http://www.relisoft.com/science/Physics/fft.html

Fast Fourier Transform (FFT)

The naive implementation of the N-point digital Fourier transform involves calculating the scalar product of the sample buffer (treated as an N-dimensional vector) with N separate basis vectors. Since each scalar product involves N multiplications and N additions, the total time is proportional to N^2 (in other words, it's an $O(N^2)$ algorithm). However, it turns out that by cleverly re-arranging these operations, one can optimize the algorithm down to O(N log(N)), which for large N makes a huge difference. The optimized version of the algorithm is called the fast Fourier transform, or the FFT.

Let's do some back of the envelope calculations. Suppose that we want to do a real-time Fourier transform of one channel of CD-quality sound. That's 44k samples per second. Suppose also that we have a 1k buffer that is being re-filled with data 44 times per second. To generate a 1000-point Fourier transform we would have to perform 2 million floating-point operations (1M multiplications and 1M additions). To keep up with incoming buffers, we would need at least 88M flops (floating-point operations per second). Now, if you are lucky enough to have a 100 Mflop machine, that might be fine, but consider that you'd be left with very little processing power to spare.

Using the FFT, on the other hand, we would perform on the order of $2*1000*\log_2(1000)$ operations per buffer, which is more like 20,000. Which requires 880k flops--less than 1 Mflop! A hundred-fold speedup.

The standard strategy to speed up an algorithm is to divide and conquer. We have to find some way to group the terms in the equation

$$V[k] = \Sigma_{n=0..N-1} W_N^{kn} v[n]$$

Let's see what happens when we separate odd ns from even ns (from now on, let's assume that N is even):

$$\begin{aligned}
V[k] &= \Sigma_{n \text{ even}} W_N^{kn} v[n] + \Sigma_{n \text{ odd}} W_N^{kn} v[n] \\
&= \Sigma_{r=0..N/2-1} W_N^{k(2r)} v[2r] + \Sigma_{r=0..N/2-1} W_N^{k(2r+1)} v[2r+1] \\
&= \Sigma_{r=0..N/2-1} W_N^{k(2r)} v[2r] + \Sigma_{r=0..N/2-1} W_N^{k(2r)} W_N^{k} v[2r+1] \\
&= \Sigma_{r=0..N/2-1} W_N^{k(2r)} v[2r] + W_N^{k} \Sigma_{r=0..N/2-1} W_N^{k(2r)} v[2r+1] \\
&= (\Sigma_{r=0..N/2-1} W_{N/2}^{kr} v[2r]) \\
&+ W_N^{k} (\Sigma_{r=0..N/2-1} W_{N/2}^{kr} v[2r+1])
\end{aligned}$$

where we have used one crucial identity:

$$\begin{aligned}
W_N^{k(2r)} &= e^{-2\pi i*2kr/N} \\
&= e^{-2\pi i*kr/(N/2)} = W_{N/2}^{kr}
\end{aligned}$$

Notice an interesting thing: the two sums are nothing else but N/2-point Fourier transforms of, respectively, the even subset and the odd subset of samples. Terms with k greater or equal N/2 can be reduced using another identity:

$W_{N/2}^{m+N/2} = W_{N/2}^{m} W_{N/2}^{N/2} = W_{N/2}^{m}$

which is true because $W_m^{\,m} = e^{-2\pi i} = \cos(-2\pi) + i \sin(-2\pi) = 1$.

If we start with N that is a power of 2, we can apply this subdivision recursively until we get down to 2-point transforms.

We can also go backwards, starting with the 2-point transform: Note $W_1^{0*k} = W_2^{0*k}$

$V[k] = W_2^{0*k} v[0] + W_2^{1*k} v[1], \quad k=0,1$

The two components are:

$V[0] = W_2^{0} v[0] + W_2^{0} v[1] = v[0] + W_2^{0} v[1]$
$V[1] = W_2^{0} v[0] + W_2^{1} v[1] = v[0] + W_2^{1} v[1]$

We can represent the two equations for the components of the 2-point transform graphically using the, so called, *butterfly*

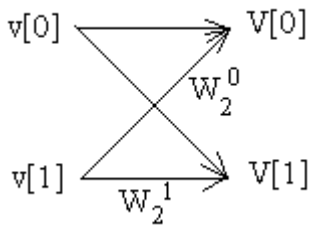

Fig. Butterfly calculation

Furthermore, using the divide and conquer strategy, a 4-point transform can be reduced to two 2-point transforms: one for even elements, one for odd elements. The odd one will be multiplied by W_4^k. Diagrammatically, this can be represented as two levels of butterflies. Notice that using the identity $W_{N/2}^{\,n} = W_N^{2n}$, we can always express all the multipliers as powers of the same W_N (in this case we choose N=4).

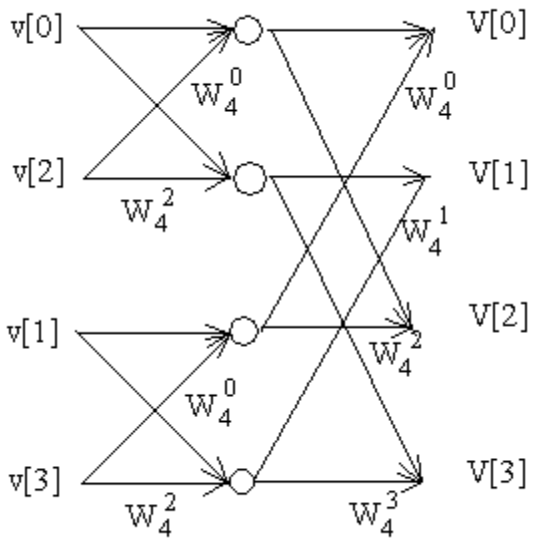

Fig. Diagrammatical representation of the 4-point Fourier transform calculation

I encourage the reader to derive the analogous diagrammatical representation for N=8. What will become obvious is that all the butterflies have similar form:

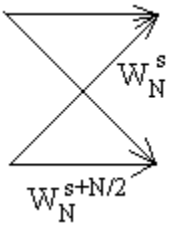

Fig. Generic butterfly graph

This graph can be further simplified using this identity:

$W_N^{s+N/2} = W_N^s W_N^{N/2} = -W_N^s$

which is true because

$W_N^{N/2} = e^{-2\pi i(N/2)/N} = e^{-\pi i} = \cos(-\pi) + i\sin(-\pi) = -1$

Here's the simplified butterfly:

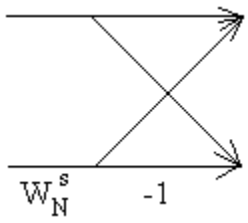

Fig. Simplified generic butterfly

Using this result, we can now simplify our 4-point diagram.

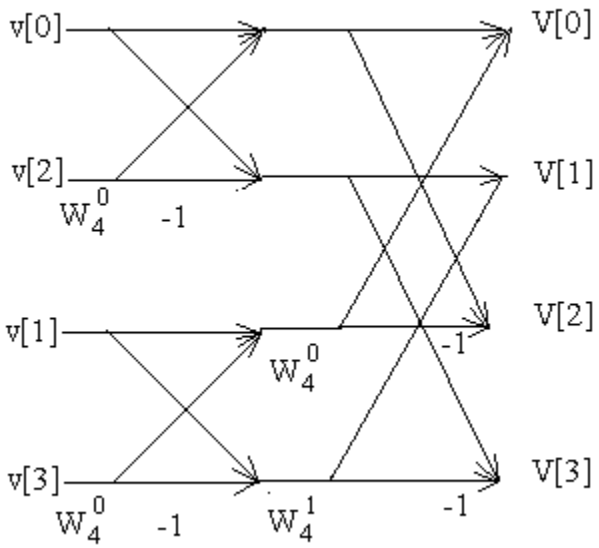

Fig. 4-point FFT calculation

This diagram is the essence of the FFT algorithm. The main trick is that you don't calculate each component of the Fourier transform separately. That would involve unnecessary repetition of a substantial number of calculations. Instead, you do your calculations in stages. At each stage you start with N (in general complex) numbers and "butterfly" them to obtain a new set of N complex numbers. Those numbers, in turn, become the input for the next stage. The calculation of a 4-point FFT involves two stages. The input of the first stage are the 4 original samples. The output of the second stage are the 4 components of the Fourier transform. Notice that each stage involves N/2 complex multiplications (or N real multiplications), N/2 sign inversions (multiplication by -1), and N complex additions. So each stage can be done in O(N) time. The number of stages is

$\log_2 N$ (which, since N is a power of 2, is the exponent m in $N = 2^m$). Altogether, the FFT requires on the order of O(N logN) calculations.

Moreover, the calculations can be done in-place, using a single buffer of N complex numbers. The trick is to initialize this buffer with appropriately scrambled samples. For N=4, the order of samples is v[0], v[2], v[1], v[3]. In general, according to our basic identity, we first divide the samples into two groups, even ones and odd ones. Applying this division recursively, we split these groups of samples into two groups each by selecting every other sample. For instance, the group (0, 2, 4, 6, 8, 10, ... 2N-2) will be split into (0, 4, 8, ...) and (2, 6, 10, ...), and so on. If you write these numbers in binary notation, you'll see that the first split (odd/even) is done according to the lowest bit; the second split is done according to the second lowest bit, and so on. So if we start with the sequence of, say, 8 consecutive binary numbers:

000, 001, 010, 011, 100, 101, 110, 111

we will first scramble them like this:

[even] (000, 010, 100, 110), [odd] (001, 011, 101, 111)

then we'll scramble the groups:

((000, 100), (010, 110)), (001, 101), (011, 111))

which gives the result:

000, 100, 010, 110, 001, 101, 011, 111

This is equivalent to sorting the numbers in bit-reversed order--if you reverse bits in each number (for instance, 110 becomes 011, and so on), you'll get a set of consecutive numbers.

So this is how the FFT algorithm works (more precisely, this is the decimation-in-time in-place FFT algorithm).

1. Select N that is a power of two. You'll be calculating an N-point FFT.
2. Gather your samples into a buffer of size N
3. Sort the samples in bit-reversed order and put them in a complex N-point buffer (set the imaginary parts to zero)
4. Apply the first stage butterfly using adjacent pairs of numbers in the buffer
5. Apply the second stage butterfly using pairs that are separated by 2
6. Apply the third stage butterfly using pairs that are separated by 4
7. Continue butterflying the numbers in your buffer until you get to separation of N/2
8. The buffer will contain the Fourier transform

Implementation

We will start by initializing some data structures and pre-computing some constants in the constructor of the FFT object.

```
// Use complex numbers from Standard Library
#include <complex>
typedef std::complex<double> Complex;

// Points must be a power of 2
Fft::Fft (int Points, long sampleRate)
    : _Points (Points), _sampleRate (sampleRate)
{
    _sqrtPoints = sqrt((double)_Points);
    // calculate binary log
    _logPoints = 0;
    Points--;
    while (Points != 0)
    {
        Points >>= 1;
        _logPoints++;
    }
    // This is where the original samples will be stored
    _aTape = new double [_Points];
    for (int i = 0; i < _Points; i++)
        _aTape[i] = 0;
    // This is the in-place FFT buffer
    _X = new Complex [_Points];

    // Precompute complex exponentials for each stage
    _W = new Complex * [_logPoints+1];
    int _2_l = 2;
    for (int l = 1; l <= _logPoints; l++)
    {
        _W[l] = new Complex [_Points];

        for ( int i = 0; i < _Points; i++ )
        {
            double re =  cos (2. * PI * i / _2_l);
            double im = -sin (2. * PI * i / _2_l);
            _W[l][i] = Complex (re, im);
        }
        _2_l *= 2;
    }

    // prepare bit-reversed mapping
    _aBitRev = new int [_Points];
    int rev = 0;
    int halfPoints = _Points/2;
    for (i = 0; i < _Points - 1; i++)
    {
        _aBitRev[i] = rev;
        int mask = halfPoints;
        // add 1 backwards
        while (rev >= mask)
```

```
        {
            rev -= mask; // turn off this bit
            mask >>= 1;
        }
        rev += mask;
    }
    _aBitRev [_Points-1] = _Points-1;
}
```

The FFT buffer is filled with samples from the "tape" in bit-reversed order

```
    for (i = 0; i < _Points; i++)
        PutAt (i, _aTape[i]);
```

The bit reversal is done inside *PutAt*, which also converts real samples into complex numbers (with the imaginary part set to zero):

```
void Fft::PutAt (int i, double val)
{
    _X [_aBitRev[i]] = Complex (val);
}
```

The calculation of the FFT is relatively simple

```
//
//              0   1   2   3   4   5   6   7
//  level   1
//  step    1                                   0
//  increm  2                                 W
//  j = 0       <--->   <--->   <--->   <--->   1
//  level   2
//  step    2
//  increm  4                                   0
//  j = 0       <------->       <------->      W      1
//  j = 1           <------->       <------->   2   W
//  level   3                                       2
//  step    4
//  increm  8                                   0
//  j = 0       <---------------->              W      1
//  j = 1           <---------------->           3   W      2
//  j = 2               <---------------->           3   W      3
//  j = 3                   <---------------->           3   W
//                                                           3
//

void Fft::Transform ()
{
    // step = 2 ^ (level-1)
    // increm = 2 ^ level;
    int step = 1;
    for (int level = 1; level <= _logPoints; level++)
    {
        int increm = step * 2;
        for (int j = 0; j < step; j++)
```

```
        {
            // U = exp ( - 2 PI j / 2 ^ level )
            Complex U = _W [level][j];
            for (int i = j; i < _Points; i += increm)
            {
                // in-place butterfly
                // Xnew[i]      = X[i] + U * X[i+step]
                // Xnew[i+step] = X[i] - U * X[i+step]
                Complex T = U;
                T *= _X [i+step];
                _X [i+step] = _X[i];
                _X [i+step] -= T;
                _X [i] += T;
            }
        }
        step *= 2;
    }
}
```

The variable *step* is the "spread" of the butterfly--distance between the two inputs of the butterfly. It starts with 1 and doubles at every level.

At each level we have to calculate *step* bunches of butterflies, each bunch consisting of butterflies separated by the distance of *increm* (*increm* is twice the *step*). The calculation is organized into bunches, because each bunch shares the same multiplier W.

The source of the FFT class is available for downloading. You can also download the sources of our real-time Frequency Analyzer which uses the FFT algorithm to analyze the sound acquired through your PC's microphone (if you have one).