Usage notes for the FFT and ComplexNumbers libraries

July 1994-January 1995

Table of Contents

Introduction

ComplexNumbers and FFT are two object libraries that implement complex numbers and the Fast Fourier Transform, respectively. The ComplexNumbers library can be used separate of the FFT library, but the reverse does not hold true.

A note on ANSI conformance

The ComplexNumbers and FFT libraries were compiled in Symantec C++ 7.0.3 for the Macintosh, with Strict ANSI conformance turned on. This means (hopefully) that the code for the libraries should easily port to other development systems and other computer systems. Source code is included with the libraries so they may be compiled and built into libraries for other development systems.

If you are re-compiling the ComplexNumbers library to utilize SANE (Standard Apple Numerics Environment), you may need to turn ANSI conformance off, because the SANE header file is not ANSI-conformant.

ComplexNumbers

About ComplexNumbers

ComplexNumbers is a library that implements complex numbers and complex number functions. The core component of ComplexNumbers is the class `Complex`. It uses overloaded operators to achieve the same level of flexibility with C++'s primitive types, such as `int` and `double`.

If you are familiar with ANSI C++, you probably know that ANSI C++ already defines a complex number library, named `complex` (notice the case difference). After considering this, the natural question is, why should one re-define a standard library? There were actually three reasons this was done.

The first pertains to the machine: the Macintosh. Looking at the class declaration of the ANSI `complex`, one will notice the operators `>>` and `<<`. These operators handle standard input/output through a console. The Macintosh has much better ways to handle standard I/O than through a console window.

The second reason has to do with the results returned by the `complex` library. I was getting inconsistent results calling functions from the ANSI math libraries, using Apple's SANE (Standard Apple Numerics Environment). I assumed this was a bug, and I have not had the incorrect results using my library.

The third reason has to do with the way the ANSI `complex` library was declared. All the overloaded operators that allow the `complex` class to be used like primitive types were declared as `friends`. The use of `friends` in C++ is generally discouraged when better alternatives are available (in this case, member functions). In addition, the use of the `long double` type permeates the declaration of the ANSI `complex` class. On Macintosh on PowerPC, the `long double` type is defined to be 128 bits long, and the operations are performed in *software*. This would be too slow for most applications, including the FFT. The floating-point type used in `Complex` is the `double`.

Using ComplexNumbers

Using the Complex class in your source code

To use the `Complex` class in your source code files, add the following three lines to your code:

```
#ifndef __COMPLEXNUMBERS__
#include "ComplexNumbers.h"
#endif
```

Make sure that the `ComplexNumbers.h` file is included *after* your standard headers and *before* any definitions begin.

Initialization

`Complex` is declared with three constructors. One takes no parameters and sets the real and imaginary components both to zero. The second takes two parameters and initializes the real and imaginary parts to the parameters. The last is a copy constructor. The following is an example of the initialization:

```
void f(void)
{
      Complex                 a;
      Complex                 b(6.12376, -3.1415926);
      Complex                 c(a);

      // a == c == (0.0, 0.0); b == (6.12376, -3.1415926)
      .
      .
      .
}
```

Assignment

The real and imaginary parts of a `Complex` object can be accessed through `Complex's public` accessor functions. The `Real()` and `Imag()` return *references* to the real and imaginary parts of a `Complex` object, respectively. Thus, they may be used on the right or left side of the equal sign. The `Re()` and `Im()` return *copies* of the real and imaginary parts of a `Complex` object, respectively. The following is an example of the usage of these member functions:

```
void UseComplex(Complex& x)
{
      double                        a;

      x.Real() = -4.2;

      a = x.Imag();

      a = atan2(x.Im(), x.Re());
}
```

Operators

`Complex` provides overloaded operators for all arithmetic operations. In effect, the `Complex` class can be used like other C++ primitive types, such as `int` and `float`.

Complex number functions

The ComplexNumbers library provides several functions that operate on the `Complex` type.

The `arg()` and `mod()` functions return the same value, performing the absolute value or modulus of a complex number z, defined to be $\sqrt{a^2 + \beta^2}$, where $z = \alpha + \iota\beta$.

The `arg()` function returns the phase of a complex number z, defined to be $\tan^{-1}\left(\frac{\beta}{\alpha}\right)$, where $z = \alpha + \iota\beta$.

The `Conj()` function returns the complex conjugate of a complex number z.

ComplexNumbers Reference

Data types

The `Complex` class is a concrete type that implements the concept of a complex number. It provides a full set of overloaded operators.

Complex number logic

The `Complex` class provides the logical operators `&&` and `||`. For a `Complex` object to be `true`, either or both of its components must be non-zero.

ComplexNumbers summary

```
//
//    File: ComplexNumbers.h
//    Copyright ©1994-1995 James Wilson
//    All rights reserved.
//
//    This file is a more Macintosh-compatible version of the complex
//    class used in the iostreams library. Among the changes, the size
//    of the real and imaginary components are now doubles instead of
//    long doubles. This is for PowerPC compatibility. I have replaced
//    all references to int with long. The << and >> I/O operators are
//    not in this class; the Macintosh interface has other and better
//    ways to handle standard I/O with the user. I have also done my
//    best to eliminate the use of friends and other questionable C++
//    constructs from this implementation. To match with with Macintosh
//    class naming style conventions, the name of this class is Complex.
//    Much of the implementation code is from the original complex
//    source files, some modified to be more Macintosh and ANSI-C++
//    compatible.

#if defined(__SC__)
#pragma once
#endif

#ifndef __COMPLEXNUMBERS__
#define __COMPLEXNUMBERS__

#if macintosh
      #if defined(powerc)
            #include <math.h>
      #else
            #if defined(__SC__)
                  #if __option(mc68881)
                        #include <math.h>
                  #else
                        #ifndef __SANE__
                              #include <SANE.h>
                        #endif
                  #endif
            #else
                  #include <math.h>
            #endif
```

```
      #endif
#else
      #include <math.h>
```

```
#endif

#if macintosh
	#ifndef __TYPES__
		enum {false, true};
	#endif
#else
	enum {false, true};
#endif

typedef double FP_TYPE;

class Complex {
public:

	// Constructors
	Complex();
	Complex(const FP_TYPE re, const FP_TYPE im);
	Complex(const Complex& copy);		// Complex's copy constructor

	// Indirect access functions
	FP_TYPE& Real();
	FP_TYPE& Imag();

	// These functions return only values not references.
	FP_TYPE Re() const;
	FP_TYPE Im() const;

	// Overloaded operators

	// operator+
	Complex operator+(const Complex&) const;
	Complex operator+(const FP_TYPE) const;
	friend Complex operator+(const FP_TYPE, const Complex&);

	// operator-
	Complex operator-(const Complex&) const;
	Complex operator-(const FP_TYPE) const;
	friend Complex operator-(const FP_TYPE, const Complex&);

	// operator*
	Complex operator*(const Complex&) const;
	Complex operator*(const FP_TYPE) const;
	friend Complex operator*(const FP_TYPE, const Complex&);

	// operator/
	Complex operator/(const Complex&) const;
	Complex operator/(const FP_TYPE) const;
	friend Complex operator/(const FP_TYPE, const Complex&);

	// operator&&
	long operator&&(const Complex&) const;
	long operator&&(const FP_TYPE) const;
	friend long operator&&(const FP_TYPE, const Complex&);

	// operator||
	long operator||(const Complex&) const;
	long operator||(const FP_TYPE) const;
	friend long operator||(const FP_TYPE, const Complex&);
```

```
// operator==
long operator==(const Complex&) const;
long operator==(const FP_TYPE) const;
friend long operator==(const FP_TYPE, const Complex&);
```

```
        // operator !=
        long operator!=(const Complex&) const;
        long operator!=(const FP_TYPE) const;
        friend long operator!=(const FP_TYPE, const Complex&);

        // Unary operators
        long operator!() const;

        Complex operator-() const;

        // Overloaded operators returning references

        // operator=
        Complex& operator=(const Complex&);
        Complex& operator=(const FP_TYPE);

        // operator+=
        Complex& operator+=(const Complex&);
        Complex& operator+=(const FP_TYPE);

        // operator-=
        Complex& operator-=(const Complex&);
        Complex& operator-=(const FP_TYPE);

        // operator*=
        Complex& operator*=(const Complex&);
        Complex& operator*=(const FP_TYPE);

        // operator/=
        Complex& operator/=(const Complex&);
        Complex& operator/=(const FP_TYPE);

private:
        FP_TYPE                     real;
        FP_TYPE                     imag;
};

// Prototypes of complex number functions

FP_TYPE Abs(const Complex&);
FP_TYPE Mod(const Complex&);
FP_TYPE Arg(const Complex&);
Complex Conj(const Complex&);

// Inline member functions

inline Complex::Complex(void) : real(0.0), imag(0.0) { }
inline Complex::Complex(const FP_TYPE re, const FP_TYPE im) : real(re),
imag(im) { }
inline Complex::Complex(const Complex& z) : real(z.real), imag(z.imag) { }

inline FP_TYPE& Complex::Real(void) { return real; }
inline FP_TYPE& Complex::Imag(void) { return imag; }

inline FP_TYPE Complex::Re(void) const { return real; }
inline FP_TYPE Complex::Im(void) const { return imag; }

inline long Complex::operator!(void) const
{
```

```
        return ((Re()==0) && (Im()==0)) ? true : false;
}
```

```
inline Complex Complex::operator-(void) const
{
	return Complex(-Re(), -Im());
}

inline FP_TYPE Mod(const Complex& z)
{
	return Abs(z);
}

#endif
```

About FFT

The FFT library, in conjunction with the ComplexNumbers library, provide an implementation of the Fast Fourier Transform (FFT).

The Fourier transform is a transform function that, given an arbitrary function, can break that function down into the sum of a (possibly infinite) number of sine and cosine waves. The original function can be re-synthesized by using the inverse Fourier transform.

The following is a typical definition of the Discrete Fourier Transform (DFT):

$$X(k) = \Delta\Phi\mathrm{T}[\xi(\nu)] = \sum_{\nu=0}^{N-1} \xi(\nu)\varepsilon^{-\iota\frac{2\pi}{N}\nu\kappa} \quad .$$

The inverse is similar:

$$x(n) = \frac{1}{N}\sum_{\kappa=0}^{N-1} \Xi(\kappa)\varepsilon^{\iota\frac{2\pi}{N}\kappa\nu} \quad .$$

With a little deduction, it can be shown that it takes n^2 multiplications to compute the Discrete Fourier Transform of an array with n data points. The Fast Fourier Transform is an algorithm that compute the same values with *much* fewer multiplications.

Using FFT

The Fast Fourier Transform, as opposed to the discrete version, requires that the number of data points be an integral power of 2 (e.g. 4, 8, 16, 32, 64, ...).

Using the FFT library in your source code files

To use the FFT library functions, add these three lines to your source code files:

```
#ifndef __FFT__
#include "FFT.h"
#endif
```

Make sure that the `FFT.h` file is included *after* your standard headers and *before* any definitions begin. `FFT.h` includes `ComplexNumbers.h`, so it is not necessary to include `ComplexNumbers.h` in a source code file that already includes `FFT.h`.

Using the FFT function

To compute the Fast Fourier Transform of an array `waveform`, you could use this code:

```
void DoFFT(Complex* data, long dataPoints, long log2DataPoints)
	// data is an array.
{
	FFT(dataPoints, log2DataPoints, data);
	// data now holds the Fourier transforms of the input.

	// show the results...
	.
	.
	.
}
```

The last parameter to the `FFT()` function should be the data array that will be transformed. If the data is real, all the imaginary fields of the array should be zero.

Using the Inverse FFT (IFFT)

The `IFFT()` function that performs an Inverse Fast Fourier transform and has the same declaration of the `FFT()` function. Referring to the example above, we can add the inverse transform:

```
void DoFFT(Complex* data, long dataPoints, long log2DataPoints)
	// waveform is an array.
{
	FFT(dataPoints, log2DataPoints, data);
	// data now holds the Fourier transforms of the input.

	// display the results...
	.
	.
	.

	// do the inverse transform and get the original function...
	IFFT(dataPoints, log2DataPoints, data);

	// display the results...
	.
	.
	.
}
```

FFT reference

Data types

The only data type that the FFT library defines is `ComplexArray`. It is defined to be an array of `Complex` objects. ANSI C++ defines `T[]` and `T*` to be equivalent types.

```
typedef Complex *ComplexArray;
```

FFT

```
void FFT(const long n, const long nu, ComplexArray spectrumOut);
```

`FFT()` performs the Fast Fourier Transform on the parameter `spectrumOut`. Input data is destroyed. `n` is the number of data points in `spectrumOut`, and should be an integral power of two. `nu` should satisfy this equation: $n = 2^{\nu}$. In other words, `nu` is the $\log_2$ of `n`. If the data in `spectrumOut` is real, all the imaginary components of the input should be zero.

IFFT

```
void IFFT(const long n, const long nu, ComplexArray functionOut);
```

`IFFT()` performs the Inverse Fast Fourier Transform on the parameter `functionOut`. Input data is destroyed. The `n` and `nu` parameters are analogous to the `n` and `nu` parameters in the `FFT()` function.

FFT summary

```
//
//	File: FFT.h
//	Copyright ©1994-1995 James Wilson
//	All rights reserved.
```

```
//
//    This is the header file for FFT.cp, an implementation of the Fast
//    Fourier Fransform (FFT).
```

```
#ifndef __FFT__
#define __FFT__

#if macintosh
	#if defined(powerc)
		#include <math.h>
	#else
		#if defined(__SC__)
			#if __option(mc68881)
				#include <math.h>
			#else
				#ifndef __SANE__
					#include <SANE.h>
				#endif
			#endif
		#else
			#include <math.h>
		#endif
	#endif
#else
	#include <math.h>
#endif

#ifndef __COMPLEXNUMBERS__
#include "ComplexNumbers.h"
#endif

// 2*pi is equal to one revolution in radians (equivalent to 360 degrees).
const double twoPi = 6.28318530717959;

typedef Complex *ComplexArray;

/*
	Function name: FFT

	Returns: none

	Parameters:
		n:	the number of data points in the spectrumOut array. n
		should always be a power of 2.
		nu:	nu should satisfy this equation: n = 2^nu. ^ signifies
		power not the binary XOR operator. In other words, nu is
		log2 of n.
		spectrumOut: on input, the real fields of the elements in
		this array should hold the input values of the time-waveform
		to be analyzed, with all the imaginary fields set to 0. On
		output, this array holds the complex values of the Fourier
		transform of the input. Input data is destroyed.

	Notes:		This function is an implementation of the Fast Fourier
	transform, a function that takes any sampled function and breaks
	in up into sine and cosine components. It is particularly useful
	in digital signal processing and the analysis of sound waves.
*/
extern void FFT(const long n, const long nu, ComplexArray spectrumOut);

/*
	Function name: IFFT
```

Returns: none

Parameters:

```
            n and nu:         these variables serve the same purpose as
            they do in the function FFT (see above).
            functionOut:      On input, the elements of this array
            should hold the complex Fourier transform values that are to
            be transformed back into function values.

      Notes:      This function performs an Inverse Fourier transform;
      it takes the transform values and creates a function out of it.
      The IFFT is defined seperately from the FFT for pure computational
      speed. Please be aware that the inverse values may not equal the
      original values exactly. This is because anything that uses the
      transcendental functions (sine, cosine, etc.) will never be exact,
      however the inverse values should close enough not to notice a
      significant difference.
*/
extern void IFFT(const long n, const long nu, ComplexArray functionOut);

/*
      Function name: BitReverse

      Returns: long
            Possible return values: theNum, with its binary digits reversed.

      Parameters:
            theNum:     the number that is to be reversed.
            nu:         the number of significant bits in theNum. This
            paramter is similar to the nu paramter in the FFT function
            (see above).

      Notes:      This function produces a number that is the binary
      reverse of the input, theNum. For example, calling BitReverse with
      theNum = 011001 and nu = 6 results with 100110. This function
      is used internally by the functions FFT and IFFT.
*/
extern long BitReverse(const long theNum, const long nu);

/*
      Function name: LongPower

      Returns: long
            Possible return values: base, raised to the power exp.

      Parameters:
            base: the base that is to be raised by exp.
            exp:  the exponent.

      Notes:      Power functions are not well defined in C/C++, so for
      simplicity and compatibility, I define my own. Its operation
      is staightforward, and probably not of that much interest to
      applications.
*/
extern long LongPower(const long base, const long exp);

#endif
```

References

Strawn, John, editor. *Digital Audio Signal Processing: An anthology.* Los Altos, Calif. : W.

Kaufmann, 1985.

Walker, James S. *Fast Fourier Transforms*. Boca Raton : CRC Press, 1991.

Brigham, E. Oran. *The fast Fourier transform and its applications*. Englewood Cliffs, N.J. : Prentice Hall, 1988.

Ludeman, Lonnie C. *Fundamentals of digital signal processing*. New York : Harper & Row, 1986.