A Simple and Efficient FFT Implementation in C++: Part I

By Vlodymyr Myrnyy, May 10, 2007

This article describes a new efficient implementation of the Cooley-Tukey fast Fourier transform (FFT) algorithm using C++ template metaprogramming. Thank to the recursive nature of the FFT, the source code is more readable and faster than the classical implementation. The efficiency is proved by performance benchmarks on different platforms.

Introduction

Fast Fourier Transformation (FFT) is not only a fast method to compute digital Fourier transformation (DFT)—having a complexity `O(Nlog(N))` (where `N` must be power of 2, $N=2^P$), it is a way to linearize many kinds of real mathematical problems of nonlinear complexity using the idiom "divide and conquer."

The discrete Fourier transform f_n of the `N` points signal x_n is defined as a sum:

$$f_k = \sum_{j=0}^{N-1} g^{jk} x_j, \quad g = e^{2\pi i/N},$$

Example 1.

where i is the complex unity. Put simply, the formula says that an algorithm for the computing of the transform will require $O(N^2)$ operations. But the Danielson-Lanczos Lemma (1942), using properties of the complex roots of unity `g`, gave a wonderful idea to construct the Fourier transform recursively (Example 1).

$$f_k = f_k^e + g^k f_k^o, \quad k = 0 \ldots N-1,$$

Example 2.

where f_k^e denotes the `k`-th component of the Fourier transform of length `N/2` formed from the even components of the original x_j, while f_k^o is the corresponding transform formed from the odd components. Although `k` in the last line of Example 2 varies from 0 to `N`-1, the transforms f_k^e and f_k^o are periodic in `k` with length `N/2`. The same formula applied to the transforms f_k^e and f_k^e reduces the problem to the transforms of length `N/4` and so on. It means, if `N` is a power of 2, the transform will be computed with a linear complexity `O(Nlog(N))`. More information on the mathematical background of the FFT and advanced algorithms, which are not limited to $N=2^P$, can be found in many books, e.g. [3,4].

I would like to start with the simplest recursive form of the algorithm, that follows directly from the relation in Example 2:

```
FFT(x) {
  n=length(x);
  if (n==1) return x;
  m = n/2;
  X = (x_{2j})_{j=0}^{m-1};
  Y = (x_{2j+1})_{j=0}^{m-1};
  X = FFT(X);
  Y = FFT(Y);
  U = (X_{k mod m})_{k=0}^{n-1};
  V = (g^{-k}Y_{k mod m})_{k=0}^{n-1};
```

```
    return U+V;
}
```

This algorithm should give only a first impression of the FFT construction. The `FFT(x)` function is called twice recursively on the even and odd elements of the source data. After that some transformation on the data is performed. The recursion ends if the data length becomes 1.

This recursion form is instructive, but the overwhelming majority of FFT implementations use a loop structure first achieved by Cooley and Tukey [2] in 1965. The Cooley-Tukey algorithm uses the fact that if the elements of the original length `N` signal `x` are given a certain "bit-scrambling" permutation, then the FFT can be carried out with convenient nested loops. The scrambling intended is reverse-binary reindexing, meaning that x_j gets replaced by x_k, where `k` is the reverse-binary representation of `j`. For example, for data length $N=2^5$, the element x_5 must be exchanged with $x_{\{20\}}$, because the binary reversal of $5=00101_2$ is $10100_2=20$. The implementation of this data permutation will be considered later, since it has been a minor part of the whole FFT. A most important observation is that the Cooley-Tukey scheme actually allows the FFT to be performed in place, that is, the original data `x` is replaced, element by element, with the FFT values. This is an extremely memory-efficient way to proceed with large data. Listing One shows the classical implementation of the Cooley-Tukey algorithm from Numerical Recipes in C++ [5], p.513.

Listing One

```



```

```
void four1(double* data, unsigned long nn)
{
    unsigned long n, mmax, m, j, istep, i;
    double wtemp, wr, wpr, wpi, wi, theta;
    double tempr, tempi;

    n = nn<<1;
    j=1;
    for (i=1; i<n; i+=2) {
        if (j>i) {
            swap(data[j-1], data[i-1]);
            swap(data[j], data[i]);
        }
        m = nn;
        while (m>=2 && j>m) {
            j -= m;
            m >>= 1;
        }
        j += m;
    };

    mmax=2;
    while (n>mmax) {
        istep = mmax<<1;
        theta = -(2*M_PI/mmax);
        wtemp = sin(0.5*theta);
        wpr = -2.0*wtemp*wtemp;
        wpi = sin(theta);
        wr = 1.0;
        wi = 0.0;
        for (m=1; m < mmax; m += 2) {
            for (i=m; i <= n; i += istep) {
                j=i+mmax;
                tempr = wr*data[j-1] - wi*data[j];
                tempi = wr * data[j] + wi*data[j-1];
                data[j-1] = data[i-1] - tempr;
                data[j] = data[i] - tempi;
                data[i-1] += tempr;
                data[i] += tempi;
            }
            wtemp=wr;
            wr += wr*wpr - wi*wpi;
            wi += wi*wpr + wtemp*wpi;
        }
        mmax=istep;
    }
}
```

The initial signal is stored in the array `data` of length `2*nn`, where each even element corresponds to the real part and each odd element to the imaginary part of a complex number.