C++Wavelets: A User's Guide

S. E. FERRANDO, L. A. KOLASA, AND N. KOVAČEVIĆ

ABSTRACT. We expalin the technical features of *C++Wavelets* and how to use the software. Presantly this users manual is still being written. Anyone compotent in wavelet analysis and C++ should be able to use this software presently. The header files and the demos supplied with this package should be enough to get one started.

1. INTRODUCTION

C++Wavelets is a collection of C/C++ based functions for performing some of the basic functions of wavelet analysis: wavelet analysis and synthesis; wavelet packet analysis and synthesis.

Our objective is to provide the serious programmer with ready to use functions that may be employed in applications or programs which call for wavelet analysis. While there are many good packages available for performing wavelet based analysis (e.g., Matlab), some applications must be written in a high level programming language. Often one has to perform a great many calculations, and the overhead associated with software packages can be prohibitively expensive. What is needed then is a library of useful functions written in a high level programming language, ready available "at ones finger tips."

We have chosen C++ as our high level language. C programmers, howerer, should not despair though; most of wavelet based analysis is algorithmic, and we use only as much C++ so as to make code writing as transparent as possible. A good introduction to C++ can be found in [2]. Chapters 2–6 contain all that is necessary to know when using *C++Wavelets*. In particular we like the memory management features of C++. We shall explain our data structures below; it suffices to say right now that in *C++Wavelets* the data structures of wavelet analysis are invoked and destroyed almost as easily as native data structures (i.e., integers, floats).

We take advantage extensively of the feature in C++ of overloaded operators and function in *C++Wavelets*. Again, this is done so as to make writing programs as natural and transparent as possible.

Otherwise we are not "object oriented". For us wavelet analysis is a process, and the algorithms are paramount. For example, we have often been criticized for not using the private data member capabilities of C++. We avoid this because we are interested in speed and ease of programming; to us private data implies costly member functions to get your hands on it. We trust the integrity of our library and the intelligence of our users. Should data be "inadvertently" modified this would be a mistake, but we prefer to allow you that possibility.

On the other hand we have taken great care in writing the algorithms using, essentially, the features of C which allow for faster running code such as pointer

Research partially supported by NSERC.

arithmetic. Therefore some of the code itself may be difficult to read; we ask only that you trust us when we say that it has been tested. We have striven for speed in writing this code, not readability.

C++Wavelets allows you to perform the one-dimensional Discrete Wavelet Transform (DWT) using orthogonal or bi-orthogonal wavelelets, in a periodic or an aperiodic setting. Discrete Wavelet Packet Analysis, along with the best wavelet basis algorithm of Coifman and Wickerhauser [3] is also available. This too comes in a periodic and an aperiodic version.

We assume some familarity on the reader's part with wavelet based analysis. But as there may be some ambiguity as to what the DWT and DWPA actually are, we outline here what they are in *C++Wavelets*.

The soul of discrete wavelet analysis is convolution-decimationand its adjoint. [3] is an excellent reference; we merely highlight those details germane to *C++Wavelets*. Let $X = \{x_i\}$ be a sequence of real numbers which represents your data or a signal to be processed; let $G = \{g_i\}$ be a fixed, given "filter". Then the convolution-decimation operator defined by G acts on X in the following way,

$$GX_i = \sum_j g_j x_{2i-j},$$

and its adjoint is given by

$$G^* X_j = \sum g_{2i-j} x_i.$$

2. The data classes

In this section we describe the classes that are used in *C++Wavelets*. The classes that appear in *C++Wavelets* are

1. Interval
2. ArrayTreePer, ArrayTreeAper
3. BinTree
4. HedgePer HedgeAper
5. QMF, PQMF, GPQMF

We discuss the Interval class in the most detail. It is the most common class and the simplest; it is all that is needed to perform the DWT

2.1. **Interval.**

The `Interval` class is an array of reals with a few extras. Typically an `Interval` is used as the input and the output when performing wavelet analysis. Input data, a sequence of floating point numbers $\{x_i\}$, are stored in an array. The main difference between a C array and an invterval is that the indexing of an interval does not have to start with zero. As noted earlier indexing is important in the wavelet theory.

The data members of the interval class are

- `Interval.origin`
- `Interval.beg`
- `Interval.end`
- `Interval.length`

Suppose we have a sequence of reals:

$$x_{-2} = 2.1, x_{-1} = -0.3, x_0 = 0.0, x_1 = -1.1, x_2 = 0.2, x_3 = 9.4,$$

and we store them in an `Interval` called *Input*. Then `Input.beg` is the integer -2, `Input.end` is the integer 3 and `Input.length` = `Input.end` − `Input.beg` + 1, which is the number of elements in the data sequence. Aside from the fact that it is crucial to keep track of `Input.beg` and `Input.end` when performing convolution-decimation, there are other instances where these data members are useful. The following example shows how to write a `for` loop involving an `Interval`

```
for (i = Input.beg; i <= Input.end; i++)
```

This relieves us from having to pass as a parameter to functions the beginning and ending indices of a sequence. The `length` data member is also useful in this regard. As we get deeper and deeper into the stack it is more and more to our advantage to avoid having the clutter of long parameter lists; the baggage of the `Interval` class carries many important parameters. Admittedly the `length` member is redundant, but we use it often enough that it deserves its place as a data member.

The `origin` data member is important and must be handled carefully; it is used to point to the actual data. If we only wish to access or modify the data without concern for speed or its exact memory location then the easiest thing to do is to use the overloaded []–operator. For example, Input[−2] = 2.1, Input[−1] = −0.3, . . . , Input[3] = 9.4. From this example we see that, by design, the indexing of the original sequence corresponds to the indexing used by the []–operator—exactly how a mathematician thinks. Thus if we desire to change the value of x_1 from −1.1 to 2.9 we write

```
Input[1] = 2.9;
```

Of course the data represented by x_1 is the fourth data member in our data array, but mathematically we think of it as having index equal to 1. Our programming corresponds to our thinking; this is by design.

But a price must be paid for this. The way this feature has been implemented is by pointer offset: `Input.origin` is not a pointer to the beginning of the data array (i.e., x_{-2}); in this case `Input.origin` points to x_0. In general, when an `Interval` is constructed, `origin` is pointing to the beginning of the data array. But then it is modified:

```
origin -= beg;
```

This then is how the []–operator works: `Input[j] = *(Input.origin + j)`. Should a pointer to the "physical" beginning of the data be needed, the proper reference is

```
Input.origin + Input.beg
```

This is very important if you work with sequences which, mathematically speaking, are not indexed starting from zero à la the C/C++ convention. In practice, however, it is often convenient or desirable to have `Input.beg` = 0; in this case, of course, `Input.origin` is a pointer to the actual beginning of the array, and no confusion results.

One notable instance when `Input.beg` = 0 is in the periodic version of the DWT, where we may assume this without loss of generality. For this common setting, when the number of data points must be a power of 2, we have special constructors,

`Interval(p, *data)` or `Interval(p)`,

which construct an interval indexed from 0 to $2^p - 1$, filled from `data`, or if `data` is not given, filled with zeros. The more generic constructor is,

`Interval(a, b, *data)` or `Interval(a, b)`.

This constructs an interval indexed from a to b. If no data is given, the `Interval` is filled with `length` $= b - a + 1$ zeros. One must be careful that `*data` points to an array of at least length elements, for otherwise that would be a mistake.

Finally there is a default constructor which, for the most part, sets everything to zero. The default constructor is useful when one wishes to create an array of `Intervals` as so,

`Interval Input[10];` or `Interval* Input = new [10]`.

In each instance we have a pointer, *Input*, to an array of 10 default intervals. When the time comes one may use the `Set` member function to fill out a desired `Interval`. Ultimately all of the `Interval` constructors call the `Set` member function, and when a default `Interval` already exists the `Set` function is the correct one to use in order to fill it out; for example:

`Input[6].Set(0, 1023);` or `Input[4].Set(-1, 1022, *data);`

The first line above fills out the seventh element of the arrray, *Input*, with an array of 1024 zeros indexed from 0 to 1023. The second line fills out the fifth element of *Input* with the 1024 elements of the array pointed to by data and indexes them from -1 to 1022.

When using `new` as in the above example one is obliged to use `delete`, but as with all *C++Wavelets* data structures memory management is transparant and self contained. One need only understand how to employ the basic constructors to bring a single data structure into existence; the rest of memory management is done behind the scenes. When you become familiar with *C++Wavelets* the header files should be enough to refresh your memory.

References

[1] F. Carrano, P. Helman, R. Verofff, *Data Abstraction and Problem Solving in C++*. Addison Wesley, 1998.
[2] H. Schildt, *Teach Yourself C++*, McGraw Hill, (1998).
[3] M. Wickerhauser, *Adaptive Wavelet Analysis: From Theory to Software*. A. K. Peters, (1994).

Department of Mathematics, Physics and Computer Science, Ryerson Polytechnic University, Toronto, Ontario M5B 2K3, Canada.

E-mail address: ferrando@acs.ryerson.ca

E-mail address: lkolasa@acs.ryerson.ca

E-mail address: natasak@home.com