1 Fast Fourier transform

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

Computing DFT of N points in the naïve way, using the definition, takes $O(N^2)$ arithmetic operations, while an FFT can compute the same result in only $O(N \log N)$ operations. The difference in speed can be substantial, especially for large data sets. This improvement made many DFT-based algorithms practical.

Since the inverse of a DFT is also a DFT, any FFT algorithm can be used in for the inverse DFT as well.

The most well known FFT algorithms, like the Cooley-Tukey algorithm [1], depend upon the factorization of N. However, there are FFTs with $O(N \log N)$ complexity for all N, even for prime N.

1.1 Discrete Fourier Transform

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

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

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 $\mathbf{x} = \{x_n\}_{n=0\dots N-1}$ in terms of the orthogonal basis of vectors $\mathbf{v}_k$,

$$\mathbf{v}_k = \{e^{\frac{2\pi i k}{N} n}\}_{n=0\dots N-1}, \tag{3}$$

$$\mathbf{v}_k^\star \cdot \mathbf{v}_{k'} = \sum_{n=0}^{N-1} \left(e^{-2\pi i \frac{kn}{N}}\right) \left(e^{2\pi i \frac{k'n}{N}}\right) = \sum_{n=0}^{N-1} e^{2\pi i \frac{(k'-k)n}{N}} = N\delta_{kk'} \ . \tag{4}$$

The DFT represents 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.

1.1.1 Applications

Data compression Several lossy (that is, with certain loss of information) image and sound compression methods employ DFT as an approximation for the Fourier series. The signal is discretized and transformed, and then the Fourier coefficients of high/low frequencies, which are assumed to be unnoticeable, are discarded. The decompressor computes the inverse transform based on this reduced number of Fourier coefficients.

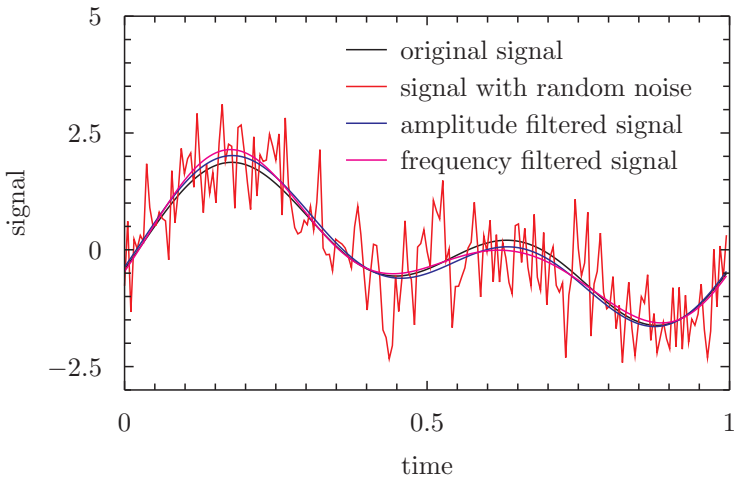

Figure 1: Noisy signal filtering using DFT

Noise filtering DFT can be used to attemt to filter out noise from a noisy signal. First, the signal is Fourier transformed. Then either the Fourier components with small amplitude are removed (set to zero) or the Fourier components with high frequencies are removed (set to zero). The filtered signal is then given as the inverse DFT of the modified set of Fourier components. And example is shown on Figure 1.

Partial differential equations Discrete Fourier transforms are often used to solve partial differential equations, where the DFT is used as an approximation for the Fourier series (which is recovered in the limit of infinite N). The advantage of this approach is that it expands the signal in complex exponentials e^{inx}, which are eigenfunctions of differentiation: $\frac{d}{dx}e^{inx} = ine^{inx}$. Thus, in the Fourier representation, differentiation is simply multiplication by in.

A linear differential equation with constant coefficients is transformed into an easily solvable algebraic equation. One then uses the inverse DFT to transform the result back into the ordinary spatial representation. This approach belongs to the group of *spectral methods*.

Convolution and Deconvolution FFT can be used to efficiently compute convolutions of two sequences. A convolution is the pairwise product of elements from two different sequences, such as in multiplying two polynomials or multiplying two long integers.

Another example comes from data acquisition processes where the detector introduces certain (typically Gaussian) blurring to the sampled signal. A reconstruction of the original signal can be obtained by deconvoluting the acquired signal with the detector's blurring function.

1.2 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,

$$\begin{aligned} c_k &= \sum_{n=0}^{N-1} x_n e^{-2\pi i \frac{nk}{N}} \\ &= \sum_{m=0}^{M-1} x_{2m} e^{-2\pi i \frac{mk}{M}} + e^{-2\pi i \frac{k}{N}} \sum_{m=0}^{M-1} x_{2m+1} e^{-2\pi i \frac{mk}{M}} \\ &= \begin{cases} c_k^{(\text{even})} + e^{-2\pi i \frac{k}{N}} c_k^{(\text{odd})} & , k < M \\ c_{k-M}^{(\text{even})} - e^{-2\pi i \frac{k-M}{N}} c_{k-M}^{(\text{odd})} & , k \geq M \end{cases} , \end{aligned} \tag{5}$$

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, see Table 1. 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.

1.3 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

$$c_{k_1 k_2} = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x_{n_1 n_2} e^{-2\pi i \frac{n_1 k_1}{N_1}} e^{-2\pi i \frac{n_2 k_2}{N_2}} . \tag{6}$$

References

[1] John.WTukey James.W.Cooley. An algorithm for the machine calculation of complex fourier series. *Math. Comput.*, 19(90):297–301, 1965.

Table 1: Csharp-implementation of the Cooley-Tukey algorithm

```
using static System.Math;
using static complex;
using static cmath;

public static partial class matlib{

public static void dfts
(int sign,int N,complex[] x,int ix,int stride,complex[] c,int ic){
   for(int k=0;k<N;k++){
      c[ic+k]=0;
      for(int n=0;n<N;n++)
         c[ic+k]+=x[ix+n*stride]*exp(sign*2*PI*I*n*k/N);
   }
}

public static void ffts
(int sign,int N,complex[] x,int ix,int stride,complex[] c,int ic){
   if(N==1) c[ic+0]=x[ix+0];
   else if(N%2==0){
      ffts(sign,N/2,x,ix+0     ,2*stride,c,ic+0   );
      ffts(sign,N/2,x,ix+stride,2*stride,c,ic+N/2);
      for(int k=0;k<N/2;k++){
         complex p=c[ic+k], q=exp(sign*2*PI*I*k/N)*c[ic+k+N/2];
         c[ic+k    ]=p+q;
         c[ic+k+N/2]=p-q;
         }
      }
   else dfts(sign,N,x,ix,stride,c,ic);
   }

public static complex[] fft(complex[] x){
   int N=x.Length;
   var c=new complex[N];
   ffts(-1,N,x,0,1,c,0);
   return c;
   }

public static complex[] ift(complex[] c){
   int N=c.Length;
   var x=new complex[N];
   ffts(+1,N,c,0,1,x,0);
   for(int i=0;i<N;i++)x[i]/=N;
   return x;
   }

}//class
```