

## Эффективное вычисление дискретного преобразования Фурье и дискретного преобразования Хартли.

### Дискретное преобразование Фурье

Введем класс комплексных чисел с операциями сложения, вычитания, умножения, деления на число и с функцией  $\text{conj}(c)$ , которая возвращает комплексное число, сопряженное аргументу (обозначается  $c^*$ ).

```
class Complex {
public:
    real r, i;
    Complex(void) { }
    Complex(real a, real b) { r=a; i=b; }
    inline const Complex operator+(const Complex &c) const {
        return Complex( r + c.r, i + c.i );
    }
    inline const Complex operator-(const Complex &c) const {
        return Complex( r - c.r, i - c.i );
    }
    inline const Complex operator*(const Complex &c) const {
        return Complex( r*c.r - i*c.i, r*c.i + i*c.r );
    }
    inline const Complex operator/(const real &divisor) const {
        return Complex( r/divisor, i/divisor );
    }
};

inline const Complex conj(const Complex &c) {
    return Complex( c.r, -c.i );
}
```

Дискретное преобразование Фурье (ДПФ)<sup>1</sup> комплексного вектора  $(a_0, a_1, \dots, a_{N-1})$  определяется как комплексный вектор с координатами  $(y_0, y_1, \dots, y_{N-1})$ :

$$y_k = \sum_{j=0}^{N-1} a_j \omega^{kj}$$

где  $\omega$  – главный комплексный корень N-й степени из единицы, т.е

$$\omega = \omega_N = e^{\frac{2\pi}{N}i} = \cos \frac{2\pi}{N} + i \sin \frac{2\pi}{N}$$

Индекс степени N будет часто отсутствовать, в этом случае степень корня полагается равной длине<sup>2</sup> преобразуемого вектора.

Дискретное преобразование Фурье обратимо, причем ДПФ<sup>-1</sup> вычисляется по формуле

$$a_k = \frac{1}{N} \sum_{j=0}^{N-1} y_j \omega^{-kj}$$

Действительно, рассмотрим значение композиции этих преобразований в точке y:

$$\text{ДПФ}^{-1}(\text{ДПФ}(a))_y = \frac{1}{N} \sum_{j=0}^{N-1} \sum_{x=0}^{N-1} (a_x \omega^{xj}) \omega^{-yj} = \frac{1}{N} \sum_x a_x \sum_j (\omega^{x-y})^j$$

---

<sup>1</sup> Существуют альтернативные определения ДПФ, но такое наиболее удобно для вычислений

<sup>2</sup> Под длиной в обсуждении ДПФ принимается количество координат вектора, а не евклидова норма.

Так как  $\omega$  - корень N-й степени из единицы, то  $\sum_{j=0}^{N-1} (\omega^{x-y})^j = N$ , если  $x=y$  и ноль иначе.

Получаем, что все выражение равно

$$\frac{1}{N} N \sum_{x=0}^{N-1} a_x \delta_x^y = a_y, \text{ где } \delta_x^y = \begin{cases} 1, & x = y \\ 0, & x \neq y \end{cases}$$

Простейший способ вычисления ДПФ – прямое суммирование, оно приводит к N операциям на каждый коэффициент  $y_k$ . Всего коэффициентов N, так что общая сложность  $\Theta(N^2)$ . Такой подход не представляет практического интереса, так как существуют гораздо более эффективные способы.

## Быстрое преобразование Фурье

Главная идея быстрого вычисления ДПФ - использование метода “разделяй-и-властвуй”. Вектор делится на части, результаты обработки которых затем сливаются.

Будем рассматривать только векторы длины  $N = 2^s$ , т.е N - степень двойки. Это предположение чрезвычайно важно, так как на нем построена вся логика работы алгоритма.

Разделим общую сумму на две: первая содержит слагаемые с четными индексами, вторая – с нечетными.

$$y_k = \sum_{x=0}^{N-1} a_x \omega_N^{xk} = \sum_{x=0}^{N/2-1} a_{2x} \omega_N^{2xk} + \sum_{x=0}^{N/2-1} a_{2x+1} \omega_N^{(2x+1)k} = \sum_{x=0}^{N/2-1} a_{2x} \omega_N^{2xk} + \omega^k \sum_{x=0}^{N/2-1} a_{2x+1} \omega_N^{2xk} \quad (0)$$

Получившееся равенство дает способ вычислять k-й коэффициент ДПФ вектора длины N через два преобразования длины N/2, одно из которых применяется к вектору  $a^{чет}$  из координат вида  $a_{2x}$ , а другое - к вектору  $a^{нечет}$  из координат вида  $a_{2x+1}$ .

Чтобы полностью выразить N-значное ДПФ в терминах N/2-значного преобразования, рассмотрим два случая.

При  $0 \leq k < N/2$  положим  $k=j$  запишем (0) в виде

$$y_j = \sum_{x=0}^{N/2-1} a_x^{чет} \omega_{N/2}^{xj} + \omega^j \sum_{x=0}^{N/2-1} a_x^{нечет} \omega_{N/2}^{xj} \quad (1)$$

При  $N/2 \leq k < N$  полагаем  $k=j+N/2$ , где  $0 \leq j < N/2$ . Подставляя в (0), получаем

$$y_{j+N/2} = \sum_{x=0}^{N/2-1} a_x \omega_N^{x(j+N/2)} = \sum_{x=0}^{N/2-1} a_x^{чет} \omega_N^{2x(j+N/2)} + \omega_N^{j+N/2} \sum_{x=0}^{N/2-1} a_x^{нечет} \omega_N^{2x(j+N/2)}$$

Учитывая, что по свойствам корня из единицы  $\omega_N^{j+N/2} = -\omega_N^j$ , упрощаем до

$$y_{j+N/2} = \sum_{x=0}^{N/2-1} a_x^{чет} \omega_{N/2}^{xj} - \omega_N^j \sum_{x=0}^{N/2-1} a_x^{нечет} \omega_{N/2}^{xj} \quad (2)$$

Первая сумма является ДПФ вектора, составленного из элементов  $a$ , стоящих на четных местах, вторая сумма – ДПФ вектора элементов с нечетными индексами.

Полученные формулы дают возможность вычислять преобразование длины N путем комбинации двух преобразований длины N/2, при этом левая часть вектора вычисляется по формуле (1), а правая – по формуле (2),  $j=0..N/2-1$ .

Таким образом, операция объединения двух ДПФ в одно занимает  $\Theta(N)$  времени.

Общая схема алгоритма состоит в повторяющемся сведении ДПФ вектора длины N к векторам длины N/2 и объединении результатов. Базисом рекурсии служат векторы длины 1, для которых ДПФ – сам вектор.

Этот алгоритм называется быстрым преобразованием Фурье (БПФ) и имеет сложность  $\Theta(N \log N)$ .

АЛГОРИТМ БПФ(a, N) {

1. Если длина вектора равна 1, вернуть a.
2. Разбить вектор a на четную часть  
 $a^{чет} = (a_0, a_2, \dots, a_{N-2})$  и нечетную  $a^{нечет} = (a_1, a_3, \dots, a_{N-1})$ .
3. Рекурсивно вызвать БПФ на каждой из частей  
 $b^{чет} = \text{БПФ}(a^{чет})$   
 $b^{нечет} = \text{БПФ}(a^{нечет})$
4. Объединение результатов по формулам (1), (2).
  - a. (инициализация) Присвоить  $\omega_N$  значение главного корня из единицы
  - b. (инициализация) Присвоить  $\omega = 1$
  - c. В цикле вычислить левую и правую часть одновременно:

$$\text{for}(j=0; j < N/2; j++) \{$$

$$y_j = b_j^{чет} + \omega b_j^{нечет} \quad // \text{ формула 1}$$

$$y_{j+N/2} = b_j^{чет} - \omega b_j^{нечет} \quad // \text{ формула 2}$$

$$\omega = \omega \omega_N \quad // \text{ переходим к следующей степени корня}$$

$$\}$$

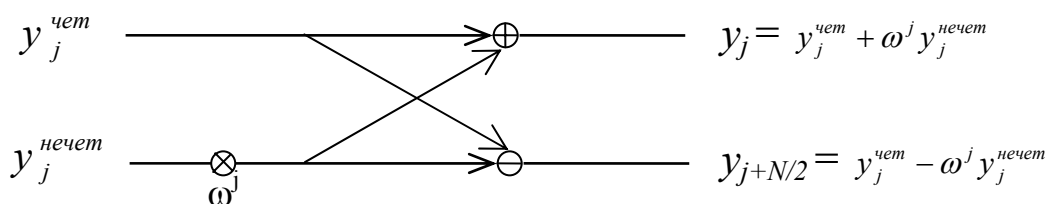
5. вернуть вектор y.
- }

Прежде, чем следовать дальше, сделаем одну небольшую оптимизацию.

На шаге 4 значение  $\omega b_j^{нечет}$  вычисляется два раза подряд. Эффективнее будет закодировать цикл так:

```
for( j=0; j < N/2; j++) {
    temp =  $\omega b_j^{нечет}$ 
     $y_j = b_j^{чет} + \text{temp}$ 
     $y_{j+N/2} = b_j^{чет} - \text{temp}$ 
     $\omega = \omega \omega_N$ 
}
```

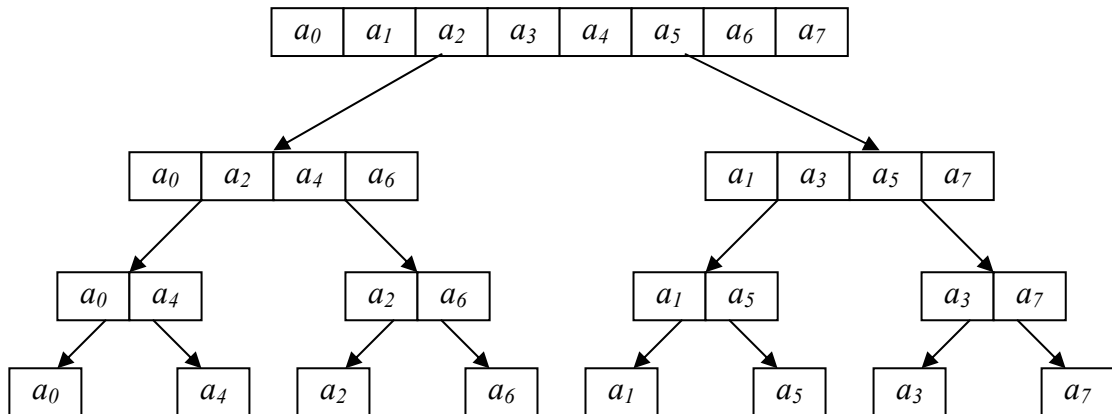
Это преобразование носит название “бабочки БПФ”, которую можно изобразить следующим образом:



Нечетное значение умножается на  $\omega^j$ , затем происходит сложение и вычитание, как показано на рисунке.

Ниже изображено дерево рекурсии. Каждый уровень, начиная снизу, соответствует проходу алгоритма по всему вектору и объединению сначала одиночных элементов в пары, затем пар в

четверки и так далее до конца. Обратите внимание на то, что порядок индексов на верхнем уровне не соответствует нижнему. Это естественно, если учесть, что нечетные индексы после бабочки идут в правую половину вектора, а четные – в левую.



**Дерево рекурсии для 8 элементов**

Преобразование бабочки может происходить “на месте”: после операции объединения участвовавшие в нем элементы больше не нужны, так что на их место можно записать преобразованные данные. Однако, для того, чтобы такая техника работала, на входе алгоритма должен быть вектор нижнего уровня ( $a_0, a_4, a_2, a_6, a_1, a_5, a_3, a_7$ ) – тогда объединения произойдут в правильном порядке, как изображено выше, и дадут в результате вектор ( $a_0, a_1, \dots, a_7$ ). Таким образом, перед вызовом процедуры следует провести перестановку элементов массива.

### Предварительная перестановка элементов вектора

Рассмотрим двоичное представление индексов от 0 до 8:

000<sub>2</sub>, 001<sub>2</sub>, 010<sub>2</sub>, 011<sub>2</sub>, 100<sub>2</sub>, 101<sub>2</sub>, 110<sub>2</sub>, 111<sub>2</sub>

Если записать каждое число задом наперед, до получим

000<sub>2</sub>, 100<sub>2</sub>, 010<sub>2</sub>, 110<sub>2</sub>, 001<sub>2</sub>, 101<sub>2</sub>, 011<sub>2</sub>, 111<sub>2</sub>

Получилась перестановка, в которой часть чисел осталась на месте, а часть – поменялась местами с симметричными: 001<sub>2</sub> ↔ 100<sub>2</sub>, 110<sub>2</sub> ↔ 011<sub>2</sub>.

Перейдя обратно в десятичную запись, получаем необходимую последовательность индексов входного вектора: 0 4 2 6 1 5 3 7.

Формализуя вышесказанное, можно поставить следующую задачу: реорганизовать вектор  $a$  длины  $n(=2^s)$  таким образом, что каждый элемент  $a_x$  поменяется местами с элементом  $a_{x'}$ , где  $x'$  получен из  $x$  обращением двоичной записи.

Заметим, что  $x'$  зависит не только от  $x$ , но и от  $n$ .

При  $n=64$ , например,  $x=6$  записывается как 000110<sub>2</sub>,  $x' = 011000_2 = 24$ .

Простейший подход – использовать функцию

```

reverse_binary( a[], n) {
    for ( x=0; x<n; x++) {
        x' = rev(x, n);
        if ( x'>x ) swap( a[x], a[x'] ); // swap меняет аргументы местами
    }
}
  
```

Здесь условие  $x'>x$  гарантирует, что обратного обмена не случится, а функция  $rev(x, n)$  возвращает обратную двоичную запись  $x$ , последовательно добавляя к ней нужные биты.

```

rev(x, n) {
    int j = 0;
    for ( bits = log2(n); bits > 0; bits-- ) {
        j = j << 1;
        j = j + (x&1);
        x = x >> 1;
    }
}
  
```

```

        return j;
    }

```

Оценим время выполнения процедуры `reverse_binary()`. Функция `rev(x,n)` производит  $\log n$  операций и вызывается  $n/2$  раз. Таким образом, общее время работы имеет порядок  $\Theta(n \log n)$ . Вся процедура выполняется только один раз, значит асимптотика БПФ не изменится. Тем не менее, эффективнее будет слегка ускорить процесс.

Идея состоит в том, чтобы получать `rev(x)` из `rev(x-1)` прибавлением обращенной единицы `rev(1)`. Например, для  $n=64$  обращенная единица `rev(1)=1000002`. Тогда из `rev(6) = 0001102` получаем `rev(7) = 0001102 + rev(1) = 1001102`

```

inline ulong rev_next(ulong r, ulong n) {
// преобразовывает r=rev(x-1) в rev(x)
    do {
        n = n >> 1;
        r = r^n;
    } while ( (r&n) == 0);
    return r;
}

```

Теперь можно написать процедуру перестановки на C++ с применением этой функции.

```

void FFTReOrder(Complex *Data, ulong Len) {
    Complex temp;
    if (Len <= 2) return;
    ulong r=0;
    for ( ulong x=1; x<Len; x++) {
        r = rev_next(r, Len);
        if (r>x) { temp=Data[x]; Data[x]=Data[r]; Data[r]=temp; }
    }
}

```

На достаточно большом векторе `FFTReOrder` производит приблизительно  $2n$  операций.

Напишем рекурсивный алгоритм в окончательном варианте на C++.

При этом для упрощения записи шагов будем использовать обозначения:

`PRoot` – текущая степень корня из единицы

Макрос `TRIG_VARS` – инициализация переменных для вычисления корней

Макрос `INIT_TRIG(Len, Dir)` – инициализация  $\omega$ ,  $\omega_N$  (т.е пункты 4a, 4b псевдокода) будет

Макрос `NEXT_TRIG_POW` - переход к следующей степени корня из единицы.

Дополнительный параметр `Dir` означает направление преобразования: `Dir=1` - прямое, `Dir=-1` – обратное. Единственное место, где он используется – при вычислении корней из единицы, так как при обратном ДПФ следующий корень получается из предыдущего домножением не на  $\omega^1$ , а на сопряженный ему  $\omega^{-1} = \omega^{N-1}$ .

```

// Перед запуском вектор должен быть обработан процедурой FFTReOrder()
void FFT_T(Complex *Data, ulong Len, int Dir) {
    ulong k;

    TRIG_VARS;
                                                                    // (*)
    Len /= 2;

    INIT_TRIG(Len, Dir);

    if (Len > 1) {
        FFT_T(Data, Len, Dir); // пока длина вектора больше единицы
        FFT_T(Data+Len, Len, Dir); // рекурсивно делим на две части
    }

    for (k=0; k<Len; k++) {
        Complex b,c; //
        b=Data[k]; // преобразование
    }
}

```

```

        c = Data[k+Len]*PRoot;      // бабочки "на месте"
        Data[k] = b + c;           //
        Data[k+Len] = b - c;       //

        NEXT_TRIG_POW;             // переход к следующему корню, учитывая Dir
    }
}

```

Если вернуться к формулам и сравнить ДПФ с  $\text{ДПФ}^{-1}$ , можно увидеть, что для обратного преобразовании каждую координату вектора нужно разделить на его длину. Такое деление называется *нормализацией* вектора. В ряде приложений она не требуется: константный множитель не играет роли в вычислениях, промежуточным моментом которых является ДПФ.

Поэтому имеет смысл возвращать *ненормализованный* вектор.

Резюмируя все вышесказанное, вызов БПФ в программе состоит из следующего фрагмента кода:

```

...
FFTReOrder (a, N); // преобразовать вектор в правильный вход алгоритма
FFT_T (a, N, Dir); // Dir=1 - прямое, Dir=-1 - обратное
if (Dir == -1)
    { разделить все координаты на N, если это необходимо для внешней программы }
...

```

## Итеративное БПФ

Рекурсию можно развернуть в итерации, где внешний цикл задает уровень дерева, а значит, и длину объединяемых на текущем шаге частей. Переменная внутреннего цикла первого уровня последовательно переходит от одной пары частей к другой, выполняя для них операцию объединения по формулам (1), (2).

```

АЛГОРИТМ ИТЕРАТИВНОЕ БПФ (Data, Len) {
    FFTReOrder (Data, Len);           // преобразовать вектор в правильный вход алгоритма
    Step=1;
    while (Step < Len) {
        Halfstep = Step;
        Step *= 2;                     // Step - длина объединяемых частей
         $\omega = 1$ ;
         $\omega_{\text{step}} = e^{i\pi / \text{halfstep}}$ ; // главный корень из единицы
        for (j=0; j<Len; j+=Step) {
            for (k=0; k<Halfstep; k++) { // объединить две части Data[j...j+Step/2-1]
                L = j + k;                // и Data[j+Step/2...j+Step-1] в одну
                R = L + Halfstep;
                TRight =  $\omega$ *data[R];     // преобразование
                Data[L] = Data[L] + TRight; // бабочки
                Data[R] = Data[L] - TRight; // БПФ
                 $\omega = \omega \omega_{\text{step}}$ ;
            }
        }
    }
}

```

Чем итеративная версия лучше рекурсивной ? Пока ничем. Однако есть общий принцип, который выполняется при разворачивании рекурсии – это получение большего контроля над процессом. Чем мы и воспользуемся при написании окончательного варианта алгоритма.

```

// до запуска вектор должен быть обработан процедурой FFTReOrder
void IFFT_T(Complex *Data, ulong Len, int Dir) {
    ulong Step, HalfStep;
    ulong b;
    TRIG_VARS;
}

```

```

Step = 1;
while (Step < Len) {
    HalfStep = Step;
    Step *= 2;

    INIT_TRIG(HalfStep,Dir);

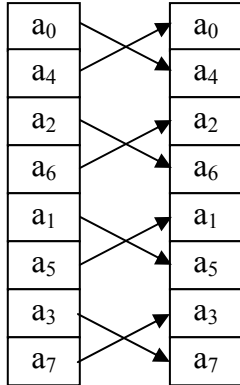
    for (b = 0; b < HalfStep; b++) {
        ulong L,R;
        for (L=b; L<Len; L+=Step) {
            Complex TRight,TLeft;
            R=L+HalfStep;
            TLeft=Data[L];TRight=Data[R];
            TRight = TRight * PRoot;
            Data[L] = TLeft + TRight;
            Data[R] = TLeft - TRight;
        }
        NEXT_TRIG_POW;
    }
}

```

Здесь внутренние циклы поменялись местами. Вычисления следуют дереву рекурсии уровень за уровнем, начиная от нижнего, и заканчивая верхним. Расчеты следуют формулам (1) и (2), однако организованы так, чтобы последовательно выполнялись сначала все операции с  $\omega^0$ , затем с  $\omega^1$  и так до конца.

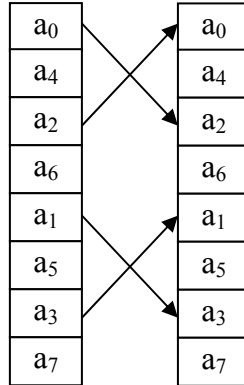
Это позволяет сильно уменьшить количество вычислений корней из единицы, так как в рекурсивной версии они вычисляются при каждой бабочке.

Первый шаг, step=2  
внутренний цикл при j=0

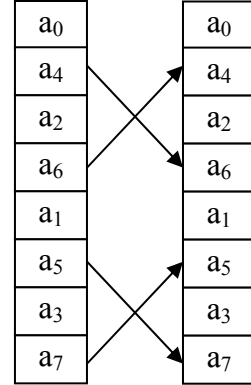


Второй шаг, step=4

j=0

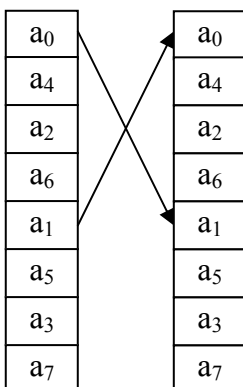


j=1

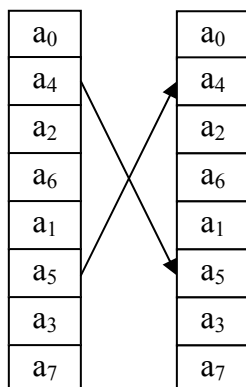


Третий шаг, step=8

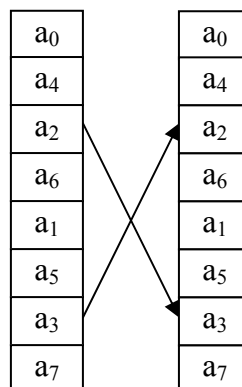
j=0



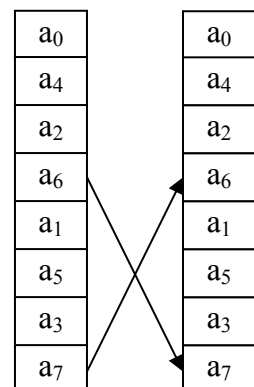
j=1



j=2



j=3



Работа итеративного алгоритма по шагам

Однако, несмотря на эту оптимизацию, на больших векторах итеративный алгоритм гораздо медленнее рекурсивного! Это происходит из-за того, что, как видно из рисунка выше, доступ к памяти осуществляется крайне нелокально. Осуществляются прыжки от одного элемента к другому через расстояния  $\text{Halfstep}$ , что приводит к нерациональному использованию кэша.

Возникшая проблема довольно типична для подхода “разделяй-и-властвуй”. Простейшее решение – использовать рекурсивный алгоритм до определенного момента, затем, когда размер вектора станет меньше размера кэша (или его половины  $\text{CACHE\_HALF}$ ), переключиться на итеративную реализацию.

Для проведения такого подхода в жизнь нужны лишь минимальные изменения в процедуре  $\text{FFT\_T}$ , а именно, на место (\*) после  $\text{INIT\_TRIG}$  следует включить код

```
if ( Len <= (CACHE_HALF/sizeof(Complex) ) ) {          // вместо CACHE_HALF
    IFFT_T(Data, Len, Dir);                             // подставить число
    return;
}
```

Также можно убрать проверку  $\text{if (Len > 1)}$ , так как это условие теперь всегда истинно.

## Точность БПФ

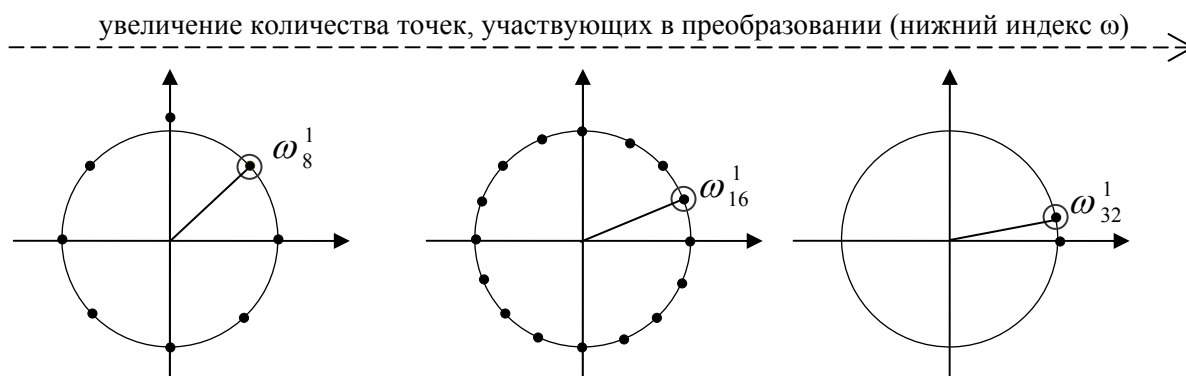
Итак, алгоритм успешно создан и запрограммирован. Закончена ли работа ?

Это зависит от сферы применения алгоритма. Если использовать его для 512, 1024 точек, то все нормально и на этом можно остановиться. Однако на более длинных векторах ошибки вычислений могут привести к неудовлетворительному результату. Поэтому чрезвычайно важно сделать преобразование более точным.

Основной момент, который необходимо улучшить – тригонометрия, а именно, вычисление корней из единицы. В рассмотренных примерах последовательные степени получаются домножением текущего корня на заранее вычисленный главный корень  $\omega^1$ . С одной стороны, это гораздо лучше, чем каждый раз вызывать очень медленные(эквивалентные сотням базовых операций и тормозящие конвейер) функции  $\sin()$ ,  $\cos()$  или  $\exp()$ . С другой - при таком подходе ошибка растет экспоненциально. Поэтому на практике используются алгоритмы семейства CORDIC (COordinate Rotation DIgital Computer), которое описывает эффективные итерационные методы поворота вектора  $a+ib$ .

Во-первых, рассмотрим обычный способ хранения  $\omega^1$ .

Значения корней из единицы равномерно распределены на окружности радиуса 1 с центром в нуле.



При этом, как видно из рисунка, по мере возрастания длины преобразования,  $\omega^1$  становится все ближе к точке (1, 0). Косинус соответствующего угла стремится к 1, а синус – к 0.

количество точек	8	16	32	64	128	256	512
cos x	0.7071067	0.9238795	0.9807852	0.9951847	0.9987954	0.9996988	0.9999247
sin x	0.7071067	0.3826834	0.1950903	0.0980171	0.0490676	0.0245412	0.0122715

Иррациональные значения синуса и косинуса не могут храниться полностью, поэтому потеря точности неизбежна.

Заметим, что формат чисел с плавающей точкой пропускает ведущие нули, т.е

0.0000012345678 будет храниться в виде 0.12345678 exp-5, всего 8 цифр

в то время как число 0.9999912345678 никак сокращено не будет и займет 13 цифр.

Поэтому потеря точности для косинуса будет гораздо больше, чем для синуса.

Этого можно избежать, если хранить  $\omega^1$  в виде Root=(a,b), где  $b = \sin(x)$ ,  $a = \cos(x)-1 = -2\sin^2(x/2)$ .

В этом случае девятки cos(x) станут нулями для  $a=\cos(x)-1$  и перестанут занимать драгоценные разряды. Однако, Root уже не будет корнем из единицы, поэтому вместо домножения на

$\omega^1$  придется использовать другую формулу перехода от текущего PRoot к следующему:

```
Temp = PRoot;
PRoot = PRoot * Root;
PRoot = PRoot + Temp;
```

Простым упражнением является проверка, действительно ли это преобразование дает в PRoot следующий корень из единицы.

После отбрасывания лишних девяток точность возрастает довольно сильно, однако при больших степенях последовательное умножение все равно приведет к увеличению ошибки. Чтобы этого избежать, можно при вычислении каждого 16-го корня вызывать sin(x) (косинус уже не используется), таким образом “обновляя” текущее значение.

```
#define CONST_PI      3.1415926535897932384626433832      // число пи

#define TRIG_VARS      \                                // переменные для тригонометрии
    ulong TLen,TNdx;int TDir; \
    Complex PRoot,Root;
```

Начальные значения Root и PRoot устанавливаются макросом

```
#define INIT_TRIG(LENGTH,DIR) \
    TNdx=0;TLen=(LENGTH);TDir=(DIR); \
    PRoot.r=1.0;PRoot.i=0.0; \
    Root.r=sin(CONST_PI/((LENGTH)*2.0)); \
    Root.r=-2.0*Root.r*Root.r; \
    Root.i=sin(CONST_PI/(LENGTH))*(DIR);
```

Каждая 16-я степень корня вычисляется через тригонометрические функции, остальные – рекуррентным соотношением.

```
#define NEXT_TRIG_POW \
    if ((++TNdx)&15)==0) { \                                // 16-я степень, вызывается sin()
        real Angle; \
        Angle=(K_PI*(TNdx))/TLen; \
        PRoot.r=sin(Angle*0.5); \
        PRoot.r=1.0-2.0*PRoot.r*PRoot.r; \
        PRoot.i=sin(Angle)*(TDir); \
    } else { \                                            // Рекуррентное соотношение
        Complex Temp; \
        Temp=PRoot; \
        PRoot = PRoot*Root; \
        PRoot = PRoot+Temp; \
    }
```

При такой системе потеря точности на тригонометрии составляет не более 2-3 бит даже при очень длинных векторах. Дополнительные ухищрения могут улучшить эту оценку, но ошибка в несколько бит, как правило, вполне терпима.

### БПФ векторов с действительными компонентами

Если компоненты вектора – действительные числа, то для вычисления дискретного преобразования Фурье его нужно сначала перевести в комплексный вид, заполнив мнимую часть нулями. При этом его фактическая длина возрастет в 2 раза, соответственно увеличивая требования к памяти и время на обработку.

Хорошо бы получить метод, который будет учитывать знание о том, что мнимые части равны нулю, но при этом не производить никаких фактических операций с ними.

Сделаем небольшой трюк. А именно, будем считать, что входной вектор  $(a_0, a_1, a_2, \dots, a_{N-1})$  на самом деле является вектором комплексных чисел  $c$  длины  $N/2$ , где каждое число представлено парой  $z_k = (a_{2k}, a_{2k+1})$ ,  $k=0 \dots N/2-1$ . Например, вектор  $a = (1, -2, 3, 4, -5, 6, -7, -8)$  на самом деле хранит комплексные числа и имеет вид  $z = (1-2i, 3+4i, -5+6i, -7-8i)$ . Будем интерпретировать вектор  $a$  таким необычным образом и посмотрим, что из этого выйдет.

Подставив  $z_k = a_{2k} + ia_{2k+1}$  в определение ДПФ, можно получить, что координаты  $c = \text{ДПФ}(z)$  имеют вид  $c_k = F_k^{\text{чет}} + iF_k^{\text{нечет}}$ ,  $k=0 \dots N/2-1$ , где

$$F_k^{\text{чет}} = \sum_{j=0}^{N/2-1} a_{2j} \omega_{N/2}^{kj} = \sum_{j=0}^{N/2-1} a_{2j} \omega_N^{2kj} \quad F_k^{\text{нечет}} = \sum_{j=0}^{N/2-1} a_{2j+1} \omega_{N/2}^{kj} = \sum_{j=0}^{N/2-1} a_{2j+1} \omega_N^{2kj}$$

Точно такие же выражения уже были раньше, при представлении преобразования длины  $N$  через ДПФ половинной длины. Если объединить  $F_k^{\text{чет}}$  и  $F_k^{\text{нечет}}$  по формуле (0), то результатом будет обычное преобразование Фурье вектора  $a$ , как если бы мы действовали по классической схеме, считая  $a_i$  комплексным числом с нулевой мнимой частью.

$$y_k = \sum_{j=0}^{N/2-1} a_{2j} \omega_N^{2kj} + \omega_N^k \sum_{j=0}^{N/2-1} a_{2j+1} \omega_N^{2kj} = F_k^{\text{чет}} + \omega_N^k F_k^{\text{нечет}} \quad (3)$$

$F_k^{\text{чет/нечет}}$  можно выразить через элементы вектора  $c$ , и, подставив в (5), получить окончательную формулу для пересчета ДПФ( $c$ ) в ДПФ( $a$ ):

$$y_k = \frac{1}{2}(c_k + c_{N/2-k}^*) - \frac{i}{2} \omega_N^k (c_k - c_{N/2-k}^*), \quad k=0 \dots N-1 \quad (3')^3$$

Для доказательства равносильности (3) и (3') достаточно заметить, что

$$\omega_{N/2}^{(N/2-n)j} = (\omega_{N/2}^{N/2})^j \omega_{N/2}^{-nj} = \omega_{N/2}^{-nj}, \text{ следовательно}$$

$$c_{N/2-k}^* = (F_k^{\text{чет}})^* + (iF_k^{\text{нечет}})^* = \sum (a_{2j} \omega_{N/2}^{(N/2-n)j})^* + \sum (ia_{2j+1} \omega_{N/2}^{(N/2-n)j})^* = F_k^{\text{чет}} - iF_k^{\text{нечет}}$$

Подставляя  $c_{N/2-k}^* = F_k^{\text{чет}} - iF_k^{\text{нечет}}$  и  $c_k = F_k^{\text{чет}} + iF_k^{\text{нечет}}$  в (3') получаем (3).

АЛГОРИТМ ДЕЙСТВИТЕЛЬНОЕ\_БПФ0( $a, N$ ) {

1.  $c$  = комплексная интерпретация вектора  $a$

2. Вычислить “на месте” БПФ( $c, N/2$ )

Пересчитать получившийся вектор в ДПФ( $a$ ) по формуле (3')

}

<sup>3</sup> Индексы вычисляются по модулю  $N$ , т.е, например,  $y_0 = y_N$ ,  $c_0 = c_N$ ,  $c_{-N/4} = c_{3N/4}$ .

Этот алгоритм вычисляет БПФ в 2 раза меньшей длины, не добавляя лишних нулей. Налицо существенное улучшение.

Однако, хотелось бы выполнять преобразование “на месте”. Для создания такого алгоритма рассмотрим структуру вектора  $y = \text{ДПФ}(a)$ , если  $a$  – действительный вектор. В этом случае

выполняется равенство  $y_{N-k} = y_k^*$ , т.е компоненты вектора от  $y[1]$  до  $y[N/2-1]$  являются сопряженными к  $y[N-1] \dots y[N/2+1]$ .

Из равенства также следует, что  $y_0 = y_0^*$  и  $y_{N/2} = y_{N/2}^*$ , так что  $y_0$  и  $y_{N/2}$  являются чисто действительными, без мнимой части.

Например, рассмотрим  $a = (4, 3, 6, 1, 0, 0, 0, 0)$ , здесь  $N=8$ .

$y = \text{ДПФ}(a) \approx (14, 5.4+8.8i, -2+2i, 2.6-3.2i, 6, 2.6+3.2i, -2-2i, 5.4-8.8i)$ .

Видно, что  $y[0]$ ,  $y[4]$  – чисто действительные числа, а элементы вида  $y[k]$   $y[8-k]$ ,  $k=1 \dots 3$  являются сопряженными друг другу.

Поэтому часть вектора от  $y[N/2+1]$  до  $y[N-1]$  можно отбросить – она тривиальным образом восстанавливается из  $y[0] \dots y[N/2]$ . Кроме того,  $y[0]$  и  $y[N/2]$  – действительные числа, значит можно хранить  $y[N/2]$  как мнимую часть  $y[0]$ .

Итак,  $\text{ДПФ}(a)$  можно без потери информации представить в виде комплексного вектора  $y[0] \dots y[N/2-1]$  длины  $N/2$ , располагающегося на месте исходного действительного вектора длины  $N$ .

действительные числа

/ \					
$y[0]$	$y[N/2]$	$y[1]$	$y[2]$	.....	$y[N/2-1]$

Теперь рассмотрим механизм собственно преобразования. Формула (3) берет 2 координаты вектора  $c$  и получает всего лишь одну компоненту  $y$ . Однако, для индекса  $N/2-k$ :

$$y_{N/2-k} = \frac{1}{2}(c_{N/2-k} + c_k^*) - \frac{i}{2}\omega_N^{N/2-k}(c_{N/2-k} - c_k^*) = \frac{1}{2}(c_{N/2-k} + c_k^*) + \frac{i}{2}\omega_N^{-k}(c_{N/2-k} - c_k^*),$$

$$\text{откуда } y_{N/2-k}^* = \frac{1}{2}(c_k + c_{N/2-k}^*) + \frac{i}{2}\omega_N^k(c_k - c_{N/2-k}^*) \quad (4)$$

АЛГОРИТМ ДЕЙСТВИТЕЛЬНОЕ\_БПФ( $a$ ,  $N$ ) {

1. Пусть  $c$  = комплексная интерпретация вектора  $a$
2. Вычислить “на месте” БПФ( $c$ ,  $N/2$ )
3. Применяя формулы (3') и (4) одновременно для  $k=1..N/2-1$ , получить значения  $y[1] \dots y[N/2-1]$ .
4. Вычислить  $y[0]$ ,  $y[N/2]$  и записать  $y[N/2]$  как мнимую часть  $y[0]$  (для индексов 0 и  $N/2$  формула (3') примет особенно простой вид).

}

Этот алгоритм не требует дополнительной памяти и использует в два раза меньше БПФ, однако использующая его программа должна учитывать нестандартность выходного формата и, при необходимости, восстанавливать отброшенные сопряженные элементы.

Если требуется провести обратное преобразование Фурье, а входной вектор находится в “сжатом формате”, то сначала проводится пересчет по формуле (3'), а затем  $\text{БПФ}^{-1}$ . При этом есть еще изменения в тригонометрических формулах, связанные с переменной знака у корня из единицы, но они уже учтены в макросах для тригонометрии.

Перед запуском RealFFT не нужно преобразовывать вектор функцией `FFTR reorder`.

```

// БПФ действительного вектора "на месте". FFTReOrder делается внутри функции.
void RealFFT(real *ddata, ulong Len, int Dir) {
    ulong i, j;
    Complex *Data=(Complex*) ddata;           // теперь вектор стал "комплексным"
    TRIG_VARS;

    Len /= 2;                                  // его длина при этом уменьшилась
                                              // 2 действительных = 1 комплексное

    if (Dir > 0) {
        FFTReOrder(Data, Len);
        FFT_T(Data, Len, 1);
    }

    INIT_TRIG(Len, Dir);
    NEXT_TRIG_POW;

    // пересчет по формулам (3'), (4)
    for (i = 1, j = Len - i; i < Len/2; i++, j--) {
        Complex p1, p2, t;

        t = conj(Data[j]);
        p1 = Data[i] + t;                      // левое слагаемое для (3')
        p2 = Data[i] - t;                      // правое слагаемое для (3')
        p2 = p2 * PRoot;
        t = Complex(-Dir*p2.i, Dir*p2.r); // умножение правого слагаемого на i

        Data[i] = p1 - t;                      // (3')
        Data[j] = p1 + t;                      // (4)
        Data[j] = conj(Data[j]);               // снимаем лишнее сопряжение

        Data[i] = Data[i]/2;
        Data[j] = Data[j]/2;

        NEXT_TRIG_POW;
    }

    // Отдельно обрабатываем индексы 0, N/2
    {
        real r, i;
        r=Data[0].r; i=Data[0].i;
        Data[0] = Complex(r+i, r-i);
    }

    if (Dir < 0) {
        Data[0] = Data[0]/2.0;
        FFTReOrder(Data, Len);
        FFT_T(Data, Len, -1);
    }
}

```

Нормализация не производится. При необходимости ее можно сделать по окончании работы функции.

### Оценка времени

Пусть  $T(N)$  – время вычисления БПФ длины вектора  $N$ . Вектор рекурсивно разбивается на две части, объединение которых происходит за  $\Theta(N)$  операций. Всего шагов разбиения  $\log_2 N$ , так что общее время  $\Theta(N \log N)$ .

Для получения более точной оценки необходимо найти общее количество базовых операций. На каждом уровне дерева рекурсии делается  $N/2$  бабочек, каждая состоит из одного комплексного умножения и двух комплексных сложений.

Учитывая, что комплексное умножение выполняется за 4 обычных умножения и 2 сложения, получаем общее количество операций:  $2N$  умножений и  $3N$  сложений/вычитаний за на каждом уровне дерева рекурсии.

Итак, всего на бабочки уходит  $2N\log_2 N$  умножений и  $3N\log_2 N$  сложений.

Теперь подсчитаем затраты на тригонометрию и, заодно, сравним рекурсивный и итеративный варианты БПФ.

В рекурсивном варианте NEXT\_TRIG\_POW вызывается при каждой бабочке, т.е. всего  $(N\log_2 N)/2$  раз. Кроме того, INIT\_TRIG вызывается по разу на каждом узле дерева рекурсии, всего узлов  $2N-1$ . Таким образом, стоимость вычислений составляет  $\Theta(N\log N)$ .

В итеративном варианте NEXT\_TRIG\_POW вызывается HalfStep раз в цикле по b. Учитывая, что значение HalfStep увеличивается с шагом в 2 раза, получаем общее число вызовов  $2 + 4 + 8 + \dots + N/2 = N-2$ . INIT\_TRIG запускается при каждом увеличении переменной Step, т.е. всего  $\log_2 N$  раз. Как видно, здесь затраты оцениваются как  $\Theta(N)$ .

### Улучшения и другие типы БПФ

- Описанный метод делит вектор на две части. Это – лишь частный случай классического БПФ, так как можно делить вектор не на 2, а на  $2^k$  частей. Более подробно об этом можно прочесть, например, в книге [1]. Весьма эффективен алгоритм “БПФ с разделенным основанием” (Split-radix FFT). Он требует около  $4N\log_2 N$  базовых операций на бабочках. Алгоритмы БПФ для векторов с длинами, не являющимися степенями 2, рассматриваются в работе Бейли[4].
- Полученный код имеет большой запас оптимизации. Его можно серьезно ускорить, если делать по 4 бабочки за итерацию. Кроме того, в качестве базиса рекурсии можно взять 4-элементные векторы и делать соответствующие преобразования “вручную”, так как для них формулы имеют особенно простой вид. Значения синусов для INIT\_TRIG имеет вычислить и сохранить в таблице при инициализации алгоритма. Если, вдобавок к этому, реализовать комплексную арифметику вручную (не как функции класса, а непосредственно), то скорость работы может серьезно возрасти (конкретные цифры сильно зависят от компилятора). Часть этих усовершенствований будут разобраны в обсуждении дискретного преобразования Хартли.
- Мы рассмотрели стиль быстрого преобразования Фурье (назовем его БПФ\_В), который называют *разбиением по времени*. Существует другой аналогичный подход, называемый *разбиением по частоте* (БПФ\_Ч). Он основывается на представлении БПФ в виде:

$$y_k = \sum_{x=0}^{N-1} a_x \omega^{xk} = \sum_{x=0}^{N/2-1} a_x \omega^{xk} + \sum_{x=N/2}^{N-1} a_x \omega^{xk} = \sum_{x=0}^{N/2-1} a_x \omega^{xk} + \sum_{x=0}^{N/2-1} a_{x+N/2} \omega^{(x+N/2)k} = \sum_{x=0}^{N/2-1} (a_x^{\text{лев}} + \omega_N^{kN/2} a_x^{\text{прав}}) \omega_N^{xk}$$

Для четных  $k=2j$  получаем

$$y_{2j} = \sum_{x=0}^{N/2-1} (a_x^{\text{лев}} + a_x^{\text{прав}}) \omega_N^{xj} \quad (1')$$

Для нечетных  $k=2j+1$

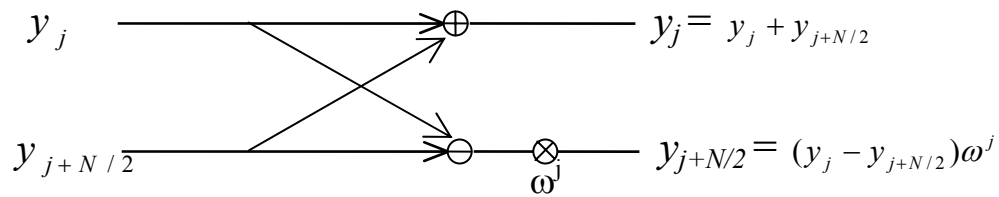
$$y_{2j+1} = \sum_{x=0}^{N/2-1} \omega^x (a_x^{\text{лев}} - a_x^{\text{прав}}) \omega_N^{xj} \quad (2')$$

Таким образом, четные индексы БПФ\_Ч( $a$ ) вычисляются по формуле (1'), а нечетные – по формуле (2'). При этом суммы являются преобразованиями Фурье половинной длины от векторов с компонентами  $a_x^{\text{лев}} + a_x^{\text{прав}}$  и  $\omega^x (a_x^{\text{лев}} - a_x^{\text{прав}})$ ,  $x=0 \dots N/2-1$ .

Отсюда вытекает другой алгоритм вычисления ДПХ “на месте”.

БПФ\_Ч {

1. Сделать преобразования бабочки БПФ\_Ч по формулам (1'), (2') для  $j=0 \dots N/2-1$ , записывая координаты с четными индексами в левую половину вектора, с нечетными – в правую.



2. Вызвать БПФ\_Ч на левой и правой половинах вектора
3. Реорганизовать вектор процедурой FFTReOrder()

В отличие от БПФ\_Т, здесь бабочки выполняются до рекурсивных вызовов. Кроме того, реорганизация должна происходить в конце работы алгоритма, а не в начале (Почему?). Количество операций в бабочке то же: 4 умножения и 6 сложений. На практике БПФ\_Ч обычно выполняется чуть медленнее, нежели БПФ\_В, но дает немного меньшую ошибку вычислений..

## Дискретное Преобразование Хартли

Преобразование Фурье – очень мощный инструмент, часто использующийся при обработке сигналов, в оптике, в томографии... Возможно, даже слишком мощный, так как в реальных приложениях комплексные числа встречаются далеко не всегда. Часто преобразование производится над действительными векторами, а комплексные числа служат лишь промежуточным звеном вычислений, приводящих к действительному ответу. Можно ли исключить комплексные числа вообще? Рассмотренная нами функция RealFFT() в этом смысле является лишь полумерой.

Интересной альтернативой ДПФ является дискретное преобразование Хартли (ДПХ). ДПХ действительного вектора  $a$  – это действительный вектор  $y$  с координатами

$$h_k = \sum_{j=0}^{N-1} a_j \left( \cos \frac{2\pi k j}{N} + \sin \frac{2\pi k j}{N} \right) \quad h = \text{ДПХ}(a) \quad (4)$$

Никаких комплексных чисел! Многие люди, видя эту формулу впервые, искренне недоумевают: что может дать преобразование по функции, являющейся всего лишь сдвинутым синусом:

$$\cos(x) + \sin(x) = \sqrt{2} \sin\left(x + \frac{\pi}{4}\right) ?$$

Оказывается, выбор именно такого синуса приводит к интересным результатам.

В частности, обратное преобразование Хартли вычисляется по формуле, отличающейся лишь множителем  $1/N$ :

$$h_k = \frac{1}{N} \sum_{j=0}^{N-1} a_j \left( \cos \frac{2\pi k j}{N} + \sin \frac{2\pi k j}{N} \right) \quad h = \text{ДПХ}^{-1}(a)$$

Для действительного вектора  $a$  между  $\text{ДПФ}(a) = (y_0, y_1, \dots, y_{N-1})$  и  $\text{ДПХ}(a) = (h_0, h_1, \dots, h_{N-1})$  существует простая связь:

$$\text{Re } y_k = \frac{1}{2}(h_k + h_{N-k}) \quad \text{Im } y_k = \frac{1}{2}(h_k - h_{N-k})$$

Существует и обратная формула, так что можно получить одно через другое. Вообще говоря, можно выразить даже ДПФ комплексного вектора через ДПХ, однако все преимущества преобразования Хартли проявляются при работе с действительными числами.

## Быстрое преобразование Хартли\*

### Разбиение по времени

Пусть длина вектора  $N$  является степенью двойки. БПХ с разбиением по времени (БПХ\_В) состоит в делении вектора на четную и нечетную части  $a^{чет} = (a_0, a_2, a_4, \dots, a_{N-2})$  и  $a^{нечет} = (a_1, a_3, a_5, \dots, a_{N-1})$  и вычисления ДПХ( $a$ ) через преобразования Хартли этих векторов. Таким образом, задача сводится к вычислению ДПХ векторов половинной длины, и так до векторов длины 2, когда вычисления можно проводить непосредственно по формуле (4).

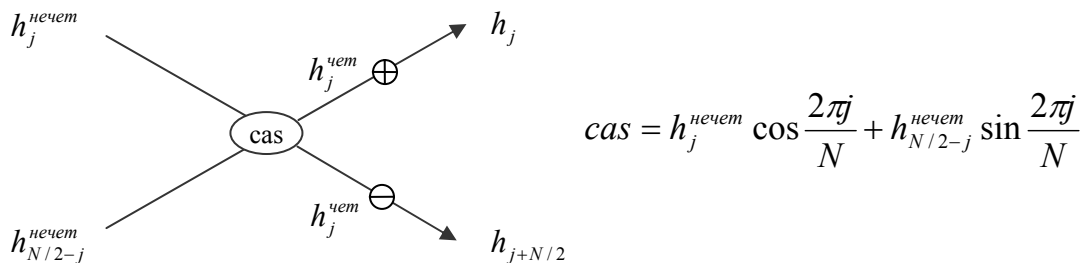
При разбиении по времени левая половина ДПХ длины  $N$  получается по формуле (5), а правая половина – по формуле (6), из ДПХ векторов с четными и нечетными номерами координат.

$$h_j = h_j^{чет} + h_j^{нечет} \cos \frac{2\pi j}{N} + h_{N/2-j}^{нечет} \sin \frac{2\pi j}{N} \quad (5)$$

$$h_{j+N/2} = h_j^{чет} - h_j^{нечет} \cos \frac{2\pi j}{N} - h_{N/2-j}^{нечет} \sin \frac{2\pi j}{N}, \quad j = 0 \dots N/2 - 1 \quad (6)$$

При этом для корректности обработки значения  $j=0$  полагаем  $h_N = h_0$ .

Формулы можно более наглядно изобразить в виде “бабочки” БПХ\_В.



Сначала вычисляется временная переменная  $cas$ , затем она складывается, либо вычитается из  $h_j^{чет}$ .

Если приглядеться к этой бабочке, то можно заметить одно существенное отличие от бабочки БПФ. А именно, она использует три значения для генерации двух. Это означает, что такие преобразования нельзя делать “на месте”, потому что затираемые значения могут использоваться в других бабочках.

### Объединение БПХ половинной длины без использования дополнительной памяти

Попробуем смоделировать ситуацию, которая происходит при работе “на месте” и найти, в каких бабочках используются одни и те же значения. Заметим, что схема разбиения вектора на части абсолютно такая же, как при БПФ\_В. Поэтому для того, чтобы преобразования прошли в правильном порядке, нужно сделать предварительную реорганизацию массива процедурой FFTReOrder().

Выпишем все преобразования, происходящие при объединении ДПХ половинной длины  $a[0..7]$  и  $a[8..15]$  в одно  $a[0..15]$ ,  $N=16$ . Благодаря предварительной перестановке FFTReOrder() можно

считать, что в левом массиве хранятся элементы с четными индексами, а в правом – с нечетными. Реальный индекс  $a_j^{нечет}$  в массиве  $a[]$ , таким образом, равен  $j+N/2$ , индекс  $a_j^{чет}$  равен  $j$ . Вычисления идут по формулам (5),(6).

Бабочка 0.

$$\begin{aligned} a[0] &= a[0] + a[8] \\ a[8] &= a[0] - a[8] \end{aligned}$$

Бабочка 1.

$$\begin{aligned} a[1] &= a[1] + \text{cas}(a[9], a[15]) \\ a[9] &= a[1] - \text{cas}(a[9], a[15]) \end{aligned}$$

Бабочка 2.

$$\begin{aligned} a[2] &= a[2] + \text{cas}(a[10], a[14]) \\ a[10] &= a[2] - \text{cas}(a[10], a[14]) \end{aligned}$$

Бабочка 3.

$$\begin{aligned} a[3] &= a[3] + \text{cas}(a[11], a[13]) \\ a[11] &= a[3] - \text{cas}(a[11], a[13]) \end{aligned}$$

Бабочка 4.

$$\begin{aligned} a[4] &= a[4] + \text{cas}(a[12], a[12]) \\ a[12] &= a[4] - \text{cas}(a[12], a[12]) \end{aligned}$$

Бабочка 5.

$$\begin{aligned} a[5] &= a[5] + \text{cas}(a[13], a[11]) \\ a[13] &= a[5] - \text{cas}(a[13], a[11]) \end{aligned}$$

Бабочка 6.

$$\begin{aligned} a[6] &= a[6] + \text{cas}(a[14], a[10]) \\ a[14] &= a[6] - \text{cas}(a[14], a[10]) \end{aligned}$$

Бабочка 7.

$$\begin{aligned} a[7] &= a[7] + \text{cas}(a[15], a[9]) \\ a[15] &= a[7] - \text{cas}(a[15], a[9]) \end{aligned}$$

Рассмотрим все бабочки, за исключением нулевой и четвертой.

Как видно, индекс первого аргумента  $\text{cas}()$  возрастает с 9 до 15, в то время как индекс второго – убывает с 15 до 9. Таким образом,  $i$ -й элемент,  $i=9..15$  используется дважды – один раз в бабочке номер  $i-N/2$  и другой – в бабочке номер  $N-i$ . Например, элемент  $a[10]$  используется в бабочках 2( $=10-8$ ) и 6( $=16-10$ ).

Значит, эти бабочки имеют в совокупности 4 входа и 4 выхода! В частности, бабочки номер 2 и 6 принимают значения  $a[2], a[10], a[14], a[6]$  и их же выдают. Поэтому можно организовать “спаренную”, двойную бабочку, которую можно выполнять “на месте” без опасения затереть нужные данные. Номера спаренных бабочек в сумме равны  $N/2$ .

Пусть номер первой бабочки равен  $N_1$ ,  $C = \cos \frac{2\pi N_1}{N}$ ,  $S = \sin \frac{2\pi N_1}{N}$  – значения, необходимые

для вычисления  $\text{cas}$ . Тогда парная бабочка будет иметь номер  $N_2 = N/2 - N_1$ ,

Для вычисления первой бабочки используется  $\text{cas}(a_{N_1}, a_{N_2}) = a_{N_1}^{нечет} * C + a_{N_2}^{нечет} * S$ , а из равенств

$$\cos \frac{2\pi N_2}{N} = \cos \frac{2\pi(N/2 - N_1)}{N} = \cos(\pi - \frac{2\pi N_1}{N}) = -C, \quad \sin \frac{2\pi N_2}{N} = \sin(\pi - \frac{2\pi N_1}{N}) = S$$

следует, что для парной бабочки  $\text{cas}(a_{N_2}, a_{N_1}) = a_{N_1}^{нечет} * S - a_{N_2}^{нечет} * C$ , то есть используются те же значения триг. функций.

Опишем соответствующую процедуру в виде готового макроса.

```
#define FHT_T2Butterfly(N1,N2,C,S) { \ // двойная бабочка ВПХ_В
    double Rx,Ri; \
    ulong i1=N1,i2=N2; \
    Rx=Right[i1]; Ri=Right[i2]; \
    { \ // бабочка с номером N1
        double cas1,Lx; \
        cas1=Rx*(C)+Ri*(S); \ // используемое значение cas
        Lx=Left[i1]; \
        Left[i1] = Lx+cas1; \ // Left[] - левая половина вектора
        Right[i1] = Lx-cas1; \
    } \
    { \ // бабочка с номером N2
        double cas2,Li; \
        cas2=Rx*(S)-Ri*(C); \
        Li=Left[i2]; \
        Left[i2] = Li+cas2; \
    } \
}
```

```

    }
    Right[i2] = Li-cas2;
}

```

Здесь Right – правая, “нечетная” половина вектора, а Left – левая, четная.

Бабочки с номерами 0, N/4 (в примере N/4=4) особые. Они могут быть сделаны “на месте” обычным образом, так имеют по два одинаковых входа и выхода.

```

#define FHT_T1Butterfly(N1,N2,C,S) {
    ulong i1=N1,i2=N2;
    double cas1=Right[i1]*(C)+Right[i2]*(S);
    double temp=Left[i1];
    Left[i1] = temp + cas1;
    Right[i2]= temp - cas1;
}

```

Итак, способ объединения “на месте” найден – это две бабочки одновременно.

## Тригонометрия

Прежде всего, посмотрим, где можно использовать одни и те же значения cos() и sin(). Одну закономерность мы уже обнаружили. Однако, можно заметить, что формулы приведения дают аналогичный результат для номеров N/4-N<sub>1</sub>, N/4+N<sub>2</sub>:

$$\cos \frac{2\pi(N/4 \pm N_1)}{N} = \cos\left(\frac{\pi}{2} \pm \frac{2\pi N_1}{N}\right) = \mu S, \quad \sin \frac{2\pi(N/4 \pm N_1)}{N} = \sin\left(\frac{\pi}{2} \pm \frac{2\pi N_1}{N}\right) = C$$

Таким образом, один раз вычисленные значения C и S можно применять для четырех бабочек: для спаренной с номерами N<sub>1</sub>, N/2-N<sub>1</sub> и для спаренной с номерами N/4-N<sub>1</sub>, N/4+N<sub>1</sub>.

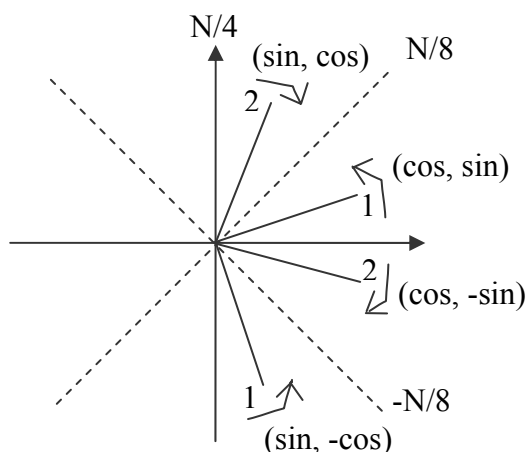
```

for (N1=1; N1 < N/8; N1++) {
    { вычислить очередные значения Cos, Sin }
    FHT_T2Butterfly(N1, N/2-N1, Cos, Sin);
    FHT_T2Butterfly(N/4-N1, N/4+N1, Sin, Cos);
}

```

В этом цикле будут проведены 4\*(N/8-1)=N/2-4 преобразования с минимумом затрат на тригонометрию. Цикл пропустит номера 0, N/4, N/8, 3N/8.

Дело в том, что при N<sub>1</sub>=0 получаются номера 0, N/2, N/4, N/4, а бабочка с номером N/2 попросту не существует. N<sub>1</sub>=N/8 также дает плохой набор, поэтому четыре пропущенных преобразования необходимо сделать вне цикла.



Геометрическая интерпретация

Если рассмотреть используемые при вычислении cas значения косинуса и синуса как координаты вектора на плоскости, то первой спаренной бабочке ставятся в соответствие перпендикулярные векторы (cos, sin) и (sin, -cos), а второй – векторы (sin, cos), (cos, -sin).

Каждый следующий шаг приводит к изменению значений cos и sin, которое является поворотом этих бабочек вокруг начала координат на угол, задаваемый вектором (sin(2π/N), cos(2π/N)). Причем вторая бабочка поворачивается по часовой стрелке, а первая – против.

Обратим внимание, что требуемые для бабочек значения триг. функций - те же, что и в БПФ, где,

как было замечено в примечаниях, также можно делать по 4 преобразования одновременно. Особые случаи происходят, когда векторы совпадают, поэтому соответствующие номера обрабатываются отдельно.

Генерация производится через ту же рекуррентную формулу, что и в БПФ, разве что значения хранятся в разных переменных Sin и Cos, а не в одном корне:  $\omega = \cos + i \sin$ .

Во время инициализации создадим таблицу синусов  $\text{SineTable}[i] = \sin(\pi/2^{i+1})$ .

```
void CreateSineTable(ulong Len) {
    int x=0; ulong P=1;
    while (P<=Len*4) {
        SineTable[x]=sin(CONST_PI/P);
        P*=2;
        x++;
    }
}
```

Если текущая длина  $N=2^x$ , то синус для первой бабочки  $\text{Sin0} = \sin(2\pi/2^x) = \text{SineTable}[x]$ .  
Вместо косинуса, для будем хранить значение  $\text{Cos0} = \cos(2\pi/2^x) - 1 = -2\sin^2(x/2)$ , где  $\sin(x/2) = \text{SineTable}[x+1]$ .

```
int Log2(ulong Num) {
    int x=-1;
    if (Num==0) return 0;
    while (Num) {x++;Num/=2;}
    return x;
}

#define INIT_TRIG(LENGTH) \
    ulong x=Log2(LENGTH); \
    real Sin0=SineTable[x]; \
    real Cos0=SineTable[x+1]; \
    Cos0=-2.0*Cos0*Cos0; \
    real Sin=Sin0, Cos=1.0+Cos0;
```

Эти значения вычисляются один раз переход к следующему углу происходит по формулам:

```
#define NEXT_TRIG_POW { \
    real temp=Cos; \
    Cos = Cos*Cos0 - Sin*Sin0 + Cos; \ // (Cos,Sin) - текущий вектор \
    Sin = Sin*Cos0 + temp*Sin0 + Sin; \
}
```

Периодического обновления значений Sin, Cos здесь не требуется – точность вполне достаточна.

### Окончательная реализация БПХ с разбиением по времени

Итак, все готово для полного рекурсивного алгоритма БПХ\_B.

С целью увеличения быстродействия базис рекурсии будут представлять векторы длины  $N=8$ .

#### 1. Инициализация:

- реорганизовать вектор процедурой  $\text{FFTReOrder}(\text{Data}, \text{Len})^4$
- создать таблицу синусов функцией  $\text{CreateSineTable}(\text{Len})$

$\text{БПХ\_B}(\text{Data}, \text{Len}) \{$

Если длина равна 8, то вычислить БПХ\_B непосредственно и выйти.

#### 2. Рекурсивно вызвать

- $\text{БПХ\_B}(\text{Data}, \text{Len}/2)$
- $\text{БПХ\_B}(\text{Data}+\text{Len}/2, \text{Len}/2)$

#### 3. Сделать по две спаренных бабочки для $N_i=1\dots N/8-1$

#### 4. Сделать бабочки с номерами 0, $N/4$ , $N/8$ , $3N/8$

$\}$

// БПХ с разбиением по времени вектора Data длины  $\text{Len} \geq 8$ .

// Шаг 1 должен быть сделан к моменту вызова функции. Используются константы:

```
#define CONST_PI 3.1415926535897932384626433832
```

<sup>4</sup>В коде из обсуждения БПФ следует поменять все объявления Complex на real

```

#define CONST_SQRT_2  0.7071067811865475244008443621
#define CONST_SQRT2  1.4142135623730950488016887242

void FHT_T(real *Data,ulong Len) {
    ulong Len2,Len4;
    real *Left,*Right;

    // Шаг 2. Все преобразования соответствуют формулам (5),(6), причем
    //           предполагается, что шага 1 в левой половине на самом деле элементы
    //           с четными индексами, а в правой - с нечетными.
    //           Для минимизации требования к регистрам используются блочные переменные
    //           Значения триг. функций вычислены заранее и представлены в виде констант
    if (Len==8) {
        real d45,d67,sd0123,dd0123;
        {
            real ss0123,ds0123,ss4567,ds4567;
            {
                real s01,s23,d01,d23;
                d01 = Data[0] - Data[1];
                s01 = Data[0] + Data[1];
                d23 = Data[2] - Data[3];
                s23 = Data[2] + Data[3];
                ds0123 = (s01 - s23);
                ss0123 = (s01 + s23);
                dd0123 = (d01 - d23);
                sd0123 = (d01 + d23);
            }
            {
                real s45,s67;
                s45 = Data[4] + Data[5];
                s67 = Data[6] + Data[7];
                d45 = Data[4] - Data[5];
                d67 = Data[6] - Data[7];
                ds4567 = (s45 - s67);
                ss4567 = (s45 + s67);
            }
            Data[4] = ss0123 - ss4567;
            Data[0] = ss0123 + ss4567;
            Data[6] = ds0123 - ds4567;
            Data[2] = ds0123 + ds4567;
        }
        d45 *= CONST_SQRT2;
        d67 *= CONST_SQRT2;
        Data[5] = sd0123 - d45;
        Data[1] = sd0123 + d45;
        Data[7] = dd0123 - d67;
        Data[3] = dd0123 + d67;
        return;
    }
    Len/=2; // N -длина вектора, то Len = N/2

    // Шаг 3.
    Right=&Data[Len];Left=&Data[0];
    FHT_T(&Left[0], Len);
    FHT_T(&Right[0],Len);

    // Шаг 4. Если
    //           Len=N/2, Len2=N/4, Len4=N/8
    INIT_TRIG(Len);

    Len2=Len/2; // Len2 = N/4
    Len4=Len/4; // Len4 = N/8
    for (x=1;x<Len4;x++) { // x=1..N/8-1 - по 4 бабочки в цикле
        FHT_T2Butterfly(x,Len-x,Cos,Sin); // спаренная бабочка 1
        FHT_T2Butterfly(Len2-x,Len2+x,Sin,Cos); // спаренная бабочка 2
        NEXT_TRIG_POW;
    }

    // Шаг 5.
    // Бабочки 0, N/4 делаются на месте
    FHT_T1Butterfly(0,0,1.0,0.0);
}

```

```

FHT_T1Butterfly(Len2,Len2,0.0,1.0);
// Бабочки N/8, 3N/8 делаются в спаренном варианте
// косинус и синус для N/8 вычислены заранее и равны SQRT(2)/2
FHT_T2Butterfly(Len4,Len-Len4,CONST_SQRT_2,CONST_SQRT_2);
}

```

## БПХ с разбиением по частоте

БПХ\_Ч представляет собой альтернативный стиль вычисления БПХ. Из левой и правой половин вектора ДПХ полной длины N вычисляется по формулам:

$$y_{2j} = \sum_{x=0}^{N/2-1} (a_x^{\text{лев}} + a_x^{\text{прав}}) \omega_{N/2}^{xj}$$

$$h_{2j} = \sum_{x=0}^{N/2-1} (h_x^{\text{лев}} + h_x^{\text{прав}}) \left( \cos \frac{2\pi jx}{N/2} + \sin \frac{2\pi jx}{N/2} \right) \quad (5')$$

$$h_{2j+1} = \sum_{x=0}^{N/2-1} \left( D_x \cos \frac{2\pi x}{N} + D_{N/2-x} \sin \frac{2\pi x}{N} \right) \left( \cos \frac{2\pi jx}{N/2} + \sin \frac{2\pi jx}{N/2} \right) \quad (6')$$

Здесь  $D_x = h_x^{\text{лев}} - h_x^{\text{прав}}$ ,  $j=0 \dots N/2-1$ .

Выражения в правой части представляют собой ДПХ половинной длины от векторов с координатами  $h_x^{\text{лев}} + h_x^{\text{прав}}$  и  $(h_x^{\text{лев}} - h_x^{\text{прав}}) \cos \frac{2\pi x}{N} + (h_{N/2-x}^{\text{лев}} - h_{N/2-x}^{\text{прав}}) \sin \frac{2\pi x}{N}$ ,  $x=0 \dots N/2-1$ .

Таким образом, возникает бабочка БПХ\_Ч, в которой для вычисления двух значений требуется четыре. Выпишем схему преобразований для N=16. Для удобства будем записывать координаты с четными индексами в левую часть вектора, а с нечетными – в правую. Элементы вида  $a_{2j}$  будут храниться как  $a[j]$ , а  $a_{2j+1}$  – как  $a[N/2+j]$ .

Бабочка 0.

$a[0] \leftarrow a[0], a[8]$   
 $a[8] \leftarrow a[0], a[8]$

Бабочка 1.

$a[1] \leftarrow a[1], a[9]$   
 $a[9] \leftarrow a[1], a[9], a[7], a[15]$

Бабочка 2.

$a[2] \leftarrow a[2], a[10]$   
 $a[10] \leftarrow a[2], a[10], a[6], a[14]$

Бабочка 3.

$a[3] \leftarrow a[3], a[11]$   
 $a[11] \leftarrow a[3], a[11], a[5], a[13]$

Бабочка 4.

$a[4] \leftarrow a[4], a[12]$   
 $a[12] \leftarrow a[4], a[12], a[4], a[12]$

Бабочка 5.

$a[5] \leftarrow a[5], a[13]$   
 $a[13] \leftarrow a[5], a[13], a[3], a[11]$

Бабочка 6.

$a[6] \leftarrow a[6], a[14]$   
 $a[14] \leftarrow a[6], a[14], a[2], a[10]$

Бабочка 7.

$a[7] \leftarrow a[7], a[15]$   
 $a[15] \leftarrow a[7], a[15], a[1], a[9]$

Как и при разбиении по времени, преобразования можно делать на месте, если образовать спаренные бабочки, сумма номеров которых равна N/2. Например, 2 и 6 бабочка вместе получают 4 координаты и выдают преобразованными их же.

```

#define FHT_F2Butterfly(N1,N2,C,S) \
{ \
    real D1,D2; \
    ulong i1=N1, i2=N2; \
    D1=Left[i1]; D2=Left[i2]; \
    { \
        real temp; \
        \
        \
        \
        \
        \
        \
    } \
} \
// спаренная бабочка \
// БПХ_Ч \

```

```

        Left[i1] =D1+(temp=Right[i1]);D1=D1-temp;      \
        Left[i2] =D2+(temp=Right[i2]);D2=D2-temp;      \
    }                                                    \
    Right[i1]=D1*(C)+D2*(S);                             \
    Right[i2]=D1*(S)-D2*(C);                             \
}

```

Из тех же соображений, что и при обсуждении разбиения по времени, следует, что пару значений триг. функций можно применять сразу для двух спаренных бабочек в цикле.

При этом четыре преобразования остаются не сделанными и их необходимо произвести вне цикла по формулам (5), (6).

```

// БПХ с разбиением по частоте вектора Data длины Len≥4.
void FHT_F(real *Data,ulong Len) {
    // Базис рекурсии - вектора длины 4. Вычисления следуют формулам (5'), (6')
    // четные элементы записываются в левую половину массива, нечетные - в правую
    if (Len==4) {
        real d0=Data[0]; real d1=Data[1];
        real d2=Data[2]; real d3=Data[3];
        {
            real d02=d0+d2; real d13=d1+d3;
            Data[0]=d02+d13; Data[1]=d02-d13;
        }
        {
            real d02=d0-d2; real d13=d1-d3;
            Data[2]=d02+d13; Data[3]=d02-d13;
        }
        return;
    }

    Len/=2;
    ulong Len2,Len4;
    real *Left = &Data[0], *Right = &Data[Len];
    // Преобразования с номерами 0, N/4 имеют 2 входа и 2 выхода
    // сделать их "на месте".
    {
        real t1,t2;
        t1=Left[0];t2=Right[0];
        Left[0]=t1+t2;Right[0]=t1-t2;

        t1=Left[Len/2];t2=Right[Len/2];
        Left[Len/2]=t1+t2;Right[Len/2]=t1-t2;
    }

    INIT_TRIG(Len);

    Len2=Len/2;
    Len4=Len/4;
    // Аналогично БПХ_В выполняем по две спаренных бабочки на каждую
    // генерацию значений триг. функций.
    for (x=1;x<Len4;x++) {
        FHT_F2Butterfly(x,Len-x,Cos,Sin);
        FHT_F2Butterfly(Len2-x,Len2+x,Sin,Cos);
        NEXT_TRIG_POW;
    }

    // Бабочки с номерами N/8, 3N/8
    FHT_F2Butterfly(Len4,Len-Len4,CONST_SQRT_2,CONST_SQRT_2);

    // Рекурсивно вызвать БПХ_Ч для преобразованных половин вектора
    FHT_F(Left, Len);
    FHT_F(Right,Len);
}

```

В процессе работы функция перетасовывает вектор, записывая четные индексы в левую половину вектора, а нечетные в правую. Поэтому для окончательного результата необходимо на получившемся векторе вызвать FFTReOrder(Data, Len).

Как и в БПФ, рассмотренные реализации не производят нормализацию вектора, оставляя этот факт на усмотрение внешней программы.

### Оценка времени

Будем оценивать БПХ с разбиением по времени. Высота дерева рекурсии составляет  $\log_2 N$ . На каждом уровне дерева, как и в БПФ, делается одинаковое количество бабочек. За исключением нижних уровней -  $N/8$ , из которых 4 – “четверенные”, одна – спаренная и две обычные. С незначительными потерями информации можно считать, что всего делается  $N/4$  спаренных бабочек.

Из макроса FHT\_T2Butterfly видно, что вычисление спаренной бабочки БПХ\_В состоит из 6 сложений и 4 умножений, так что на одном уровне происходит порядка  $3N/2$  сложений и  $N$  умножений.

Разбиение по частоте оценивается аналогично, с теми же результатами.

### Что лучше для действительных чисел: БПФ или БПХ ?

ДПФ вектора имеет физический смысл, а именно, если вектор представляет собой дискретизированный сигнал, то ДПФ раскладывает его по частотам. Преобразование Хартли не имеет такой явной интерпретации.

Однако, как говорилось ранее, преобразование Хартли можно превратить в преобразование Фурье за  $N$  сложений и умножений, что быстрее, чем послеобработка RealFFT().

Поэтому, если исходные данные действительны, то БПХ может быть более эффективным, нежели БПФ. При этом из ряда формул при подстановке представлений коэффициентов Фурье, выраженных через элементы ДПХ, получается весьма удобное для вычислений выражение. Поэтому пересчета ДПХ в ДПФ иногда можно избежать, и пример такого подхода рассматривается при умножении длинных чисел.

Как видно из оценок, количество операций для БПФ и для БПХ совпадает. В данном случае реализация БПХ получилась более оптимизированной, однако все улучшения и вариации алгоритмов, которые работают для одного из этих преобразований, подходят и для другого. Параллелизм на уровне инструкций процессора также приблизительно одинаковый, поэтому реальная эффективность практически одна и та же. Однако, есть несколько моментов, которые отличаются.

1. Код для БПХ проще. Формула обратного преобразования совпадает с формулой для прямого, за исключением множителя  $1/N$ , в то время как при вычислении обратного БПФ приходится вводить дополнительный параметр Dir или делать новую функцию.
2. Точность БПХ, как правило, немного выше, чем у БПФ.
3. При вычислении БПФ действительного вектора сначала вычисляется “комплексное” БПФ половинной длины, а потому производится послеобработка, которая отсутствует в БПХ. Это дает БПХ дополнительное упрощение кода, по сравнению с БПФ и влияет на эффективность при малых длинах векторов. Однако, чем длиннее вектор – тем это влияние слабее.

Однако, добавим в бочку меда и восхвалений БПХ небольшой половник дегтя...

БПФ активно используется уже много лет, программисты соревновались, кто его лучше напишет и, как следствие, существует много очень хорошо оптимизированных исходных кодов различных вариаций алгоритма.

Обратная ситуация с БПХ. Этот алгоритм был запатентован Брэйсуэллом в 1984 году и каждый, кто хотел его использовать, должен был платить большие деньги. Лишь

сравнительно недавно патент истек и алгоритм стал доступен широкой публике. Поэтому найти аналогичную по качеству реализацию в общем доступе довольно сложно.

(с) Кантор Илья, 2002

Отзывы и предложения по адресу [algotlist@manual.ru](mailto:algotlist@manual.ru)