1 Fast Fourier transform

Fast Fourier transform (FFT) is an efficient algorithm to compute the discrete Fourier transform (DFT). Computing DFT of a size-N vector 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 for the inverse DFT as well.

The most well known FFT algorithms, like the Cooley-Tukey algorithm [**?**], 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 $\{y_n\}_{n=0,\dots,N-1}$ the DFT is defined as a set of complex numbers $\{Y_k\}_{k=0,\dots,N-1}$ given as

$$Y_k = \sum_{n=0}^{N-1} y_n e^{-2\pi i \frac{nk}{N}} \ . \tag{1}$$

The inverse DFT is given by

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

These transformations can be viewed as expansion of the vector $\mathbf{y} = \{y_n\}$ in terms of the orthogonal basis of vectors $\mathbf{v}_k$,

$$\mathbf{v}_k = \{e^{\frac{2\pi i k}{N} n}\} \tag{3}$$

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

The DFT represents the amplitude and phase of the different sinusoidal components in the input data y_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.

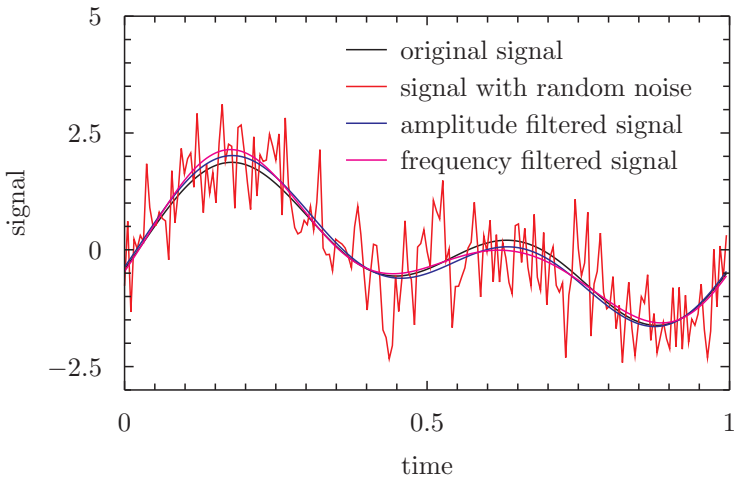

Figure 1: Noisy signal filtering using DFT

Noise filtering DFT can be used to attempt 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.

Differential equations Discrete Fourier transforms are often used to solve 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 $e^{i\omega t}$, which are eigenfunctions of differentiation: $\frac{d}{dt}e^{i\omega t} = i\omega e^{i\omega t}$. Thus in the Fourier representation (in the frequency domain) differentiation is simply multiplication by $i\omega$.

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 representation (time domain). This approach belongs to the group of *spectral methods*.

For example, consider the following first-order differential equation with a periodic boundary condition,

$$\frac{dy(t)}{dt} + 2y(t) = f(t) \ , \tag{5}$$

where $f(t)$ is a given periodic force-function with period T and where we try to find a solution $y(t)$ with the same period T.

First, we discretize the time-variable over one period of T and sample both $y(t)$

and $f(t)$ at N equally spaced points

$$t_n = n\Delta t \, , n = 0, \ldots, N-1 \tag{6}$$

where $\Delta t = T/N$. That gives us the discrete sequences $\{y_n = y(t_n)\}$ and $\{f_n = f(t_n)\}$.

Now, we apply the DFT to both $\{y_n\}$ and $\{f_n\}$,

$$Y_k = \sum_{n=0}^{N-1} y_n e^{-2\pi i k n/N} \, , \; F_k = \sum_{n=0}^{N-1} f_n e^{-2\pi i k n/N} \, . \tag{7}$$

If the DFT $\{Y_k\}$ is known, the function $y(t)$ is approximated as

$$y(t) \approx \sum_{k=0}^{N-1} Y_k e^{i\omega_k t} \, , \tag{8}$$

where $\omega_k = 2\pi k/T$. The derivative $dy(t)/dt$ in the same approximation is given as

$$\frac{dy(t)}{dt} \approx \sum_{k=0}^{N-1} i\omega_k Y_k e^{i\omega_k t} \, , \tag{9}$$

that is, the DFT of the derivative is given as

$$\text{DFT}\left(\frac{dy(t)}{dt}\right) = \left\{ i\frac{2\pi k}{T} Y_k \right\} \, . \tag{10}$$

Now we can take the DFT of the entire differential equation,

$$i\frac{2\pi k}{T} Y_k + 2Y_k = F_k \, . \tag{11}$$

This algebraic equation is can be solved for Y_k,

$$Y_k = \frac{F_k}{2 + i\frac{2\pi k}{T}} \, . \tag{12}$$

Finally, we obtain the (approximation of the) solution $y(t)$ as the inverse DFT of the obtained $\{Y_k\}$,

$$y_n = \frac{1}{N} \sum_{k=0}^{N-1} Y_k e^{2\pi i k n/N} \, . \tag{13}$$

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.

One 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. Let us say that we measure the original signal $\{y_n\}$ with a detector that slightly mixes the neighboring samples. The measured signal $\{s_n\}$ is than given as a linear superposition of several points, something like

$$s_n = \frac{1}{10} y_{n-1} + \frac{8}{10} y_n + \frac{1}{10} y_{n+1} \,. \tag{14}$$

Generally this is written as a *convolution* of two sequences,

$$s_n = \sum_m y_m \epsilon_{n-m} \,, \tag{15}$$

where ϵ_{n-m} represents the detector efficiency function (a narrow bell-shape function for a good detector, approaching the delta-function for the perfect detector). Formally, if we have a size-N sequence $\{y_n\}$, the convolution is defined as

$$s_n = \sum_{m=0}^{N} y_m \epsilon_{(n-m) \bmod N} \,. \tag{16}$$

Now the interesting thing is that the DFT of a convolution of two sequences is given by a product of DFT's of the sequences. Indeed,

$$\begin{aligned} \mathrm{DFT}(\{s_n\})_k \doteq S_k &= \sum_n e^{-2\pi i k n/N} \sum_{m=0}^{N} y_m \epsilon_{(n-m) \bmod N} \\ &= \sum_{k'} \sum_n e^{-2\pi i (k-k') n/N} \frac{1}{N} Y_{k'} E_{k'} \\ &= Y_k E_k \,. \end{aligned} \tag{17}$$

Therefore the DFT of the original signal can be obtained by dividing the DFT of the measured signal by the DFT of the detector efficiency,

$$Y_k = \frac{S_k}{E_k} \,. \tag{18}$$

The original signal is then restored by inverse DFT of $\{S_k/E_k\}$. Note however that if the DFT of the detector efficiency $\{E_k\}$ has values close to zero for some frequencies, the division can significantly amplify noise present in the blurred signal $\{s_n\}$, leading to a noisy and unstable restoration. This is a common issue in deconvolution.

1.2 Cooley-Tukey algorithm

In its simplest incarnation this algorithm re-expresses the size $N = 2M$ discrete Fourier transform $\{c_k\} = \mathrm{DFT}\left(\{x_n\}_{n=0,\ldots,N}\right)$ in terms of two DFTs of size M,

$$\begin{aligned} c_k &= \sum_{n=0}^{N-1} x_n e^{-2\pi i \frac{nk}{N}} \\ &= \sum_{m=0}^{M-1} x_{2m} e^{-2\pi i \frac{mk}{M}} + e^{-2\pi i \frac{k}{N}} \sum_{m=0}^{M-1} x_{2m+1} e^{-2\pi i \frac{mk}{M}} \\ &= \begin{cases} c_k^{(\mathrm{even})} + e^{-2\pi i \frac{k}{N}} c_k^{(\mathrm{odd})} & , k < M \\ c_{k-M}^{(\mathrm{even})} - e^{-2\pi i \frac{k-M}{N}} c_{k-M}^{(\mathrm{odd})} & , k \geq M \end{cases} , \end{aligned} \tag{19}$$

where $c^{(\mathrm{even})}$ and $c^{(\mathrm{odd})}$ are the DFTs of the even- and odd-numbered sub-sets of $\{x_n\}$. 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[1] to $O(N \ln N)$ compared to the original $O(N^2)$.

A naïve implementation of the Cooley-Tukey algorithm in Python is shown in Table 1. However this implementation makes a lot of unnecessary list operations. A more refined implementation that works on thesame array is shown in Table 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] Let $T(N)$ be the number of operations to calculate size-N DFT using Cooley-Tukey algorithm. Applying the Lanczos-Danielson lemma once gives

$$T(N) = 2T(N/2) + O(N) \, . \tag{20}$$

Applying it again and again for $N = 2^p$ gives

$$\begin{aligned} T(N) &= 2T(N/2) + O(N) \\ &= 4T(N/4) + 2O(N) \\ &= 8T(N/8) + 3O(N) \\ &= \ldots \\ &= NT(1) + pO(N) \, . \end{aligned} \tag{21}$$

Now, $T(1) = 0$ as it takes no operations. That gives $T(N) = O(N \log_2 N)$.

Table 1: Naïve Python implementation of the Cooley-Tukey algorithm

```
import cmath

def dft (x, sign=-1) :
    N = len(x)
    w = cmath.exp(sign*2*cmath.pi*1j/N)
    return [sum( x[n]*w**(n*k) for n in range(N) ) for k in range(N)]

def fft (x, sign=-1) :
    N = len(x)
    if N % 2 == 0 :
        M = N//2
        ce = fft(x[0::2], sign)
        co = fft(x[1::2], sign)
        w = cmath.exp(sign*2*cmath.pi*1j/N)
        c = [ce[k] +w**k*co[k]    for k in range(M)   ] \
          +[ce[k-M]+w**k*co[k-M] for k in range(M,N)]
        return c
    else : return dft(x,sign)

def ift (x) : return [z/len(x) for z in fft(x,+1)]
```

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

$$c_{k_1 k_2} = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x_{n_1 n_2} e^{-2\pi i \frac{n_1 k_1}{N_1}} e^{-2\pi i \frac{n_2 k_2}{N_2}} \ . \tag{22}$$

Table 2: 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
```