Free small FFT in multiple languages

Introduction

The fast Fourier transform (FFT) is a versatile tool for digital signal processing (DSP) algorithms and applications. On this page, I provide a free implementation of the FFT in multiple languages, small enough that you can even paste it directly into your application (you don't need to treat this code as an external library).

Also included is a fast circular convolution function based on the FFT. Note that the FFT, with a bit of pre- and postprocessing, can quickly calculate the discrete cosine transform (DCT), which is used in many multimedia compression algorithms.

My implementation has a reasonable amount of optimization (such as building trigonometric tables), but is not intended to squeeze every last drop of performance. Instead, the implementation is optimized for clarity and conciseness. You're welcome to add more optimizations on your own if you wish.

Math documentation

Note: In the text below, all input and output vectors are complex, and n is the length of each vector.

The forward FFT is based on this formula: $\mathrm{Out}(k) = \sum_{t=0}^{n-1} \mathrm{In}(t)\, e^{-2\pi itk/n}$.

The inverse FFT is based on this formula: $\mathrm{Out}(k) = \sum_{t=0}^{n-1} \mathrm{In}(t)\, e^{2\pi itk/n}$.

This FFT does not perform any scaling. So for a vector of length n, after performing a transform and an inverse transform on it, the result will be the original vector multiplied by n (plus approximation errors).

The circular convolution is based on this formula: $\mathrm{Out}(j) = \sum_{k=0}^{n-1} \mathrm{X}((j-k) \bmod n)\, \mathrm{Y}(k)$.

In other words, $\mathrm{Out}(j)$ is the sum of all the products $\mathrm{X}(a)\mathrm{Y}(b)$ where the indices satisfy $a + b \equiv j \bmod n$.

Source code and API

The test code are runnable main programs that check the FFT and fast convolution for correctness against a naive algorithm. They also demonstrate how this FFT code is called. You do not need to include the test code into your own program. Also, feel free to strip down the set of FFT functions to the bare minimum that you need.

The radix-2 FFT code is essential since every function depends on it. The Bluestein FFT code can be omitted if you are not doing FFTs or convolutions of non-power-of-2 sizes. The convolution code can be omitted if you are not computing convolutions and you are not using the Bluestein FFT.

If you are running many FFTs on the same array length, please consider modifying the code to compute the cosine and sine tables once and reuse them. This is especially important when benchmarking the speed of the algorithm.

License summary (MIT License): Please keep the copyright notice and URL intact. You may use, edit, and publish the code for any purpose including commercial use. No warranty is provided by Nayuki. The official license text is included in the source code.

Java

Download: Fft.java, FftTest.java

- Forward FFT (wrapper) (in-place): `void transform(double[] real, double[] imag)`
- Inverse FFT (wrapper) (in-place): `void inverseTransform(double[] real, double[] imag)`
- Forward FFT (radix-2) (in-place): `void transformRadix2(double[] real, double[] imag)`
- Forward FFT (Bluestein) (in-place): `void transformBluestein(double[] real, double[] imag)`
- Circular convolution (real): `void convolve(double[] x, double[] y, double[] out)`
- Circular convolution (complex): `void convolve(double[] xreal, double[] ximag, double[] yreal, double[] yimag, double[] outreal, double[] outimag)`

When calling any of the above functions, all the argument arrays must have the same length.

JavaScript

Download: fft.js, fft-test.html

- Forward FFT (wrapper) (in-place): `transform(real, imag)`
- Inverse FFT (wrapper) (in-place): `inverseTransform(real, imag)`
- Forward FFT (radix-2) (in-place): `transformRadix2(real, imag)`
- Forward FFT (Bluestein) (in-place): `transformBluestein(real, imag)`
- Circular convolution (real): `convolveReal(x, y, out)`
- Circular convolution (complex): `convolveComplex(xreal, ximag, yreal, yimag, outreal, outimag)`

For each function above, the API is that every parameter is an array of floating-point numbers and the return value is nothing. When calling any of the above functions, all the argument arrays must have the same length. The API is almost the same as the Java version's.

Python

Download: fft.py, ffttest.py

- Forward FFT (wrapper): `transform(list of numeric, False)` returning a list of complex
- Inverse FFT (wrapper): `transform(list of numeric, True)` returning a list of complex
- Forward FFT (radix-2): `transform_radix2(list of numeric, False)` returning a list of complex
- Inverse FFT (radix-2): `transform_radix2(list of numeric, True)` returning a list of complex
- Forward FFT (Bluestein): `transform_bluestein(list of numeric, False)` returning a list of complex
- Inverse FFT (Bluestein): `transform_bluestein(list of numeric, True)` returning a list of complex
- Circular convolution (real): `convolve(list of int/long/float, list of int/long/float, True)` returning a list of float
- Circular convolution (complex): `convolve(list of numeric, list of numeric, False)` returning a list of complex

The *numeric* types are int, long, float, and complex. When calling any of the above functions, the returned list has the same length as the argument list(s). When calling `convolve()`, the two argument lists must have the same length. Note that this Python implementation *returns* the result instead of storing it back into the argument list(s); this is unlike the implementations for the other languages.

C#

Download: Fft.cs, FftTest.cs

- Forward FFT (wrapper) (in-place): `void Transform(double[] real, double[] imag)`
- Inverse FFT (wrapper) (in-place): `void InverseTransform(double[] real, double[] imag)`
- Forward FFT (radix-2) (in-place): `void TransformRadix2(double[] real, double[] imag)`
- Forward FFT (Bluestein) (in-place): `void TransformBluestein(double[] real, double[] imag)`
- Circular convolution (real): `void Convolve(double[] x, double[] y, double[] outreal)`
- Circular convolution (complex): `void Convolve(double[] xreal, double[] ximag, double[] yreal, double[] yimag, double[] outreal, double[] outimag)`

When calling any of the above functions, all the argument arrays must have the same length. If you look carefully, this codebase is extremely similar to the Java code it is based on.

C

Download: fft.c, fft.h, ffttest.c

- Forward FFT (wrapper) (in-place): `int transform(double real[], double imag[], size_t n)`
- Inverse FFT (wrapper) (in-place): `int inverse_transform(double real[], double imag[], size_t n)`
- Forward FFT (radix-2) (in-place): `int transform_radix2(double real[], double imag[], size_t n)`
- Forward FFT (Bluestein) (in-place): `int transform_bluestein(double real[], double imag[], size_t n)`
- Circular convolution (real): `int convolve_real(const double x[], const double y[], double out[], size_t n)`
- Circular convolution (complex): `int convolve_complex(const double xreal[], const double ximag[], const double yreal[], const double yimag[], double outreal[], double outimag[], size_t n)`

When calling any of the above functions, all the argument arrays must have the same length. Each function returns 1 to indicate success or 0 to indicate failure.

C++

Download: fft.cpp, fft.hpp, ffttest.cpp

- Forward FFT (wrapper) (in-place): `void transform(vector<double> &real, vector<double> &imag)`
- Inverse FFT (wrapper) (in-place): `void inverseTransform(vector<double> &real, vector<double> &imag)`
- Forward FFT (radix-2) (in-place): `void transformRadix2(vector<double> &real, vector<double> &imag)`
- Forward FFT (Bluestein) (in-place): `void transformBluestein(vector<double> &real, vector<double> &imag)`
- Circular convolution (real): `void convolve(const vector<double> &x, const vector<double> &y, vector<double> &out)`
- Circular convolution (complex): `void convolve(const vector<double> &xreal, const vector<double> &ximag, const vector<double> &yreal, const vector<double> &yimag, vector<double> &outreal, vector<double> &outimag)`

When calling any of the above functions, all the argument vectors must have the same length.

More info

- Wikipedia: Fast Fourier transform, Convolution theorem
- Wikipedia: Cooley-Tukey FFT algorithm, Bluestein's FFT algorithm
- Wikipedia: FFTW (a high-quality open-source library in C)

Categories: Programming, Math, Java, JavaScript, Python, C/C++ Last updated: 2016-04-19