Fast Fourier transform

A fast Fourier transform (FFT) is an efficient algorithm to compute the discrete Fourier transform (DFT).

For a set of complex numbers x_n, $n = 0, \ldots, N-1$, the DFT is defined as a set of complex numbers c_k,

$$c_k = \sum_{n=0}^{N-1} x_n e^{-2\pi i \frac{nk}{N}} \;, k = 0, \ldots, N-1 \;. \tag{1}$$

The inverse DFT is given by

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} c_k e^{+2\pi i \frac{nk}{N}} \;. \tag{2}$$

These transformations can be viewed as expansion of the vector x_n in terms of the orthogonal basis of vectors $e^{2\pi i \frac{kn}{N}}$,

$$\sum_{n=0}^{N-1} \left(e^{2\pi i \frac{kn}{N}}\right)\left(e^{-2\pi i \frac{k'n}{N}}\right) = N\delta_{kk'} \tag{3}$$

The DFT represent the amplitude and phase of the different sinusoidal components in the input data x_n.

The DFT is widely used in different fields, like spectral analysis, data compression, solution of partial differential equations and others.

Cooley-Tukey algorithm

In its simplest incarnation this algorithm re-expresses the DFT of size $N = 2M$ in terms of two DFTs of size M,

$$\begin{aligned} c_k &= \sum_{n=0}^{N-1} x_n e^{-2\pi i \frac{nk}{N}} \\ &= \sum_{m=0}^{M-1} x_{2m} e^{-2\pi i \frac{mk}{M}} + e^{-2\pi i \frac{k}{N}} \sum_{m=0}^{M-1} x_{2m+1} e^{-2\pi i \frac{mk}{M}} \\ &= \begin{cases} c_k^{(\text{even})} + e^{-2\pi i \frac{k}{N}} c_k^{(\text{odd})} & , k < M \\ c_{k-M}^{(\text{even})} - e^{-2\pi i \frac{k-M}{N}} c_{k-M}^{(\text{odd})} & , k \geq M \end{cases} , \end{aligned} \tag{4}$$

where $c^{(\text{even})}$ and $c^{(\text{odd})}$ are the DFTs of the even- and odd-numbered sub-sets of x.

This re-expression of a size-N DFT as two size-$\frac{N}{2}$ DFTs is sometimes called the Danielson-Lanczos lemma. The exponents $e^{-2\pi i \frac{k}{N}}$ are called *twiddle factors*.

The operation count by application of the lemma is reduced from the original N^2 down to $2(N/2)^2 + N/2 = N^2/2 + N/2 < N^2$.

For $N = 2^p$ Danielson-Lanczos lemma can be applied recursively until the data sets are reduced to one datum each. The number of operations is then reduced to $O(N \ln N)$ compared to the original $O(N^2)$. The established library FFT routines, like FFTW and GSL, further reduce the operation count (by a constant factor) using advanced programming techniques like precomputing the twiddle factors, effective memory management and others.

Multidimensional DFT

For example, a two-dimensional set of data $x_{n_1 n_2}$, $n_1 = 1 \ldots N_1$, $n_2 = 1 \ldots N_2$ has the discrete Fourier transform

$$c_{k_1 k_2} = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x_{n_1 n_2} e^{-2\pi i \frac{n_1 k_1}{N_1}} e^{-2\pi i \frac{n_2 k_2}{N_2}} \;. \tag{5}$$

C implementation

```
#include<complex.h>
#include<math.h>
#include<stdlib.h>
#define PI 3.14159265358979323846264338327950288
void dft (int N, complex* x, complex* c, int sign){
  complex w = cexp(sign*2*PI*I/N);
  for(int k=0;k<N;k++){
    complex sum=0; for(int n=0;n<N;n++) sum+=x[n]*cpow(w,n*k);
    c[k]=sum/sqrt(N);}
}//end dft
void fft (int N, complex* x, complex* c, int sign){
  if(N%2==0){
    complex w = cexp(sign*2*PI*I/N);
    int M=N/2;
    complex*xo =(complex*)malloc(M*sizeof(complex));
    complex*co =(complex*)malloc(M*sizeof(complex));
    complex*xe =(complex*)malloc(M*sizeof(complex));
    complex*ce =(complex*)malloc(M*sizeof(complex));
    for(int m=0;m<M;m++){xo[m]=x[2*m+1];xe[m]=x[2*m];}
    fft(M,xo,co,sign); fft(M,xe,ce,sign);
    for(int k=0;k<M;k++) c[k]=(ce[k]+cpow(w,k)*co[k])/sqrt(2);
    for(int k=M;k<N;k++) c[k]=(ce[k-M]+cpow(w,k)*co[k-M])/sqrt(2);
    free(xo);free(co);free(xe);free(ce);
    }
  else dft(N,x,c,sign);
}//end fft
```