A Fast Fourier Transform Library

Xiankun Xu

June 23, 2018

Contents

1 Introduction

The head file *FFT_handler.hpp* is a C++ implementation of the fast Fourier transform (FFT) algorithms. It can do the following operations:

1. Discrete Fourier transform (DFT) and inverse discrete Fourier transform (IDFT);
2. Radix-2 decimation in frequency fast Fourier transform(FFT) and inverse Fourier transform(IFFT);
3. Radix-2 decimation in time fast Fourier transform and inverse Fourier transform;
4. Radix-3 decimation in frequency fast Fourier transform and inverse Fourier transform;
5. Discrete convolution;
6. Numerical calculation of the Fourier transform of a given function.

The C++ code utilizes the template feature, makes the functions can take operation on various containers (such as array, STL vector, STL deque), and various data types (such as integer, double, and complex double, etc.).

2 Use of the functions

10 functions have been defined in this head file. To use the head file, include it in your source file:

```
#include "FFT_handler.hpp"
```

In the following, I will describe how to use each functions.

2.1 Discrete Fourier transform

This section describe how to use the discrete Fourier transform (DFT) and inverse discrete Fourier transform (IDFT) functions. The DFT and IDFT are defined as follows:

$$\begin{aligned} &\text{DFT:} && X(k) = \sum_{j=0}^{N-1} x(j) W_N^{kj} && k = 0, 1, \cdots, N-1 \\ &\text{IDFT:} && x(j) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) W_N^{-jk} && j = 0, 1, \cdots, N-1 \end{aligned} \tag{1}$$

where x and X are two sets of data with size N. $W_N = \exp(-\mathrm{i}\, 2\pi/N)$, where i is the imaginary number.

The signature of the DFT function is as follows:

```
template <typename iter_data_type, typename iter_complexdouble_type>
void dft(iter_data_type x_beg, std::size_t N, iter_complexdouble_type X_beg)
```

This *dft* function will do the naive DFT on a data set x with size N. x_beg is an iterator (pointer, STL iterators) pointing to the first element of x. The time complexity is $\mathcal{O}(N^2)$. The result will be written to a data set X, where X_beg is an iterator to the first element of X. x and X can be any contiguous containers such as array, STL vector, and STL deque. The data type in x can be int, double, complex int, and complex double. While the data type in X must be in complex double. Make sure X can hold at least N complex double data before using this function.

The signature of the IDFT function is as follows:

```
template <typename iter_data_type, typename iter_complexdouble_type>
void idft(iter_data_type X_beg, size_t N, iter_complexdouble_type x_beg)
```

This *idft* function will do the naive IDFT on a data set X with size N. X_beg is an iterator (pointer, STL iterators) pointing to the first element of X. The time complexity is $\mathcal{O}(N^2)$. The result will be written to a data set x, where x_beg is an iterator to the first element of x. x and X can be any contiguous containers such as array, STL vector, and STL deque. The data type in X can be int, double, complex int, and complex double. While the data type in x must be in complex double. Make sure x can hold at least N complex double data before using this function.

Coding example of using these two functions:

```
FFT_handler ffthandler;
unsigned N(100);
double x[N];
vector<complex<double> > X(N);
//here remember to assign data into x;
ffthandler.dft(x, N, X.begin()); //DFT from x to X
complex<double> x1[N];
ffthandler.idft(X.begin(), N, x1); //IDFT from X to x1
//now can compare x and x1 to see if the DFT and IDFT is correct, note x1 is complex double
```

2.2 Radix-2 decimation in frequency (Radix-2 DIF)

This section describe how to use the Radix-2 decimation in frequency FFT and IFFT algorithms to calculate the DFT and IDFT of a given data. The time complexity is $\mathcal{O}(Nlog_2N)$.

The signature of the Radix-2 DIF FFT function is as follows:

```
template <typename iter_data_type, typename iter_complexdouble_type>
void fft_radix2_dif(iter_data_type x_beg, size_t N, iter_complexdouble_type X_beg)
```

This *fft_radix2_dif* function will implement the Radix-2 decimation in frequency FFT algorithm on a data set x with size N , where N must be an integer power of 2. x_beg is an iterator (pointer, STL iterators) pointing to the first element of x. The result will be written to a data set X, where X_beg is an iterator to the first element of X. x and X can be any contiguous containers such as array, STL vector, and STL deque. The data type in x can be int, double, complex int, and complex double. While the data type in X must be in complex double. Make sure X can hold at least N complex double data before using this function.

The signature of the Radix-2 DIF IFFT function is as follows:

```
template <typename iter_data_type, typename iter_complexdouble_type>
void ifft_radix2_dif(iter_data_type X_beg, size_t N_data, iter_complexdouble_type x_beg)
```

This *ifft_radix2_dif* function will implement the Radix-2 decimation in frequency IFFT algorithm on a data set X with size N , where N must be an integer power of 2. X_beg is an iterator (pointer, STL iterators) pointing to the first element of X. The result will be written to a data set x, where x_beg is an iterator to the first element of x. x and X can be any contiguous containers such as array, STL vector, and STL deque. The data type in X can be int, double, complex int, and complex double. While the data type in x must be in complex double. Make sure x can hold at least N complex double data before using this function.

Coding example of using these two functions:

```
FFT_handler ffthandler;
unsigned p(6), N( pow(2,p) );
vector<double> x(N);
complex<double> X[N];
//here remember to assign data into x;
ffthandler.fft_radix2_dif(x.begin(), N, X); //DFT from x to X
complex<double> x1[N];
ffthandler.ifft_radix2_dif(X, N, x1); //IDFT from X to x1
//now can compare x and x1 to see if the DFT and IDFT is correct, note x1 is complex double
```

2.3 Radix-2 decimation in time (Radix-2 DIT)

The signature of the Radix-2 DIT FFT function is as follows:

```
template <typename iter_data_type, typename iter_complexdouble_type>
void fft_radix2_dit(iter_data_type x_beg, size_t N, iter_complexdouble_type X_beg)
```

The signature of the Radix-2 DIT IFFT function is as follows:

```
template <typename iter_data_type, typename iter_complexdouble_type>
void ifft_radix2_dit(iter_data_type X_beg, size_t N_data, iter_complexdouble_type x_beg)
```

The use of *fft_radix2_dit* and *ifft_radix2_dit* are exactly the same as *fft_radix2_dif* and *ifft_radix2_dif*.

2.4 Radix-3 decimation in frequency (Radix-3 DIF)

The signature of the Radix-3 DIF FFT function is as follows:

```
template <typename iter_data_type, typename iter_complexdouble_type>
void fft_radix3_dif(iter_data_type x_beg, size_t N, iter_complexdouble_type X_beg)
```

The signature of the Radix-3 DIF IFFT function is as follows:

```
template <typename iter_data_type, typename iter_complexdouble_type>
void ifft_radix3_dif(iter_data_type X_beg, size_t N_data, iter_complexdouble_type x_beg)
```

The use of *fft_radix3_dif* and *ifft_radix3_dif* are exactly the same as *fft_radix2_dif* and *ifft_radix2_dif*. However, to use Radix-3 DIF, the number of data N must be integer power of 3.

2.5 Discrete convolution

The discrete convolution about two sets of data x and y is defined as follows:

$$c(n) = \sum_{m=0}^{M-1} x(m)y(n-m) \qquad n = 0, 1, \cdots N-1 \tag{2}$$

where x has data size M, y has data size $N - M + 1$, and c has data size N.

The signature of the FFT implementation of the discrete convolution is as follows:

```
template <typename T>
void convolution(const vector<T>& x, const vector<T>& y, vector<T>& c)
```

This *convolution* function will implement the Radix-2 decimation in frequency FFT and IFFT algorithms to calculate the discrete convolution of x and y. The data size must be fit, that is, x has data size M, y has data size $N - M + 1$, and c has data size N, where M and N can be any positive integers. The convolution will be written to c. The time complexity is $\mathcal{O}((M+N)log_2(M+N))$.

2.6 Numerical calculation of the Fourier transform of a given function

For a given function $h(t)$, its Fourier transform $H(f)$ is defined as:

$$H(f) = \int_{-\infty}^{\infty} h(t)\mathrm{e}^{-\mathrm{i}2\pi ft}dt \tag{3}$$

To make the Fourier integral be convergent, it requires $h(t) \to 0$ as $t \to \infty$. The numerical calculation of $H(f)$ is to assume $h(t) = 0$ in ranges outside of $[tl, tr]$. And the numerical approximation (finite difference integration) will be applied on

$$H(f) = \int_{tl}^{tr} h(t)\mathrm{e}^{-\mathrm{i}2\pi ft}dt \tag{4}$$

The range $[tl, tr]$ is uniformly divided into N_t intervals, at each interval, take the middle point to calculate $h(t_i)$, and Eq. (4) is approximated as:

$$H(f) = \frac{b-a}{N_t} \sum_{i=0}^{N_t} h(t_i)\mathrm{e}^{-\mathrm{i}2\pi ft_i} \tag{5}$$

This function *numerical_fourier_transform* will utilize the Radix-2 DIF FFT and IFFT algorithms, to calculate N_f data of $H(f_i)$, where f_i is evenly distributed in range $[fl, fr]$.

The signature of the numerical Fourier transform function is as follows:

```
template <typename retn_type, typename iter_complexdouble_type>
void numerical_fourier_transform(retn_type(*func)(double), double tl, double tr, size_t Nt, double
    fl, double fr, iter_complexdouble_type H_beg, size_t Nf)
```

Where *func* is a function pointer which takes a double as argument, and its return type can be int, double, complex int, and complex double. tl, tr, N_t, fl, fr, N_f have been described in above. The $H(f_i)$ will be written to a contiguous container which first element is pointed by H_beg. The time complexity of this function is $\mathcal{O}((N_t + N_f)log_2(N_t + N_f))$.

Coding example of using the *numerical_fourier_transform* function:

```
double hfun(double t)
{
   double alpha(20);
   return 0.5*alpha*exp(-alpha*abs(t));
}

double tl(-2),tr(2);
unsigned N_t(10000);
double fl(-20),fr(20);
unsigned N_f(20);
vector<complex<double> > H(N_f);
FFT_handler ffthandler;
ffthandler.numerical_fourier_transform(hfun,tl,tr,N_t,fl,fr,H.begin(),N_f);
```