника — это разрешение по частоте, а ширина — разрешение по времени.

Вот перед нами и стоит задача, получить такие же картинки спектров, на самом деле всю сложную работу мы уже сделали, осталось приспособить подпрограммы расчета БПФ для расчета спектра wav файла. Самым сложным в данном случае является понять формат хранения данных в wav файлах, рассмотрение которого выходит за рамки данной работы.

Пример проекта для расчета спектра mono wav файла при помощи нашей реализации БПФ расположен в директории WavSpectr_ASM_O4\.

Последовательность работы программы следующая: мы открываем диалог выбора wav файла, после чего проходит проверка на «правильность» файла.

После того, как мы выбрали правильный файл, мы рассчитываем размер картинки. Далее мы начинаем цикл расчета спектра для выбранной длины окна (Fcount), после расчета очередного окна, мы проводим нормализацию и передаем полученные данные в подпрограмму Draw, которая рисует нужную нам картинку. Для получения очередных данных из открытого wav файла используется процедура ReadNextSempls. Что может показаться необычным, так это функция AtoColor, которая предназначена для разделения по цвету различных амплитуд, если этого не делать мы получим монотонный спектр. По простому, функция AtoColor предназначена для того, чтобы более высокие амплитуды имели более яркие цвета. Реализация функции AtoColor мной написано если так можно сказать «в лоб», по этому она далека от правильного варианта. Если вам интересно, можете изменить эту функцию и посмотреть что изменится.

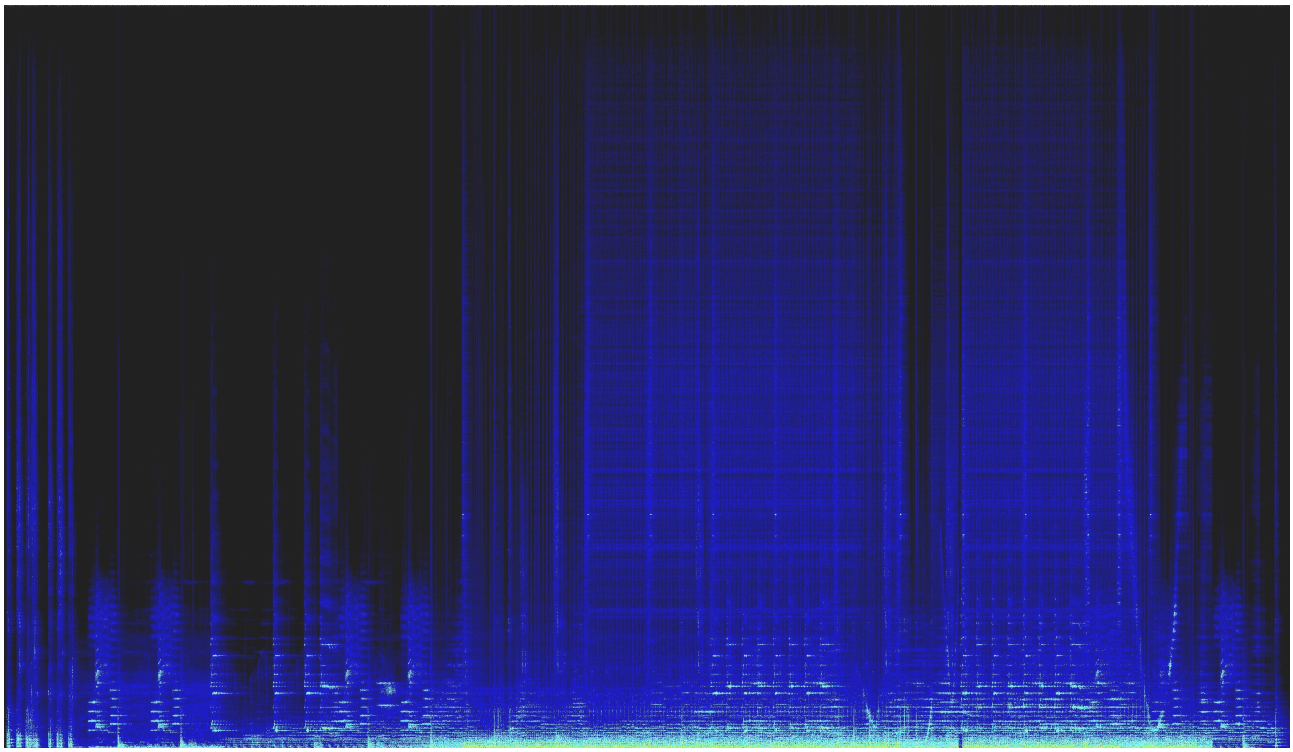


Рис. 7.3 Спектр звукового файла (Traume.wav), рассчитанный в нашей программе.

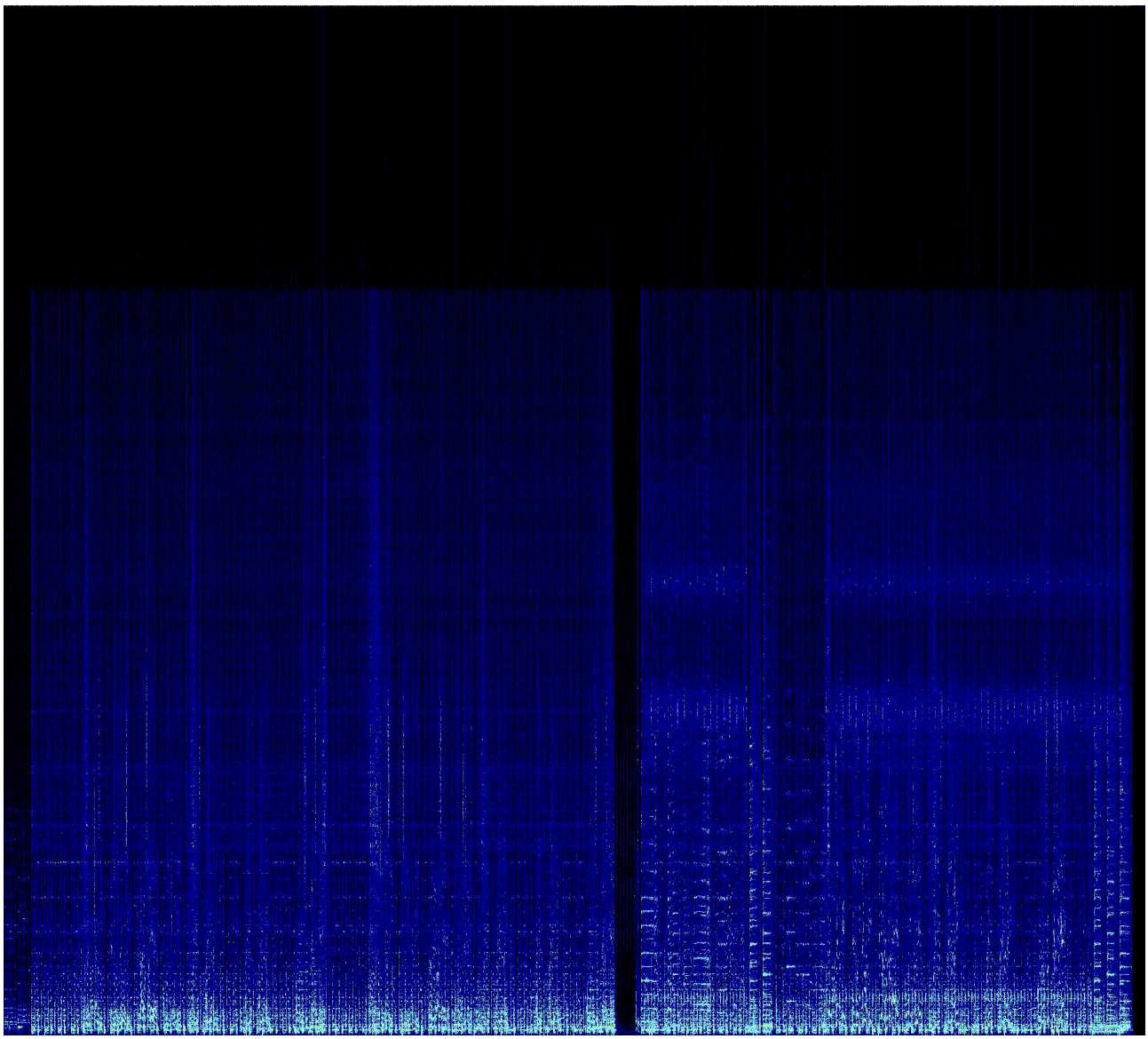


Рис. 7.4 Спектр звукового файла (Stretch - Why Did You Do It.wav), рассчитанный в нашей программе.

Функция AtoColor отвечает не только за выделение больших амплитуд, но и за всю цветовую гамму спектра. Меняя величину сдвига в последней строчке функции, мы можем менять цветовую гамму. Например на рис. 7.5 изображен спектр построенный при помощи программы Audacity для файла (04 - High Roller.wav), а на рис. 7.6 представлены спектры полученные при помощи нашей программы в различных цветовых гаммах.

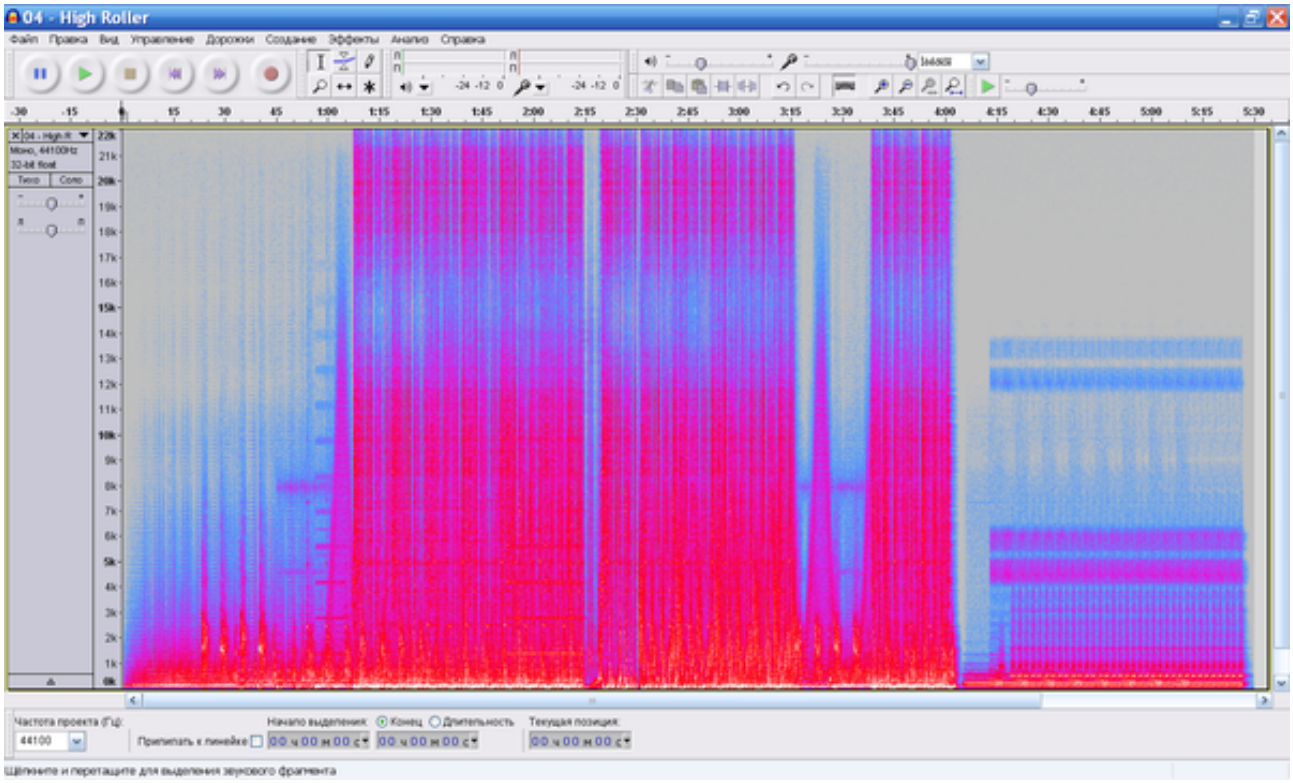
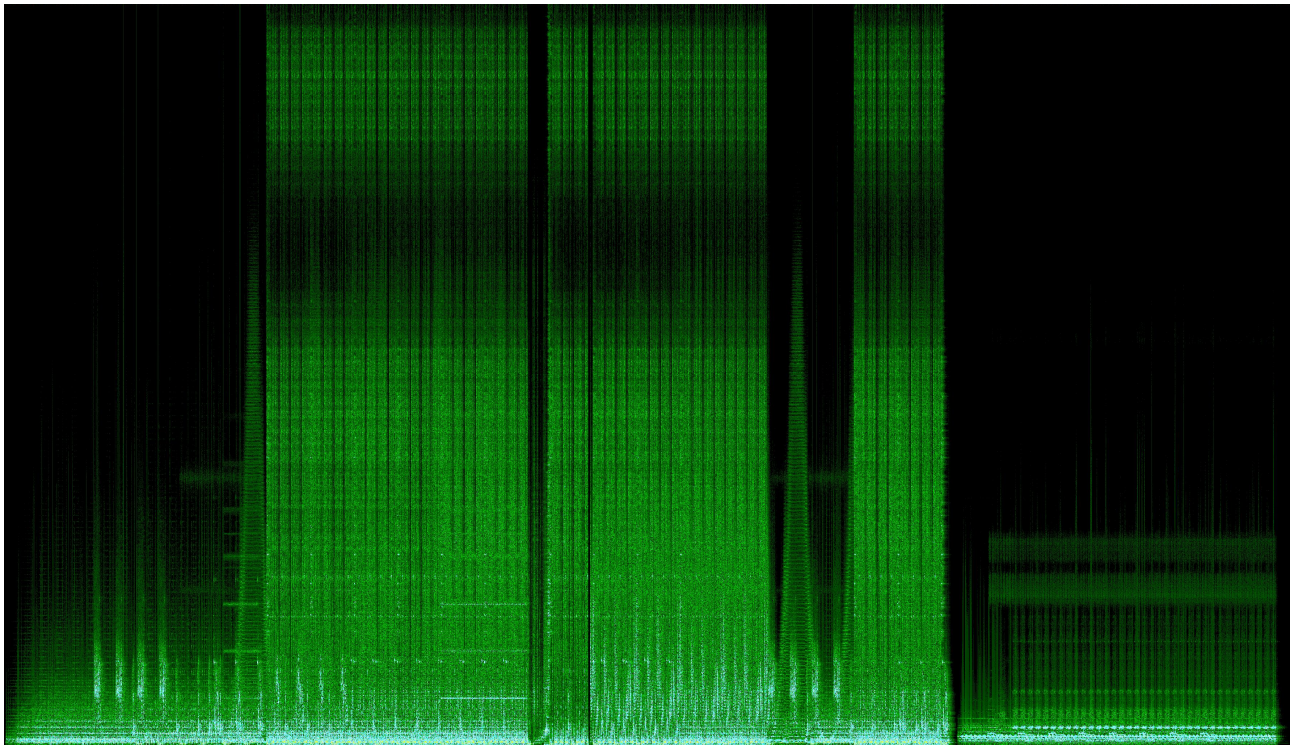


Рис. 7.5 Спектр звукового файла (04 - High Roller.wav), рассчитанный в программе Audacity.



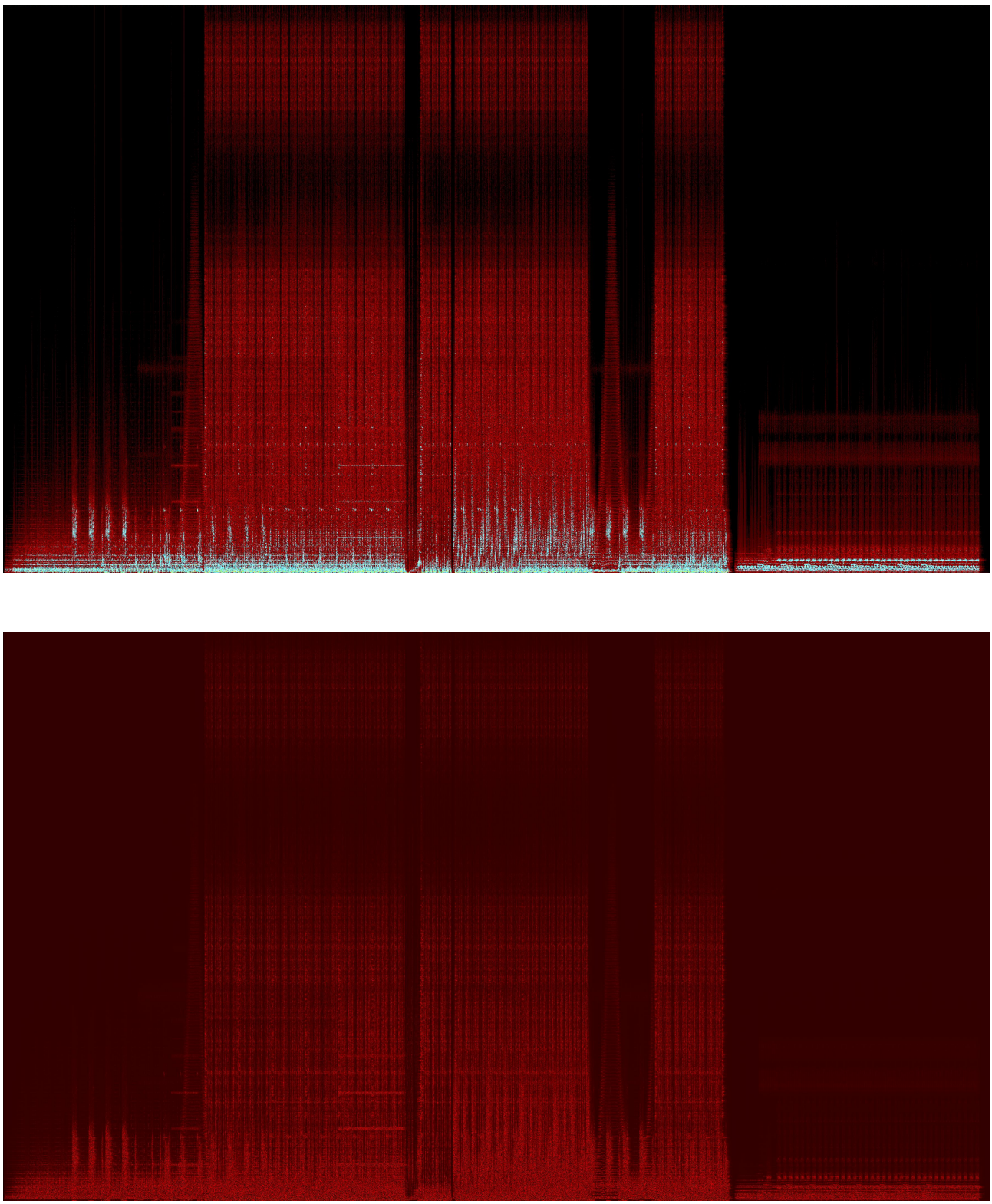


Рис. 7.6 Спектр звукового файла (04 - High Roller.wav), рассчитанный в нашей программе, при различных параметрах в функции AtoColor.

На моем ПК, расчет данных спектров занимает порядка 7-8 секунд, а теперь сравните это время со временем расчета ДПФ для одного окна. При этом 80% времени занимает процедура Draw. На этом положительном результате мы закончим эту главу.

8 Заключение

В работе была рассмотрена одна из возможных реализаций алгоритма БПФ на языке Free Pascal. Был проведен вывод формул БПФ из ДПФ, была предпринята попытка доходчиво объяснить ДПФ и БПФ, как они работают и что они могут посчитать. Приведен ряд возможный оптимизаций на языке встроенного ассемблера с использованием как простого сопроцессора FPU, так и набора команд SSE, SSE2, SSE3. Так же приведена программа расчета спектрограмм звуковых файлов, которая может послужить руководством по применению разработанных нами модулей расчета БПФ.

В заключении я приведу сравнительные диаграммы, в различных сочетаниях, которые наглядно покажут были ли удачны наши шаги по оптимизации.

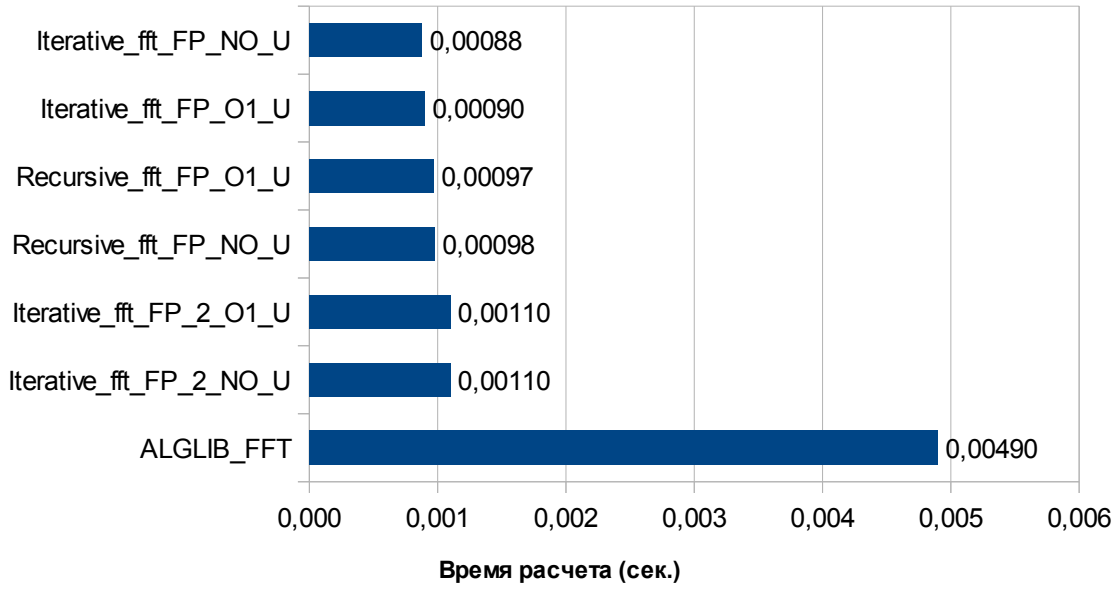


Рис. 8.1 Временная диаграмма расчета спектра для данных (4096Sampls_20KHz_15000.txt). Для сравнения нашей реализации БПФ на языке Free Pascal и библиотеки ALGLIB

Как можно было заметить из рис. 8.1, мы превзошли библиотеку ALGLIB ver. 2.6.0, которая также написана на языке Free Pascal.

Но в мире большой популярностью пользуется библиотека FFTW, которая написана на языке Си. В директории Andere\FFTW_xx располагаются проекты для Lazarus, которые для расчета спектра используют функцию БПФ данной библиотеки версии xx.

Сравнение библиотеки FFTW и наших реализаций на языке Free Pascal, можно увидеть на рис. 8.2.

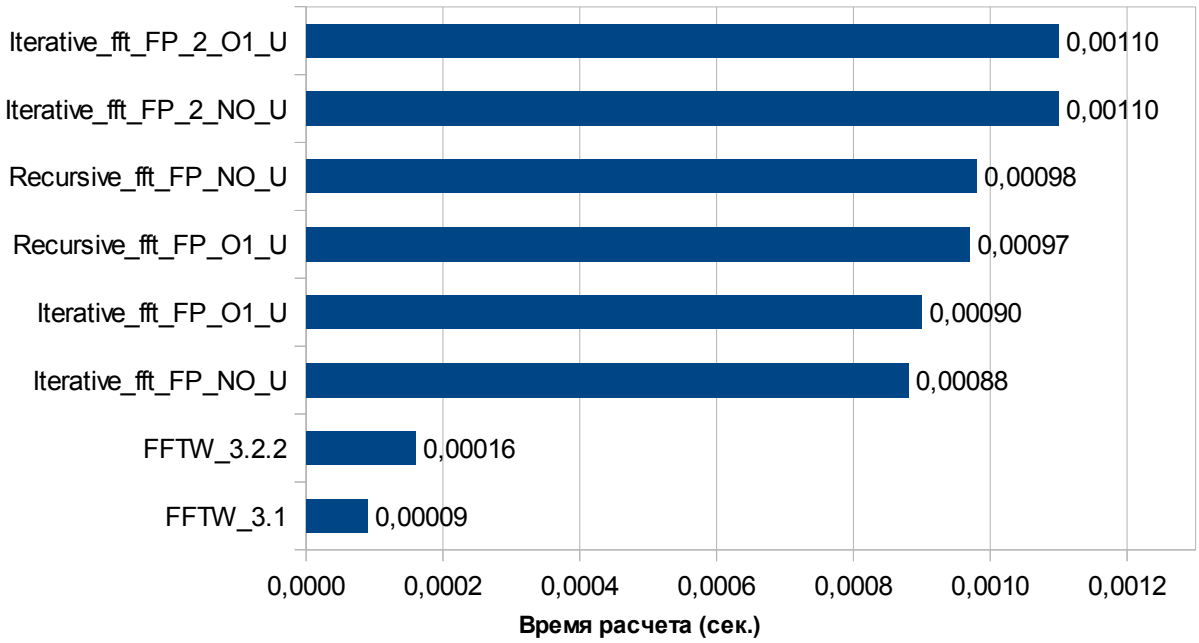


Рис. 8.2 Временная диаграмма расчета спектра для данных (4096Sampls_20KHz_15000.txt). Для сравнения нашей реализации БПФ на языке Free Pascal и библиотеки FFTW

Из рис. 8.2 видно, что наша реализация БПФ на языке Free Pascal отстает от библиотеки FFTW, при это очень забавно, что более новая версия FFTW (ver 3.2.2) считает медленнее чем старая (ver 3.1) возможно это связано с тем, что новая версия требует более новый процессор.

Понятно, что наши реализации БПФ на языке Free Pascal не способны соперничать с библиотекой FFTW написанной на Си с использованием SSE команд. На рис. 8.3 представлена временная диаграмма, которая показывает, что использование SSE команд позволяет нам обойти библиотеку FFTW ver 3.2.2.

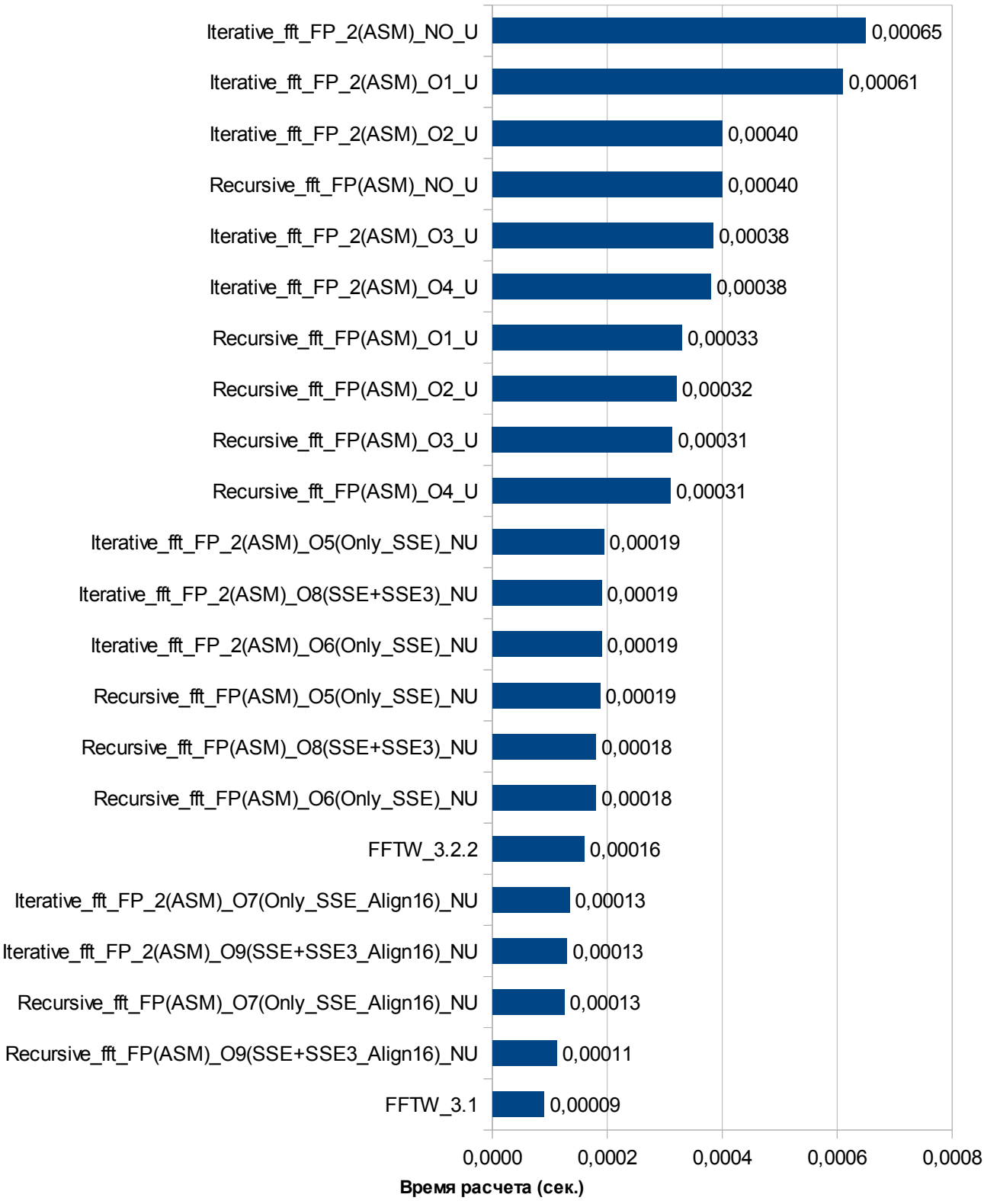


Рис. 8.3 Временная диаграмма расчета спектра для данных (4096Sampls_20KHz_15000.txt). Для сравнения нашей реализации БПФ на языке Free Pascal с оптимизацией и библиотеки FFTW

Как говорится: «без комментариев», мы превзошли новую версию библиотеки FFTW, но не смогли покорить вершину 0.00009 секунды. Так же заметно, что

выравнивание данных и набор команд SSE это круто :). К сожалению, на момент написания и реализации соответствующих подпрограмм, во Free Pascal непонятным образом работают команды SSE требующих выравнивание на 16 байт [12,13].

Приложение 1

Сокращения принятые при именовании директорий (папок)

None Optimization and None Universal = NO_NU
None Optimization and Universal = NO_U
Optimization and Universal = O_U
Optimization №X and Universal = OX_U

Universal - возможность работать с комплексными числами, у которых Re,Im могут иметь тип Single\Double, либо изменением данного типа непосредственно в модуле Complex, либо при использовании директивы компилятора **ComplexIsSingle.** Если директория имеет обозначения NU, то реализация жестко завязана на размере Re, Im части и изменение типа может привести к краху системы.

Содержимое архива, прилагаемого к данной работе:

DFT/DFT_NO	Директория содержит проект и исходные коды для IDE Lazarus 0.9.30 и выше (в дальнейшем просто **проект**), программы реализующей расчет спектра при помощи ДПФ.
DFT/DFT_NO_Hamming_window	Проект программы реализующей расчет спектра при помощи ДПФ с использованием сглаживающей оконной функцией, на примере оконной функции Хемминга.
DFT/DFT_O1	Директория содержит проекты реализации ДПФ на языке Free Pascal. оптимизация №1
Andere	Директория содержит проекты в которых реализован расчет БПФ при помощи различных сторонних библиотек таких как FFTW и ALGLIB
FFT\FreePascal	Директория содержит проекты реализации БПФ на языке Free Pascal
FFT\FreePascal_ASM	Директория содержит проекты реализации БПФ на языке Free Pascal с оптимизацией на ассемблере
Doc	Директория содержит дополнительную литературу, а так же литературу, которая использовалась при написании этой работы
WavSpectr	Директория содержит проекты программ построения спектра звукового файла в формате WAV PCM Mono 16 bit per sempl.
Sempls_and_wav	Директория содержит различные файлы необходимые для проведения тестов такие как::
4096Sampls_10KHz_32000.txt	Файл содержит в текстовом виде сэмплы чисто синусоидального звука, с частотой дискретизации 44100Гц, частотой колебания 10КГц, и амплитудой 32000
4096Sampls_20KHz_15000	Файл содержит в текстовом виде сэмплы чисто синусоидального звука, с частотой дискретизации 44100Гц, частотой колебания 20КГц, и амплитудой 15000
***.wav**	Mono Wav файлы различной длительности и частоты сигнала.

Литература

1. Дмитрий Михайлов «FFT анализ»// http://www.websound.ru/articles/theory/fft.htm E-mail автора: dmitry.mih@mtu-net.ru
2. Серегин Н.И «Особенности использования дискретного преобразования Фурье при спектральном анализе»// Екатеринбург. 2006. 36 с.
3. http://www.dsplib.ru/content/fft/fft.html
4. http://www.dsplib.ru/content/dft/dft.html
5. http://www.fftw.org/
6. Кантор Илья «Эффективное вычисление дискретного преобразования Фурье и дискретного преобразования Хартли»// 2002. algolist@manual.ru
7. http://ru.wikipedia.org/wiki/Быстрое_преобразование_Фурье
8. http://ru.wikipedia.org/wiki/Дискретное_преобразование_Фурье
9. Гольденберг Л.М. «Цифровая обработка сигналов»// Справочник. Москва: Радио и связь. 1985. 312 с.
10. http://www.dsplib.ru/content/win/win.html
11. Длина массива, различие в Delphi и FPC (форум) http://www.freepascal.ru/forum/viewtopic.php?f=23&t=6873
12. SUBPD xmm, mem = SIGSEGV (форум) http://www.freepascal.ru/forum/viewtopic.php?f=1&t=7001
13. SSE+SSE3 в IDE работает вне IDE Вылет (форум) http://www.freepascal.ru/forum/viewtopic.php?f=1&t=7047&st=0&sk=t&sd=a